




Fast Gaussian Elimination for Low Treewidth Matrices

Martin Fürer  

Department of Computer Science and Engineering,
Pennsylvania State University, University Park, PA, USA

Carlos Hoppen  

Instituto de Matemática e Estatística, Universidade Federal do Rio Grande do Sul, Brazil

Vilmar Trevisan  

Instituto de Matemática e Estatística, Universidade Federal do Rio Grande do Sul, Brazil

Abstract

Let $A = (a_{ij})$ be an $m \times n$ matrix whose elements lie in an arbitrary field \mathbb{F} , and let G be the bipartite graph with vertex set $\{v_1, \dots, v_m\} \cup \{w_1, \dots, w_n\}$ such that vertices v_i and w_j are adjacent if and only if $a_{ij} \neq 0$. We introduce an algorithm that finds an $m \times n$ matrix U in row echelon form and a permutation matrix Q of order n , such that AQ is row equivalent to U . If a tree decomposition \mathcal{T} of G of width k and size $O(k(m+n))$ is part of the input, then Q and the columns of U that contain a pivot can be computed in time $O(k^2(m+n))$. Among other things, this allows us to compute the rank and the determinant of A in time $O(k^2(m+n))$. It also allows us to decide in time $O(k^2(m+n))$ whether the linear system $Ax = b$ has a solution and to compute a solution of the linear system in case it exists.

2012 ACM Subject Classification Theory of computation \rightarrow Design and analysis of algorithms

Keywords and phrases Gaussian elimination, FPT algorithms, treewidth

Digital Object Identifier 10.4230/LIPIcs.ESA.2025.116

Funding *Carlos Hoppen:* C. Hoppen acknowledges the support of CNPq 315132/2021-3.

Vilmar Trevisan: V. Trevisan acknowledges partial support of CNPq grants 409746/2016-9 and 303334/2016-9, CAPES under project MATHAMSUD 18-MATH-01 and FAPERGS under Project PqG 17/2551-0001. CNPq is Conselho Nacional de Desenvolvimento Científico e Tecnológico, CAPES is Coordenação de Aperfeiçoamento de Pessoal de Nível Superior and FAPERGS is Fundação de Amparo à Pesquisa do Estado do Rio Grande do Sul.

1 Introduction

Reducing a matrix to a desired form or decomposing a matrix into a product of matrices with a given structure are basic operations in linear algebra. Problems involving triangular matrices and matrices in row echelon form are often easy to solve, and a large number of tasks may be accomplished by first reducing a general matrix into a matrix of this form. To be precise, a matrix $A = (a_{ij})$ is in *row echelon form* if all rows consisting entirely of zeros are at the bottom, and the following holds for the remaining rows. If i is a nonzero row, its *pivot* is the element $a_{ij} \neq 0$ with least j . If rows $i_1 < i_2$ have pivots in columns j_1 and j_2 , respectively, then $j_1 < j_2$.

A basic result in linear algebra states that every matrix can be transformed into a matrix in row echelon form by a sequence of three types of row operations, known as *elementary row operations*. In this paper, we shall refer to two of these types: a type I elementary row operation on a matrix involves interchanging two rows. A type II elementary row operation involves adding a multiple of a row to another row. We say that A and B are *row equivalent* if A can be transformed into B by a sequence of elementary row operations. The process of



© Martin Fürer, Carlos Hoppen, and Vilmar Trevisan;
licensed under Creative Commons License CC-BY 4.0

33rd Annual European Symposium on Algorithms (ESA 2025).

Editors: Anne Benoit, Haim Kaplan, Sebastian Wild, and Grzegorz Herman; Article No. 116; pp. 116:1–116:15

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

reducing a matrix to row echelon form by elementary row operations is known as Gaussian elimination. The standard school algorithm reduces an arbitrary $m \times n$ input matrix A with entries in a field \mathbb{F} to row echelon form in $O(mn \min\{m, n\})$ field operations. However, it can be faster if more is known about the structure of A . For instance, if A is sparse, its structure can be exploited if we find a *pivoting scheme* (also known as *elimination ordering*) that minimizes the *fill-in*, defined as the set of matrix positions that were initially 0, but became nonzero at some point during the computation. It is worth noting that sparse matrices often occur for some structural reason, such as small treewidth. Gaussian elimination cannot be used efficiently on random sparse matrices with any elimination order, as we would expect such matrices to quickly get dense as the algorithm progresses. We should mention that other strategies can be used to exploit the sparsity of the input matrix, see for instance the work of Peng and Vempala [12].

Much of the early focus has been on the question of characterizing the graphs corresponding to matrices with a perfect elimination order, meaning that there is a pivot strategy for Gaussian elimination without any fill-in. Parter [11] has assigned a graph to a matrix A with an edge $\{i, j\}$ if a_{ij} or a_{ji} are non-zero. He has then shown that there is a perfect elimination order if the graph is a tree. As became customary afterwards, he chooses all pivots in the diagonal and makes the strong assumption that all diagonal elements of the matrix are non-zero and remain so during the computation, usually referred to as not having any accidental cancellation.

Obviously, this assumption simplifies Gaussian elimination enormously. It is justified because it is true for symmetric positive definite matrices, which are the most important special class of matrices for systems of linear equations. However, one would like to see more widely applicable methods.

Rose [16] shows that Parter's result can be extended to all triangulated graphs (also known as chordal graphs). Rose, Tarjan, and Lueker [18] show that a graph has a perfect elimination order if and only if the graph is triangulated. In this case, they find a perfect elimination order in $O(|V| + |E|)$ time. Rose and Tarjan [17] efficiently find a perfect elimination order in a directed graph if one exists, and show that finding a minimal elimination ordering (minimizing the fill-in) is NP-hard. A directed graph describes the non-zero off-diagonal entries in an $n \times n$ matrix, where Gaussian elimination is done by choosing pivots in the diagonal under the assumption that there will never be a 0 in the diagonal.

The perfect elimination question is more complicated for the case of asymmetric $m \times n$ matrices A , where any nonzero pivot is allowed. Any $m \times n$ matrix $A = (a_{ij})$ may be naturally associated with a bipartite graph G with vertex set $\{v_1, \dots, v_m\} \cup \{w_1, \dots, w_n\}$ such that vertices v_i and w_j are adjacent if and only if $a_{ij} \neq 0$. We say that G is the *underlying bipartite graph* of A . This allows us to employ structural decompositions of graph theory to deal with the nonzero entries of A in an efficient way. Different from the symmetric case with pivots restricted to the diagonal, there is no nice characterization known for the bipartite graphs with perfect elimination order when arbitrary pivots are allowed. Nevertheless, a very large class is given by the *chordal bipartite* graphs defined by Golumbic and Goss [7]. These graphs are not chordal, as obviously only forests are bipartite and chordal. Still, chordal bipartite graphs are characterized similarly to chordal graphs. For chordal graphs, the defining property is that every cycle of length at least 4 has a chord, while chordal bipartite graphs are defined by the requirement that every cycle of length at least 6 has a chord. Golumbic and Goss [7] have shown that chordal bipartite graphs have a perfect elimination order.

They are not the only bipartite graphs with a perfect elimination order, but the only ones where one can always pick any simplicial edge (definition follows) as a pivot [7]. An edge $\{u, v\}$ is simplicial if the union of the neighborhoods of u and v induces a complete bipartite graph. It is easy to see that a bipartite graph G has a perfect elimination order if there exists an ordering of a matching $\{(u_1, v_1), (u_2, v_2), \dots, (u_p, v_p)\}$ in G such that (u_i, v_i) is simplicial in the subgraph $G[V \setminus \{u_1, v_1, u_2, v_2, \dots, u_{i-1}, v_{i-1}\}]$ for $i \in \{1, \dots, p\}$.

Radhakrishnan, Hunt, and Stearns [13] do Gaussian elimination for symmetric matrices with a given tree decomposition of width k in time $O(k^2n)$. As has been common, they implicitly assume that the diagonal is non-zero and there is no accidental cancellation. Naturally, given today's familiarity with algorithms based on tree decompositions, it would now be an easy exercise to design such an algorithm when pivots can always be selected in the diagonal.

Only in recent years has there finally been some progress towards the difficult problem of efficient Gaussian elimination for matrices of small treewidth without the simplifying assumption that pivots can always be chosen in the diagonal. Fomin et al. [5] have shown a Gaussian elimination algorithm running in time $O(p^2n)$ where p is the lesser known tree-partition width parameter or the pathwidth. The tree-partition width of a graph can be arbitrarily large already for graphs of constant treewidth, unless the maximum degree is bounded [19]. However, Fomin et al. [5] still obtained an $O(k^3n)$ algorithm for treewidth k . We should also refer to the work of Dong, Lee, and Ye [4], whose main result implies an $O(k^2n)$ algorithm for the solution of a linear system $Ax = b$ such that the matrix $A \in \mathbb{R}^{m \times n}$ is full-rank (actually, their result is about the solution of a linear program).

Naturally, the longstanding main goal is an $O(k^2n)$ Gaussian elimination algorithm for treewidth k without restrictive assumptions. This would provide a smooth transition to the school algorithm when k approaches n . It would also achieve the same running time bound for the general case as for the easy case of non-zero diagonal and no accidental cancellation. It actually seems that the $O(k^2n)$ goal has not been explicitly formulated until recently [5], maybe because it had not seemed achievable. However, when this goal has been obtained by simple algorithms under very strong assumptions, the question whether these assumptions (that convenient entries are nonzero and thus can be used as pivots) are necessary must have been obvious.

We make substantial progress in this direction by devising the algorithm **Fast Gaussian Elimination**. Given an $m \times n$ matrix A over a field \mathbb{F} with underlying bipartite graph G , and given a tree decomposition \mathcal{T} of G of width k and size $O(k(m+n))$, it produces a matrix U in row echelon form that is row equivalent to A . The entries of the columns of U that contain a pivot (and therefore generate a space of dimension $\text{rank}(A)$) are fully computed in time $O(k^2n)$. The entries of the remaining columns of U are described implicitly as linear combinations of at most k other columns, and may be computed explicitly in time $O(km(n - \text{rank}(A)))$. In particular, this algorithm allows us to compute the rank and the determinant of A in time $O(k^2(m+n))$. It also allows us to decide in time $O(k^2(m+n))$ whether the linear system $Ax = b$ has a solution, and to compute a solution of the linear system in case it exists. Our main result is stated more formally below.

► **Theorem 1.** *Let A be an $m \times n$ matrix over a field \mathbb{F} , let G be its underlying bipartite graph, and let \mathcal{T} be a tree decomposition of G of width k and size $O(k(m+n))$. Algorithm **Fast Gaussian Elimination** produces an $m \times n$ matrix U in row echelon form, and a permutation matrix Q of order n with the property that AQ is row equivalent to U . The permutation matrix Q , and the columns of U that contain a pivot may be computed with $O(k^2(m+n))$ field operations. The full matrix U may be computed with $O(km(n - \text{rank}(A)))$ additional field operations.*

We actually also compute a permutation matrix P of order m such that U is the row echelon form that would be obtained from PAQ by the most standard algorithm (adding multiples of earlier rows to later rows).

Our algorithm is not based on the $O(k^3n)$ algorithm of Fomin et al. [5], as it uses standard tree decompositions in a natural way, and does not rely on the somewhat obscure notion of tree-partition width. Even though our algorithm removes the decades old severe restriction to pivoting in the main diagonal, the main idea of our Gaussian elimination algorithm is quite simple. The tree decomposition provides an elimination order for rows (equations) and columns (variables). This order sequentially hands out permissions to rows and columns to be eliminated. However, when a row i receives a green light to be processed, it is only processed immediately if it contains a variable that has already received a green light. Otherwise, this row is put into a buffer to be delayed until one of its variables x_j also has a green light. Then a_{ij} is chosen as a pivot. Likewise, when a variable receives a green light then, if needed, its column is put into a buffer until some row containing this variable has a green light.

As was the case in Fomin et al. [5] we use a *nice tree decomposition* of the underlying bipartite graph G as defined by Kloks [9]. However, to deal with 0's at intended pivot locations, we create temporary buffers for rows and columns that allow us to delay the selection of pivots as discussed above. As is typical for algorithms based on tree decompositions, row operations are actually performed on small submatrices of the input matrices. For simplicity, the algorithm does not manipulate the large given matrix. Instead, these small submatrices, called boxes, are maintained separately. The small submatrices in this paper include additional information to keep track of the rows and columns that lie in the temporary buffer, which allows us to make headway towards the output matrix in row echelon form even when accidental cancellations occur. It is crucial to perform so-called bookkeeping operations that ensure that the temporary buffers remain small during the entire application of the buffer.

The paper is organized as follows. The algorithm is described in Section 2, where we also discuss its correctness. The running time is analyzed in Section 3. We should mention that, to achieve this running time, we need to keep track of the field operations in a global way, rather than adding the contributions of the worst-case scenarios at each step. Together, these two sections establish Theorem 1. We provide several consequences of our result in Section 4.

2 The Algorithm

We now describe the algorithm **Fast Gaussian Elimination**. We start with the definition of a tree decomposition, which we now state. Let $G = (V, E)$ be a ν -vertex graph, with the standard assumption that $V = [\nu] = \{1, \dots, \nu\}$. A *tree decomposition* of a graph G is a tree \mathcal{T} with nodes $\{1, \dots, r\}$, where each node t is associated with a *bag* $B_t \subseteq V$, satisfying the following properties: (1) $\bigcup_{t=1}^r B_t = V$; (2) For every edge $\{v, w\} \in E$, there exists B_t containing v and w ; (3) For any $v \in V$, the subgraph of \mathcal{T} induced by the nodes whose bags contain v is connected. The *width* of the tree decomposition \mathcal{T} is defined as $\max_t (|B_t| - 1)$ and the *treewidth* $\text{tw}(G)$ of graph G is the smallest k such that G has a tree decomposition of width k . Tree decompositions were popularized by the seminal work of Robertson and Seymour [14, 15], but other definitions that are similar or even equivalent have appeared in earlier work, see [1, 8]. As mentioned in the introduction, we use the concept of *nice tree decomposition* introduced by Kloks [9], which is a rooted tree decomposition \mathcal{T} of a graph G such that all nodes are of one of the following types:

- (a) **(Leaf)** The node t is a leaf of \mathcal{T} ;
- (b) **(Introduce)** The node t introduces vertex v , that is, it has a single child s , $v \notin B_s$ and $B_t = B_s \cup \{v\}$.
- (c) **(Forget)** The node t forgets vertex v , that is, t has a single child s , $v \notin B_t$ and $B_s = B_t \cup \{v\}$;
- (d) **(Join)** The node t is a join, that is, it has two children s and s' , and $B_t = B_s = B_{s'}$.

We further assume that the bag associated with the root is empty. This is easy to accomplish, as any tree decomposition \mathcal{T} that satisfies (a)-(d), but does not satisfy this additional requirement, may be turned into a nice tree decomposition with empty root bag by appending a path of length at most $k + 1$ to the root of \mathcal{T} , where all nodes have type forget. Having a root with an empty bag ensures that each vertex of $G = (V, E)$ is associated with exactly one forget node that forgets it. Moreover, it ensures that, for any $v \in V$, the node of \mathcal{T} that is closest to the root among all t such that $v \in B_t$ is the child of the node that forgets v . Despite the additional structure, a nice tree decomposition may be efficiently derived from an arbitrary tree decomposition. More precisely, Kloks [9, Lemma 13.1.2] has shown that, if G is a graph of order ν and we are given an arbitrary tree decomposition of G with width k and r nodes, it is possible to turn it into a nice tree decomposition of G with at most 4ν nodes and width at most k in time $O(k(\nu + r))$. Regarding the computation of a tree decomposition, Bodlaender's theorem [2] provides an optimal tree decomposition in linear time when the treewidth k is a constant. Bodlaender et al. [3] and Korhonen [10] compute faster constant factor approximations in linear time. Thus, for bounded k , there is no need to assume a tree decomposition is given, since both the tree decomposition and the Gaussian elimination are then computed in time $O(n)$. However, the constants hidden in the O -notation grow quickly with k .

We are now ready to describe the algorithm. Let A be an $m \times n$ matrix with entries in a field \mathbb{F} and let $G = (V, E)$ be the underlying bipartite graph with vertex set $V = \{v_1, \dots, v_m\} \cup \{w_1, \dots, w_n\}$ associated with it. We wish to find an $m \times m$ permutation matrix P , an $n \times n$ permutation matrix Q and an $m \times n$ matrix U in row echelon form such that PAQ is row equivalent to U in a direct way. We shall operate on a nice tree decomposition \mathcal{T} of G with node set $[r]$ and width k rather than on the matrix A itself. The algorithm **Fast Gaussian Elimination** works bottom-up on the rooted tree \mathcal{T} , that is, it only processes a node t after its children have been processed. Each node t except the root produces a data structure known as a *box* and transmits it to its parent. A box is a matrix A_t with $O(k)$ rows and columns. It may be recorded as a triple of matrices (M_t, N_t, J_t) whose rows and columns are labeled by distinct elements of $\{v_1, \dots, v_m\}$ and $\{w_1, \dots, w_n\}$, respectively, associating rows and columns of the boxes with rows and columns of the original matrix. We may view it as

$$A_t = \begin{array}{|c|c|} \hline \mathbf{0}_{m_t \times n_t} & M_t \\ \hline N_t & J_t \\ \hline \end{array}, \quad (1)$$

where J_t is an $r_t \times c_t$ matrix whose rows and columns are labeled by the elements in B_t , the bag associated with node t . Here r_t is the number of row vertices (i.e., vertices v_i) in B_t and c_t is the number of column vertices (i.e., vertices w_j) in B_t . The matrices M_t and N_t have dimensions $m_t \times c_t$ and $r_t \times n_t$, respectively, where $m_t < 2c_t$ and $n_t < 2r_t$. Observe

that some of these matrices can be degenerated, in the sense that $m_t = 0$, $n_t = 0$, $c_t = 0$ or $r_t = 0$. In a case where $m_t = 0$, $n_t = 2$, $c_t = 2$ and $r_t = 3$, for instance, the box A_t is 3×4 matrix, M_t is a matrix with no rows and two empty columns, and both N_t and J_t are 3×2 matrices. We should note that a similar data structure, also called a box, has been exploited in the context of computing a diagonal matrix that is congruent to a given symmetric matrix using a tree decomposition of its underlying graph [6].

It will also be useful to assign a *canonical ordering* to the rows and columns of the input matrix. We assume that they are ranked according to the order in which their Forget nodes appear in some topological ordering (e.g., obtained by a post-order traversal of \mathcal{T}), that is, v_1 is the first vertex that is forgotten among all vertices that represent rows, v_2 is the second, and so on. The same applies to w_1, \dots, w_n .

We start with an intuition about the meaning of the boxes and about how the algorithm works. Recall that the nodes of the nice tree decomposition \mathcal{T} are ordered bottom-up as $1 \dots, r$, where r is the root and $B_r = \emptyset$. At each node $t \in V(\mathcal{T})$, the algorithm initializes box A_t (if t is a leaf) or produces a new box A_t based on its bag B_t and on the boxes transmitted by its children. While producing the box A_t , the algorithm may access entries a_{ij} of the input matrix such that both i and j are associated with vertices in B_t . The box A_t is then transmitted to its parent, except if t is the root of \mathcal{T} . We call this *processing* node t .

It is also useful to keep in mind that the algorithm performs two types of elementary row operations, which we call *authentic operations* and *bookkeeping operations*. The former are operations that can be performed in the full matrix A in order to reach row echelon form. The latter are operations that are only performed on the boxes as a way to limit their sizes, but which do not have full matrix counterparts. The effect of processing nodes may be viewed as a sequence of $m \times n$ matrices

$$\tilde{A}_0 = A, \tilde{A}_1, \dots, \tilde{A}_r = U, \quad (2)$$

where U is in row echelon form and \tilde{A}_t is obtained from \tilde{A}_{t-1} by performing the authentic elementary row operations that have been performed while node t is processed. We should mention that this sequence of row equivalent matrices is a simplification of what actually happens, but that it could be achieved if an oracle gave us the proper ordering of rows and columns of A . In reality, the proper orderings are also computed by the algorithm while the nodes are processed, leading to the permutation matrices P and Q mentioned in the statement of Theorem 1.

To achieve the complexity stated in Theorem 1, row operations cannot be actually performed on $m \times n$ matrices, but are instead performed only in the (smaller) boxes. It is useful to think that any row or column of the input matrix A , say row i , is initially *untouched* and changes to a *regular* row when the vertex v_i associated with it appears in the bag of a node of the branch¹ of the tree decomposition rooted at t . When v_i is removed from the bag (at the node that forgets it in the nice tree decomposition), either row i is assigned a pivot in some column j , in which case row i is classified as *processed* (column j is also classified as processed), or row i becomes a *buffer* row, meaning that it has been given the green light for a pivot, but that the decision of assigning a pivot has been deferred to a later step because no column has received a green light. In the process of keeping the size of the box bounded, the algorithm may decide that it is impossible or unnecessary to assign a pivot to a buffer row or to a buffer column, in which case the status of the row or column also

¹ We use “branch” to refer to the subtree induced by all descendants of a node.

changes to *processed*. We observe that the classification of each row and column is defined independently for disjoint branches of the tree decomposition, and an important feature of the algorithm is that these independent classifications can be made compatible when two branches are merged (which happens at join nodes).

This finally gets us to the meaning of a box $A_t = (M_t, N_t, J_t)$ at node t . The rows and columns of the matrix J_t are precisely the regular rows and columns of the branch \mathcal{T}_t of \mathcal{T} rooted at node t . In other words, these are the rows and columns corresponding to the vertices in the bag B_t . Moreover, each entry ij of J_t records the *net change* to the original entry a_{ij} due to row operations performed while processing nodes in \mathcal{T}_t . The matrix M_t records the entries ij of the matrix \tilde{A}_t defined in (2) in the case where i is a buffer row (with respect to \mathcal{T}_t) and j is a regular column. Analogously, N_t records the entries ij of \tilde{A}_t in the case where i is a regular row and j is a buffer column. This reveals another important feature of the algorithm: The entries of the full $m \times n$ matrix associated with regular rows and regular columns can easily be modified in parallel by computations in disjoint branches, as the algorithm does not capture their exact values while processing a single branch. The boxes just store the net changes to the values due to operations in this branch. However, this does not happen for buffer rows and buffer columns, for which exact values are recorded in the box. The top left corner of A_t in (1) shows that, if i is a buffer row and j is a buffer column, the entry ij in \tilde{A}_t is equal to zero. This is consistent with the idea that the algorithm selects a new pivot whenever there are a buffer row i and a buffer column j for which the element ij is nonzero, as both the row and the column had already been given the green light as possible rows and columns for a pivot. So, if row i and column j are in the buffer at the end of step t , then the entry ij must be 0.

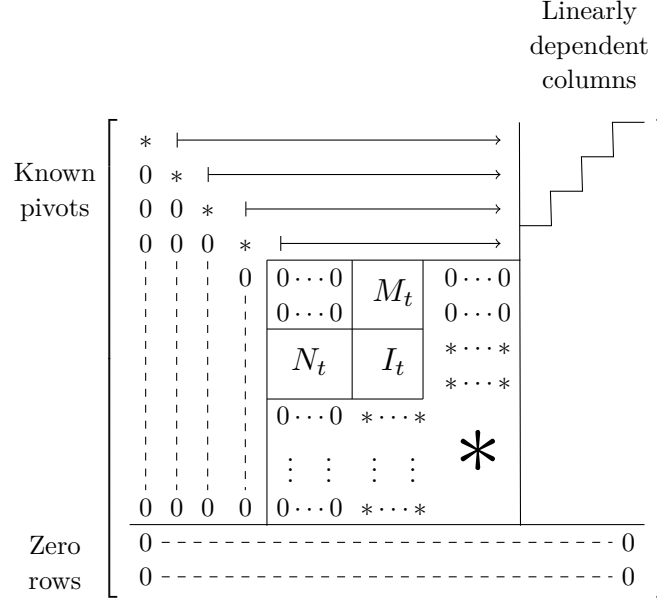
While producing the boxes, the algorithm may select pivots for the output matrix U , it may identify zero rows that cannot contain a pivot and it may identify columns that are linear combinations² of other columns, and therefore cannot contain pivots together with the other columns. The status of these rows and columns changes to *processed*, which means that information about them is recorded by global variables, while they are removed from the box.

After this description, we come back to a matrix \tilde{A}_t in (2), depicted in Figure 1³. The top left entries lie in processed rows and processed columns that contain pivots. These entries are not modified later in the algorithm. The bottom rows are (processed) zero rows that have been produced while limiting the size of the set of buffer rows (it is indeed impossible to find a pivot in such a row), while the rightmost columns are (processed) columns that have been identified as linear combinations of other columns (except for possible entries in rows whose pivot had already been computed when this column was processed). As a consequence, it is unnecessary to assign a pivot to them. Moreover, while the actual values in the rightmost columns may be modified in later steps of the algorithm, we need not keep track of them, as each such column may be computed as a linear combination of (a known set of) $O(k)$ columns with smaller index at the end of the algorithm. The central submatrix represents entries in the buffer, and in regular or untouched rows and columns. Some of them lie in the box where computations are actually performed in a given step.

Before giving a formal description of every step of the algorithm, we briefly describe what is done at a node of each type. A node of type leaf merely initializes a box with regular rows or columns, while a node of type introduce simply adds a new regular row or column to the

² Actually, they are not necessarily linear combinations of the other columns with respect to the entire matrix, but they are linear combinations with respect to a submatrix as depicted in Figure 1.

³ The authors thank Elizandro Max Borba for the figure.



■ **Figure 1** A description of an intermediate matrix in the Gaussian elimination process.

box transmitted by its child. Nodes of type join combine the information that is passed on by two different branches of the decomposition, which have been processed independently of each other. Let A_j and A_ℓ denote the boxes produced by the children of t . A new matrix M_t is produced by stacking the matrix M_j on top of M_ℓ , and a new matrix N_t is produced by juxtaposing N_j to the left of N_ℓ . Such matrices M_t and N_t may become too large, so that row operations are performed to remove some rows and/or columns from the buffer. Producing J_t also requires merging J_j and J_ℓ , as the regular rows and columns are the same. Being able to easily merge the matrices transmitted by the children is the reason why the rows and columns are considered with the canonical ordering mentioned above.

Nodes of type forget concentrate most of the action. Recall that each row i of A is associated with a vertex v_i of G , which in turn is associated with the unique Forget node of \mathcal{T} that forgets v_i . Analogously, each column j of A is associated with a vertex w_j of G and the unique Forget node of \mathcal{T} that forgets w_j . Assume that the algorithm is processing a Forget node t , which means that it received a box A_s of its child s . Further assume that node t forgets a vertex v_i corresponding to row i . This means that the algorithm singles out this row as a candidate for having a pivot. In the box A_s , it checks whether there is a column j that has already been singled out for a possible pivot, but which has no pivot so far. These are precisely the columns of N_s . If such a column exists, it checks whether the entry ij in N_s is nonzero. If this is also satisfied, the algorithm performs operations to create a pivot in row i and column j , and they are both removed from the box. If one of the conditions fails, row i is removed from J_s and added to M_t , which contains rows singled out for a pivot, but which have no pivot so far. Furthermore, if the condition on the maximum number of rows of M_t fails after the addition of this new row, the algorithm acts to remove some rows from the buffer.

Analogously, if node t forgets vertex w_j corresponding to column j , it first checks whether there are rows that have been singled out for pivots, but have not been used, namely the rows of M_s . It then looks for one such row i for which the entry ij is nonzero. If such a row

exists, the algorithm performs operations to create a pivot in row i and column j , and they are both removed from the box. If one of the conditions fails, column j is removed from J_s and added to N_t , which contains columns singled out for a pivot, but which have no pivot so far. If the condition on the number of columns of N_t fails after the addition of this new column, the algorithm acts to remove some columns from the buffer.

A high level description of the algorithm may be found in Figure 2. Note that, in this description, we have five global variables: the number ρ of pivots already computed, two vectors \mathbf{u} and \mathbf{p} of length m and two vectors \mathbf{w} and \mathbf{q} of length n . Initially, these are zero vectors. At the end of the algorithm, vector \mathbf{u} allows us reconstruct the rows of the output matrix U , while vectors \mathbf{p} and \mathbf{q} encode the permutations that generate P and Q , respectively. As we discussed, the algorithm may also find columns of A that are linear combinations of other columns and remove them from boxes. When this happens, the linear combination that produces it is recorded in the vector \mathbf{w} , so that the values of entries of U in these columns can also be recovered at the end of the algorithm.

Fast Gaussian Elimination(A)

input: a nice tree decomposition \mathcal{T} of width k of the underlying graph G associated with the $m \times n$ matrix A , the nonzero entries of A

output: matrices U , P and Q

Initialize zero vectors \mathbf{p} and \mathbf{u} of length m

Initialize zero vectors \mathbf{q} and \mathbf{w} of length n

Initialize the counter $\rho = 0$ for the number of pivots

Order the nodes of \mathcal{T} as $1, 2, \dots, r$ in post-order

for t **from** 1 **to** $|\mathcal{T}|$ **do**

if **is-Leaf**(t) **then** $(M_t, N_t, J_t) = \text{LeafBox}(B_t)$

if **is-Introduce**(t) **then** $(M_t, N_t, J_t) = \text{IntroBox}(B_t, v, A_s)$

if **is-Join**(t) **then** $(M_t, N_t, J_t) = \text{JoinBox}(B_t, A_s, A_{s'})$

if **is-Forget**(t) **then** $(M_t, N_t, J_t) = \text{ForgetBox}(B_t, v, A_s, A[B_t])$

■ **Figure 2** High level description of the algorithm **Fast Gaussian Elimination**. The entries of matrix U and the permutations leading to P and Q are constructed stepwise during the **ForgetBox** and **JoinBox** operations.

In the remainder of this section, we shall describe each operation in detail. To simplify our discussion, for a node t , we say that buffer rows and buffer columns have type 1, while regular rows and regular columns have type 2.

Leaf Box

When a node is a leaf corresponding to a bag B_t , then we apply procedure **Leaf Box**. This procedure simply initializes a box A_t where $m_t = 0$, $n_t = 0$, and J_t is a zero matrix of dimension $r_t \times c_t$ whose rows and columns are labeled by the elements of B_t , which contains r_t row vertices and c_t column vertices. No elementary row operations are performed.

Introduce Box

When a node t introduces a vertex u , its bag satisfies $B_t = B_s \cup \{u\}$, where B_s (which does not contain u) is the bag of its child s . The input of **Introduce Box** is the vertex u that has been introduced and the box A_s transmitted by its child. The box A_t is obtained by the

insertion of a new type 2 row or column labeled u whose elements are all zero⁴, taking care to insert it in the correct order (with respect to the canonical order of rows and columns of J_t). More precisely, if u is a row vertex v_i , this means that N_t and J_t are obtained from N_s and J_s , respectively, by the addition of a zero row, while $M_t = M_s$. It is clear that $n_t = n_s < 2r_s < 2(r_s + 1) = 2r_t$. Similarly, if v is a column vertex w_j , the addition of this column means that M_t and J_t are obtained from M_s and J_s , respectively, by the addition of a zero column, while $N_t = N_s$. It is clear that $m_t = m_s < 2c_s < 2(c_s + 1) = 2c_t$. No elementary row operations are performed.

Join Box

Let t be a node of type Join and let A_s and $A_{s'}$ be the boxes transmitted by its children, where $s < s' < t$. By the definition of box, A_s and $A_{s'}$ have the same rows and columns of type 2, that is, J_s and $J_{s'}$ have the same dimension and have rows and columns labeled by the same elements (in the same order). Moreover, because vertices of type 1 have been forgotten in their respective branches, the labels of the rows of M_s and $M_{s'}$ are all different, as are the labels of the columns of N_s and $N_{s'}$. The **JoinBox** operation first creates a matrix A_t^* whose rows and columns are labeled by the union of the labels of A_s and $A_{s'}$ with the structure below.

$$A_t^* = \begin{array}{|c|c|c|} \hline \mathbf{0}_{m_s \times n_s} & \mathbf{0}_{m_s \times n_{s'}} & M_s \\ \hline \mathbf{0}_{m_{s'} \times n_s} & \mathbf{0}_{m_{s'} \times n_{s'}} & M_{s'} \\ \hline N_s & N_{s'} & J_t^* \\ \hline \end{array}. \quad (3)$$

Here $J_t^* = J_s + J_{s'}$ has order $r_t \times c_t$ and records the net change of the entries of the original matrix because of row operations in the two branches that are being merged at this step. Let M_t^* be the $(m_s + m_{s'}) \times c_t$ matrix obtained by stacking M_s on top of $M_{s'}$ ⁵. Let N_t^* be the $r_t \times (n_s + n_{s'})$ matrix obtained by placing N_s to the left of $N_{s'}$.

If $m_s + m_{s'} < 2c_t$, define $M_t = M_t^*$. Otherwise $m_s + m_{s'} \geq 2c_t$. Apply Gaussian elimination to M_t^* until we get a matrix \tilde{M}_t in row echelon form. Here, we can use a standard algorithm for dense matrices that takes $O(c_t^3)$ operations, as M_t^* has c_t columns and at most $4c_t - 2$ rows. Since $m_s + m_{s'} \geq 2c_t$, this process ends with at least c_t zero rows. The matrix M_t is obtained from \tilde{M}_t by removing all the zero rows, and by reordering the rows so that they remain in the original ordering of M_t^* . Assume that the zero rows are labeled by the vertices $v_{i_1}, \dots, v_{i_\ell}$. The algorithm updates the permutation vector \mathbf{p} by assigning the values i_1, \dots, i_ℓ to the last ℓ components of \mathbf{p} with value 0, which means that the ℓ zero rows of M_t^* become the top zero rows in the description of Figure 1. Given that the new box is defined based on the matrix that has been modified, the type II row operations used for Gaussian elimination are authentic row operations. Given that the order of the rows has been preserved, type I row operations are bookkeeping operations (we observe that the labels of the rows used for authentic row operations will always refer to the original input matrix to account for the fact that row interchanging is not actually performed)⁶.

Similarly, if $n_s + n_{s'} < 2r_t$, define $N_t = N_t^*$. Otherwise, make an auxiliary copy of N_t^* and apply Gaussian elimination on it until we get a matrix \tilde{N}_t in row echelon form. As above, we can use a standard algorithm for dense matrices that takes $O(r_t^3)$ operations. There is

⁴ The added row or column may actually be empty if a new row vertex is introduced, but $c_s = 0$ and $n_s = 0$, or a new column vertex is introduced, but $r_s = 0$ and $m_s = 0$.

⁵ Alternatively, to preserve the canonical ordering of rows, we could merge the two matrices M_s and $M_{s'}$ in a way that preserves the canonical ordering. However, this does not affect the final result.

⁶ The algorithm could be easily adapted to a version where row interchanges are also authentic row operations, but this makes the analysis slightly more involved.

a set S_t of at most r_t columns that form a basis of the column space of N_t^* (precisely the columns of N_t^* with pivots in \tilde{N}_t). The matrix N_t is obtained from N_t^* by removing the columns that are not in S_t , from which there are at least $n_s + n_{s'} - r_t \geq r_t$.

► **Remark 2.** We emphasize that N_t is defined in terms of N_t^* rather than \tilde{N}_t in this case, which means that the row operations performed to obtain \tilde{N}_t are *not* performed in the full matrix while producing the sequence (2), but are instead auxiliary operations to find a basis of the column space of N_t^* and to write the remaining columns as linear combinations of the columns in this basis. This is crucial. If we consider the effect of the row operations in the full matrix, while the nonzero entries of the rows corresponding to M_t in the full matrix lie entirely within M_t (except perhaps for entries in columns that are linear combinations of other columns), there may be nonzero entries in the rows corresponding to N_t that lie outside the box A_t . Indeed, the rows of N_t are regular rows, so that they may even be processed in parallel by different branches of the tree decomposition. Adding multiples of them to other rows might affect entries that are not being monitored at this step. This is why the operations performed to turn a copy of N_t^* to row echelon form must be bookkeeping operations.

Coming back to the description of the algorithm, assume that $w_{j_1}, \dots, w_{j_\ell}$ are the labels of the columns of N_t^* with no pivot in \tilde{N}_t , so that they are linear combinations of the remaining columns of \tilde{N}_t . The algorithm updates the permutation vector \mathbf{q} by assigning the values j_1, \dots, j_ℓ to the last ℓ components of \mathbf{q} with value 0, which may be seen as moving these columns to the right (see Figure 1). Moreover, the corresponding entries of the vector \mathbf{w} record information about the linear combinations. For each column w_{j_i} , this information consists of the indices of the columns that produce the linear combination, the coefficients of this linear combination, and the number of pivots ρ at this step of the algorithm (as the linear combinations are with respect to the part of the column that does not include rows for which pivots have been assigned.)

Forget Box

Assume that t forgets vertex v and let s be its child, so that $B_t = B_s \setminus \{v\}$. This procedure uses A_s to produce a new box A_t where the row or column associated with v gets a pivot (and is therefore removed from the box) or changes to type 1.

Assume first that $v = v_i$ for a row i . Consider the matrix A_t^* given by⁷

$$A_t^* = \begin{array}{|c|c|} \hline \mathbf{0}_{m_s \times n_s} & M_t^* \\ \hline N_t^* & J_t^* \\ \hline \mathbf{x}_i & \mathbf{y}_i \\ \hline \end{array}, \quad (4)$$

where $M_t^* = M_s$, \mathbf{x}_i is row v_i of N_s , N_t^* is obtained from N_s by removing row \mathbf{x}_i , J_t^* is obtained from J_s by removing row v_i , and \mathbf{y}_i is obtained from row v_i of J_s by adding the entry a_{ij} of the input matrix to the entry corresponding to each column w_j .

⁷ Representing the row associated with v_i as the last row is helpful for visualizing the operations, but this need not be part of an implementation.

116:12 Gaussian Elimination for Low Treewidth Matrices

If \mathbf{x}_i is empty or zero, the algorithm sets $J_t = J_t^*$, and adds \mathbf{y}_i to M_t^* as the new bottom row to produce M_t^{**} . If M_t^{**} has at most $2c_t - 1$ rows, the algorithm calls this matrix M_t and concludes the step. Intuitively, this means that row v_i is a new buffer row. Otherwise M_t^{**} has $2c_t$ rows. The algorithm turns this matrix into a matrix \tilde{M}_t in row echelon form and removes $\ell \geq c_t$ zero rows of \tilde{M}_t to produce M_t , as was done in **Join Box**. As a byproduct of these authentic row operations, the algorithm updates \mathbf{p} . Similarly, if $n_s < 2r_t$, we set $N_t = N_t^*$. However, because $r_t = r_s - 1$ and the number of columns of N_t^* is n_s , it may be that $n_s \geq 2r_t$. If this happens, as was the case for **Join Box**, the algorithm performs bookkeeping operations to define N_t , and update \mathbf{q} and \mathbf{w} .

If \mathbf{x}_i is nonempty and nonzero, the algorithm selects the first nonzero entry of \mathbf{x}_i , suppose that it is in column w_j . It then performs row operations in A_t^* in order to eliminate all the other nonzero elements in column w_j (note that these elements are necessarily in rows of type 2, so that only rows in N_t^* and J_t^* are modified). The algorithm defines $\mathbf{p}(\rho) = i$ and $\mathbf{q}(\rho) = j$, while $\mathbf{u}(\rho)$ records information about the nonzero entries of the vector $[\mathbf{x}_i \ \mathbf{y}_i]$, namely the labels j of the nonzero columns of the vector $[\mathbf{x}_i \ \mathbf{y}_i]$ and the actual values of the nonzero entries. The algorithm then adds one to the value of ρ . Matrix J_t is assigned this modified J_t^* , matrix M_t is defined as M_t^* . Moreover, column j is removed from N_t^* to produce N_t^{**} . The row operations performed up to this point are authentic. If the number of columns in N_t^{**} is less than $2r_t$ (note that $r_t = r_s - 1$ and that N_t^{**} has $n_s - 1$ columns), N_t is set to be N_t^{**} . Otherwise, $n_t = 2r_t$ and the algorithm performs bookkeeping operations as in **Join Box** to define N_t , and update \mathbf{q} and \mathbf{w} .

Next assume that $v = w_j$ for a column j . Consider the matrix A_t^* given by

$$A_t^* = \begin{array}{|c|c|c|} \hline \mathbf{0}_{m_s \times n_s} & M_t^* & \mathbf{c}_j \\ \hline N_t^* & J_t^* & \mathbf{d}_j \\ \hline \end{array}, \quad (5)$$

where $N_t^* = N_s$, M_t^* is obtained from M_s by removing column w_j , \mathbf{c}_j is column w_j of M_s , J_t^* is obtained from J_s by removing column w_j and \mathbf{d}_j is obtained from column w_j of J_s by adding the entry a_{ij} (of the input matrix) to the entry corresponding to each row v_i .

If \mathbf{c}_j is empty or zero, the algorithm sets $J_t = J_t^*$ and adds \mathbf{d}_j as a new column of to N_t^* to produce N_t^{**} . If N_t^{**} has fewer than $2r_t$ columns, the algorithm calls this matrix N_t and concludes the step. Otherwise N_t^{**} has $2r_t$ columns and the algorithm defines N_t with bookkeeping operations as was done in **Join Box** (this leads to changes in \mathbf{q} and \mathbf{w}). Similarly, if $m_s < 2c_t$, we set $M_t = M_t^*$. However, because $c_t = c_s - 1$ and the number of rows of M_t^* is m_s , it may be that $m_s \geq 2c_t$. If this happens, the algorithm turns M_t^* into a matrix \tilde{M}_t in row echelon form and removes the $\ell \geq c_t$ zero rows of \tilde{M}_t to produce M_t , as was done above (this leads to changes in \mathbf{p}). These operations are authentic.

If \mathbf{c}_j is nonempty and nonzero, the algorithm selects the first nonzero entry of \mathbf{c}_j , suppose that it is in row i (of type 1). Let c_{ij} be this nonzero entry and let \mathbf{x}_i be row i in M_t^* . The algorithm then performs authentic row operations in A_t^* using row i to eliminate all the other nonzero elements in column j (note that only rows in M_t^* and J_t^* are modified). It then removes row i from M_t^* to produce M_t^{**} . As in the case of rows, the algorithm defines $\mathbf{p}(\rho) = i$ and $\mathbf{q}(\rho) = j$, while $\mathbf{u}(\rho)$ records the nonzero entries of $[\mathbf{x}_i, \mathbf{y}_i]$. It then updates ρ to $\rho + 1$. If the number of columns rows of M_t^{**} is less than $2c_t = 2c_s - 2$, M_t is set to be M_t^{**} . Otherwise, the algorithm turns M_t^{**} into a matrix \tilde{M}_t in row echelon form with authentic operations and removes $\ell \geq c_t$ zero rows of \tilde{M}_t to produce M_t , as was done in **Join Box** (this leads to changes in \mathbf{p}). Matrix N_t is defined as N_t^* .

3 Running time of the algorithm

In this section, we find an upper bound on the number of operation required to produce matrices P , Q and U as in the statement of Theorem 1 using algorithm **Fast Gaussian Elimination**. If the algorithm is given a tree decomposition \mathcal{T} with r nodes and width k of the underlying graph G of matrix M , it first computes a nice tree decomposition \mathcal{T}' of the same width k with fewer than $4(m+n)$ nodes in time $O(k(r+m+n))$ as discussed above. As it turns out, the number of nodes of each type in \mathcal{T}' is at most $m+n$.

We consider the number of operations performed at each type of node. **Leaf Box** initializes a matrix in $r_t \times c_t = O(k^2)$ trivial steps. **Introduce Box** uses $O(k)$ steps to create a row or column filled with zeros.

For **Join Box** and **Forget Box**, the main cost comes from row operations. As each row has size at most $n_t + c_t \leq 2r_t + c_t \leq 2(r_t + c_t) \leq 2k + 2$, each such row operation requires $O(k)$ additions and multiplications. Regarding **Forget Box**, when vertex v_i associated with a row i or vertex w_j associated with a column j is forgotten, either a new type 1 row or column is created, or a new pivot is found. If the latter happens, the algorithm performs at most $m_t + r_t - 1 = O(k)$ row operations, where each row has size at most $O(k)$, leading to $O(k^2)$ field operations. Additionally, the algorithm may perform row operations to keep M_t or N_t of the right size, just as in **Join Box** (see below). If a new type 1 row or column is created, the algorithm just inserts the new row in M_t or the new column in N_t , possibly performing row operations to keep M_t or N_t of the right size, just as in **Join Box**.

It remains to account for the row operations performed to keep matrices M_t and N_t of the right size. Recall that, every time row operations of this type are necessary to produce M_t , we start with a matrix of dimension $m^* \times c_t$, where $m^* \geq 2c_t$ and we create at least $m^* - c_t \geq c_t$ zero rows. These rows become processed. For N_t , we start with a matrix of dimension $r_t \times n^*$, where $n^* \geq 2r_t$ and we identify at least $n^* - r_t \geq r_t$ columns that are linear combinations of other columns. Again, the columns become processed. We shall keep track of the overall number of field operations performed at these steps by “charging” the rows and columns processed at this step, that is, if f field operations are performed and ℓ rows are processed, each of the ℓ rows is charged with f/ℓ operations. Because of the dimensions of the matrices, getting them to row echelon form takes $O(m^{*2}c_t)$ and $O(n^*r_t^2)$ field operations, respectively. In particular, each row can be charged at most $O(m^*c_t) = O(k^2)$ and each column can be charged at most $O(r_t^2) = O(k^2)$. Since any row or column may be processed at most once in this way, the overall number of field operations that the algorithm performs to keep matrices M_t or N_t of the right size is $O((m+n)k^2)$.

Overall, the number of operations taken by the algorithm to produce the vectors \mathbf{p} and \mathbf{q} , in addition to vectors $\mathbf{u}(i)$ and $\mathbf{w}(j)$ for every $i \leq \rho = \text{rank}(A)$ and every $j > \rho$ is $O(k(|\mathcal{T}| + m + n) + k^2(m + n))$.

We now consider the decomposition $U_t = P_t \tilde{A}_t Q_t$. The matrices P and Q are defined using \mathbf{p} and \mathbf{q} with no additional field operations. Indeed, $P = (p_{ij})$ is the permutation matrix such that $p_{\mathbf{p}(i)i} = 1$ for $1 \leq i \leq m$, while $Q = (q_{ij})$ is the permutation matrix such that $q_{\mathbf{q}(i)i} = 1$ for $1 \leq i \leq n$.

For U , the entries may be computed in two ways. They are either obtained from a vector $\mathbf{u}(i)$ with no additional computation, or they are obtained as a linear combination of $O(k)$ earlier values using coefficients recorded in $\mathbf{w}(j)$, which requires $O(k)$ sums and products. The number of operations performed to obtain entries of the second type is at most

$$\sum_{\ell=\text{rank}(A)+1}^n (m - \gamma_\ell) O(k) = O((n - \text{rank}(A))km). \quad (6)$$

This concludes the analysis.

4 Consequences of Theorem 1

We conclude the paper with the statement of results that are immediate consequences of Theorem 1.

► **Corollary 3.** *If A is an $m \times n$ matrix whose bipartite graph is given with a tree decomposition of width k with $O(m+n)$ nodes and b is a column vector of length n , then using $O(k^2(m+n))$ time and arithmetic operations, it can be decided whether the system of linear equations $Ax = b$ has a solution, and if so, then such a solution can be produced.*

Proof. To get one solution of the system of linear equations $Ax = b$, the algorithm to transform A into row echelon form is augmented by doing all row operations on the right hand side as well. If one ever gets a zero row on the left hand side together with a non-zero value on the right hand side, then the system has no solutions. Otherwise, with the left hand side in row echelon form, a solution is obtained by back-substitution. Hereby, all variables that correspond to columns that have been identified as linearly dependent during the algorithm, or just have no pivot, are set to 0. ◀

► **Corollary 4.** *The following hold if A is an $m \times n$ matrix whose bipartite graph is given with a tree decomposition of width k with $O(n)$ nodes:*

- (a) *The rank of A can be computed using $O(k^2(m+n))$ time and arithmetic operations.*
- (b) *For $m = n$, the determinant of A can be computed using $O(k^2n)$ time and arithmetic operations.*

Proof. The rank is just the number of non-zero rows of the output matrix U . If one of the columns is a linear combination of previous columns, the determinant is zero. Otherwise, the determinant is just the product of the signs of the permutation matrices P and Q with the product of the diagonal elements of the equivalent matrix in row echelon form. ◀

► **Corollary 5.** *For a graph given with a tree decomposition of width k with $O(n)$ nodes, the size of a maximum matching can be computed using $O(k^2n)$ time and arithmetic operations by a randomized algorithm with one-sided error that is correct with probability at least $1 - 1/n^c$ for an arbitrary constant. In case of an error, the algorithm reports a suboptimal value.*

Proof. This follows from Theorem 1.4 of Fomin et al. [5] by replacing their $O(k^3n)$ rank algorithm by our $O(k^2n)$ rank algorithm. ◀

References

- 1 U. Bertelè and F. Brioschi. *Nonserial Dynamic Programming*. Elsevier, 1972.
- 2 Hans L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput.*, 25(6):1305–1317, 1996. doi:10.1137/S0097539793251219.
- 3 Hans L. Bodlaender, Pål Grønås Drange, Markus S. Dregi, Fedor V. Fomin, Daniel Lokshtanov, and Michał Pilipczuk. A $c^k n$ 5-approximation algorithm for treewidth. *SIAM Journal on Computing*, 45(2):317–378, 2016. doi:10.1137/130947374.
- 4 Sally Dong, Yin Tat Lee, and Guanghai Ye. A nearly-linear time algorithm for linear programs with small treewidth: a multiscale representation of robust central path. In Samir Khuller and Virginia Vassilevska Williams, editors, *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21–25, 2021*, pages 1784–1797. ACM, 2021. doi:10.1145/3406325.3451056.

- 5 Fedor V. Fomin, Daniel Lokshtanov, Saket Saurabh, Michal Pilipczuk, and Marcin Wrochna. Fully polynomial-time parameterized computations for graphs and matrices of low treewidth. *ACM Trans. Algorithms*, 14(3):34:1–34:45, 2018. doi:10.1145/3186898.
- 6 Martin Fürer, Carlos Hoppen, and Vilmar Trevisan. Efficient diagonalization of symmetric matrices associated with graphs of small treewidth. *Theoretical Computer Science*, 1040:115187, 2025. doi:10.1016/j.tcs.2025.115187.
- 7 Martin C. Golumbic and Clinton F. Goss. Perfect elimination and chordal bipartite graphs. *Journal of Graph Theory*, 2(2):155–163, 1978. doi:10.1002/jgt.3190020209.
- 8 Rudolf Halin. S-functions for graphs. *Journal of Geometry*, 8(1):171–186, 1976. doi:10.1007/BF01917434.
- 9 Ton Kloks. *Treewidth, Computations and Approximations*, volume 842 of *Lecture Notes in Computer Science*. Springer, 1994. doi:10.1007/BFB0045375.
- 10 Tuukka Korhonen. A single-exponential time 2-approximation algorithm for treewidth. *SIAM Journal on Computing*, pages FOCS21–174, November 2023. doi:10.1137/22M147551X.
- 11 Seymour V. Parter. The use of linear graphs in Gauss elimination. *SIAM Review*, 3(2):119–130, 1961. doi:10.1137/1003021.
- 12 Richard Peng and Santosh S. Vempala. Solving sparse linear systems faster than matrix multiplication. *Commun. ACM*, 67(7):79–86, 2024. doi:10.1145/3615679.
- 13 Venkatesh Radhakrishnan, Harry B. Hunt III, and Richard E. Stearns. Efficient algorithms for solving systems of linear equations and path problems. In Alain Finkel and Matthias Jantzen, editors, *STACS 92, 9th Annual Symposium on Theoretical Aspects of Computer Science, Cachan, France, February 13-15, 1992, Proceedings*, volume 577 of *Lecture Notes in Computer Science*, pages 109–119. Springer, 1992. doi:10.1007/3-540-55210-3_177.
- 14 Neil Robertson and Paul D. Seymour. Graph minors. I. excluding a forest. *J. Comb. Theory B*, 35(1):39–61, 1983. doi:10.1016/0095-8956(83)90079-5.
- 15 Neil Robertson and Paul D. Seymour. Graph minors. II. algorithmic aspects of tree-width. *J. Algorithms*, 7(3):309–322, 1986. doi:10.1016/0196-6774(86)90023-4.
- 16 Donald J. Rose. Triangulated graphs and the elimination process. *Journal of Mathematical Analysis and Applications*, 32(3):597–609, 1970. doi:10.1016/0022-247X(70)90282-9.
- 17 Donald J. Rose and Robert E. Tarjan. Algorithmic aspects of vertex elimination on directed graphs. *SIAM Journal on Applied Mathematics*, 34(1):176–197, 1978.
- 18 Donald J. Rose, Robert E. Tarjan, and George S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal on Computing*, 5(2):266–283, 1976. doi:10.1137/0205021.
- 19 David R. Wood. On tree-partition-width. *European Journal of Combinatorics*, 30(5):1245–1253, 2009. Part Special Issue on Metric Graph Theory. doi:10.1016/j.ejc.2008.11.010.