


Efficient Diagonalization of Symmetric Matrices Associated with Graphs of Small Treewidth

Martin Fürer 

Pennsylvania State University, University Park, PA, USA
furer@cse.psu.edu

Carlos Hoppen 

Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil
choppen@ufrgs.br

Vilmar Trevisan 

Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil
trevisan@mat.ufrgs.br

Abstract

Let $M = (m_{ij})$ be a symmetric matrix of order n and let G be the graph with vertex set $\{1, \dots, n\}$ such that distinct vertices i and j are adjacent if and only if $m_{ij} \neq 0$. We introduce a dynamic programming algorithm that finds a diagonal matrix that is congruent to M . If G is given with a tree decomposition \mathcal{T} of width k , then this can be done in time $O(k|\mathcal{T}| + k^2n)$, where $|\mathcal{T}|$ denotes the number of nodes in \mathcal{T} .

2012 ACM Subject Classification Computing methodologies \rightarrow Linear algebra algorithms; Theory of computation \rightarrow Fixed parameter tractability; Mathematics of computing \rightarrow Graph theory

Keywords and phrases Treewidth, Diagonalization, Eigenvalues

Digital Object Identifier 10.4230/LIPIcs.ICALP.2020.52

Category Track A: Algorithms, Complexity and Games

Funding *Carlos Hoppen*: CNPq 308054/2018-0 and FAPERGS 19/2551-0001727-8.

Vilmar Trevisan: CNPq 409746/2016-9 and 303334/2016-9, CAPES-PRINT 88887.467572/2019-00, and FAPERGS 17/2551-0001.

1 Introduction and main result

Two matrices M and N are said to be *congruent*, which is denoted $M \cong N$, if there exists a nonsingular matrix P for which $N = P^T M P$. Matrix congruence naturally appears when studying Gram matrices associated with a quadratic form on a finite-dimensional vector space; finding a diagonal matrix D that is congruent to a symmetric matrix M allows us to classify the quadratic form.

In a different direction, finding a diagonal matrix that is congruent to M allows us to determine the number of eigenvalues of M in a given real interval. Indeed, fix real numbers $c < d$. Let $D_c \cong N = M - cI$ and $D_d \cong M - dI$ be diagonal matrices. By Sylvester's Law of Inertia [16, p. 568], the number n_1 of eigenvalues of M greater than c equals the number positive entries in D_c . Moreover, the number of eigenvalues equal to c , or less than c , are given by the number of zero diagonal entries, or by the number of negative entries in D_c , respectively. As a consequence, if n_2 is the number of positive entries in D_d , then $n_1 - n_2$ is the number of eigenvalues of M in the interval $(c, d]$.

Given a symmetric matrix $M = (m_{ij})$ of order n , we may associate it with a graph G with vertex set $[n] = \{1, \dots, n\}$ such that distinct vertices i and j are adjacent if and only if $m_{ij} \neq 0$. We say that G is the *underlying graph* of M . This allows us to employ structural decompositions of graph theory to deal with the nonzero entries of M in an efficient way.



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

47th International Colloquium on Automata, Languages, and Programming (ICALP 2020).

Editors: Artur Czumaj, Anuj Dawar, and Emanuela Merelli; Article No. 52; pp. 52:1–52:18

Leibniz International Proceedings in Informatics



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



One such decomposition is the tree decomposition, which has been extensively studied since the seminal paper of Robertson and Seymour [18]. The graph parameter associated with this decomposition is the *treewidth*.

A tree decomposition of a graph $G = (V, E)$ is a tree \mathcal{T} with nodes $1, \dots, m$, where each node i is associated with a bag $B_i \subseteq V$, satisfying the following properties: (1) $\bigcup_{i=1}^m B_i = V$; (2) For every edge $\{v, w\} \in E$, there exists B_i containing v and w ; (3) For any $v \in V$, the subgraph of \mathcal{T} induced by the nodes that contain v is connected.

The *width* of the tree decomposition \mathcal{T} is defined as $\max_i (|B_i| - 1)$ and the *treewidth* $tw(G)$ of graph G is the smallest k such that G has a tree decomposition of width k . Clearly, it is always the case that $tw(G) < |V|$, and for connected graphs, $tw(G) = 1$ if and only if G is a tree on two or more vertices. Even though computing the treewidth of a graph is hard, this is a widely studied parameter and tree decompositions with bounded width are known for several graph classes. Moreover, there is an extensive literature on approximation algorithms for this problem. For instance, Fomin et al. [9] devised an algorithm such that, given an n -vertex graph and an integer k , runs in time $O(k^7 n \log n)$ and either correctly reports that the treewidth of G is larger than k , or constructs a tree decomposition of G of width $O(k^2)$.

Often, graph widths have been used to design algorithms for NP-complete or even harder problems that are efficient on graphs of bounded width, by which we mean that their running time is equal to $O(f(k)n^c)$ for a constant c and an arbitrary computable function f , where k is the width of the graph. Problems that can be solved with such a running time are called fixed parameter tractable (FPT). For more information, interested readers are referred to [5, 7, 8, 17], and to the references therein.

On the other hand, graph widths may be quite useful for polynomial time solvable problems. Indeed, treewidth has a strong connection to the solution of sparse linear systems and Gaussian elimination. For a long time heuristics have been designed to maintain sparsity throughout Gaussian elimination. The goal is to minimize the *fill-in*, defined as the set of matrix positions that were initially 0, but have received nonzero values during the computation. Fill-in minimization has been analyzed as a graph problem in the case of symmetric [19] as well as asymmetric [20] matrices. As in our paper, a nonzero matrix entry a_{ij} is interpreted as an edge (or arc) from vertex i to vertex j . However, unlike here, it was assumed that the matrix was diagonally dominant, so that all the pivots could be chosen in the diagonal.

A major source of applications are symmetric positive definite matrices due to their prevalence. When the i th diagonal pivot is chosen in the symmetric case, then the fill-in consists of all missing edges between those neighbors of vertex i that have not yet been eliminated. The minimum k over all elimination orders of the maximal number of higher numbered neighbors is precisely the treewidth of the graph associated with a symmetric matrix [2]. When such an optimal elimination order or, equivalently, a tree decomposition of width k is given, then Gaussian elimination can trivially be done in time $O(k^2 n)$, if the pivot can always be chosen in the diagonal. This is clearly the case for symmetric positive definite matrices. On the other hand, it long seemed impossible to get an efficient algorithm when off-diagonal pivots have to be chosen. Note that, if the graph corresponding to a symmetric matrix has treewidth k , then you can always find a diagonal element with at most k off-diagonal nonzero elements in its row and column. But if this diagonal element is zero, and the pivot is chosen somewhere else in its row, then there can be an arbitrary number of nonzero elements in the pivot's column. The main contribution of this paper is an algorithm that deals with these diagonal elements.

The paper of Bodlaender et al. [2] is dedicated to the approximation of treewidth, pathwidth, and minimum elimination tree height. (Pathwidth is defined by tree decompositions where the tree is required to be a path.) In this context, they discuss the important matrix tasks of Cholesky factorization ($A = LL^T$) and Gaussian elimination for sparse matrices. These tasks have efficient FPT algorithms parameterized by treewidth or pathwidth, while their parallel complexity is determined by the minimum elimination tree height.

A recent paper of Fomin et al. [9] explores topics very similar to ours. For a matrix¹ with a given tree decomposition of width k , these authors present various efficient algorithms. In time $O(k^3n)$ they solve systems of linear equations, compute the determinant and the rank of a matrix. These problems are also solved by our approach, in the case of symmetric matrices. They also solve various matching and flow problems and compute an $O(k^2)$ approximation to treewidth in $k^{O(1)}n \log^{O(1)} n$ time.

Here, we design a dynamic programming algorithm with running time $O(k^2n)$ to find a diagonal matrix that is congruent to an input symmetric matrix M of order n , provided that a tree decomposition of width k for the underlying graph G is part of the input. In particular, for bounded treewidth, we find a linear-time solution to this problem. To achieve our goal, we were inspired by an algorithm with the same purpose for graphs with small clique-width (*clique-width* is defined based on another structural decomposition introduced by Courcelle and Olariu [4]). This algorithm was proposed by Jacobs and the current authors [12].

Astonishingly, the problem with the treewidth parameter turned out to be more challenging. This is somewhat counterintuitive and unusual, because graphs of bounded treewidth are also of bounded clique-width. Clearly, there is no direct implication. There are graphs whose clique-width is exponentially bigger than their treewidth [3]. However the implication of a running time of $2^{O(k)}n$ for a polynomially solvable problem is rather unimpressive and useless. The conclusion in the current paper is stronger. It is also much more useful, because practical examples of sparse matrices often have small treewidth, while a pattern of large minors with the same entry, typical of bounded clique-width matrices, is rather unusual.

As [9] and [12], the current paper fits into the recent trend “FPT within P”, which investigates fundamental problems that are solvable in polynomial time, but for which a lower exponent may be achieved using an FPT algorithm that is also polynomial in terms of the parameter k (see [14]). For our problem, it does not seem that $O(\text{poly}(k)n)$ algorithms are possible for graphs of fusion-width or multi-clique-width k [10, 11]. For some problems, like the independent set problem, there are algorithms whose running time as a function of these parameters is not worse than as a function of the clique-width, even though the clique-width can be exponentially larger. An $O(k^2n)$ algorithm for multi-clique-width k would imply the same time bound when k is the clique-width or the tree-width.

► **Theorem 1.** *Given a symmetric matrix M of order n and a tree decomposition \mathcal{T} of width k for the underlying graph of M , algorithm *CongruentDiagonal* (see Sect. 3) produces a diagonal matrix D congruent to M in time $O(k|\mathcal{T}| + k^2n)$.*

Naturally, we assume throughout this paper, that the input matrix M is given in a compact form because, in standard form, just reading M would already take quadratic time. A possible representation could be a list of triples (i, j, m_{ij}) containing all nonzero entries. For convenience, we assume the value m_{uv} for $u \neq v$ is just attached to the edge uv in the given tree decomposition. Likewise the value m_{uu} is attached to the vertex u . This representation could be obtained efficiently from the list representation. We will not discuss

¹ The matrices in [9] are not necessarily symmetric (in fact, not necessarily square), and they associate an $m \times n$ -matrix with a bipartite graph whose partition classes have size m and n .

such a transformation even though it is not trivial. An interesting discussion by Bodlaender et al. [1] shows how a data structure of size $O(kn)$ enables an $O(k)$ test whether two given vertices are adjacent, a detail that had previously often been overlooked.

Our paper is organized as follows. In Section 2, we define the concept of an edge-explicit tree decomposition and state auxiliary results about it. We then describe our algorithm in Section 3 and justify why it works as claimed, ending with an example.

2 Edge-explicit tree decomposition

In this paper, we consider graph decompositions based on the concept of *nice tree decomposition*, introduced by Kloks [15]. The current variation is due to Fürer and Yu [13], and to distinguish it from the original version, we call it an *edge-explicit tree decomposition*. (Nice tree decompositions with explicit nodes to introduce edges have been considered by Cygan et al. [6] before.) A rooted tree decomposition \mathcal{T} of a graph G whose nodes are associated with bags B_1, \dots, B_m is *edge-explicit* if each node i is of one of the following types:

- (a) **(Leaf)** The node i is a leaf of \mathcal{T} ;
- (b) **(Introduce Vertex)** The node i introduces vertex v if it has a single child j , $v \notin B_j$ and $B_i = B_j \cup \{v\}$.
- (c) **(Introduce Edges)** The node i introduces edges if it has a single child j and $|B_i| = |B_j|$. This node is labelled by a vertex $v \in B_i$ and inserts all edges $\{u, v\}$ of $E(G)$ such that $u \in B_i$.
- (d) **(Forget)** The node i forgets vertex v if i has a single child j , $v \notin B_i$ and $B_j = B_i \cup \{v\}$;
- (e) **(Join)** The node i is a join if it has two children j and ℓ , where $B_i = B_j = B_\ell$.

Unlike the other operations, the vertex v whose adjacencies are introduced by an Introduce Edges node must be given as part of the operation. We assume that every edge $uv \in E$ is introduced exactly once. In fact, for every edge uv , we will further suppose that, if j is the node that introduces the edge uv and it is labelled by vertex v , then the parent of j forgets v . Another assumption that will simplify our discussion is that $B_r = \emptyset$ for the root r of the tree, so that every vertex will be forgotten.

For constant k , Kloks [15] shows how to construct a nice tree decomposition of size at most $4n$ from a k -tree in time $O(n)$. We want to construct an edge-explicit tree decomposition from an arbitrary tree decomposition efficiently. With the help of the given tree decomposition of G of width k , we could embed G into a k -tree, apply the algorithm of Kloks, and finally add the Introduce Edges nodes. For our application, we want to analyze the dependence of the time on k precisely. For this purpose, we do a direct construction avoiding the k -trees.

► **Lemma 2.** *From a tree decomposition of width k and m nodes, an edge-explicit tree decomposition of the same width k with less than $5n$ nodes can be computed in time $O(k(m+n))$.*

Proof. We assume that all bags are given by a sorted list of their vertices. If these lists were unsorted, we could trivially sort them all in time $O((k \log k)m)$, or in a more sophisticated manner in time $O(km)$. As an additional preprocessing step, we produce a new node with empty bag and declare it to be the root. It is connected to an arbitrary node of the original tree decomposition.

Now we modify the tree decomposition in a sequence of depth-first tree traversals. Some of these traversals could be combined, but obviously without improving the asymptotic running time. During the first traversal, every node whose bag is contained in the bag of its parent is merged with the parent. Initially, the number of nodes m is not bounded by any function of the number of vertices n , because many subtrees could represent the same subgraph. From our argument below, it follows that the tree has a size less than $4n$ after this step.

In four more depth-first traversals, we produce an edge-explicit tree decomposition. In the second traversal, whenever the bag of a node has size less than the size of the bag of its parent, we add some nodes from the parent until both bags have the same size. This is a crucial step. Avoiding it, could increase the number of nodes in the fourth depth-first traversal to $\Omega(kn)$.

In the third traversal, all nodes with more than one child are replaced by binary trees with identical bags such that the original children appear as children of their leaves. In the fourth traversal, for any node i with a single child j , if necessary, some new nodes are inserted such that the bags only change by one vertex in each step. This is done from j to i by a sequence of nodes alternating between Forget nodes and Introduce Vertex nodes possibly followed by more Forget nodes. Now we have a nice tree decomposition, which we will show to have at most $4n$ nodes. Finally, in the fifth depth-first traversal, immediately below every Forget node of a vertex v , we insert an Introduce Edges node introducing the edges to v , to make the nice tree decomposition edge-explicit.

Now we count the nodes. Note that the root and the Leaf nodes are incident with a single edge, the Join nodes are incident with three edges and the other nodes are incident with two edges. As a consequence, there is one less Join node than Leaf node. Every vertex can be forgotten only once. Thus the number of Forget nodes is at most n . Between every Leaf node and the first Join node above it in the tree, there is at least one Forget node, otherwise, the leaf would have been merged with its parent in the first traversal. Thus there are at most n Leaf nodes and at most $n - 1$ Join nodes. The single child of every Introduce Vertex node is a Forget node. The same is true for the parent of every Introduce Edges node. Therefore, the number of Introduce Vertex nodes and the number of Introduce Edges nodes are at most n each. ◀

► **Remark 3.** The somewhat wasteful second traversal could be avoided. Its effect is to push Introduce Vertex nodes down the tree in order to avoid some of them when the corresponding vertices are introduced in leaves. This guarantees a bound of $O(n)$ rather than $O(kn)$ on the number of Introduce Vertex nodes.

Without changing the asymptotic running time nor the treewidth, a tree decomposition with typically many smaller bags could be obtained by allowing Introduce Vertex nodes introducing many vertices at once. For our application, this would work, because handling a node introducing many vertices could still be done in time $O(k^2)$.

For later use, we state an auxiliary result that records facts about an edge-explicit tree decomposition. We do not include a proof, as it follows directly from the definition of this concept. Let $G = (V, E)$ be a graph with tree decomposition \mathcal{T} , whose bags are B_1, \dots, B_m . For $v \in V$ and a node j of \mathcal{T} , let $\mathcal{T}(v)$ and \mathcal{T}_j be the subtree of \mathcal{T} induced by the nodes containing v and the branch of \mathcal{T} rooted at j , respectively.

► **Lemma 4.** *In an edge-explicit tree decomposition \mathcal{T} of a graph $G = (V, E)$, the following statements hold.*

- (a) *Every $v \in V$ is forgotten exactly once in \mathcal{T} .*
- (b) *Let $uv \in E$ and let i be the node that introduces the edge uv . If ℓ is an ancestor of i and ℓ is a join or a node that introduces a vertex, then $\{u, v\} \not\subseteq B_\ell$.*
- (c) *For every $v \in V$, the subtree $\mathcal{T}(v)$ of \mathcal{T} is rooted at the child of the node that forgets v . Moreover, the leaves of $\mathcal{T}(v)$ are precisely the leaves of \mathcal{T} that contain v and the nodes of \mathcal{T} that introduce v .*
- (d) *Suppose i forgets vertex v and j is its child. If $w \notin B_i$ and $\mathcal{T}(v) \cap \mathcal{T}(w) \neq \emptyset$, then $\mathcal{T}(w)$ is a subtree of \mathcal{T}_j . In particular, $\mathcal{T}(v)$ is a subtree of \mathcal{T}_j .*

3 The Algorithm

We now describe our diagonalization algorithm, which we call `CongruentDiagonal`. Let M be a symmetric matrix of order n and let $G = (V, E)$ be the underlying graph with vertex set $V = [n]$ associated with it. We wish to find a diagonal matrix congruent to M . Let \mathcal{T} be an edge-explicit tree decomposition of G of width k . The algorithm works bottom-up in the rooted tree \mathcal{T} , so we order the nodes $1, \dots, m$ of \mathcal{T} in post-order and operate on a node i after its children have been processed. It is well-known that two matrices are congruent if we can obtain one matrix from the other by a sequence of *pairs* of elementary operations, each pair consisting of a row operation followed by the *same* column operation. In our algorithm we only use congruence operations that permute rows and columns or add a multiple of a row and column to another row and column respectively. To achieve linear-time we must operate on a sparse representation of the graph associated with M , rather than on the matrix itself.

We start with a high-level description of the algorithm, which is summarized below. Each node i in the tree produces a pair of matrices $(N_i^{(1)}, N_i^{(2)})$, which may be combined into a symmetric matrix N_i of order at most $2(k+1)$. The algorithm traverses the tree decomposition from the leaves to the root so that, at node i , the algorithm either initializes a pair $(N_i^{(1)}, N_i^{(2)})$, or it produces $(N_i^{(1)}, N_i^{(2)})$ based on the matrices produced by its children, transmitting the pair to its parent. During this step, the algorithm may also produce diagonal elements of a matrix congruent to M . These diagonal elements are not transmitted by a node to its parent, but are appended to a global array as they are produced. At the end of the algorithm, the array consists of the diagonal elements of a diagonal matrix D that is congruent to M .

■ **Algorithm 1** High level description of the algorithm `CongruentDiagonal`.

```

CongruentDiagonal(M)
input:  an edge-explicit tree decomposition  $\mathcal{T}$  of the underlying
graph  $G$  associated with  $M$  of width  $k$  and the nonzero entries of  $M$ 
output: diagonal entries in  $D \cong M$ 
Order the nodes of  $\mathcal{T}$  as  $1, 2, \dots, m$  in post order
for  $i$  from 1 to  $m$  do
    if is-Leaf( $i$ ) then construct  $(N_i^{(1)}, N_i^{(2)}) = \text{LeafBox}(B_i)$ 
    if is-IntroduceVertex( $i$ ) then construct  $(N_i^{(1)}, N_i^{(2)}) = \text{IntroVertexBox}(B_i)$ 
    if is-IntroduceEdge( $i$ ) then construct  $(N_i^{(1)}, N_i^{(2)}) = \text{IntroEdgesBox}(B_i)$ 
    if is-Join( $i$ ) then construct  $(N_i^{(1)}, N_i^{(2)}) = \text{JoinBox}(B_i)$ 
    if is-Forget( $i$ ) then construct  $(N_i^{(1)}, N_i^{(2)}) = \text{ForgetBox}(B_i)$ 

```

In the remainder, we shall describe each operation in detail and justify that the algorithm `CongruentDiagonal` yields the desired output. Step i of the algorithm refers to the i th iteration of the loop above, and we assume that Step i processes the node i . To describe the matrix produced by each node, we need the concept of a matrix $M = (m_{ij})$ in *row echelon form*. This means that $m_{ij} = 0$ for all $j < i$. Moreover, let the *pivot* of row i be the first j such that $m_{ij} \neq 0$, if such an element exists. We require that distinct rows have different pivots.

Each matrix N_i produced by a node on the tree has the form

$$N_i = \begin{array}{|c|c|} \hline N_i^{(0)} & N_i^{(1)} \\ \hline N_i^{(1)T} & N_i^{(2)} \\ \hline \end{array}, \quad (1)$$

where $N_i^{(0)}$ is a matrix of dimension $k'_i \times k'_i$ whose entries are zero, $N_i^{(2)}$ is a symmetric matrix of dimension $k''_i \times k''_i$ and $N_i^{(1)}$ is a $k'_i \times k''_i$ matrix in row echelon form. Moreover, $0 \leq k'_i \leq k''_i \leq k + 1$. Observe that k'_i can be zero, in which case we regard $N_i^{(0)}$ and $N_i^{(1)}$ as empty. An important fact about N_i is that each of its rows (and the corresponding column) is associated with a vertex of G (equivalently, a row of M). Let $V(N_i)$ denote the set of vertices of G associated with the rows of N_i . We say that the k'_i rows in $N_i^{(0)}$ have *type-i* and the k''_i rows of $N_i^{(2)}$ have *type-ii*. This is represented by the partition $V(N_i) = V_1(N_i) \cup V_2(N_i)$, where $V_1(N_i)$ and $V_2(N_i)$ are the vertices of type-i and type-ii, respectively. As it turns out, the vertices of type-ii are precisely the vertices in B_i . When proving facts about the algorithm, we shall often refer to the matrix N_i as the result of processing B_i , even if the actual output is the pair $(N_i^{(1)}, N_i^{(2)})$.

The structure of the matrix N_i is described by the following lemma.

► **Lemma 5.** *For all $i \in [m]$, the matrix N_i defined in terms of the pair $(N_i^{(1)}, N_i^{(2)})$ produced by node i satisfies the following properties:*

- (a) $0 \leq k'_i \leq k''_i \leq k + 1$.
- (b) $N_i^{(1)}$ is a matrix in row echelon form.
- (c) $V_2(N_i) = B_i$.

To give an intuition about how the algorithm works, consider that we are trying to apply the strategy for Gaussian elimination described in the introduction. Vertices of type-ii would represent the rows that have never been used to eliminate elements of other rows, while vertices of type-i would be the nonzero rows that have already been used to eliminate elements in other rows, but for which the basic strategy of using the diagonal element as a pivot failed because it was equal to 0. The algorithm keeps these rows in a temporary buffer, which is maintained in row echelon form to make sure that its size k' satisfies $k' \leq k + 1$. In the process of maintaining row echelon form, some of these rows become diagonalised. In our algorithm, to preserve congruence, we perform the same Gaussian operations on rows and columns. Any row v of the input matrix M begins as a type-ii row. It can either be diagonalized during the application of ForgetBox to the node that forgets v , or it becomes a type-i row at this step, and finally becomes diagonalized in a later application of JoinBox or ForgetBox. Finally, we discuss the content of the boxes. Let $\tilde{M}(i)$ be the matrix that would be obtained by performing all row and column operations performed by the algorithm up to step i to the original matrix M . It turns out that, for a type-i row u in N_i and any row v in the matrix, the entries uv and vu in $\tilde{M}(i)$ and N_i coincide, if $v \in V(N_i)$, and the entries uv and vu in $\tilde{M}(i)$ are equal to 0, if $v \notin V(N_i)$. This is consistent with the intuition that rows of type-i have already been partially diagonalized and that their diagonal elements are 0. However, this connection does not hold in general for the entries of $\tilde{M}(i)$ and $N_i^{(2)}$, as $N_i^{(2)}$ can only capture changes the operations made for nodes in its branch of the tree decomposition, but vertices of type-ii could simultaneously lie in many different branches. This needs to be dealt with when looking at the effect of JoinBox.

To record the diagonal entries produced by the algorithm, let D_i be the set of all pairs (v, d_v) , where v is a vertex of G (equivalently, a row of M) and d_v is the diagonal entry associated with it, produced up to the end of step i . Let $\pi_1(D_i)$ and $\pi_2(D_i)$ be the projections of D_i onto their first and second coordinates, respectively, so that $\pi_1(D_i)$ is the set of rows that have been diagonalized up to the end of step i and $\pi_2(D_i)$ is the (multi)set of diagonal elements found up to this step. Note that, if we only require the algorithm to produce the diagonal entries of a diagonal matrix that is congruent to the input matrix, it is not necessary to actually keep track of the particular pairs in D_i .

Let M_0 be the input matrix M . Let $\tilde{M}(i)$ be the matrix that is congruent to M obtained by performing the row and column operations performed by the algorithm up to step i . Let M_i be the matrix obtained from M by replacing by 0 any entry m_{uv} such that $u \neq v$ and the edge uv has not been introduced in \mathcal{T}_i , or such that $u = v$ and $\mathcal{T}(v) \cap \mathcal{T}_i = \emptyset$. Let \tilde{M}_i be the matrix that is congruent to M_i produced by performing the row and column operations performed by the algorithm for all nodes in \mathcal{T}_i , in the order in which they have been performed. We only keep track of the matrices $\tilde{M}(i)$, M_i and \tilde{M}_i to prove the correctness of the algorithm, they are not stored by the algorithm. In what follows, given $S \subset V$ and a matrix Q whose rows and columns are indexed by V , we write $Q[S]$ for the principal submatrix of M indexed by S . Moreover, if $u, v \in V$, $Q[u, v]$ denotes the entry uv in Q .

Our main technical lemmas control the relationship between N_i and the diagonal elements produced in step i with the matrices M_i , \tilde{M}_i and $\tilde{M}(i)$. At the start of the algorithm, we set $\tilde{M}(-1) = \tilde{M}(0) = M$, $D_{-1} = D_0 = \emptyset$, $\mathcal{T}_0 = \emptyset$ and $V(N_0) = \emptyset$. The matrices N_0 , M_0 and \tilde{M}_0 are empty.

► **Lemma 6.** *The following facts hold for all $i \in \{0, \dots, m\}$.*

- (a) $\tilde{M}(i)$ and \tilde{M}_i are symmetric matrices congruent to M and M_i , respectively.
- (b) $D_{i-1} \subseteq D_i$.
- (c) If a multiple of row (or column) v has been added to a row (or column) u in step i , then $v \in \pi_1(D_i \setminus D_{i-1}) \cup V_1(N_i)$ and $u \in \pi_1(D_i \setminus D_{i-1}) \cup V(N_i)$.

The second lemma relates subtrees $\mathcal{T}(v)$ with the matrices produced by the algorithm.

► **Lemma 7.** *The following facts hold for all $i \in \{0, \dots, m\}$.*

- (a) If $v \in \pi_1(D_i) \cup V_1(N_i)$ and $\mathcal{T}_i \cap \mathcal{T}(v) \neq \emptyset$, then $\mathcal{T}(v)$ is a subtree of \mathcal{T}_i .
- (b) Let v be such that $\mathcal{T}(v) \cap \mathcal{T}_i \neq \emptyset$. Then $v \in V(N_i) \cup \pi_1(D_i)$.
- (c) If $v \in \pi_1(D_i \setminus D_{i-1}) \cup V(N_i)$, then $\mathcal{T}_i \cap \mathcal{T}(v) \neq \emptyset$.

The third result relates the entries of the matrices N_i and the set D produced by the algorithm with the entries of $\tilde{M}(i)$ and \tilde{M}_i .

► **Lemma 8.** *The following facts hold for all $i \in \{0, \dots, m\}$.*

- (a) If $\mathcal{T}(v) \cap \mathcal{T}(w) = \emptyset$ and $\tilde{M}(i)[v, w] \neq 0$, then $v, w \in V(N_j)$, where $j \leq i$ is the largest index for which $\mathcal{T}(v) \cap \mathcal{T}_j \neq \emptyset$ or $\mathcal{T}(w) \cap \mathcal{T}_j \neq \emptyset$. For \tilde{M}_i , $\tilde{M}_i[v, w] \neq 0$ only if $v, w \in V(N_i)$.
- (b) If $(v, d_v) \in D_i$, the row (and column) associated with v in $\tilde{M}(i)$, consists of zeros, with the possible exception of the v th entry, which is equal to d_v . If $\mathcal{T}_i \cap \mathcal{T}(v) \neq \emptyset$, then the row (and column) associated with v in \tilde{M}_i satisfy the same property.
- (c) If $v \in V_1(N_i)$, then the row (and column) associated with v in \tilde{M}_i coincides with the row (and column) associated with v (restricted to the elements of $u \in V(N_i)$) in $\tilde{M}(i)$. The entries uv and vu are equal to 0 if $u \notin B_i$ and are equal to the corresponding entries in N_i if $u \in B_i$. Moreover, the entries uv and vu of $\tilde{M}(i)$ for $u \notin V(N_i)$ are equal to 0.
- (d) Assume that $u, v \in B_i$. The entry uv of \tilde{M}_i is equal to the entry uv of $N_i^{(2)}$.

The proof of Lemmas 6, 7 and 8 is by induction on i . As $M_0 = M$, Lemma 6(a) is obviously true for $i = 0$, while Lemma 8(a) holds by definition of tree decomposition. The remaining items are vacuously true.

Before detailing each step of the algorithm, we show that, if the above lemmas hold for $i = m$, where m is the the number of nodes in the tree decomposition, then Algorithm `CongruentDiagonal` correctly computes a diagonal matrix congruent to M . To see why this is true, by Lemma 6(a), we know that M is congruent to $\tilde{M}(m)$. Moreover, by Lemma 8(b),

if $(v, d_v) \in D_m$, then the row (and column) associated with v in $\tilde{M}(m)$ consists of zeros, with the possible exception of the v th entry, which is equal to d_v . It remains to prove that $\pi_1(D) = V$. To this end, let $v \in V$ and let i be the node that forgets v given by Lemma 4(a). Let j be its child. Since $v \in B_j$, we have $\mathcal{T}(v) \cap \mathcal{T}_i \neq \emptyset$, so that $v \in V_1(N_i) \cup \pi_1(D_i)$ by Lemma 7(b) (we are using that $v \notin B_i = V_2(N_i)$, a consequence of Lemma 5(c)). Then $\mathcal{T}(v) \subset \mathcal{T}_i$ by Lemma 7(a). If $v \in \pi_1(D_i)$ we are done, so assume that $v \in V_1(N_i)$. Let ℓ be the parent of i . By Lemma 7(b), $v \in V_1(N_\ell) \cup \pi_1(D_\ell)$. We would again be done if $v \in \pi_1(D_\ell)$, otherwise we repeat the argument to show that $v \in V(N_p)$, where p is the parent of ℓ . This argument may be repeated inductively. The result now follows from the fact that the root m of the tree satisfies $B_m = \emptyset$, which implies that $V_2(N_m) = \emptyset$ by Lemma 5(c). This implies that $V_1(N_m) = \emptyset$ by Lemma 5(a), as required.

We now describe each step of Algorithm **CongruentDiagonal**. When the node is a leaf corresponding to a bag B_i of size b_i , then we apply procedure **LeafBox**. This procedure only initializes a matrix N_i to be transmitted up the tree. The matrix N_i is such that $k' = 0$ and $k'' = b_i$, where $N_i^{(2)}$ is the diagonal matrix such that, for every $v \in B_i$, the entry vv is given by the element vv in M . Observe that no off-diagonal entries appear in this initialization, as the edges involving vertices in B_i have yet to be introduced.

LeafBox(B_i)

input: a set B_i of size b_i

output: a matrix $N_i = (N_i^{(1)}, N_i^{(2)})$

Set $N_i^{(1)} = \emptyset$

$N_i^{(2)}$ is a diagonal matrix of order b_i

for each vertex $v \in B_i$ set entry vv of $N_i^{(2)}$ as m_{vv} .

■ **Figure 1** Procedure **LeafBox**.

By construction, the matrix N_i defined by **LeafBox** satisfies the properties of Lemma 5. It is not hard to show that, if Lemmas 6, 7 and 8 hold up to the end of step $i - 1$ and step i processes a leaf B_i , the lemmas must also hold at the end of step i .

Next, we explain the procedures associated with nodes of type **IntroduceVertex** and **IntroduceEdge**. For vertices, the input is the set B_i , the vertex v that has been introduced and the matrix $N_j = (N_j^{(1)}, N_j^{(2)})$ obtained after processing the child B_j of B_i . The matrix N_i is obtained from N_j by adding a new type-ii row/column corresponding to vertex v (this becomes the last row/column of the matrix). This row is zero everywhere with the exception of the diagonal entry vv , which is equal to m_{vv} .

For edges, the input is the set B_i , a vertex $v \in B_i$, the set $\Gamma_{B_i}(v)$ of neighbors of v in B_i and the matrix $N_j = (N_j^{(1)}, N_j^{(2)})$ produced after processing the child B_j of B_i . The matrix N_i is obtained from N_j by replacing the entries uv and vu in $N_j^{(2)}$, which are equal to some value α , by $\alpha + \beta$, where β is the entry uv in M .

It is obvious that the matrices N_i produced by **IntroVertexBox** and **IntroEdgesBox** satisfy the properties of Lemma 5. In both cases, no row/column operation is performed, $\tilde{M}(i) = \tilde{M}(i - 1)$ and $D_i = D_{i-1}$.

We now address the operation associated with nodes of type **join**. Let i be a node of type **join** and let N_j and N_ℓ be the matrices transmitted by its children, where $j < \ell < i$. By Lemma 5(c) and the definition of the join operation, we have $V_2(N_j) = V_2(N_\ell)$. By Lemma 7(a), we have $V_1(N_j) \cap V_1(N_\ell) = \emptyset$.

52:10 Efficient Diagonalization of Symmetric Matrices

IntroVertexBox(B_i, v, N_j)

input: a node i with bag B_i , child j , $B_j = B_i - v$, and $N_j = (N_j^{(1)}, N_j^{(2)})$

output: a matrix $N_i = (N_i^{(1)}, N_i^{(2)})$

$$N_i^{(1)} = N_j^{(1)}$$

$$N_i^{(2)} = N_j^{(2)}$$

Add zero row and zero column v to $N_i^{(2)}$

Add diagonal element vv to $N_i^{(2)}$ as m_{vv}

Add zero column v to $N_i^{(1)}$

■ **Figure 2** Procedure IntroVertexBox.

IntroEdgesBox($B_i, v, \Gamma_{B_i}(v), N_j$)

input: $v \in B_i$, $\Gamma_{B_i}(v)$ and a matrix $N_j = (N_j^{(1)}, N_j^{(2)})$

output: a matrix $N_i = (N_i^{(1)}, N_i^{(2)})$

$$N_i^{(1)} = N_j^{(1)}$$

$$N_i^{(2)} = N_j^{(2)}$$

For all $u \in \Gamma_{B_i}(v)$, set entries uv and vu of $N_i^{(2)}$ as $N_i^{(2)}[uv] + m_{uv}$

■ **Figure 3** Procedure IntroEdgesBox.

The JoinBox operation first creates a matrix N_i^* whose rows and columns are labelled by $V_1(N_j) \cup V_1(N_\ell) \cup V_2(N_j)$ with the structure below. Assume that $|V_1(N_j)| = r$, $|V_1(N_\ell)| = s$ and $|B_i| = t$.

$$N_i^* = \begin{array}{|c|c|c|} \hline \mathbf{0}_{r \times r} & \mathbf{0}_{r \times s} & N_j^{(1)} \\ \hline \mathbf{0}_{s \times r} & \mathbf{0}_{s \times s} & N_\ell^{(1)} \\ \hline N_j^{(1)T} & N_\ell^{(1)T} & N_i^{*(2)} \\ \hline \end{array}, \quad (2)$$

where $N_i^{*(2)} = N_j^{(2)} + N_\ell^{(2)} - M_i[B_i]$. We observe that, at this point, $M_i[B_i]$ is a diagonal matrix, as no edges with both endpoints in B_i may have been introduced by Lemma 4(b).

Note that the matrix

$$N_i^{*(1)} = \begin{array}{|c|} \hline N_j^{(1)} \\ \hline N_\ell^{(1)} \\ \hline \end{array}$$

is an $(r + s) \times t$ matrix consisting of two matrices in row echelon form on top of each other. We perform row and column operations on N_i^* involving rows associated with $V_1(N_j)$ (the *left rows*) and $V_1(N_\ell)$ (the *right rows*) to turn N_i^* into a matrix $N_i^{*(1)}$ in row echelon form. To do this, we proceed by steps in which we always add a multiple of a left or right row (and the corresponding column) to a right row (and the corresponding column): to choose the next operation, at each step we look at the pivots of the right rows and select the leftmost such pivot that coincides with a pivot of a left row or with the pivot of another right row (say w and v are the right and left/right rows that satisfy this, and j is the pivot. At the first step, v is always a left row). If R_w and C_w are the row and column corresponding to w , while $\alpha_j = R_w(j) = C_w(j)$ and $\beta_j = R_v(j) = C_v(j)$, we define

$$R_w \leftarrow R_w - \frac{\alpha_j}{\beta_j} R_v, C_w \leftarrow C_w - \frac{\alpha_j}{\beta_j} C_v.$$

JoinBox(B_i, N_j, N_ℓ)
input: a node i with bag B_i and matrices N_j, N_ℓ associated with its two children
output: a matrix $N_i = (N_i^{(1)}, N_i^{(2)})$
 $N_i^{(2)} = N_j^{(2)} + N_\ell^{(2)}$
 For every $v \in B_i$, set the entry vv of $N_i^{(2)}$ as $N_i^{(2)}[v, v] - m_{vv}$
 Construct $N_i^{(1)*} = \begin{bmatrix} N_j^{(1)} \\ N_\ell^{(1)} \end{bmatrix}$
 Do row and column operations on $N_i^{(1)*}$, putting it in row echelon form
 For each zero row of $N_i^{(1)*}$ (indexed by a vertex u), add $(u, 0)$ to D_i
 $N_i^{(1)}$ is $N_i^{(1)*}$ with zero row/columns removed

■ **Figure 4** Procedure JoinBox.

This eliminates the pivot of row w . Note that the entries in $N_j^{(2)}$ are not affected by these operations. Moreover, for all $u, v \in V_1(N_j) \cup V_1(N_\ell)$, the entry uv in N_i^* is equal to 0.

As we do this, we may create rows and columns (associated with the right rows) whose entries are all zero (for instance, this will certainly happen if $r + s > t$). If Z_i denotes the set of vertices associated with rows whose entries are all zero, where $|Z_i| = z$, we let $D_i = D_{i-1} \cup \{(v, 0) : v \in Z_i\}$, we remove the rows and columns associated with vertices in Z_i from $N_i^{*(1)}$ to produce the matrix

$$N_i = \begin{array}{|c|c|} \hline \mathbf{0}_{k' \times k'} & N_i^{(1)} \\ \hline N_i^{(1)T} & N_i^{(2)} \\ \hline \end{array}, \quad (3)$$

where $k' = r + s - z$, $k'' = t$ and $N_i^{(1)}$ is a matrix of dimension $k' \times k''$ in row echelon form and $N_i^{(2)} = N_i^{*(2)}$. We observe that N_i satisfies the properties of Lemma 5. Items (b) and (c) are satisfied by construction. For (a), the inequality $k' \leq k''$ is a consequence of the fact that $N_i^{(1)}$ is in row echelon form, while $k'' \leq k + 1$ follows from $k'' = |B_i|$. Proving that Lemmas 6, 7 and 8 hold after step i uses induction and the properties discussed above.

To conclude the description of the algorithm, we describe **ForgetBox**. Assume that i forgets vertex v and let j be its child, so that $B_i = B_j \setminus \{v\}$. By Lemma 4(c), we know that $\mathcal{T}(v)$ is a subtree of \mathcal{T}_j , and therefore all edges incident with v have been introduced. This procedure starts with $N_i^* = N_j$ and produces a new matrix N_i so that $v \in V_1(N_i)$ or $v \in \pi_1(D_i \setminus D_{i-1})$.

We look at N_i^* in the following way:

$$N_i^* = \begin{array}{|c|c|c|} \hline d_v & \mathbf{x}_v & \mathbf{y}_v \\ \hline \mathbf{x}_v^T & \mathbf{0}_{k' \times k'} & N_i^{*(1)} \\ \hline \mathbf{y}_v^T & N_i^{*(1)T} & N_i^{*(2)} \\ \hline \end{array}. \quad (4)$$

Here, the first row and column represent the row and column in N_j associated with v , while $N_i^{*(1)}$ and $N_i^{*(2)}$ are given by the submatrices of $N_j^{(1)}$ and $N_j^{(2)}$ obtained by removing the row and/or column associated with v . In particular \mathbf{x}_v and \mathbf{y}_v are row vectors of size k'_j and $k''_j - 1$, respectively.

52:12 Efficient Diagonalization of Symmetric Matrices

```

ForgetBox( $B_i, v, N_j,$ )
input: a node  $i$  with bag  $B_i$ , child  $j$  with  $B_i = B_j \setminus \{v\}$  and matrix  $N_j$ 
output: a matrix  $N_i = (N_i^{(1)}, N_i^{(2)})$ 
 $N_i = N_j$ 
Perform row/column exchange so that  $N_i$  has the form of (4)
if  $\mathbf{x}_v$  is empty or 0 then
  if  $\mathbf{y}_v$  is empty or 0 then
    add  $(v, d_v)$  to  $D$ 
    remove row  $v$  from  $N_i$ 
  else if  $d_v \neq 0$  then // Here  $\{\mathbf{y}_v \neq 0\}$ .
    use  $d_v$  to diagonalize row/column  $v$ 
    add  $(v, d_v)$  to  $D$  and remove row  $v$  from  $N_i$ 
  else // Here  $d_v = 0$ .
    Set  $u = \min\{w : y_w \neq 0\}$ 
    do row and column operations inserting row  $v$  to  $N_i^{(1)}$ 
    if a zero row is obtained add  $(v, 0)$  to  $D$  and remove row from  $N_i$ 
  else // Here  $\mathbf{x}_v \neq 0$ .
    use operations as in (5) to diagonalize rows  $u$  and  $v$ 
    add  $(v, d_v)$  and  $(u, d_u)$  to  $D$  and eliminate rows  $v$  and  $u$  from  $N_i$ .

```

■ **Figure 5** Procedure ForgetBox.

Depending on the properties of the vectors \mathbf{x}_v and \mathbf{y}_v , we proceed in different ways.

Case 1: \mathbf{x}_v is empty or $\mathbf{x}_v = [0 \cdots 0]$. If $\mathbf{y}_v = [0 \cdots 0]$ (or \mathbf{y}_v is empty), we add (v, d_v) to D_i and remove the row and column associated with v from N_i^* to produce N_i .

If $\mathbf{y}_v \neq [0 \cdots 0]$, there are again two options. If $d_v = 0$, the aim is to turn v into a row of type-i. To do this, we need to insert \mathbf{y}_v into the matrix $N_i^{*(1)}$ in a way that the ensuing matrix is in row echelon form. Note that this may be done by only adding multiples of rows of $V(N_i^{*(1)})$ to the row associated with v . At each step, if the pivot α_j of the (current) row associated with v is in the same position of the pivot β_j of R_u , the row associated with vertex u already in $N_i^{*(1)}$, we use R_u to eliminate the pivot of R_v :

$$R_v \leftarrow R_v - \frac{\alpha_j}{\beta_j} R_u, C_v \leftarrow C_v - \frac{\alpha_j}{\beta_j} C_u.$$

This is done until the pivot of the row associated with v may not be cancelled by pivots of other rows, in which case the row associated with v may be inserted in the matrix (to produce the matrix $N_i^{(1)}$), or until the row associated with v becomes a zero row, in which case $(v, 0)$ is added to D_i and we remove the row and column associated with v from N_i^* to produce N_i . If $d_v \neq 0$, we use d_v to eliminate the nonzero entries in y_v and diagonalize the row corresponding to v . For each element $u \in B_i$ such that the component α_v of y_v associated with u is nonzero, we perform

$$R_u \leftarrow R_u - \frac{\alpha_v}{d_v} R_v, C_u \leftarrow C_u - \frac{\alpha_v}{d_v} C_v.$$

When all such entries have been eliminated, we add (d_v, v) to D_i and we let N_i be the remaining matrix. Observe that, in this case, $N_i^{(1)} = N_i^{*(1)}$, only the elements of $N_i^{*(2)}$ are modified to generate $N_i^{(2)}$.

Case 2: \mathbf{x}_v is nonempty and $\mathbf{x}_v \neq [0 \cdots 0]$.

Let u be the vertex associated with the rightmost nonzero component of x_v . Let α_j be this component. We use this element to eliminate all the other nonzero entries in x_v , from right to left. Let w be the vertex associated with the entry α_ℓ . We perform

$$R_w \leftarrow R_w - \frac{\alpha_\ell}{\alpha_j} R_u, \quad C_w \leftarrow C_w - \frac{\alpha_\ell}{\alpha_j} C_u.$$

A crucial fact is that the entries corresponding to the matrix $N_i^{*(1)}$ in the matrix produced by these operations is still in row echelon form and has the same pivots as $N_i^{*(1)}$. If $d_v \neq 0$, we still use R_u to eliminate this element:

$$R_v \leftarrow R_v - \frac{d_v}{2\alpha_j} R_u, \quad C_v \leftarrow C_v - \frac{d_v}{2\alpha_j} C_u.$$

At this point, the only nonzero entries in the $(k' + 1) \times (k' + 1)$ left upper corner of the matrix obtained after performing these operations are in positions uv and vu (and are equal to α_j). We perform the operations

$$R_u \leftarrow R_u + \frac{1}{2} R_v, \quad C_u \leftarrow C_u + \frac{1}{2} C_v, \quad R_v \leftarrow R_v - R_u, \quad C_v \leftarrow C_v - C_u$$

The relevant entries of the matrix are modified as follows:

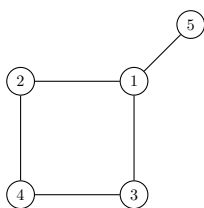
$$\begin{pmatrix} 0 & \alpha_j \\ \alpha_j & 0 \end{pmatrix} \rightarrow \begin{pmatrix} 0 & \alpha_j \\ \alpha_j & \alpha_j \end{pmatrix} \rightarrow \begin{pmatrix} -\alpha_j & 0 \\ 0 & \alpha_j \end{pmatrix}. \quad (5)$$

We are now in the position to use the diagonal elements to diagonalize the rows associated with v and u , as was done in Case 1, when $x_v = [0, \dots, 0]$ and $d_v \neq 0$. At the end of the step, we add $(v, -\alpha_j)$ and (u, α_j) to D_i .

Finally, it is time to analyze the complexity of Algorithm `CongruentDiagonal`, and prove Theorem 1.

Proof. The correctness of Algorithm `CongruentDiagonal` follows from the Lemmas and the justifications of every step of the algorithm as it is described throughout the paper. By Lemma 2, the time bound of $O(k|\mathcal{T}| + k^2n)$ is sufficient to transform an arbitrary given tree decomposition into an edge-efficient tree decomposition.

For the running time of the main computation, we have to analyze the procedures done at each type of tree node. `LeafBox` initializes a matrix in $O(k^2)$ trivial steps. `IntroVertexBox` and `IntroEdgesBox` use only $O(k)$ steps. For the other procedures, the main cost comes from row and column operations. As the matrices have order at most $k + 1$ each such operation costs $O(k)$. Regarding `ForgetBox`, when v is forgotten, either v is turned into a type-i vertex, or its row and column, and possibly the row and column of another vertex u , are diagonalized. The latter requires at most $O(k)$ row and column operations. If v is turned into a type-i vertex, then inserting it into the matrix $N_i^{(1)}$ in row echelon form takes at most $k + 1$ row operations. `JoinBox` can be most time consuming. To insert just one row vector into a matrix of order k in row echelon form, and preserving this property by adding multiples of one vector to another, can require up to $k + 1$ row operations. Each such operation can be done in time $O(k)$. To combine two matrices in row echelon form into one such matrix, up to $k + 1$ row vectors are inserted. Thus the total time for this operation is $O(k^3)$. This immediately results in an upper bound of $O(k^3n)$ for the whole computation.

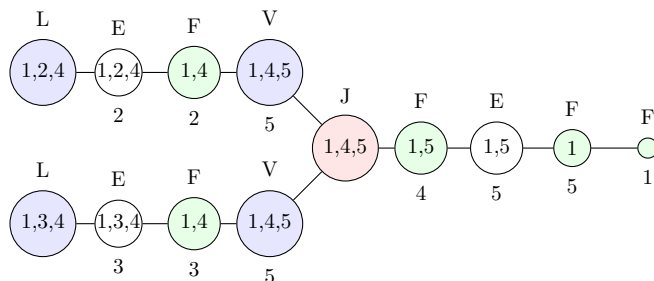


■ **Figure 6** Graph with 5 labeled vertices.

To obtain an $O(k|\mathcal{T}| + k^2n)$ bound for the whole computation, we have to employ a different accounting scheme for the time spent to merge two matrices in row echelon form into one during the `JoinBox` procedure. We notice that every row operation in any $N_i^{(1)}$ creates at least one 0 entry in some row, meaning that its pivot moves at least one position to the right or creates a zero row. For every vertex v , its row is added at most once to some $N_i^{(1)}$, namely when v is forgotten. Its pivot can move at most k times and disappear at most once. Thus for all n vertices together, at most $(k + 1)n$ row operations can occur in all $N_i^{(1)}$ together. This only uses time $O(k^2n)$. Thus, while during a single join procedure, $\Omega(k^2)$ row operations might be needed, the average is $O(k)$ such operations per join procedure. ◀

4 Example

In this section, we illustrate how the algorithm acts on a concrete example. To this end, we consider the graph in Figure 6. An edge-explicit tree decomposition representing this graph may be seen in Figure 7.



■ **Figure 7** An edge-explicit tree representing the graph, where the root is on the right. A label above describes the type of node, a label below indicates which vertices or edges are introduced or forgotten.

Note that G is the underlying graph of the symmetric matrix

$$M = \begin{pmatrix} 1 & 1 & 1 & 0 & -1 \\ 1 & 0 & 0 & 2 & 0 \\ 1 & 0 & 1 & -1 & 0 \\ 0 & 2 & -1 & 1 & 0 \\ -1 & 0 & 0 & 0 & -1 \end{pmatrix}.$$

Suppose that we want to find the number of eigenvalues greater than 0 (and equal to and less than 0). We apply our algorithm with $c = 0$, that is, originally $M - cI = M$.

Assume that we have ordered the nodes of the tree in Figure 7 in post order so that the first five nodes are in the upper branch of Figure 7, followed by the five nodes on the lower branch and by the five nodes starting from the node of type join. When we start, the node

Leaf calls the LeafBox with bag of vertices $\{1, 3, 4\}$ producing the matrix $N_1 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$,

whereas the node Introduce Edges labelled by vertex 3 introduces edges 13 and 34, leading to the matrix

$$N_3 = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & -1 \\ 0 & -1 & 1 \end{pmatrix}.$$

The node Forget Vertex $F3$ receives the matrix N_3 and, after exchanging rows and columns 1 and 2 (so that vertex 3 corresponds to first row), processes the matrix

$$\begin{pmatrix} 1 & 1 & -1 \\ 1 & 1 & 0 \\ -1 & 0 & 1 \end{pmatrix}.$$

According to our description, we are in Case 1 of Procedure ForgetBox, with \mathbf{x}_v empty and $\mathbf{y}_v = [1, -1] \neq [0, 0]$. Since $d_v = 1$, the algorithm diagonalizes the row/columns corresponding to v . To this end, we perform the operations $R_2 \leftarrow R_2 - R_1$, followed by $C_1 \leftarrow C_1 - C_2$, producing the matrix

$$N_3 = \begin{pmatrix} 1 & 0 & -1 \\ 0 & 0 & 1 \\ -1 & 1 & 1 \end{pmatrix},$$

followed by the operations $R_3 \leftarrow R_3 + R_1$, followed by $C_3 \leftarrow C_3 + C_1$, giving

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}.$$

We have diagonalized row 1, corresponding to vertex 3. Hence, the node $F3$ sets the diagonal vector $D = (v, d_v) = (3, 1)$ and transmits the matrix $N_4 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$, whose rows are indexed by vertices 1 and 4, respectively, to its parent. The node $V5$ introduces vertex 5, producing the matrix

$$N_5 = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & -1 \end{pmatrix},$$

indexed by the vertices 1, 4 and 5, respectively. We notice that this matrix N is such that $N_5^{(1)}$ is empty and $N_5^{(2)} = N$.

Working on the lower branch of the tree of Figure 7 in a similar way, we arrive at node $F2$, after exchanging the rows/columns, with the matrix

$$N_9^* = \begin{pmatrix} 0 & 1 & 2 \\ 1 & 1 & 0 \\ 2 & 0 & 1 \end{pmatrix},$$

indexing vertices 2, 1 and 4, respectively. This corresponds to Case 1 of Procedure ForgetBox, with empty \mathbf{x}_v and $\mathbf{y}_v = [1, 2]$, but $d_v = 0$. We notice that, in this particular case, the matrix $N_9^{(1)} = [1, 2]$ is already in row echelon form, so that $N_9 = (N_9^{(1)}, N_9^{(2)})$, $N_9^{(2)} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, is transmitted by F_2 to its parent.

52:16 Efficient Diagonalization of Symmetric Matrices

Now the Introduce Vertex node $V5$ processes the matrix N_9 and produces matrix

$$N_{10} = \left(\begin{array}{c|ccc} 0 & 1 & 2 & 0 \\ \hline 1 & 1 & 0 & 0 \\ 2 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & -1 \end{array} \right),$$

where the rows are indexed by the vertices 2, 1, 4 and 5, respectively.

Now the JoinBox procedure will process the matrices N_5 and N_{10} . We first do the operation $N_{11}^{(2)} = N_{10}^{(2)} + N_5^{(2)} - M_D[1, 4, 5]$, where $M_D[1, 4, 5]$ is the diagonal matrix whose rows and columns are indexed by 1, 4 and 5 such that each entry ii is given by m_{ii} . We then merge $N_{10}^{(1)}$ on top of $N_5^{(1)}$. Since $N_5^{(1)}$ is empty, these operation produce the matrix

$$N_{11} = \left(\begin{array}{c|ccc} 0 & 1 & 2 & 0 \\ \hline 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \\ \hline 0 & 0 & 0 & -1 \end{array} \right),$$

whose rows index the vertices 2, 1, 4 and 5, respectively.

We now process node $F4$ of the tree, where the vertex 4 is forgotten. We first exchange rows and columns so that the first row is indexed by 4. The matrix becomes

$$N_{12}^* = \left(\begin{array}{c|ccc} 0 & 2 & 1 & 0 \\ \hline 2 & 0 & 1 & 0 \\ \hline 1 & 1 & 0 & 0 \\ \hline 0 & 0 & 0 & -1 \end{array} \right),$$

whose rows index the vertices 4, 2, 1 and 5, respectively. We look at this matrix as in equation (4). We are in case 2 of Procedure ForgetBox with $d_v = 0$, $\mathbf{x}_v = [2]$, $\mathbf{y}_v = [1, 0]$, $N_{12}^{*(1)} = [1, 0]$ and $N_{12}^{*(2)} = \begin{pmatrix} 0 & 0 \\ 0 & -1 \end{pmatrix}$. Since $\mathbf{x}_v = [2]$, there is no operation to perform in order to put \mathbf{x}_v in row echelon form. The goal now is to transform the left upper corner of the above matrix $\begin{pmatrix} 0 & 2 \\ 2 & 0 \end{pmatrix}$ into a diagonal matrix. We perform the operations $R_2 \leftarrow R_2 + 1/2R_1$, $C_2 \leftarrow C_2 + 1/2C_1$ followed by $R_1 \leftarrow R_1 - R_2$ and $C_1 \leftarrow C_1 - C_2$, obtaining the matrix

$$N_{12}^* = \left(\begin{array}{cccc} -2 & 0 & -1/2 & 0 \\ 0 & 2 & 3/2 & 0 \\ -1/2 & 2 & 2 & 0 \\ 0 & 0 & 0 & -1 \end{array} \right).$$

We now use the nonzero pivots obtained in order to diagonalize rows 1 and 2. To achieve this, we perform the operations $R_3 \leftarrow R_3 - 1/4R_1$, $C_3 \leftarrow C_3 - 1/4C_1$, followed by $R_3 \leftarrow R_3 - 3/4R_2$, $C_3 \leftarrow C_3 - 3/4C_2$. This produces the matrix

$$N_{12}^* = \left(\begin{array}{cccc} -2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{array} \right).$$

At this point, the first two rows are diagonalized, corresponding to vertices 4 and 2. To the diagonal vector $D = (v, d_v)$ we added the components (4, -2) and (2, 2). Node $F4$ transmits the matrix $N_{12} = \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix}$, corresponding the edges 1 and 5. The Introduce Edges node

$E5$ puts the edge 15, and the matrix returned by `IntroduceEdgesBox` is $N_{13} = \begin{pmatrix} -1 & -1 \\ -1 & -1 \end{pmatrix}$ in the form the Forget node $F5$ receives it. We see that \mathbf{x}_v is empty and $\mathbf{y}_v = [-1]$ and $d_v = -1$, meaning that we are in case 2. Using d_v as pivot we arrive at $N_{14}^* = \begin{pmatrix} -1 & 0 \\ 0 & 0 \end{pmatrix}$, adding to the diagonal vector D the component $(v, d_v) = (5, -1)$, and returning the matrix $N_{14} = [0]$. The final node $F1$ forgets the vertex 1. Since the matrix received is $[0]$ already in diagonal form, it adds to D the component $(v, d_v) = (1, 0)$. The diagonal vector D returned by the algorithm is

$$\begin{pmatrix} v \\ d_v \end{pmatrix} = \begin{pmatrix} 3 & 4 & 2 & 5 & 1 \\ 1 & -2 & 2 & -1 & 0 \end{pmatrix},$$

meaning that M has 2 positive eigenvalues, 2 negative eigenvalues and 0 is an eigenvalue with multiplicity 1.

References

- 1 Hans L. Bodlaender, Paul S. Bonsma, and Daniel Lokshtanov. The fine details of fast dynamic programming over tree decompositions. In Gregory Z. Gutin and Stefan Szeider, editors, *IPEC*, volume 8246 of *Lecture Notes in Computer Science*, pages 41–53. Springer, 2013.
- 2 Hans L. Bodlaender, John R. Gilbert, Hjálmtýr Hafsteinsson, and Ton Kloks. Approximating treewidth, pathwidth, frontsize, and shortest elimination tree. *J. Algorithms*, 18(2):238–255, 1995.
- 3 Derek G. Corneil and Udi Rotics. On the relationship between clique-width and treewidth. *SIAM J. Comput.*, 34(4):825–847, 2005.
- 4 Bruno Courcelle and Stephan Olariu. Upper bounds to the clique width of graphs. *Discrete Appl. Math.*, 101(1-3):77–114, 2000.
- 5 Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2015.
- 6 Marek Cygan, Jesper Nederlof, Marcin Pilipczuk, Michal Pilipczuk, Joham M. M. van Rooij, and Jakub Onufry Wojtaszczyk. Solving connectivity problems parameterized by treewidth in single exponential time. In *Proceedings of the 2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, pages 150–159, 2011.
- 7 Rodney G. Downey and Michael R. Fellows. *Fundamentals of Parameterized Complexity*. Texts in Computer Science. Springer, 2013.
- 8 Jörg Flum and Martin Grohe. *Parameterized Complexity Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2006.
- 9 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.
- 10 Martin Fürer. A natural generalization of bounded tree-width and bounded clique-width. In Alberto Pardo and Alfredo Viola, editors, *LATIN 2014: Theoretical Informatics - 11th Latin American Symposium, Montevideo, Uruguay, March 31 - April 4, 2014. Proceedings*, volume 8392 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 2014. doi:10.1007/978-3-642-54423-1_7.
- 11 Martin Fürer. Multi-clique-width. In Christos H. Papadimitriou, editor, *8th Innovations in Theoretical Computer Science Conference, ITCS 2017, January 9-11, 2017, Berkeley, CA, USA*, volume 67 of *LIPICs*, pages 14:1–14:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICs.ITCS.2017.14.
- 12 Martin Fürer, Carlos Hoppen, David P. Jacobs, and Vilmar Trevisan. Eigenvalue location in graphs of small clique-width. *Linear Algebra and its Applications*, 560:56–85, 2019.

- 13 Martin Fürer and Huiwen Yu. Space saving by dynamic algebraization based on tree-depth. *Theory of Computing Systems*, 61(2):283–304, 2017.
- 14 Archontia C. Giannopoulou, George B. Mertzios, and Rolf Niedermeier. Polynomial fixed-parameter algorithms: A case study for longest path on interval graphs. *Theor. Comput. Sci.*, 689:67–95, 2017. doi:10.1016/j.tcs.2017.05.017.
- 15 T. Kloks. *Treewidth: Computations and Approximations*, volume 842 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.
- 16 Carl Meyer. *Matrix analysis and applied linear algebra*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2000.
- 17 Rolf Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Oxford University Press, 2006.
- 18 Neil Robertson and Paul D. Seymour. Graph minors II. Algorithmic aspects of tree-width. *J. Algorithms*, 7(3):309–322, 1986.
- 19 Donald J. Rose, R. Endre Tarjan, and George S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal on Computing*, 5(2):266–283, 1976.
- 20 Donald J. Rose and Robert Endre Tarjan. Algorithmic aspects of vertex elimination on directed graphs. *SIAM Journal on Applied Mathematics*, 34(1):176–197, 1978.