Linear MIM-Width of Trees ⋆

Svein Høgemo, Jan Arne Telle, and Erlend Raa Vågset

Department of Informatics, University of Bergen, Norway.
{svein.hogemo, jan.arne.telle, erlend.vagset}@uib.no

Abstract. We provide an $O(n \log n)$ algorithm computing the linear maximum induced matching width of a tree and an optimal layout.

1 Introduction

The study of structural graph width parameters like tree-width, clique-width and rank-width has been ongoing for a long time, and their algorithmic use has been steadily increasing [11, 17]. The maximum induced matching width, denoted MIM-width, and the linear variant LMIM-width, are graph parameters having very strong modelling power introduced by Vatshelle in 2012 [19]. The LMIM-width parameter asks for a linear layout of vertices such that the bipartite graph induced by edges crossing any vertex cut has a maximum induced matching of bounded size. Belmonte and Vatshelle [2] [1] showed that INTERVAL graphs, BI-INTERVAL graphs, CONVEX graphs and PERMUTATION graphs, where clique-width can be proportional to the square root of the number of vertices [10], all have LMIM-width 1 and an optimal layout can be found in polynomial time.

Since many well-known classes of graphs have bounded MIM-width or LMIM-width, algorithms that run in XP time in these parameters will yield polynomial-time algorithms on several interesting graph classes at once. Such algorithms have been developed for many problems: by Bui-Xuan et al [4] for the class of LCVS-VP - Locally Checkable Vertex Subset and Vertex Partitioning - problems, by Jaffke et al for non-local problems like FEEDBACK VERTEX SET [14, 13] and also for GENERALIZED DISTANCE DOMINATION [12], by Golovach et al [9] for output-polynomial ENUMERATION OF MINIMAL DOMINATING sets, by Bergougnoux and Kanté [3] for several Connectivity problems and by Galby et al for SEMITOTAL DOMINATION [8]. These results give a common explanation for many classical results in the field of algorithms on special graph classes and extends them to the field of parameterized complexity.

Note that very low MIM-width or LMIM-width still allows quite complex cuts compared to similarly defined graph parameters. For example, carving-width 1 allows just a single edge, maximum matching-width 1 a star graph, and rank-width 1 a complete bipartite graph. In contrast, LMIM-width 1 allows any cut

⋆ This is the appendix of our WG submission, the long version with extra figures and full proofs

[1] In [2], results are stated in terms of d-neighborhood equivalence, but in the proof, they actually gave a bound on LMIM-width.

where the neighborhoods of the vertices in a color class can be ordered linearly w.r.t. inclusion. In fact, it is an open problem whether the class of graphs having LMIM-width 1 can be recognized in polynomial-time or if this is NP-complete. Sæther et al [18] showed that computing the exact MIM-width and LMIM-width of general graphs is W-hard and not in APX unless NP=ZPP, while Yamazaki [20] shows that under the small set expansion hypothesis it is not in APX unless P=NP. The only graph classes where we know an exact polynomial-time algorithm computing LMIM-width are the above-mentioned classes INTERVAL, BI-INTERVAL, CONVEX and PERMUTATION that all have structured neighborhoods implying LMIM-width 1 [2]. Belmonte and Vatshelle also gave polynomial-time algorithms showing that CIRCULAR ARC and CIRCULAR PERMUTATION graphs have LMIM-width at most 2, while DILWORTH k and k-TRAPEZOID have LMIM-width at most k [2]. Recently, Fomin et al [7] showed that LMIM-width for the very general class of H-GRAPHS is bounded by $2|E(H)|$, and that a layout can be found in polynomial time if given an H-representation of the input graph. However, none of these results compute the exact LMIM-width. On the negative side, Mengel [15] has shown that STRONGLY CHORDAL SPLIT graphs, CO-COMPARABILITY graphs and CIRCLE graphs all can have MIM-width, and LMIM-width, linear in the number of vertices.

Just as LMIM-width can be seen as the linear variant of MIM-width, path-width can be seen as the linear variant of tree-width. Linear variants of other well-known parameters like clique-width and rank-width have also been studied. Arguably, the linear variant of MIM-width commands a more noteworthy position, since in contrast to these other linear parameters, for almost all well-known graph classes where the original parameter (MIM-width) is bounded then also the linear variant (LMIM-width) is bounded.

In this paper we give an $O(n \log n)$ algorithm computing the LMIM-width of an n-node tree. This is the first graph class of LMIM-width larger than 1 having a polynomial-time algorithm computing LMIM-width and thus constitutes an important step towards a better understanding of this parameter. The path-width of trees was first studied in the early 1990s by Möhring [16], with Ellis et al [6] giving an $O(n \log n)$ algorithm computing an optimal path-decomposition, and Bodlaender [**?**] an $O(n)$ algorithm. In 2013 Adler and Kanté [1] gave linear-time algorithms computing the linear rank-width of trees and also the linear clique-width of trees, by reduction to the path-width algorithm. Even though LMIM-width is very different from path-width, the basic framework of our algorithm is similar to the path-width algorithm in [6].

In Section 2 we give some standard definitions and prove the Path Layout Lemma, that if a tree T has a path P such that all components of $T \setminus N[P]$ have LMIM-width at most k then T itself has a linear layout with LMIM-width at most $k+1$. We use this to prove a classification theorem stating that a tree T has LMIM-width at least $k+1$ if and only if there is a node v such that after rooting T in v, at least three children of v *themselves have at least one child* whose rooted subtree has LMIM-width at least k. From this it follows that the LMIM-width of an n-node tree is no more than $\log n$. Our $O(n \log n)$ algorithm computing

LMIM-width of a tree T picks an arbitrary root r and proceeds bottom-up on the rooted tree T_r. In Section 3 we show how to assign labels to the rooted subtrees encountered in this process giving their LMIM-width. However, as with the algorithm computing pathwidth of a tree, the label is sometimes complex, consisting of LMIM-width of a sequence of subgraphs, of decreasing LMIM-width, that are not themselves full rooted subtrees. Proposition 1 is an 8-way case analysis giving a subroutine used to update the label at a node given the labels at all children. In Section 4 we give our bottom-up algorithm, which will make calls to the subroutine underlying Proposition 1 in order to compute the complex labels and the LMIM-width. Finally, we use all the computed labels to lay out the tree in an optimal manner.

2 Classifying LMIM-width of Trees

We use standard graph theoretic notation, see e.g. [5]. For a graph $G = (V, E)$ and subset of its nodes $S \subseteq V$ we denote by $N(S)$ the set of neighbors of nodes in S, by $N[S] = S \cup N(S)$ its closed neighborhood, and by $G[S]$ the graph induced by S. For a bipartite graph G we denote by MIM(G), or simply MIM if the graph is understood, the size of its Maximum Induced Matching, the largest number of edges whose endpoints induce a matching. Let σ be the linear order corresponding to the enumeration $v_1, \ldots, v_n$ of the nodes of G, this will also be called a linear layout of G. For any index $1 \leq i < n$ we have a cut of σ that defines the bipartite graph on edges "crossing the cut" i.e. edges with one endpoint in $\{v_1, \ldots, v_i\}$ and the other endpoint in $\{v_{i+1}, \ldots, v_n\}$. The maximum induced matching of G under layout σ is denoted $mim(\sigma, G)$, and is defined as the maximum, over all cuts of σ, of the value attained by the MIM of the cut, i.e. of the bipartite graph defined by the cut. The linear induced matching width – LMIM-width – of G is denoted $lmw(G)$, and is the minimum value of $mim(\sigma, G)$ over all possible linear orderings σ of the vertices of G.

We start by showing that if we have a path P in a tree T then the LMIM-width of T is no larger than the largest LMIM-width of any component of $T \setminus N[P]$, plus 1 . To define these components the following notion is useful.

Definition 1 (Dangling tree). *Let T be a tree containing the adjacent nodes v and u. The dangling tree from v in u, $T\langle v, u\rangle$, is the component of $T \setminus (u, v)$ containing u.*

Given a node $x \in T$ with neighbours $\{v_1, \ldots, v_d\}$, the forest obtained by removing $N[x]$ from T is a collection of dangling trees $\{T\langle v_i, u_{i,j}\rangle\}$, where $u_{i,j} \neq x$ is some neighbour of v_i. We can generalise this to a path $P = (x_1, \ldots, x_p)$ in place of x, such that $T \backslash N[P] = \{T\langle v_{i,j}, u_{i,j,m}\rangle\}$, where $v_{i,j} \in N(P)$ is a neighbour of x_i and $u_{i,j,m} \notin N[P]$. See top part of Figure 1. This naming convention will be used in the following.

Lemma 1 (Path Layout Lemma). *Let T be a tree. If there exists a path $P = (x_1, \ldots, x_p)$ in T such that every connected component of $T \backslash N[P]$ has*

LMIM-width $\leq k$ then $lmw(T) \leq k+1$. Moreover, given the layouts for the components we can in linear time compute the layout for T.

Proof. Using the optimal linear orderings of the connected components of $T \backslash N[P]$, we give the below algorithm LinOrd constructing a linear order σ_T on the nodes of T showing that lmwof T is $\leq k+1$. The ordering σ_T starts out empty and the algorithm has an outer loop going through vertices in the path $P = (x_1, \ldots, x_p)$. When arriving at x_i it uses the concatenation operator $\oplus$ to add the path node x_i before looping over all neighbors $v_{i,j}$ of x_i adding the linear orders of each dangling tree from $v_{i,j}$ and then $v_{i,j}$ itself. See Figure 1 for an illustration.

function LinOrd(T: tree, $P = (x_1, \ldots, x_p)$: path, $\{\sigma_{T\langle v_{i,j}, u_{i,j,m}\rangle}\}$: lin-ords)
  $\sigma_T \leftarrow \emptyset$ ▷ The list starts out empty
  for $i \leftarrow 1, p$ **do** ▷ For all nodes on path $(x_1, \ldots, x_p)$
    $\sigma_T \leftarrow \sigma_T \oplus x_i$ ▷ Append path node
    for $j \leftarrow 1, |N(x_i) \backslash P|$ **do** ▷ For all nbs of x_i not on path: $v_{i,j}$
      for $m \leftarrow 1, |N(v_{i,j}) \backslash x_i|$ **do** ▷ For all dangling trees from $v_{i,j}$
        $\sigma_T \leftarrow \sigma_T \oplus \sigma_{T\langle v_{i,j}, u_{i,j,m}\rangle}$ ▷ Append given order of $T\langle v_{i,j}, u_{i,j,m}\rangle$
      $\sigma_T \leftarrow \sigma_T \oplus v_{i,j}$ ▷ Append $v_{i,j}$

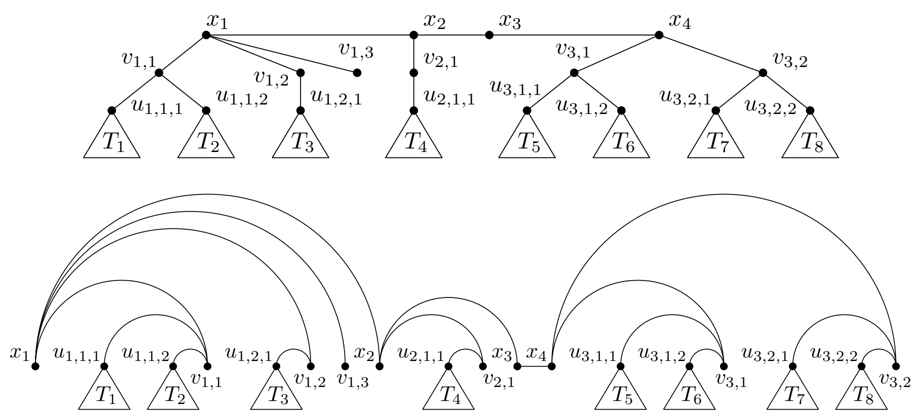

Fig. 1. A tree with a path $P = (x_1, x_2, x_3, x_4)$, with nodes in $N[N[P]]$ and dangling trees featured, and below it the order given by the Path Layout Lemma

Firstly, from the algorithm it should be clear that each node of T is added exactly once to σ_T, that it runs in linear time, and that there is no cut containing two crossing edges from two separate dangling trees. Now we must show that σ_T does not contain cuts with MIM larger than $k+1$. By assumption the layout of each dangling tree has no cut with MIM larger than k, and since these layouts can be found as subsequences of σ_T it follows that then also σ_T has no cut with more than k edges from a single dangling tree $T\langle v_{i,j}, u_{i,j,m}\rangle$. Also, we know that edges from two separate dangling trees cannot both cross the same cut. The only edges of T left to account for, i.e. not belonging to one of the dangling trees, are those with both endpoints in $N[N[P]]$, the nodes at distance at most 2 from a node in P. For every cut of σ_T that contains more than a single crossing edge (x_i, x_{i+1}) there is a unique $x_i \in P$ and a unique $v_{i,j} \in N(x_i)$ such that every edge with both endpoints in $N[N[P]]$ that crosses the cut is incident on either x_i or $v_{i,j}$, and since the edge connecting x_i and $v_{i,j}$ also crosses the cut at most one of these edges can be taken into an induced matching. With these observations in mind, it is clear that $lmw(T) \leq mim(\sigma_T, T) \leq k + 1$.

Definition 2 (k-neighbour and k-component index). *Let x be a node in the tree T and v a neighbour of x. If v has a neighbour $u \neq x$ such that $lmw(T\langle v, u\rangle) \geq k$, then we call v a **k-neighbour** of x. The k-**component index of** x is equal to the number of k-neighbours of x and is denoted $D_T(x, k)$, or shortened to $D(x, k)$.*

Theorem 1 (Classification of LMIM-width of Trees). *For a tree T and $k \geq 1$ we have $lmw(T) \geq k + 1$ if and only if $D(x, k) \geq 3$ for some node x.*

Proof. We first prove the backward direction by contradiction. Thus we assume $D(x, k) \geq 3$ for a node x and there is a linear order σ such that $mim(\sigma, T) \leq k$.

Let v_1, v_2, v_3 be the three k-neighbors of x and T_1, T_2, T_3 the three trees of $T \setminus N[x]$ each of LMIM-width k, with v_i connected to a node of T_i for $i = 1, 2, 3$, that we know must exist by the definition of $D(x, k)$. We know that for each $i = 1, 2, 3$ we have a cut C_i in σ with MIM=k and all k edges of this induced matching coming from the tree T_i. Wlog we assume these three cuts come in the order C_1, C_2, C_3, i.e. with the cut having an induced matching of k edges of T_2 in the middle. Note that in σ all nodes of T_1 must appear before C_2 and all nodes of T_3 after C_2, as otherwise, since T is connected and the distance between T_2 and the two trees T_1 and T_3 is at least two, there would be an extra edge crossing C_2 that would increase MIM of this cut to $k + 1$. It is also clear that v_1 has to be placed before C_2 and v_3 has to be placed after C_2, for the same reason, e.g. the edge between v_1 and a node of T_1 cannot cross C_2 without increasing MIM. But then we are left with the vertex x that cannot be placed neither before C_2 nor after C_2 without increasing MIM of this cut by adding at least one of (v_1, x) or (v_3, x) to the induced matching. We conclude that $D(x, k) \geq 3$ for a node x implies LMIM-width at least $k + 1$.

To prove the forward direction we first show the following partial claim: if $lmw(T) \geq k + 1$ then there exists a node $x \in T$ such that $D(x, k) \geq 3$; or there exists a strict subtree S of T with $lmw(S) \geq k + 1$. We will prove the contrapositive statement, so let us assume that every node in T has $D(x, k) < 3$ and no strict subtree of T has LMIM-width $\geq k + 1$ and show that then $lmw(T) \leq k$. For every node $x \in T$, it must then be true that $D(x, k) \leq 2$ and that $D(x, k + 1) = 0$. The strategy of this proof is to show that there is always a path P in T such that all the connected components in $T \setminus N[P]$ have LMIM-width $\leq k - 1$. When we have shown this, we proceed to use the Path

Layout Lemma, to get that $lmw(T) \leq k$. To prove this, we define the following two sets of vertices:

$$X = \{x | x \in V(T) \ and \ D(x,k) = 2\}, Y = \{y | y \in V(T) \ and \ D(y,k) = 1\}$$

Case 1: $X \neq \emptyset$
If x_i and x_j are in X, then every vertex on the path $P(x_i, \ldots, x_j)$ connecting x_i and x_j must be elements of X, as every node on this path clearly has a dangling tree with LMIM-width k in the direction of x_i and in the direction of x_j. The fact that every pair of vertices in X are connected by a path in X means that X must be a connected subtree of T. Furthermore, this subtree must be a path, otherwise there are three disjoint dangling trees $T\langle v_1, u_1\rangle, T\langle v_2, u_2\rangle, T\langle v_3, u_3\rangle$, each with LMIM-width k, and each hanging from a separate node. But then there is some vertex w such that $T\langle v_1, u_1\rangle, T\langle v_2, u_2\rangle$ and $T\langle v_3, u_3\rangle$ are subtrees of dangling trees from different neighbours of w. But this implies that $D(w,k) \geq 3$, which we assumed were not the case, so this leads to a contradiction. We therefore conclude that all nodes in X must lie on some path $P = (x_1, \ldots, x_p)$. The final part of the argument lies in showing that we can apply the Path Layout Lemma. For some $x_i \in P, i \in \{2, \ldots, p-1\}$, its k-neighbours are x_{i-1} and x_{i+1}. For x_1, these neighbours are x_2 and some $x_0 \notin X$. For x_p, these neighbours are x_{p-1} and some $x_{p+1} \notin X$. x_0 and x_{p+1} may only have one k-neighbour – x_1 and x_p respectively – or else they would be in X. If we make $P' = (x_0, \ldots, x_{p+1})$, we then see that every connected component in $T \backslash N[P']$ must have LMIM-width $\leq k-1$. By the Path Layout Lemma, $lmw(T) \leq k$.

Case 2: $X = \emptyset$, $Y \neq \emptyset$
We construct the path P in a simple greedy manner as follows. We start with $P = (y_1, y_2)$, where y_1 is some arbitrary node in Y, and y_2 its only k-neighbour. Then, if the highest-numbered node in P, call it y_q, has a k-neighbour $y' \notin P$, then we assign y_{q+1} to y', and repeat this process exhaustively. Since we look at finite graphs, we will eventually reach some node y_p such that either $y_p \notin Y$ or y_p's k-neighbour is y_{p-1}. We are then done and have $P = (y_1, \ldots, y_p)$, which must be a path in T, since every node $y_{i+1} \in P$ is a neighbour of y_i and for y_i we only assign maximally one such y_{i+1}. Also, every connected component of $T \backslash N[P]$ must have LMIM-width $\leq k-1$. If not, some node $y_i \in P$ would have a k-neighbour $y' \notin P$, but by the assumption $X = \emptyset$ this is impossible, since then either $i < p$ and y_i has two k-neighbours y' and y_{i+1}, or else $i = p$ and $y_p \in Y$ and y_i has the two k-neighbors y' and y_{i-1} (in case $i = p$ and $y_p \notin Y$ then by definition of Y the node y_i could not have a k-neighbor y'). By the Path Layout Lemma, $lmw(T) \leq k$.

Case 3: $X = \emptyset$, $Y = \emptyset$
If you make $P = (x)$ for some arbitrary $x \in T$, it is obvious that every connected component of $T \backslash N[P]$ has LMIM-width $\leq k-1$. By the Path Layout Lemma, $lmw(T) \leq k$.

We have proven the partial claim that if $lmw(T) \geq k+1$ then there exists a node $x \in T$ such that $D(x,k) \geq 3$; or there exists a strict subtree S of T with $lmw(S) \geq k+1$. To finish the backward direction of the theorem we need to show that if $lmw(T) \geq k+1$ then there exists a node $x \in T$ with $D(x,k) \geq 3$. Assume for a contradiction that there is no node with k-component index at least 3 in T. By the partial claim, there must then exist a strict subtree S with $lmw(S) \geq k+1$. But since we look at finite trees, we know that there in S must exist a minimal subtree $S_0, lmw(S_0) = k+1$ with no strict subtree with LMIM-width $> k$. By the partial claim, S_0 must contain a node x_0 with $D_{S_0}(x_0,k) \geq 3$. But every dangling tree $S_0\langle v,u\rangle$ is a subtree of $T\langle v,u\rangle$, and so if $D_{S_0}(x_0,k) \geq 3$, then $D_T(x_0,k) \geq 3$ contradicting our assumption.

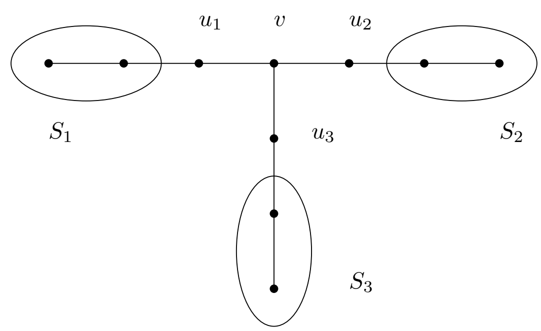

Fig. 2. The smallest tree with LMIM-width 2, having a node v with three 1-neighbors u_1, u_2, u_3 having dangling trees S_1, S_2, S_3, respectively, so that $D(v,1) = 3$

By Theorem 1, every tree with LMIM-width $k \geq 2$ must be at least 3 times bigger than the smallest tree with LMIM-width $k-1$, which implies the following.

Remark 1. The LMIM-width of an n-node tree is $\mathcal{O}(log\ n)$.

3 Rooted trees, *k*-critical nodes and labels

Our algorithm computing LMIM-width will work on a rooted tree, processing it bottom-up. We will choose an arbitrary node r of the tree T and denote by T_r the tree rooted in r. For any node x we denote by $T_r[x]$ the standard *complete subtree* of T_r rooted in x. During the bottom-up processing of T_r we will compute a label for various subtrees. The notion of a k-critical node is crucial for the definition of labels.

Definition 3 (k-critical node). *Let T_r be a rooted tree with $lmw(T_r) = k$. We call a node x in T_r k-**critical** if it has exactly two children v_1 and v_2 that each has at least one child, u_1 and u_2 respectively, such that $lmw(T_r[u_1]) = lmw(T_r[u_2]) = k$. Thus x is k-critical if and only if $lmw(T) = k$ and $D_{T_r[x]}(x,k) = 2$.*

Remark 2. If T_r has LMIM-width k it has at most one k-critical node.

Proof. For a contradiction, let x and x' be two k-critical nodes in T_r. There are then four nodes, v_l, v_r, v'_l, v'_r, the two k-neighbours of x and x' respectively, such that there exist dangling trees $T\langle v_l, u_l\rangle, T\langle v_r, u_r\rangle, T\langle v'_l, u'_l\rangle, T\langle v'_r, u'_r\rangle$ that all have LMIM-width k. If x and x' have a descendant/ancestor relationship in T_r, then assume wlog that x' is a descendant of v_l, and note that $T\langle v_r, u_r\rangle, T\langle v'_l, u'_l\rangle$ and $T\langle v'_r, u'_r\rangle$ are disjoint trees in different neighbours of x', thus $D_{T_r}(x', k) = 3$ and by Theorem 1 T_r should have LMIM-width $k+1$ Otherwise, all the dangling trees are disjoint, thus $D_T(x, k) = D_T(x', k) = 3$ and we arrive at the same conclusion.

Definition 4 (label). *Let rooted tree T_r have $lmw(T_r) = k$. Then* $\mathbf{label(T_r)}$ *consists of a list of decreasing numbers, $(a_1, \ldots, a_p)$, where $a_1 = k$, appended with a string called last_type, which tells us where in the tree an a_p-critical node lies, if it exists at all. If $p = 1$ then the label is simple, otherwise it is complex. The* $\mathbf{label(T_r)}$ *is defined recursively, with type 0 being a base case for singletons and for stars, and with type 4 being the only one defining a complex label.*

- *Type 0: r is a leaf, i.e. T_r is a singleton, then $label(T_r) = (0, t.0)$; or all children of r are leaves, then $label(T_r) = (1, t.0)$*
- *Type 1: No k-critical node in T_r, then $label(T_r) = (k, t.1)$*
- *Type 2: r is the k-critical node in T_r, then $label(T_r) = (k, t.2)$*
- *Type 3: A child of r is k-critical in T_r, then $label(T_r) = (k, t.3)$*
- *Type 4: There is a k-critical node u_k in T_r that is neither r nor a child of r. Let w be the parent of u_k. Then $label(T_r) = k \oplus label(T_r \backslash T_r[w])$*

In type 4 we note that $lmw(T_r \backslash T_r[w]) < k$ since otherwise u_k would have three k-neighbors (two children in the tree and also its parent) and by Theorem 1 we would then have $lmw(T_r) = k+1$. Therefore, all numbers in $label(T_r \backslash T_r[w])$ are smaller than k and a complex label is a list of decreasing numbers followed by $last\_type \in \{t.0, t.1, t.2, t.3\}$. We now give a Proposition that for any node x in T_r will be used to compute $label(T_r[x])$ based on the labels of the subtrees rooted at the children and grand-children of x. The subroutine underlying this Proposition, see the decision tree in Figure 3, will be used when reaching node x in the bottom-up processing of T_r.

Proposition 1. *Let x be a node of T_r with children $Child(x)$, and given $label(T_r[v])$ for all $v \in Child(x)$. We define (and compute) $k = max_{v \in Child(x)} \{lmw(T_r[v])\}$ and $N_k = \{v \in Child(x) \mid lmw(T[v]) = k\}$ and denote by $N_k = \{v_1, \ldots, v_q\}$ and by $l_i = label(T_r[v_i])$. Define (compute) $t_k = D_{T_r[x]}(x, k)$ by noting that $t_k = |\{v_i \in N_k \mid v_i \text{ has child } u_j \text{ with } lmw(T_r[u_j]) = k\}|$. Given this information, we can find $label(T_r[x])$ as follows:*

- ***Case 0:*** *if $|Child(x)| = 0$ then $label(T_r[x]) = (0, t.0)$; else if $k = 0$ then $label(T_r[x]) = (1, t.0)$*
- ***Case 1:*** *Every label in N_k is simple and has last_type equal to $t.1$ or $t.0$, and $t_k \leq 1$. Then, $label(T_r[x]) = (k, t.1)$*

– ***Case 2:*** *Every label in N_k is simple and has last_type equal to $t.1$ or $t.0$, but $t_k = 2$. Then, $label(T_r[x]) = (k, t.2)$*
– ***Case 3:*** *Every label in N_k is simple and has last_type equal to $t.1$ or $t.0$, but $t_k \geq 3$. Then, $label(T_r[x]) = (k+1, t.1)$*
– ***Case 4:*** *$|N_k| \geq 2$ and for some $v_i \in N_k$, either l_i is a complex label, or l_i has last_type equal to either $t.2$ or $t.3$. Then, $label(T_r[x]) = (k+1, t.1)$*
– ***Case 5:*** *$|N_k| = 1$, l_1 is a simple label and l_1 has last_type equal to $t.2$. Then, $label(T_r[x]) = (k, t.3)$*
– ***Case 6:*** *$|N_k| = 1$, l_1 is either complex or has last_type equal to $t.3$, and $k \notin label(T_r[x] \backslash T_r[w])$, where w is the parent of the k-critical node in $T_r[v_1]$. Then, $label(T_r[x]) = k \oplus label(T_r[x] \backslash T_r[w])$*
– ***Case 7:*** *$|N_k| = 1$, l_1 is either complex or has last_type equal to $t.3$, and $k \in label(T_r[x] \backslash T_r[w])$, where w is the parent of the k-critical node in $T_r[v_1]$. Then, $label(T_r[x]) = (k+1, t.1)$*

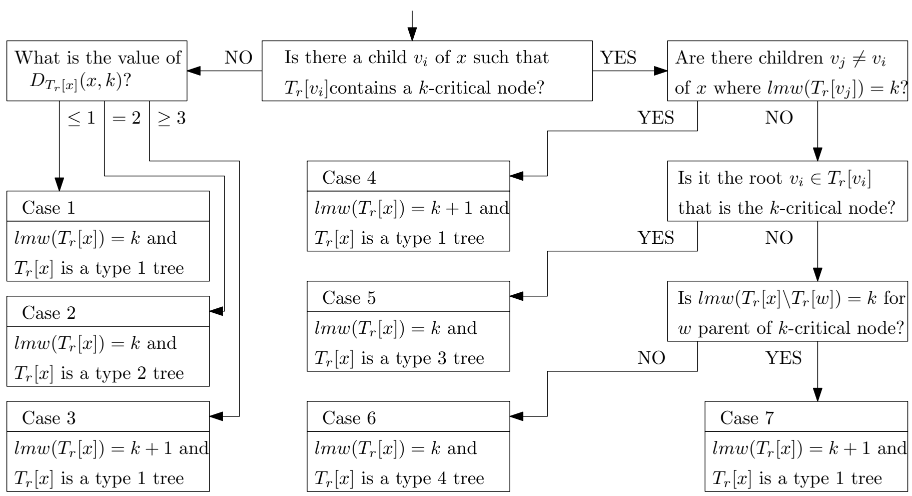

Fig. 3. A decision tree corresponding to the case analysis of Proposition 1

Proof. We show that exactly one case applies to every rooted tree and in each case we assign the label according to Definition 4. First the base case: either x is a leaf or all its children are leaves and we are in Case 0 and the label is assigned according to Def. 4. Otherwise, observe the decision tree in Figure 3. It follows from Def. 4, k, N_k and t_k that cases 1 up to 7 of Prop. 1 corresponds to cases 1 up to 7 in the decision tree - we mention this correspondence in the below - and this proves that exactly one case applies to every rooted tree. The following facts simplify the case analysis: $lmw(T_r[x])$ is equal to either k or $k+1$, and since no subtree rooted in a child of x has LMIM-width $k+1$ there cannot be any $(k+1)$-critical node in $T_r[x]$, therefore if $lmw(T_r[x]) = k+1$, $T_r[x]$ is always a type 1 tree and by Theorem 1 it must contain a node v such that $D_{T_r[x]}(v,k) >= 3$. This node must either be a k-critical node in a rooted subtree of $T_r[x]$, or x itself. We go through the cases 1 to 7 in order.

Note that in Cases 1, 2, and 3 the condition 'Every label in N_k is simple and has *last_type* equal to $t.1$ or $t.0$' means there are no k-critical nodes in any subtree of $T_r[x]$, because every $T_r[v]$ for $v \in Child(x)$ is either of type 1 or has LMIM-width $< k$:

Case 1: By definition of t_k, $D_{T_r[x]}(x,k) \leq 1$. Therefore, $lmw(T_r[x]) = k$, and $T_r[x]$ is a type 1 tree.

Case 2: By definition of t_k, $D_{T_r[x]}(x,k) = 2$, and no other nodes are k-critical, therefore $lmw(T_r[x]) = k$. But now x is k-critical in $T_r[x]$ so $T_r[x]$ is a type 2 tree.

Case 3: By definition of t_k, $D_{T_r[x]}(x,k) = 3$ and $lmw(T_r[x]) = k+1$.

For the remaining Cases 4, 5, 6 and 7, some $T_r[v]$ for $v \in Child(x)$ has LMIM-width k and is of type 2, 3 or 4, so at least one k-critical node exists in some subtree of $T_r[x]$:

Case 4: There is a k-critical node u_k in some $T_r[v_i]$ (not of type 1), and some other v_j has $lmw(T_r[v_j]) = k$ (because $|N_k| \geq 2$). Now observe w the parent of u_k. The dangling tree $T_r[x] \backslash T_r[w]$ is a supertree of $T_r[v_j]$ and thus has LMIM-width $\geq k$. Therefore w is a k-neighbour of u_k and by Theorem 1 $lmw(T_r[x]) = k+1$.

Case 5: x has only one child v with $lmw(T_r[v]) = k$, and v is itself k-critical ($T_r[v]$ is type 2). x cannot be a k-neighbour of v in the unrooted $T_r[x]$, because every dangling tree from x is some $T_r[v_i], v_i \neq v$ of x, which we know has LMIM-width $< k$. Since no other node in T is k-critical, $lmw(T_r[x]) = k$, and since v, a child of x, is k-critical in $T_r[x]$, $T_r[x]$ is a type 3 tree.

Case 6: x has only one child v with $lmw(T_r[v]) = k$, and there is a k-critical node u_k with parent w – neither of which are equal to x – in $T_r[v]$ ($T_r[v]$ is a type 3 or type 4 tree). Moreover, no tree rooted in another child of w, apart from u_k, can have LMIM-width $\geq k$, since this would imply $D_{T_r[v]}(u_k,k) = 3$ and thus $lmw(T_r[v]) > k$; nor can $T_r[x] \backslash T_r[w]$ have LMIM-width $= k$, since then we would have k in $label(T_r[x] \backslash T_r[w])$ disagreeing with the condition of Case 6. Therefore $D_{T_r[x]}(u,k) = 2$, and $lmw(T_r[x]) = k$. $T_r[x]$ is thus a type 4 tree and the label is assigned according to the definition.

Case 7: $T_r[v]$, u_k and w are as described in Case 6. But here, $lmw(T_r[x] \backslash T_r[w]) = k$ (since the condition says that k is in its label), and thus w is a k-neighbour of its child u_k and by Theorem 1 $lmw(T_r[x]) = k+1$.

We conclude that $label(T_r[x])$ has been assigned the correct value in all possible cases.

4 Computing LMIM-width of Trees and Finding a Layout

The subroutine underlying Prop. 1 will be used in a bottom-up algorithm that starts out at the leaves and works its way up to the root, computing labels

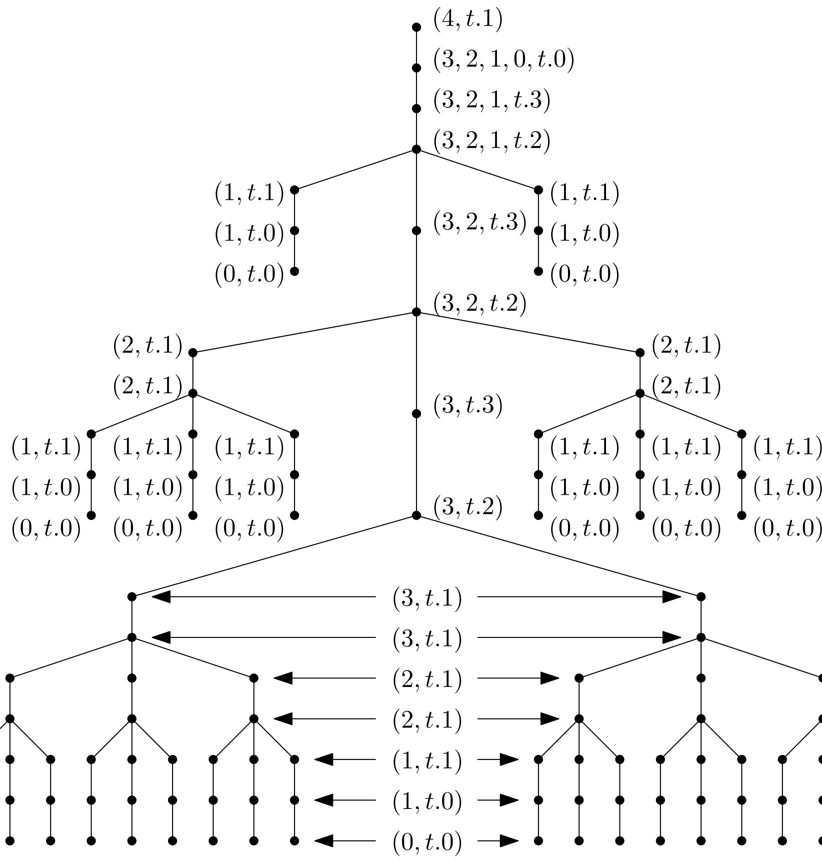

Fig. 4. A rooted tree of LMIM-width 4 with labels of subtrees. We explain the labels $(3, t.2), (3, t.3), (3, 2, t.2)$ assigned to subtrees rooted at the nodes we call a, b, c, with $\mathrm{parent}(a) = b$ and $\mathrm{parent}(b) = c$. The sub-tree rooted at a, with label $(3, t.2)$ has precisely two children that have a child-tree each of LMIM-width 3, hence a is 3-critical and it is a type 2 tree (Case 2 of Prop. 1). The sub-tree rooted at b, labelled $(3, t.3)$, is thus the parent of a 3-critical node, and so it is of type 3 (Case 5 of Prop. 1). The sub-tree rooted at c with label $(3, 2, t.2)$ has maximum LMIM-width of a child-tree being 3, and it has a 3-critical node a which is neither c nor a child of c, so it is of type 4 (Case 6 of Prop. 1); and moreover the subtree $T_r[c] \setminus T_r[a]$ has LMIM-width 2 with node c as 2-critical so it is of type 2 (Case 2 of Prop. 1), and the label of $T_r[c]$ becomes $3 \oplus (2, t.2)$.

of subtrees $T_r[x]$. However, in two cases (Case 6 and 7) we need the label of $T_r[x]\backslash T_r[w]$, which is not a complete subtree rooted in any node of T_r. Note that the label of $T_r[x]\backslash T_r[w]$ is again given by a (recursive) call to Prop. 1 and is then stored as a suffix of the complex label of $T_r[x]$. We will compute these labels by iteratively calling Prop. 1 (substituting the recursion by iteration). We first need to carefully define the subtrees involved when dealing with complex labels.

From the definition of labels it is clear that only type 4 trees lead to a complex label. In that case we have a tree $T_r[x]$ of LMIM-width k and a k-critical node u_k that is neither x nor a child of x, and the recursive definition gives $label(T_r[x]) = k \oplus label(T_r[x] \setminus T_r[w])$ for w the parent of u_k. Unravelling this recursive definition, this means that if $label(T_r[x]) = (a_1, \ldots, a_p, last\_type)$, we can define a list of nodes $(w_1, \ldots, w_{p-1})$ where w_i is the parent of an a_i-critical node in $T_r[x] \setminus (T_r[w_1] \cup \ldots \cup T_r[w_{i-1}])$. We expand this list with $w_p = x$, such that there is one node in $T_r[x]$ corresponding to each number in $label(T_r[x])$, and $T_r[x] \setminus (T_r[w_1] \cup \ldots \cup T_r[w_p]) = \emptyset$.

Now, in the first level of a recursive call to Prop. 1 the role of $T_r[x]$ is taken by $T_r[x] \setminus T_r[w_1]$, and in the next level it is taken by $(T_r[x] \setminus T_r[w_1]) \setminus T_r[w_2]$ etc. The following definition gives a shorthand for denoting these trees.

Definition 5. *Let x be a node in T_r, $label(T_r[x]) = (a_1, a_2, \ldots, a_p, last\_type)$ and the corresponding list of vertices $(w_1, \ldots, w_p)$ is as we describe in the above text. For any non-negative integer s, the tree $\mathbf{T_r[x,s]}$ is the subtree of $T_r[x]$ obtained by removing all trees $T_r[w_i]$ from $T_r[x]$, where $a_i \geq s$. In other words, if q is such that $a_q \geq s > a_{q+1}$, then $T_r[x,s] = T_r[x] \setminus (T_r[w_1] \cup T_r[w_2] \cup \ldots \cup T_r[w_q])$*

Remark 3. Some important properties of $T_r[x,s]$ are the following. Let $T_r[x,s]$, $label(T_r[x,s])$, $(w_1, \ldots, w_p)$ and q as in the definition. Then

1. if $s > a_1$, then $T_r[x,s] = T_r[x]$
2. $label(T_r[x,s]) = (a_{q+1}, \ldots, a_p, last\_type)$
3. $lmw(T_r[x,s]) = a_{q+1} < s$
4. $lmw(T_r[x,s+1]) = s$ *if and only if* $s \in label(T_r[x])$
5. $T_r[x,s+1] \neq T_r[x,s]$ if and only if $s \in label(T_r[x])$

Proof. These follow from the definitions, maybe the last one requires a proof:
Backward direction: Let $s = a_q$ for some $1 \leq q \leq p$. Then $T_r[x,s+1] = T_r[x] \setminus (T_r[w_1] \cup \ldots \cup T_r[w_{q-1}])$ and $T_r[x,s] = T_r[x] \setminus (T_r[w_1] \cup \ldots \cup T_r[w_q])$. These two trees are clearly different.
Forward direction: Let $T_r[x,s] = T_r[x] \setminus (T_r[w_1] \cup \ldots \cup T_r[w_q])$ and $T_r[x,s+1] = T_r[x] \setminus (T_r[w_1] \cup \ldots \cup T_r[w_{q'}])$ with $q' < q$ and $a_{q'} > a_q$ (because numbers in a label are strictly descending). $a_q < s+1$ and $a_q \geq s$, ergo $a_q = s$.

Note that for any s the tree $T_r[x,s]$ is defined only after we know $label(T_r[x])$. In the algorithm, we compute $label(T_r[x])$ by iterating over increasing values of s (until $s > lmw(T_r[x])$ since by Remark 3.1 we then have $T_r[x,s] = T_r[x]$) and we could hope for a loop invariant saying that we have correctly computed $label(T_r[x,s])$. However, $T_r[x,s]$ is only known once we are done. Instead, each iteration of the loop will correctly compute the label of the following subtree called $T_{union}[x,s]$, which is not always equal to $T_r[x]$, but importantly for $s > lmw(T_r[x])$, we will have $T_{union}[x,s] = T_r[x,s] = T_r[x]$.

Definition 6. *Let x be a node in T_r with children $v_1, \ldots, v_d$. $T_{union}[x,s]$ is then equal to the tree induced by x and the union of all $T_r[v_i,s]$ for $1 \leq i \leq d$. More technically, $T_{union}[x,s] = T_r[V']$ where $V' = x \cup V(T_r[v_1,s]) \cup \ldots \cup V(T_r[v_d,s])$.*

Given a tree T, we find its LMIM-width by rooting it in an arbitrary node r, and computing labels by processing T_r bottom-up. The answer is given by the first element of $label(T_r[r])$, which by definition is equal to $lmw(T)$. At a leaf x of T_r we initialize by $label(T_r[x]) \leftarrow (0, t.0)$, and at a node x for which all children are leaves we initialize by $label(T_r[x]) \leftarrow (1, t.0)$, according to Definition 4. When reaching a higher node x we compute label of $T_r[x]$ by calling function MakeLabel(T_r, x).

function MakeLabel$(T_r,\ x)$ ▷ finds $cur\_label = label(T_r[x])$
  $cur\_label \leftarrow (0, t.0)$ ▷ This is $label(T_{union}[x, 0])$
  $\{v_1, \dots, v_d\}$ = children of x
  if $0 \in label(T_r[v_i])$ for some i **then**
    $cur\_label \leftarrow (1, t.0)$ ▷ This is then $label(T_{union}[x, 1])$
  for $s \leftarrow 1, \max_{i=1}^{d}\{$first element of $label(T_r[v_i])\}$ **do**
    $\{l'_1, \dots, l'_d\} = \{label(T_r[v_i, s+1]) \mid 1 \le i \le d\}$
    $N_s = \{v_i \mid 1 \le i \le d,\ s \in l'_i\}$
    $t_s = |\{v_i \mid v_i \in N_s,\ v_i$ has child u_j s.t. $s \in label(T_r[u_j, s+1])\}|$
    if $|N_s| > 0$ **then**
      $case \leftarrow$ the case from Prop. 1 applying to s, $\{l'_1, \dots, l'_d\}$, N_s and t_s
      $cur\_label \leftarrow$ as given by $case$ in Prop. 1 ($s \oplus cur\_label$ if Case 6)

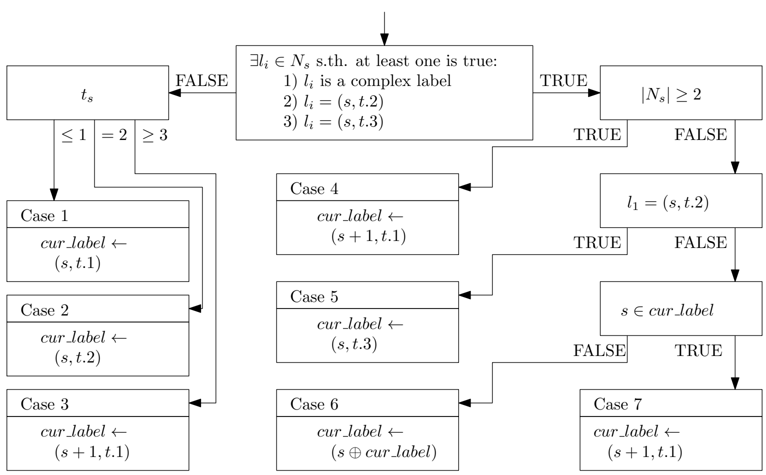

Fig. 5. The same decision tree as shown in Prop. 1, but adapted to MakeLabel

Lemma 2. *Given labels at descendants of node x in T_r,* MakeLabel(T_r, x) *computes $label(T_r[x])$ as the value of cur_label.*

Proof. Assume that x has the children $v_1, \dots, v_d$, and denote their set of labels as $L = \{l_1, \dots, l_d\}$. MakeLabel keeps a variable cur_label that is updated maximally k times in a for loop, where k is the biggest number in any label of children of x. The following claim will suffice to prove the lemma, since for $s > lmw(T_r[x])$, we have $T_{union}[x, s] = T_r[x]$..

Claim: At the end of the s'th iteration of the for loop the value of *cur_label* is equal to $label(T_{union}[x, s + 1])$.

Base case: We have to show that before the first iteration of the loop we have $cur\_label = label(T_{union}[x, 1])$. If some label $l_i \in L$ has 0 as an element then $T_{union}[x, 1]$ is isomorphic to a star with x as the center and v_i as a leaf. By Prop. 1, in this case $label(T_{union}[x, 1]) = (1, t.0)$ and this is what *cur_label* is initialized to. If no $l_i \in L$ has 0 as an element, then by Remark 3.5 $T_{union}[x, 1] = T_{union}[x, 0]$ which by definition is the singleton node x and by Prop. 1 the label of this tree is $(0, t.0)$ and this is what *cur_label* is initialized to.

Induction step: We assume $cur\_label = label(T_{union}[x, s])$ at the start of the s'th iteration of the for loop and show that at the end of the iteration, $cur\_label = label(T_{union}[x, s + 1])$.
The first thing done in the for loop is the computation of $\{l'_i \mid 1 \leq i \leq d,\ l'_i = label(T_r[v_i, s + 1])\}$. By Remark 3.2, $label(T_r[v_i, s + 1]) \subseteq label(T_r[v_i])$ for all i, therefore $l'_1, \ldots, l'_d$ are trivial to compute. The second thing done is to set N_s as the set of all children of x whose labels contain s, and t_s as the number of nodes in N_s that themselves have children whose labels contain s. Let us first look at what happens when $|N_s| = 0$:
By Remark 3.5, for every child v_i of x, $T_r[v_i, s+1] = T_r[v_i, s]$ if $s \notin label(T_r[v_i])$. Therefore, if $|N_s| = 0$, then $T_{union}[x, s+1] = T_{union}[x, s]$, and from the induction assumption, $label(T_{union}[x, s + 1]) = cur\_label$, and indeed when $|N_s| = 0$ then iteration s of the loop does not alter *cur_label*.
Otherwise, we have $|N_s| > 0$ and make a call to the subroutine given by Prop. 1, see the decision tree in Figure 5, to compute $label(T_{union}[x, s + 1])$ and argue first that the variables used in that call correspond to the variables used in Prop. 1 to compute $label(T_r[x])$. The correspondence is given in Table 4. Most of these are just observations: $T_{union}[x, s + 1]$ corresponds to $T_r[x]$

Proposition 1	for loop iteration s	Explanation
$T_r[x], k$	$T_{union}[x, s + 1], s$	Tree needing label, max lmw of children
$T_r[v_1], ..., T_r[v_d]$	$T_r[v_i, s], ..., T_r[v_d, s]$	Subtrees of children
$l_1, ..., l_d, N_k, t_k$	$l'_1, ..., l'_d, N_s, t_s$	Child labels, those with max, root comp. index
$label(T_r[x]\backslash T_r[w])$	*cur_label*	This is also $label(T_{union}[x, s + 1]\backslash T_r[w, s + 1])$

in Prop. 1, and $T_r[v_1, s + 1], \ldots, T_r[v_d, s + 1]$ corresponds to $T_r[v_1], \ldots, T_r[v_d]$. $\{l'_i \mid 1 \leq i \leq d,\ l'_i = label(T_r[v_i, s + 1])\}$ correspond to $\{label(T_r[v]) \mid v \in Child\}$ in Prop. 1. N_s is defined in the algorithm so that it corresponds to N_k in Prop. 1. Since $|N_s| > 0$, some v_i has s in its label l'_i. By Remark 3.3 and 3.4, we can infer that s is the maximum LMIM-width of all $T_r[v_i, s + 1]$, therefore s corresponds

to k in Proposition 1.
It takes a bit more effort to show that t_s computed in iteration s of the for loop corresponds to $t_k = D_{T_r[x]}(x,k)$ in Prop. 1 – meaning we need to show that $t_s = D_{T_{union}[x,s+1]}(x,s)$. Consider v_i, a child of x. In accordance with MakeLabel we say that v_i *contributes to* t_s if $v_i \in N_s$ and v_i has a child u_j with s in its label. We thus need to show that v_i contributes to t_s if and only if v_i is an s-neighbour of x in $T_{union}[x,s+1]$. Observe that by Remark 3.4, $lmw(T_r[v_i,s+1]) = lmw(T_r[u_j,s+1]) = s$ if and only if s is in the labels of both $T_r[v_i]$ and $T_r[u_j]$. If $s \notin label(T_r[u_j,s+1])$, then $lmw(T_r[u_j,s+1]) < s$, and if this is true for all children of v_i, then v_i is not an s-neighbour of x in $T_{union}[x,s+1]$. If $s \notin label(T_r[v_i,s+1])$, then $lmw(T_r[v_i,s+1]) < s$ and no subtree of $T_r[v_i,s+1]$ can have LMIM-width s. However, if $s \in label(T_r[u_j,s+1])$ and $s \in label(T_r[v_i,s+1])$ (this is when v_i contributes to t_s), then $T_r[v_i,s+1] \cap T_r[u_j]$ must be equal to $T_r[u_j,s+1]$ and $T_r[u_j,s+1] \subseteq T_{union}[x,s+1]$, and we conclude that v_i is an s-neighbour of x in $T_{union}[x,s+1]$ if and only if v_i contributes to t_s, so $t_s = D_{T_{union}[x,s+1]}(x,s)$.
Lastly, we show that if $T_{union}[x,s+1]$ is a Case 6 or Case 7 tree – that is, $|N_s| = 1$, and $T_r[v_1,s+1]$ is a type 3 or type 4 tree, with w being the parent of an s-critical node – then the algorithm has $label(T_{union}[x,s+1] \backslash T_r[w,s+1])$ available for computation, indeed that this is the value of *cur_label*. We know, by definition of label and Remark 3.5 that $T_r[v_i,s+1] \backslash T_r[v_i,s] = T_r[w,s+1]$. But since $|N_s| = 1$, for every $j \neq i$, $T_r[v_j,s+1] \backslash T_r[v_j,s] = \emptyset$. Therefore $T_{union}[x,s+1] \backslash T_{union}[x,s] = T_r[w,s+1]$ and $T_{union}[x,s+1] \backslash T_r[w,s+1] = T_{union}[x,s]$. But by the induction assumption, $cur\_label = label(T_{union}[x,s])$. Thus *cur_label* corresponds to $label(T_r[x] \backslash T_r[w])$ in Prop. 1.
We have now argued for all the correspondences in Table 4. By that, we conclude from Prop. 1 and Definition **??** and the inductive assumption that $cur\_label = label(T_{union}[x,s+1])$ at the end of the s'th iteration of the for loop in MakeLabel. It runs for k iterations, where k is equal to the biggest number in any label of the children of x, and *cur_label* is then equal to $label(T_{union}[x,k+1])$. Since $k \geq lmw(T_r[v_i])$ for all i, by definition $T_r[v_i,k+1] = T_r[v_i]$ for all i, and thus $T_{union}[x,k+1] = T_r[x]$. Therefore, when MakeLabel finishes, $cur\_label = label(T_r[x])$.

Theorem 2. *Given any tree T, $lmw(T)$ can be computed in $\mathcal{O}(n\log(n))$-time.*

Proof. We find $lmw(T)$ by bottom-up processing of T_r and returning the first element of $label(T_r)$. After correctly initializating at leaves and nodes whose children are all leaves, we make a call to MakeLabel for each of the remaining nodes. Correctness follows by Lemma 2 and induction on the structure of the rooted tree. For the timing we show that each call runs in $\mathcal{O}(log\ n)$ time. For every integer s from 1 to m, the biggest number in any label of children of x, which is $O(\log n)$ by Remark 1, the algorithm checks how many labels of children of x contain s (to compute N_s), and how many labels of grandchildren of x contain s (to compute t_s). The labels are sorted in descending order, therefore the whole loop goes only once through each of these labels, each of length

$O(\log n)$. Other than this, MakeLabel only does a constant amount of work. Therefore, MakeLabel(T_r, x), if x has a children and b grandchildren, takes time proportional to $O(\log n)(a+b)$. As the sum of the number of children and grandchildren over all nodes of T_r is $O(n)$ we conclude that the total runtime to compute $lmw(T)$ is $\mathcal{O}(n \cdot log\ n)$.

Theorem 3. *A layout of LMIM-width $lmw(T)$ of a tree T can be found in $\mathcal{O}(n \cdot log\ n)$-time.*

Proof. Given T we first run the algorithm computing $lmw(T)$ by finding labels of all nodes and various subtrees. Given T we first run the algorithm computing $lmw(T)$ finding the label of every full rooted subtree in T_r. We give a recursive layout-algorithm that uses these labels in tandem with LinOrd presented in the Path Layout Lemma. We call it on a rooted tree where labels of all subtrees are known. For simplicity we call this rooted tree T_r even though in recursive calls this is not the original root r and tree T. The layout-algorithm goes as follows:
1) Let $lmw(T_r) = k$ and find a path P in T_r such that all trees in $T_r \backslash N[P]$ have LMIM-width $< k$. The path depends on the type of T_r as explained in detail below.
2) Call this layout-algorithm recursively on every rooted tree in $T_r \backslash N[P]$ to obtain linear layouts; to this end, we need the correct label for every node in these trees.
3) Call LinOrd on T_r, P and the layouts provided in step 2.

Every tree in the forest $T \backslash N[P]$ is equal to a dangling tree $T\langle v, u\rangle$, where v is a neighbour of some $x \in P$.
We observe that if $lmw(T) = k$, then by definition $lmw(T\langle v, u\rangle) = k$ if and only if v is a k-neighbour of x. It follows that every tree in $T \backslash N[P]$ has LMIM-width at most $k-1$ if and only if no node in P has a k-neighbour that is not in P. We use this fact to show that for every type of tree we can find a satisfying path in the following way:

Type 0 trees: Choose $P = (r)$. Since $T \backslash N[r] = \emptyset$ in these trees, this must be a satisfying path.
Type 1 trees: These trees contain no k-critical nodes, which by definition means that for any node x in T_r, at most one of its children is a k-neighbour of x. Choose P to start at the root r, and as long as the last node in P has a k-neighbour v, v is appended to P. This set of nodes is obviously a path in T_r. No node in P can possibly have a k-neighbour outside of P, therefore all connected components of $T \backslash N[P]$ have LMIM-width $\leq k-1$. Furthermore, all components of $T - N[P]$ are full rooted sub-trees of T_r and so the labels are already known.
Type 2 trees: In these trees the root r is k-critical. We look at the trees rooted in the two k-neighbours of r, $T_r[v_1]$ and $T_r[v_2]$. By Remark 2 these must both be Type 1 trees, and so we find paths P_1, P_2 in $T_r[v_1]$ and $T_r[v_2]$ respectively, as described above. Gluing these paths together at r we get a satisfying path for T_r, and we still have correct labels for the components $T \backslash N[P]$.

Type 3 trees: In these trees, r has exactly one child v such that $T_r[v]$ is of type 2 and none of its other children have LMIM-width k. We choose P as we did above for $T_r[v]$. r is clearly not a k-neighbour of v, or else $D_T(v, k) = 3$. Every other node in P has all their neighbours in $T_r[v]$. Again, every tree in $T\backslash N[P]$ is a full rooted subtree, and every label is known.

Type 4 trees: In these trees, T_r contains precisely one node $w \neq r$ such that w is the parent of a k-critical node, x. This w is easy to find using the labels, and clearly the tree $T_r[w]$ is a type 3 tree with LMIM-width k. We find a path P that is satisfying in $T_r[w]$ as described above. w is still not a k-neighbour of x, therefore P is a satisfying path. In this case, we have one connected component of $T\backslash N[P]$ that is not a full rooted subtree of T_r, that is $T_r\backslash T_r[w]$. Thus for every ancestor y of w (the blue path in Figure 6) $T_r[y]\backslash T_r[w]$ is not a full rooted subtree either, and we need to update the labels of these trees. However, $T_r[y]\backslash T_r[w]$ is by definition equal to $T_r[y, k]$, whose label is equal to $label(T_r[y])$ without its first number. Thus we quickly find the correct labels to do the recursive call.

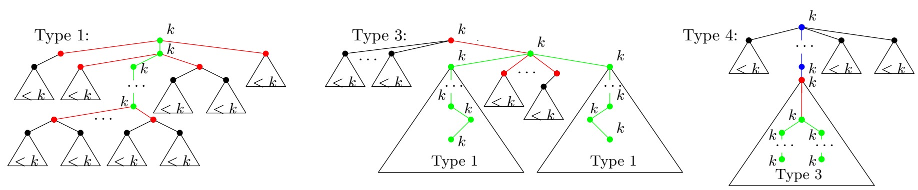

Fig. 6. The path P in green for the proof of Theorem 3.

5 Conclusion

We have given an $O(n \log n)$ algorithm computing the LMIM-width and an optimal layout of an n-node tree. This is the first graph class of LMIM-width larger than 1 having a polynomial-time algorithm computing LMIM-width and thus constitutes an important step towards a better understanding of LMIM-width. Indeed, for the development of FPT algorithms computing tree-width and path-width of general graphs, one could argue that the algorithm of [6] computing optimal path-decompositions of a tree in time $O(n \log n)$ was a stepping stone. The situation is different for MIM-width and LMIM-width, as it is W-hard to compute these parameters [18], but it is similar in the sense that our objective has been to achieve an understanding of how to take a graph and assemble a decomposition of it, in this case a linear one, such that it has cuts of low MIM. To achieve this objective a polynomial-time algorithm for trees has been our main goal.

Bibliography

[1] Adler, I. and Kanté, M. M. (2015). Linear rank-width and linear clique-width of trees. *Theor. Comput. Sci.*, 589:87–98.

[2] Belmonte, R. and Vatshelle, M. (2013). Graph classes with structured neighborhoods and algorithmic applications. *Theor. Comput. Sci.*, 511:54 – 65.

[3] Bergougnoux, B. and Kanté, M. M. (2018). Rank based approach on graphs with structured neighborhood. *CoRR*, abs/1805.11275.

[4] Bui-Xuan, B.-M., Telle, J. A., and Vatshelle, M. (2013). Fast dynamic programming for locally checkable vertex subset and vertex partitioning problems. *Theor. Comput. Sci.*, 511:66 – 76.

[5] Diestel, R. (2012). *Graph Theory, 4th Edition*, volume 173 of *Graduate texts in mathematics*. Springer.

[6] Ellis, J. A., Sudborough, I. H., and Turner, J. S. (1994). The vertex separation and search number of a graph. *Inf. Comput.*, 113(1):50–79.

[7] Fomin, F. V., Golovach, P. A., and Raymond, J.-F. (2018). On the tractability of optimization problems on H-graphs. In *Proc. ESA 2018*, pages 30:1 – 30:14.

[8] Galby, E., Munaro, A., and Ries, B. (2018). Semitotal domination: New hardness results and a polynomial-time algorithm for graphs of bounded mim-width. *CoRR*, abs/1810.06872.

[9] Golovach, P. A., Heggernes, P., Kanté, M. M., Kratsch, D., Sæther, S. H., and Villanger, Y. (2018). Output-polynomial enumeration on graphs of bounded (local) linear mim-width. *Algorithmica*, 80(2):714–741.

[10] Golumbic, M. C. and Rotics, U. (2000). On the clique-width of some perfect graph classes. *Int. J. Found. Comput. Sci.*, 11(3):423–443.

[11] Hlinený, P., Oum, S., Seese, D., and Gottlob, G. (2008). Width parameters beyond tree-width and their applications. *Comput. J.*, 51(3):326–362.

[12] Jaffke, L., Kwon, O., Strømme, T. J. F., and Telle, J. A. (2018a). Generalized distance domination problems and their complexity on graphs of bounded mim-width. In *13th International Symposium on Parameterized and Exact Computation, IPEC 2018, August 20-24, 2018, Helsinki, Finland*, pages 6:1–6:14.

[13] Jaffke, L., Kwon, O., and Telle, J. A. (2017). Polynomial-time algorithms for the longest induced path and induced disjoint paths problems on graphs of bounded mim-width. In *12th International Symposium on Parameterized and Exact Computation, IPEC 2017, September 6-8, 2017, Vienna, Austria*, pages 21:1–21:13.

[14] Jaffke, L., Kwon, O., and Telle, J. A. (2018b). A unified polynomial-time algorithm for feedback vertex set on graphs of bounded mim-width. In *35th Symposium on Theoretical Aspects of Computer Science, STACS 2018, February 28 to March 3, 2018, Caen, France*, pages 42:1–42:14.

[15] Mengel, S. (2018). Lower bounds on the mim-width of some graph classes. *Discrete Applied Mathematics*, 248:28–32.

[16] Möhring, R. H. (1990). Graph problems related to gate matrix layout and pla folding. In *Computational graph theory*, pages 17–51. Springer.

[17] Oum, S. (2017). Rank-width: Algorithmic and structural results. *Discrete Applied Mathematics*, 231:15–24.

[18] Sæther, S. H. and Vatshelle, M. (2016). Hardness of computing width parameters based on branch decompositions over the vertex set. *Theor. Comput. Sci.*, 615:120–125.

[19] Vatshelle, M. (2012). *New width parameters of graphs*. PhD thesis, University of Bergen, Norway.

[20] Yamazaki, K. (2018). Inapproximability of rank, clique, boolean, and maximum induced matching-widths under small set expansion hypothesis. *Algorithms*, 11(11):173.