University of Bergen

Department of Informatics

Algorithms

On the Linear MIM-width of Trees

Student:
Svein Høgemo

Supervisor:
Jan Arne Telle

Master thesis
June 2019

Acknowledgements

My deepest thanks to my supervisor, Jan Arne, for always supporting me on the long winding road of finding out what this thesis was supposed to be about. Additionally, I would like to thank Erlend Raa Vågset for working with me on the paper before I knew how to write, and also for kindly providing some of the illustrations to this thesis. A special thank you to prof. Saket Saurabh, who persuaded me to continue in this field.

Thanks to my friends and family, who are there, even when I do not know I need you. To Ida, you have been my life these last few months.

Contents

Chapter 1

Introduction

Graphs are mathematical objects that represent binary relationships between entities. In practice, graphs are used as a versatile data structure that can model almost anything, from social networks and financial transactions to ecosystems and molecular reactions. Efficient algoritms that work on graphs are therefore highly valuable, as the problems encountered "in the wild" often can be reduced to one of a collection of highly understood problems in graph theory, and efficient solutions to these problems can be applied to a variety of fields. To our detriment, there are many problems that evade efficient solutions. Most people are of the opinion that that the so-called *NP-hard problems* are impossible to solve fast in every instance. Among these problems are several seemingly simple, like Subset Sum: Given a list of integers, does a subset of these integers add to zero? The most naïve solution here would be to simply check the sum of every subset, which takes exponential time in the number of integers. While there certainly exist more clever algorithms to solve Subset Sum, no known algorithm is faster in an *essential* way – they still demand exponential time to give the exact answer in the worst case.

One of the most important tools to overcome the difficulties of NP-hard problems is parameterized algorithms. The runtime of parameterized algorithms is dependent not only on the size of the input, but also on some parameter(s) that somehow measures the difficulty of finding our desired output. Broadly speaking, parameterized algorithms work by selecting a parameter that does one of two things: constrain the input, or constrain the output.

Graph decompositions, the main algorithmic technique concerning this thesis, is a way of constraining input. Broadly speaking, a graph decomposition is a way of dividing a graph into managable "chunks" or subgraphs. A graph algorithm that works on such a decomposition need not consider the graph as a whole; it need only solve the problem on the subgraphs given by the decomposition, and put together the solutions in a suitable way. The *width parameter* associated with each decomposition should then ideally be a measure of how

difficult it is to find the partial solutions and put them together.
Let us take *tree decompositions*, and their related parameter *tree-width* (perhaps the most well-known graph width parameter) as an example: A tree decomposition of a graph G is a tree T where every node represents a subset of the vertices in G, with certain restrictions. The tree-width of T is then the size of the biggest subset that is represented in T. The bigger the tree-width, the longer the algorithm takes to find partial solutions.
In the case of *MIM-width*, the parameter most closely discussed in this thesis, the decomposition of G is a so-called *binary decomposition tree*, a binary tree where every leaf represents one vertex in G and every other node represents the vertices in G represented by leaves in its subtree. For every $S \subseteq V_G$ that is represented by some node in the decomposition tree, the MIM-width gauges the complexity of the bipartite graph crossing the cut $(S, \overline{S})$ as the maximum induced matching number of this graph. The bigger the MIM-width, the longer the algorithm takes to put together partial solutions. Width parameters thus constrain the input in the way that many graphs have no decomposition with a low value of the given parameter.

The decompositions and width parameters discussed here have algorithmic applications: There are numerous results that give polynomial-time algorithms for NP-complete problems when e.g. the MIM-width of the graph is bounded (see e.g. [6], [17]). We can thus directly infer that for every useful parameter, there exists some graph with a value of the parameter that is dependent on the size of the graph, lest $\mathsf{P} = \mathsf{NP}$. That is, there is no single decomposition method that works well on all graphs. That is part of the reason for the plethora of width parameters that have been defined: Different parameters have low values for different types of graphs. For example, tree-width model sparse graphs well, while having big cliques makes the tree-width of that graph big. Other parameters such as *clique-width* or *rank-width* model dense graphs well, while also being bounded on some sparse graphs. MIM-width has the interesting property that it is bounded by a constant on several important graph classes, like interval graphs and cirular-arc graphs, see [3].

In classical complexity, all the NP-complete problems are regarded as equally hard to solve, since they can be reduced to one another in polynomial time (. This is likely not the case when parameters are brought into scope; several complexity classes have been defined that do not seem to collapse into one another. The most important is FPT; that is, all problems that can be solved in time $\mathcal{O}(f(k) \cdot n^c)$ for a parameter k, a computable function f and some constant c. This means that if for example $c = 1$ for some problem, then there exists an algorithm that for any fixed k solves the problem in linear time. k only decides the size of the hidden constant (which may admittedly become huge as k grows). Many problems are solvable in FPT time parameterized by the tree-width of the input graph. In his famous theorem, Courcelle gives a characterization of many problems that can be solved efficiently on tree decompositions; these are the problems that can be stated in monadic second-order logic [7].

The other important complexity class is XP, or "slicewise" polynomially solvable problems. These are all problems solvable in time $n^{\mathcal{O}(f(k))}$ for a parameter k and a computable function f. This means that for every fixed k, the problem can be solved in polynomial time. This is a much weaker advantage though, as one has less control over the exponent. E.g., for vertex subset problems, it basically implies that there is (likely) no way of solving the problem that is *significantly* faster than the naïve approach of trying all potential solutions of size exactly k (which has runtime $\mathcal{O}(n^k)$).
The W-hierarchy is an infinite collection of complexity classes that seeks to explain what parameterized problems are unlikely to be reduced to one another. A strict definition of this complexity hierarchy falls outside the scope of this thesis (the textbook by Downey and Fellows [9], who invented the notion, gives a thorough treatment of the subject), however, a short explanation of the term is in place. W[0] is defined to be equal to FPT, and for every parameterized problem Π in W[N], there exist problems in W[N+1] which cannot be reduced to Π in such a way that the parameter only varies within a constant. There is no hard evidence for the existence of the W-hierarchy, as its existence would imply P $\neq$ NP, but much of the work in parameterized complexity is built on the assumption that it exists.

No algorithms for NP-complete problems parameterized by MIM-width have been devised that run in FPT time. Rather, MIM-width has been utilized in algorithms for W[1]-hard or W[2]-hard problems; these algorithms naturally run in XP time, and subsequently run in polynomial time on graph classes with bounded MIM-width.
Since MIM-width itself is built around the notion of finding a maximum induced matching in bipartite subgraphs, a problem that is NP-complete and W[1]-hard parameterized by the size of the matching, finding decompositions with low MIM-width is also hard. Sæther & Vatshelle have shown [24] that finding a binary decomposition of a graph G of optimal f-width is at least as hard as computing f on G (given some desired qualities of the function f), thus finding a decomposition of MIM-width (say) k must also be at least W[1]-hard parameterized by k. Anyway, we do not know if it even is in XP. It is possible that finding a decomposition of MIM-width k is actually NP-complete for some $k \geq 1$. At the time of writing, it is an open problem to even recognize graphs with linear MIM-width 1 in polynomial time.

This thesis is split into two parts. The first part (sections 1-3) gives an overview on linear MIM-width. This includes:

- a detailed definition of the parameter and its generalization MIM-width in the context of binary decompositions and linear layouts
- the algorithmic use of graph decompositions parameterized by MIM-width
- a recount of some of the main results concerning the modelling power of these parameters, and in what ways they differ from one another

- Some notes on what is still unknown concerning MIM-width

The second part (sections 4-8) gives a new result (developed jointly with Jan Arne Telle and Erlend Raa Vågset in [14]): a polynomial-time algorithm for computing the linear MIM-width of trees, as well as linear layouts of optimal MIM-width. This is to our knowledge the first exact algorithm for linear MIM-width on any graph class with unbounded linear MIM-width that runs in polynomial time. We come to this result through the following steps:

- First, we give a characterization of all trees with linear MIM-width $k+1$ for every $k \geq 1$. This mirrors the characterization of trees with *pathwidth* $k+1$, as described by Ellis, Sudburough & Turner in [10]. A similar characterization also holds for the linear rank-width and the linear clique-width of trees, as shown by Adler and Kanté in [1].

- We then use the characterization to devise a DP algorithm to compute the linear MIM-width of a tree T; the algorithm works on a rooted version of T by assigning a label to every rooted subtree. This algorithm is structurally similar to the algorithm in [10] that computes the path-width of trees, and runs in $\mathcal{O}(n \cdot log(n))$ time.

- Finally, we use the output from the algorithm mentioned above to produce a linear layout of T with optimal MIM-width; this procedure also runs in $\mathcal{O}(n \cdot log(n))$ time.

The above findings have been summarized in a paper [14]; at the time of writing, this paper has been accepted to the 45th International Workshop on Graph-Theoretic Concepts in Computer Science (WG 2019), and a revised version will appear in the proceedings from this conference. The interested reader can find the paper as an appendix to this thesis.

Chapter 2

Preliminaries

2.1 Basic defintions

What follows here is an overview of the basic concepts from graph theory that are used in this thesis. The terminology is meant to follow the standard set by [8] and thus be as easy as possible to follow for the reader. This is by no means a comprehensive dictionary of graph theoretic concepts; for an introduction to the subject the reader should consult [8].

A **graph** G is a pair of sets (V_G, E_G), where E is a family of 2-sets over V_G. V_G is the **vertex set** of G, and E_G is the **edge set** of G. The graphs discussed in this thesis are **undirected**; this means that every edge is an unordered set, i.e. the edge $e = (v, u)$ is equal to $e' = (u, v)$ where u and v are vertices in G. Graphs are often visualized as a set of dots (the vertices) with lines connecting the dots (the edges).
Let v and u be two vertices in the graph G. If (v, u) is an edge in G, then u is said to be the **neighbor** of v in G, and vice versa. Alternatively, v and u are said to be **adjacent** in G. The **neighborhood** $N(v)$ of a vertex v in G is the set of all vertices in G that are neighbors of v, and the closed neighborhood $N[v]$ is defined as $N(v) \cup \{v\}$. Similarly, given a subset of vertices $S \subseteq V_G$, we define $N[S]$ as the union of $N[v]$ for every $v \in S$, and $N(S)$ as $N[S] - S$.
Let G be a graph. A **subgraph** H of G is a graph that has $V_H \subseteq V_G$ and $E_H \subseteq E_G$. The fact that H is a graph implies that E_H is a family over V_H. Given a subset of vertices of G, A, the **induced subgraph** over A, denoted $G[A]$, is a subgraph H such that $V_H = A$ and every edge in E_G with both endpoints in A is also a edge in E_H.
The **complement** of a graph G, denoted $\overline{G}$ is a graph having $V_{\overline{G}} = V_G$, such that for any pair of vertices u, v, $(u, v) \in E_{\overline{G}}$ if and only if $(u, v) \notin E_G$.
A **path** P between two vertices u and v in G is a sequence of vertices $x_1, \ldots, x_p$ such that $x_1 = u$ and $x_p = v$, and for every $1 \leq i < p$, $(x_i, x_{i+1}) \in E_G$. Since G is undirected, P can be traversed both ways. The length of P is $p - 1$.

A graph G is **connected** iff there for every pair of vertices u, v in G exists a path between u and v. A **connected component** in G is a maximal subgraph of G that is connected.
The **distance** between vertices u and v in G, $dist(u, v)$ is equal to the length of the shortest path that connects u and v. $dist(v, v)$ is always 0, and $dist(u, v) = 1$ if and only if they are adjacent. If u and v are in different connected components of G, there exists no path between them and their distance is assumed to be infinite. The distance between two edges uv and xy is the minimum of $(dist(u, x), dist(u, y), dist(v, x), dist(v, y))$.
A **cycle** is a non-trivial path in G that goes from a node back to the same node. If a graph contains no cycles, it is known as a **forest**, and if a forest is connected, it is known as a **tree**. The vertices of a tree are commonly called **nodes**. In this thesis, we usually use the letter T to denote trees.
A **rooted tree** T_r is a tree with one distinguished node r, called the **root**. For every neighbor v of r in T_r, r is called the **parent** of v, while v is a **child** of r. If a node v has parent u, then every node in $N(v) \backslash u$ is a child of v. Every node in T_r except r has exactly one parent.
Given a rooted tree T_r and a node v, we define the **descendants** recursively as all nodes in T_r that are either v itself, or a child of a descendant of v. The **ancestors** of v are likewise defined as all nodes that are either v itself, or the parent of an ancestor of v.
The set of all ancestors of v in T_r induce the path from v to r. The set of all descendants of v in T_r induce the **subtree rooted in** v, written $T[v]$.
A **leaf** in T_r is a node that has no children. In unrooted trees, a leaf is a node with only one neighbor.
A **binary tree** is a rooted tree in which every node has at most two children. In a **full binary tree**, every non-leaf (also called **internal**) node has exactly two children.
Two subclasses of trees observed in this thesis are **caterpillars** and **stars**. A **caterpillar** is a tree T that contains a path P such that every node in T either is on P, or is adjacent to a node on P. A **star** is a caterpillar where P consists of a single node. It applies to all caterpillars that every node in $V_T \backslash P$ must be a leaf.
A graph G whose vertices can be partitioned into two sets X, Y such that every edge in G has one endpoint in X and one endpoint in Y, is called **bipartite**. X and Y are then called the *color classes* of G. Given an arbitrary graph G and two disjoint subsets of vertices in G, A, B, the **induced bipartite subgraph** of G over A, B, denoted $G[A, B]$, is a bipartite subgraph H of G with color classes A and B, such that every edge in E_G with one endpoint in A and one in B is an edge in E_H. In this thesis we mostly consider induced bipartite subgraphs for which $A \cup B = V_G$.

We use the following standard notation from set theory:
Given two sets A and B, $A \cup B$ denotes the **union** of A and B, and $A \cap B$ denotes the **intersection** of A and B. $A \backslash B$ denotes the **subtraction** of B from A.

Given a set S subset of a universe U, $\overline{A}$ denotes the **complement** of A, that is $U \backslash A$. In a graph G, the complement of a vertex set $S \subseteq V_G$ is thus always equal to $V_G \backslash S$.
Given a set S, 2^S denotes the **power set** of S.

2.2 Big-O notation

In algorithmic complexity analysis, big-O notation is a useful tool to simplify functions that we use to describe the time and space demands of an algorithm. It omits the slow-growing components of a function and shows only the fast-growing ones.
Given two functions f, g, $f = \mathcal{O}(g)$ if there exist real and positive constants c and N such that for all $n \geq N$, $f(n) \leq c \cdot g(n)$. It thus embodies the sense of "function f grows at most proportionally as fast as function g". $f = \Omega(g)$ if there exist real and positive constants c and N such that for all $n \geq N$, $f(n) \geq c \cdot g(n)$. $f = \Theta(g)$ if $f = \mathcal{O}(g)$ and $f = \Omega(g)$.
We also have an alternative notation, big-O-star notation, that omits all polynomial factors, this is useful when analyzing exponential algorithms:
Given two functions f, g, $f = \mathcal{O}^*(g)$ if there exist real and positive constants c and N such that for all $n \geq N$, $f(n) \leq n^c \cdot g(n)$.

2.3 Binary Decompositions, Linear Layouts, and f-width

Binary decompositions is a general model for facilitating divide-and-conquer algorithms for problems on graphs. A binary decomposition tree represents a way of recursively partitioning the vertices of a graph, and an algorithm that works on the decomposition tree will traverse it in a bottom-up fashion, for each node finding partial solutions for the set of vertices in the graph corresponding to that node. When it reaches the root, it thus finds a satisfactory solution for the whole graph.

Definition 2.3.1 (Binary Decomposition). Given a graph G with n vertices, a **binary decomposition** of G is a pair (T, δ) where T is a full binary tree and δ is a bijection from V_G to the leaves of T. We let $L(T)$ be the leaves in T. For every node t of T, we then obtain a subset of vertices in G: $V_t^T = \{v \in V_G \ : \ \exists v' \in (V_{T[t]} \cap L(T))[\delta(v) = v']\}$. If $t \in L(T)$, then $V_t^T = \{\delta^{-1}(t)\}$. V_r^T, where r is the root of T, is equal to V_G.

Every full binary tree with exactly n leaves (and by extension exactly $n-1$ internal nodes) can be a binary decomposition of G if given a δ, but not every decomposition is equally well-suited to run our algorithms on. We therefore

introduce a function that somewhat captures the complexity of the binary decomposition tree.
Given a complexity measuring function $f : 2^{V_G} \mapsto \mathbb{N}$, we define the *f-width* of (T, δ) or $fw(G, (T, \delta))$ as $\max_{t \in T} f(V_t^T)$. The f-width of G or $fw(G)$ is then taken to be the minimum f-width over all binary decompositions of G.

Definition 2.3.2 (Linear layout). Given a graph G, $|V_G| = n$, a **linear layout** σ of G is an ordering of its vertices: σ is a bijective function from V_G to $\{1, \ldots, n\}$. For every integer $1 \leq i \leq n$, we obtain a subset of V_G: $V_\sigma^i = \{v \in V_G \ : \ \exists j \in \{1, \ldots, i\}[\sigma(v) = j]\}$.

In the same fashion as with binary decomposition trees, we define the f-width of a linear layout σ or $fw(G, \sigma)$ as $\max_{i=1}^{n} f(V_\sigma^i)$. The *linear f-width* of G or $lfw(G)$ is thus the minimum f-width over all linear layouts of G.

It is worth to notice that every linear layout σ of G has an equivalent binary tree decomposition, called a **caterpillar decomposition**. A **caterpillar decomposition** is a binary tree decomposition (T, δ) where T is also a caterpillar. It should be clear that, given a linear layout σ_G, for every $1 \leq i < n$, there are two corresponding cuts in a corresponding caterpillar decomposition; the first is equal to $G[V_\sigma^i, \overline{V_\sigma^i}]$, and the other, $G[\{v_i\}, V_G \backslash v_i]$, has always maximum induced matching 1 (see Figure 2.1). Since a caterpillar decomposition is a binary decomposition, it follows that the MIM-width of a graph G is no greater than the linear MIM-width of G.

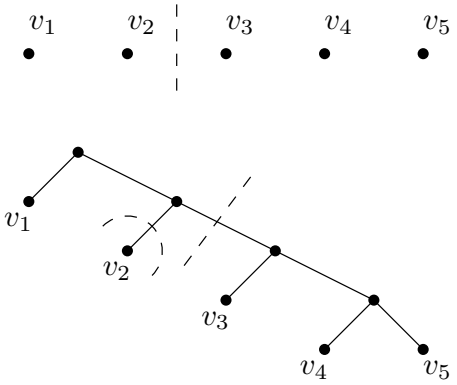

Figure 2.1: A linear layout of 5 vertices and a corresponding caterpillar decomposition, with some cuts indicated

Much of the existing literature talks about linear f-width in terms of caterpillar decompositions. However, in this thesis we will exclusively talk in terms of linear layouts when we introduce linear MIM-width and how to compute it on trees, since they are somewhat easier to reason about.

Chapter 3

MIM-width and linear MIM-width: An Overview

3.1 Induced matchings, MIM-width and linear MIM-width

A **matching** in a graph G is a collection $M \subseteq E_G$ such that no two edges in M have a vertex in common; this can be seen as a pairing up of vertices in G. An **induced matching** in G is a matching M' where no two edges in M' share an edge between them; that is, if A is the set of all vertices in G that are endpoints of edges in M', then $E(G[A]) = M'$. $mim(G)$ denotes the biggest number such that there exists an induced matching in G of that size. It was shown in [25] that finding an induced matching of maximum size is NP-complete, even in bipartite graphs. This contrasts with plain matchings; there is a famous algorithm that given a bipartite graph finds a maximum matching in polynomial time [15].

Definition 3.1.1 (MIM-width). Given a graph G and a binary decomposition (T, δ) of G, we define the **MIM-width** (Maximum Induced Matching-width) of G, or $mw(G)$, as the f-width of G where, for every $A \subseteq V_G$, $f(A) = mim(G[A, \overline{A}])$.

MIM-width is one of three width parameters based on f-width first explored in [27], the others being MM-width (Maximum Matching-width) and Boolean-width. Of these three, MIM-width is the strongest in terms of modelling power, and conversely the weakest in terms of analytical power.

Definition 3.1.2 (Linear MIM-width). Given a graph G, the **linear MIM-width** of G, or $lmw(G)$, is the linear f-width of G where, for every $A \subseteq V_G$, $f(A) = mim(G[A, \overline{A}])$.

The following illustrations show two layouts of a simple graph, one with suboptimal linear MIM-width and one that is optimal.

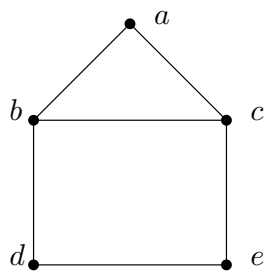

Figure 3.1: The house graph H

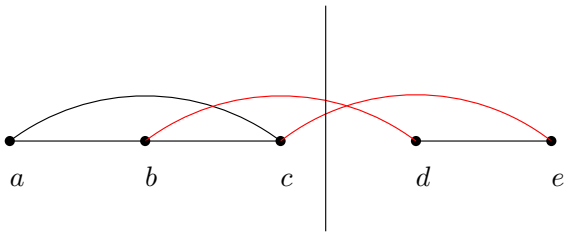

Figure 3.2: This layout of H has MIM-width 2; the indicated cut induces the matching $2K_2$

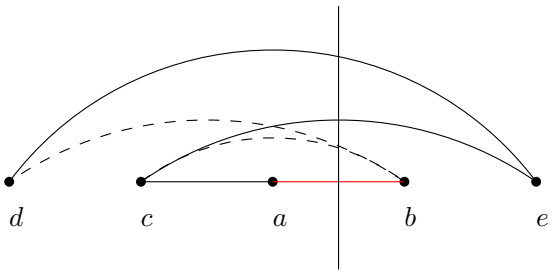

Figure 3.3: In this layout of H, all cuts induce a graph with MIM 1; ergo, $lmw(H) = 1$

3.2 Algorithmic use of MIM-width

(Linear) MIM-width has been utilized in algorithms to solve the general class of locally checkable vertex subset problems, also called (σ, ρ)-domination problems. The framework of (σ, ρ)-domination was introduced by Telle in [26] and includes several interesting problems like Independent Set, Dominating Set, Induced Matching and Perfect Code. We do not explore (σ, ρ)-domination beyond a simple definition in this thesis. However, [6] gives a detailed explanation on how (σ, ρ)-problems can be solved given a decomposition with a bounded number on so-called d-neighborhood equivalence classes. We will below sketch out the connection between MIM-width and d-neighborhood equivalence.
Very briefly explained, a vertex subset problem is a (σ, ρ)-problem if every solution $S \subseteq V_G$ meets some constraint on how many neighbors in S every vertex in G may have, and there is a constant limit d on the number of neighbors one need to check for. More formally, σ and ρ are finite or co-finite sets of non-negative integers that indicate how many neighbors in S every vertex in S and $\overline{S}$ is allowed to have, respectively. For example, S is an independent set if and only if every vertex in S has exactly zero neighbors in S. Therefore, for Independent Set, $\sigma = \{0\}$, $\rho = \mathbb{N}$, and $d = 1$.

More recently, algorithms parameterized by MIM-width have been devised for connectivity and acyclicity problems like Feedback Vertex Set [17], and also (σ, ρ)-problems generalized by distance [16].
In [4], Bergougnoux and Kanté modifies the so-called *rank-based approach* by Bodlaender et al. [5] to work on binary decompositions parameterized by MIM-width or other parameters. This forms a powerful technique for solving connected and/or acyclic versions of (σ, ρ)-problems.

Algorithms that are parameterized by MIM-width take advantage of the notion of d-neighborhood equivalences.

Definition 3.2.1 (d-neighbor equivalence). Given a graph G and a subset of vertices S, two sets of vertices $A, B \subseteq S$ are **d-neighbor equivalent** (written $A \equiv_S^d B$) if and only if for every vertex $v \in \overline{S}$, $\min(d, N(v) \cap A) = \min(d, N(v) \cap B)$.

An equivalent statement is that if $A \equiv_S^d B$, then for every non-negative integer $c < d$, every vertex $v \in \overline{S}$ that has c neighbors in A must also have exactly c neighbors in B. It is not hard to see that $A \equiv_S^1 B$ if and only if they have the same neighborhood in $\overline{S}$.
$\equiv_S^d$ is clearly an equivalence relation on subsets of S.

Definition 3.2.2 (nec_S^d). Given G, S and $\equiv_S^d$ as above, nec_S^d is defined as the number of equivalence classes under $\equiv_S^d$, that is, the maximum number of subsets of S that are pairwise d-neighborhood non-equivalent.

The motivation behind defining d-neighbor equivalence should be clear: When one solves e.g. a locally checkable vertex subset problem, where one for each

vertex need only check if it has up until d neighbors in the solution, partial solutions that are d-neighbor equivalent can be regarded as equivalent, and only the best (minimum/maximum) of these solutions is needed. Assume that we are given a binary decomposition (T, δ) of G such that for every $m \in V_T$, $\text{nec}^d_{V_m^T}$ is polynomial in the size of G. A dynamic programming algorithm using (T, δ) need only store a polynomial amount of partial solutions for each node in the decomposition tree. The algorithm will then run in polynomial time in the size of G.

We show how neighborhood equivalences relate to MIM-width through the next lemma (taken from Belmonte & Vatshelle in [3]), which implies that if G admits a binary decomposition (T, δ) of bounded MIM-width, then for every V_m^T, the number of neighborhoods in $G[V_m^T, \overline{V_m^T}]$ is polynomial in the size of G.

Lemma 3.2.3. *[3] Given a graph G with n vertices and a subset of vertices A, $mim(G[A, \overline{A}]) \leq k$ if and only if for every $S \subseteq A$ there exists $R \subseteq S, |R| \leq k$ such that $N_G(R) \cap \overline{A} = N_G(S) \cap \overline{A}$.*

Proof. We include the proof here for completeness.
Forward direction: We assume that there exists a set $S' \subset A, |S| > k$ such that no proper subset of S' shares its neighborhood in $G[A, \overline{A}]$. This implies that for every vertex $s \in S'$, $N_G(S' \backslash s) \cap \overline{A} \neq N_G(S') \cap \overline{A}$. But this means that every s has a neighbor in $\overline{A}$ that is not a neighbor of any other vertices in S'. We call this vertex $\overline{s}$ and the edge between them e_s. We then collect the set of edges $M = \{e_s : \ s \in S'\}$. But M must be an induced matching, and $|M| = |S'| > k$. This is a contradiction, therefore the forward statement is true.
Backward direction: We assume that there exists a matching M in $G[A, \overline{A}]$, $|M| > k$. Consider the set of vertices $S' = V(G[M]) \cap A$; $|S'| = |M| > k$. But every vertex s in S' has a neighbor in $V(G[M]) \cap \overline{A}$ that it clearly does not share with any other vertex in S', therefore $N_G(S' \backslash s) \cap \overline{A} \neq N_G(S') \cap \overline{A}$. This is a contradiction, therefore the backward statement is true. □

Since every $S \subset A$ has a subset R of size at most k with identical neighborhood in $G[A, \overline{A}]$, we can bound the number of neighborhoods, nec^1_A:

$$nec^1_A \leq \binom{|A|}{k} \leq \binom{n}{k} \leq n^k$$

.

Belmonte & Vatshelle further generalize this result to d-neighborhoods in [3]; this is proven by induction on d.

3.3 Structural Properties of Linear MIM-width

Now, we turn the focus towards the linear parameter. We regard linear MIM-width as interesting for several reasons: A linear ordering of vertices is a simpler structure than a binary decomposition tree and thus easier to reason about, and

at the same time several interesting graph classes have bounded linear MIM-width. The following results are also from [3]:

- Interval graphs, permutation graphs, and convex graphs have linear MIM-width 1
- Circular arc graphs, circular permutation graphs, and trapezoid graphs have linear MIM-width at most 2
- k-trapezoid graphs and k-Dilworth graphs have linear MIM-width at most k
- Circular k-trapezoid graphs and k-polygon graphs have linear MIM-width at most $2k$.

We will repeat the proof that interval graphs have linear MIM-width 1. Most of the other proofs follow the same tactic.

Firstly, we observe from Lemma 3.2.3 that a bipartite graph $G[X, Y]$ has maximum induced matching number 1 if and only if there for every nonempty set $S \subseteq X$ exists one vertex $s \in S$ such that $N(s) = N(S)$. We call these graphs **bipartite chains**:

Definition 3.3.1 (Bipartite chain). A bipartite chain is a bipartite graph $G[X, Y]$ with nested neighborhoods; that is, assuming $|X| = p$ we can find an ordering of the vertices in X, $(x_1, \ldots, x_p)$, such that $N(x_{i+1}) \subseteq N(x_i)$ for all $1 \leq i < p$.

From this, we see that a graph G has linear MIM-width 1 if and only if there exists a linear layout of G such that every cut on the layout induces a bipartite chain.

Lemma 3.3.2. *[3] Given any interval graph G, one can in polynomial time find a linear layout of G of MIM-width 1.*

Proof. It is known that given an interval graph G, $V_G = \{v_1, \ldots, v_n\}$, one can find an intersection model consisting of intervals $\mathcal{I} = \{(s_1, e_1), \ldots, (s_n, e_n)\}$ in linear time (for example, the method given by [13] is applicable to both recognizing and describing interval graphs in linear time). See [3] for a definition of intersection models and interval graphs).
We sort the intervals by ascending start points; this forms the linear layout σ of the vertices:

$$\sigma(v_i) < \sigma(v_j) \Leftrightarrow s_i < s_j$$

Note that this can be done in polynomial time.
Furthermore, we sort the intervals by descending end points:

$$\rho(v_i) < \rho(v_j) \Leftrightarrow e_i > e_j$$

Now, consider some cut of the linear layout $(V_i^\sigma, \overline{V_i^\sigma})$. Every vertex v_a in V_i^σ must have a crossing edge to every vertex v_b in $\overline{V_i^\sigma}$ obeying $s_b < e_a$. It is now

clear that if $e_j > e_m$ for two vertices $v_j, v_m \in V_i^\sigma$, that is, $\rho(v_j) < \rho(v_m)$, then v_j must be adjacent to every vertex in $\overline{V_i^\sigma}$ that v_m is adjacent to. Therefore, ρ shows a nesting of the neighborhoods in V_i^σ; $G[V_i^\sigma, \overline{V_i^\sigma}]$ must thus be a bipartite chain. From this, we deduce that σ is a linear layout of V_G with MIM-width 1. □

Additionally, we show that the complement of a graph with MIM-width 1 also has MIM-width 1; that is, the class of all graphs of MIM-width 1 is self-complementary. This result gives some more graph classes that have MIM-width 1.

Observation 3.3.3. *Given a bipartite chain G with color classes X, Y, its bipartite complement $\overline{G}[X, Y]$ is also a bipartite chain.*

Proof. By definition of bipartite chain, if we assume $|X| = p$, there is an ordering of the vertices in X $\chi = (x_1, \ldots, x_p)$ such that $N_G(x_{i+1}) \subseteq N_G(x_i)$ for every $1 \leq i < p$. But for every $x \in X$, $N_{\overline{G}[X,Y]}(x) = Y \backslash N_G(x)$. By this, it is clear that $N_{\overline{G}[X,Y]}(x_i) \subseteq N_{\overline{G}[X,Y]}(x_{i+1})$, and the reverse of χ is thus an ordering of X that satisfies the chain property of $\overline{G}[X, Y]$. □

From this, we derive the following two corollaries.

Corollary 3.3.4. *Given a graph G and a binary decomposition (T, δ) of G with MIM-width 1, (T, δ) is a binary decomposition of $\overline{G}$ which also has MIM-width 1.*

Corollary 3.3.5. *If a graph G has MIM-width 1, then $\overline{G}$ also has MIM-width 1. Specifically, co-interval graphs and co-convex graphs have linear MIM-width 1 (permutation graphs are self-complementary). Co-cycle-free graphs have MIM-width 1.*

An interesting aspect of MIM-width is how it is perfectly suited to represent the graphs on which (σ, ρ)-domination problems are solvable in polynomial time. For specific graph classes, these results are not necessarily new; already in 1995, it was shown that in interval graphs, $\exists(\sigma, \rho)$-domination problems are solvable in polynomial time for all (σ, ρ) (where σ and ρ, as before, are assumed to be finite or co-finite), while on chordal graphs, the generalized problem is NP-complete [20]. The importance of MIM-width rests upon the fact that it provides a framework around which we can reason and generalize these graphs. There is a similar parameter SIM-width, short for subset induced matching-width, that captures chordal graphs: all chordal graphs have SIM-width 1 ([18], in this paper SIM-width is also introduced). While a decomposition of a graph G of MIM-width k requires that every cut $G[X, \overline{X}]$ induced by the decomposition (T, δ) has the property $mim(G[X, \overline{X}]) \leq k$, SIM-width k requires that for every such $G[X, \overline{X}]$, every induced matching in G contains at most k edges from $E_{G[X,\overline{X}]}$. There are many problems that can be solved in polynomial time on chordal graphs (see e.g. [11] or [19]); so far, no significant algorithmic results

have been proven on graph decompositions of bounded SIM-width. It is possible that SIM-width is too strong a parameter to be useful as an algorithmic tool. Moreover, to our knowledge no interesting parameter has been found that captures all perfect graphs. On these graphs, there are still several problems that are solvable in polynomial time, eg. CLIQUE or INDEPENDENT SET (see e.g. Chapter 9 of [12]). This could be a potentially interesting line of research, but since there is no unified way of decomposing perfect graphs, it could prove hard to find.

We accordingly know that linear MIM-width is bounded on several graph classes on which few other interesting parameters are bounded. This stems mainly from how much complexity is allowed in the induced bipartite graphs arising from cuts in the layout. Linear MIM-width k allows all cuts inducing bipartite graphs whose neighborhoods are contained within subsets of size k. This allows graph classes with a loose sense of linear structure to have bounded linear MIM-width. Contrast this with the related parameter MM-width: The bipartite graphs with maximum matching at most k are exactly the bipartite graphs with no subgraph consisting of $k+1$ disjoint stars. Linear MM-width 1 therefore implies a linear layout such that all cuts induce stars. This is clearly much more restrictive than linear MIM-width 1.

Note that there are graph classes that have bounded MIM-width, but unbounded linear MIM-width. For example, [27] shows that the MIM-width of a graph G is no greater than the tree-width of G. Forests, which have tree-width 1, consequently have MIM-width 1. On the contrary, it is known that trees (or graphs with bounded tree-width) have unbounded linear MIM-width. In fact, we show in Chapter 5 that trees have linear MIM-width at most $\log_3(n)$ and that we can construct trees with linear MIM-width $\Theta(log(n))$. Loosely stated, the graphs that seem to have bounded MIM-width but unbounded linear MIM-width, are graphs that branch nicely into disjoint subgraphs with bounded MIM-width, but branch in more than two directions and thus cannot be bounded within a linear layout. Trees are the simplest example of this. The observation below gives a concrete formulation of this structure.

Observation 3.3.6. *The tree $T_{3,3,3}$ has linear MIM-width 2.*

Proof. We can easily see that the linear layout of $T_{3,3,3}$ provided in the illustration has MIM-width 2, thus $lmw(T_{3,3,3}) \leq 2$. To prove that $lmw(T_{3,3,3}) = 2$, we then need to show that no layout of linear MIM-width 1 exists. Assume to the contrary that there is a layout $\sigma_{T_{3,3,3}}$ (from here simply denoted σ) with MIM-width 1. Consider the edges y_1z_1, y_2z_2, y_3z_3. These three edges form an

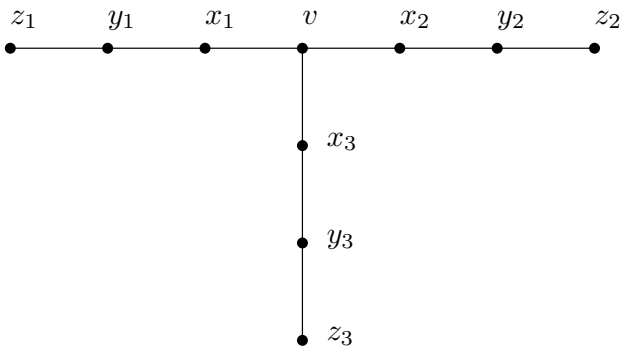

Figure 3.4: The tree $T_{3,3,3}$ with every node labelled

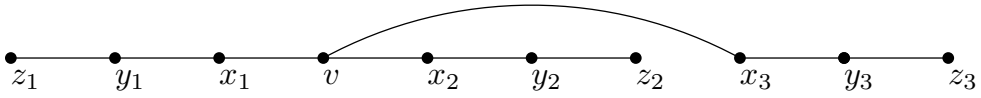

Figure 3.5: A linear layout of $T_{3,3,3}$ with MIM-width 2

induced matching, and can therefore not overlap in σ. Therefore we WLOG assume that $\sigma(y_2) < \sigma(z_2)$ and that

$$y_1 < y_2, z_1 < z_2$$

$$z_2 < y_3, z_2 < z_3$$

There must then exist a subset of nodes V_σ^i such that y_1, z_1 and y_2 are in V_σ^i and z_2, y_3 and z_3 are not. But the cut $T_{3,3,3}[V_\sigma^i, \overline{V_\sigma^i}]$ must also contain an edge on the path from y_1 to y_3. Every edge on this path can be taken into an induced matching together with y_2z_2, but this is a contradiction of the assumption that σ has MIM-width 1. We then conclude that $lmw(T_{3,3,3}) = 2$. □

This observation naturally generalizes to trees of linear MIM-width $k+1$ for any $k \geq 1$; this is proven in Lemma 4.3.2 later in the thesis.

3.4 Hardness of Computing (Linear) MIM-width

It is important to note that for every graph class described by Belmonte & Vatshelle in [3] whose linear MIM-width is not bounded by 1, but by some other number, we do not know of any fast algorithm that finds an optimal linear ordering in every instance. For example, in the paper they describe an algorithm that, given a circular arc graph, finds a linear layout with MIM-width at most 2, but there is no guarantee that this layout is optimal.

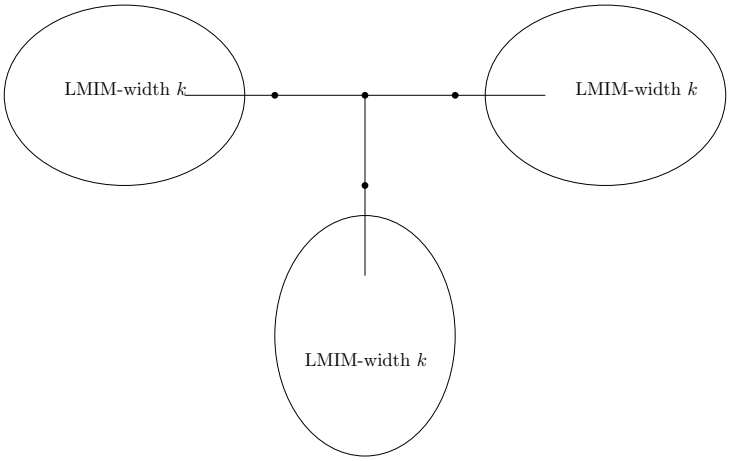

Figure 3.6: Generalization of the observation above: a tree with linear MIM-width $k + 1$

It is currently unknown whether there exists an XP-time algorithm to decide whether an arbitrary graph admits a linear ordering of MIM-width at most k. In fact, we do not yet know whether we can recognize graphs of linear MIM-width 1 in polynomial time, or if this is an NP-complete problem. This is perhaps the most interesting unsolved problem regarding MIM-width: Deciding whether a given graph admits a binary decomposition or linear layout of MIM-width 1 in polynomial time.
It has been shown that finding a single cut $(S, \overline{S})$ with $min(|S|, |\overline{S}|) \geq 2$ such that $G[S, \overline{S}]$ is a bipartite chain is NP-complete ([23], in the paper it is called a $2K_2$-free decomposition). Knowing this, we can imagine that the possibility of a polynomial-time algorithm for the problem of finding a linear layout such that all cuts induce such graphs, is dimished. On the other hand, graphs of linear MIM-width are admittedly much more structured than graphs with a single $2K_2$-free decomposition; this structure may be used to an advantage.

The fastest exact algorithm currently known that computes MIM-width for arbitrary graphs, is a fairly standard dynamic programming algorithm that finds an optimal decomposition or layout for all subsets of vertices [22]. It runs in $\mathcal{O}^*((2t)^n)$ time, where n is the number of vertices in the graph, and t is the smallest constant for which we can find a maximum induced matching in a bipartite graph in $\mathcal{O}^*(t^n)$ time. To our knowledge, this constant is currently 1.3752 [28]; ergo, the fastest exact algorithm runs in $\mathcal{O}^*(2.751^n)$ time.

Theorem 3.4.1. *Given a graph G, $|V_G| = n$, we can compute the linear MIM-width of G and a linear layout of optimal MIM-width in $\mathcal{O}^*(2.751^n)$ time.*

Proof. We give an algorithm for finding a linear layout of a graph $G, |V_G| = n$

that is optimal with regards to MIM-width (provided by Olav Røthe Bakken [2]). For every $S \subseteq V_G$, we define the number $\mu(S)$ as the minimum of $min_{i=1}^{|S|}(mim(G[V_{\sigma_S}^i, \overline{V_{\sigma_S}^i}]))$ over every ordering σ_S of the vertices in S. We furthermore define the vertex $\omega(S)$ as last vertex in the optimal layout σ_S giving $\mu(S)$:

- Let $\mathcal{V} = (V_0, V_1, \ldots, V_n)$ such that for every i, V_i is the set of all subsets of V_G of size exactly i. Clearly, $\bigcup_{i=0}^{n} V_i = 2^{V_G}$.
- Set $\mu(\emptyset) = 0$ and $\mu(S) = \infty$ for all $S \neq \emptyset$.
- For every i from 1 to n, do:
  - For every $S \in V_i$, do:
    - For every $v \in S$, compute $\mu(S, v) = \max(\mu(S \backslash \{v\}), mim(G[S, \overline{S}]))$. If $\mu(S, v) < \mu(S)$, set $\mu(S)$ equal to $\mu(S, v)$ and set $\omega(S)$ equal to v.
- $lmw(G) = \mu(V_G)$
- We find the linear layout in backwards order by picking the last vertex from each relevant set:

$$v_n = \omega(V_G)$$
$$v_{n-1} = \omega(V_G \backslash \{v_n\})$$
$$\ldots$$
$$v_1 = \omega(V_G \backslash \{v_2, \ldots, v_n\})$$

Assuming that $mim(G[A, \overline{A}])$ takes $\mathcal{O}(t^n)$ time to find for some t for every $A \subseteq V_G$, this algorithm clearly runs in $\mathcal{O}((2t)^n \cdot n^2)$ time. □

Chapter 4

Characterization of Trees with given Linear MIM-width

The rest of this thesis, except Chapters 5 and 9, can be considered a reworking of the paper written by Høgemo, Telle and Vågset [14].

4.1 Some Comments on Trees

As we have seen, the linear MIM-width remains hard to find for arbitrary graphs. To learn more about this parameter and what types of graphs on which it is bounded, one possible strategy is to look at graphs with well-understood structure, and try to draw conclusions on the linear MIM-width of these graphs. To this end trees are excellent candidates: Having a single path between any pair of nodes means that we are able to control how many edges are crossing the given cuts. At the same time, they have unbounded linear MIM-width, and optimal linear layouts of trees are therefore not trivial to compute.
The downside of focusing on trees is obviously that the practical value of an algorithm that computes linear MIM-width on trees is minimal. There are already hundreds of fast algorithms that are specifically designed to solve problems on trees. Additionally, trees have MIM-width 1, and there are no big advantages in parameterizing by linear MIM-width rather than simply MIM-width – as far as we know, the two parameters have basically the same algorithmic power.
Our results are thus most interesting from a theoretical point of view; we provide the first fast exact algorithm for computing linear MIM-width on any graph class for which the parameter is unbounded. It is our hope that this result and the classification of trees with a given linear MIM-width will shed new light on and contribute to a deeper understanding of the properties of this parameter.

4.2 Dangling Trees and Linear Layouts Derived from Paths

Our first step towards an efficient algorithm for the linear MIM-width of trees, is finding an invariant that holds for all trees with linear MIM-width $k+1$, for any $k \geq 1$. This invariant states that $lmw(T) \geq k+1$ if and only if there is a node x in T with the property that there are three trees in the forest $T \backslash N[x]$ with linear MIM-width $\geq k$, each adjacent to a distinct neighbor of x in T. We dedicate most of this section to prove that this characterization holds for every tree, but to be able to talk about this property in simpler terms, we find the following definition useful:

Definition 4.2.1 (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.

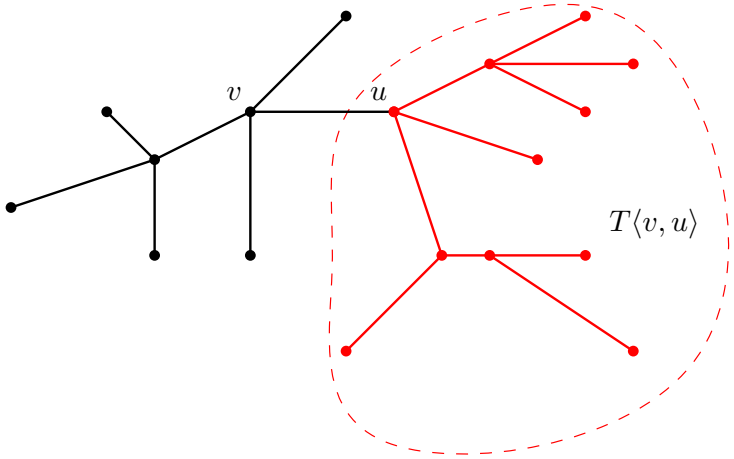

Figure 4.1: The dangling tree $T\langle v, u\rangle$

Given a node $x \in V_T$ with neighbors $\{v_1, \ldots, v_d\}$, $T \backslash N[x]$ is a forest of dangling trees $\{T\langle v_i, u_{i,j}\rangle\}$ in T, where for every $1 \leq j < |N(v_i)|$, $u_{i,j} \neq x$ is a neighbor 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 neighbor of x_i and $u_{i,j,m} \notin N[P]$. See top part of Figure 4.2. We uphold this naming convention during the following sections.

The following lemma is important both to prove Theorem 4.3.4 and as a subroutine to the layout-algorithm outlined in Chapter 8.

Lemma 4.2.2 (Path Layout Lemma). *Let T be a tree and $P = (x_1, \ldots, x_p)$ a path in T. If every connected component of $T \backslash N[P]$ has linear MIM-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. Assuming that we are given optimal linear layouts of the connected components of $T \backslash N[P]$, we give the below algorithm LinOrd constructing a linear layout σ_T on the nodes of T showing that lmwof T is $\leq k+1$. We construct σ_T as a list of nodes; it starts out empty, and nodes are added to the list in the order specified by the algorithm. This list clearly can represent a linear layout by the property that $\sigma_T(u) < \sigma_T(v)$ if and only if u sits to the left of v in the list.

---

function LinOrd(T: tree, $(x_1, \ldots, x_p)$: 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 $m \leftarrow 1, |N(v_{i,j}) \backslash v_i|$ **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 $T\langle v_{i,j}, u_{i,j,m}\rangle$
      end for
      $\sigma_T \leftarrow \sigma_T \oplus v_{i,j}$ ▷ Append $v_{i,j}$
    end for
  end for
  return σ_T
end function

To prove Lemma 4.2.2, we must show that σ_T as returned by the LinOrd procedure is a linear layout of the nodes in T with MIM-width at most $k+1$.

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, 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.

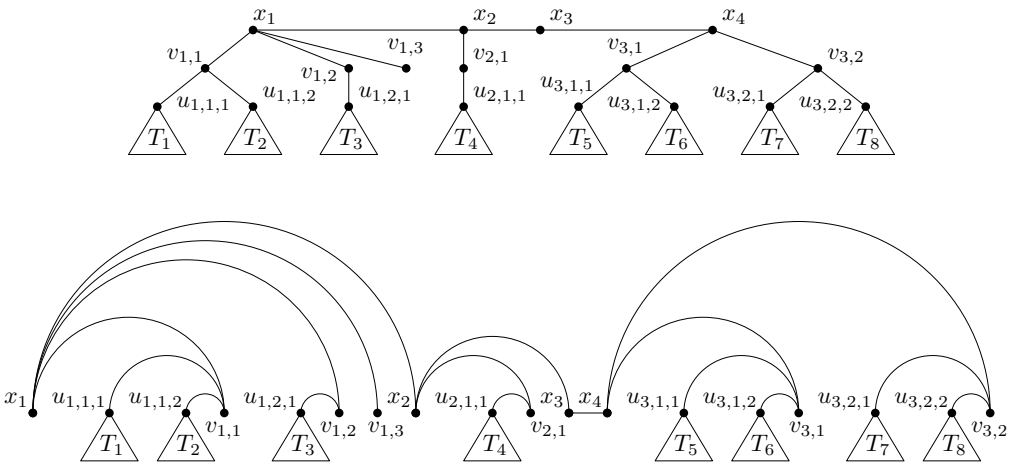

Figure 4.2: A tree T with a path P and trees in $T \backslash N[P]$, assume that $\forall i$: $lmw(T_i) \leq k$; the ordering of the nodes given by LinOrd shows that $lmw(T) \leq k+1$

With these observations in mind, it is clear that $lmw(T) \leq mw(T, \sigma_T) \leq k+1$. □

4.3 Theorem of Characterization

To prove the main theorem of the section, we prove two helping lemmas,, roughly equaling the "if" and "only if" directions of the theorem.
First, we give two definitions that help us talk about the structural properties of trees we are interested in.

Definition 4.3.1 (k-neighbor and k-component index). Let x be a node in the tree T and v a neighbor of x. If v has a neighbor $u \neq x$ such that $lmw(T\langle v, u\rangle) \geq k$, then we call v a **k-neighbor** of x. The **k-component index of** x is equal to the number of k-neighbors of x and is denoted $D_T(x,k)$, or shortened to $D(x,k)$ if T is obvious from context.

Lemma 4.3.2 (Sufficiency). *Let T be a tree. If there is a node x in T such that $D_T(x,k) \geq 3$ for some $k \geq 1$, then $lmw(T) \geq k+1$.*

Proof. We prove this lemma by contradiction. Let T and x fulfill the property given by Lemma 4.3.2 By assumption, x has three neighbors, v_1, v_2, v_3, each of which having a neighbor $u_1, u_2, u_3 \neq x$ such that for $1 \leq i \leq 3$, the tree $T_i = T\langle v_i, u_i\rangle$ has linear MIM-width at least k. Since there exist subtrees in T

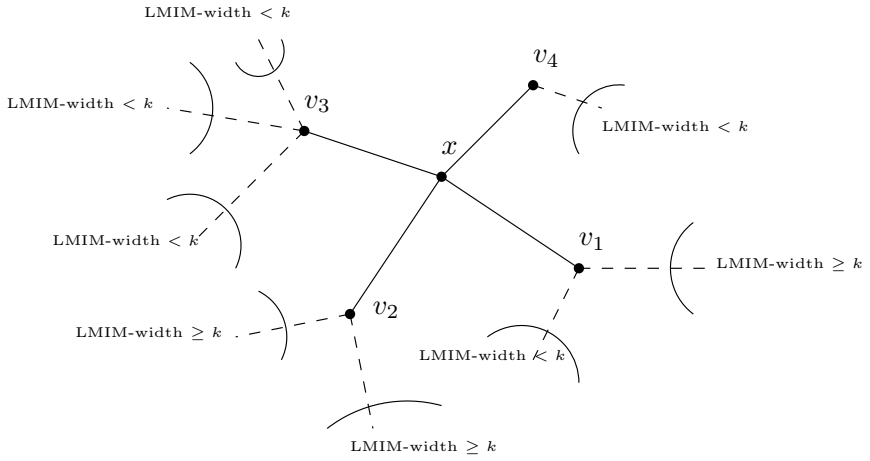

Figure 4.3: The node x has two k-neighbors, v_1 and v_2

with linear MIM-width at least k, it follows that $lmw(T) \geq k$.

Assume that there exists a linear layout σ_T such that $mw(T, \sigma_T) = k$. We know that for each $i \in \{1, 2, 3\}$ we have a cut $C_i = V_\sigma^{c_i}$ in σ_T with $mim(G[C_i, \overline{C_i}]) = k$ and all k edges of this induced matching contained in the tree T_i. WLOG we assume these three cuts come in the order C_1, C_2, C_3, that is, $c_1 < c_2 < c_3$. 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_1 and T_3 are connected and the distance between T_2 and the two trees is at least two, there would be an extra edge crossing C_2 that would otherwise 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. But then the vertex x cannot be placed before C_2 or after C_2 without increasing MIM of this cut by adding at least one of the edges v_1x or v_3x to the induced matching. By contradiction, we conclude that $D(x, k) \geq 3$ for a node x implies LMIM-width at least $k + 1$. □

Lemma 4.3.3 (Necessity). *Let T be a tree. If $lmw(T) \geq k + 1$ for some $k \geq 1$; then there exists a node x in T such that $D_T(x, k) \geq 3$; or there exists a $S \subsetneq T$ with $lmw(S) \geq k + 1$.*

Proof. We want to prove the contrapositive statement, namely that if for every node $x \in T$, $D(x, k) \leq 2$ and $D(x, k + 1) = 0$, then $lmw(T) \leq k$. The strategy of this proof is to show that we can always find a path P in T such that all the connected components in $T \backslash N[P]$ have linear MIM-width $\leq k-1$. We can then infer by the Path Layout Lemma that $lmw(T) \leq k$.

Let T be a tree, for every $x \in V_T$, $D(x, k) \leq 2$ and $D(x, k + 1) = 0$. We define two disjoint subsets of V_T as follows:

$$X_1 = \{x \in V_T \mid D(x, k) = 1\}$$

$$X_2 = \{x \in V_T \mid D(x,k) = 2\}$$

We need to prove the lemma for three distinct cases.

Case 1: $X_2 \neq \emptyset$
If x_i and x_j are in X_2, then every vertex on the path $P(x_i, \ldots, x_j)$ connecting x_i and x_j must be elements of X_2, as every node on this path clearly has a dangling tree with linear MIM-width k in the direction of x_i and in the direction of x_j. The fact that every pair of vertices in X_2 are connected by a path contained within $T[X_2]$ implies that X_2 induces a connected subtree of T.
Furthermore, this subtree must be a path, otherwise there would exist 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 linear MIM-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 neighbors of w. This implies that $D(w,k) \geq 3$, which we assumed were not the case, thus this leads to a contradiction. We therefore conclude that all nodes in X_2 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-neighbors are x_{i-1} and x_{i+1}. For x_1, these neighbors are x_2 and some $x_0 \notin X_2$. For x_p, these neighbors are x_{p-1} and some $x_{p+1} \notin X_2$. x_0 and x_{p+1} may only have one k-neighbor – x_1 and x_p respectively – or else they would be in X_2. If we make $P' = (x_0, \ldots, x_{p+1})$, we then see that every connected component in $T \backslash N[P']$ must have linear MIM-width $\leq k-1$. By the Path Layout Lemma, $lmw(T) \leq k$.

Case 2: $X_2 = \emptyset$, $X_1 \neq \emptyset$
We construct the path P in a greedy manner as follows. We start with $P = (x_1, x_2)$, where x_1 is some arbitrary node in X_1, and x_2 its only k-neighbor. Then, if the highest-numbered node in P, call it x_q, has a k-neighbor $x' \notin P$, then we assign x_{q+1} to x', and repeat this process exhaustively.
Considering only finite graphs, we will eventually reach some node x_p such that either $x_p \notin X_1$ or x_p's k-neighbor is x_{p-1}. We now have $P = (x_1, \ldots, x_p)$. This is a path in T, since every node $x_{i+1} \in P$ is a neighbor of x_i and for x_i we only assign maximally one such x_{i+1}.
Also, every connected component of $T \backslash N[P]$ must have linear MIM-width $\leq k-1$. If not, some node $x_i \in P$ would have a k-neighbor $x' \notin P$, but under the assumption $X_2 = \emptyset$ this is impossible. Otherwise, either $i < p$ and x_i has two k-neighbors x' and x_{i+1}, or else $i = p$ and $x_p \in X_1$ and x_i has the two k-neighbors x' and x_{i-1} (in case $i = p$ and $x_p \notin X_1$, then by definition of X_1 the node x_i could not have a k-neighbor x'). By the Path Layout Lemma, $lmw(T) \leq k$.

Case 3: $X = \emptyset$, $Y = \emptyset$
Take $P = (x)$ for some arbitrary $x \in V_T$. It holds trivially that every connected component of $T \backslash N[P]$ has linear MIM-width $\leq k-1$. By the Path Layout Lemma, $lmw(T) \leq k$. □

Theorem 4.3.4. *Let T be a tree. $lmw(T) \geq k+1$ if and only if there is a node $x \in V_T$ such that $D(x,k) \geq 3$.*

Proof. The backward direction of this statement follows directly from Lemma 4.3.2.
We prove the forward direction by contradiction. Let T be a tree, $lmw(T) \geq k+1$ for some $k \geq 1$, and assume that for all $x \in V_T$, $D_T(x,k) \leq 2$. By Lemma 4.3.3, we have that there must exist a proper subtree $S \subsetneq T$ with $lmw(S) \geq k+1$. But since we work on finite graphs, we know that there must exist a tree $S_0 \subseteq S$ with linear MIM-width $k+1$, with no proper subtree of linear MIM-width $\geq k+1$. By Lemma 4.3.3, there must be a node $x_0 \in S_0$ with $D_{S_0}(x_0,k) \geq 3$. But every dangling tree $S_0\langle v,u\rangle$ must be a subtree of $T\langle v,u\rangle$, and therefore $D_T(x_0,k) \geq 3$. By this, the theorem must be true. □

Chapter 5

Limits on the Linear MIM-width of Trees

5.1 Trees of Minimum Cardinality with a given Linear MIM-width

From Theorem 4.3.4 alone, we can already say much about the linear MIM-width of trees. We know that every tree with linear MIM-width at least 2 contains a node such that the removal of its neighborhood induces a forest containing at least three trees with linear MIM-width at most one lower. This implies that every tree with linear MIM-width (say) $k+1$ is at least three times as big as some tree with linear MIM-width k. We can thus derive the following remark.

Remark 5.1.1. For any tree T, $|V_T| = \Omega(3^{lmw(T)})$, and conversely, $lmw(T) = \mathcal{O}(log(|V_T|)$.

Looking at Theorem 4.3.4, we can furthermore devise a method of explicitly constructing a tree of minimum cardinality with a given linear MIM-width.
We define for every $k \geq 1$ the class $\mathcal{T}_{min}(k)$ as a set consisting of every tree T that has the following property: $lmw(T) = k$, and every tree $T', |V_{T'}| < |V_T|$ has linear MIM-width strictly less than k. The only tree in $\mathcal{T}_{min}(1)$ obviously is K_2 (the pair), the smallest graph that contains an edge. The only tree in $\mathcal{T}_{min}(2)$ is $T_{3,3,3}$.
For every k, we can construct a tree in $\mathcal{T}_{min}(k+1)$ by taking three trees T_1, T_2, T_3, $T_i \in \mathcal{T}_{min}(k)$, gluing one extra node v_i onto one node in T_i and connect v_1, v_2 and v_3 to a central node x. It is not hard to see that the resulting tree must be in $\mathcal{T}_{min}(k+1)$. Given a tree $T \in \mathcal{T}_{min}(k)$, we can thereby calculate the size $min_{lmw}(k) = |T|$ by means of the following two equivalent formulas:

$$k = 1 \Rightarrow min_{lmw}(k) = 2;\ k > 1 \Rightarrow min_{lmw}(k) = 3 \cdot min_{lmw}(k-1) + 4$$

$$min_{lmw}(k) = 4 \cdot 3^{k-1} - 2$$

k	$min_{lmw}(k)$
1	2
2	10
3	34
4	106
5	322
6	970
...	...

Table 5.1: The minimum number of nodes in a tree of linear MIM-width k

5.2 Relation between Linear MIM-width and Path-width

In [10], Ellis et al. show a similar characterization of trees with a given path-width as the one we have given for linear MIM-width, only simpler to explain: A tree T has path-width $\geq k+1$ for some $k \geq 1$ if and only if there is some node $x \in V_T$ such that there are three trees in $T \backslash \{x\}$ with path-width $\geq k$. We observe the following:

Lemma 5.2.1. *For any tree T, $lmw(T) \leq pw(T) \leq 2 \cdot lmw(T)$. This bound is tight.*

Proof. First we prove $lmw(T) \leq pw(T)$; we do this by induction over the path-width of T. For the base case, it is trivial to see that if a tree has path-width 1, then it also must have linear MIM-width 1. For the induction step, we assume that the claim holds for every $1 \leq j \leq k$, and show that it holds for $k+1$. Let T be a tree with path-width $k+1$. Then for every node $x \in V_T$ there are at most two trees in $T \backslash \{x\}$ with pathwidth strictly larger than k. But then x can have maximally two $k+1$-neighbors, and $lmw(T) \leq k+1$.
Next we prove $pw(T) \leq 2 \cdot lmw(T)$; this is also proven by induction over the path-width of T. For the base case, it is trivial to see that if $pw(T)$ is equal to 1 or 2, then the $lmw(T)$ is at least 1. For the induction step, assume that the claim holds for every $1 \leq j \leq 2k$, and show that it holds for $2k+2$. let T be a tree with path-width $2k+2$. Then there exists a node $x \in V_T$ such that there are three trees T_1, T_2, T_3 in $T \backslash \{x\}$ with path-width k, which again implies that there exist nodes in these three trees such that in the forests obtained by removing these nodes, three trees have pathwidth $2k$. But in every T_i, at least one of these trees must be a subtree of some dangling tree (that is a subtree of T_i) with linear MIM-width at least k. Ergo, x has at least 3 k-neighbors, and $lmw(T) \geq k$.
For the tightness, it is sufficient to show the two trees pictured below. □

The following corollary follows from [1]:

Corollary 5.2.2. *For any tree T, $lmw(T) \leq lrw(T) \leq 2 \cdot lmw(T)$, where $lrw(T)$ denotes the* linear rank-width *of T.*

Chapter 6

Rooted Trees, k-critical Nodes and Labelling

6.1 Specifics on the Linear MIM-width of Rooted Trees

Until now we have only looked at unrooted trees. However, in our algorithm we will work on rooted trees. In this section we introduce the bookkeeping devices of k-critical nodes and labelling of subtrees (both are adaptations of definitions in [10]). These will be vital both in our computation of the linear MIM-width of trees, and in the computation of a linear layout of optimal MIM-width.

Definition 6.1.1 (k-critical node). Let T_r be a tree that has $lmw(T) = k$, and x a node in T_r. x is **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 $lmw(T_r[u_1]) = lmw(T_r[u_2]) = k$. In other words, x is k-critical in T_r if and only if $lmw(T) = k$ and $D_{T_r[x]}(x, k) = 2$.

Remark 6.1.2. Let T_r be a rooted tree with $lmw(T) = k$, then there can be at most one k-critical node in T_r.

Proof. We prove this by contradiction. Let x and x' be two k-critical nodes in T_r. This means that there are four nodes, v_l, v_r, v'_l, v'_r, the children of x and x' respectively that are k-neighbors, which implies that there exist 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 linear MIM-width k. If x and x' have a descendant/ancestor relationship in T_r, we can WLOG assume that x' is a ldescendant of v_l. Now, $T\langle v_r, u_r\rangle, T\langle v'_l, u'_l\rangle$ and $T\langle v'_r, u'_r\rangle$ are disjoint trees in different neighbors of x', thus $D_T(x', k) = 3$.
If x and x' do not have a relationship in T_r, all the dangling trees are disjoint, thus $D_T(x, k) = D_T(x', k) = 3$.

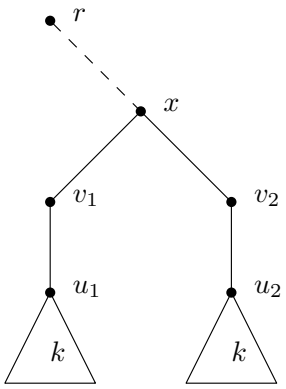

Figure 6.1: Assuming that $lmw(T_r) = k$, the node x is k-critical in T_r. Note that the illustrated tree might be a subtree of $T_r[x]$

In both of these cases, at least one of x, x' have 3 k-neighbors, and so by Theorem 4.3.4 T should have $linearMIM - width\ k+1$. By contradiction, the above remark is true. □

Definition 6.1.3 (label). Let T_r be a rooted tree having $lmw(T_r) = k$. 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 $p = 1$, $label(T_r)$ is a **simple label**, otherwise it is a **complex label**. We define five **types** of trees that all have distinct labels; Type 0 is a base case for singletons and stars, while Type 4 is 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, i.e. T_r is a star, then $label(T_r) = (1, t.0)$
- Type 1: There is 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 $label(T_r) = k \oplus label(T_r \backslash T_r[w])$

Regarding Type 4 trees, 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 w) and by Theorem 4.3.4 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 $last\_type \in \{t.0, t.1, t.2, t.3\}$.

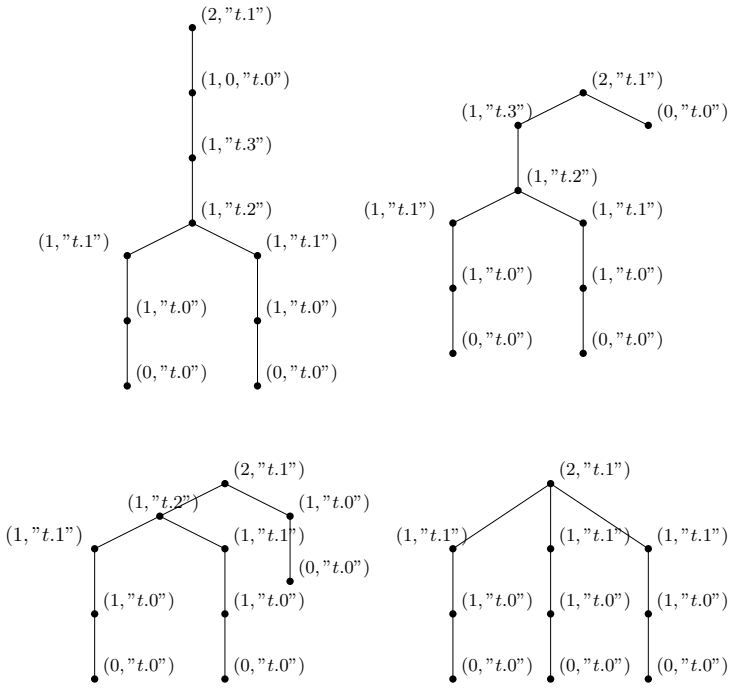

Figure 6.2: An overview of every way of rooting the tree $T_{3,3,3}$, not counting isomorphisms. To every node is a label attached, denoting the label of the subtree rooted in that node. Note that the first number in the label of the root is 2, since $lmw(T_{3,3,3}) = 2$

Why do we need such a complicated structure to keep track of the linear MIM-width of decreasing subtrees of T_r? When we want to find the linear MIM-width of a tree, we want to find the highest number k for which there exists a node x with $D(x, k) \geq 3$. But in a rooted tree, one of x's k-neighbors might be its parent (call it w), and the dangling tree with linear MIM-width k might be $T \backslash T_r[w]$. We will show in later sections that using a structure which encodes exactly this information in a recursive manner – i.e. labels – we can devise an efficient algorithm that computes both the linear MIM-width of a tree and a linear layout of its nodes that is optimal with respect to linear MIM-width .

6.2 Deducing Linear MIM-width from Labels of Subtrees

We now give a Proposition that for any node x in T_r can be used to compute $label(T_r[x])$ based on the labels of the subtrees rooted at the children and grand-

children of x. This proposition takes the form of a procedure that will be called as a subroutine to the algorithm in Section 7.2, see also the decision tree in Figure 6.3.

Proposition 6.2.1. *Let x be a node of T_r and let $Child(x)$ be the set of children of x in T_r, and assume we are 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 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, $label(T_r[x]) = (k + 1, t.1)$*

Proof. We show that exactly one case applies to every rooted tree, and in each case we assign the label according to Definition 6.1.3. 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. 6.1.3. Otherwise, observe the decision tree in Figure

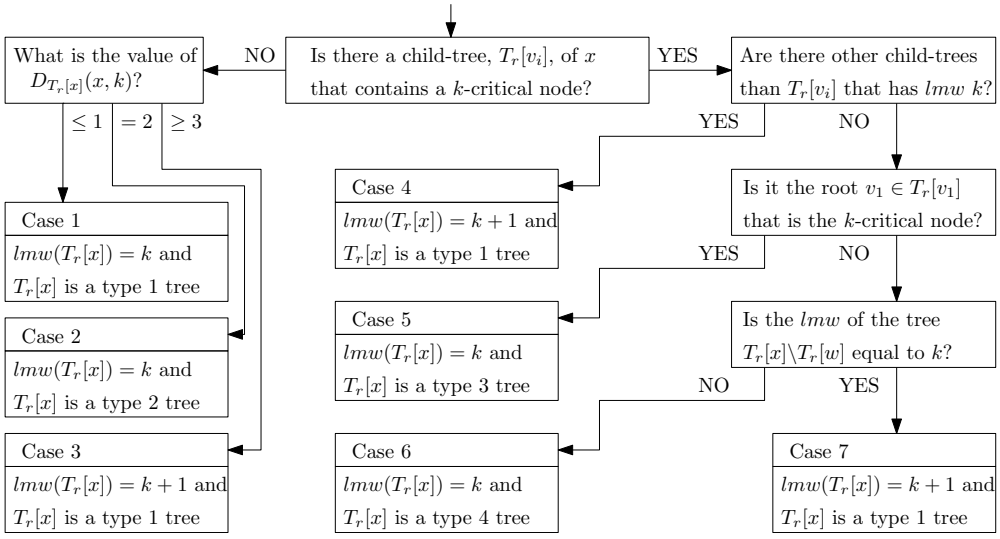

Figure 6.3: The information outlined in Proposition 6.2.1, in the form of a flowchart

6.3. It follows from Def. 6.1.3, and of k, N_k and t_k that cases 1 up to 7 of Prop. 6.2.1 corresponds to cases 1 up to 7 in the decision tree - we explain this correspondence case for case below - and this proves that exactly one case applies to every rooted tree.
The following facts simplify the case analysis:
$lmw(T_r[x])$ must be 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. Also, by Theorem 4.3.4 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 linear MIM-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 linear MIM-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 linear MIM-width $\geq k$. Therefore w is a k-neighbor of u_k and by Theorem 4.3.4 $lmw(T_r[x]) = k + 1$.
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-neighbor 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 linear MIM-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 linear MIM-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 linear MIM-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-neighbor of its child u_k and by Theorem 4.3.4 $lmw(T_r[x]) = k + 1$.
We conclude that $label(T_r[x])$ has been assigned the correct value in all possible cases. □

It should be clear that when Proposition 6.2.1 is run as a subroutine, it runs in linear time in the combined size of the labels provided, as it only needs to recognize and count how many labels of each type have a maximum number in them, and eventually concatenate two labels. This is an important consideration towards the runtime of the algorithm we come up with in the next section.

Chapter 7

Efficient Computation of the Linear MIM-width of Trees

7.1 Subtrees with Respect to Labels

We have shown in Prop. 6.2.1 that in a rooted tree T_r, one can, for each node $x \in V_{T_r}$, compute a label that implies the LMIM-width of $T_r[x]$ based on the labels of subtrees rooted in its descendants. This strongly calls for a recursive or bottom-up algorithm that starts out at the leaves and works its way up to the root, computing labels of bigger and bigger subtrees. However, in two cases (Cases 6 and 7) we need the label of $T_r[x] \backslash T_r[w]$, which is not a rooted subtree of any node of T_r. This complicates the algorithm somewhat; fortunately, labels are defined in such a way that this value is easily computable, as this information is stored as a suffix in complex labels. This is also what justifies the need for a complicated data structure like the label.

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 $label(T_r[x]) = k \oplus label(T_r[x] \backslash T_r[w])$ 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 $T_r[x] \backslash (T_r[w_1] \cup \ldots \cup T_r[w_{i-1}])$. 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 $T_r[x] \backslash (T_r[w_1] \cup \ldots \cup T_r[w_p]) = \emptyset$.

When analyzing how our algorithm is going to work, we need some new infrastructure on which we are able to talk about on exactly what subtrees

the linear MIM-width is measured in complex labels. We give the following definitions:

Definition 7.1.1. 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, for every s there exists a $0 \leq q \leq p$ such that $a_q \geq s > a_{q+1}$, and

$$T_r[x, s] = T_r[x] \backslash (T_r[w_1] \cup T_r[w_2] \cup \ldots \cup T_r[w_q])$$

Remark 7.1.2. 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 remarks follow from the definitions, but the last one a proof:
Backward direction: Let $s = a_q$ for some $1 \leq q \leq p$. Then $T_r[x, s+1] = T_r[x] \backslash (T_r[w_1] \cup \ldots \cup T_r[w_{q-1}])$ and $T_r[x, s] = T_r[x] \backslash (T_r[w_1] \cup \ldots \cup T_r[w_q])$. These two trees are clearly different.
Forward direction: Let $T_r[x, s] = T_r[x] \backslash (T_r[w_1] \cup \ldots \cup T_r[w_q])$ and $T_r[x, s+1] = T_r[x] \backslash (T_r[w_1] \cup \ldots \cup T_r[w_{q'}])$ 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 $a_q = s$. □

From these definitions, the idea behind the algorithm we want is clear: It works its way bottom up on the rooted tree T_r, and for every node $x \in V_{T_r}$ we iterate over every integer from 1 $lmw(T_r[x])$, incrementally constructing $label(T_r[x])$ in the process. To this end, we would like that after the s'th step we had obtained $label(T_r[x, s+1])$, but unfortunately this is not possible, since we do not yet know $label(T_r[x])$. What we instead show is that after the s'th step we obtain the label of a tree we call $T_{union}[x, s+1]$, which has the property that if $lmw(T_r[x]) = k$, then $T_{union}[x, k] = T_r[x]$. We are then still able to use Proposition 6.2.1 to compute the linear MIM-width of T.

Definition 7.1.3. 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 $V' = x \cup V(T_r[v_1, s]) \cup \ldots \cup V(T_r[v_d, s])$.

7.2 An Algorithm for Computing Linear MIM-width of Trees

We are now ready to state the algorithm. Given a tree T, we find its linear MIM-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 $label(T_r)$, 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 6.1.3. When reaching a higher node x we compute label of $T_r[x]$ by calling the procedure MakeLabel(T_r, x).

---

function MakeLabel(T_r, x) ▷ finds $cur\_label = label(T_r[x])$
  $cur\_label \leftarrow (0, t.0)$ ▷ This is $label(T_{union}[x, 0])$
  $\{v_1, \ldots, v_d\}$ = children of x
  if $0 \in label(T_r[v_i])$ for some i **then**
    $cur\_label \leftarrow (1, t.0)$ ▷ This is then $label(T_{union}[x, 1])$
  end if
  for $s \leftarrow 1, \max_{i=1}^{d}\{$first element of $label(T_r[v_i])\}$ **do**
    $\{l'_1, \ldots, l'_d\} = \{label(T_r[v_i, s+1]) \mid 1 \leq i \leq d\}$
    $N_s = \{v_i \mid 1 \leq i \leq d,\ s \in l'_i\}$
    $t_s = |\{v_i \mid v_i \in N_s,\ v_i$ 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. 6.2.1 applying to s, $\{l'_1, \ldots, l'_d\}$, N_s and t_s
      $cur\_label \leftarrow$ as given by $case$ in Prop. 6.2.1 ($s \oplus cur\_label$ if Case 6)
    end if
  end for
end function

Lemma 7.2.1. *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 $v_1, \ldots, v_d$, and denote their set of labels as $L = \{l_1, \ldots, l_d\}$. 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 $label(T_{union}[x, s+1])$.

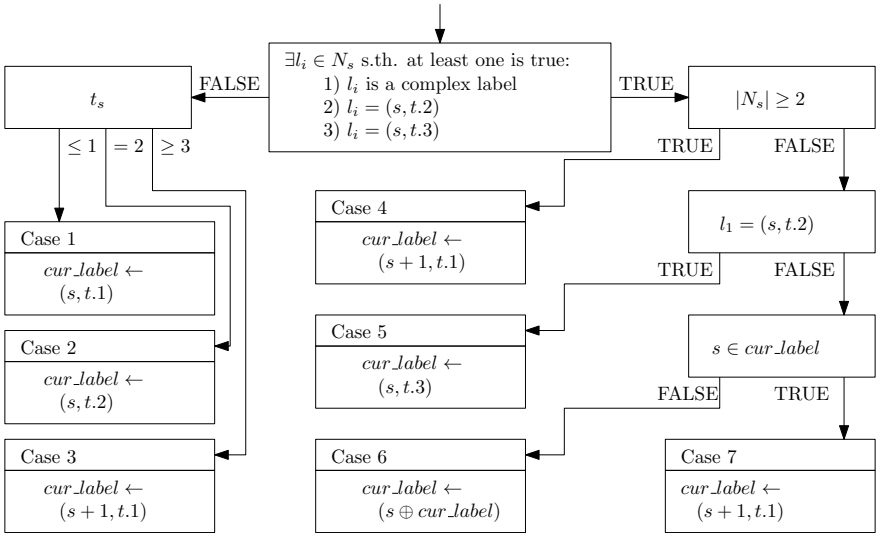

Figure 7.1: The flowchart from Figure 6.3, but adapted to the notation used in MakeLabel

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. 6.2.1, 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 7.1.2.5 $T_{union}[x, 1] = T_{union}[x, 0]$ which by definition is the singleton node x and by Prop. 6.2.1 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 7.1.2.2, $label(T_r[v_i, s + 1]) \subseteq label(T_r[v_i])$ 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 7.1.2.5, 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. 6.2.1, see the decision tree in Figure 7.1, to compute $label(T_{union}[x, s + 1])$ and

Proposition 6.2.1	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 $label(T_{union}[x, s+1] \backslash T_r[w, s+1])$

argue first that the variables used in that call correspond to the variables used in Prop. 6.2.1 to compute $label(T_r[x])$. The correspondence is given in Table 7.2.

Most of these are just observations: $T_{union}[x, s+1]$ corresponds to $T_r[x]$ in Prop. 6.2.1, 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. 6.2.1. N_s is defined in the algorithm so that it corresponds to N_k in Prop. 6.2.1. Since $|N_s| > 0$, some v_i has s in its label l'_i. By Remark 7.1.2.3 and 7.1.2.4, we can infer that s is the maximum linear MIM-width of all $T_r[v_i, s+1]$, therefore s corresponds to k in Proposition 6.2.1.

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. 6.2.1 – 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-neighbor of x in $T_{union}[x, s+1]$. Observe that by Remark 7.1.2.4, $lmw(T_r[v_i, s+1]) = lmw(T_r[u_j, s+1]) = s$ 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-neighbor 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 linear MIM-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-neighbor 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 7.1.2.5 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. 6.2.1.

We have now argued for all the correspondences in Table 7.2. By that, we conclude from Prop. 6.2.1 and Definition 7.1.3 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 7.2.2. *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 7.2.1 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 5.1.1, 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 $\mathcal{O}(n \cdot log\ n)$. □

Chapter 8

Computing Optimal Linear Layouts of Trees

To compute an optimal layout with regards to the linear MIM-width of the tree T, we first mention a useful remark which is effectively the reverse implication of the Path Layout Lemma:

Remark 8.0.1 (Path Existence). If $lmw(T) = k$; then there exists a path $P \subseteq T$ such that every connected component in $T \backslash N[P]$ has linear MIM-width at most $k - 1$.

Proof. If $lmw(T) = k$, then by Theorem 4.3.4, $D(v, k) < 3$ for every node $v \in V_T$. If this is true, then we show in the proof of Lemma 4.3.3 that there always exists such a path. □

Knowing that this is true, we have an obvious route towards finding an optimal layout: We employ a procedure to find a path P in the tree T such that all connected components of $T \backslash N[P]$ have linear MIM-width strictly less than $lmw(T)$. We can thereby find optimal layouts for each of these components by applying the procedure recursively and finally call LinOrd (outlined in the proof of Lemma 4.2.2) on the resulting structures to obtain an optimal layout for T.

In the pseudocode below, the rooted tree given as input is denoted T_x, in contrast to earlier T_r. This is done to highlight that T_x is not necessarily the whole tree, but rather a rooted subtree that is given as input in a recursive call to the procedure.

How is chosen depends on what type type of tree T_x is. We choose a path in T_x using the following strategy; it takes advantage of the fact that every tree of Type n has one (or two) subtree(s) of Type $(n - 1)$ hidden within it.

Type 1 trees: Choose P to start at the root x, and as long as the last node in P has a child v that is a k-neighbor, v is appended to P.

function LayoutFromPath(T_x: rooted tree, L: $\{label(T_x[v]) \mid v \in V_{T_x}\}$)
  $l = (a_1, \ldots, a_p, last\_type) \leftarrow label(T_x)$
  $k \leftarrow a_1$
  if $last\_type = t.0$ and $k = 0$ **then** ▷ T_x = singleton
    return (x)
  else if $last\_type =$ "$t.0$" and $k = 1$ **then** ▷ T_x = star
    $v_1, \ldots, v_d \leftarrow$ children of x
    return $(x, v_1, \ldots, v_d)$
  else ▷ Main case
    $P \leftarrow$ a path found according to the type of T_x
    $\mathcal{C} \leftarrow \{$connected components in $T_x \backslash N[P]\}$
    $\mathcal{S} \leftarrow \emptyset$ ▷ Collection of layouts
    for $C \in \mathcal{C}$ **do**
      $x' \leftarrow$ root in C according to orientation in T_x
      $L' \leftarrow \{label(C_{x'}[v]) \mid v \in C_{x'}\}$
      $\mathcal{S} \leftarrow \mathcal{S} \cup$ LayoutFromPath($C_{x'}$,L')
    end for
    return LinOrd(T_x,P,$\mathcal{S}$)
  end if
end function

Type 2 trees: Let v_1 and v_2 be the two k-neighbors of x that we know exist by definition of Type 2 trees. Choose paths P_1, P_2 in $T_x[v_1]$ and $T_x[v_2]$ respectively by the strategy for Type 1 trees, then glue together the paths in x.
Type 3 trees: Let v be the child of x that is k-critical. Choose a path P in $T_x[v]$ by the strategy for Type 2 trees.
Type 4 trees: Let w be the parent of the k-critical node in T_x. Choose a path P that is satisfying in $T_x[w]$ as described above.

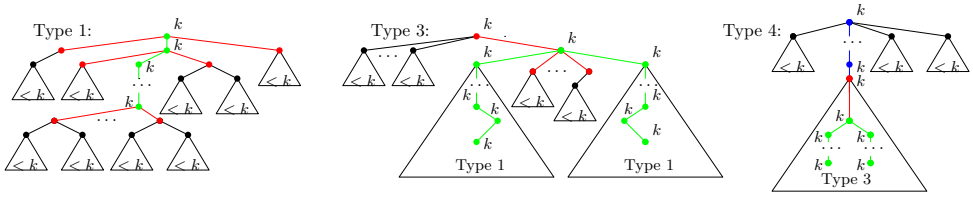

Figure 8.1: The desired path P is shown in green for all types of trees

Lemma 8.0.2. *This algorithm, given a tree T, computes a linear ordering of the nodes in T that is optimal with regards to linear MIM-width, in time $\mathcal{O}(n \cdot log\ n)$.*

Proof. We will begin with proving correctness of the algorithm.

We show that, given a tree T with $lmw(T) = k$ as input, LayoutFromPath calculates an actual path P in T, that no connected components in $T \backslash N[P]$ can have linear MIM-width equal to k, and finally, that the final ordering always is optimal.

In the two first cases, it is obvious that the resulting layout is optimal – in fact, no suboptimal layout of a singleton ($k = 0$) or a star ($k = 1$) exists.

In the more interesting case, we need to show that the path we have found in each case satisfies the property of Remark 8.0.1, and that labels are easily found. This is not hard to show, but takes some space.

Type 1: These trees contain no k-critical nodes, which by definition means that for any node v in T_x, at most one of its children is a k-neighbor of v. The resulting P is obviously a path in T_x. No node in P can possibly have a k-neighbor outside of P, therefore all connected components of $T \backslash N[P]$ have LMIM-width $\leq k - 1$. Furthermore, every component of $T - N[P]$ is a rooted sub-tree in some node of T_x, and so the labels are already known when compiling L'.
Type 2: In these trees the root x is k-critical. We look at the trees rooted in the two k-neighbors of x, $T_x[v_1]$ and $T_x[v_2]$. By Remark 6.1.2 these must both be Type 1 trees, and so we find paths P_1, P_2 in $T_x[v_1]$ and $T_x[v_2]$ respectively, as described above. Gluing these paths together at x we get a satisfying path for T_x, and we still have correct labels for the components in $T \backslash N[P]$.
Type 3: In these trees, x has exactly one child v such that $T_x[v]$ is of type 2 and none of its other children have LMIM-width k. x is not a k-neighbor of v, since else $D_T(v, k) = 3$. Every other node in P has all their neighbors in $T_x[v]$. Again, every tree in $T \backslash N[P]$ is a rooted subtree, thus every label is known when compiling L'.
Type 4: In these trees, T_x contains precisely one node $w \neq r$ such that w is the parent of a k-critical node, v. This w is easy to find using the labels, and clearly the tree $T_x[w]$ is a type 3 tree with LMIM-width k. w is still not a k-neighbor of v, therefore P is a satisfying path in T_x. In this case, we have one connected component of $T_x \backslash N[P]$ that is not a rooted subtree in any node of T_x, namely $T_x \backslash T_x[w]$. Thus for every ancestor y of w (the blue path in Figure 8.1), $T_x[y] \backslash T_x[w]$ is not a full rooted sub-tree either, and we need to update the labels of these trees. However, $T_x[y] \backslash T_x[w]$ is by definition equal to $T_x[y, k]$, whose label is equal to $label(T_x[y])$ with the first integer omitted. Thus we quickly find the correct labels to deliver to the recursive call.

We have already proven in Lemma 4.2.2 that the algorithm LinOrd, given a path P in T and layouts of the connected components of $T \backslash N[P]$ with linear MIM-width $< k$, computes a layout of T with linear MIM-width $= k$ in linear time. For any tree $S \subseteq T \backslash N[P]$, $lmw(S) < lmw(T)$; every branch of computation will therefore eventually come down to one of the base cases: S_0 is a singleton, or S_0 is a star. From this, we conclude that LayoutFromPath

computes an optimal layout.

For the time bound, observe that all operations done by LayoutFromPath can be done in at most linear time: find type of tree, find k-critical node, find path (in the worst case, check all nodes on the path for linear MIM-width until you hit a leaf), extract connected components and labels of these, and execute LinOrd. It is clear that no two connected components with linear MIM-width (say) s can overlap. The sum of all calls of LayoutFromPath on subtrees with linear MIM-width s will therefore take time ω proportional to the size of T. All subtrees of T that LayoutFromPath is called upon, can be partitioned into at most $k+1$ classes where all trees in the same class have the same linear MIM-width , and every class forms a forest of subtrees within T. Then it is clear that the total runtime of every call to LayoutFromPath, Ω, is equal to the sum of the runtime of each class, which is at most $(k+1) \cdot \omega = \mathcal{O}(k \cdot n)$. By Remark 5.1.1, we know that $k = \mathcal{O}(log\ n)$, thus the runtime of LayoutFromPath is $\mathcal{O}(n \cdot log\ n)$. □

Chapter 9

Conclusion

We have shown that the linear MIM-width of a tree of size n can be computed in $\mathcal{O}(n \cdot log(n))$ time, as well as a linear layout of the tree with optimal MIM-width. This result is significant in two ways:

First and foremost, we show the first polynomial-time exact algorithm for either MIM-width or linear MIM-width on any graph class on which either of these parameters is not bounded.

Additionally, we have shown that when restricting our scope to trees, linear MIM-width behaves in much the same way as other linear parameters, like pathwidth [10] or linear rank-width [1]. That is, as the case is with the other parameters, there is a single structure discriminating between trees of linear MIM-width k and $k+1$, namely the existence of a node with three k-neighbors. We ask this question: Is the same true for every parameter with a linear version? Will linear graph width parameters always have a single discriminating structure on trees, stemming from the fact that a tree can branch out in more than two directions? This is not always the case, e.g. *band-width* is a linear parameter that is NP-complete to calculate on trees [21]. This structure might still be useful to have in mind for anyone who wants to explore how other linear parameters behave on trees.

As we have stated several times in this thesis, apart from the nice results on specific graph classes in [3], not much is understood regarding MIM-width and linear MIM-width, and how to compute them. The results shown in this thesis could provide a small stepping stone towards a deeper understanding of the parameters. For example, it is clear that $T_{3,3,3}$ is a forbidden structure (i.e. as an induced subgraph) in graphs with linear MIM-width 1. It could be interesting to investigate other forbidden structures in graphs with low linear MIM-width. Is it possible to recognize graphs with linear MIM-width 1 in polynomial time through showing that no such structure exists in the graph? If not the problem turns out to be NP-complete, this may be a fruitful strategy.

Bibliography

[1] Isolde Adler and Mamadou Moustapha Kanté. Linear rank-width and linear clique-width of trees. In *Graph-Theoretic Concepts in Computer Science - 39th International Workshop, WG 2013, Lübeck, Germany, June 19-21, 2013, Revised Papers*, pages 12–25, 2013.

[2] Olav Røthe Bakken. Private communication. 2019.

[3] Rémy Belmonte and Martin Vatshelle. Graph classes with structured neighborhoods and algorithmic applications. *Theor. Comput. Sci.*, 511:54 – 65, 2013.

[4] Benjamin Bergougnoux and Mamadou Moustapha Kanté. Rank based approach on graphs with structured neighborhood. *CoRR*, abs/1805.11275, 2018.

[5] Hans L. Bodlaender, Marek Cygan, Stefan Kratsch, and Jesper Nederlof. Solving weighted and counting variants of connectivity problems parameterized by treewidth deterministically in single exponential time. *CoRR*, abs/1211.1505, 2012.

[6] Binh-Minh Bui-Xuan, Jan Arne Telle, and Martin Vatshelle. Fast dynamic programming for locally checkable vertex subset and vertex partitioning problems. *Theor. Comput. Sci.*, 511:66 – 76, 2013.

[7] Bruno Courcelle. The monadic second-order logic of graphs. i. recognizable sets of finite graphs. *Information and Computation*, 85(1):12 – 75, 1990.

[8] Reinhard Diestel. *Graph Theory, 4th Edition*, volume 173 of *Graduate texts in mathematics*. Springer, 2012.

[9] Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity*. Monographs in Computer Science. Springer, 1999.

[10] John A. Ellis, Ivan Hal Sudborough, and Jonathan S. Turner. The vertex separation and search number of a graph. *Inf. Comput.*, 113(1):50–79, 1994.

[11] András Frank. Some polynomial algorithms for certain graphs and hypergraphs. In *Proceedings of the 5th British Combinatorial Conference, 1975*. Utilitas Mathematica, 1975.

[12] Martin Grötschel, László Lovász, and Alexander Schrijver. *Geometric Algorithms and Combinatorial Optimization*, volume 2 of *Algorithms and Combinatorics*. Springer, 1988.

[13] Michel Habib, Ross McConnell, Christophe Paul, and Laurent Viennot. Lex-bfs and partition refinement, with applications to transitive orientation, interval graph recognition and consecutive ones testing. *Theoretical Computer Science*, 234(1):59 – 84, 2000.

[14] Svein Høgemo, Jan Arne Telle, and Erlend Raa Vågset. Linear mim-width of trees. February 2019. Submitted to WG 2019, to appear in proceedings.

[15] J. Hopcroft and R. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

[16] Lars Jaffke, O-joung Kwon, Torstein J. F. Strømme, and Jan Arne Telle. Generalized distance domination problems and their complexity on graphs of bounded mim-width. In *Proc. IPEC '18*, pages 6:1–6:13, 2018.

[17] Lars Jaffke, O-joung Kwon, and Jan Arne Telle. A unified polynomial-time algorithm for feedback vertex set on graphs of bounded mim-width. In *Proc. STACS '18*, pages 42:1–42:14, 2018.

[18] Dong Yeap Kang, O joung Kwon, Torstein J.F. Strømme, and Jan Arne Telle. A width parameter useful for chordal and co-comparability graphs. *Theor. Comput. Sci.*, 704:1 – 17, 2017.

[19] Mamadou Moustapha Kanté and Lhouari Nourine. Polynomial time algorithms for computing a minimum hull set in distance-hereditary and chordal graphs. In Peter van Emde Boas, Frans C. A. Groen, Giuseppe F. Italiano, Jerzy Nawrocki, and Harald Sack, editors, *SOFSEM 2013: Theory and Practice of Computer Science*, pages 268–279, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[20] Jan Kratochvíl, Paul D. Manuel, and Mirka Miller. Generalized domination in chordal graphs. *Nordic J. of Computing*, 2(1):41–50, March 1995.

[21] Burkhard Monien. The bandwidth minimization problem for caterpillars with hair length 3 is np-complete. *SIAM J. Algebraic Discrete Methods*, 7(4):505–512, October 1986.

[22] Sang-il Oum. Computing rank-width exactly. *Information Processing Letters*, 109(13):745 – 748, 2009.

[23] Irena Rusu and Jeremy Spinrad. Forbidden subgraph decomposition. *Discrete Mathematics*, 247(1):159 – 168, 2002.

[24] Sigve Hortemo Sæther and Martin Vatshelle. Hardness of computing width parameters based on branch decompositions over the vertex set. *Theor. Comput. Sci.*, 615:120–125, 2016.

[25] Larry J. Stockmeyer and Vijay V. Vazirani. Np-completeness of some generalizations of the maximum matching problem. *Information Processing Letters*, 15(1):14 – 19, 1982.

[26] Jan Arne Telle. Complexity of domination-type problems in graphs. *Nordic J. of Computing*, 1(1):157–171, March 1994.

[27] Martin Vatshelle. *New width parameters of graphs*. PhD thesis, University of Bergen, Norway, 2012.

[28] Mingyu Xiao and Huan Tan. Exact algorithms for maximum induced matching. *Inf. Comput.*, 256(C):196–211, October 2017.

Linear MIM-Width of Trees*

Svein Høgemo, Jan Arne Telle, and Erlend Raa Vågset

Department of Informatics, University of Bergen, Norway.
{svein.hogemo@student., 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 [20]. 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] 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 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.

* Long version with extra figures and full proofs is in the Appendix, and later on arxiv

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 [21] 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 Skodinis [19] 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 $T \setminus N[P] = \{T\langle v_{i,j}, u_{i,j,m}\rangle\}$, 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 $T \setminus N[P]$ 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 LMIM-width of 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, $(x_1, \ldots, x_p)$: 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 $m \leftarrow 1, |N(v_{i,j}) \backslash v_i|$ **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 $v_{i,j}$

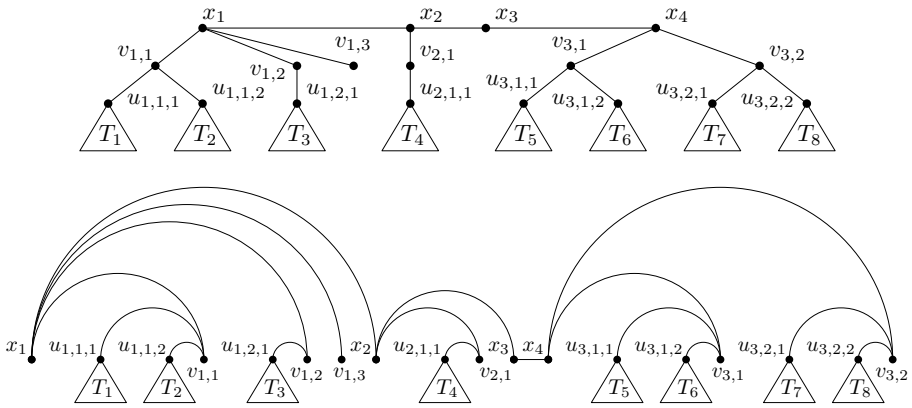

Fig. 1. A tree with a path $P = (x_1, x_2, x_3, x_4)$, with nodes in $N[N[P]]$ and dangling trees featured, and below it the order given by the Path Layout Lemma

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 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 T a tree 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 $k + 1$.

For the full proof of the forward direction, please see the Appendix, here we give a brief sketch. We assume that every node in T has $D(x, k) < 3$ and show that then $lmw(T) \leq k$. We define the following $X = \{x | x \in V(T)$ *and* $D(x, k) = 2\}$ and $Y = \{y | y \in V(T)$ *and* $D(y, k) = 1\}$ and by a case analysis show that there is always a path P in T such that all the connected components in $T \setminus N[P]$ have LMIM-width $\leq k - 1$, and proceed to use the Path Layout Lemma, to get that $lmw(T) \leq k$.

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 $D_{T_r[x]}(x, k) = 2$.*

If $lmw(T_r) = k$ then T_r cannot have two k-critical nodes as it would then by Theorem 1 T_r have $LMIM-width\ k+1$. For a detailed proof see the Appendix.

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

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 $label(T_r) = k \oplus label(T_r \backslash T_r[w])$*

See the Appendix for a Figure giving an example of a big tree and its labels. 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, se the decision tree in Figure 2, 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 $v \in Child(x)$. Let $k = max_{v \in Child(x)} \{lmw(T_r[v])\}$ and $N_k = \{v \in Child(x) \mid lmw(T[v]) = k\}$ and denote by $N_k = \{v_1, \ldots, v_q\}$ and by $l_i = label(T_r[v_i])$. Let $t_k = D_{T_r[x]}(x, k)$ by noting that $t_k = |\{v_i \in N_k \mid v_i$ has child u_j with $lmw(T_r[u_j]) = k\}|$. Given this, we 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,* $label(T_r[x]) = (k+1, t.1)$

Proof. For a full proof see the Appendix, here we only give a sketch. Observe the decision tree in Figure 2 which takes care of all cases, 1 up to 7, apart from the base cases. It follows from the definition of labels, 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, and this shows that exactly one case applies to every possible rooted tree. To prove that labels are assigned correctly a case analysis is made based on Definition 4 and position of k-critical nodes. We argue for the two most complicated cases only.
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 $lmw(T_r[x]) = k + 1$ and $T_r[x]$ is a type 1 tree.

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 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

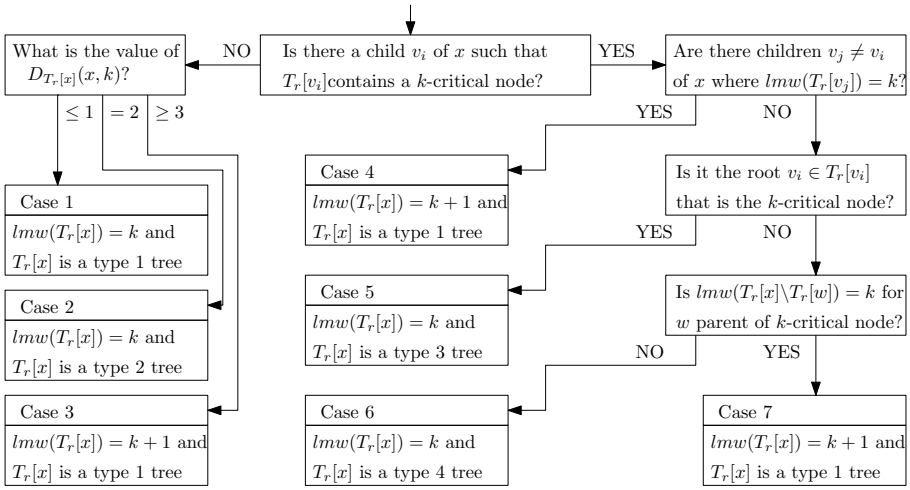

Fig. 2. A decision tree corresponding to the case analysis of Proposition 1

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 $label(T_r[x]) = k \oplus label(T_r[x]\backslash T_r[w])$ 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 $T_r[x]\backslash(T_r[w_1] \cup \ldots \cup T_r[w_{i-1}])$. 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 $T_r[x]\backslash(T_r[w_1] \cup \ldots \cup T_r[w_p]) = \emptyset$.

Now, in the first level of a recursive call to Prop. 1 the role of $T_r[x]$ is taken by $T_r[x]\backslash T_r[w_1]$, and in the next level it is taken by $(T_r[x]\backslash T_r[w_1])\backslash T_r[w_2]$ 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 $T_r[x, s] = T_r[x]\backslash(T_r[w_1] \cup T_r[w_2] \cup \ldots \cup T_r[w_q])$*

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 $s \in label(T_r[x])$

The above Remarks follow from the definitions, for a proof see Appendix. 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 $label(T_r)$, 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 $label(T_{union}[x, 0])$
  $\{v_1, \ldots, v_d\}$ = 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 $s \leftarrow 1, \max_{i=1}^{d}\{\text{first element of } label(T_r[v_i])\}$ **do**
    $\{l'_1, \ldots, l'_d\} = \{label(T_r[v_i, s+1]) \mid 1 \leq i \leq d\}$
    $N_s = \{v_i \mid 1 \leq i \leq d,\ s \in l'_i\}$
    $t_s = |\{v_i \mid v_i \in N_s,\ v_i \text{ has child } u_j \text{ s.t. } s \in label(T_r[u_j, s+1])\}|$
    if $|N_s| > 0$ **then**
      $case \leftarrow$ the case from Prop. 1 applying to s, $\{l'_1, \ldots, l'_d\}$, N_s and t_s
      $cur\_label \leftarrow$ as given by $case$ in Prop. 1 ($s \oplus cur\_label$ if Case 6)

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. See the Appendix for a full proof, here we give only a sketch. The crucial issue is to prove the loop invariant: "At the end of the s'th iteration of the for loop the value of cur_label is equal to $label(T_{union}[x, s+1])$." which suffices since for $s > lmw(T_r[x])$, we have $T_{union}[x, s] = T_r[x]$. We first argue for the correspondence given by the below Table, between parameters used in Proposition 1 and parameters used in the for loop of MakeLabel.

Let us here give only the most complicated of these arguments, showing that t_s computed in iteration s of the for loop corresponds to $t_k = D_{T_r[x]}(x, k)$ in

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 $label(T_{union}[x, s+1] \backslash T_r[w, s+1])$

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.
By the correspondences in Table 4, we conclude from Prop. 1 and Definition 6 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, with k 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, there-

fore 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. For a detailed proof see Appendix. 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:
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 below.
2) Call this layout-algorithm recursively on every rooted tree in $T_r \backslash N[P]$ to obtain linear layouts; for this, we need the correct labels for these trees.
3) Call LinOrd on T_r, P and the layouts provided in step 2.
Type 0 trees: Choose $P = (r)$.
Type 1 trees: Choose P to start at the root r, and as long as the last node in P has a k-neighbour $v \notin P$, v is appended to P.
Type 2 trees: We look at the trees rooted in the two k-neighbours of r, $T_r[v_1]$ and $T_r[v_2]$. These are Type 1 trees. Choose paths P_1, P_2 for $T_r[v_1]$ and $T_r[v_2]$ as described above. Gluing these paths together at r we get the path for T_r.
Type 3 trees: r has exactly one child v such that $T_r[v]$ is of type 2. Choose P as described above for $T_r[v]$.
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. $T_r[w]$ is a type 3 tree. Choose P for $T_r[w]$ as described above which will be the path for T_r.
For all paths chosen above, the trees in $T \backslash N[P]$ have LMIM-width strictly less than k since no node in P has a k-neighbour that is not in P. For the recursive calls we need labels for all subtrees in $T \backslash N[P]$. In every case except Type 4 trees, all these subtrees are full rooted subtrees of T_r, and the label is clearly known. In Type 4 trees, the subtree $T_r \backslash T_r[w]$, where w the parent of a k-critical node, is not a full rooted subtree. In this case we must update the label of every ancestor y of w but this is simple, since $label(T_r[y] \backslash T_r[w]) = label(T_r[y, k])$ which we get by removing the first element from $label(T_r[y])$.

Bibliography

[1] Isolde Adler and Mamadou Moustapha Kanté. Linear rank-width and linear clique-width of trees. *Theor. Comput. Sci.*, 589:87–98, 2015.
[2] Rémy Belmonte and Martin Vatshelle. Graph classes with structured neighborhoods and algorithmic applications. *Theor. Comput. Sci.*, 511:54 – 65, 2013.

[3] Benjamin Bergougnoux and Mamadou Moustapha Kanté. Rank based approach on graphs with structured neighborhood. *CoRR*, abs/1805.11275, 2018.
[4] Binh-Minh Bui-Xuan, Jan Arne Telle, and Martin Vatshelle. Fast dynamic programming for locally checkable vertex subset and vertex partitioning problems. *Theor. Comput. Sci.*, 511:66 – 76, 2013.
[5] Reinhard Diestel. *Graph Theory, 4th Edition*, volume 173 of *Graduate texts in mathematics*. Springer, 2012.
[6] John A. Ellis, Ivan Hal Sudborough, and Jonathan S. Turner. The vertex separation and search number of a graph. *Inf. Comput.*, 113(1):50–79, 1994.
[7] Fedor V. Fomin, Petr A. Golovach, and Jean-Florent Raymond. On the tractability of optimization problems on H-graphs. In *Proc. ESA 2018*, pages 30:1 – 30:14, 2018.
[8] Esther Galby, Andrea Munaro, and Bernard Ries. Semitotal domination: New hardness results and a polynomial-time algorithm for graphs of bounded mim-width. *CoRR*, abs/1810.06872, 2018.
[9] Petr A. Golovach, Pinar Heggernes, Mamadou Moustapha Kanté, Dieter Kratsch, Sigve Hortemo Sæther, and Yngve Villanger. Output-polynomial enumeration on graphs of bounded (local) linear mim-width. *Algorithmica*, 80(2):714–741, 2018.
[10] Martin Charles Golumbic and Udi Rotics. On the clique-width of some perfect graph classes. *Int. J. Found. Comput. Sci.*, 11(3):423–443, 2000.
[11] Petr Hlinený, Sang-il Oum, Detlef Seese, and Georg Gottlob. Width parameters beyond tree-width and their applications. *Comput. J.*, 51(3):326–362, 2008.
[12] Lars Jaffke, O-joung Kwon, Torstein J. F. Strømme, and Jan Arne Telle. 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, 2018.
[13] Lars Jaffke, O-joung Kwon, and Jan Arne Telle. 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, 2017.
[14] Lars Jaffke, O-joung Kwon, and Jan Arne Telle. 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, 2018.
[15] Stefan Mengel. Lower bounds on the mim-width of some graph classes. *Discrete Applied Mathematics*, 248:28–32, 2018.
[16] Rolf H Möhring. Graph problems related to gate matrix layout and pla folding. In *Computational graph theory*, pages 17–51. Springer, 1990.
[17] Sang-il Oum. Rank-width: Algorithmic and structural results. *Discrete Applied Mathematics*, 231:15–24, 2017.
[18] Sigve Hortemo Sæther and Martin Vatshelle. Hardness of computing width parameters based on branch decompositions over the vertex set. *Theor. Comput. Sci.*, 615:120–125, 2016.
[19] Konstantin Skodinis. Construction of linear tree-layouts which are optimal with respect to vertex separation in linear time. *J. Algorithms*, 47(1):40–59, 2003.
[20] Martin Vatshelle. *New width parameters of graphs*. PhD thesis, University of Bergen, Norway, 2012.
[21] Koichi Yamazaki. Inapproximability of rank, clique, boolean, and maximum induced matching-widths under small set expansion hypothesis. *Algorithms*, 11(11):173, 2018.