Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

On the thinness of trees and other graph classes

Sobre la thinness de árboles y otras clases de grafos

Tesis de Licenciatura en Ciencias de la Computación

Eric Brandwein, LU 349/16, brandweineric@gmail.com
Agustín Sansone, LU 99/17, agustinsansone@gmail.com

Director: Flavia Bonomo
Codirector: Carolina Lucía Gonzalez

Buenos Aires, March 19, 2022

Abstract

The thinness of a graph is a width parameter that generalizes some properties of interval graphs, which are exactly the graphs of thinness one. Many NP-complete problems can be solved in polynomial time for graphs with bounded thinness, given a suitable representation of the graph. In this work we present a constructive $\mathcal{O}(n \log(n))$-time algorithm to compute the thinness for any given tree, along with an optimal consistent solution (ordering and partition). We use some intermediate results of this construction to improve known bounds of the thinness in some special trees. We also show the exact thinness of crown graphs CR_n, and give new upper bounds for the thinness of other graph classes (including grids GR_r). Finally, we propose some heuristics to construct a consistent solution for some more general graphs.

Keywords: trees, thinness, polynomial algorithm, crown graphs, grid graphs, heuristics.

Resumen

La thinness de un grafo es un parámetro de anchura que generaliza algunas propiedades de grafos de intervalo, los cuales son exactamente los grafos con thinness uno. Muchos problemas NP-completos pueden ser resueltos en tiempo polinomial para grafos de thinness acotada, dada una representación adecuada. En este trabajo presentamos una algoritmo constructivo con complejidad temporal $\mathcal{O}(n \log(n))$ para computar la thinness de un árbol dado, junto a una solución óptima consistente (orden y partición). Utilizamos resultados intermedios de esta construcción para mejorar cotas conocidas de thinness en árboles para algunos casos. También mostramos la thinness exacta the los grafos corona CR_n, y damos una cota superior para la thinness de otras clases de grafos (incluyendo grafos grilla GR_r). Finalmente, proponemos algunas heurísticas para construir una solución consistente para algunos grafos más generales.

Palabras clave: árboles, thinness, algoritmo polinomial, grafos corona, grafos grilla, heurísticas.

Table of Contents

CHAPTER 1

Introduction

The study of structural graph "width" parameters like tree-width, clique-width and rank-width has been ongoing since at least three decades ago, and their algorithmic use has also been increasing [5].
The concept of *thinness* of a graph was introduced for the first time in [10] as a new graph invariant. The main intuition from the definition presented is a generalization of *interval graphs*, which are exactly the graphs with thinness equal to one. This concept is interesting because if a representation of a graph as a *k-thin* graph is given for a constant value k, then some known NP-complete problems can be solved in polynomial time. Some examples are the maximum weighted independent set problem [10] and the bounded coloring with fixed number of colors [2].
The definition of k-thinness involves a vertex ordering and partition into k classes satisfying certain properties. In that case, the order and partition are said to be *consistent*. For a given order of the vertices of G, there exists a known algorithm to compute an optimal consistent partition of the vertices of G with time complexity $\mathcal{O}(n^3)$, where n is the number of vertices in G [1], since the problem can be reduced in linear time to the optimal coloring in an auxiliary *co-comparability* graph of n vertices [7]. On the other hand, computing a consistent ordering of the vertices for a given partition, or detect it does not exist, is NP-complete [1]. Very recently, by a reduction from that problem, it was proved that deciding whether the thinness of a graph is at most k, without any given order or partition, is NP-complete [15]. One of the objectives of this thesis is to solve a part of this problem: we will show an algorithm that computes the thinness of any tree in polynomial time. This will be

the first non-trivial class in the literature for which we know how to compute the thinness (and consistent order and partition of the vertices) efficiently.[1] Some efforts were made before to study the thinness of trees in [12].
For this algorithm, we will heavily rely on the proof and the algorithm shown in [9]. In fact, the structure of the proof and the algorithm are both very similar to those shown there for another graph invariant, the *Linear Maximum Induced Matching Width*, which is a known lower bound for the thinness [1].
The organization of this thesis is the following. There are two main sections with new results. In the first one we will show how to compute in $\mathcal{O}(n \log(n))$ time and memory the thinness of any tree with an optimal consistent partition and ordering of the vertices. This result can also be generalized to forests. In the second main section we will prove the exact thinness or some bounds related to the thinness for some specific graph classes or graph families; in particular we are going to determine the thinness of all *complete m-ary trees* and *crowns* CR_n and we are also going to show some new bounds for the thinness of trees and *grids* GR_n.

1.1 Definitions and Preliminary Results

All graphs in this thesis are finite, undirected and have no loops or multiple edges. For all graph-theoretic notions and notation not defined here, we refer to D. West's book [17].
Let G be a graph, we denote by $V(G)$ the set of its vertices or nodes and by $E(G)$ the set of its edges. We denote by $N(v)$ the neighborhood of a vertex $v \in V$, i.e., the set of vertices that are adjacent to v, and by $N[v]$ the closed neighborhood of v: $N(v) \cup \{v\}$. If $X \subseteq V(G)$, we denote by $N(X)$ the set of vertices not in X having at least one neighbor in X, and by $N[X]$ the closed neighborhood $N(X) \cup X$.
We denote by $G[W]$ the subgraph of G induced by a set $W \subseteq V(G)$, and by $G - W$ or $G \setminus W$ the graph $G[V(G) \setminus W]$. We use $G \setminus (u, v)$ to denote the graph with vertices $V(G)$ and edges $E(G) \setminus \{(u, v)\}$. A subgraph H of G is a *spanning subgraph* if $V(H) = V(G)$.
The *degree* of a vertex v in a graph G is the number of edges that have v as one of their endpoints. A *path* is a list of ordered vertices where each vertex is adjacent to the one before it, if any. A *simple path* is a path where each vertex appears at most once. Two vertices are *connected* in a graph G if there exists a path starting on one and ending on the other. A *connected graph* is one where all vertices are pairwise

[1] A polynomial-time algorithm and forbidden induced subgraphs characterization are known for cographs [1], but the algorithm and proofs follow from their decomposition theorem without much more complication.

connected. A *connected component* or simply *component* of a graph G is a maximal induced subgraph of G which is connected.
A *cycle* is a path with at least four vertices in which the first and last vertices are equal, and no other vertex is repeated. A *tree* is a connected graph with no cycles. A *leaf* of a tree T is a vertex with degree one in T. A *rooted tree on vertex* r is a tree in which vertex r is labeled as the root, and we will usually denote it by T_r. The *ancestors* of a vertex v in a rooted tree T_r are all vertices in *the* simple path between v and r which are not v. Note that r has no ancestors in T_r. The *descendants* of a vertex v in T_r are all vertices for which v is an ancestor in T_r. The *direct children* or simply *children* of a vertex v are those neighbors of v which are also descendants. Conversely, the *direct parent* or *parent* of a vertex $v \neq r$ is the only neighbor of v which is also an ancestor of v. In rooted tree T_r, the vertex r does not have any parent, while all other vertices have exactly one. The *grandchildren* of a vertex v are the children's children, and the *grandparents* are the parents' parents.
A *strict subtree* of a tree T is a tree induced by some vertices of T which is different from T.
Let T be a tree containing the adjacent vertices 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.
A *complete graph* is a graph where all vertices are pairwise adjacent. We denote by K_n the complete graph of n vertices. A *clique* of a graph G is a complete induced subgraph of G. The maximum size (number of vertices) of a clique of G is denoted by $\psi(G)$.
Given k *cliques* $Q_1, Q_2, \ldots Q_k$ and one vertex q_i for each Q_i, we say the graph G is a *tree of cliques* if $V(G) = \bigcup_{i=1}^{k} V(Q_i)$ and $E(G) = S \cup \bigcup_{i=1}^{k} E(Q_i)$, where S is a set of edges of the form (q_i, q_j), such that the graph induced by the vertices $\{q_1, q_2, \ldots, q_k\}$ in G is a tree.
For a positive integer r, the $(r \times r)$-*grid*, noted GR_r, is the graph whose vertex set is $\{(i, j) : 1 \leq i, j \leq r\}$ and whose edge set is $\{((i, j), (k, l)) : |i - k| + |j - l| = 1, \text{ where } 1 \leq i, j, k, l \leq r\}$.
A *complete bipartite graph* $K_{n,n}$ is a graph whose vertices can be partitioned into two subsets V_1 and V_2, both of size n, such that no edge has both endpoints in the same subset, and every vertex $v_1 \in V_1$ is adjacent to every vertex $v_2 \in V_2$. The *crown graph* CR_n is the graph on $2n$ vertices obtained from a complete bipartite graph $K_{n,n}$ by removing a perfect matching.
A *coloring* of a graph is an assignment of colors to its vertices such that any two adjacent vertices are assigned different colors. The minimum number of colors needed to color G is called the *chromatic number* of G and denoted by $\chi(G)$.
A graph G is *perfect* whenever for every induced subgraph H of G, $\chi(H) = \psi(H)$.
A graph G is a *comparability graph* if there exists a partial order in $V(G)$ such that

two vertices of G are adjacent if and only if they are comparable by that order. A graph G is a *co-comparability graph* if its complement $\overline{G}$ is a comparability graph.
A graph G is an *interval graph* if to each vertex $v \in V(G)$ can be associated a closed interval $I_v = [l_v, r_v]$ of the real line, such that two distinct vertices u, $v \in V(G)$ are adjacent if and only if $I_u \cap I_v \neq \emptyset$. The family $\{I_v\}_{v \in V(G)}$ is an *interval representation* of G.
A graph $G = (V, E)$ is *k-thin* if there exists an ordering $\sigma = v_1, \ldots, v_n$ of V and a partition S of V into k classes such that, for each triple (r, s, t) with $r < s < t$, if v_r,v_s belong to the same class and $(v_t, v_r) \in E$, then $(v_t, v_s) \in E$. An order and a partition satisfying those properties are said to be *consistent*. We call the tuple (σ, S) a *consistent solution* or *consistent layout*. The minimum k such that G is *k-thin* is called the *thinness* of G, and we note it $\text{thin}(G)$.
Let G be a graph and $<$ an ordering of its vertices. The graph $G_<$ has $V(G)$ as a vertex set, and $E(G_<)$ is such that for $v < w$ in the ordering, $(v, w) \in E(G_<)$ if and only if there is a vertex z in G such that $w < z$, $(z, v) \in E(G)$, and $(z, w) \notin E(G)$. An edge of $G_<$ represents that its endpoints cannot belong to the same class in a vertex partition that is consistent with the ordering $<$.

Theorem 1. *[1, 2] Given a graph G and an ordering $<$ of its vertices, the graph $G_<$ has the following properties:*

1. *the chromatic number of $G_<$ is equal to the minimum integer k such that there is a partition of $V(G)$ into k sets that is consistent with the order $<$, and the color classes of a valid coloring of $G_<$ form a partition consistent with $<$;*

2. *$G_<$ is a co-comparability graph;*

3. *if G is a co-comparability graph and $<$ a comparability ordering of $\overline{G}$, then $G_<$ is a spanning subgraph of G.*

Since co-comparability graphs are perfect [11], $\chi(G_<) = \psi(G_<)$. We thus have the following.

Corollary 1.1. *Let G be a graph, and k a positive integer. Then* $\text{thin}(G) \geq k$ *if and only if, for every ordering $<$ of $V(G)$, the graph $G_<$ has a clique of size k.*

The *linear MIM-width* of a graph G, denoted as $\text{lmimw}(G)$, is the smallest integer k such that the vertices of G can be arranged in a linear layout $v_1, \ldots, v_n$ in such a way that for every $i = 1, \ldots, n-1$, the size of a maximum induced matching in the bipartite graph formed by the edges of G with an endpoint in $\{v_1, \ldots, v_i\}$ and the other one in $\{v_{i+1}, \ldots, v_n\}$ is at most k. This is the linear version of a parameter called *MIM-width* [16], that is a lower bound for the linear MIM-width. It was proven

in [1] that $\text{lmimw}(G) \leq \text{thin}(G)$. Moreover, a linear ordering $v_1, \ldots, v_n$ realizing the thinness, satisfies that the size of a maximum induced matching in the bipartite graph formed by the edges of G with an endpoint in $\{v_1, \ldots, v_i\}$ and the other one in $\{v_{i+1}, \ldots, v_n\}$ is at most $\text{thin}(G)$.
The *path-width* [13] of a graph is the minimum value of k such that the graph can be obtained from a sequence of graphs $G_1, \ldots, G_r$ each of which has at most $k+1$ vertices, by identifying some vertices of G_i pairwise with some of G_{i+1} $(1 \leq i < r)$.
In [10] it was proven that the thinness of a graph is at most the pathwidth plus one, and that the gap may be high, since the pathwidth of a complete graph with r vertices is $r-1$, while its thinness is 1. Also it was proven in [6, 14] that the pathwidth of the complete m-ary tree of height h is $\Theta(h)$.

CHAPTER 2

Polynomial Algorithm for Thinness of Trees

A considerable part of the ideas related to the construction of the algorithm to compute the thinness (and a consistent ordering and partition of the vertices) we are presenting in this section was inspired by an algorithm to compute the linear maximum induced matching width of a tree and an optimal layout [9], which was at the same time inspired by the framework behind the path-width algorithm presented in [6].

2.1 Path Layout Lemma

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 thinness less than or equal to k then* $\text{thin}(T) \leq k + 1$. *Moreover, given the consistent orderings and partitions for the components in at most k classes we can in linear time compute a consistent ordering and partition for T in at most $k + 1$ classes.*

Proof. Using the consistent orderings $\{\sigma_{T\langle v_{i,j}, u_{i,j,m}\rangle}\}$ and partitions $\{S_{T\langle v_{i,j}, u_{i,j,m}\rangle}\}$ into k classes of the connected components $T\langle v_{i,j}, u_{i,j,m}\rangle$ of $T \setminus N[P]$, we give the Algorithm 1 constructing an order σ_T and partition S_T into $k + 1$ classes of the vertices of T, showing that $\text{thin}(T) \leq k + 1$. Here, $v_{i,j}$ corresponds to the neighbors of $x_i \in P$ which are not themselves in P, and $u_{i,j,m}$ the neighbors of $v_{i,j}$ different from x_i. We will use C_c to denote the c-th class of S_T, and $C_{T\langle v_{i,j}, u_{i,j,m}\rangle, c}$ to the c-th class of $S_{T\langle v_{i,j}, u_{i,j,m}\rangle}$. The $\oplus$ will denote the list append operation or the list concatenation operation interchangeably.

Algorithm 1 Consistent layout given the layouts of the dangling subtrees of a path.

function ConsistentLayout(T: tree, $P = (x_1, \ldots, x_p)$: path, $\{\sigma_{T\langle v_{i,j}, u_{i,j,m}\rangle}\}$: orderings, $\{S_{T\langle v_{i,j}, u_{i,j,m}\rangle}\}$: partitions)
  $\sigma_T \leftarrow \emptyset$
  for $class \leftarrow 1, k+1$ **do**
    $C_{class} \leftarrow \emptyset$
  end for
  for $x_i \in P$ **do**
    for $v_{i,j} \in N(x_i) \setminus P$ **do**
      $\sigma_T \leftarrow \sigma_T \oplus v_{i,j}$
      $C_{k+1} \leftarrow C_{k+1} \cup \{v_{i,j}\}$
      for $u_{i,j,m} \in N(v_{i,j}) \setminus x_i$ **do**
        $\sigma_T \leftarrow \sigma_T \oplus \sigma_{T\langle v_{i,j}, u_{i,j,m}\rangle}$
        for $c \leftarrow 1, k$ **do**
          $C_c \leftarrow C_c \cup C_{T\langle v_{i,j}, u_{i,j,m}\rangle, c}$
        end for
      end for
    end for
    $\sigma_T \leftarrow \sigma_T \oplus x_i$
    $C_{k+1} \leftarrow C_{k+1} \cup \{x_i\}$
  end for
  $S_T \leftarrow \{C_1, \ldots, C_{k+1}\}$
  return σ_T, S_T
end function

Firstly, from the algorithm we can see that each vertex of T is added exactly once to σ_T and to only one class of S_T, and as those are the only operations performed in the algorithm apart from the initialization of the consistent solution, it has linear time complexity on the size of the tree. Now we must show that the ordering and partition are consistent, meaning, there are no three vertices $a < b < c$ in σ_T such that a and b are in the same class of the partition, and $(c, a) \in E(T)$ and $(c, b) \notin E(T)$.
We will prove it by contradiction, assuming there are three vertices $a < b < c$ of T that violate consistency. We will separate by cases and prove each one separately.

- $\boldsymbol{a \in N[P], b \notin N[P]}$, **or** $\boldsymbol{a \notin N[P], b \in N[P]}$: If one of $\{a, b\}$ belongs to $N[P]$ and the other does not, they are both added to different classes in S_T, so this triple is consistent.

- $\boldsymbol{a \notin N[P], c \in N[P]}$: This cannot happen. The only vertices of $N[P]$ adjacent to vertices not in $N[P]$ are the $v_{i,j}$ for some i, j, and they are all appended to the order before all their neighbors, so there cannot be a vertex a such that $a < v_{i,j}$ and $(v_{i,j}, a) \in E(T)$.

- $\boldsymbol{a \in N[P], c \notin N[P]}$: As before, the only possibility for a is to be $v_{i,j}$ for some i, j so as to be adjacent to a vertex $c \notin N[P]$. Also, the only vertices not in $N[P]$ adjacent to $v_{i,j}$ are the $u_{i,j,m}$ for some m, so $c = u_{i,j,m}$. And the only possible vertices between $v_{i,j}$ and $u_{i,j,m}$ in the order are the vertices of $T\langle v_{i,j}, u_{i,j,m}\rangle$, which means that $b \in T\langle v_{i,j}, u_{i,j,m}\rangle$. But then $a \in N[P]$ and $b \notin N[P]$, which means they are in different classes of the partition, and so this triple is consistent.

  Combining the last three cases, we see that, to violate consistency, the three vertices must belong to $N[P]$, or none of the three can.

- $\boldsymbol{\{a, b, c\} \subseteq N[P]}$: We begin by noting that no vertex $v_{i,j}$ is adjacent to another vertex α in $N[P]$ such that $\alpha < v_{i,j}$, because $v_{i,j}$ is always added to the order before their corresponding x_i, which is the only vertex in $N[P]$ adjacent to $v_{i,j}$. This means that $c = x_i$ for some $1 \leq i \leq p$. The only vertices adjacent to x_i that are lower in the order are all $v_{i,j}$, and x_{i-1} if $i > 1$.

  If $a = v_{i,j}$ for some j, then all vertices b such that $a < b < x_i$ are some $v_{i,k}$ for some $k > j$. As $(x_i, v_{i,k}) \in T$, this triple is consistent. If, on the other hand, $a = x_{i-1}$, then again, $b = v_{i,k}$ for some k, which means it is also adjacent to x_i, and so this triple is also consistent.

- $\boldsymbol{\{a, b, c\} \subseteq T\langle v_{i,j}, u_{i,j,m}\rangle}$ **for some** $\boldsymbol{i, j, m}$: Because $\sigma_{T\langle v_{i,j}, u_{i,j,m}\rangle}$ is a subsequence of σ_T, and all classes of $S_{T\langle v_{i,j}, u_{i,j,m}\rangle}$ are subsets of the corresponding

classes of S_T, the consistency is preserved between three vertices of the same dangling tree.

- $\boldsymbol{a \in T\langle v_{i,j}, u_{i,j,m}\rangle}$, $\boldsymbol{c \in T\langle v_{i',j'}, u_{i',j',m'}\rangle}$ **for some** $\boldsymbol{i \neq i', j \neq j', m \neq m'}$: Vertices in different dangling trees are not adjacent, so this cannot happen.
- $\boldsymbol{\{a, c\} \subseteq T\langle v_{i,j}, u_{i,j,m}\rangle}$, $\boldsymbol{b \in T\langle v_{i',j'}, u_{i',j',m'}\rangle}$ **for some** $\boldsymbol{i \neq i', j \neq j', m \neq m'}$: Either all the vertices of $T\langle v_{i,j}, u_{i,j,m}\rangle$ are added before all the vertices of $T\langle v_{i',j'}, u_{i',j',m'}\rangle$, or vice versa. This means that either $a < b$ and $c < b$, or $b < a$ and $b < c$, so an inconsistent triple cannot be found this way.

We have proven that any possible triple in the order and partition generated by the algorithm is consistent, and then the order and partition given by the algorithm are consistent. □

2.2 k-component Index Theorem

Definition 1 (k-neighbor)**.** *Let x be a vertex of a tree T, and v a neighbor of x in T. If there exists a vertex $u \neq x$ neighbor of v such that* $\text{thin}(T\langle v, u\rangle) \geq k$*, then v is a* k-neighbor *of x.*

Definition 2 (k-component index)**.** *The k-component index of x is the number of k-neighbors of x, and we note it as $D(x, k)$.*

Lemma 2. *If $D(x, k) \geq 3$ for some vertex x in T, then* $\text{thin}(T) \geq k + 1$*. We say that x is k-saturated in T.*

Proof. Let x be a vertex in T such that $D(x, k) \geq 3$. Let v_1, v_2, v_3 be three neighbors of x such that they have neighbors u_1, u_2, u_3 respectively which satisfy that $\text{thin}(T\langle v_i, u_i\rangle) \geq k$. Let $T_i = T\langle v_i, u_i\rangle$ for $i \in \{1, 2, 3\}$. Let $<$ be an ordering of the vertices of T which is part of a consistent solution for T that minimizes the amount of classes used in the partition. Let a and b be, respectively, the lowest and greatest vertex in the order $<$ that belong to any subtree T_i.
There must be at least one subtree T_j such that $a \notin T_j$ and $b \notin T_j$. As $\text{thin}(T_j) \geq k$, we know that $T_{j<}$ has a clique Q of size at least k. Let t be the greatest vertex according to $<$ that belongs to Q.
We know that $a < t < b$, because a is lower in the order than all the vertices in T_j, and b is greater. Also, there is a simple path P between a and b that does not include any vertex of T_j. Let us see why. If a and b belong to the same subtree T_i, then there is a simple path P between the two that includes only vertices of T_i, because T_i is a tree. If a and b belong to different subtrees T_h and T_i, then the simple path is

$a \to \cdots \to u_h \to v_h \to x \to v_i \to u_i \to \cdots \to b$, which does not include any vertex of T_j.
Because P begins with a vertex which is lower than t in $<$, and finishes in a vertex greater than t, there exist two adjacent vertices $a', b' \in P$ such that $a' < t < b'$. We will see that a' is adjacent in $T_<$ to all the vertices in Q, which means that there is a clique of size $k+1$ in $T_<$.
We know that $a' < t < b'$, and $(b', a') \in E(T)$, but $(b', t) \notin E(T)$, so this means that a' is adjacent to t in $T_<$.
Now, given $t' < t$ a vertex of Q, let us see that $(a', t') \in E(T_<)$. As t' is adjacent to t in $T_<$, there is a vertex $c > t$ adjacent to t' but not to t in T_j. As $a' < t < c$, $a' < c$. And as $t' < t < b'$, $t' < b'$. This leaves us with two possibilities:

- $a' < t'$: b' is adjacent in T to a' but not to t', so a' and t' are adjacent in $T_<$.
- $t' < a'$: c is adjacent in T to t' but not to a', so a' and t' are adjacent in $T_<$.

This shows that every vertex of Q is adjacent to a vertex a' in $T_<$, and so $T_<$ has a clique of size at least $k+1$, which implies that $\text{thin}(T) \geq k+1$. □

Lemma 3. *If* $\text{thin}(T) \geq k+1$*, then there exists a vertex* x *in* T *such that* $D(x,k) \geq 3$.

Proof. To prove this we first prove the following partial claim: if $\text{thin}(T) \geq k+1$ then there exists a vertex $x \in T$ such that $D(x,k) \geq 3$; or there exists a strict subtree S of T with $\text{thin}(S) \geq k+1$. We will prove the contrapositive statement, so let us assume that every vertex in T has $D(x,k) < 3$ and no strict subtree of T has *thinness* $\geq k+1$ and show that then $\text{thin}(T) \leq k$. For every vertex $x \in T$, it must then be true that $D(x,k) \leq 2$ and that $D(x,k+1) = 0$. The strategy of this proof is to show that there is always a path P in T such that all the connected components in $T \setminus N[P]$ have *thinness* $\leq k-1$. When we have shown this, we proceed to use the Path Layout Lemma, to get that $\text{thin}(T) \leq k$.
We begin by defining the following two sets of vertices:

$$X = \{x \mid x \in V(T) \text{ and } D(x,k) = 2\}$$

$$Y = \{y \mid y \in V(T) \text{ and } D(y,k) = 1\}$$

Case 1: $X \neq \emptyset$
If x_i and x_j are in X, take the simple path $P = (x_i, ..., x_j)$ connecting x_i and x_j. Both x_i and x_j have at least one k-neighbor outside of P, as they each have only one neighbor in P, and $D(x_i,k) = D(x_j,k) = 2$. This means that each element of P has two dangling subtrees with thinness greater or equal to k; one in the direction of x_i, and another one in the direction of x_j. So for all $v \in P$, $D(v,k) \geq 2$. As we assumed no vertex in T has more than 2 k-neighbors, $D(v,k) = 2$, and so v is also in X.

The fact that every pair of vertices in X are connected by a path in X means that X must be a connected subtree of T. Furthermore, this subtree must be a path, otherwise there is a vertex $w \in X$ with three neighbors in X, all of which have $D(x,k) = 2$, meaning they have at least one k-neighbor different from w. Then, $D(w,k) \geq 3$, which cannot happen as w is in X.
We therefore conclude that all vertices in X must lie on some path $P = (x_1, ..., x_p)$. The final part of the argument lies in showing that we can apply the Path Layout Lemma. For some $x_i \in P$ with $i \in \{2, ..., p-1\}$, its k-neighbors are x_{i-1} and x_{i+1}. For x_1, these neighbors are x_2 and some $x_0 \notin X$. For x_p, these neighbors are x_{p-1} and some $x_{p+1} \notin X$. Vertices x_0 and x_{p+1} only have one k-neighbor (x_1 and x_p respectively) or else they would be in X. If we make $P' = (x_0, ..., x_{p+1})$, we then see that every connected component in $T \setminus N[P']$ must have *thinness* $\leq k-1$. By the Path Layout Lemma, thin$(T) \leq k$.

Case 2: $X = \emptyset, Y \neq \emptyset$
We construct the path P in a simple greedy manner as follows. We start with $P = (y_1, y_2)$, where y_1 is some arbitrary vertex in Y, and y_2 its only k-neighbor. Then, if the last vertex in P has a k-neighbor $y' \notin P$, then we append y' to P, and repeat this process exhaustively. Since we look at finite graphs, we will eventually reach some vertex y_p such that either $y_p \notin Y$ or the k-neighbor of y_p is in P . We are then done and have $P = (y_1, ..., y_p)$, which is a path in T by construction.
One property of P is that no vertex $y_i \in P$ can have a k-neighbor outside P. In the case of $i = p$, this is by construction. In the case of $i \neq p$, if y_i had a k-neighbor outside P, it would have at least two k-neighbors (the other one being y_{i+1}) which cannot happen because X is empty. This means that $T \setminus N[P]$ has no subtrees with thinness greater than $k-1$. By the Path Layout Lemma, thin$(T) \leq k$.
Case 3: $X = \emptyset, Y = \emptyset$
As both X and Y are empty, all vertices $t \in T$ have no k-neighbors. Taking $P = (t)$ then gives us a path such that no subtree of $T \setminus N[P]$ has thinness greater than $k-1$. By the Path Layout Lemma, thin$(T) \leq k$.

We have proven the partial claim that if thin$(T) \geq k+1$ then there exists a vertex $x \in T$ such that $D(x,k) \geq 3$; or there exists a strict subtree S of T with thin$(S) \geq k+1$. To finish proving the theorem we need to show that if thin$(T) \geq k+1$ then there exists a vertex $x \in T$ with $D(x,k) \geq 3$. Assume that there is no vertex with k-component index at least 3 in T. By the partial claim, there must then exist a strict subtree S with thin$(S) \geq k+1$. But since we look at finite trees, we know that in S there must exist a minimal subtree S_0 with thinness $k+1$ with no strict subtree with thinness $> k$. By the partial claim, S_0 must contain a vertex x_0 with $D_{S_0}(x_0, k) \geq 3$.

But every dangling tree $S_0\langle v,u\rangle$ is a subtree of $T\langle v,u\rangle$, and so if $D_{S_0}(x_0,k)\geq 3$, then $D_T(x_0,k)\geq 3$ contradicting our assumption. □

Directly from the two implications we proved we can summarize:

Theorem 2 (Classification of Thinness of Trees)**.** *Let T be a tree, then* $\text{thin}(T)\geq k+1$ *if and only if $D(x,k)\geq 3$ for some vertex x in T.*

Corollary 2.1. *The thinness of an n-vertex tree T is $\mathcal{O}(\log(n))$. In fact* $\text{thin}(T)\leq \log_3(n+2)$.

Proof. Let us do an induction in the thinness of the tree.

- Base case:
  For any given tree T such that $\text{thin}(T)=1$, since $n\geq 1$ we know that $1\leq log_3(1+2)\leq log_3(n+2)$.
- Inductive step:
  Suppose the property holds for all trees such that their thinness is strictly less than a given $k>1$. Then for a given tree T such that $\text{thin}(T)=k$, we want to prove that $\text{thin}(T)=k\leq log_3(n+2)$, where n is the number of vertices in T. Since $\text{thin}(T)=k$, because of Theorem 2, there must exist a vertex x in T such that $D(x,k-1)\geq 3$. We name T_1, T_2 and T_3 the tree disjoint subtrees of thinness equal to $k-1$ such that they have their root adjacent to one of the children of x. Without loss of generality, T_1 has the lowest number of vertices (we call it n_1) among $\{T_1,T_2,T_3\}$. Notice this implies that $n\geq 3n_1+4$ (implying $n_1\leq \frac{n-4}{3}$), because of the three subtrees, x and their tree children. From the Inductive Hypothesis we can say that $\text{thin}(T_1)\leq log_3(n_1+2)$. Then

$$\text{thin}(T)-1=\text{thin}(T_1)\leq \log_3(n_1+2)\leq \log_3(\frac{n-4}{3}+2)$$

  Which implies that

$$\text{thin}(T)\leq \log_3(\frac{n-4}{3}+2)+1=\log_3(3\left[\frac{n-4}{3}+2\right])=\log_3(n+2).$$

□

Corollary 2.2. *A nontrivial tree of thinness k has at least $\frac{3^{k-1}+3}{2}$ leaves. In particular, the thinness of a tree with ℓ leaves is at most* $\log_3(6\ell-9)$.

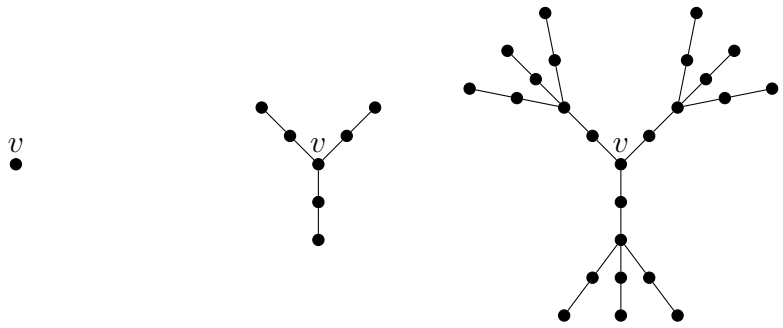
Proof. Let us do an induction in the thinness of the tree.

- Base case:
  A nontrivial tree of thinness 1 has at least 2 leaves. Since every path has thinness 1, every tree of thinness 2 has at least 3 leaves.

- Inductive step:
  Suppose the property holds for all trees with thinness at most $k > 1$, and let T be a tree with $\text{thin}(T) = k + 1$. By Theorem 2, there must exist a vertex x in T such that $D(x, k) \geq 3$. We name T_1, T_2 and T_3 the tree disjoint subtrees of thinness equal to k such that they have their root adjacent to one of the neighbors of x. From the Inductive Hypothesis, each of T_1, T_2, T_3 has at least $\frac{3^{k-1}+3}{2}$ leaves. Since some of them can be rooted at a leaf, we can ensure that T has at least $3\frac{3^{k-1}+3}{2} - 3 = \frac{3^k+3}{2}$ leaves.

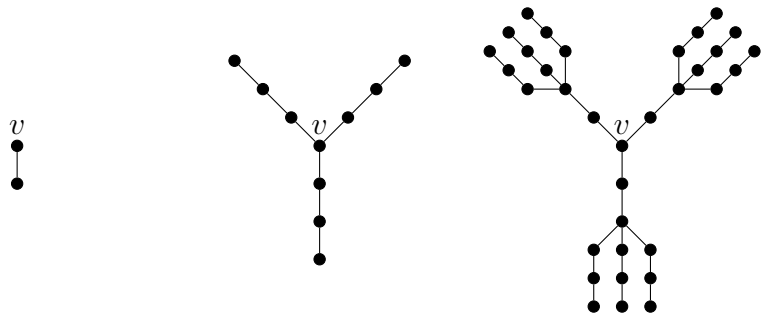
□

Theorem 2 also suggests how to build the minimum trees for each thinness value:



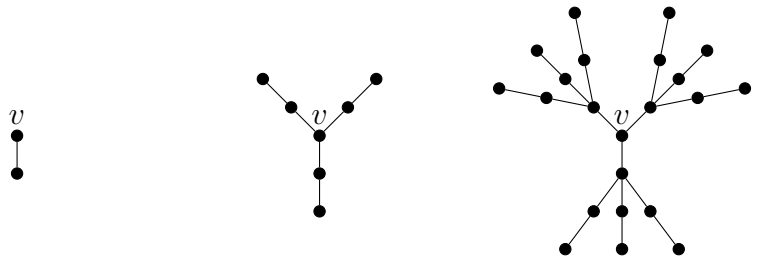
$\text{thin}(T) = 1$ $\text{thin}(T) = 2$ $\text{thin}(T) = 3$

For each thinness value k, v is such that $D(v, k-1) = 3$. The minimum tree with thinness k can be constructed by replacing each leaf in the minimum tree with thinness 2 into the minimum tree with thinness $k - 1$, thus achieving $D(v, k) = 3$ with the minimum amount of vertices.
Compare this with the minimum trees with linear MIM-width 1, 2 and 3 [8]:



$lmimwidth(T) = 1$ $lmimwidth(T) = 2$ $lmimwidth(T) = 3$

These are pretty similar, except that the leaves in the trees with thinness k are replaced by two vertices. This is because a theorem very similar to Theorem 2 is also true for linear MIM-width [9], and the smallest tree with linear MIM-width 1 is the path of two vertices, while for thinness 1 it is a single vertex. This produces slightly bigger trees than for the thinness, which corresponds with the fact that the linear MIM-width is a lower bound for the thinness.
Regarding the pathwidth, instead, the minimum trees are smaller, which also corresponds with the fact that the pathwidth plus one is an upper bound for the thinness. Again, a theorem similar to Theorem 2 holds for pathwidth [6], but with subtrees instead of dangling trees.



$\mathrm{pw}(T) = 1$ $\mathrm{pw}(T) = 2$ $\mathrm{pw}(T) = 3$

2.3 Rooted Trees, k-critical Vertices and Labels

Our algorithm computing thinness will work on a rooted tree, processing it bottom-up. We will choose an arbitrary vertex r of the tree T and denote by T_r the tree rooted at r. During the bottom-up processing of T_r we will compute a label for various subtrees. The notion of a k-critical vertex is crucial for the definition of labels.

Definition 3 (Rooted complete subtree)**.** *We define the* rooted complete subtree $T_r[x]$ *of* T_r *as the subtree of* T_r *rooted at* x*.*

Definition 4 (k-critical vertex)**.** *Let T_r be a rooted tree. We call a vertex x in T_r* k-critical *if it has exactly two children v_1 and v_2 that have at least one child each, u_1 and u_2 respectively, such that* $\text{thin}(T_r[u_1]) = \text{thin}(T_r[u_2]) = k$.

It could be the case that x has a k-neighbor index greater than 2 in T_r, even if in $T_r[x]$ it is equal to 2.

Lemma 4. *If T_r has thinness k, then it has at most one k-critical vertex.*

Proof. For a contradiction, let x and x' be two k-critical vertices in T_r. There are then four vertices, v_1, v_2, v'_1, v'_2, the two k-neighbors of x and the two neighbors of x' respectively, such that there exist dangling trees $T\langle v_1, u_1\rangle$, $T\langle v_2, u_2\rangle$, $T\langle v'_1, u'_1\rangle$, $T\langle v'_2, u'_2\rangle$ that all have thinness k. If x and x' have a descendant/ancestor relationship in T_r, then assume without loss of generality that x' is v_1 or a descendant of v_1. Let p be the direct parent of x'. Then, as $T\langle v_2, u_2\rangle$ is a subtree of $T\langle x', p\rangle$, and $T\langle v'_1, u'_1\rangle$ and $T\langle v'_2, u'_2\rangle$ are disjoint trees in different neighbors of x', then $D_{T_r}(x', k) = 3$, and, by Theorem 2, T_r should have thinness greater or equal than $k + 1$. Otherwise, all the dangling trees are disjoint, thus $D_T(x, k) = D_T(x', k) = 3$ and we arrive to the same conclusion. □

Definition 5 (label and last type)**.** *Let T_r be a rooted tree with* $\text{thin}(T_r) = k$. *Then* $\text{label}(T_r)$ *consists of a list of decreasing numbers, $(a_1, ..., a_p)$, where $a_1 = k$, and* $\text{lastType}(T_r)$ *is an integer between 0 and 3 which will have the information of where in the tree an a_p-critical vertex lies, if it exists at all, according to the following list. If $p = 1$ then we define the label as being* simple*, otherwise it is* complex*. The label(T_r) and lastType(T_r) are 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:*** *In this type of trees, r is a leaf, i.e. T_r is a singleton, or all children of r are leaves. $label(T_r) = (1)$ and $lastType(T_r) = 0$.*
- ***Type 1:*** *Trees of this type are not Type 0 trees, and have no k-critical vertex. $label(T_r) = (k)$ and $lastType(T_r) = 1$.*
- ***Type 2:*** *r is the k-critical vertex of trees of this type. $label(T_r) = (k)$ and $lastType(T_r) = 2$.*
- ***Type 3:*** *In these trees a child of r is k-critical. $label(T_r) = (k)$ and $lastType(T_r) = 3$.*
- ***Type 4:*** *There is a k-critical vertex u_k in T_r that is neither r nor a child of r. Let w be the parent of u_k. Then $label(T_r) = k \oplus label(T_r \setminus T_r[w])$, and $lastType(T_r) = lastType(T_r \setminus T_r[w])$.*

In type 4 we note that $\text{thin}(T_r \setminus T_r[w]) < k$ since otherwise u_k would have three k-neighbors (two children in the tree and also its parent) and by Theorem 2 we would then have $\text{thin}(T_r) = k + 1$. Therefore, all numbers in $label(T_r \setminus T_r[w])$ are smaller than k and a complex label is a list of decreasing numbers. We also note that each element of a complex label corresponds to the thinness of some subtree of T_r, with the first element being the thinness of T_r. We now give a Proposition that for any vertex x in T_r will be used to compute $label(T_r[x])$ and $lastType(T_r[x])$ based on the labels of the subtrees rooted at the children and grand-children of x. The subroutine underlying this Proposition will be used when reaching vertex x in the bottom-up processing of T_r.
Let us first prove the following Lemma, which will be useful when proving Proposition 1:

Lemma 5. *Let T_r be a rooted tree on r, and x be a vertex of T_r. Let x be a vertex of T_r with children $Child(x)$. Define $k = \max_{v \in Child(x)}\{\text{thin}(T_r[v])\}$, meaning, the maximum thinness of a subtree rooted on a child of x. If no descendant of x is k-critical, then no descendant v of x has $D_{T_r[x]}(v, k) \geq 3$.*

Proof. For there to be some v such that $D(v, k) \geq 3$, v should have at least 3 k-neighbors. As v has no more than one parent, at least 2 of those k-neighbors must be children of v. And v cannot have more than 2 children k-neighbors, as this would mean that $D_{T_r[v]}(v, k) \geq 3$. This would then mean that $\text{thin}(T_r[v]) \geq k+1$. But then some child subtree of x has thinness greater or equal than $k+1$, which cannot happen by the definition of k. So v has exactly 2 k-children, and so is a k-critical vertex. But this also cannot happen by the statement, which means that v cannot exist. □

Proposition 1. *Let x be a vertex of T_r with children $Child(x)$, and assume we are given $label(T_r[v])$ and $lastType(T_r[v])$ for all $v \in Child(x)$. Let k the maximum thinness of a child subtree of x as in Lemma 5. Also define $N_k = \{v \in Child(x) \mid \text{thin}(T_r[v]) = k\}$, meaning, the set of children for whom the subtrees rooted at them have thinness k. Denote $N_k = \{v_1, ..., v_q\}$, $l_i = label(T_r[v_i])$, and $t_i = lastType(T_r[v_i])$. Define $d_k = D_{T_r[x]}(x, k)$ by noting that $d_k = |\{v_i \in N_k \mid v_i$ has child u_j with $\text{thin}(T_r[u_j]) = k\}|$. Given this information, we can find $label(T_r[x])$ and $lastType(T_r[x])$ as follows.*

- ***Case 0:*** *x is a leaf or all children of x are leaves, and then $label(T_r[x]) = (1)$ and $lastType(T_r[x]) = 0$.*
- ***Case 1:*** *x is not a leaf and not all children of x are leaves, and for every $v_i \in N_k$, l_i is simple and t_i is equal to 1 or 0, and $d_k \leq 1$. Then, $label(T_r[x]) = (k)$, and $lastType(T_r[x]) = 1$.*

- ***Case 2:*** *For every $v_i \in N_k$, l_i is simple and t_i is equal to 1 or 0, but $d_k = 2$. Then, $label(T_r[x]) = (k)$ and $lastType(T_r[x]) = 2$.*
- ***Case 3:*** *For every $v_i \in N_k$, l_i is simple and t_i is equal to 1 or 0, but $d_k \geq 3$. Then, $label(T_r[x]) = (k+1)$, and $lastType(T_r[x]) = 1$.*
- ***Case 4:*** *$|N_k| \geq 2$ and for some $v_i \in N_k$, either l_i is a complex label, or t_i is equal to either 2 or 3. Then, $label(T_r[x]) = (k+1)$, and $label(T_r[x]) = 1$.*
- ***Case 5:*** *$|N_k| = 1$, l_1 is a simple label and t_1 is equal to 2. Then, $label(T_r[x]) = (k)$, and $lastType(T_r[x]) = 3$.*
- ***Case 6:*** *$|N_k| = 1$, l_1 is either complex or t_1 is equal to 3, and $k \notin label(T_r[x] \setminus T_r[w])$, where w is the parent of the k-critical vertex in $T_r[v_1]$. Then, $label(T_r[x]) = k \oplus label(T_r[x] \setminus T_r[w])$, and $lastType(T_r[x]) = lastType(T_r[x] \setminus T_r[w])$.*
- ***Case 7:*** *$|N_k| = 1$, l_1 is either complex or t_1 is equal to 3, and $k \in label(T_r[x] \setminus T_r[w])$, where w is the parent of the k-critical vertex in $T_r[v_1]$. Then, $label(T_r[x]) = k+1$, and $lastType(T_r[x]) = 1$.*

Proof. We will assume for the time being that this Proposition covers all possible trees, as we will prove later. Now, we will prove that each case assigns values to label and lastType according to Definition 5. We will use k as it was used in the Proposition, meaning to signify the maximum thinness of a child subtree of x.
We go case by case, proving that the thinness and the type of $T_r[x]$ are correctly determined to show that label and lastType are assigned the correct values.

- **Case 0:** As x is a leaf or all children of x are leaves, $T_r[x]$ is a type 0 tree. As the assignments in Case 0 are the same as the assignments for type 0 trees, they are correct.
- **Case 1:** We must prove two things: that $\text{thin}(T_r[x]) = k$, to show that $label(T_r[x])$ is being assigned the correct value; and that $T_r[x]$ is a type 1 tree, to show that $lastType(T_r[x])$ is being assigned the correct value and that $T_r[x]$ has a simple label.
  - $\text{thin}(T_r[x]) = k$: All children subtrees of x which have thinness k are type 0 or 1 trees, meaning that they do not have any k-critical vertex. As seen in Lemma 5, this means that no descendant v of x has $D_{T_r[x]}(v, k) \geq 3$. On the other hand, as $d_k \leq 1$, $D_{T_r[x]}(x, k) = 1$, and so there is no vertex $u \in T_r[x]$ such that $D_{T_r[x]}(u, k) \geq 3$. By Theorem 2, $\text{thin}(T_r[x]) \leq k$. Given that there is at least one child v of x such that $\text{thin}(T_r[v]) = k$, $\text{thin}(T_r[x]) \geq k$, and so $\text{thin}(T_r[x]) = k$.

- $T_r[x]$ is a type 1 tree: First of all, $T_r[x]$ is not a type 0 tree, as x is not a leaf and not all children of x are leaves. We must now show that it has no k-critical vertex. We know that there are no k-critical vertices in any child subtree of x, so we must only show that x is not a k-critical vertex, and this is given by the fact that $d_k \leq 1$.

By Definition 5, this tree should have $label(T_r[x]) = (k)$ and $lastType(T_r[x]) = 1$, which are the values that the Proposition assigns.

- **Case 2:**
  - $\text{thin}(T_r[x]) = k$: Again by Lemma 5, x has no descendant k-saturated in $T_r[x]$. And, as $d_k = 2$, $D_{T_r[x]}(x, k) = 2$, and so x is not k-saturated in $T_r[x]$. As there is no vertex k-saturated in $T_r[x]$, and as in Case 1, $\text{thin}(T_r[x]) \geq k$, $\text{thin}(T_r[x]) = k$.
  - $T_r[x]$ is a type 2 tree: As $d_k = 2$, x is a k-critical vertex. There are no other k-critical vertices in $T_r[x]$, and so this is a type 2 tree.
- **Case 3:**
  - $\text{thin}(T_r[x]) = k + 1$: $d_k \geq 3$, which means that x is a k-saturated vertex in $T_r[x]$. So we have $\text{thin}(T_r[x]) \geq k+1$. By Lemma 5, no vertex v other than x is k-saturated in $T_r[x]$, which in turn means that v is not $k+1$-saturated. As there is no child subtree of x with thinness greater than k, x is also not $k+1$-saturated, and so $\text{thin}(T_r[x]) \leq k + 1$ by Theorem 2. This means that $\text{thin}(T_r[x]) = k + 1$.
  - $T_r[x]$ is a type 1 tree: As $d_k \geq 3$, x has at least three grandchildren, which means that it is not a type 0 tree. We must then show that there is no $k + 1$-critical vertex in $T_r[x]$. As no strict subtree of $T_r[x]$ has thinness equal to $k + 1$, there cannot be a vertex with two $k + 1$-neighbors in $T_r[x]$, and so by definition there is no $k + 1$-critical vertex in $T_r[x]$.
- **Case 4:**
  - $\text{thin}(T_r[x]) = k + 1$: There is some $v_i \in N_k$ for which either l_i is a complex label, meaning that $T_r[v_i]$ is a type 4 tree; or t_i is either 2 or 3, meaning that $T_r[v_i]$ is a type 2 or 3 tree. In both cases we know that $T_r[v_i]$ has a k-critical vertex. Let u be the k-critical vertex in $T_r[v_i]$, and let p be the parent of u in $T_r[x]$. As $|N_k| \geq 2$, there exists a vertex $v_j \in N_k$, $v_j \neq v_i$. As $\text{thin}(T_r[v_j]) = k$ and $T_r[v_j]$ is a subtree of $T_r[x]\langle u, p\rangle$, $\text{thin}(T_r[x]\langle u, p\rangle) \geq k$. This, compounded with the fact that u is k-critical, means that u is k-saturated in $T_r[x]$, and so $\text{thin}(T_r[x]) \geq k + 1$.

As no strict subtree of $T_r[x]$ has thinness greater than k, we know that there is no descendant $k+1$-critical vertex, and so by Lemma 5, no descendant of x is $k+1$-saturated. For the same reason, x is not $k+1$-saturated, as it has no $k+1$-neighbors in $T_r[x]$. This means that $\text{thin}(T_r[x]) \leq k+1$. With both results we arrive at the conclusion that $\text{thin}(T_r[x]) = k+1$.

- $T_r[x]$ is a type 1 tree: As seen earlier, $T_r[x]$ has no $k+1$-critical vertex. Also, it is not a type 0 tree, as it contains some k-critical vertex in some of its subtrees. We see then that the definition of type 1 trees matches.

- **Case 5:**

  - $\text{thin}(T_r[x]) = k$: x has only one child v such that $\text{thin}(T_r[v]) = k$. Also, $T_r[v]$ is a type 2 tree, and so v is its only k-critical vertex. All other vertices in $T_r[v]$ have less than two children k-neighbors. Also, there is no other k-critical vertex in $T_r[x]$, as all other children subtrees of x have thinness lower than k, and x has only one child subtree with thinness k. As shown in the proof of Lemma 5, if a vertex u is k-saturated, it must have at least two child k-neighbors. This means that the only vertex that could be k-saturated in $T_r[x]$ is v. But for this to be true, x should be a k-neighbor of v, which means that there should be another neighbor u of x for which $\text{thin}(T_r[u]) = k$. As v is the only vertex in N_k, this cannot happen, and so there is no k-saturated vertex in $T_r[x]$. This gives us $\text{thin}(T_r[x]) \leq k$. As there is some subtree of $T_r[x]$ with thinness equal to k, $\text{thin}(T_r[x]) \geq k$, and so $\text{thin}(T_r[x]) = k$.
  - $T_r[x]$ is a type 3 tree: In this case, v, which is a child of x, is k-critical, and so this tree matches the definition of a type 3 tree.

- **Case 6:**

  - $\text{thin}(T_r[x]) = k$: We start by noting that $\text{thin}(T_r[x]) \geq k$, as it contains at least one subtree with thinness k, namely $T_r[v_1]$.

    As $T_r[v_1]$ is a type 3 or 4 tree, it has a k-critical vertex $u \neq v_1$. With w being the parent of u, we know that $k \notin \mathit{label}(T_r[x] \setminus T_r[w])$, and so $\text{thin}(T_r[x] \setminus T_r[w]) \neq k$. Also, we know that there is no vertex in $T_r[x]$ other than u with two or more child k-neighbors. This is because, by Lemma 4, u is the only k-critical vertex in $T_r[v_1]$, and there are no other child subtrees of x with thinness greater or equal to k, which means that x has no more than one k-neighbor, and that there are no vertices in other subtrees which have child k-neighbors. This means that there is no vertex in $T_r[x] \setminus T_r[w]$ which has more than one child k-neighbor, which in turn means that there

is no k-saturated vertex in $T_r[x] \setminus T_r[w]$. So, $\text{thin}(T_r[x] \setminus T_r[w]) \leq k$. This leaves us with $\text{thin}(T_r[x] \setminus T_r[w]) < k$, as it must be different from k as seen earlier.

As seen in the proof of Lemma 5, the only possible k-saturated vertex in $T_r[x]$ is the one with at least two children k-neighbors, meaning u. But as $\text{thin}(T_r[x] \setminus T_r[w]) < k$, w is not a k-neighbor of u, which tells us that u has only two k-neighbors and so is not k-saturated. We arrive at the conclusion that $\text{thin}(T_r[x]) \leq k$, and so $\text{thin}(T_r[x]) = k$.

- $T_r[x]$ is a type 4 tree: The k-critical vertex in $T_r[x]$ is u, which is a descendant of v_1, and so is neither x nor a child of x. This is the definition of a type 4 tree.

To finish this case, we note that the label and the lastType of $T_r[x]$ are assigned exactly as written in the definition of a type 4 tree.

- **Case 7:**
  - $\text{thin}(T_r[x]) = k+1$: As $T_r[v_1]$ is a type 3 or 4 tree, it has a k-critical vertex $u \neq v_1$. Also, as $k \in \mathit{label}(T_r[x] \setminus T_r[w])$, $\text{thin}(T_r[x] \setminus T_r[w]) \geq k$. This means that u is k-saturated, and so $\text{thin}(T_r[x]) \geq k+1$.

    Let us see that there is no $k+1$-saturated vertex in $T_r[x]$ to see that $\text{thin}(T_r[x]) = k+1$. For there to be a $k+1$-saturated vertex y, it must have at least two child $k+1$-neighbors, which means that there must be two vertices z_1 and z_2 such that $\text{thin}(T_r[z_1]) = \text{thin}(T_r[z_2]) = k+1$, with $T_r[z_1]$ and $T_r[z_2]$ disjoint subtrees. As k is the maximum thinness of a subtree of $T_r[x]$, this cannot happen.
  - $T_r[x]$ is a type 1 tree: First we note that $T_r[x]$ is not a type 0 tree, as it has at least one k-neighbor. We also note that, as seen earlier, $T_r[x]$ has no $k+1$-critical vertices, because no vertex has two child $k+1$-neighbors. This then follows the definition of a type 1 tree.

We have gone through each case of the Proposition showing that the assignments of labels and lastTypes correspond to the tree types in Definition 5. Now we will continue by showing that every possible tree is covered by exactly one of these eight cases. Observe the decision tree in Figure 2.1. We will show that cases of Proposition 1 correspond to cases in the decision tree, and with that prove that exactly one case applies to every rooted tree. In the following case analysis, k represents the maximum thinness of a child subtree of $T_r[x]$, as in the Proposition.

- **Case 0:** This case is reached if and only if x is a leaf or all its children are leaves, and so corresponds to Case 0 in Proposition 1.

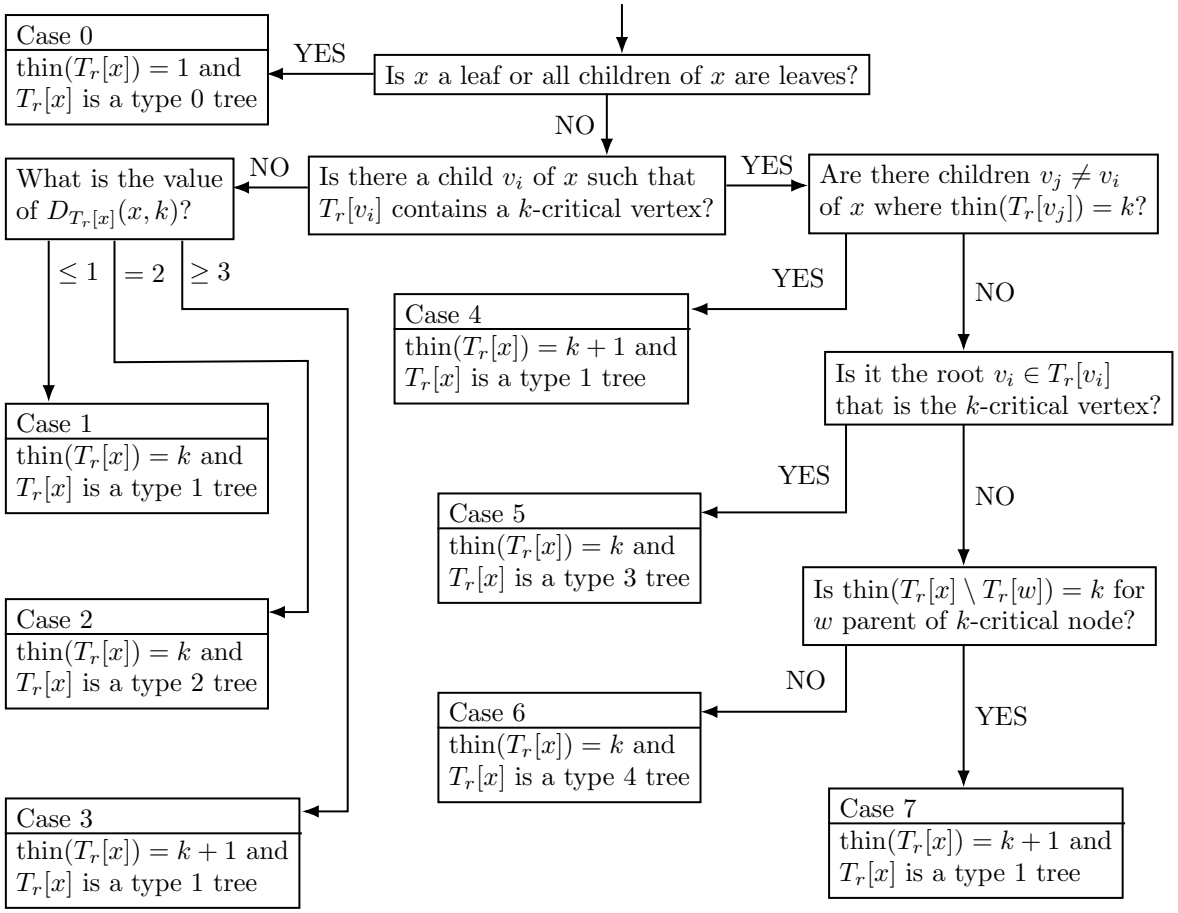


Figure 2.1: A decision tree corresponding to the case analysis of Proposition 1. Here, T_r and k are defined as in the Proposition.

- **Case 1:** Firstly, this case is not reached if x is a leaf or all its children are leaves, which matches the first condition of Case 1. Secondly, it is reached if and only if $T_r[x]$ has no child subtree with a k-critical vertex, and $D_{T_r[x]}(x,k) \leq 1$. We know $T_r[v_i]$ does not have a k-critical vertex if and only if it has thinness lower than k or is a type 0 or type 1 tree by Definition 5, which is the second condition of Case 1. And the third condition, $d_k \leq 1$, is given by the fact that $D_{T_r[x]}(x,k) \leq 1$.

- **Case 2 and 3:** This proofs are very similar as the one for Case 1, and so they are omitted.

- **Case 4:** In this case of the Proposition, the condition dictates that some child subtree $T_r[v_i]$ with thinness k must be of type 2, 3, or 4. This happens if and only if $T_r[v_i]$ has a k-critical vertex, which is one of the conditions checked in the decision tree. The following condition is true if and only if $|N_k| \geq 2$, which is the other condition in the Proposition. Finally, given that in a Case 4 tree there is at least one k-critical vertex, x has at least one grandchild, and so the statement "Is x a leaf or all children of x are leaves?" is not true. This last remark will be applied to all following cases, because all of their corresponding trees will contain at least one k-critical vertex.

- **Case 5:** $|N_k| = 1$ if and only if there are no two vertices $v_i \neq v_j$ such that $\text{thin}(T_r[v_i]) = \text{thin}(T_r[v_j]) = k$, and so this two conditions correspond to each other. And $t_1 = 2$ if and only if $T_r[v_1]$ is a type 2 tree, which means that v_1 is the k-critical vertex in $T_r[v_1]$, which in turn is the last condition in the decision tree before reaching Case 5. Of course, Case 5 trees do have a k-critical vertex, because v_1 is a k-critical vertex, so that condition in the decision tree is correct as well.

- **Case 6 and 7:** The only difference between these cases is if k is either included or not in $label(T_r[x] \setminus T_r[w])$, which is the last condition in the decision tree. In both cases $|N_k| = 1$, which as we have seen is already checked in the decision tree, and also $T_r[v_1]$ is a type 3 or 4 tree, which happens if and only if $T_r[v_1]$ has a k-critical vertex different from v_1. This is the contrary to the condition to enter Case 5 in the decision tree. Lastly, $|N_k| = 1$, which as seen is also checked in the decision tree.

This proves that every possible tree is covered by the cases in Proposition 1. Combined with the fact that each case assigns the label and lastType following Definition 5, we have shown that the thinness of every possible tree can be calculated as the first element of its label following this case analysis. □

2.4 Computing Thinness of Trees and Finding a Consistent Solution

The subroutine underlying Proposition 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 and lastTypes of subtrees $T_r[x]$. However, in cases 6 and 7 we need the label and lastType of $T_r[x] \setminus T_r[w]$, which is not a complete subtree rooted at any vertex of T_r. Note that the label and lastType of $T_r[x] \setminus T_r[w]$ are again given by a recursive call to Proposition 1, and then the label is stored as a suffix of the complex label of $T_r[x]$, and the lastType is the same. We will compute these labels and lastTypes by iteratively calling Proposition 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 thinness k and a k-critical vertex u_k that is neither x nor a child of x, and the recursive definition gives $label(T_r[x]) = k \oplus label(T_r[x] \setminus T_r[w])$ for w the parent of u_k. Unravelling this recursive definition, we have the following:

Definition 6. *Let x be a vertex in T_r, and let $h = |label(T_r[x])|$. Denote $label(T_r[x]) = (a_1, \ldots, a_h)$. Then $\omega(T_r[x])$ is a list $(\omega_1, \ldots, \omega_h)$ of vertices in $T_r[x]$ in which $\omega_h = x$, and every other ω_i with $1 \leq i < h$ is the parent of the a_i-critical vertex in $T_r[x] \setminus (T_r[\omega_1] \cup \cdots \cup T_r[\omega_{i-1}])$. We will use $\omega(T_r[x])_i$ to denote the element number i of the list, or simply use ω_i when it is clear which tree we are referring to.*

Now, in the first level of a recursive call to Proposition 1 the role of $T_r[x]$ is taken by $T_r[x] \setminus T_r[\omega_1]$, and in the next level it is taken by $(T_r[x] \setminus T_r[\omega_1]) \setminus T_r[\omega_2]$ etc. The following definition gives a shorthand for denoting these trees.

Definition 7. *Let x be a vertex in T_r, and $label(T_r[x]) = (a_1, a_2, \ldots, a_p)$. For any non-negative integer s, the tree $T_r[x, s]$ is the subtree of $T_r[x]$ obtained by removing all trees $T_r[\omega_i]$ from $T_r[x]$, where $a_i \geq s$. In other words, if q is such that $a_q \geq s > a_{q+1}$, then $T_r[x, s] = T_r[x] \setminus (T_r[\omega_1] \cup T_r[\omega_2] \cup \cdots \cup T_r[\omega_q])$.*

To ease the following proofs, we will define the following list of vertices.

Definition 8. *Let x be a vertex in T_r. Let $h = |label(T_r[x])|$. Denote $label(T_r[x]) = (a_1, \ldots, a_h)$. Then $\varphi(T_r[x])$ is the list of vertices $(\varphi_1, \ldots, \varphi_{h-1})$, where $\varphi_i \in T_r[x]$ is the a_i-critical vertex in $T_r[x]$. We will use $\varphi(T_r[x])_i$ to denote the element number i of the list, or simply use φ_i when it is clear which tree we are referring to.*

Note that each ω_i is the parent of the corresponding φ_i, except when $i = |label(T_r[x])|$, in which case there is no φ_i.

Lemma 6. *Some important properties of $T_r[x,s]$ are the following. Let $T_r[x,s]$, $label(T_r[x,s])$, and q be as in the definition. Then*

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

Proof. These follow from the definitions. We will show a proof for the last one:
Backward direction: Let $s = a_q$ for some $1 \leq q \leq p$. Then $T_r[x,s+1] = T_r[x] \setminus (T_r[\omega_1] \cup \cdots \cup T_r[\omega_{q-1}])$ and $T_r[x,s] = T_r[x] \setminus (T_r[\omega_1] \cup \cdots \cup T_r[\omega_q])$. These two trees are clearly different.
Forward direction: Let $T_r[x,s] = T_r[x] \setminus (T_r[\omega_1] \setminus \cdots \setminus T_r[\omega_q])$ and $T_r[x,s+1] = T_r[x] \setminus (T_r[\omega_1] \cup \cdots \cup T_r[\omega_{q_0}])$ with $q_0 < q$ and $q_0 > a_q$ (because numbers in a label are strictly descending). $a_q < s+1$ and $a_q \geq s$, ergo $a_q = s$. □

Lemma 7. *Let $x \in V(T_r)$, and let u be a child of x in T_r. Let $s \in \mathbb{N}$ such that $T_r[x,s]$ and $T_r[u,s]$ are not empty, that is to say, s is greater both than the last element of $label(T_r[x])$ and than the last element of $label(T_r[u])$. Let $T_s^* = T_r[x,s] \cap T_r[u]$. Then $T_s^* = T_r[u,s]$, meaning, the child subtree of $T_r[x,s]$ rooted at u is equal to $T_r[u,s]$.*

Proof. We will do induction on s, and prove that $T_s^* = T_r[u,s]$. Let $a_1 = label(T_r[x])_1$.

- *Base case, $s > a_1$:* In this case, $T_r[x,s] = T_r[x]$. Note that $label(T_r[x])_1 \geq label(T_r[u])_1$, because $T_r[u]$ is a subtree of $T_r[x]$ and so its thinness is smaller or equal to $\text{thin}(T_r[x])$. So $T_r[u,s] = T_r[u]$. Then

$$T_s^* = T_r[x,s] \cap T_r[u] = T_r[x] \cap T_r[u] = T_r[u] = T_r[u,s]$$

- *Inductive step, $s \leq a_1$:* We will assume the statement holds for $s+1$, meaning that $T_{s+1}^* = T_r[u,s+1]$. Note that this implies that $T_r[u,s+1]$ is a child subtree of $T_r[x,s+1]$. We have four possible cases here:

  1. $s \notin label(T_r[x])$, $s \notin label(T_r[u])$: This means by Lemma 6 that $T_r[x,s] = T_r[x,s+1]$, and that $T_r[u,s] = T_r[u,s+1]$. This in turn shows that

$$T_s^* = T_r[x,s] \cap T_r[u] = T_r[x,s+1] \cap T_r[u] = T_{s+1}^* = T_r[u,s+1] = T_r[u,s]$$

2. $s \in label(T_r[x])$, $s \notin label(T_r[u])$: This means that $T_r[u, s+1]$ has thinness smaller than s, while $T_r[x, s+1]$ has thinness s. This also means that $T_r[u, s] = T_r[u, s+1]$, which gives us $T^*_{s+1} = T_r[u, s]$. Let i be the position of s in $label(T_r[x])$. As s is not the last element of $label(T_r[x])$, $v = \varphi(T_r[x])_i$ exists as a vertex in $T_r[x]$.

   The fact that $T_r[u, s+1]$ has thinness smaller than s tells us in particular that it has no s-critical vertices. Also, by inductive hypothesis, $T_r[u, s+1]$ is a subtree of $T_r[x, s+1]$. These two facts together show that v, which is an s-critical vertex in $T_r[x, s+1]$, cannot be in $T_r[u, s+1]$. As v is outside $T_r[u, s+1] = T_r[u, s]$, also $w = \omega(T_r[x])_i$ is. As $T_r[x, s]$ is not empty, w is not x but instead is a descendant of x in $T_r[x, s+1]$. The only vertices removed from $T_r[x, s+1]$ to get $T_r[x, s]$ are w and its descendants, which are not descendants of u and so are not in T^*_{s+1}. We conclude then that $T^*_s = T^*_{s+1} = T_r[u, s]$.
3. $s \notin label(T_r[x])$, $s \in label(T_r[u])$: This means that $T_r[u, s+1]$ has thinness s, while $T_r[x, s+1]$ has thinness smaller than s. But this cannot happen, as by inductive hypothesis $T_r[u, s+1]$ is a subtree of $T_r[x, s+1]$, and so must have thinness smaller or equal to that of $T_r[x, s+1]$. This case then is nonexistent.
4. $s \in label(T_r[x])$, $s \in label(T_r[u])$: As s is not the last element of $label(T_r[x])$, $label(T_r[x, s+1])$ has at least two elements, and so $T_r[x, s+1]$ is a type 4 tree. This means that there is an s-critical vertex $v \in \varphi(T_r[x])$ in $T_r[x, s+1]$. By a similar argument, there is also an s-critical vertex $v' \in \varphi(T_r[u])$ in $T_r[u, s+1]$.

   By Lemma 4 there can only be one s-critical vertex in $T_r[x, s+1]$ and in $T_r[u, s+1]$, because their thinness is s. By the inductive hypothesis, $T_r[u, s+1]$ is a subtree of $T_r[x, s+1]$, and so their s-critical vertices are the same, in other words, $v' = v$. As both s-critical vertices are the same, the parents are also the same, and so the subtree U removed when going from $T_r[x, s+1]$ to $T_r[x, s]$ is the same as the one removed when going from $T_r[u, s+1]$ to $T_r[u, s]$. So we have

$$\begin{aligned} T_r[u, s] &= T_r[u, s+1] \setminus U \\ &= T^*_{s+1} \setminus U \\ &= T_r[x, s+1] \cap T_r[u] \setminus U \\ &= T_r[x, s] \cap T_r[u] \\ &= T^*_s \end{aligned}$$

□

Corollary 2.3. *Let T_r, x and u be as in Lemma 7. If $s \in label(T_r[x])$ and $s \in label(T_r[u])$, $T^*_{s+1} = T_r[u, s+1]$.*

Proof. That s be both in $label(T_r[x])$ and in $label(T_r[u])$ means that $s+1$ is bigger than the last element of both labels, and so the conditions for Lemma 7 are satisfied for $s+1$. □

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 $s > \text{thin}(T_r[x])$ since by Lemma 6.1 we then have $T_r[x, s] = T_r[x]$. This poses a problem: we cannot know which subtrees to calculate the labels for until we have finished with all subtrees. To solve this, each iteration of the loop will correctly compute the label of another subtree called $T_{union}[x, s]$, which is not always equal to $T_r[x]$. Nonetheless, for $s > \text{thin}(T_r[x])$, the equality $T_{union}[x, s] = T_r[x, s] = T_r[x]$ will hold, and so calculating labels for these subtrees will aid in calculating labels for bigger subtrees.

Definition 9. *Let x be a vertex 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 thinness by rooting it at an arbitrary vertex r, and computing labels by processing T_r bottom-up. The answer is given by the first element of $label(T_r[r])$, which by definition is equal to $\text{thin}(T)$. At a vertex x of T_r which is a leaf or all their children are leaves we initialize by $label(T_r[x]) \leftarrow (1)$, according to Definition 5. When reaching a higher vertex x we compute the label of $T_r[x]$ by calling function MakeLabel(T_r, x) defined in Algorithm 2.

Lemma 8. *Given labels at descendants of vertex x in T_r,* MakeLabel(T_r, x) *computes $label(T_r[x])$ as the value of cur_label and $lastType(T_r[x])$ as the value of cur_type.*

Proof. Assume that x has children $v_1, \ldots, v_d$, and denote their set of labels as $L = \{l_1, \ldots, l_d\}$. MakeLabel keeps variables *cur_label* and *cur_type* that are 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 $s > \text{thin}(T_r[x])$, we have $T_{union}[x, s] = T_r[x]$.
Claim: At the end of the iteration number s of the for loop the value of *cur_label* is equal to $label(T_{union}[x, s+1])$, and *cur_type* is equal to $lastType(T_{union}[x, s+1])$.

- *Base case:* We have to show that before the first iteration of the loop we have $cur\_label = label(T_{union}[x, 1])$ and $cur\_type = lastType(T_{union}[x, 1])$. As no $l_i \in L$ has 0 as an element by Definition 5, then $T_{union}[x, 1]$ is by definition the

Algorithm 2 Compute $cur\_label = label(T_r[x])$ and $cur\_type = lastType(T_r[x])$

function MAKELABEL(T_r: tree, x : vertex)
  $cur\_label \leftarrow (1)$
  $cur\_type \leftarrow 1$
  $\{v_1, \ldots, v_d\} \leftarrow$ children of x in T_r
  for $s \leftarrow 1, \max_{i=1}^{d}\{$ first element of $label(T_r[v_i])\}$ **do**
    $\{l'_1, \ldots, l'_d\} \leftarrow \{label(T_r[v_i, s+1]) \mid 1 \leq i \leq d\}$
    $\{t'_1, \ldots, t'_d\} \leftarrow \{lastType(T_r[v_i, s+1]) \mid 1 \leq i \leq d\}$
    $N_s \leftarrow \{v_i \mid 1 \leq i \leq d, s \in l'_i\}$
    $d_s \leftarrow |\{v_i \mid v_i \in N_s, v_i$ has a child u_j s.t. $s \in label(T_r[u_j, s+1])\}|$
    if $N_s \neq \emptyset$ **then**
      $case \leftarrow$ the case from Prop. 1 applying to $s, \{l'_1, \ldots, l'_d\}, \{t'_1, \ldots, t'_d\}, N_s$ and d_s
      $cur\_label \leftarrow$ as given by $case$ in Prop. 1 ($s \oplus cur\_label$ if Case 6)
      $cur\_type \leftarrow$ as given by $case$ in Prop. 1
    end if
  end for
end function

singleton vertex x and by Proposition 1 the label of this tree is (1), which is what cur_label is initialized to, and its last type is 1, which is what cur_type is initialized to.

- *Inductive step:* We assume $cur\_label = label(T_{union}[x, s])$ at the start of iteration number s 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 Lemma 6.2, $label(T_r[v_i, s{+}1])$ is contained in $label(T_r[v_i])$ for all i, therefore $l'_1, \ldots, l'_d$ are trivial to compute. Next, the $\{t'_1, \ldots, t'_d\}$ are calculated, following the same strategy as for the labels. Afterwards N_s is set as the set of all children of x whose labels contain s, and d_s as the number of vertices in N_s that themselves have children whose labels contain s. Let us first look at what happens when $N_s = \emptyset$:

  By Lemma 6.5, 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 is empty, then $T_{union}[x, s+1] = T_{union}[x, s]$, and from the inductive hypothesis, $label(T_{union}[x, s{+}1]) = cur\_label$, and indeed when $N_s = \emptyset$ 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

Proposition 1	for loop iteration s	Explanation
$T_r[x]$	$T_{union}[x, s+1]$	Tree needing label
k	s	Max thinness of children
$T_r[v_1], \ldots, T_r[v_d]$	$T_r[v_i, s], \ldots, T_r[v_d, s]$	Subtrees of children
$l_1, \ldots, l_d$	$l'_1, \ldots, l'_d$	Child labels
$t_1, \ldots, t_d$	$t'_1, \ldots, t'_d$	Child lastTypes
N_k	N_s	Children with max thinness
d_k	d_s	Root k-component index
$label(T_r[x] \setminus T_r[w])$	cur_label	This is also $label(T_{union}[x, s+1] \setminus T_r[w])$

Table 2.1: Correspondence between values computed in Proposition 1 and variables in MakeLabel.

Proposition 1 to compute $label(T_{union}[x, s+1])$ and argue first that the variables used in that call correspond to the variables used in Proposition 1 to compute $label(T_r[x])$. The correspondence is given in Table 2.1. Most of these are just observations: $T_{union}[x, s+1]$ corresponds to $T_r[x]$ in Proposition 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 $\{label(T_r[v]) | v \in Child\}$ in Proposition 1, and similarly for the t'_i with the t_i. N_s is defined in the algorithm so that it corresponds to N_k in Proposition 1. Since $|N_s| > 0$, some v_i has s in its label l'_i. By Lemma 6.3 and 6.4, we can infer that s is the maximum thinness of all $T_r[v_i, s+1]$, therefore s corresponds to k in Proposition 1.

It takes a bit more effort to show that d_s computed in iteration s of the for loop corresponds to $d_k = D_{T_r[x]}(x, k)$ in Proposition 1 – meaning we need to show that $d_s = D_{T_{union}[x,s+1]}(x, s)$. Consider v_i, a child of x. In accordance with MakeLabel we say that v_i contributes to d_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 d_s if and only if v_i is an s-neighbor of x in $T_{union}[x, s+1]$. Observe that by Lemma 6.4, $\text{thin}(T_r[v_i, s+1]) = \text{thin}(T_r[u_j, s+1]) = s$ 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 $\text{thin}(T_r[u_j, s+1]) < s$, 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 $\text{thin}(T_r[v_i, s+1]) < s$ and no subtree of $T_r[v_i, s+1]$ can have thinness 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 d_s), then, by Corollary 2.3, $T_r[u_j, s+1]$ must be a child subtree of $T_r[v_i, s+1]$. This means that $T_r[u_j, s+1] \subseteq T_{union}[x, s+1]$, 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 d_s, so $d_s = D_{T_{union}[x,s+1]}(x, s)$.

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 vertex – then *cur_label* has the value corresponding to $label(T_r[x] \setminus T_r[w])$ in Proposition 1, which would in this case be $label(T_{union}[x, s+1] \setminus T_r[w])$. We know, by definition of label and Lemma 6.5, that $T_r[v_i, s+1] \setminus T_r[v_i, s] = T_r[w] \cap T_r[v_i, s+1]$. As $T_r[w]$ is contained entirely in $T_r[v_i]$, we have that

$$T_r[w] \cap T_r[v_i, s+1] = T_r[w] \cap T_{union}[x, s+1]$$

But since $|N_s| = 1$, for every $j \neq i$, $T_r[v_j, s+1] \setminus T_r[v_j, s] = \emptyset$. Therefore

$$T_{union}[x, s+1] \setminus T_{union}[x, s] = T_r[w] \cap T_{union}[x, s+1]$$

which in turn gives us

$$T_{union}[x, s+1] \setminus (T_r[w] \cap T_{union}[x, s+1]) = T_{union}[x, s+1] \setminus T_r[w] = T_{union}[x, s]$$

But by the induction assumption,

$$cur\_label = label(T_{union}[x, s]) = label(T_{union}[x, s] \setminus T_r[w])$$

Thus *cur_label* corresponds to $label(T_r[x] \setminus T_r[w])$ in Proposition 1.

We have now argued for all the correspondences in Table 2.1. By that, we conclude from Proposition 1 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 $k \geq \text{thin}(T_r[v_i])$ 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 3. *Given any tree T,* $\text{thin}(T)$ *can be computed in $\mathcal{O}(n \log(n))$-time.*

Proof. We find $\text{thin}(T)$ by bottom-up processing of T_r and returning the first element of $label(T_r)$. After correctly initializing at leaves and vertices whose children are all leaves, we make a call to MakeLabel for each of the remaining vertices. Correctness follows by Lemma 8 and induction on the structure of the rooted tree. We will now show that each call runs in $\mathcal{O}(\log(n))$ time to prove the time complexity.
Let m be the biggest number in any label of children of x, which is $\mathcal{O}(\log(n))$ by Corollary 2.1. For every integer s from 1 to m, 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 $\mathcal{O}(\log(n))$. 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 $\mathcal{O}(\log(n)(a+b))$. As the sum of the number of children and grandchildren over all vertices of T_r is $\mathcal{O}(n)$ we conclude that the total runtime to compute thin(T) is $\mathcal{O}(n\log(n))$. □

Theorem 4. *A consistent solution using* thin(T) *classes can be found in* $\mathcal{O}(n\log(n))$*-time.*

Proof. Given T we first run the algorithm computing thin(T) to find the label and lastType of every full rooted subtree in T_r. We give a recursive algorithm that uses these labels in tandem with ConsistentLayout presented in the Path Layout Lemma. We call it on a rooted tree where labels of all subtrees are known. For simplicity we call this rooted tree T_r even though in recursive calls this is not the original root r and tree T. The layout algorithm goes as follows:

0. Calculate $critical(x)$ as the k'-critical vertex, if it exists, of each subtree $T_r[x]$ of T_r, where k' is the thinness of $T_r[x]$.
1. Let thin$(T_r) = k$ and use $critical(r)$ to find a path P in T_r such that all trees in $T_r \setminus N[P]$ have thinness lower than k. The path depends on the type of T_r as explained in detail below.
2. Repeat this algorithm starting from step 1 recursively on every rooted tree in $T_r \setminus N[P]$ to obtain consistent solutions; to this end, we need the correct label for every vertex in these trees.
3. Call ConsistentLayout on T_r, P and the solutions provided in step 2.

First, let us see how to calculate the critical vertices of Step 0. To calculate $critical(v)$ for some subtree $T_r[v]$, checking $critical(u)$ for each child of v is sufficient. We use k' to denote the thinness of $T_r[v]$.

- *Type 0 and 1 trees*: These do not contain any k'-critical vertex, so there is no $critical(v)$.
- *Type 2 trees*: v is the k'-critical vertex, so $critical(v) = v$
- *Type 3 trees*: We know that some child u of v is the root of a type 2 tree. We check every child of v to find the only child subtree $T_r[u]$ with thinness k' that is a type 2 tree, and we set $critical(v) = critical(u)$.

- *Type 4 trees*: Some child subtree will have thinness k' and will also have a k'-critical vertex, by definition of a type 4 tree. We can then treat it as a type 3 tree and check the lastTypes of the child subtrees with thinness k' to see which one has a k'-critical vertex.

This procedure can be done in $\mathcal{O}(n)$ by traversing the whole tree in a bottom-up fashion.
Let us proceed with the next steps. Algorithm 3 shows an implementation in pseudocode of steps 1 to 3.

Algorithm 3 Consistent solution of a given tree and its labels, lastTypes, and critical vertices.

```
1: function ConsistentSolution(T_r, {label(x)}, {lastType(x)}, {critical(x)})
2:     k ← label(T_r[r])_1
3:     P ← FindPath(T_r)
4:     for x ∈ T_r do
5:         Update label(x) if necessary, as determined by lastType(T_r[r]).
6:     end for
7:     solutions ← ∅
8:     for each connected component T' of T_r[r] \ N[P] do
9:         Add to solutions the result of
10:            ConsistentSolution(T', {label(x)}, {lastType(x)}, {critical(x)}).
11:    end for
12:    return ConsistentLayout(T_r, P, solutions)
13: end function
```

Every tree in the forest $T \setminus N[P]$ is equal to a dangling tree $T\langle v, u\rangle$ where v is a neighbor of some $x \in P$.
We observe that if $\text{thin}(T) = k$, then by definition $\text{thin}(T\langle v, u\rangle) = k$ if and only if v is a k-neighbor of x. It follows that every tree in $T \setminus N[P]$ has thinness at most $k - 1$ if and only if no vertex in P has a k-neighbor that is not in P. We use this fact to show that, for every type of tree, the function FindPath can effectively find a satisfying path in the following way:

- *Type 0 trees*: Choose $P = (r)$. Since $|T \setminus N[r]| = 0$ in these trees, this must be a satisfying path.

- *Type 1 trees*: These trees contain no k-critical vertices, which by definition means that for any vertex x in T_r, at most one of its children is a k-neighbor of x. Choose P to start at the root r, and as long as the last vertex in P has a k-neighbor v, v is appended to P. This set of vertices is obviously a path

in T_r. No vertex in P can possibly have a k-neighbor outside of P, therefore all connected components of $T \setminus N[P]$ have thinness lower or equal to $k-1$. Furthermore, all components of $T \setminus N[P]$ are full rooted subtrees of T_r and so the labels are already known.

- *Type 2 trees*: In these trees the root r is k-critical. We look at the trees rooted in the two k-neighbors of r, $T_r[v_1]$ and $T_r[v_2]$. By Remark 4 these must both be Type 1 trees, and so we find paths P_1, P_2 in $T_r[v_1]$ and $T_r[v_2]$ respectively, as described above. Gluing these paths together at r we get a satisfying path for T_r, and we still have correct labels for the components $T \setminus N[P]$.

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

- *Type 4 trees*: In these trees, T_r contains precisely one vertex $w \neq r$ such that w is the parent of a k-critical vertex, x. This w is simply the parent of $critical(r)$.

  Clearly, the tree $T_r[w]$ is a type 3 tree with thinness k. We find a path P that is satisfying in $T_r[w]$ as described above. w is still not a k-neighbor of x, therefore P is a satisfying path.

There is a peculiarity in case 4. Namely, we have one connected component of $T \setminus N[P]$ that is not a full rooted subtree of T_r, that is $T_r \setminus T_r[w]$. Thus, for every ancestor y of w, $T_r[y] \setminus T_r[w]$ is not a full rooted subtree either, and we need to update the labels of these trees to maintain correctness.
As each $T_r[y]$ contains the k-critical vertex x, it has thinness greater or equal to k. Also, as $T_r[y]$ is a subtree of T_r, it has thinness lower or equal to k. These two facts tell us that $\text{thin}(T_r[y]) = k$, which means that k is the first element of its label. Also, they are all type 4 trees, because they each have the parent w of x as a descendant. This means that $w = \omega(T_r[y])_1$ for each y. With this, we see that $T_r[y] \setminus T_r[w]$ is by definition equal to $T_r[y,k]$, whose label is the result of removing the first position from $label(T_r[y])$. We then update every $label(y)$ to reflect this change before proceeding with the recursive call.
To prove that these operations are performed a number of times proportional to $\mathcal{O}(n \log n)$, we have to make a more in-depth analysis:
Firstly, each vertex is added to a single path, so the operation of adding a vertex to a path is made a number of times proportional to n. Every time we add a vertex, we check its neighbors. In a tree, the number of edges is $n-1$, so again, this operation is

done a linear number of times. The only thing remaining is to prove the complexity of the label update operations.
Each single operation is of constant complexity when using appropiate data structures, as it consists of simply removing the first element of a list. Each update operation removes one element from a label, and as seen earlier, each label has at most $\text{thin}(T) \in \mathcal{O}(\log n)$ elements. As we do not remove the same element from a label twice, we do at most $\mathcal{O}(n \log n)$ label updates in the whole algorithm.
So, even though any single call to FindPath could possibly have a bigger complexity than $\mathcal{O}(\log n)$, in total the amount of operations executed inside this function, along with the label updates, has complexity $\mathcal{O}(n \log n)$.
We follow the proof by noting that each recursive call to ConsistentSolution reduces the thinness of the remaining trees by one. We then see that the function ConsistentLayout is called with a tree containing v at most $thin(T_r)$ times for any vertex v in T_r. Remembering that $\text{thin}(T_r) \in \mathcal{O}(\log n)$, and that ConsistentLayout has linear time complexity, we conclude that the operations performed in the calls to ConsistentLayout also have a total time complexity of $\mathcal{O}(n \log n)$. □

Corollary 4.1. *For any given tree T,* $\text{thin}(T) - \text{lmimw}(T) \leq 1$.

Proof. In this section, for a given tree with $\text{thin}(T) = k$, we showed a way to divide it into paths so that we could apply the *Path Layout Lemma* recursively (up to k levels in total) and construct an optimal consistent solution. That is to say we can always show a path P in T such that all connected components in $T \setminus N[P]$ have thinness at most $k - 1$ and we can find paths of this form recursively. Similarly, on the paper presented in [9], for a given tree T such that $\text{lmimw}(T) = l$, it is shown a way to divide the tree into paths (up to $l + 1$ levels in total). The construction of the solution for linear MIM-width is of course different but we are interested here in the fact that we can guarantee that T can be divided this way. As a result, considering that we have a tree divided recursively into paths for up to $l + 1$ levels, we can simply apply the *Path Layout Lemma* presented on this work from that construction and get a consistent solution using at most $l + 1$ partitions. Since it was proven in [1] that $\text{lmimw}(G) \leq \text{thin}(G)$ for any graph G, we can say that for a tree T the thinness and the linear MIM-width differ at most in one. □

CHAPTER 3

Thinness of Graph Classes

We present here some bounds for the thinness of trees and grids, we compute exactly the thinness of complete m-ary trees and crowns, and we use our algorithm to find new bounds for the thinness of general graphs.

3.1 Thinness of Complete m-ary Trees

The *height* of a rooted tree is the maximum number of edges in a simple path from the root to a leaf. Notice that a rooted tree of height h has $h+1$ *levels* of vertices, being the root the only vertex at level 1, and saying that a vertex has level $t+1$ if and only if its parent has level t. It was proven in [1] that for a fixed value m, the thinness of a complete m-ary tree on n vertices is $\Theta(\log(n))$, and it was also proven in [12] that the thinness of a non-trivial tree is less than or equal to its height; but, until now, it was an open problem to compute the exact thinness of a complete m-ary tree.

Theorem 5. *Let be m an integer number greater or equal than 3 and T a complete m-ary tree with height h, then* $\text{thin}(T) = \lceil \frac{h+1}{2} \rceil$.

Proof. For a given $m \geq 3$ we proceed by induction on h.

- *Base case:*

  For $h = 0$ and $h = 1$, since they are interval graphs, $\text{thin}(T) = 1$. Then the condition $\text{thin}(T) = \lceil \frac{h+1}{2} \rceil$ holds.

- *Inductive step:*

  Assume the property holds for all m-ary trees of height less or equal than a given $h \geq 1$. We want to see the property also holds for $h+1$.

  Let T be a complete m-ary tree with height $h+1$. Let x be the only vertex at level 1; $v_1, v_2, \ldots, v_m$ the level 2 vertices; and $t_{i,1}, t_{i,2}, ..., t_{i,m}$ the vertices adjacent to v_i at level 3. Then by the inductive hypothesis we can say that $\text{thin}(T\langle v_i, t_{i,j}\rangle) = \lceil \frac{h}{2} \rceil$.

  Since there are at least three vertices v_i, each one with at least one dangling tree $T\langle v_i, t_{i,j}\rangle$, $D(x, \lceil \frac{h}{2} \rceil) \geq 3$. Due to Theorem 2, we have that $\text{thin}(T) \geq \lceil \frac{h}{2} \rceil + 1$. On the other hand, applying the Path Layout Lemma with $P = (x)$ shows us that $\text{thin}(T) \leq \lceil \frac{h}{2} \rceil + 1$. As $\lceil \frac{h}{2} \rceil + 1 = \lceil \frac{h+2}{2} \rceil$, we have that $\lceil \frac{h+2}{2} \rceil \leq \text{thin}(T) \leq \lceil \frac{h+2}{2} \rceil$, and so the property holds for $h+1$.

□

Theorem 6. *Let T be a complete binary tree with height h, then* $\text{thin}(T) = \lceil \frac{h+1}{3} \rceil$.

Proof. First we prove that $\text{thin}(T) \geq \lceil \frac{h+1}{3} \rceil$. We proceed by induction on h.

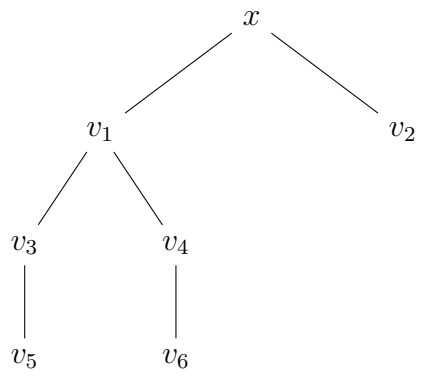
- *Base case:*

  For $h \in \{0, 1, 2\}$, since T is an interval graph, $\text{thin}(T) = 1 \geq \lceil \frac{h+1}{3} \rceil$.

- *Inductive step:*

  Suppose the property holds for all trees of height less of equal than h (with $h \geq 2$), we want to see that the property is true for any given tree T of height $h+1$.

  Let T be a complete binary tree with height $h+1$. We name some of the vertices of the first 4 levels as described in the following figure:



From the inductive hypothesis we know that $\text{thin}(T\langle v_3, v_5\rangle) \geq \lceil \frac{h-2+1}{3} \rceil = \lceil \frac{h-1}{3} \rceil$ and $\text{thin}(T\langle v_4, v_6\rangle) \geq \lceil \frac{h-1}{3} \rceil$. In addition to that, since that $T\langle v_3, v_5\rangle$ is isomorphic to an induced subgraph of $T\langle x, v_2\rangle$, we can also say that $\text{thin}(T\langle x, v_2\rangle) \geq \text{thin}(T\langle v_3, v_5\rangle) \geq \lceil \frac{h-1}{3} \rceil$.

These conditions imply that $D(v_1, \lceil \frac{h-1}{3} \rceil) \geq 3$, so due to Theorem 2 we can say that $\text{thin}(T) \geq \lceil \frac{h-1}{3} \rceil + 1 = \lceil \frac{h-1+3}{3} \rceil = \lceil \frac{h+2}{3} \rceil$. Then the property holds for $h+1$.

Now we show that $\text{thin}(T) \leq \lceil \frac{h+1}{3} \rceil$. By proceed by an induction on h.

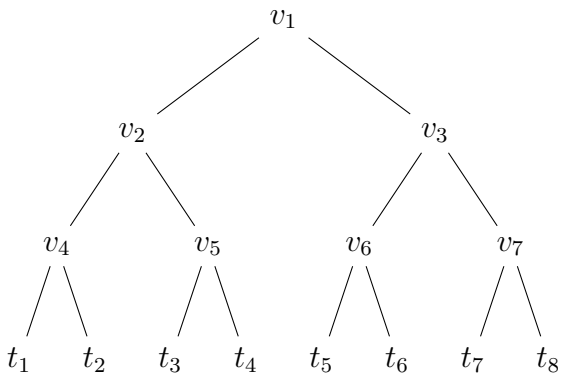
- *Base case:*

  For $h \in \{0, 1, 2\}$, since T is an interval graph, $\text{thin}(T) = 1$; then the property is true since $1 \leq \lceil \frac{h+1}{3} \rceil$.

- *Inductive step:*

  Suppose the property holds for some height $h \geq 2$ and for all heights lesser than h; we want to see that the property is also true for $h+1$.

  Let T be a complete binary graph with height $h+1$. Let us call the vertices of the first 4 levels of T as it is shown in the next figure:



  Due to the inductive hypothesis we know that $\text{thin}(T\langle v_i, t_j\rangle) \leq \lceil \frac{h-1}{3} \rceil$. This means that for all trees $T\langle v_i, t_j\rangle$ there exists a consistent ordering and partition of its vertices in no more than $\lceil \frac{h-1}{3} \rceil$ classes. We can then apply the Path Layout Lemma with $P = (v_2, v_1, v_3)$ to deduce that $\text{thin}(T) \leq \lceil \frac{h-1}{3} \rceil + 1 = \lceil \frac{h-1+3}{3} \rceil = \lceil \frac{h+2}{3} \rceil$.

We proved $\lceil \frac{h+1}{3} \rceil \leq \text{thin}(T) \leq \lceil \frac{h+1}{3} \rceil$. □

3.2 Improving the Diameter Bound for Trees

The *diameter* of a tree is the maximum number of edges in a simple path joining two vertices. From [12] it can be shown by construction that it is always possible to have a consistent solution for a given tree with approximately $\frac{diameter}{2}$ classes. In this subsection we are going to adjust this bound.

Theorem 7 (Upper Bound Diameter Theorem)**.** *Let T be a tree and d its diameter, then* $\text{thin}(T) \leq \lceil \frac{d+1}{4} \rceil$. *Moreover, if the maximum degree of a vertex in T is at most 3, then* $\text{thin}(T) \leq \lceil \frac{d+3}{6} \rceil$.

Proof. Let m be the maximum degree among all vertices of T. If d is even, consider the complete m-ary tree T' with height $h' = \frac{d}{2}$. Since the diameter of T' is equal to $2h' = d$ and it is an complete m-ary tree, and every vertex of T has degree at most m, T is an induced subgraph of T'. This implies that $\text{thin}(T) \leq \text{thin}(T')$. Now from the result proven in Theorem 5 we can say that

$$\text{thin}(T') = \left\lceil \frac{h'+1}{2} \right\rceil = \left\lceil \frac{\frac{d}{2}+1}{2} \right\rceil = \left\lceil \frac{d+2}{4} \right\rceil$$

By transitivity, if d is even, $\text{thin}(T) \leq \lceil \frac{d+2}{4} \rceil$.
If d is odd, consider the tree T'' obtained by joining the roots of two complete $(m-1)$-ary trees of height $h'' = \frac{d-1}{2}$ by an edge (notice that every vertex that is not a leaf has degree m in T''). The diameter of T'' is equal to $2h'' + 1 = d$. Let P be a maximum length path in T, and uv the middle edge of that path. Every leaf w of T such that u is not in the path between w and v is at distance at most h'' from v, otherwise there will be a path of length greater than d in T. Symmetrically, every leaf w of T such that v is not in the path between w and u is at distance at most h'' from u. This and the way of building T'' shows that T is an induced subgraph of T'' in this case. This implies that $\text{thin}(T) \leq \text{thin}(T'')$. Now from the Path Layout Lemma applied to the path uv, and the result proven in Theorem 5, we can say that

$$\text{thin}(T'') \leq \left\lceil \frac{h''-1}{2} \right\rceil + 1 = \left\lceil \frac{\frac{d-1}{2}-1}{2} \right\rceil + 1 = \left\lceil \frac{d-3}{4} \right\rceil + 1 = \left\lceil \frac{d+1}{4} \right\rceil$$

By transitivity, if d is odd, $\text{thin}(T) \leq \lceil \frac{d+1}{4} \rceil$. It is easy to see that we can combine the even and odd case and say that for every tree T with diameter d, $\text{thin}(T) \leq \lceil \frac{d+1}{4} \rceil$. For $m \leq 3$ and d odd, we can use Theorem 6 instead of Theorem 5, obtaining $\text{thin}(T) \leq \left\lceil \frac{\frac{d-1}{2}-1}{3} \right\rceil + 1 = \lceil \frac{d+3}{6} \rceil$. For d even, we make a similar construction by joining the roots of three complete binary trees to a new vertex x, and applying the Path Layout Lemma to the path x, $\text{thin}(T) \leq \lceil \frac{d+4}{6} \rceil$. Again, we can combine the even

and odd case and say that for every tree T with diameter d and maximum degree at most 3, $\text{thin}(T) \leq \left\lceil \frac{d+3}{6} \right\rceil$. □

Corollary 7.1. *The thinness of a tree T of n vertices and diameter d is at most* $\min(\left\lceil \frac{d+1}{4} \right\rceil, log_3(n+2))$.

Corollary 2.2 establishes an upper bound for the thinness of a graph in terms of the number of leaves. Let us call an *almost-leaf* a vertex which is not a leaf that has at most one neighbor that is not a leaf. Using some ideas from [3], we can prove this other bound, useful for trees with a big number of leaves but a small number of internal vertices of degree greater than two after trimming the leaves.

Theorem 8. *Let T be a tree with t almost-leaves, $t \geq 2$. Then* $\text{thin}(T) \leq t - 1$.

Proof. We start by trimming all the leaves of T, obtaining a tree T'. The leaves of T' are the almost-leaves of T. We then root T' at a leaf and start a new class of the partition, containing the root. If a vertex has one child, then it is assigned to the same class as its parent. If it has more than one child, then one child is assigned to the same class as its parent, and the other children are assigned to a new class each. So, we have at most $t - 1$ classes.
Now we order the vertices of T' by postorder (meaning, setting the children of a vertex to be all smaller than the vertex), and we add the trimmed leaves adjacent to a vertex v right before v and in its same class.
We will now show that the order and the partition are consistent. Suppose $x < y < z$, with x, y in the same class of the partition and $(z, x) \in E(T)$. Notice that z cannot be a leaf of T, since, by the way of defining the order, if w is a leaf then its only neighbor is greater than it. So z is a vertex of T'. If x is a leaf of T, then y is also a leaf of T adjacent to z, since we added the trimmed leaves adjacent to a vertex v right before v. If x is not a leaf in T, then z is the parent of x in T', since we have ordered T' by postorder. By the way of defining the ordering and the classes, either x and z are in the same class or x is the greatest vertex in his class, contradicting the existence of y; moreover, no vertex between x and z in the order of $V(T')$ belongs to the same class as x, so y must be a leaf of T adjacent to z, otherwise the vertex adjacent to y in T has to be in the same class and between y and z. This completes the proof of consistency. □

3.3 Thinness of CR_n

Recall that the *crown graph* CR_n (also known as Hiraguchi graph) is the graph on $2n$ vertices obtained from a complete bipartite graph $K_{n,n}$ by removing a perfect matching.

It was proven in [4] that $\text{thin}(CR_n) \geq \frac{n}{2}$; here we will show the exact thinness and a consistent ordering and partition of the vertices for CR_n.

Theorem 9. *Thinness*$(CR_n) = \max(1, n-1)$

Proof. The vertices of the CR_n graph are defined as $V = \{x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_n\}$, and the edges as $E = \{(x_i, y_j) : 1 \leq i, j \leq n, i \neq j\}$.
Some other definitions we are going to use:

- $X := \{x_1, x_2, \ldots, x_n\}$ and $Y := \{y_1, y_2, \ldots, y_n\}$.
- We define $mirror(v)$ as y_i if $v = x_i$ or as x_i if $v = y_i$.
- Given an order of the vertices, we say v is a *little* vertex if $v < mirror(v)$.
- Similarly, for a set of vertices A, we define *Littles(A)* as the set of vertices in the graph that belong to A and are *little* vertices.
- Given an order and a partition of the vertices, and a triple (r, s, t), we say they *"break the consistency"* meaning that $r < s < t$ and r and s belong to the same class and there is and edge between r and t but there is no edge between s and t.

Claim 9.1. If x_i and x_j (or y_i and y_j) with $i \neq j$ are both *little* vertices, then they cannot belong to the same class in a consistent solution.

Proof. Suppose, without loss of generality, that $x_i < x_j$. If x_i and x_j belonged to the same class, then the triple (x_i, x_j, y_j) would break the consistency. □

Notice that, since for each pair of vertices (x_i, y_i) there is exactly one *little* vertex, by the pigeonhole principle, X or Y should contain at least half of the *little* vertices in the whole graph. This implies that $\max(|Little(X)|, |Little(Y)|) \geq \lceil \frac{n}{2} \rceil$. Then, because of Claim 9.1, in a consistent solution there are at least $\lceil \frac{n}{2} \rceil$ *little* vertices all in different classes, and so $\text{thin}(CR_n) \geq \lceil \frac{n}{2} \rceil$. This is the bound we knew for now, which was proved in [4].

Claim 9.2. If, in a consistent solution, $x \in X$ and $y \in Y$ belong to the same class and $x < y$, then y must be the last vertex or the penultimate vertex of the order in Y.

Proof. Suppose the contrary, then there exist two distinct vertices y' and y'' in Y such that $y < \min(y', y'')$. Clearly one of y' or y'' is not $mirror(x)$, let us assume that $y'' \neq mirror(x)$. But then (x, y, y'') would break the consistency. □

Claim 9.3. If, in a consistent solution, $x \in X$ and $y \in Y$ with $x < y$ belong to the same class and y is the penultimate vertex in Y, then the last vertex in Y must be $y' = mirror(x)$.

Proof. Let us call y_{last} to the last vertex in Y. Suppose $y_{last} \neq mirror(x)$. Then the triple (x, y, y_{last}) would be breaking the consistency. □

Claim 9.4. Given $y, y' \in Y$, $x, x' \in X$, $y \neq y'$ and $x \neq x'$, such that, in a consistent solution:

- x and y belong to the same class;
- x' and y' belong to the same class;
- $x < y$;
- $x' < y'$;

then at least one of $\{y, y'\}$ is not a *little* vertex.

Proof. Without loss of generality let us assume that $y < y'$. By Claim 9.2 we know that y must be the last vertex or the penultimate vertex of the order in Y; but since $y < y'$, we can say that y and y' are respectively the penultimate and last vertex of the order in Y. Now by Claim 9.3 we can say that y' and $mirror(x)$ must be the same vertex. And since $x < y'$, this implies that y' is not a *little* vertex. □

We are now going to show that in a consistent solution, the number of classes containing a *little* vertex is at least $n - 1$. In other words, that at most one class can contain two little vertices, one of X and one of Y, since by Claim 9.1 no class can contain more than one little vertex of either X or Y.

Suppose, on the contrary, that there are 4 distinct little vertices x, x', y and y' such as $\{x, x'\} \in X$ and $\{y, y'\} \in Y$, x and y belong to the same class, x' and y' belong to the same class.
By Claim 9.4, it is neither possible that $x < y$ and $x' < y'$, nor that $x > y$ and $x' > y'$. Suppose then, without loss of generality, that $x < y$ and $x' > y'$. By Claim 9.2, x' and y can only be the last or penultimate vertices in X and Y, respectively. Without loss of generality, let us assume $x' > y$. Notice as $mirror(x') > x' > y$ the vertex y cannot be the last vertex in Y; implying that y is the penultimate vertex in Y. Let us call $\tilde{y}$ the last vertex in Y. We can say $\tilde{y} = mirror(x')$, since it is the only vertex in Y which is greater than y, which is the penultimate vertex in Y. However, by Claim 9.3, $\tilde{y} = mirror(x)$; then $x' = x$, a contradiction.

By the arguments above, there is at most one pair of *little* vertices (x, y) such that they belong to the same class. Then there must be at least

$$|Littles(X)| + |Littles(Y)| - 1 = n - 1$$

different classes containing the little vertices, which implies $\text{thin}(CR_n) \geq n - 1$.

Now let us see that $n - 1$ classes are enough for a consistent solution to the CR_n graph with $n > 1$.
If n is even, we define the order of the vertices σ and the partition S according to Algorithm 4. Similarly, if n is odd, we define the order of the vertices σ and the partition S according to Algorithm 5. The only difference of the *odd* case is how we assign the last vertices, but the rest of the algorithm is exactly the same. Notice the algorithms output the ordering and a list of classes containing the set of vertices corresponding to each class.
We can show that the partition and ordering are consistent by seeing that all triples of vertices satisfy the consistency condition. Given a triple (r, s, t) such as $r < s < t$ we can see all possibilities:

- $r = x_i$: If r and s belong to different classes then the consistency is already satisfied. The vertex s belongs to the same class of r only when $s = x_{i+1}$; and, by construction, $t = y_i$ or $t = y_{i+1+c}$ or $t = x_{i+1+c}$ with $c \geq 1$.
  - $t = y_i$: $(x_i, y_i) \notin E$, so (r, s, t) satisfies the consistency.
  - $t = y_{i+1+c}$: $(x_i, y_{i+1+c}) \in E$ but also $(x_{i+1}, y_{i+1+c}) \in E$, so (r, s, t) satisfies the consistency.
  - $t = x_{i+1+c}$: $(x_i, x_{i+1+c}) \notin E$, so (r, s, t) satisfies the consistency.
- $r = y_i$: If r and s belong to different classes then the consistency is already satisfied. The vertex s belongs to the same class of r only when $s = y_{i-1}$; and, by construction, $t = x_{i+c}$ or $t = y_{i+c}$ with $c \geq 1$.
  - $t = x_{i+c}$: $(y_{i-1}, x_{i+c}) \in E$ but also $(y_i, x_{i+c}) \in E$, so (r, s, t) satisfies the consistency.
  - $t = y_{i+c}$: $(y_{i-1}, x_{i+c}) \notin E$, so (r, s, t) satisfies the consistency.

It is easy to see that the assignment done for the last vertices holds the consistency. Since in the *main loop* of the algorithm we are assigning 4 vertices to 2 new classes;

- and in the *even* case we are using one extra class for the last 4 vertices;
- and in the *odd* case we are not using new classes for the last 2 vertices;

we can say that both algorithms use exactly $n - 1$ classes.
We proved $\text{thin}(CR_n) \geq n-1$ and $\text{thin}(CR_n) \leq n-1$, which concludes the proof. □

Algorithm 4 Consistent layout of a CR_n graph for even $n > 1$.

function ConsistentLayoutCR($x_1, x_2, \ldots x_n$: vertices in X, $y_1, y_2, \ldots y_n$: vertices in Y)
  $\sigma \leftarrow \emptyset$ ▷ the ordering starts out empty
  for odd $i \in [1..n-3)$ **do**
    ▷ in each iteration we set the order and classes of 4 vertices
    $\sigma \leftarrow \sigma \oplus (x_i, y_{i+1}, x_{i+1}, y_i)$
    $C_i \leftarrow \{x_i, x_{i+1}\}$
    $C_{i+1} \leftarrow \{y_{i+1}, y_i\}$
  end for
  $\sigma \leftarrow \sigma \oplus (x_{n-1}, y_n, y_{n-1}, x_n)$ ▷ We set the order and class of the remaining vertices.
  $C_{n-1} \leftarrow \{x_{n-1}, y_n, x_n, y_{n-1}\}$
  $S \leftarrow \{C_1, \ldots, C_{n-1}\}$
return σ, S
end function

Algorithm 5 Consistent layout of a CR_n graph for odd $n > 1$.

function ConsistentLayoutCR($x_1, x_2, \ldots x_n$: vertices in X, $y_1, y_2, \ldots y_n$: vertices in Y)
  $\sigma \leftarrow \emptyset$ ▷ the ordering starts out empty
  for odd $i \in [1..n-2)$ **do**
    ▷ in each iteration we set the order and classes of 4 vertices
    $\sigma \leftarrow \sigma \oplus (x_i, y_{i+1}, x_{i+1}, y_i)$
    $C_i \leftarrow \{x_i, x_{i+1}\}$
    $C_{i+1} \leftarrow \{y_{i+1}, y_i\}$
  end for
  $\sigma \leftarrow \sigma \oplus (x_n, y_n)$ ▷ We set the order and class of the remaining vertices.
  $C_{n-1} \leftarrow C_{n-1} \cup \{x_n\}$
  $C_{n-2} \leftarrow C_{n-2} \cup \{y_n\}$
  $S \leftarrow \{C_1, ..., C_{n-1}\}$
return σ, S
end function

3.4 Bound for GR_r

Recall that, for a positive integer r, the $(r \times r)$-*grid*, noted GR_r, is the graph whose vertex set is $\{(i,j) : 1 \leq i,j \leq r\}$ and whose edge set is $\{((i,j),(k,l)) : |i-k|+|j-l| = 1, \text{ where } 1 \leq i,j,k,l \leq r\}$.
We know from [10] that $\frac{n}{4} \leq \text{thin}(GR_r) \leq r+1$; here we are slightly adjusting the upper bound and we are showing an additional lemma.
We will start by showing the following lemma, so as to ease the next proofs.

Definition 10. *Given a consistent solution Ω with ordering σ and partition S of the vertices in a graph G, we define* $\text{prev}_\Omega(v_i)$ *for $v_i \in V(G)$ as the last vertex v_j before v_i in σ such that v_j and v_i belong to the same class in S, if such vertex exists. We use* $\text{prev}(v_i)$ *when the solution can be implied from the context.*

Lemma 9. *An ordering $\sigma = (v_1, \ldots, v_n)$ and partition S of the vertices of a graph G are consistent if and only if for every vertex v_i in σ either:*

- *there is no $j < i$ such that v_j belongs to the same class as v_i in S; or*
- *let $v_j = \text{prev}(v_i)$, and let H be the set of neighbors v_h of v_j such that $h > i$. Then every vertex in H is a neighbor of v_i.*

Proof. We will prove the lemma by proving both directions of the implication.
$\Longrightarrow$) Suppose we have a consistent ordering $\sigma = (v_1, \ldots, v_n)$ and partition S of the vertices of a graph G. We know by definition of consistency that for each triplet (r,s,t) such that $v_r < v_s < v_t$, if v_r and v_s belong to the same class in S and $(v_r, v_t) \in E(G)$, then $(v_s, v_t) \in E(G)$. Now suppose there exists a vertex v_i that contradicts the lemma. This means that there is a vertex $v_j = \text{prev}(v_i)$ and another vertex $v_h \in N[v_j]$ such that $i < h$ and $(v_i, v_h) \notin E(G)$. Notice this triple (v_j, v_i, v_h) is contradicting the consistency, since v_j and v_i both belong to the same class, (v_r, v_h) is an edge in G, and (v_i, v_h) is not an edge in G, so such a vertex cannot exist.
$\Longleftarrow$) Suppose we have an ordering $\sigma = \{v_1, \ldots, v_n\}$ and partition S of the vertices of a graph G and the conditions of this implication are valid for this solution. We must show that this solution is consistent. Suppose that this is not the case, meaning, there are three vertices (v_i, v_j, v_k), with $i < j < k$, such that v_i and v_j belong to the same class C in S, and v_k is a neighbor of v_i but not of v_j. Let $r < j$ such that v_r is the last member of C before v_j according to σ which is a neighbor of v_k. Let $s > r$ such that v_s is the first vertex in C after v_r. Note both these numbers exist in the range $[i,j]$, because v_i is a neighbor of v_k and v_j is not. Then, v_s is not a neighbor of v_k, as v_r is the last neighbor of v_k in C before v_j. Both v_r and v_s belong to the same class, and v_r is adjacent to v_k, while v_s is not. As $k > j \geq s$, and $v_r = \text{prev}(v_s)$, this means that the condition stated in the lemma is not met on v_j, which is a contradiction. □

Theorem 10 (GR_r bound)**.** *Let there be a consistent ordering σ and partition S of the vertices of GR_r using k classes such that all of the following are true:*

1. $(r,1) < (r,2) < \cdots < (r,r-1) < (r,r) < (r-1,r) < \cdots < (2,r) < (1,r)$.
2. $(r-1,r-1) < (r,r)$.
3. *If there is a vertex v in the same class C as (r,r) in S which is lower or equal to $(r,r-1)$ according to σ, then there is no neighbor w of v greater than $(r,r-1)$ in σ other than (r,r).*

Then GR_{r+3} is $(k+2)$-thin.

Proof. Given the consistent solution Ω with ordering σ and partition S into k classes of the vertices of GR_r such that its vertices satisfy the conditions in the statement, we will construct a solution Ω' with ordering σ' and partition S' into $k+2$ classes in a graph that has GR_{r+3} as an induced subgraph and show that Ω' is consistent. Since GR_{r+3} will be induced, the induced relative ordering and partition will also be consistent. Following Lemma 9, we will show that every vertex v in this construction meets that either $\text{prev}_{\Omega'}(v)$ does not exist, or every neighbor of $\text{prev}_{\Omega'}(v)$ greater than v is also a neighbor of v, and that will be enough to show that the solution is consistent.
The vertices of the form (i,j) with $1 \leq i,j \leq n$ will belong to the same classes in S' as in S, and their relative order in σ' will be the same as in σ. Regarding the vertices outside that square, $(r+1,r)$ and $(r,r+1)$ will belong to the same class as (r,r), and we will assign the rest of the vertices to two extra classes A and B. All the vertices of the forms $(i,r+3)$ and $(r+3,i)$ will belong to class A in S'. Also, if r is odd, all the vertices of the forms $(2i,r+2)$ and $(r+2,2i)$ will belong to class A, and if r is even, all the vertices of the forms $(2i-1,r+2)$ and $(r+2,2i-1)$ will belong to class A. All the remaining vertices will belong to class B.
The order σ' will be defined by the following pseudocode. Supposing σ and r are global variables, the method *extendOrder* will return σ' (notice the classes are already predefined). Here, we use $\sigma[v,u]$ to denote the contiguous subarray of σ which starts with v and ends with u. We use $\sigma[,u]$ to denote $\sigma[\sigma_0,u]$ and $\sigma[v,]$ to denote $\sigma[v,\sigma_{|\sigma|}]$. We use $successor_\sigma(v)$ to denote the vertex immediately after v in σ. We use $\oplus$ to denote the array concatenation operation, or the vertex appending operation, as appropriate. Lastly, given a vertex (i,j) of GR_{r+3}, $(i,j)+(x,y)$ corresponds to the vertex $(i+x,j+y)$.

```
1: function extendOrder()
2:     σ' ← [(r+2,1)]
3:     if r is odd then
```

```
4:          arrowRight((r + 3, 1))
5:          next ← (r, 2)
6:      else
7:          next ← (r, 1)
8:      end if
9:      horizontalArrows(σ_0, next)
10:     corner()
11:     nextInSigma ← verticalArrows((r, r), (r − 1, r))
12:     σ' ← σ' ⊕ σ[nextInSigma, ]
13: end function
14: function horizontalArrows(previous: vertex, next: vertex)
15:     while next ≠ (r, r + 1) do
16:         σ' ← σ' ⊕ σ[previous, next]
17:         arrowLeft(next + (1, 0))
18:         arrowRight(next + (3, 1))
19:         previous ← successor_σ(next)
20:         next ← next + (0, 2)
21:     end while
22: end function
23: function arrowLeft(from: vertex)
24:     σ' ← σ' ⊕ [from, from + (1, 1), from + (2, 0)]
25: end function
26: function arrowRight(from: vertex)
27:     σ' ← σ' ⊕ [from, from + (−1, 1), from + (−2, 0)]
28: end function
29: function corner()
30:     σ' ← σ' ⊕ (r + 1, r + 1)
31:     σ' ← σ' ⊕ (r, r + 1)
32:     σ' ← σ' ⊕ (r + 2, r + 2)
33:     σ' ← σ' ⊕ (r + 3, r + 1)
34:     σ' ← σ' ⊕ (r + 3, r + 2)
35:     σ' ← σ' ⊕ (r + 3, r + 3)
36:     σ' ← σ' ⊕ (r + 2, r + 3)
37:     σ' ← σ' ⊕ (r + 1, r + 2)
38:     σ' ← σ' ⊕ (r, r + 2)
39:     σ' ← σ' ⊕ (r + 1, r + 3)
40:     σ' ← σ' ⊕ (r, r + 3)
41:     σ' ← σ' ⊕ (r − 1, r + 2)
42:     σ' ← σ' ⊕ (r − 1, r + 1)
43: end function
```

44: **function** VERTICALARROWS(previous: vertex, next: vertex)
45: $\sigma' \leftarrow \sigma' \oplus \sigma[previous, next]$
46: $\sigma' \leftarrow \sigma' \oplus [(r-2, r+2), (r-1, r+3)]$
47: next $\leftarrow$ next $+ (-1, 0)$
48: **while** $next.first \geq 1$ **do**
49: $\sigma' \leftarrow \sigma' \oplus \sigma[previous, next]$
50: ARROWDOWN(next $+ (0, 1)$)
51: previous $\leftarrow successor_\sigma(next)$
52: ARROWUP(next $+ (1, 3)$)
53: next $\leftarrow$ next $+ (-2, 0)$
54: **end while**
55: **return** previous
56: **end function**
57: **function** ARROWUP(from: vertex)
58: $\sigma' \leftarrow \sigma' \oplus [\text{from}, \text{from} + (-1, 1), \text{from} + (0, 2)]$
59: **end function**
60: **function** ARROWDOWN(from: vertex)
61: $\sigma' \leftarrow \sigma' \oplus [\text{from}, \text{from} + (-1, -1), \text{from} + (0, -2)]$
62: **end function**

Now, we will go line by line showing that each vertex addition to σ' satisfies the condition on Lemma 9. When proving the correctness of a line, we will use v to refer to the vertex currently being added to σ'. We will use $v.1$ to denote the first coordinate of v, and $v.2$ to denote the second coordinate of v.

2: $\boldsymbol{\sigma' \leftarrow ((r+2, 1))}$
There is no $\text{prev}_{\Omega'}(v)$, so the condition is satisfied.

4: **arrowRight($(r+3, 1)$)**
Replacing in the function arrowRight the parameter from with v, the vertices added to σ' are $(r+3, 1), (r+2, 2), (r+1, 1)$. Notice that here, r is odd, so $(r+2, 1)$ belongs to class B.

- $(r+3, 1)$: This vertex belongs to class A, and so there is no $\text{prev}_{\Omega'}(v)$.
- $(r+2, 2)$: This vertex belongs to class A, as r is odd. $\text{prev}_{\Omega'}(v)$ is then $v' = (r+3, 1)$. The neighbors of v' are $(r+2, 1)$ and $(r+3, 2)$, which are also neighbors of v, so the condition is satisfied.
- $(r+1, 1)$: This vertex belongs to class B. $\text{prev}_{\Omega'}(v)$ is then $(r+2, 1)$. Its only neighbors other than v are $(r+3, 1)$ and $(r+2, 2)$, which are already in σ', so the condition is satisfied.

9: **horizontalArrows(σ_0, next)**
The internal while statement of horizontalArrows will stop only when next $= (r, r+1)$. When finishing one cycle, we add $(0, 2)$ to next, and so this while will stop only if the second coordinate of the original value of next has the same parity as $r+1$. If r is odd, we call this function with next $= (r, 2)$, and $r+1$ is even. If r is even, we call this function with next $= (r, 1)$, and $r+1$ is odd. We see then that the loop always terminates.

We will prove that horizontalArrows maintains consistency. For that, we will prove that before each cycle the following preconditions are met:

1. next $+ (2, 0)$ is the last vertex in A contained in σ'.
2. Either there is no vertex in B contained in σ', or the last vertex in B contained in σ' is next $+ (1, -1)$.
3. The vertex denoted by "previous" is in σ, and all vertices in $\sigma[, \text{previous}]$ except previous are already in σ'.
4. All vertices of the form $(r+i, j)$ are already in σ', with $1 \leq i \leq 3$ and $j <$ next.2.
5. All vertices in σ' belonging to a class in S are also in σ.
6. "next" is of the form (r, i), with $r - i$ odd.

First, we will see that the preconditions are met before entering the loop.

1. If r is odd, next $+ (2, 0) = (r+2, 2)$, which as seen in the justification of line 4 is the last vertex in A added to σ'. If r is even, next$+(2, 0) = (r+2, 1)$, which is the only vertex in σ', and also belongs to A, because it is of the form $(r+2, 2i-1)$.
2. If r is odd, as per the justification of line 4, the last vertex in B already in σ' is $(r+1, 1) =$ next $+ (1, -1)$. If r is even, there is no vertex belonging to B in σ' yet.
3. The vertex denoted by "previous" is the first vertex in σ, and there are no vertices belonging to σ in σ' yet.
4. If r is odd, next.2 $= 2$. Vertices $(r+1, 1)$, $(r+2, 1)$ and $(r+3, 1)$ were already added. If r is even, next.2 $= 1$, so there are no vertices of the form $(r+i, j)$.
5. All vertices added to σ' are of either class A or B, which are not in S.
6. If r is odd, next.2 $= 2$. If r is even, next.2 $= 1$. In both cases, $r -$ next.2 is odd and next.1 $= r$.

Now, we will go line by line in the while loop showing that consistency is preserved, assuming the preconditions described above are true.

28: $\boldsymbol{\sigma' \leftarrow \sigma' \oplus \sigma[previous, next]}$
We know that σ is a consistent ordering for the vertices in GR_r, and all vertices added to σ' belonging to a class in S also are in σ. For every vertex $u \in \sigma[previous, next]$, if $\text{prev}_\Omega(u)$ exists, then either:

- $\text{prev}_\Omega(u)$ is of the form (i, j) with $i, j < r$, which means it has no neighbors outside the ones in σ and so consistency is preserved, as it meets the condition in Lemma 9; or
- $\text{prev}_\Omega(u)$ is of the form (r, i) with $i < next.2$. By the preconditions stated, the only neighbor of $\text{prev}_\Omega(u)$ outside σ is already in σ', so again consistency is preserved.

29: **arrowLeft(next + (1, 0))**
Following the definition of "arrowLeft", the vertices added here are, in order:

1. $v = \text{next} + (1, 0)$: This vertex belongs to B, as its first coordinate is $r + 1$. By precondition 2, if $\text{prev}_{\Omega'}(v)$ exists, it is equal to $\text{next} + (1, -1) = v + (0, -1) = v'$. By precondition 4, $v' + (1, 0)$ and $v' + (0, -1)$ are already in σ'. Its only remaining neighbor apart from v is $v' + (-1, 0)$, which we will see is also in σ'.
   After line 28, all vertices belonging to $\sigma[, next]$ are in σ'. Because of the conditions on the statement of this theorem, along with precondition 6, we see that $v' + (-1, 0)$, which is of the form (r, i) with $i < \text{next}.2$, is in $\sigma[, next]$, and so it was already added to σ'.
   Thus, there are no neighbors of v' not yet added to σ' apart from v, and then the condition in Lemma 9 is met.
2. $v = \text{next} + (2, 1)$: This vertex belongs to B, regardless of the parity of r. This is because $r - v.2$ is even, and $v.1 = r + 2$. Then $\text{prev}_{\Omega'}(v) = \text{next} + (1, 0) = v + (-1, -1) = v'$. The neighbors of v' are:
   - $v' + (-1, 0) = \text{next}$, which is already in σ';
   - $v' + (0, -1)$, which by precondition 4 is already in σ';
   - $v' + (1, 0) = next + (2, 0)$, which by precondition 1 is already in σ'; and
   - $v' + (0, 1) = v$.
3. $v = \text{next} + (3, 0)$: This vertex belongs to class A, as $v.1 = r + 3$. By precondition 1, $\text{prev}_{\Omega'}(v) = \text{next} + (2, 0) = v'$. The neighbors of v' are:
   - $v' + (-1, 0) = \text{next} + (1, 0)$, already in σ';

- $v' + (0, -1)$, which by precondition 4 is already in σ';
- $v' + (1, 0) = v$; and
- $v' + (0, 1) = \text{next} + (2, 1)$, just added to σ'.

30: **arrowRight(next + (3, 1))**
Following the definition of "arrowRight", the vertices added here are, in order:

1. $v = \text{next} + (3, 1)$: This vertex belongs to class A, as $v.1 = r + 3$. Thus $\text{prev}_{\Omega'}(v) = \text{next} + (3, 0) = v'$. The neighbors of v' are:
   - $v' + (0, -1) = \text{next} + (3, 0)$, added to σ' on line 29;
   - $v' + (-1, 0) = \text{next} + (2, 1)$, also added to σ' on line 29; and
   - $v' + (0, 1) = v$.
2. $v = \text{next} + (2, 2)$: This vertex belongs to class A, as $v.1 = r + 2$ and $r - v.2$ is odd. Thus $\text{prev}_{\Omega'}(v) = \text{next} + (3, 1) = v'$. The neighbors of v' are:
   - $v' + (0, -1) = \text{next} + (2, 1)$, added to σ' on line 29;
   - $v' + (-1, 0) = \text{next} + (1, 2)$, also added to σ' on line 29; and
   - $v' + (0, 1) = v + (1, 0)$, which is a neighbor of v.
3. $v = \text{next} + (1, 1)$: This vertex can belong to two possible classes. If $\text{next} = (r, r - 1)$, then v belongs to the same class C as (r, r). Otherwise, it belongs to class B, as $v.1 = r + 1$.
   If v belongs to class C, $\text{prev}_{\Omega'}(v)$ is the last element of C in σ that is lower or equal to $(r, r - 1)$, if it exists. This is because all elements of $\sigma[, (r, r - 1)]$ are already in σ', and there are no elements belonging to a class in S apart from the ones in σ, according to precondition 5. Because of condition 3 on the statement of this theorem, there are no neighbors of $\text{prev}_{\Omega'}(v) = w$ belonging to σ which are not yet in σ', except (r, r), which is also a neighbor of v. We need to see then that the remaining neighbors of w - the ones which do not belong to σ - are either also neighbors of v or are lower than v in σ' to confirm consistency. Here, we have three cases. Either:
   (a) w has no neighbors outside σ, in which case the condition holds;
   (b) w is of the form (r, i), in which case its only neighbor z outside σ is $(r + 1, i)$, which by precondition 4 is already in σ'; or
   (c) w is of the form (i, r), which actually cannot happen because of the first condition on the statement of this theorem.

   If, in the other hand, v belongs to class B, $\text{prev}_{\Omega'}(v) = \text{next} + (2, 1) = v'$. All of the neighbors of this vertex except v itself are already in σ', so consistency is preserved when adding v.

Lastly, we prove that when beginning an iteration of the loop, the preconditions are preserved. For that, we prove that at the end of every iteration except the last one, the preconditions are met. For the following proof, let next′ and previous′ be the values of next and previous when starting an iteration, respectively. Note that next′ + (0, 2) = next because of line 32, and that previous = $successor_\sigma$(next′) because of line 31.

1. The last vertex v in A added to σ' is next′ + (2, 2) = next + (2, 0).
2. The last vertex v in B added to σ' is next′ + (1, 1) = next + (1, −1).
3. As next′ is not the last element of σ, $successor_\sigma$(next′) exists, and by definition is inside of σ. Also, all elements in $\sigma[, \text{previous}']$ are already in σ' because of this same precondition, and the elements in $\sigma[\text{previous}', \text{next}']$ were added in line 28. All elements in $\sigma[, \text{previous}]$ except previous are then already in σ'.
4. By this same precondition, we know that all vertices of the form $(r+i, j')$ with $1 \leq i \leq 3$ and $j' <$ next′.2 are already in σ'. We have to prove that the remaining vertices are already in σ', namely, the ones of the form $(r+i, j)$ with next′.2 $\leq j <$ next.2. These are all added on lines 29 and 30 of the loop, except for next′ + (1, −1), which by precondition 2 is already in σ'.
5. All vertices outside σ added to σ' belong to classes A and B. The only vertex which could belong to the class C of (r, r) is next′ + (1, 1), but this one belongs to class B if this is not the last iteration.
6. As next′ is of the form (r, i), with $r - i$ odd, next = next′ + (0, 2) also meets this criteria.

10: **corner()**
For each line in corner(), we will use v to denote the vertex being added.

42: $v = (r + 1, r + 1)$ is of class B. Then prev(v) is then $(r + 2, r)$, which was added in the last iteration of the while loop in horizontalArrows. All its neighbors are already in σ'.

43: $v = (r, r + 1)$ is of the same class C as (r, r). Then prev(v) = $(r + 1, r)$, which was added in the last iteration of the while loop in horizontalArrows. All its neighbors except (r, r) are already in σ', and (r, r) is also neighbor of v.

44: $v = (r + 2, r + 2)$ is of class B. Then prev(v) = $(r + 1, r + 1)$, added in line 42. All its neighbors except $(r + 1, r + 2)$ are already in σ', and $(r + 1, r + 2)$ is also neighbor of v.

45: $v = (r+3, r+1)$ is of class A. Then $\text{prev}(v) = (r+2, r+1)$, added in the last iteration of the while loop in horizontalArrows. All its neighbors except $(r+3, r+1)$ itself are already in σ'.

46: $v = (r+3, r+2)$ is of class A. Then $\text{prev}(v) = (r+3, r+1)$, added in line 45. All its neighbors except $(r+3, r+2)$ itself are already in σ'.

47: $v = (r+3, r+3)$ is of class A. Then $\text{prev}(v) = (r+3, r+2)$, added in line 46. All its neighbors except $(r+3, r+3)$ itself are already in σ'.

48: $v = (r+2, r+3)$ is of class A. Then $\text{prev}(v) = (r+3, r+3)$, added in line 47. All its neighbors except $(r+2, r+3)$ itself are already in σ'.

49: $v = (r+1, r+2)$ is of class A. Then $\text{prev}(v) = (r+2, r+3)$, added in line 48. All its neighbors except $(r+1, r+3)$ are already in σ', and $(r+1, r+3)$ is also neighbor of v.

50: $v = (r, r+2)$ is of class B. Then $\text{prev}(v) = (r+2, r+2)$, added in line 44. All of its neighbors are already in σ'.

51: $v = (r+1, r+3)$ is of class A. Then $\text{prev}(v) = (r+1, r+2)$, added in line 49. All its neighbors except $(r+1, r+3)$ itself are already in σ'.

52: $v = (r, r+3)$ is of class A. Then $\text{prev}(v) = (r+1, r+3)$, added in line 51. All its neighbors except $(r, r+3)$ itself are already in σ'.

53: $v = (r-1, r+2)$ is of class A. Then $\text{prev}(v) = (r, r+3)$, added in line 52. All its neighbors except $(r-1, r+3)$ are already in σ', and $(r-1, r+3)$ is also neighbor of v.

54: $v = (r-1, r+1)$ is of class B. Then $\text{prev}(v) = (r, r+2)$, added in line 50. All its neighbors are already in σ'.

11: **nextInSigma $\leftarrow$ verticalArrows($(r, r), (r-1, r)$)**
For simplicity, we omit this part of the proof, since it is analogous to horizontalArrows. Notice in this case there are vertices outside GR_{r+3} ($(0, r+2)$ for example), but this construction can be used to have a consistent solution in GR_{r+3} since it is an induced graph.

12: $\boldsymbol{\sigma' \leftarrow \sigma' \oplus \sigma[nextInSigma,]}$
We know that σ is a consistent ordering for the vertices in GR_r, and the only vertices in σ' belonging to a class in S that are not in σ are $(r, r+1)$ and $(r+1, r)$, which belong to the same class as (r, r). For every vertex $u \in \sigma[nextInSigma,]$, if $\text{prev}_\Omega(u)$ exists, then either:

- $\text{prev}_\Omega(u)$ is of the form (i, j) with $i, j < r$, which means it has no neighbors outside the ones in σ and so consistency is preserved, as it meets the condition in Lemma 9;

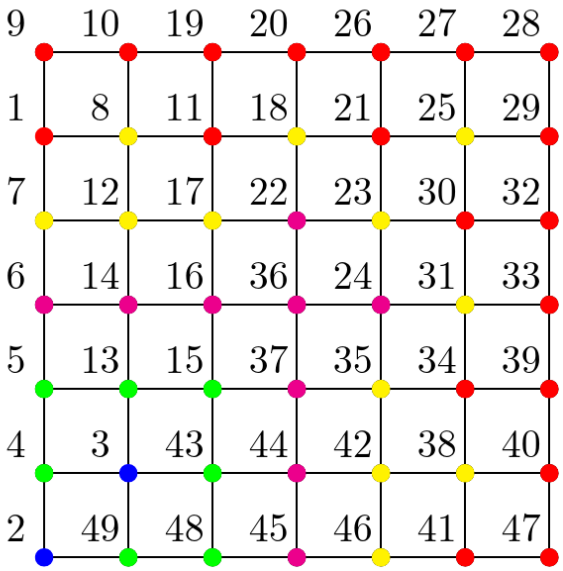


Figure 3.1: An example ordering and partition of the vertices of the GR_7 graph following the construction described in Theorem 10.

- $\text{prev}_\Omega(u)$ is of the form (r, i) with $i < next.2$. By the preconditions stated, the only neighbor of $\text{prev}_\Omega(u)$ outside σ is already in σ', so again consistency is preserved; or
- $\text{prev}_\Omega(u)$ is $(r+1, r)$ or $(r, r+1)$, for which all neighbors are already in σ'.

Given a consistent solution in GR_r using k classes, we showed by construction a consistent solution using $k+2$ classes for a graph for which GR_{r+3} is an induced subgraph. □

Lemma 10. *The construction given in Theorem 10 preserves the preconditions 1 to 3 in the statement.*

Proof.

1. $(r+3,1) < (r+3,2) < \cdots < (r+3, r+3-1) < (r+3, r+3) < (r+3-1, r) < \cdots < (2, r+3) < (1, r+3)$ Notice all vertices here belong to class A in the construction. The method *horizontalArrows* guarantees that the vertices of the form $(r+3, i)$ will respect the relative order until *corner*. Then *corner* places the vertices $(r+3, r+1)$, $(r+3, r+2)$, $(r+3, r+3)$, $(r+2, r+3)$, $(r+1, r+3)$ and $(r, r+3)$ in that order until *verticalArrows*. Then in *verticalArrows* all the vertices of the form $(i, r+3)$ are placed in order.

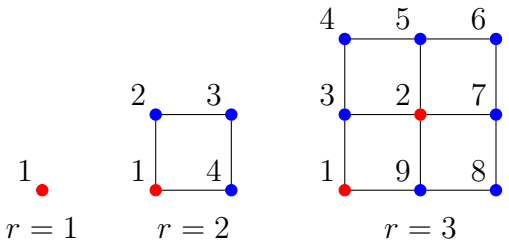


Figure 3.2: Base cases of Theorem 11. The number on the vertices correspond to their relative order in the solution. The vertices in V_1 are shown in red and the vertices in V_2 are shown in blue.

2. $(r+3-1, r+3-1) < (r+3, r+3)$ This condition holds from the order given by *corner*.

3. If there is a vertex v in the same class C as $(r+3, r+3)$ in S which is lower or equal to $(r+3, r+3-1)$ according to σ, then there is no neighbor w of v greater than $(r+3, r+3-1)$ in σ other than $(r+3, r+3)$ Notice all the neighbors from vertices that come before $(r+3, r+3)$ in A only belong to A or B. Then all the vertices placed by *horizontalArrows* (which are all the vertices from A before *corner*) preserves this condition since *corner* is called after it. Then for the vertices placed in *corner* in class A before $(r+3, r+3)$, which are $(r+3, r+1)$ and $(r+3, r+2)$, also preserve the condition.

□

Theorem 11. $\text{thin}(GR_r) \leq \lceil \frac{2r}{3} \rceil$

Proof. We will do induction on the value of r, proving that there is a consistent solution with $\lceil \frac{2r}{3} \rceil$ classes that satisfies the conditions in Theorem 10.
For the base case we show it for $r \in \{1, 2, 3\}$

- $r = 1$: order:$[(1,1)]$, class $V_1 : [(1,1)]$. Then $\text{thin}(GR_1) \leq 1 \leq \lceil \frac{2 \cdot 1}{3} \rceil$.
- $r = 2$: order:$[(1,1),(2,1),(2,2),(1,2)]$, class $V_1 : [(1,1)]$, class $V_2 : [(1,2),(2,2),(2,1)]$. Then $\text{thin}(GR_2) \leq 2 \leq \lceil \frac{2 \cdot 2}{3} \rceil$.
- $r = 3$: order:$[(1,1),(2,2),(2,1),(3,1),(3,2),(3,3),(2,3),(1,3),(1,2)]$, class $V_1 : [(1,1),(2,2)]$, class $V_2 : [(1,2),(1,3),(2,3),(3,3),(3,2),(3,1)]$. Then $\text{thin}(GR_3) \leq 2 \leq \lceil \frac{2 \cdot 3}{3} \rceil$.

For the inductive step we want to show that the property also holds for $k' + 3$ in the case we know it is true for all smaller cases. Since the property is true for k' due to the inductive hypothesis, simply by using the GR_r *bound* and lemma 10 we can say that the property holds for $k' + 3$. □

3.5 Heuristics

We will now show how our algorithm can be used as a heuristic for other graph classes that are not trees.

Theorem 12. *Given a tree of cliques G, there is an $\mathcal{O}(n\log(n))$ algorithm that returns a consistent solution using k classes guaranteeing that $k - \text{thin}(G) \leq 1$.*

Proof. Given the *tree of cliques* G obtained from the cliques $\{Q_1, Q_2, \dots Q_t\}$ and the set of edges S between vertices from $\{q_1, q_2, \dots q_t\}$, we will prove the statement with help of the $\mathcal{O}(n\log(n))$ algorithm on trees we proposed on this work plus a heuristic construction.
Let us call T to the forest induced by the vertices $\{q_1, q_2, \dots q_t\}$. Notice $\text{thin}(T) \leq \text{thin}(G)$. We run the algorithm we proposed on each tree of T, and with this method can easily we obtain a consistent order and partitions of the vertices on T. We will use this information to generate a consistent solution in G. For all the vertices in $\{q_1, q_2, \dots q_t\}$ we will use the exact same classes and keep the relative order of the vertices. For all the rest of the vertices we will use one extra class A and we will set the ordering this way: exactly after q_i we put all the other vertices that belong to Q_i (the order does not matter because of the symmetry).
Since we are using $k = \text{thin}(T) + 1$ classes and $\text{thin}(T) \leq \text{thin}(G)$, we can say that, if the construction is consistent, $k - \text{thin}(G) = \text{thin}(T) + 1 - \text{thin}(G) \leq 1$. The construction has a complexity of $\mathcal{O}(t\log(t) + (n-t))$ which is also $\mathcal{O}(n\log(n))$. Now we show the construction is indeed consistent. For that we are going to see that all triples (r, s, t) such that $r < s < t$ satisfy the *consistency condition*. Given a triple, we define p as the number of vertices of the triple that belong to the extra class A.

- $p = 0$:
  We know all the vertices of the triple belong to $V(T)$. Since this triple was consistent in T, by construction, the property holds; since the relative order, the classes and the edges between those vertices remained invariant.

- $p = 1$:
  There is exactly one vertex x on the triple that belongs to $V(Q_i) - q_i$ for some i.

  - If $x = r$ or $x = t$:
    Notice that if $(r, t) \notin E$ the consistency holds for this triple. Since $p = 1$ the only option to break the consistency will be in the case $r = q_i$ and $t = x$, because by construction x comes after q_i in the order and there is no other neighbor of x that does not belong to the same class. If this happens, by construction, all vertices between r and t belong to $V(Q_i) - q_i$; but since $p = 1$ there is no such s that breaks the consistency in this case.

- If $x = s$:
  Notice that if r and s belong to different classes the consistency holds. Also r and s cannot belong to the same class because $p = 1$.

- $p = 2$:
  There are exactly two different vertices x and y in $V(G) - V(T)$ on the triple. Without loss of generality suppose $x < y$.

  - If $r = x$ and $s = y$:
    Notice that if $(r, t) \notin E$ then the consistency holds. Since $p = 2$, t cannot be one of the vertices in $Q_i - q_i$, also t is not q_i because by construction the vertex q_i comes exactly before all the vertices in $Q_i - q_i$. Because of this, none of the neighbors of r is a candidate to break the consistency.
  - Else:
    If we are not in the case $r = x$ and $s = y$, since $p = 2$, r and s cannot belong to the same class, because one of them will be inside $V(G) - V(T)$ and the other outside.

- $p = 3$:
  We know all vertices belong to $V(G) - V(T)$. If $(r, t) \notin E$ the condition holds. So let us suppose r and t both belong to $Q_i - q_i$ for a certain i. Notice that since s must be between r and t in the order and, by construction, all vertices between a pair of vertices in $Q_j - q_j$ belong to $Q_j - q_j$, the only possibility is that the vertex s also belongs to $Q_i - q_i$. Because Q_i is a clique, (r, t) is an edge in G but, for the same reason, (s, t) is also an edge. So the consistency condition holds.

□

Theorem 13. *Given a graph G and a set of vertices S in $V(G)$, then G is $(|S| + \text{thin}(G - S))$-thin.*

Proof. Let us prove the statement by construction.
The vertices in $G-S$ retain the exact same class as a given optimal consistent solution in $G - S$ in the solution for G. We are going to use one extra class for each vertex in S. Notice that we are using exactly $|S|+\text{thin}(G-S)$ classes. Regarding the ordering, we are going to put first all the vertices that are in S in any order, and after those we put the vertices from $G - S$ preserving the relative order we mentioned.
Let us see why this solution is consistent by analyzing the consistency property on all triples of vertices. We have to see that for each (r, s, t) such that $r < s < t$ if r and s belong to the same class and $(r, t) \in E$, then there must be an edge between s and t. Given a triple (r, s, t) of the solution we proposed such that $r < s < t$, if r

and s belong to different classes the condition holds. If r and s belong to the same class, notice that neither r nor s belong to S since by construction all of those vertices belong to different classes. Also by construction t is not in S since t comes after r and s in the ordering and all vertices in S come before. But the triple (r, s, t) with vertices only in $V(G - S)$ must be consistent in G, because they were consistent in $G - S$ and the relative order, the classes and edges between them remained invariant. Since we showed a consistent solution using $(|S| + \text{thin}(G - S))$ different classes, the statement holds. □

Corollary 13.1. *Given a graph G and a set of vertices S in $V(G)$ such that $G - S$ forms a tree of cliques and $|S| = \mathcal{O}(\min(\log(n), d))$, then* $\text{thin}(G) = \mathcal{O}(\min(\log(n), d))$.

Note that these results could be used as a heuristic for the more general problem of computing the thinness of an arbitrary graph.

CHAPTER 4

Future Work

Some points left for future work:

- Characterize graphs G that have a subset of vertices S such that $G-S$ is a forest and $|S| = \mathcal{O}(\min(\log(n), d))$. For this cases we can guarantee $\mathcal{O}(\min(\log(n), d))$ thinness.
- Characterize graphs G that have a subset of vertices S such that $G - S$ is a *tree of cliques* and $|S| = \mathcal{O}(\min(\log(n), d))$. For this cases we can guarantee $\mathcal{O}(\min(\log(n), d))$ thinness.
- Investigate if computing the thinness of a tree is achievable in linear time.
- Try to extend the ideas presented in the k-component index theorem to compute the thinness of other classes of graphs, in particular chordal graphs.
- Can a similar algorithm be defined for identifying the proper thinness of a tree? And for its independent and complete thinness?

Bibliography

[1] F. Bonomo and D. De Estrada, On the thinness and proper thinness of a graph, *Discrete Applied Mathematics* 261 (2019), 78–92.

[2] F. Bonomo, S. Mattia, and G. Oriolo, Bounded coloring of co-comparability graphs and the pickup and delivery tour combination problem, *Theoretical Computer Science* 412(45) (2011), 6261–6268.

[3] F. Bonomo-Braberman, N. Brettell, A. Munaro, and D. Paulusma, *Solving problems on generalized convex graphs via mim-width*, arXiv: 2008.09004.

[4] F. Bonomo-Braberman, C. Gonzalez, F. S. Oliveira, M. Sampaio, and J. Szwarcfiter, Thinness of product graphs, *Discrete Applied Mathematics*, To appear.

[5] M. Cygan, F. Fomin, L. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk, and S. Saurabh, *Parameterized algorithms*, Springer, 2015.

[6] J. Ellis, I. Sudborough, and J. Turner, The vertex separation and search number of a graph, *Information and Computation* 113(1) (1994), 50–79.

[7] M. Golumbic, The complexity of comparability graph recognition and coloring, *Computing* 18 (1977), 199–208.

[8] S. Høgemo, *On the linear MIM-width of trees*, M.Sc. Thesis, The University of Bergen, Bergen, Norway, 2019.

[9] S. Høgemo, J. A. Telle, and E. R. Vågset, Linear MIM-width of trees, *Proceedings of the International Workshop on Graph-Theoretic Concepts in Computer*

Science - WG 2019 (I. Sau and D. M. Thilikos, eds.), Lecture Notes in Computer Science, vol. 11789, Springer, 2019, pp. 218–231.

[10] C. Mannino, G. Oriolo, F. Ricci, and S. Chandran, The stable set problem and the thinness of a graph, *Operations Research Letters* 35 (2007), 1–9.

[11] H. Meyniel, A new property of critical imperfect graphs and some consequences, *European Journal of Combinatorics* 8 (1987), 313–316.

[12] L. Rabinowicz, *Sobre la thinness de árboles*, M.Sc. Thesis, Departamento de Computación, FCEyN, Universidad de Buenos Aires, Buenos Aires, 2019 (in Spanish).

[13] N. Robertson and P. Seymour, Graph minors. I. Excluding a forest, *Journal of Combinatorial Theory. Series B* 35(1) (1983), 39–61.

[14] P. Scheffler, A linear algorithm for the pathwidth of trees, In: *Topics in Combinatorics and Graph Theory: Essays in Honour of Gerhard Ringel* (R. Bodendiek and R. Henn, eds.), Physica-Verlag HD, Heidelberg, 1990, pp. 613–620.

[15] Y. Shitov, *Graph thinness is NP-complete*, Manuscript, 2021.

[16] M. Vatshelle, *New width parameters of graphs*, Ph.D. Thesis, Department of Informatics, University of Bergen, 2012.

[17] D. West, *Introduction to graph theory*, 2nd ed., Prentice Hall, 2000.