Thread graphs, linear rank-width and their algorithmic applications

Robert Ganian *

Faculty of Informatics, Masaryk University
Botanická 68a, Brno, Czech Republic
xganian1@fi.muni.cz

Abstract. Many NP-hard graph problems can be efficiently solved on graphs of bounded tree-width. Several articles have recently shown that the so-called rank-width parameter also allows efficient solution of most of these NP-hard problems, while being less restrictive than tree-width. On the other hand however, there exist problems of practical importance which remain hard on graphs of bounded rank-width, and even of bounded tree-width or trees. In this paper we consider a more restrictive version of rank-width called linear rank-width, analogously to how path-width is obtained from tree-width. We first provide a characterization of graphs of linear rank-width 1 and then show that on such graphs it is possible to obtain better algorithmic results than on distance hereditary graphs and even trees. Specifically, we provide polynomial algorithms for computing path-width, dominating bandwidth and a 2-approximation of ordinary bandwidth on graphs of linear rank-width 1.

Key words: rank-width, linear rank-width, thread graphs, bandwidth, path-width.

1 Introduction

The introduction of tree-width by Robertson and Seymour [10] was a breakthrough in the design of graph algorithms since it allowed efficient solution of many NP-hard problems on all graphs having bounded tree-width. A lot of research since then has focused on obtaining a width measure which would be more general and still allowed efficient algorithms for a wide range of NP-hard problems on graphs of bounded width. Clique-width was considered to be such a parameter [2], however it turned out to have several disadvantages – most notably it was not possible to compute clique-width k-expressions of input graphs, which had a severe negative impact on the runtime of parameterized algorithms. To this end, Oum and Seymour have proposed rank-width [9], which addresses and solves all the problems of clique-width and possesses better algorithmic properties (see e.g. [3, 4]).

* This research has been supported by the Czech research grants 201/09/J021 and MUNI/E/0059/2009.

But what about problems which are NP-hard even on graphs of bounded tree-width or even on trees? Such problems exist and have practical importance, yet rank-width is strictly less restrictive than tree-width and so it cannot help in these cases. The parameter used most often for these exceptionally hard problems is path-width, which is defined as tree-width with the additional requirement that the tree-decomposition must form a path. However, path-width is extremely restrictive – the graphs of path-width 1 are exactly paths.

In the article we study a new width measure called linear rank-width, defined by an additional requirement on the rank-decomposition of graphs analogous to the requirement path-width imposes on tree-decompositions. The goal is to obtain a width measure which on one hand is less restrictive than path-width and yet on the other hand allows efficient algorithms for problems which are hard on graphs of bounded rank-width or even tree-width. We first provide a constructive characterization of graphs having linear rank-width 1 (further referred to as thread graphs), and then continue by providing positive algorithmic results on this class of graphs.

The algorithmic section contains three new polynomial algorithms on thread graphs, each solving some problem which remains hard on other severely restrictive classes of graphs. The first is a 2-approximation algorithm for the classical bandwidth problem (NP-hard to 2-approximate even on trees [11]). A polynomial algorithm for computing *dominating bandwidth* – a natural variation of bandwidth – follows, as well as a proof that computing dominating bandwidth on trees is NP-hard. The third algorithm computes the path-width of thread graphs, the problem otherwise being NP-hard on weighted trees and graphs of rank-width 1 [8].

2 Rank-width and linear rank-width

2.1 Rank-width

The usual way of defining rank-width is via the branch-width of the cut-rank function. A set function $f : 2^M \to \mathbb{Z}$ is *symmetric* if $f(X) = f(M \setminus X)$ for all $X \subseteq M$. Given a symmetric function $f : 2^M \to \mathbb{Z}$ on a finite ground set M, a *branch-decomposition* of f is a pair (T, μ) of a subcubic tree T and a bijective function $\mu : M \to \{t : t \text{ is a leaf of } T\}$. For an edge e of T, the connected components of $T \setminus e$ induce a bipartition (X, Y) of the set of leaves of T. The *width* of an edge e of a branch-decomposition (T, μ) is $f(\mu^{-1}(X))$. The *width* of (T, μ) is the maximum width over all edges of T. The *branch-width* of f is the minimum of the width of all branch-decompositions of f.

For a simple graph G, let $\boldsymbol{A}_G[U, W]$ be the bipartite adjacency matrix of a bipartition (U, W) of the vertex set $V(G)$ defined over the two-element field GF(2) as follows: the entry $a_{u,w}$, $u \in U$ and $w \in W$, of $\boldsymbol{A}_G[U, W]$ is 1 if and only if uw is an edge of G. The *cut-rank* function $\rho_G(U) = \rho_G(W)$ then equals the rank of $\boldsymbol{A}_G[U, W]$ over GF(2). A *rank-decomposition* (see Figure 1) and *rank-width* of a graph G is the branch-decomposition and branch-width of the cut-rank function ρ_G of G on $M = V(G)$, respectively.

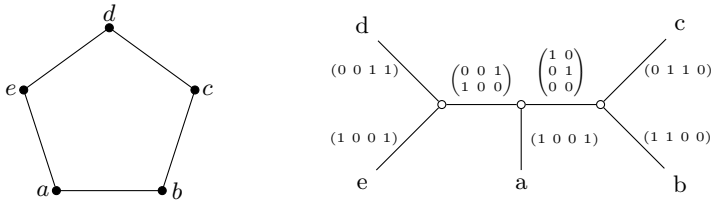

Fig. 1: A rank-decomposition of the graph cycle C_5 incl. matrices at edges.

As already mentioned in the introduction, rank-width is closely related to clique-width and more general than tree-width. Indeed:

Theorem 2.1. *Let G be a simple graph, and* $\mathrm{pw}(G)$, $\mathrm{tw}(G)$, $\mathrm{cwd}(G)$, $\mathrm{rwd}(G)$ *denote in this order the path-width, tree-width, clique-width, and rank-width of G. Then the following holds*

a) [9] $\mathrm{rwd}(G) \leq \mathrm{cwd}(G) \leq 2^{\mathrm{rwd}(G)+1} - 1$,
b) [7] $\mathrm{rwd}(G) \leq \mathrm{tw}(G) + 1 \leq \mathrm{pw}(G) + 1$,
c) [folklore] $\mathrm{tw}(G)$ *cannot be bounded from above by* $\mathrm{rwd}(G)$, *and* $\mathrm{pw}(G)$ *cannot be bounded from above by* $\mathrm{tw}(G)$.

Although rank-width and clique-width are "tied together" (one is bounded if the other is bounded), there are two very good reasons to favor rank-width over clique-width for algorithmic design. First, it is possible to design exponentially faster algorithms by using rank-width instead of clique-width [3, 4]. Second, clique-width algorithms need a so-called k-expression to run, while rank-width algorithms require rank-decompositions. The difference is that while we cannot compute k-expressions for clique-width efficiently, it is possible to compute a rank-decomposition in polynomial (FPT to be precise) time if the rank-width is bounded.

Theorem 2.2 ([6]). *There is an FPT algorithm that, for a fixed parameter t and a given graph G, either finds a rank-decomposition of G of width at most t or confirms that the rank-width of G is more than t.*

2.2 Linear rank-width

Based on Theorem 2.1 (b), a certain hierarchy of width parameters seems to be present. Rank-width is the most general of the three parameters, though if some problem is hard on rank-width one can try solving it on graphs of bounded tree-width, and if that again fails then there is path-width. This relationship is illustrated well in the following table:

	paths	trees	cliques
path-width	bounded	unbounded	unbounded
tree-width	bounded	bounded	unbounded
rank-width	bounded	bounded	bounded

The catch here is that many of the problems which are hard on tree-width and trees tend to be solvable on cliques as well, not just paths. It is a natural question to ask whether there exists a width parameter which remains capable of solving problems hard on tree-width, but at the same time relaxes the restrictions of path-width and achieves low values also on cliques. This is strong motivation for linear rank-width.

Definition 2.3. *A rank-decomposition* (T, μ) *is* linear *if* T *is a caterpillar (i.e. a path with pendant vertices). The linear rank-width of a graph* G *is the minimum of the width of all linear rank-decompositions of* G*.*

Since almost all degree 3 nodes correspond to a single vertex of G, we will abuse the notation slightly and in these cases refer to the internal nodes by the names of vertices in G. The only exceptions to this are the first and last internal nodes of the decomposition, which are usually handled separately anyway.

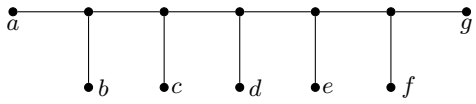

Fig. 2: An example of a linear rank-decomposition

Theorem 2.4. *The linear rank-width of paths and cliques is* 1*, and the linear rank-width of trees is not bounded by any constant.*

Proof located in Appendix.

3 Thread graphs

The classes of graphs of rank-width 1, tree-width 1 or path-width 1 each possess interesting structural properties. For rank-width these are called distance hereditary graphs, while for tree-width and path-width we speak of forests and disjoint unions of paths respectively. In this section we introduce a new graph class called *thread graphs* and prove that this is exactly the class of graphs which have linear rank-width 1, answering a question asked by Oum at GROW 2009. The nice structural properties of thread graphs are then used in the next section for algorithmic design.

Definition 3.1. *A thread graph is a graph which can be constructed by gradually creating vertices. Every new vertex is created with 3 attributes, as follows:*

1. *Passive* ($\mathcal{P}$) *or Active* ($\mathcal{A}$)*;*
2. *Disconnect* ($\mathcal{D}$) *or Join* ($\mathcal{J}$)*;*
3. *Normal or Reset* ($\mathcal{R}$)*;*

Each new vertex is either $\mathcal{P}$ or $\mathcal{A}$ and either $\mathcal{D}$ or $\mathcal{J}$ and additionally may or may not be $\mathcal{R}$.

A $\mathcal{D}$ vertex is created without any incident edges. A $\mathcal{J}$ vertex on the other hand is created with incident edges to all vertices which are currently $\mathcal{A}$.

Finally, an $\mathcal{R}$ vertex changes all previous $\mathcal{A}$ vertices to $\mathcal{P}$. Every vertex is normal (not $\mathcal{R}$) unless explicitly said otherwise.

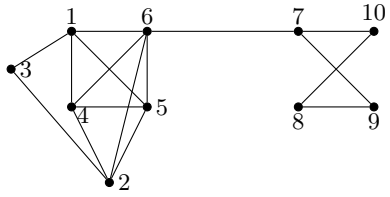

Fig. 3: A thread graph created by the following sequence : $\mathcal{AJ}$ $\mathcal{AD}$ $\mathcal{PJ}$ $\mathcal{AJ}$ $\mathcal{AJ}$ $\mathcal{AJR}$ $\mathcal{AJ}$ $\mathcal{AD}$ $\mathcal{PJ}$ $\mathcal{AJ}$. The numbers capture the order of created vertices.

See Figure 3 for an example of a thread graph. Notice that for connected thread graphs it is enough to consider $\mathcal{AJ}$, $\mathcal{AD}$, $\mathcal{PJ}$, $\mathcal{AJR}$ vertices (any other type of vertices disconnects the graph). Furthermore, it is a trivial observation that any thread graph can be created by disjoint union of connected thread graphs – simply by creating vertices in the same order they originally appeared in each connected component. Our goal is to prove the following:

Theorem 3.2. *A graph G has linear rank-width 1 if and only if G is a thread graph.*

Proof. First, assume G has linear rank-width 1. We will show how to create G as a thread graph from its linear rank-decomposition of width 1 by induction on the vertices of G in the order they appear in the linear rank-decomposition from one of its ends to the other; our inductive assumption is that a created vertex is $\mathcal{A}$ if and only if it still has some neighbour in G which we have not created yet.

We create the first vertex as $\mathcal{A}$ if and only if it is adjacent to some vertex in G. Now whenever a new vertex v is created, we look at:

1. The bipartite adjacency matrix at the edge of the decomposition between v and what we have already created. All the non-zero rows of the matrix must be identical to keep the rank equal to 1, and so the column of v either contains only zeros (in which case no edges are created and v is $\mathcal{D}$) or contains ones exactly at those rows which are active (v is $\mathcal{J}$).
2. The bipartite adjacency matrix at the edge between v and what we have yet to create. First, v is $\mathcal{A}$ if and only if its row is non-zero (i.e. it has some future neighbours). Second, if v is $\mathcal{A}$ then either its row is identical to the rows of all previous $\mathcal{A}$ vertices – in which case v is normal – or all the previous $\mathcal{A}$ vertices now have rows containing only zeros – in which case v is $\mathcal{R}$.

On the other hand, assume G is a thread graph. We may create a linear rank-decomposition by ordering the vertices from one end (say the left) to the other as they are created in the thread graph. Consider a bipartite adjacency matrix at some edge between two internal nodes of the decomposition. Vertices occurring as rows in the bipartite adjacency matrix are those we have created so far in the thread graph, and each is either $\mathcal{P}$ or $\mathcal{A}$ at this point. Now the rows corresponding to $\mathcal{P}$ vertices contain only zeros (since no new vertex can create an edge to passive vertices) and the rows corresponding to $\mathcal{A}$ vertices are all identical (new $\mathcal{J}$ vertices will create a column of ones at $\mathcal{A}$ vertex rows until a $\mathcal{R}$ vertex changes current $\mathcal{A}$ vertices to $\mathcal{P}$). The width of such a decomposition is one, which concludes the proof.

□

Structural properties of thread graphs

We proceed by providing several useful structural properties of thread graphs. A cluster is the set of vertices containing two consecutive $\mathcal{R}$ vertices and all vertices between them. Formally, let $\mathcal{R}_i$ denote the i-th created $\mathcal{R}$ vertex and C_i be the set of vertices containing two consecutive $\mathcal{R}$ vertices $\mathcal{R}_i$ and $\mathcal{R}_{i+1}$ and all vertices created between them. Although clusters may contain vertices of arbitrary degree, they are fairly isolated from the rest of the graph:

Proposition 3.3. *Given a thread graph G and a cluster C_i in G, the following holds:*

1. *$\mathcal{R}_i$ may only be adjacent to vertices in C_i and C_{i-1}*
2. *$\mathcal{R}_{i+1}$ may only be adjacent to vertices in C_i and C_{i+1}*
3. *vertices in C_i other than $\mathcal{R}_i$ and $\mathcal{R}_{i+1}$ may only be adjacent to vertices in C_i.*

Furthermore, we will always assume that all $\mathcal{PD}$ vertices are created first and that the first and last created vertices are both $\mathcal{R}$ (so that each vertex belongs to some cluster). Now any thread graph is formed by a sequence of clusters. The next step is to look at the structure of individual clusters.

Proposition 3.4. *In any cluster of a thread graph, the following holds:*

1. *$\mathcal{AJ}$ vertices form a clique in G.*
2. *all $\mathcal{PJ}$ vertices are adjacent to every $\mathcal{AJ}$ and $\mathcal{AD}$ vertex created before them.*
3. *all $\mathcal{AD}$ vertices are adjacent to every $\mathcal{AJ}$ vertex created after them.*
4. *there are no other edges in the cluster other than those covered by the previous cases.*

Finally, our algorithms also require information about the order and attributes of created vertices in thread graphs. This information can be computed from any thread graph in polynomial time (the proof is located in the Appendix).

Theorem 3.5. *For a thread graph G, it is possible to compute in polynomial time a creating sequence for G.*

4 Algorithmic applications

4.1 Computing bandwidth

Now that we have some knowledge of the structure of graphs of linear rank-width 1, it is time to look at whether it is possible to solve hard problems on this class of graphs. Let us begin with the bandwidth problem.

Definition 4.1. *Given a graph G and a one-to-one mapping $f : V \to \{1, \dots |V|\}$, the bandwidth of f is defined as the maximum difference between the labels of vertices sharing an edge. The bandwidth of G, denoted by $bwd(G)$ is then the minimum bandwidth over all such f.*

Bandwidth has many applications in theory as well as practice, ranging from networking to biology (see e.g. [12] or the dedicated survey [1]). Unfortunately, it turns out that computing the bandwidth of graphs is extremely hard. Even on trees, approximating bandwidth within some constant factor is NP-hard and the best known polynomial-time approximation bound is $O(log^{2.5}n)$ [5]. Before giving the main result of this section, we first need a few technical results:

Proposition 4.2. *The bandwidth of a clique K_n is $n-1$ and the bandwidth of a bipartite clique $K_{n,m}$ is the minimum of $n + \frac{m}{2} - 1$, $\frac{n}{2} + m - 1$ (rounded up).*

Lemma 4.3. *Consider a bipartite graph $G = (\{s_1 \dots s_n\} \cup \{u_1 \dots u_m\}, E)$ with the following structure on E:*

1. *every vertex has degree at least 1.*
2. *$\{s_i, u_j\} \in E$ implies $\{s_{i+1}, u_j\} \in E$.*
3. *$\{s_i, u_j\} \in E$ implies $\{s_i, u_{j-1}\} \in E$.*

Then mapping s_i to i and u_j to $n+j$ results is a 2-approximation of the optimal bandwidth of G.

Proof. Consider an arbitrary s_i adjacent to $u_1, u_2 \dots u_j$. By our mapping, the bandwidth of edges incident to s_i is $n-i+j$ (induced by $\{s_i, u_j\}$). The subgraph induced by s_k ($k \geq i$) and u_l ($l \leq j$) is a complete bipartite graph $K_{n-i+1,j}$, and by Proposition 4.2 the optimal bandwidth of this subgraph is at least $min(n - i + \frac{j}{2}, \frac{n-i-1}{2} + j)$. But $n - i + j \leq 2 \cdot min(n - i + \frac{j}{2}, \frac{n-i-1}{2} + j)$, concluding our proof. □

Theorem 4.4. *There exists a polynomial time algorithm for 2-approximation of bandwidth on thread graphs.*

Proof. We may assume that our thread graph is connected, since the bandwidth of a disconnected graph equals the maximum of its connected components. Proposition 3.3 tells us that any thread graph is a sequence of clusters. Our mapping will follow the sequence of clusters and in each cluster C_i is defined as follows (see Figure 4 for an illustration):

1. $\mathcal{PJ} \to p \dots q-1$ in the order of creation, p being the first free number.
2. $\mathcal{R}_i \to q$.
3. $\mathcal{AD} \cup \mathcal{AJ} \to q+1 \dots r$ in the order of creation.
4. $\mathcal{R}_{i+1}$ is mapped in cluster C_{i+1}; if C_i is the last cluster, then $\mathcal{R}_{i+1}$ is simply $r+1$.

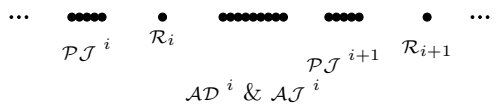

Fig. 4: Order of vertices for bandwidth 2-approximation

Now it is necessary to prove that the bandwidth of this mapping cannot be greater than twice the optimal bandwidth. Every edge belongs to some cluster of G, and so we may look at the bandwidth induced by edges of an arbitrary cluster C_i with a_i $\mathcal{PJ}$ vertices, b_i $\mathcal{AD}$ vertices and c_i $\mathcal{AJ}$ vertices.

Consider the edges incident to $\mathcal{R}_i$. For those between $\mathcal{R}_i$ and $\mathcal{PJ}$ vertices, each $\mathcal{PJ}$ vertex is adjacent to $\mathcal{R}_i$ and so the optimal bandwidth cannot be less than $\frac{a_i}{2}$ – in our mapping the greatest bandwidth induced by these edges is a_i and so the approximation holds. There are no edges between $\mathcal{R}_i$ and $\mathcal{AD}$ vertices. The bandwidth induced by the remaining edges incident to $\mathcal{R}_i$ is maximized at the edge between $\mathcal{R}_i$ and $\mathcal{R}_{i+1}$ and equals $b_i + c_i + a_{i+1} + 1$, however $\mathcal{R}_{i+1}$ has at least $b_i + c_i + a_{i+1} + 1$ neighbours (all $\mathcal{AJ}$ and $\mathcal{AD}$ vertices in this cluster, $\mathcal{R}_i$ and $\mathcal{PJ}$ vertices in the next cluster are adjacent to $\mathcal{R}_{i+1}$) and so this again does not break the 2-approximation. For the same reason, the 2-approximation holds for the bandwidth induced by edges between $\mathcal{AJ}$ and $\mathcal{AD}$ vertices and $\mathcal{R}_{i+1}$. $\mathcal{PJ}$ vertices in this cluster cannot be adjacent to $\mathcal{R}_{i+1}$, and so all edges incident to $\mathcal{R}_{i+1}$ or $\mathcal{R}_i$ are problem-free.

The bandwidth induced by edges between $\mathcal{AJ}$ and $\mathcal{AD}$ vertices is strictly lower than the bandwidth of the edge between $\mathcal{R}_{i+1}$ and $\mathcal{R}_i$, so the last remaining case to argue are the edges between $\mathcal{PJ}$ and $\mathcal{AJ}$ / $\mathcal{AD}$ vertices. Consider the subgraph induced by $\mathcal{PJ}$, $\mathcal{AJ}$, $\mathcal{AD}$ and $\mathcal{R}_i$. We may ignore the edges between $\mathcal{AJ}$, $\mathcal{AD}$ and $\mathcal{R}_i$ since these have already been dealt with and also ignore vertices which have degree 0 in this subgraph – the resulting subgraph is bipartite with $s_1 \dots s_n$ being $\mathcal{PJ}$ vertices in the order of creation and $u_1 \dots u_m$ being $\mathcal{R}_i$, $\mathcal{AJ}$ and $\mathcal{AD}$ vertices in the order of creation. Now it is a trivial consequence of Proposition 3.4 that all neighbours of s_i must also be neighbours of s_{i+1} and all neighbours of u_j must also be neighbours of u_{j-1}. So, due to Lemma 4.3, we see that the edges incident to $\mathcal{PJ}$ also induce a 2-approximation of the optimum bandwidth.

Since in our mapping no edge in G may induce a bandwidth greater than twice the lowest possible bandwidth, computing a 2-approximation is merely a question of first constructing this mapping (which can be done in linear time once we have access to the creating sequence as per Theorem 3.5) and then running through all edges and finding the maximum bandwidth with respect to this mapping.

□

4.2 Dominating bandwidth

While bandwidth is a well-known problem, in this subsection we introduce a related problem called dominating banwidth. We believe the idea behind the problem is simple and at the same time natural: while in ordinary bandwidth each vertex of a graph is assigned its own value, in dominating bandwidth we allow areas of the graph which are "close" – distance 2 – to be assigned the same value. This may have practical applications in communication (i.e. constructing an array of communicating relays with bandwidth restrictions, each relay covering the surrounding areas), but our main goal here is to show that there exist interesting problems which are NP-hard on trees and at the same time polynomially solvable on thread graphs.

Definition 4.5. *The dominating bandwidth problem for a given graph G and a minimum dominating set $X \subseteq V(G)$ of G is the problem of computing a mapping $f : V \to \{1, \dots |X|\}$ such that:*

1. *each $v \in X$ receives a unique label.*
2. *each $u \in V(G) - X$ receives the same label as some u-neighbour $v \in X$.*
3. *the bandwidth of f (defined as the maximum difference between the labels of vertices sharing an edge) is minimized.*

Remark 4.6. One could also define the problem differently – to compute a minimum dominating set and a mapping f such that the bandwidth is minimized. Both problems are hard on trees, however our definition requires consideration of all possible minimum dominating sets and so makes the result of Theorem 4.8 stronger.

Theorem 4.7. *The dominating bandwidth problem is NP-hard on trees.*

See Appendix for proof. On the other hand, computing the dominating bandwidth of thread graphs is not hard. In fact:

Theorem 4.8. *The dominating bandwidth of thread graphs is 1.*

Proof. Again we may assume that our thread graph G is connected, since the bandwidth of disconnected graphs equals the maximum of their connected components. First let us look at the provided minimum dominating set X. Any cluster C_i may contain at most two dominating vertices – all vertices in C_i and vertices adjacent to C_i are dominated by $\mathcal{R}_i$ and $\mathcal{R}_{i+1}$. Similarly, any subsequent clusters C_i and C_{i+1} may contain at most three dominating vertices, since choosing $\mathcal{R}_i$, $\mathcal{R}_{i+1}$ and $\mathcal{R}_{i+2}$ dominates both clusters and all their neighbours. Finally, it is a trivial observation that if $v, u \in X$ are both dominating, then they cannot both be $\mathcal{AD}$ or both be $\mathcal{PJ}$ (again due to minimality of X).

Consider a linear ordering of vertices in X, where for $v \in C_i$ and $u \in C_j$, $v < u$ if:

1. $i < j$, or
2. $i = j$ and v is $\mathcal{PJ}$, or

3. $i = j$ and u is $\mathcal{AD}$.
4. $i = j$ and both u and v are $\mathcal{AJ}$ and u was created after v.

Let us construct f to match the linear ordering on X. Since each cluster has at most two dominating vertices and each edge belongs to some cluster, the bandwidth of edges between dominating vertices is one. Non-$\mathcal{R}$ vertices do not have neighbours outside a given cluster and so the same argument applies and it is sufficient to choose the label of any adjacent dominating vertex to preserve a bandwidth of 1.

What remains are the $\mathcal{R}$ vertices, which may have neighbours in two clusters. If the clusters contain two or less dominating vertices, it suffices to choose the label of any of them (they will have labels $f(x)$ and $f(x)+1$). On the other hand, if the clusters contain three dominating vertices in total – the maximum possible – then they must be assigned labels $f(x)$, $f(x)+1$ and $f(x)+2$, so choosing $f(x)+1$ certifies that the bandwidth will remain 1; the second dominating vertex of the three will always be adjacent to our $\mathcal{R}$vertex due to it being $\mathcal{J}$ in the second cluster or $\mathcal{A}$in the first (otherwise it would contradict the linear ordering), so this choice is indeed valid and the proof is finished.

□

4.3 The path-width problem

The final algorithm in this section is a polynomial time algorithm for computing the path-width of thread graphs.

Definition 4.9. *A path-decomposition of a graph $G = (V, E)$ is a path $P = (T, A)$ where the nodes T are subsets of V (also called bags) such that the following holds:*

1. *Each vertex $v \in V$ appears in some bag.*
2. *For every edge $\{v, w\}$ there exists a bag containing both v and w.*
3. *For every vertex $v \in V$, the bags containing v induce a subpath in P (the interpolation property).*

   The width of the path-decomposition $P = (T, A)$ equals the cardinality of the largest bag in T minus one. The path-width of G, denoted by $pwd(G)$, is the minimum width over all path decompositions of G.

Path-width itself is a powerful (albeit extremely restrictive) width parameter. However, computing path-width is a hard problem – it remains NP-hard even when restricted to weighted trees and distance hereditary graphs (graphs of rank-width 1) [8]. This is another example of a problem where the linearity restriction helps: it is in fact possible to compute the path-width of graphs of linear rank-width 1 in polynomial time.

Lemma 4.10. *It is possible to compute the path-width of a connected thread graph G consisting of a single cluster in polynomial time. The computed minimal path-decomposition will furthermore have the property that $\mathcal{R}_1$ is present in the first bag and $\mathcal{R}_2$ in the last bag.*

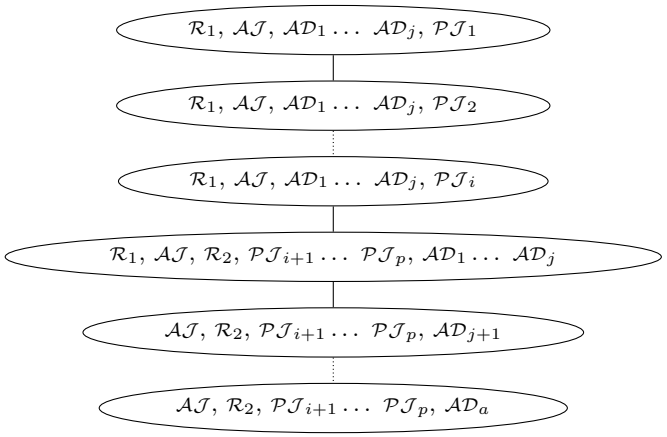

Fig. 5: Optimal path-decomposition for connected thread graphs with p $\mathcal{PJ}$ and a $\mathcal{AD}$ vertices; See Appendix for proof.

Theorem 4.11. *There exists a polynomial time algorithm for computing the path-width of thread graphs.*

Proof. Our input graph G may again be assumed to be connected, since the path-width of disconnected graphs equals the maximum of their components. Then G is formed by a sequence of clusters and for each cluster we may compute a minimal path-decomposition with the appropriate $\mathcal{R}$ vertices in the first and last bag. The path-width of G equals the maximum of the path-width of its clusters, since sequencing path-decompositions for the clusters results in a path-decomposition for the whole G. □

5 Conclusion

The main contribution of the article may be summarized in two main points. First, it gives a constructive characterization of graphs of linear rank-width 1 and provides insight into the structure of such graphs, which we call thread graphs. This new graph class contains paths and cliques, but is also much more general. Second, the article uses the obtained results in the design of new polynomial algorithms for bandwidth, dominating bandwidth and path-width on thread graphs. Each of these problems remains hard on other well-known classes of graphs, such as distance hereditary graphs and trees. Further research in this area should focus on possible parameterized algorithms on linear rank-width – it is not clear whether or how our polynomial algorithms might be extended to graphs of bounded linear rank-width.

Acknowledgment

The author wishes to thank Jan Obdržálek for his contribution and help in characterizing graphs of linear rank-width 1.

References

1. P. Chinn, J. Chvátalová, A. Dewdney, and N. Gibbs. The bandwidth problem for graphs and matrices—a survey. *J. Graph Theory*, 6:223–254, 1982.
2. B. Courcelle and S. Olariu. Upper bounds to the clique width of graphs. *Discrete Appl. Math.*, 101(1-3):77–114, 2000.
3. R. Ganian and P. Hliněný. Better polynomial algorithms on graphs of bounded rank-width. In *IWOCA'09*, volume 2874 of *LNCS*, pages 266–277. Springer, 2009.
4. R. Ganian and P. Hliněný. On parse trees and Myhill–Nerode–type tools for handling graphs of bounded rank-width. *Discrete Appl. Math.*, 2009. To appear.
5. A. Gupta. Improved bandwidth approximation for trees and chordal graphs. *J. Algorithms*, 40(1):24–36, 2001.
6. P. Hliněný and S. Oum. Finding branch-decomposition and rank-decomposition. *SIAM J. Comput.*, 38:1012–1032, 2008.
7. S. il Oum. Rank-width is less than or equal to branch-width. *J. Graph Theory*, 57(3):239–244, 2008.
8. R. Mihai and I. Todinca. Pathwidth is np-hard for weighted trees. In *FAW*, volume 5598 of *LNCS*, pages 181–195. Springer, 2009.
9. S. Oum and P. Seymour. Approximating clique-width and branch-width. *J. Combin. Theory Ser. B*, 96(4):514–528, 2006.
10. N. Robertson and P. D. Seymour. Graph minors. II. Algorithmic aspects of tree-width. *Journal of Algorithms*, 7(3):309–322, September 1986.
11. W. Unger. The complexity of the approximation of the bandwidth problem. In *IEEE Symposium on Foundations of Computer Science*, pages 82–91, 1999.
12. Q. Zhu, Z. Adam, V. Choi, and D. Sankoff. Generalized gene adjacencies, graph bandwidth, and clusters in yeast evolution. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 6(2):213–220, 2009.

Appendix

Proof of Theorem 2.4:

For cliques, we simply construct a linear rank-decomposition by adding vertices in an arbitrary order – all bipartite adjacency matrices will be filled with 1s. For paths it suffices to introduce vertices in the same order as they appear on the path. The matrices will then always contain a single row with a single 1 and the rest being rows filled with 0s.

For trees, assume there exists a minimal number c such that all trees have linear rank-width at most c. Thus there exists a tree T_c such that it has linear rank-width c and contains at least 2 vertices. We show it is possible to construct a tree T_{c+1} such that its linear rank-width is greater than c, contradicting the minimality of c. So, let T_{c+1} consist of a single vertex r connected by $c+2$ edges to any vertex of $c+2$ copies of T_c.

Now assume there exists a c-width linear rank-decomposition of T_{c+1}. Let us look at the order of vertices as they appear from the end of the linear rank-decomposition which is farther from r. We say that a copy of T_c is incomplete if some but not all of its vertices have appeared so far in the linear rank-decomposition. Notice that if there were more than c incomplete copies of T_c at any point in the rank-decomposition, the width of the decomposition would exceed c. Since the linear rank-width of T_c is c, it also cannot happen that some T_c is completed while any other T_c is incomplete (an incomplete copy of T_c will increase the rank of the matrix by at least 1).

However, after the first T_c is completed, if we try to complete a second T_c then a bipartite adjacency matrix of rank $c+1$ has to be present at some edge of the decomposition – since T_c has linear rank-width c, the submatrix between vertices of the second copy of T_c must have rank c at some edge of the decomposition, and additionally there will be a unique "1" entry between r and the vertex of the first T_c it is connected to. Now, three cases may occur: Either the vertex in the second T_c connected to r is a column, or it is a row and all its neighbours in T_c are rows, or it is a row and at least one of its neighbours in T_c is a column. In each of these cases the rank of the matrix goes up by one, thus concluding our proof. □

Proof of Theorem 3.5:

Since any disconnected thread graph may be created by sequencing the creating sequences of its connected components, we may assume that G is connected. The first step is to check for every vertex whether deleting it disconnects the graph – if it does, then it is a $\mathcal{R}$ vertex. Since G is connected, all $\mathcal{R}$ vertices are automatically $\mathcal{AJ}$, and all $\mathcal{R}$ vertices form a path.

This way we have identified all $\mathcal{R}$ vertices except for the first and last (which are also the first and last vertices of the creating sequence). While there exists a non-trivial method for determining the first and last $\mathcal{R}$ vertices, it is possible

to simply try all neighbours of the endpoints of the path of $\mathcal{R}$ vertices by brute force in polynomial time.

Once we have all the reset vertices ordered, we proceed by identifying the creating sequence of each cluster. Assume we have completed C_{i-1}. Then all neighbours of $\mathcal{R}_i$ which are not in C_{i-1} are $\mathcal{PJ}$ iff they are not adjacent to $\mathcal{R}_{i+1}$ (otherwise they are $\mathcal{AJ}$). Finally, all neighbours of $\mathcal{R}_{i+1}$ which are not adjacent to other $\mathcal{R}$ vertices are $\mathcal{AD}$ iff they are adjacent to some $\mathcal{PJ}$ or $\mathcal{AJ}$ vertex in C_i.

Now that each vertex in the cluster has its attributes, determining the creating sequence is simple. Whenever a $\mathcal{PJ}$ vertex is not adjacent to any not-yet-added $\mathcal{AJ}$ or $\mathcal{AD}$ vertices, we create it – otherwise, we create an $\mathcal{AJ}$ or $\mathcal{AD}$ vertex which has the lowest number of remaining $\mathcal{PJ}$ neighbours. Determining which of these $\mathcal{AJ}$ or $\mathcal{AD}$ vertices to create first then depends on the edges between them; a $\mathcal{AD}$ vertex is created before a $\mathcal{AJ}$ one iff there is no edge between them.

□

Proof of Theorem 4.7:

The proof is a reduction from the bandwidth problem on trees. Given an input tree T for the bandwidth problem, we construct a tree T' by first subdividing every edge of T twice and then adding a pendant vertex to each leaf. There exists a single minimum dominating set X on T', and $X = V(T)$. The bandwidth problem on T is then exactly the same as the dominating bandwidth problem on T' – each $v \in V(T') - X$ is adjacent to precisely one dominating vertex so these must have the same bandwidth value, and two vertices of T are adjacent in T iff two vertices of T' which are neighbours of some dominating vertices are adjacent.

□

Proof of Lemma 4.10:

It is a well known fact that all vertices in a clique must be in some bag of P, so there exists some bag B containing all $\mathcal{AJ}$ vertices together with $\mathcal{R}_1$ and $\mathcal{R}_2$. Thus $pw(G)$ must be at least $|B|$, however it may also be much larger – for example $\mathcal{AD}$ and $\mathcal{PJ}$ vertices could form an arbitrarily large bipartite clique together.

If there were no edges between $\mathcal{AD}$ and $\mathcal{PJ}$ vertices, we could create a path-decomposition of width $|B|$ by having one bag B, on one side for each $\mathcal{PJ}$ vertex a bag containing the $\mathcal{PJ}$ vertex and $B - \{\mathcal{R}_2\}$ and on the other the same for $\mathcal{AD}$ vertices without $\mathcal{R}_1$. So, assume there is some $\mathcal{PJ}$ vertex v and $\mathcal{AD}$ vertex u such that $\{u, v\} \in E$. This implies that u was created before v, and so each $\mathcal{AJ}$ vertex is adjacent to v (it was created before v) or to u (it was created after u). Then there exists some bag $B_u = B \cup \{u\}$ or $B_v = B \cup \{v\}$. Indeed, either B is between u and v in the path-decomposition, in which case one of the vertices needs to "travel" to meet the other, or u and v lie on the same side from B in the decomposition, but then each vertex of B must "travel" to u or v.

Now let us consider some set of $\mathcal{AD}$ vertices X and $\mathcal{PJ}$ vertices Y of those vertices which appear in some bag with B. If X and Y contain all $\mathcal{AD}$ and $\mathcal{PJ}$ vertices, then for any such path-decomposition P we may create a path decomposition P' as follows:

1. Trivially alter P so that each bag introduces at most one new vertex.
2. Find the vertex $z \in X \cup Y$ which appears last in some bag of P.
3. Remove $\mathcal{R}_2$ (if z is $\mathcal{PJ}$) or $\mathcal{R}_1$ (if z is $\mathcal{AD}$) from all bags containing z.

The width of P' is at most that of P, and so we may assume that $X \cup Y$ do not contain all $\mathcal{AD}$ and $\mathcal{PJ}$ vertices.

So let there be some $\mathcal{PJ}$ vertex $q \notin Y$ (the proof for $\mathcal{AD}$ $q \notin X$ is analogous). Then the closest bag to q on any such path-decomposition containing B, all bags between them and the first bag containing q contain all neighbours of q by interpolation, and no other bags with q are necessary. More importantly though, any $\mathcal{PJ}$ vertex q' created before q now has all its neighbours in the bag containing q, and so we may simply create a new bag next to that one with q' instead of q – this takes care of all edges incident to q', so no other bags containing q' are necessary and the size of the newly created bag may not increase the width of the path-decomposition. Thus it is safe to assume that if $q \notin Y$ (X for $\mathcal{AD}$), then all q' created before (after) q are not in Y (X) either.

Now finally we can give the algorithm for computing path-width. At the top level, we loop through at most $|V|$ possible choices for Y, each corresponding to the first $\gamma = 1 \ldots |\mathcal{PJ}|$ vertices not being in Y. For each γ we minimize X by only assuming neighbours of the first γ $\mathcal{PJ}$ vertices are in X. Since every vertex in Y is adjacent to every vertex in X now (remember that $\mathcal{PJ}$ vertices in Y were created after γ) and every vertex in X is present in the nearest bag to γ containing B, there is a bag containing X, Y and B in the decomposition. Additionally, all other vertices may be dealt with without increasing the width of the decomposition (as per q' in the previous paragraph), and therefore the path-width of G is the minimum of $|X| + |Y| + |B|$ over all choices of γ.

□