TU Delft

Delft University of Technology

Document Version
Final published version

Licence
CC BY

Citation (APA)
Holtgrefe, N., van Iersel, L., & Jones, M. (2026). Exact and heuristic computation of the scanwidth of directed acyclic graphs. *Journal of Computer and System Sciences*, *160*, Article 103802. https://doi.org/10.1016/j.jcss.2026.103802, https://doi.org/10.1016/j.jcss.2026.103802

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
In case the licence states "Dutch Copyright Act (Article 25fa)", this publication was made available Green Open Access via the TU Delft Institutional Repository pursuant to Dutch Copyright Act (Article 25fa, the Taverne amendment). This provision does not affect copyright ownership.
Unless copyright is transferred by contract or statute, it remains with the copyright holder.

Sharing and reuse
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.

Journal of Computer and System Sciences 160 (2026) 103802

Contents lists available at ScienceDirect

Journal of Computer and System Sciences

journal homepage: www.elsevier.com/locate/jcss

Exact and heuristic computation of the scanwidth of directed acyclic graphs ☆

Niels Holtgrefe [a,*], Leo van Iersel [a], Mark Jones [b]

[a] *Delft Institute of Applied Mathematics, Delft University of Technology, Delft, the Netherlands*
[b] *Department of Computer Science, Middlesex University, London, United Kingdom*

ARTICLE INFO

Article history:
Received 5 February 2024
Received in revised form 17 February 2026
Accepted 22 March 2026
Available online 31 March 2026

Dataset link: https://github.com/nholtgrefe/scanwidth

Keywords:
Scanwidth
Width measure
Directed acyclic graphs
Phylogenetic networks
Parametrized algorithms

ABSTRACT

To measure the tree-likeness of a directed acyclic graph (DAG), a new width parameter that considers the directions of the arcs was recently introduced: *scanwidth*. We present the first algorithm that efficiently computes the exact scanwidth of general DAGs. For DAGs with one root and scanwidth k it runs in $O(k \cdot n^k \cdot m)$ time. The algorithm also functions as an FPT algorithm with complexity $O(2^{4\ell-1} \cdot \ell \cdot n + n^2)$ for phylogenetic networks of level-ℓ, a type of DAG used to depict evolutionary relationships among species. Our algorithm performs well in practice, being able to compute the scanwidth of synthetic networks up to 30 reticulations and 100 leaves within 500 seconds. Furthermore, we propose a heuristic that obtains an average practical approximation ratio of 1.5 on these networks. While we prove that the scanwidth is bounded from below by the treewidth of the underlying undirected graph, experiments suggest that for networks the parameters are close in practice.
© 2026 The Authors. Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

1. Introduction

Treewidth is an extensively researched width measure that quantifies the 'tree-likeness' of an undirected graph. The parameter comes with a corresponding tree decomposition, which represents the graph in a tree-like way (see e.g. [1]). When considering directed acyclic graphs (DAGs), tree decompositions become less natural since they do not preserve the natural top-to-bottom structure of a DAG. With this in mind, Berry, Scornavacca, and Weller [2] recently developed a DAG-specific measure of tree-likeness: scanwidth. Contrary to treewidth, scanwidth is not agnostic to the directions of the arcs, thus making it a more natural parameter for DAGs.

Berry, Scornavacca, and Weller motivate the use of scanwidth from the perspective of phylogenetics: the study of evolutionary relationships among different genes, species, or populations. These relationships can be represented by a type of DAG with a single root: a phylogenetic network. The leaves of such a network represent the studied set of species (or taxa), whereas the root is a common ancestor. The arcs and their directions depict how species evolve over time, while the internal vertices are either reticulate events (where multiple species converge), or speciation events (where a species diverges into multiple species).

Phylogenetics is a research area with many computationally hard problems such as Network Inference [3], Tree Containment [4,5], Small Parsimony [6] and Hybridization Number [7,8]. A common way to solve such problems is by using

☆ This paper received funding from the Dutch Research Council (NWO) under projects OCENW.M.21.306 and OCENW.KLEIN.125.
* Corresponding author.
E-mail addresses: N.A.L.Holtgrefe@tudelft.nl (N. Holtgrefe), L.J.J.vanIersel@tudelft.nl (L. van Iersel), M.Jones@mdx.ac.uk (M. Jones).

https://doi.org/10.1016/j.jcss.2026.103802
0022-0000/© 2026 The Authors. Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

(a) DAG with single root

(b) Scanning of the DAG

Fig. 1. A DAG with a single root (a), and a tree extension of the DAG (b), functioning as a route for the scanner lines. The tree extension is indicated by the grey edges, while the arcs of the DAG are drawn back in, following the edges of the tree extension. The scanner lines start at the leaves and move up through the tree extension, at each step becoming brighter red. The tree extension is optimal, thus the scanwidth of this DAG is 3: the maximum number of DAG arcs that are cut by one of the scanner lines. (For interpretation of the colors in the figures, the reader is referred to the web version of this article.)

so-called parameterized algorithms, where the time complexity is expressed as a function of both the input size and an additional parameter that is small in practice. In phylogenetics these algorithms tend to exploit the observation that reticulate events are fairly rare for most practical phylogenetic networks (when compared to the overall size of the network). In some sense, these networks thus still remain somewhat tree-like. Two well-known parameters in phylogenetics that measure this tree-likeness are the reticulation number and level of a network (see e.g. [9]). Both measures have seen a vast amount of successful uses as a parameter in parameterized algorithms (see e.g. [5,8] and the survey [10]).

Since the treewidth is smaller than both the reticulation number and the level, it has recently attracted interest from the phylogenetics community. Although treewidth has already successfully been applied for parameterized algorithms in phylogenetics [4,6], using scanwidth seems to be more natural, thus allowing for a more intuitive design of parameterized algorithms [6]. Indeed, algorithms in phylogenetics relying on scanwidth are already starting to appear [3].

Berry, Scornavacca, and Weller named scanwidth after the informal concept of ‘scanning’ a DAG. Imagine a ‘scanner line’ for each leaf of a DAG. As these scanner lines move up through the DAG, they scan the arcs of the DAG. A scanner line merges with another scanner line when they meet at a vertex. The order in which the arcs and vertices are scanned is determined by a tree extension: a tree on the same vertex set as the original DAG, with the constraint that it maintains the natural ordering of the DAG. Therefore, this tree extension functions as a route for the scanner lines. The goal is now to find a tree extension that minimizes the maximum number of arcs that are cut by a line during the scanning. This number is referred to as the scanwidth of the DAG. Fig. 1 provides an illustration of the concept of scanning a DAG.

An NP-hardness proof and some structural results were already presented in [2]. Apart from that, scanwidth has only very briefly been mentioned in [3,6]. Magne et al. [11] independently introduced the closely related edge-treewidth, which can be considered the undirected analogue of scanwidth. Similar to scanwidth, edge-treewidth has not seen other research efforts yet. A second closely related parameter is the directed cutwidth (see e.g. [12]). Its solution space is more restrictive than it is for scanwidth since it only allows linear orderings instead of tree extensions. Consequently, most results for directed cutwidth are not immediately transferable to scanwidth.

This paper aims to advance the understanding of scanwidth, with a special focus on exact and heuristic algorithms to compute the NP-hard parameter. The tree extensions returned by these algorithms can then be used by future algorithms relying on scanwidth as a parameter.

The structure of the paper is as follows. The first three subsections of Section 2 contain the formal introduction to scanwidth as well as some preliminary graph theoretical notions. In Subsection 2.4 we prove that the scanwidth is bounded from below by the treewidth. Although this bound is already mentioned in [2], it has not previously been formally proved. We also generalize the result by Rabier et al. [3], that the scanwidth of a binary level-k network is at most $k+1$, to non-binary networks. Section 3 discusses some reduction rules to decrease the size of instances.

Section 4 is focuses on exact algorithms that compute the scanwidth. Using a recursive approach, the $O(n! \cdot n \cdot m)$ brute force time complexity is first improved to $O(4^n \cdot \text{polylog}(n))$. This algorithm forms the basis of an algorithm with running time $O((k+r-1) \cdot m \cdot n^{k+r-1})$ for DAGs with r roots, where k is the scanwidth. This proves that with respect to the parameter scanwidth, the scanwidth problem for DAGs with a fixed number of roots falls within the complexity class XP, containing problems solvable in *slicewise polynomial* time: $n^{f(k)}$ time for some computable function f. Using reduction rules, this same algorithm can also be parametrized by the level ℓ of a network, running in $O(2^{4\ell-1} \cdot \ell \cdot n + n^2)$ time. Therefore, with the level as parameter, the scanwidth problem is in the class of *fixed-parameter tractable* (FPT) problems, which contains problems solvable in $f(\ell) \cdot n^c$ time for some computable function f and constant c.

Section 5 explores different heuristics, with a cut-splitting heuristic showing positive results. Although the heuristic is proved to be non-optimal in general, computational experiments in Section 6 are promising. They show that the heuristic attains an average practical approximation ratio of 1.5 for networks of 30 reticulations and 100 leaves, when enhanced with simulated annealing. The XP algorithm is shown to be the fastest exact computation method in practice, being able

to compute the scanwidth of networks up to 30 reticulations and 100 leaves within 500 seconds. Moreover, 88.9% of those networks are solvable within 60 seconds.

Finally in Section 7, we discuss the results and directions for further research.

2. Preliminaries

We will use the big-Oh notation to analyze complexities of algorithms. Additionally, we use the less common $\tilde{O}$ to suppress the polynomial and logarithmic factors in the time complexity, resulting in, for example, $O(2^n \cdot n^3 \cdot \log n) = \tilde{O}(2^n)$. Regarding graph theoretical concepts, we mostly follow the standard notation as presented in [1]. Below, we will present some additional conventions and possibly less common notions.

Similarly to the degree $\delta(v)$ of a vertex v, we write $\delta(W)$ for the degree of $W \subseteq V$, which counts the number of edges with one endpoint in W and one endpoint in $V \setminus W$ for a graph $G = (V, E)$. To avoid confusion, we sometimes use $\delta_G(v)$ or $\delta_G(W)$ to emphasize the graph G. We write $u \overset{G}{\leftrightsquigarrow} v$ if two vertices u and v are connected (i.e. there exists a path between them) in a graph G. A *block* (or 'biconnected component') is a maximal connected induced subgraph without any cut vertices (i.e. vertices whose removal increases the number of connected components).

A *weakly connected* directed graph is a graph whose underlying undirected graph is connected. Analogously, two vertices u and v are weakly connected in G (denoted by $u \overset{G}{\leftrightsquigarrow} v$) if they are connected in the underlying undirected graph. A *(weakly connected) component* of G is a maximal weakly connected induced subgraph of G. A vertex in a directed graph is a *cut vertex* if its deletion increases the number of weakly connected components. A *block* of a directed graph is a maximal weakly connected induced subgraph without any cut vertices.

Directed acyclic graphs If a directed graph G contains no directed cycles, we call it a *directed acyclic graph* (DAG). In a DAG, a vertex with indegree 0 is called a *root* (often labeled as ρ), and a vertex with outdegree 0 is referred to as a *leaf*. If G has exactly one root, we call G *rooted*. Otherwise, if G has multiple roots, it is *multi-rooted*. The tails of arcs that enter a vertex v are the *parents* of v. Similarly, the heads of arcs coming out of v are *children* of v. We call $W \subseteq V$ a *sinkset* if $\delta^{\text{out}}(W) = 0$. We write $W \sqsubseteq U$ for any $U \subseteq V$ if both $W \subseteq U$ and W is a sinkset. Since a DAG contains no directed cycles, it naturally exhibits a top-to-bottom structure. More formally, it defines a partial order on its vertices: we write $v <_G u$ if there exists a directed path from u to v.

Unless otherwise specified, this paper will consider each graph G to be a weakly connected, directed acyclic graph without self-loops and parallel arcs. If it is clear from the context, we sometimes drop the adjective 'directed' from the notions defined above.

Phylogenetic networks A rooted, weakly connected DAG $G = (V, E)$ is a *(rooted) network* if each vertex $v \in V$ is of one of the following types: (i) (unique) *root* with $\delta^{\text{in}}(v) = 0$; (ii) *leaf* with $\delta^{\text{in}}(v) = 1$ and $\delta^{\text{out}}(v) = 0$; (iii) *tree-vertex* with $\delta^{\text{in}}(v) = 1$ and $\delta^{\text{out}}(v) \geq 2$; (iv) *reticulation (vertex)* with $\delta^{\text{in}}(v) \geq 2$ and $\delta^{\text{out}}(v) = 1$. Furthermore, if the root has degree 2, the leaves degree 1, and all other vertices degree 3, G is a *binary network*.

The *reticulation number* $r(G)$ of a network $G = (V, E)$ is the sum of indegrees of all reticulation vertices, minus the number of reticulation vertices. A network has *level k* (or is level-k) if each block of the network has reticulation number at most k. The reticulation number and level of a network are often only defined for binary networks but were generalized to non-binary networks in [9].

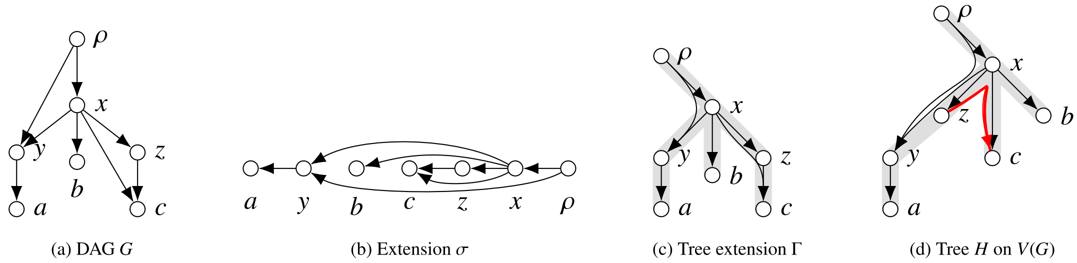
A network N is a *phylogenetic network* on a set of labels X if its leaves are bijectively labeled by the elements of X. As an example, consider the graph from Fig. 1a, which serves as a phylogenetic network on the set of labels $\{a, b, c, d\}$.

2.1. Graph layouts

A *(linear) layout* σ (also known as a 'linear ordering', 'linear arrangement', 'numbering', or 'labeling') of a graph G is a total ordering of its vertex set. This ordering can be represented by a directed path on $V(G)$. A *tree layout* Γ (also known as an 'agreeing tree') of a graph G is a partial ordering of its vertex set with a unique largest element, and the constraint that for all $uv \in E(G)$, the vertices u and v must be comparable in Γ (i.e. $u <_\Gamma v$ or $v <_\Gamma u$). It is represented by a rooted directed tree on $V(G)$, where the root corresponds to this largest element. Due to the constraint, edges of the graph are not allowed to 'cross' different branches of the tree layout.

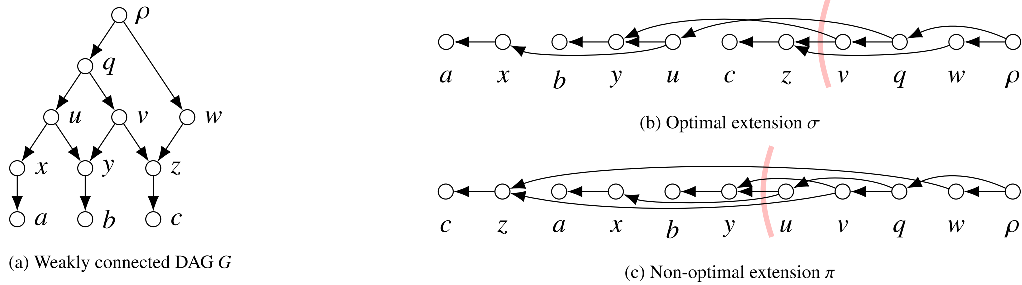
A linear layout σ (resp. tree layout Γ) of a DAG G is *G-respecting* if $u <_G v$ implies $u <_\sigma v$ (resp. $u <_\Gamma v$) for all $u, v \in V(G)$. A G-respecting linear layout (resp. tree layout) of G is called an *extension* (resp. *tree extension*) of G. Consequently, all arcs of a DAG point in the same direction when drawn in a (tree) extension. For an extension, the arcs point backwards (i.e. towards the first element of the ordering), while in a tree extension, the arcs point away from the root. Fig. 2 serves as a visualization of the above concepts. As a convention, we will always draw (tree) extensions as presented in this figure. To avoid confusion, Fig. 2d shows a graph H that may mistakenly be interpreted as a tree extension.

We will now cover some notation, partially adopted from [2], on the above notions. Throughout this paper, we will exclusively reserve the Greek letters σ and π for extensions, and the Greek capital letters Γ and Ω for tree extensions. For a weakly connected DAG G (resp. a connected undirected graph G) and $U \subseteq V(G)$, we use $\Pi[U]$ to denote the set of



(a) DAG G (b) Extension σ (c) Tree extension Γ (d) Tree H on $V(G)$

Fig. 2. (a): Weakly connected DAG G. (b): An extension σ of G with the arcs of G also drawn. (c): A tree extension Γ of G indicated by the grey arcs, whose direction is downwards. The arcs of G are also drawn in Γ and are made to follow the grey arcs. (d): A tree H on $V(G)$ that is not a tree extension, because z and c are not comparable in H, while they are adjacent in G. Visually this means that the corresponding thick red arc zc of G 'crosses' two branches of the tree.



(a) Weakly connected DAG G (b) Optimal extension σ (c) Non-optimal extension π

Fig. 3. (a): Weakly connected DAG G. (b): An optimal extension σ of G with cutwidth 4, attained at the red cut. (c): A non-optimal extension π of G with cutwidth 5, attained at the red cut.

extensions (resp. linear layouts) of $G[U]$. Thus, $\Pi[V(G)]$ is the set of all extensions (resp. linear layouts) of G. Similarly, we use $\mathcal{T}[U]$ to denote the set of tree extensions (resp. tree layouts) of $G[U]$.

The vertex at position i of a layout σ is denoted by $\sigma(i)$. The suborder of σ starting at position i till position j is written as $\sigma[i\dots j]$, while the order starting at i till the last vertex is $\sigma[i\dots]$. The restriction of an ordering to a subset $U \subseteq V(G)$ is written as $\sigma[U]$. If A and B are two disjoint subsets of $V(G)$, and σ (resp. π) is a layout of $G[A]$ (resp. $G[B]$), we write $\sigma \circ \pi$ for the concatenation of σ and π (that is, σ followed by π). Note that the positions and vertices of a linear layout are in bijection. Therefore, we will sometimes treat the vertices as interchangeable with their position in the linear layout. In particular, we will write $\sigma[1\dots v]$ to denote $\sigma[1\dots\sigma^{-1}(v)]$ and $\sigma[v\dots]$ to denote $\sigma[\sigma^{-1}(v)\dots]$. For a vertex set V, we also write $[V] = \{1,\dots,|V|\}$. Since subgraphs of the type $G[\sigma[1\dots i]]$ will appear throughout this paper, we often denote them as $G[1\dots i]$ if the extension σ is clear from the context.

2.2. Cutwidth

Following [2], we will first define cutwidth, which will make it easier to explain scanwidth. Cutwidth is a width parameter for graphs that has seen a lot of attention since the 1970s (see the survey [13] and its addendum [14]). Multiple variants exist, but we focus on the specific version of the parameter for DAGs. There is no consensus on the naming of this DAG-variant of cutwidth. In [15] it is referred to as 'minimum cutwidth for directed acyclic graphs'. Other authors call it 'directed cutwidth' [2,12]. For the sake of brevity, we will refer to it simply as *cutwidth*.

Definition 2.1 *(Cutwidth).* Let $G = (V, E)$ be a weakly connected DAG. For an extension σ and a position i of σ, we will denote $CW_i^\sigma = \{uv \in E : u \in \sigma[i+1\dots], v \in \sigma[1\dots i]\}$. Then the *cutwidth* of G is $\mathrm{cw}(G) = \min_{\sigma\in\Pi[V]} \max_{i\in[V]} |CW_i^\sigma|$. Furthermore, we let $\mathrm{cw}(\sigma, G) = \max_{i\in[V]} |CW_i^\sigma|$ be the cutwidth of σ, where $|CW_i^\sigma|$ is the cutwidth of σ at position i.

Intuitively, an extension of a DAG is considered optimal in terms of cutwidth if the maximum number of arcs crossing a gap between two vertices is as small as possible. An example of an optimal and a non-optimal extension in terms of cutwidth is shown in Fig. 3.

It is implicitly shown in [2] that computing the cutwidth of a DAG is NP-hard. Regarding exact algorithms, one can compute the cutwidth of a DAG in $\tilde{O}(2^n)$ time [15]. The theoretically fastest parametrized algorithm for the cutwidth of a DAG was introduced in [12], and it runs in linear time for fixed cutwidth of k, showing that the problem is FPT.

2.3. Scanwidth

In the introductory section of this paper, we provided some intuition on the idea of 'scanning' a phylogenetic network. This concept can be extended to all weakly connected DAGs. Throughout this paper we aim to provide results for this broader class of graphs.[1] With cutwidth in mind, we are ready to define scanwidth. The scanwidth of a DAG was formally introduced by Berry, Scornavacca, and Weller [2] as follows.

Definition 2.2 *(Scanwidth).* Let $G=(V,E)$ be a weakly connected DAG. For an extension σ and a position i of σ, we will denote $\text{SW}_i^\sigma(G)=\{uv\in E : u\in\sigma[i+1\ldots], v\in\sigma[1\ldots i], v \overset{G[1\ldots i]}{\rightsquigarrow} \sigma(i)\}$. Then the *scanwidth of* G is

$$\text{sw}(G)=\min_{\sigma\in\Pi[V]}\max_{i\in[V]}|\text{SW}_i^\sigma(G)|.$$

Furthermore, we let $\text{sw}(\sigma,G)=\max_{i\in[V]}|\text{SW}_i^\sigma(G)|$ be the scanwidth of σ, where $|\text{SW}_i^\sigma(G)|$ is the scanwidth of σ at position i. If G is clear from the context, we mostly write SW_i^σ instead of $\text{SW}_i^\sigma(G)$.

This definition is closely related to the definition of cutwidth. However, instead of counting all arcs in the cut-set CW_i^σ, we only count those arcs entering a vertex v that is weakly connected to $\sigma(i)$ in the graph $G[\sigma[1\ldots i]]$. Recall that we write this as $v \overset{G[1\ldots i]}{\rightsquigarrow} \sigma(i)$. Before illustrating this definition with Fig. 4, we introduce an alternative characterization of scanwidth.

The previous definition involves extensions and will turn out the be more convenient in proofs, as induction-based proofs are often easier when iterating over the positions of an extension. In [2, Prop. 1] it is shown that the next definition equivalently defines scanwidth.[2] This alternative definition relies on tree extensions, thus aligning more closely with the 'scanning'-intuition given in the introduction.

Definition 2.3 *(Scanwidth).* Let $G=(V,E)$ be a weakly connected DAG. For a tree extension Γ and a vertex v of V, we will denote $\text{GW}_v^\Gamma(G)=\{xy\in E : x>_\Gamma v\geq_\Gamma y\}$. Then the *scanwidth* of G is

$$\text{sw}(G)=\min_{\Gamma\in\mathcal{T}(V)}\max_{v\in V}|\text{GW}_v^\Gamma(G)|.$$

Furthermore, we let $\text{sw}(\Gamma,G)=\max_{v\in V}|\text{GW}_v^\Gamma(G)|$ be the scanwidth of Γ, where $|\text{GW}_v^\Gamma(G)|$ is the scanwidth of Γ at vertex v. If G is clear from the context, we mostly write GW_v^Γ instead of $\text{GW}_v^\Gamma(G)$.

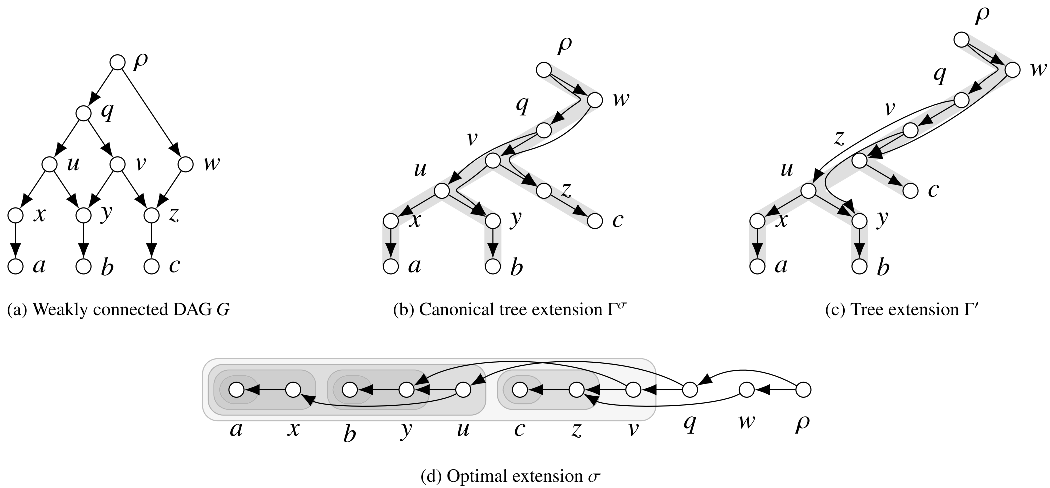
To illustrate the two definitions and their relation, we take a look at Fig. 4. Fig. 4a depicts the same weakly connected DAG as in Fig. 3, while Fig. 4b shows a tree extension Γ^σ (the thick grey arcs, whose direction is downwards) with the arcs of the original DAG drawn in it. For each vertex v in the graph, the set $\text{GW}_v^{\Gamma^\sigma}$ contains the arcs that enter the vertex or pass it to reach a vertex lower in the tree extension. Visually, these sets correspond to cuts in the tree extension. As mentioned in the introduction, scanwidth can thus be viewed as a tree analogue of cutwidth. One can quickly check that the scanwidth of this (optimal) tree extension Γ^σ equals 3, which is attained at the vertex v, where we have that $\text{GW}_v^{\Gamma^\sigma}=\{qv,qu,wz\}$. In a similar fashion, Fig. 4c shows a tree extension of the same graph, with a non-optimal scanwidth of 4, attained at the vertex z.

We just stated that the scanwidth of the graph G in Fig. 4a equals 3, meaning that there also exists an extension with a scanwidth of 3 when considering Definition 2.2. One such extension σ is shown in Fig. 4d. In comparison to tree extensions, testing membership of an arc in one of the sets SW is slightly more involved. It requires knowledge of the weak connectivity relations within subgraphs of the graph. In Fig. 4d these relations are depicted by the grey shaded areas. Consider for example the vertex $z=\sigma(7)$. From the figure we can see that $\{a,x,b,y,u\}$ and $\{c,z\}$ form the two components of $G[1\ldots 7]$. Therefore, $\text{SW}_z^\sigma=\{vz,wz\}$, and the set does not contain the arcs vy and qu, as they enter the other component. Compare this to the cutwidth of this extension, where these two arcs would also be counted (see Fig. 3b).

Upon further examination of the example in Fig. 4, an interesting observation emerges. Note that the sets GW associated with the tree extension Γ^σ and the sets SW associated with the extension σ coincide at each vertex. This is not coincidental, since Γ^σ is the *canonical tree extension* for σ, a crucial notion from [2] used to prove the equivalence between the two scanwidth definitions. First, recall that the *transitive reduction* of G is another graph H on the same vertex set and with as few arcs as possible, such that for all pairs of vertices u,v, there exists a directed path from u to v in G if and only if there exists a directed path from u to v in H. It is a well-known fact that the transitive reduction of a DAG is unique and can be obtained by exhaustively deleting arcs uv for which there is a directed path from u to v containing at least one other vertex. Formally, the canonical tree extension is now defined as:

[1] Most results in this paper, if not all, generalize to disconnected graphs by considering a 'forest extension', which contains a separate tree extension for each component of the graph.

[2] Berry, Scornavacca, and Weller prove the equivalence only for networks, but the proof does not rely on labeled leaves and a single root. We can therefore extend the equivalence to arbitrary weakly connected DAGs.



(a) Weakly connected DAG G (b) Canonical tree extension Γ^σ (c) Tree extension Γ'

(d) Optimal extension σ

Fig. 4. (a): Weakly connected, rooted DAG G. (b): Optimal canonical tree extension Γ^σ with scanwidth 3, attained at the vertex v. (c): Non-canonical tree extension Γ' with scanwidth 4, attained at the vertex z. (d): Optimal extension σ with scanwidth 3, attained at the vertex v. For each $i \leq 8$, the outermost grey shaded areas containing only vertices belonging to $\sigma[1\dots i]$ depict the weakly connected components of $G[1\dots i]$. For $i \geq 8$, $G[1\dots i]$ is weakly connected and therefore consists of just one component.

Definition 2.4 *(Canonical tree extension).* Let $G = (V, E)$ be a weakly connected DAG and σ an extension of G. Then, we denote the *canonical tree extension* for σ as Γ^σ, and it is defined as the transitive reduction of the DAG $H = (V, \{uv : u >_\sigma v, u \overset{G[1\dots u]}{\leftrightsquigarrow} v\})$.

Lemma 5 from [2] establishes the relation between canonical tree extensions and extensions. It shows that Γ^σ is indeed a tree extension of G that has the same scanwidth as σ. Moreover, any extension of the canonical tree extension has the same scanwidth again. Note that such an extension can easily be found in linear time with a breadth-first-search, given that the tree extension is canonical.

Using Lemma 5 from [2] and the fact that the sets GW_v^Γ uniquely determine the tree extension Γ (Lemma A.1), it is possible to completely characterize the canonical tree extensions by a more easily checkable condition. The proof of the result is deferred to Appendix A. Recall that for a tree extension Γ of G and a vertex v, the graph $G[V(\Gamma_v)]$ is the subgraph of G induced by all vertices in the subtree of Γ rooted at v.

Proposition 2.5. *Let $G = (V, E)$ be a weakly connected DAG, Γ a tree extension of G, and σ an extension of Γ. For each $v \in V$, let Γ_v be the subtree of Γ rooted at v. Then, Γ is the canonical tree extension for σ if and only if $G[V(\Gamma_v)]$ is weakly connected for all $v \in V$.*

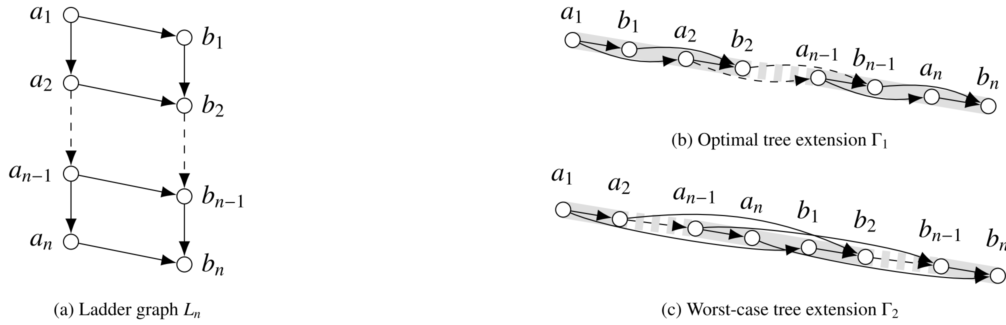
Recalling the examples from Fig. 4, we can now quickly deduce that Γ^σ in Fig. 4b is indeed canonical for σ. On the other hand, the tree extension Γ' from Fig. 4c can now be shown to not be canonical.

One might be interested in the canonical tree extension, while only an extension is at hand. The previous proposition enables us to perform this task in quadratic time. The algorithm (formally described in [16]) directly builds up the tree extension from its leaves to the root. Specifically, the algorithm makes sure that the extension is also an extension of the built-up tree extension, and furthermore, that the subgraphs $G[V(\Gamma_v)]$ are weakly connected.

2.4. Treewidth ≤ scanwidth ≤ level + 1

The proofs of the two bounds presented in this subsection are deferred to Appendix A, since these results are not vital to the other parts of this paper. However, the bounds do paint a nice picture of how the scanwidth fits within the landscape of other graph parameters. Note that it immediately follows from the definitions that scanwidth is bounded from above by cutwidth. It is also trivial to prove that scanwidth is bounded from below by the very similar *edge-treewidth*, as introduced in [11].

Treewidth In the introduction, the use of scanwidth as a parameter in algorithms was motivated by the successful applicability of another tree measure: *treewidth*. Although normally defined by a so-called *tree-decomposition* (see e.g. [1]), we will use a different - yet equivalent - formulation for treewidth. This allows us to relate treewidth to scanwidth in a more straightforward manner. The formulation we use is adapted from [6].



(a) Ladder graph L_n

(b) Optimal tree extension Γ_1

(c) Worst-case tree extension Γ_2

Fig. 5. (a): The *ladder-graph* L_n (with $n \geq 3$), which is a rooted binary network with level $n-1$ and $2n$ vertices. (b): An optimal tree extension Γ_1 of L_n with scanwidth 3. (c): The worst-case tree extension Γ_2 of L_n with scanwidth n.

Definition 2.6 *(Treewidth).* Let $G = (V, E)$ be a connected undirected graph. For a tree layout Γ and a vertex v of V, we will denote $\mathrm{TW}_v^\Gamma(G) = \{u \in V : u >_\Gamma v, \exists w \leq_\Gamma v \text{ s.t. } uw \in E\}$. Then the *treewidth* of G is $\mathrm{tw}(G) = \min_{\Gamma \in \mathcal{T}(V)} \max_{v \in V} |\mathrm{TW}_v^\Gamma(G)|$. Furthermore, we denote $\mathrm{tw}(\Gamma, G) = \max_{v \in V} |\mathrm{TW}_v^\Gamma(G)|$.

Recall that $\mathcal{T}(V)$ is the set of all tree layouts of G, since G is an undirected graph. To exemplify the relation between treewidth and scanwidth, consider the canonical tree extension Γ^σ in Fig. 4b, where $\mathrm{GW}_v^{\Gamma^\sigma}(G) = \{wz, qu, qv\}$. For the treewidth, we can disregard the directions, and the set $\mathrm{TW}_v^{\Gamma^\sigma}(G)$ now contains only the endpoints in $\mathrm{GW}_v^{\Gamma^\sigma}(G)$ that are higher up in the tree than v. Therefore, $\mathrm{TW}_v^{\Gamma^\sigma}(G) = \{w, q\}$.

In [2] it is mentioned without proof that the treewidth of the underlying undirected graph of a DAG lower bounds its scanwidth.[3] This fact is far from obvious when looking at the common definition of the treewidth and heavily relies on the uncommon alternative definition we have given here. As this definition is not referred to in [2], we formally state the result here and provide a proof in the appendix. The proof relies on the fact that a tree extension of a DAG is also a valid tree layout of its underlying undirected graph. As a side note, we remark that another well-known parameter, 'tree-depth', is also defined via such tree layouts [17]; hence a tree extension is also a valid tree-depth decomposition, albeit optimizing a different objective.

Lemma 2.7. *Let $G = (V, E)$ be a weakly connected DAG and $\tilde{G}$ its underlying undirected graph, then*

$$\mathrm{tw}(\tilde{G}) \leq \mathrm{sw}(G).$$

Level of a network Using that indegrees of sinksets of a network are at most the reticulation number + 1 (Lemma A.2) and that the scanwidth of a network is the maximum scanwidth of its blocks (Corollary 3.3), we can obtain the following result:

Lemma 2.8. *Let $G = (V, E)$ be a level-k network, then*

$$\mathrm{sw}(G) \leq k + 1.$$

The fact that the scanwidth of a binary network is at most its level + 1 has already been stated without proof in [2] and is proved in the appendix of [3] in a different setting. Our more graph-theoretical proof is self-contained and also holds for non-binary networks.

The above bound is certainly not tight in general. Consider for example the network in Fig. 5, which is a variation of a network from [3]. This network always has a scanwidth of 3 but can be extended to have an arbitrarily large level.

3. Reduction rules

Before discussing algorithms that compute the scanwidth and find its corresponding (tree) extension, we present some reduction rules. In Subsection 3.1 we will explain that one can split the scanwidth problem over the blocks of a network. Subsection 3.2 will cover a rule that suppresses indegree-1 outdegree-1 vertices, while Subsection 3.3 summarizes the complete reduction scheme and provides a bound on the size of a reduced instance.

[3] It was actually stated the other way round (and also mistakenly stated that cutwidth bounds scanwidth from below). However, one of the authors confirmed that it was intended as expressed here (Mathias Weller, personal communication).

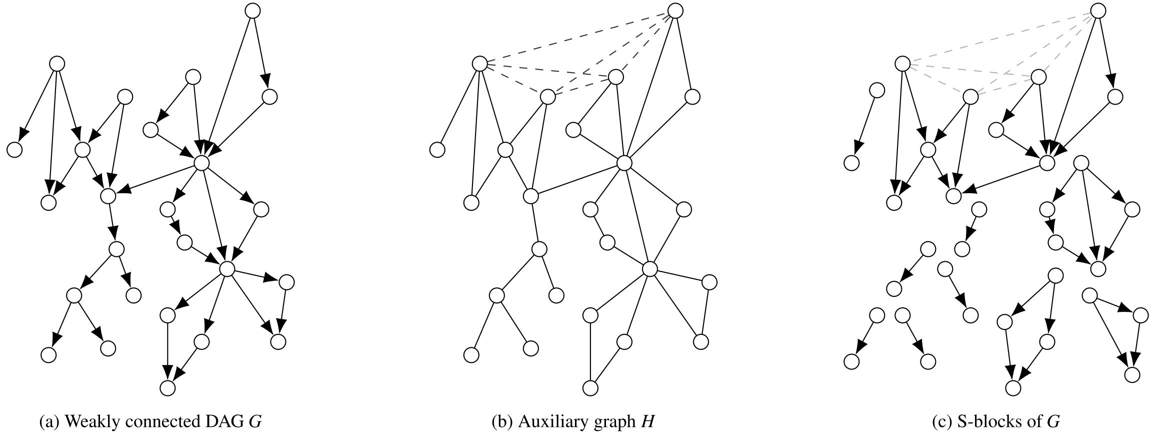


Fig. 6. (a): A multi-rooted weakly connected DAG G. (b): The auxiliary undirected graph H (as defined in Definition 3.1) where the newly added edges are dashed. (c): The s-blocks of the DAG G.

3.1. Splitting into (s-)blocks

Magne et al. [11] mention that the edge-treewidth can be split into subproblems for each block of a graph. A similar result is true for the scanwidth of a rooted DAG. The intuitive reason is that one may recursively place the vertices of a highest block at the end of an extension. One needs to be careful, however, as this is not the case for DAGs with multiple roots. Then, if multiple blocks contain a root, none of these blocks necessarily needs to be completely above the others. To remedy this problem, we introduce a non-standard generalization of a block for DAGs with multiple roots, which we call *scanwidth-blocks* or *s-blocks*, and whose definition is illustrated in Fig. 6. If a DAG has a single root, its s-blocks coincide with its blocks.

Definition 3.1 *(S-block).* Let $G=(V,E)$ be a weakly connected DAG, and let H be the underlying undirected graph of G where edges are added between all roots of G. Then for all $W\subseteq V$, the subgraph $G[W]$ is an s-block of G if $H[W]$ is a block of H.

The proof of the following theorem is rather technical, yet the result is very intuitive: we can split the graph into its s-blocks when computing the scanwidth. Note that the proof is constructive and can therefore be used to reconstruct solutions of the whole problem from its subproblems.

Theorem 3.2. *Let G be a weakly connected DAG and $\mathcal{S}(G)$ the set of s-blocks of G. Then, we have that* $\mathrm{sw}(G)=\max_{H\in\mathcal{S}(G)}\mathrm{sw}(H)$.

Proof. We will prove this by induction on the number of s-blocks k. The base case where $k=1$ follows trivially.

Let $k\geq 1$, and assume that the theorem holds for any graph with k s-blocks. Let G be a weakly connected DAG with $k+1$ s-blocks. By definition these s-blocks are exactly the blocks of the graph H from Definition 3.1. But then, there must be an s-block that contains all the roots of G (the 'root-block'), as those roots form a clique in the auxiliary graph H (see also Fig. 6c). Let $G_1=G[V_1]$ (with $V_1\subset V$) be an s-block that is not the root-block and such that G_1 only shares one vertex v with the rest of G. As all vertices in V_1 have a path from a root in G and all such paths must pass through v, we must have that v is the unique root of G_1.

Now define $G_2=G[V\setminus V_1\cup\{v\}]=(V_2,E_2)$. Then, G_1 and G_2 are subgraphs that only have v in common. Let $\sigma_1\in\Pi[V_1]$ and $\sigma_2\in\Pi[V_2]$. Since v is the root of G_1, we know that it always holds that $\sigma_1=\sigma_1'\circ(v)$, where $\sigma_1'\in\Pi[V_1\setminus\{v\}]$. Because G_1 is a pendent s-block of G, we can define $\sigma=\sigma_1'\circ\sigma_2\in\Pi[V]$. Since no arcs are directed from $V_2\setminus\{v\}$ to $V_1\setminus\{v\}$ (because both parts are only connected through v), we then have:

$$\begin{aligned}\mathrm{sw}(\sigma,G)=\max_{w\in V}|\mathrm{SW}_w^{\sigma}(G)|&=\max\left\{\max_{w\in V_1\setminus\{v\}}|\mathrm{SW}_w^{\sigma}(G)|,\max_{w\in V_2}|\mathrm{SW}_w^{\sigma}(G)|\right\}\\&=\max\left\{\max_{w\in V_1}|\mathrm{SW}_w^{\sigma_1}(G_1)|,\max_{w\in V_2}|\mathrm{SW}_w^{\sigma_2}(G_2)|\right\}\\&=\max\left\{\mathrm{sw}(\sigma_1,G_1),\mathrm{sw}(\sigma_2,G_2)\right\}.\end{aligned}\tag{1}$$

Here, we used that $|\mathrm{SW}_v^{\sigma_1}(G_1)|=0$ as v is the root of G_1.

We will now prove that $\mathrm{sw}(G)=\max\{\mathrm{sw}(G_1),\mathrm{sw}(G_2)\}$. Since the $k+1$ s-blocks of G are exactly G_1 and the k s-blocks of G_2, the induction hypothesis will then prove the theorem.

($\leq$) Let $\sigma_1 = \sigma_1' \circ (v) \in \Pi[V_1]$ and $\sigma_2 \in \Pi[V_2]$, such that $\mathrm{sw}(G_1) = \mathrm{sw}(\sigma_1, G_1)$ and $\mathrm{sw}(G_2) = \mathrm{sw}(\sigma_2, G_2)$. Let $\sigma = \sigma_1' \circ \sigma_2 \in \Pi[V]$. Then, using equation (1), $\mathrm{sw}(G) \leq \max\{\mathrm{sw}(G_1), \mathrm{sw}(G_2)\}$.

($\geq$) G_1 and G_2 are both weakly connected subgraphs of G. Thus their respective scanwidth values both form a lower bound on the scanwidth of G, providing us with $\mathrm{sw}(G) \geq \max\{\mathrm{sw}(G_1), \mathrm{sw}(G_2)\}$. $\square$

An immediate corollary is of this result is that one can split over the blocks to compute the scanwidth if G has a single root, since the s-blocks and blocks then coincide.

Corollary 3.3. *Let G be a weakly connected rooted DAG and $\mathcal{B}(G)$ the set of blocks of G. Then, we have that* $\mathrm{sw}(G) = \max_{H \in \mathcal{B}(G)} \mathrm{sw}(H)$.

3.2. Suppressing indegree-1 outdegree-1 vertices

We now introduce another safe reduction rule that allows us to simplify the graph by removing certain arcs and vertices. It formalizes the idea that if a vertex has only one incoming arc and one outgoing arc, the two arcs can be replaced by a single arc and the vertex can be deleted. In other words, we suppress vertices of indegree-1 and outdegree-1.

Lemma 3.4. *Let $G = (V, E)$ be a weakly connected DAG, and let $v \in V$ be a vertex with a single parent u and a single child w such that $uw \notin E$. Define a new DAG $H = (V \setminus \{v\}, E \setminus \{uv, vw\} \cup \{uw\})$. Then,* $\mathrm{sw}(G) = \mathrm{sw}(H)$.

Proof. We first present the following obvious claim without proof (see [16] for its straightforward proof).

Claim: Let σ be an extension of G, and let $\pi = \sigma[V \setminus \{v\}]$ be an extension of H. Then for all $a, b \in V \setminus \{v\}$, we have that $a \overset{G[\sigma[1\ldots b]]}{\leftrightsquigarrow} b$ if and only if $a \overset{H[\pi[1\ldots b]]}{\leftrightsquigarrow} b$. Furthermore, $a >_G b$ if and only if $a >_H b$.

($\mathrm{sw}(G) \geq \mathrm{sw}(H)$): Let σ be an optimal extension for G, and define the extension $\pi = \sigma[V \setminus \{v\}]$ for H. Now let $z \in V \setminus \{v\}$ be arbitrary, and let $xy \in \mathrm{SW}_z^{\pi}(H)$. If $xy \neq uw$, then $xy \in \mathrm{SW}_z^{\sigma}(G)$ by both parts of the claim. If $xy = uw$, then clearly either uv or vw is in $\mathrm{SW}_z^{\sigma}(G)$. Thus, we get that $|\mathrm{SW}_z^{\pi}(H)| \leq |\mathrm{SW}_z^{\sigma}(G)|$. Therefore,

$$\mathrm{sw}(G) = \mathrm{sw}(\sigma, G) = \max_{z \in V} |\mathrm{SW}_z^{\sigma}(G)| \geq \max_{z \in V \setminus \{v\}} |\mathrm{SW}_z^{\pi}(H)| = \mathrm{sw}(\pi, H) \geq \mathrm{sw}(H).$$

($\mathrm{sw}(G) \leq \mathrm{sw}(H)$): Let π be an optimal extension for H with $w = \pi(i)$. We also have that $\sigma = \pi[1 \ldots i] \circ (v) \circ \pi[i+1 \ldots]$ is an extension of G. Note that then $\sigma[V \setminus \{v\}] = \pi$, as in the claim. Now let $z \in V \setminus \{v\}$ be arbitrary, and let $xy \in \mathrm{SW}_z^{\sigma}(G)$. If $xy \neq uv$ and $xy \neq vw$, then $xy \in \mathrm{SW}_z^{\pi}(H)$ by both parts of the claim and the definition of the sets SW. In the other case, xy is either uv or vw (note that they can not both be in the set $\mathrm{SW}_z^{\sigma}(G)$). But then, $uw \in \mathrm{SW}_z^{\pi}(H)$. Overall, we get that $|\mathrm{SW}_z^{\sigma}(H)| \leq |\mathrm{SW}_z^{\pi}(H)|$. By similar arguments, one finds that $|\mathrm{SW}_v^{\sigma}(G)| = |\mathrm{SW}_w^{\pi}(H)|$. Therefore,

$$\begin{aligned} \mathrm{sw}(G) \leq \mathrm{sw}(\sigma, G) &= \max_{z \in V} |\mathrm{SW}_z^{\sigma}(G)| = \max\left\{ |\mathrm{SW}_v^{\sigma}(G)|, \max_{z \in V \setminus \{v\}} |\mathrm{SW}_z^{\sigma}(G)| \right\} \\ &\leq \max\left\{ |\mathrm{SW}_w^{\pi}(H)|, \max_{z \in V \setminus \{v\}} |\mathrm{SW}_z^{\pi}(H)| \right\} = \max_{z \in V \setminus \{v\}} |\mathrm{SW}_z^{\pi}(H)| = \mathrm{sw}(\pi, H) = \mathrm{sw}(H). \quad \square \end{aligned}$$

The reason we impose the restriction that $uw \notin E$ in the above lemma is that this would create a multigraph. Of course, one can generalize scanwidth to multigraphs and remove the restriction. Similarly, if we were to generalize scanwidth to weighted graphs, the restriction can be removed by increasing the weight of the already existing arc.

3.3. Complete decomposition scheme

In combination with any exact algorithm, the reduction rules from the previous two subsections can be used to compute the scanwidth. These rules are gathered in Algorithm 1.

To summarize, the algorithm first decomposes a DAG into its s-blocks. It then checks for each s-block whether it is a single arc or a rooted cycle. If this is the case, the scanwidth of that s-block is already known. For the remaining s-blocks, we use the vertex suppression rule to decrease their size. Then, we only need to exactly calculate the scanwidth of those s-blocks with some exact algorithm. Fig. 7 provides an illustration of the decomposition on an example graph.

In the next lemma, we formally prove the algorithm's correctness and its time complexity. We note that the proofs of Theorem 3.2 and Lemma 3.4 can be used to make the algorithm constructive if `ExactSW` also returns an optimal extension.

Lemma 3.5. *Let $G = (V, E)$ be a weakly connected DAG of n vertices and m arcs, and let `ExactSW` be an algorithm that finds the scanwidth of any weakly connected DAG $H' = (V', E')$ in $O(f(|V'|, |E'|))$ time for some function f. Then, Algorithm 1 returns the scanwidth of G in $O(n^2 + n \cdot f(n', m'))$ time. Here, $n' \leq n$ (resp. $m' \leq m$) is the maximum number of vertices (resp. arcs) of any of the graphs H' that appear in Algorithm 1.*

Algorithm 1: Decomposition algorithm

Input: Weakly connected DAG $G = (V, E)$, an exact algorithm `ExactSW` that computes the scanwidth of a weakly connected DAG.
Output: Scanwidth sw of G.

1 **initialize** sw $\leftarrow 0$
2 $G' \leftarrow$ underlying undirected graph of G, with edges added between all roots of G
3 $\mathcal{S} \leftarrow \{G[W] : W \subseteq V, G'[W]$ is a block of $G'\}$; // Set of s-blocks of G
4 **for each** $H \in \mathcal{S}$ **do**
5   **if** *H is a single arc* **then**
6   sw $\leftarrow \max\{\text{sw}, 1\}$
7   **else if** *H has a unique root and its underlying undirected graph is a cycle* **then**
8   sw $\leftarrow \max\{\text{sw}, 2\}$
9   **else**
10     $H' \leftarrow H$ after exhaustively suppressing vertices using Lemma 3.4
11   sw $\leftarrow \max\{\text{sw}, \texttt{ExactSW}(H')\}$
12 **return** sw

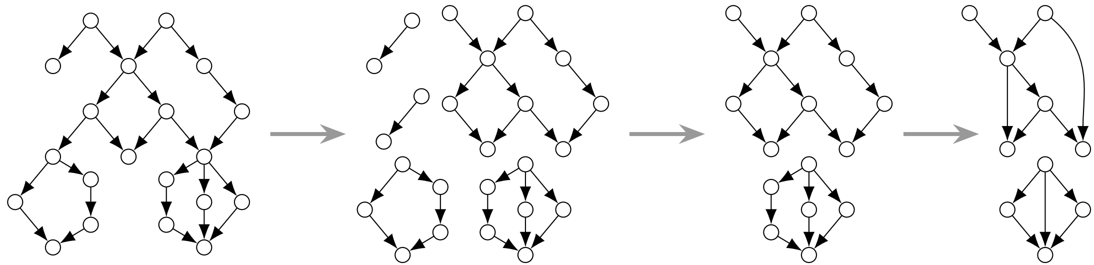


Fig. 7. Illustration of the decomposition algorithm. The first graph shows the original graph. Then, the graph is split into its s-blocks. Thereafter, the single arcs and the cycle with a unique root are 'deleted', since their scanwidths are known. Note that we then also know that the scanwidth of the complete graph is at least 2. Lastly, we exhaustively suppress indegree-1 outdegree-1 vertices in the remaining s-blocks, until it will create a multigraph.

Proof. *Correctness:* By Definition 3.1, $\mathcal{S}$ will be the set of s-blocks of G. From Theorem 3.2 it follows that we can indeed maximize over the scanwidth of the s-blocks of G. Since a single arc has just one extension of scanwidth 1, the first if statement is correct. Any extension of a cycle with a unique root has a scanwidth of 2, showing correctness of the second if statement. Correctness of the else statement now follows from Lemma 3.4, and the assumption that `ExactSW` correctly returns the scanwidth of H'.

Time complexity: The time complexity of the creation of G' is bounded by $O(n^2)$. We can find the blocks of G' in $O(|V(G')| + |E(G')|) = O(n^2)$ time [18]. We then loop over at most n s-blocks. Each of those s-blocks H contains at most n vertices. Thus, it takes $O(n)$ time to check whether H is a single arc, or if a cycle with a unique root. Exhaustively suppressing vertices also takes $O(n)$ time. Using the algorithm `ExactSW` on H' then takes $f(n', m')$ time, where n' and m' are bounds on the size of H', as defined in the lemma. Summarizing, the complexity becomes $O(n^2 + n \cdot (n + f(n', m'))) = O(n^2 + n \cdot f(n', m'))$. □

Due to this lemma, it is valuable to search for an upper bound on the size of the graphs H' in the algorithm. This could help to bound the running time of an exact algorithm.

It turns out that if G is a network, we can bound the size of the graphs H' by a linear function of its level. A crucial observation here is that the graphs H' are very similar to the so-called *simple level-k generators*: building blocks of binary networks introduced in [19,20]. In [19, Lem 4.2] it is shown that the number of vertices of a level-k generator is at most $3k - 1$, while the number of arcs is at most $4k - 2$. Using techniques from this proof, we create similar bounds for our graphs H', although we need some adjustments of the proof to allow for non-binary networks.

Lemma 3.6. *Let $G = (V, E)$ be a level-k network with $k \geq 1$, and let H be a block of G. Let H' be the graph obtained from H after exhaustively applying the vertex suppression reduction rule from Lemma 3.4. Then, $|V(H')| \leq 4k - 1$ and $|E(H')| \leq 5k - 2$.*

Proof. We can assume that H' is not a single arc, as the lemma then follows trivially. With a careful analysis one can prove the following claim on the types of vertices that can occur in H' (see [16] for the formal proof). Note that H' does not include the leaves of the network or their incident edges, since these form single-arc blocks and they have already been disregarded.

Claim: Every vertex $v \in V(H')$ is of one of the following types: (i) unique *root* with $\delta^{\text{in}}(v) = 0$ and $\delta^{\text{out}}(v) \geq 2$; (ii) *flow vertex* with $\delta^{\text{in}}(v) = \delta^{\text{out}}(v) = 1$; (iii) *tree-vertex* with $\delta^{\text{in}}(v) = 1$ and $\delta^{\text{out}}(v) \geq 2$; (iv) *reticulation vertex* with $\delta^{\text{in}}(v) \geq 2$ and $\delta^{\text{out}}(v) \leq 1$.

Now suppose that the unique root of H' has outdegree $\alpha \geq 2$. Furthermore, suppose that H' contains f flow vertices, t tree-vertices with average outdegree $\beta \geq 2$, and r reticulations with average indegree $\gamma_1 \geq 2$ and average outdegree $\gamma_2 \leq 1$. By the claim, this covers all possible vertices in H'.

The sum of indegrees of all vertices in H' is now $f + t + \gamma_1 r$, while the sum of outdegrees is $\alpha + f + t\beta + \gamma_2 r$. Since these values must be equal, we get that $(\beta - 1)t = (\gamma_1 - \gamma_2)r - \alpha$, and thus $t \leq \gamma_1 r - \alpha$. Every flow vertex in H' must be the parent of a reticulation, otherwise we would be able to suppress the flow vertex. Furthermore, each reticulation v can have at most $\delta^{\text{in}}(v) - 1$ flow vertices as its parents. Otherwise, if all its parents were flow vertices, we would be able to suppress at least one of them. Thus, we also have that $f \leq (\gamma_1 - 1) \cdot r$. Using these two inequalities, we now get for the total number of vertices:

$$|V(H')| = 1 + f + t + r \leq 1 + (\gamma_1 - 1) \cdot r + \gamma_1 r - \alpha + r.$$

As the total number of arcs equals the sum of indegrees, we also have:

$$|E(H')| = f + t + \gamma_1 r \leq (\gamma_1 - 1) \cdot r + \gamma_1 r - \alpha + \gamma_1 r.$$

Combining these inequalities with $\gamma_1 \geq 2$ and $\alpha \geq 2$, we obtain $|V(H')| \leq 4(\gamma_1 - 1)r - 1$ and $|E(H')| \leq 5(\gamma_1 - 1)r - 2$. Now note that $(\gamma_1 - 1) \cdot r = \sum_{v \in V(H'): \delta^{\text{in}}(v) \geq 2} (\delta^{\text{in}}(v) - 1)$. Because G is level-k, the graph H must have a reticulation number of at most k. Suppressing some indegree-1 outdegree-1 vertices in H will not increase this number. By the definition of the reticulation number from Section 2, we thus get that $\sum_{v \in V(H'): \delta^{\text{in}}(v) \geq 2} (\delta^{\text{in}}(v) - 1) \leq k$. This gives us $(\gamma_1 - 1)r \leq k$. Filling this in into the two upper bounds then proves the lemma. □

4. Exact methods

In this section, we focus on methods that compute the scanwidth exactly. From an optimization point of view, we aim to solve the following problem:

SCANWIDTH
Instance: Weakly connected DAG G.
Objective: Find an extension σ of G with minimum scanwidth.

Note that we could also define the problem to search for an optimal tree extension (see Definition 2.3), but in this section we only consider solution methods that construct optimal extensions. As discussed in Subsection 2.3 it is possible to create an optimal tree extension from an optimal extension in quadratic time.

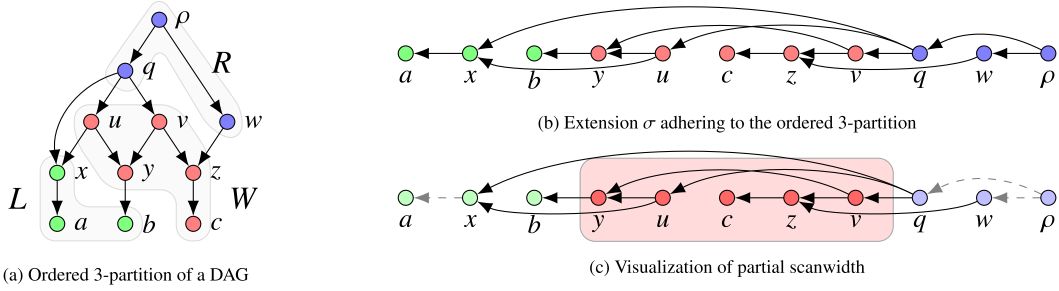
As a benchmark, we will first shortly mention an exhaustive search method that solves SCANWIDTH. The intuitive idea behind the approach is to generate all possible permutations of the vertices in a DAG and check each permutation to determine if it is a valid extension. If it is, we calculate the scanwidth of the extension, and finally, we select the extension with the smallest scanwidth as the optimal solution. The combinatorial explosion of the search space results in such a brute force solution taking $O(n! \cdot n \cdot m)$ time on a weakly connected DAG of n vertices and m arcs.

4.1. Recursive algorithm

In this subsection, we develop a recursive algorithm that solves SCANWIDTH. The main idea is to recursively keep splitting the graph into two (almost) equal-sized parts. The method we will develop is based on an algorithm from Bodlaender et al. [15]. They present a solution method for a range of ordering problems on undirected graphs, using a technique by Gurevich and Shelah [21] for the TRAVELLING SALESMAN PROBLEM. We extend this approach to DAGs and specifically tailor it towards scanwidth. To adapt the approach to DAGs, we introduce the following concept.

Definition 4.1 *(Ordered partition).* Let $G = (V, E)$ be a weakly connected DAG and $\mathcal{P} = (A_1, \ldots, A_r)$ an r-partition of V, with $r \geq 2$ and the sets A_i allowed to be empty. We call $\mathcal{P}$ an *ordered r-partition* of V, if for all $1 \leq i < r$, no arcs in E are directed from $\bigcup_{j \leq i} A_j$ towards $\bigcup_{j > i} A_j$.

Intuitively, an ordered partition respects the DAG's direction: for any vertex $v \in A_j$, all outgoing edges from v go only to sets A_i with $i \leq j$. As a consequence, it is possible to concatenate extensions of the induced subgraphs of the ordered partition, resulting in an extension of the entire graph. See Figs. 8a and 8b for an illustration of this concept, with $A_1 = L$, $A_2 = W$, and $A_3 = R$, where we use this relabeling for later reference. Note that arcs are allowed to 'skip' over parts of the partition and that the set L is forced to be a sinkset.

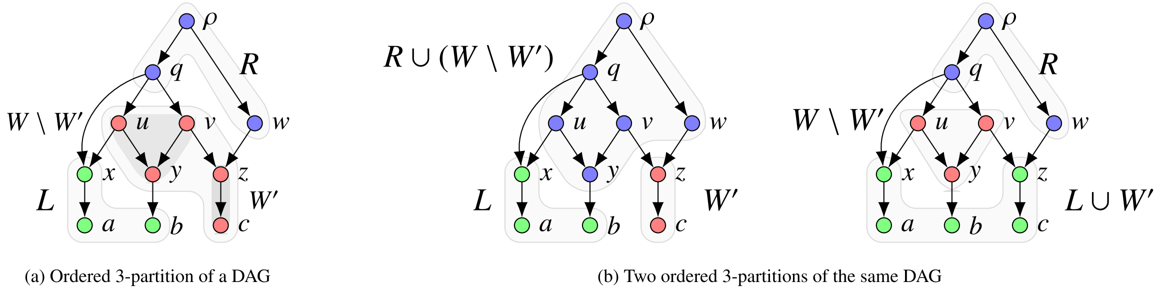


(a) Ordered 3-partition of a DAG

(b) Extension σ adhering to the ordered 3-partition

(c) Visualization of partial scanwidth

Fig. 8. (a): A weakly connected DAG $G = (V, E)$, with the colors indicating an ordered 3-partition $\mathcal{P} = (L, W, R)$ of V. (b): An extension σ of G which is a concatenation of extensions of the three subgraphs induced by the partition $\mathcal{P}$. (c): The same extension σ but with a 'window' drawn around the vertices in W, aiding the interpretation of partial scanwidth. The arcs between a pair of vertices within L or a pair vertices within R are grey and dashed because they never count towards the partial scanwidth for this ordered 3-partition.



(a) Ordered 3-partition of a DAG

(b) Two ordered 3-partitions of the same DAG

Fig. 9. Example illustrating an iteration of Algorithm 2. (a): The weakly connected DAG $G = (V, E)$ and the ordered 3-partition $\mathcal{P} = (L, W, R)$ of V from Fig. 8a. The set W is split further into W' and $W \setminus W'$, with $|W'| = \lfloor |W|/2 \rfloor$ and no arc directed from W' to $W \setminus W'$. (b): The two ordered 3-partitions $(L, W', R \cup (W \setminus W'))$ and $(L \cup W', W \setminus W', R)$ used in both Lemma 4.3(b) and Algorithm 2.

Our recursive approach uses a natural generalization of scanwidth: *partial scanwidth*. This concept allows us to analyze the scanwidth of only a subset of the vertices of a graph. This will be useful to break down the problem into smaller subproblems. By solving these subproblems recursively, we can build up the scanwidth of the entire graph. Before we state the formal definition, recall that $\Pi[W]$ is the set of all extensions of $G[W]$.

Definition 4.2 *(Partial scanwidth).* Let $G = (V, E)$ be a weakly connected DAG and $\mathcal{P} = (L, W, R)$ an ordered 3-partition of V such that $W \neq \emptyset$. For $\sigma \in \Pi[W]$ and a position i of σ, we will denote

$$\mathcal{P}\text{-PSW}_i^\sigma(G) = \{uv \in E : u \in \sigma[i+1\dots] \cup R,\ v \overset{G[\sigma[1\dots i] \cup L]}{\rightsquigarrow} \sigma(i)\}.$$

Then the *partial scanwidth* of G for $\mathcal{P}$ is

$$\mathcal{P}\text{-psw}(G) = \min_{\sigma \in \Pi[W]} \max_{i \in W} |\mathcal{P}\text{-PSW}_i^\sigma(G)|.$$

Furthermore, we let $\mathcal{P}\text{-psw}(\sigma, G) = \max_{i \in W} |\mathcal{P}\text{-PSW}_i^\sigma(G)|$ be the partial scanwidth of σ for $\mathcal{P}$. If G is clear from the context, we mostly write $\mathcal{P}\text{-PSW}_i^\sigma$ instead of $\mathcal{P}\text{-PSW}_i^\sigma(G)$.

Essentially, partial scanwidth only considers the scanwidth within a 'window' W of the vertices of G, while assuming the vertices in the set L to be positioned on the left of W in an extension, and those in the set R to be right of W. We emphasize that arcs from R to L may belong to a set $\mathcal{P}\text{-PSW}_i^\sigma(G)$, whereas arcs between pairs of vertices within R or within L are never part of these sets. Fig. 8c clarifies this concept.

We can now proceed with the main recursive idea of the algorithm. It builds upon Lemma 4 from [15], which presents a similar concept for more general functions on undirected graphs. The following lemma is in some sense an adapted version for DAGs, specifically tailored to (partial) scanwidth. See Fig. 9 for an accompanying illustration with a specific choice of W'.

Lemma 4.3. *Let $G = (V, E)$ be a weakly connected DAG and $\mathcal{P} = (L, W, R)$ an ordered 3-partition of V, with $W \neq \emptyset$.*

(a) If $|W| = 1$ (with $W = \{w\}$), then $\mathcal{P}\text{-psw}(G) = |\{uv \in E : u \in R,\ v \overset{G[L \cup W]}{\rightsquigarrow} w\}|$.

(b) If $|W| \geq 2$, then for any $1 \leq k < |W|$,

$$\mathcal{P}\text{-psw}(G) = \min_{W' \in \mathcal{W}_k} \max \left\{ \mathcal{P}_1^{W'}\text{-psw}(G), \mathcal{P}_2^{W'}\text{-psw}(G) \right\},$$

where $\mathcal{W}_k = \{W' \subseteq W : |W'| = k$ and no arc in E is directed from W' to $W \setminus W'\}$ and for all $W' \in \mathcal{W}_k$ we let $\mathcal{P}_1^{W'} = (L, W', R \cup (W \setminus W'))$ and $\mathcal{P}_2^{W'} = (L \cup W', W \setminus W', R)$.

Proof. *(a):* By Definition 4.2 and the fact that $G[W]$ only has a single extension (w), we get

$$\mathcal{P}\text{-psw}(G) = \min_{\sigma \in \Pi[W]} \max_{i \in W} |\mathcal{P}\text{-PSW}_i^{\sigma}| = |\mathcal{P}\text{-PSW}_w^{(w)}| = |\{uv \in E(G) : u \in R, v \overset{G[L \cup W]}{\leftrightsquigarrow} w\}|.$$

(b): First note that both $\mathcal{P}_1^{W'}$ and $\mathcal{P}_2^{W'}$ are also ordered 3-partitions of V with the middle set non-empty. Now, we prove an equality that is at the core of the result. Let $W' \in \mathcal{W}_k$ be arbitrary, and let $\sigma_1 \in \Pi[W']$, $\sigma_2 \in \Pi[W \setminus W']$ and $\sigma = \sigma_1 \circ \sigma_2 \in \Pi[W]$. We then have that

$$\begin{aligned} \mathcal{P}\text{-psw}(\sigma, G) &= \max\left\{\max_{w \in W'} \mathcal{P}\text{-PSW}_w^{\sigma}, \max_{w \in W \setminus W'} \mathcal{P}\text{-PSW}_w^{\sigma}\right\} \\ &= \max\left\{\max_{w \in W'} \mathcal{P}_1^{W'}\text{-PSW}_w^{\sigma_1}, \max_{w \in W \setminus W'} \mathcal{P}_2^{W'}\text{-PSW}_w^{\sigma_2}\right\} \\ &= \max\left\{\mathcal{P}_1^{W'}\text{-psw}(\sigma_1, G), \mathcal{P}_2^{W'}\text{-psw}(\sigma_2, G)\right\}. \end{aligned} \tag{2}$$

Here, the second equality is essential. It uses the important observation that if we only maximize over a consecutive subsequence of vertices in σ, we can just as well put the vertices to the left of this subsequence in the set L and the ones to the right in the set R. We are now ready to prove the lemma.

($\leq$) Let $W' \in \mathcal{W}_k$ be arbitrary. Furthermore, let $\sigma_1 \in \Pi[W']$ be such that $\mathcal{P}_1^{W'}\text{-psw}(\sigma_1, G) = \mathcal{P}_1^{W'}\text{-psw}(G)$ and $\sigma_2 \in \Pi[W \setminus W']$ be such that $\mathcal{P}_2^{W'}\text{-psw}(\sigma_2, G) = \mathcal{P}_2^{W'}\text{-psw}(G)$. Both exist by definition of the partial scanwidth. We now define $\sigma = \sigma_1 \circ \sigma_2 \in \Pi[W]$. Using equation (2), we then have that $\mathcal{P}\text{-psw}(G) \leq \max\left\{\mathcal{P}_1^{W'}\text{-psw}(G), \mathcal{P}_2^{W'}\text{-psw}(G)\right\}$. Because $W' \in \mathcal{W}_k$ was arbitrary, we obtain

$$\mathcal{P}\text{-psw}(G) \leq \min_{W' \in \mathcal{W}_k} \max\left\{\mathcal{P}_1^{W'}\text{-psw}(G), \mathcal{P}_2^{W'}\text{-psw}(G)\right\}.$$

($\geq$) Let $\sigma \in \Pi[W]$ be such that $\mathcal{P}\text{-psw}(G) = \mathcal{P}\text{-psw}(\sigma, G)$, which exists by Definition 4.2. Now choose W'' to be the set of the first k vertices of σ (clearly $W'' \in \mathcal{W}_k$). We denote by σ_1 the ordering consisting of the first k vertices of σ (in the same order). Similarly, σ_2 denotes the $|W| - k$ other vertices (again keeping the order). Thus, $\sigma = \sigma_1 \circ \sigma_2$, with $\sigma_1 \in \Pi[W'']$ and $\sigma_2 \in \Pi[W \setminus W'']$. Then, again using equation (2), $\mathcal{P}\text{-psw}(G) \geq \max\{\mathcal{P}_1^{W''}\text{-psw}(G), \mathcal{P}_2^{W''}\text{-psw}(G)\}$. As W'' was an element of $\mathcal{W}_k$, we can minimize over all $W' \in \mathcal{W}_k$ to obtain

$$\mathcal{P}\text{-psw}(G) \geq \min_{W' \in \mathcal{W}_k} \max\left\{\mathcal{P}_1^{W'}\text{-psw}(G), \mathcal{P}_2^{W'}\text{-psw}(G)\right\},$$

which proves the lemma. □

With this at first glance quite complicated recursive relation, we can formulate the relatively concise and elegant Algorithm 2 that solves Scanwidth, by setting k equal to half the size of the set W. In this way, we will be able to bound the number of 3-partitions that are considered. See again Fig. 9 for an example of the recursion of the algorithm for a specific choice of W' in the for-loop. Note that in a practical implementation of the algorithm (and also in the subsequent algorithm), we can replace the value ∞ with $|E| + 1$: the trivial upper bound of the scanwidth.

It turns out that Algorithm 2 runs in $\tilde{O}(4^n)$ time, which is a major improvement over the earlier discussed brute force solution running in $\tilde{O}(n!)$ time. In the following theorem, we prove correctness of the algorithm and its time complexity.

Theorem 4.4. *Let $G = (V, E)$ be a weakly connected DAG of n vertices and m arcs, then Algorithm 2 solves* Scanwidth *in $\tilde{O}(4^n)$ time and polynomial space.*

Proof. *Correctness:* The correctness of the partial scanwidth that is returned by the subroutine (and consequently the correctness of the scanwidth of G) follows directly from Lemma 4.3, while the correctness of the corresponding extension is a direct consequence of the constructive nature of the proof of that lemma. The algorithm terminates, as each recursive call is made for a strictly smaller set W.

Time complexity: The following analysis mimics the analysis of the recursive algorithm in [15]. Let $T(m, k)$ denote the time it takes to run the subroutine `PartialScanwidth` for a set W with $|W| = k$, and with m the number of arcs of G. If $k \geq 2$, then we loop over at most all subsets of size $\lfloor k/2 \rfloor$ of the set W. There are $\binom{k}{\lfloor k/2 \rfloor}$ such subsets. For each of these subsets, we have two recursive calls: one for a set of size $\lfloor k/2 \rfloor$ and one for a set of size $k - \lfloor k/2 \rfloor = \lceil k/2 \rceil$. Outside of the for-loop, we do some work in $O(m)$ time. Furthermore, each iteration of the for-loop can also be performed in $O(m)$ time

Algorithm 2: Recursive algorithm to solve Scanwidth.

```
Input: Weakly connected DAG G = (V, E).
Output: Scanwidth sw of G, optimal extension σ_opt.
1  sw, σ_opt ← PartialScanwidth(∅, V, ∅)
2  return sw, σ_opt
procedure PartialScanwidth(L, W, R)
1   initialize psw ← ∞; σ ← ()
2   if |W| = 1 with W = {w} then
3       psw ← |{uv ∈ E : u ∈ R, v ⇝_{G[L∪W]} w}|
4       σ ← (v)
5   else if |W| > 1 then
6       for each W' ⊆ W : |W'| = ⌊|W|/2⌋ and no arc in E is directed from W' to W \ W' do
7           psw'_1, σ'_1 ← PartialScanwidth(L, W', R ∪ (W \ W'))
8           psw'_2, σ'_2 ← PartialScanwidth(L ∪ W', W \ W', R)
9           psw' ← max{psw'_1, psw'_2}
10          if psw' < psw then
11              psw ← psw'
12              σ ← σ'_1 ∘ σ'_2
13  return psw, σ
```

per recursive call. Overall, there exists some constant $c \geq 0$, such that all these computations are bounded by $c \cdot m$. Thus, we obtain the following recurrence relation:

$$\begin{cases} T(m,1) \leq c \cdot m, & \text{if } k = 1; \\ T(m,k) \leq \binom{k}{\lfloor k/2 \rfloor}\big(T(m, \lfloor k/2 \rfloor) + T(m, \lceil k/2 \rceil)\big) + c \cdot m, & \text{if } k \geq 2. \end{cases}$$

As in [15], it then follows that $T(m,k) = \tilde{O}(4^k)$. Since the algorithm runs in $T(m,n)$ time, we obtain the time complexity of $\tilde{O}(4^n)$.

Space complexity: The recursion depth of the algorithm is $O(\log n)$, due to the sets W being split in half. Furthermore, within each recursive step, only polynomial space is used. Therefore, the complete algorithm uses polynomial space. (See also [22], where the same explanation is given for a specific case of the algorithm from [15], applied to treewidth.) □

4.2. Dynamic programming

In this subsection, we employ a dynamic programming approach to efficiently solve Scanwidth in polynomial time when the value of the scanwidth is bounded by a constant. More formally, we try to solve the following fixed-parameter problem:

k-Scanwidth	
Instance:	Weakly connected DAG G.
Objective:	Find an extension σ of G with a scanwidth of at most k, if it exists. Otherwise, certify that the scanwidth of G is larger than k.

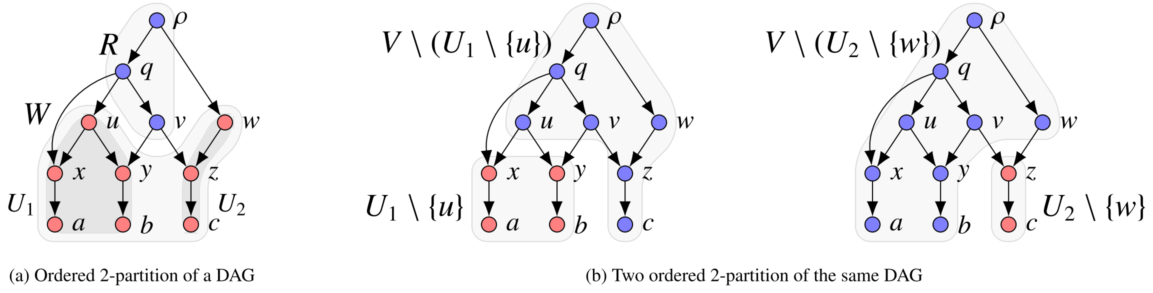
Part of the algorithm we will present shows some similarities with a dynamic programming algorithm from [15], whose authors adapt a classical technique by Held and Karp [23] for the Travelling Salesman Problem to address general ordering problems on undirected graphs. Particularly, both our and their algorithm consider subsets (or equivalently, 2-partitions) of the vertex set of a graph at most once, and recursively decrease the size of the considered sets by one. We introduce some new machinery specific to the scanwidth to further bound the time complexity, and provide an interpretation of the time complexity in terms of roots of subgraphs of the graph.

Our new algorithm neatly fits within the framework of ordered partitions and the partial scanwidth from before. Specifically, we consider ordered 2-partitions instead of ordered 3-partitions. This leads to a restricted case of the partial scanwidth where L (the set of vertices to the left of the set W) is empty.

Definition 4.5 *(Restricted partial scanwidth).* Let G be a weakly connected DAG and $\mathcal{Q} = (W, R)$ an ordered 2-partition of V such that $W \neq \emptyset$. The *restricted partial scanwidth* of G for $\mathcal{Q}$ is

$$\mathcal{Q}\text{-rpsw}(G) = \mathcal{P}\text{-psw}(G),$$

where $\mathcal{P} = (\emptyset, W, R)$. Similarly, we define $\mathcal{Q}\text{-rpsw}(\sigma, G) = \mathcal{P}\text{-psw}(\sigma, G)$, and $\mathcal{Q}\text{-RPSW}_i^{\sigma}(G) = \mathcal{P}\text{-PSW}_i^{\sigma}(G)$.



(a) Ordered 2-partition of a DAG

(b) Two ordered 2-partition of the same DAG

Fig. 10. Example illustrating an iteration of Algorithm 3. (a): The weakly connected DAG $G = (V, E)$ from Fig. 8a and an ordered 2-partition $\mathcal{Q} = (W, R)$ of V. The set W is split further into U_1 and U_2, with $G[U_1]$ and $G[U_2]$ forming the two connected components of $G[W]$. Both subgraphs have a single root: u is the root of $G[U_1]$ and w is the root of $G[U_2]$. (b): The two ordered 2-partitions $(U_1 \setminus \{u\}, V \setminus (U_1 \setminus \{u\}))$ and $(U_2 \setminus \{w\}, V \setminus (U_2 \setminus \{w\}))$ that appear when combining Lemma 4.6(b) and Lemma 4.7, which form the basis of Algorithm 3.

Note that if (W, R) is an ordered 2-partition, then $(\emptyset, W, R)$ is indeed an ordered 3-partition of a graph. Hence, in light of Definition 4.2, the notions from Definition 4.5 are well-defined. Furthermore, we emphasize that a partition (W, R) being an ordered 2-partition means nothing more than that W is a sinkset of the graph, since arcs are not allowed to be oriented from W to R (see Fig. 10a).

Similar to the partial scanwidth, the restricted partial scanwidth focuses only at a 'window' of an extension. One only considers the vertices in a sinkset W while considering the other vertices in the set R to be right of W, disregarding the exact position these other vertices may have.

A useful by-product of the connection to partial scanwidth is that the main recursion of the dynamic programming algorithm can be seen as a special case of Lemma 4.3. Another key recursion applies when the subgraph $G[W]$ is not weakly connected, allowing the problem to be split across the weakly connected components of $G[W]$. These ideas are formalized in Lemma 4.6 and Lemma 4.7, respectively. See Fig. 10 for an illustration that combines the two recursive steps.

Lemma 4.6. *Let $G = (V, E)$ be a weakly connected DAG and $\mathcal{Q} = (W, R)$ an ordered 2-partition of V with $W \neq \emptyset$.*

(a) If $|W| = 1$ (with $W = \{w\}$), then $\mathcal{Q}$-rpsw$(G) = \delta^{\text{in}}(w)$.
(b) If $|W| \geq 2$ and $G[W]$ is weakly connected, then

$$\mathcal{Q}\text{-rpsw}(G) = \min_{\rho \in P(G[W])} \max \Big\{ (W \setminus \{\rho\}, R \cup \{\rho\})\text{-rpsw}(G), \delta^{\text{in}}(W) \Big\},$$

where $P(G[W])$ is the set of roots of $G[W]$.

Proof. *(a):* First note that $R = V \setminus \{w\}$ in this case. Then from Lemma 4.3a and Definition 4.5 we obtain $\mathcal{Q}$-rpsw$(G) = |\{uv \in E : u \in V \setminus \{w\}, v \overset{G[\emptyset \cup \{w\}]}{\leftrightsquigarrow} w\}| = \delta^{\text{in}}(w)$.

(b): Recall that in Lemma 4.3b, we defined the collection of sets $\mathcal{W}_k = \{W' \subseteq W : |W'| = k$ and no arc in E is directed from W' to $W \setminus W'\}$ for ordered 3-partitions (L, W, R) and some $k \in \{1, \ldots, |W| - 1\}$. We now use this lemma with $k = |W| - 1$, to get

$$\begin{aligned}
\mathcal{Q}\text{-rpsw}(G) &= \min_{W' \in \mathcal{W}_{|W|-1}} \max \Big\{ (\emptyset, W', R \cup (W \setminus W'))\text{-psw}(G), (\emptyset \cup W', W \setminus W', R)\text{-psw}(G) \Big\} \\
&= \min_{W' = W \setminus \{\rho\} : \rho \in P(G[W])} \max \Big\{ (W', R \cup \{\rho\})\text{-rpsw}(G), (W', \{\rho\}, R)\text{-psw}(G) \Big\} \\
&= \min_{\rho \in P(G[W])} \max \Big\{ (W \setminus \{\rho\}, R \cup \{\rho\})\text{-rpsw}(G), (W \setminus \{\rho\}, \{\rho\}, R)\text{-psw}(G) \Big\}.
\end{aligned}$$

The crucial observation in the above equality is that $\mathcal{W}_{|W|-1}$ contains precisely the subsets of W obtained by removing one vertex that is a root of $G[W]$.

Now using Lemma 4.3a, we immediately have that $(W \setminus \{\rho\}, \{\rho\}, R)$-psw$(G) = |\{uv \in E(G) : u \in R, v \overset{G[W]}{\leftrightsquigarrow} \rho\}|$. This last expression equals $\delta^{\text{in}}(W)$, since $G[W]$ is weakly connected. □

Lemma 4.7. *Let $G = (V, E)$ be a weakly connected DAG and $\mathcal{Q} = (W, R)$ an ordered 2-partition of V with $W \neq \emptyset$. Then*

$$\mathcal{Q}\text{-rpsw}(G) = \max_{U_i \lhd W} \Big\{ (U_i, V \setminus U_i)\text{-rpsw}(G) \Big\},$$

where $U_i \lhd W$ indicates that $G[U_i]$ is a weakly connected component of $G[W]$.

Proof. Because W must be a sinkset, each $U_i \lhd W$ must also be a sinkset. Therefore, $(U_i, V \setminus U_i)$ is indeed an ordered 2-partition of V, for each $U_i \lhd W$.

First, we prove a critical equality. Let r be the number of weakly connected components of $G[W]$. For each $i \in \{1, \dots, r\}$, let $\sigma_i \in \Pi[U_i]$ be arbitrary, and define $\sigma_1 \circ \dots \circ \sigma_r = \sigma \in \Pi[W]$. This is indeed an extension because the different σ_i are not weakly connected in $G[W]$. Using Definition 4.5 and denoting $|\mathcal{Q}\text{-RPSW}_w^\sigma(G)|$ by $\text{rpsw}_w^\sigma(\mathcal{Q})$ for ease of notation (and similar for PSW), we have:

$$\begin{aligned}\mathcal{Q}\text{-rpsw}(\sigma, G) = \max_{w \in W} \text{rpsw}_w^\sigma(\mathcal{Q}) &= \max_{U_i \lhd W} \left\{ \max_{w \in U_i} \text{psw}_w^\sigma((\emptyset, W, R)) \right\} \\ &= \max_{U_i \lhd W} \left\{ \max_{w \in U_i} \text{psw}_w^{\sigma_i} \left(\left(\bigcup\nolimits_{1 \le j < i} U_j, U_i, R \cup \bigcup\nolimits_{j > i} U_j \right) \right) \right\} \\ &= \max_{U_i \lhd W} \left\{ \max_{w \in U_i} \text{psw}_w^{\sigma_i} \left(\left(\emptyset, U_i, R \cup \bigcup\nolimits_{j \neq i} U_j \right) \right) \right\} \\ &= \max_{U_i \lhd W} \left\{ (U_i, V \setminus U_i)\text{-rpsw}(\sigma_i, G) \right\}. \end{aligned} \tag{3}$$

The third equality is similar to the third equality of equation (2) that appears in the proof of Lemma 4.3b. It uses the observation that if we only maximize over vertices that form a consecutive subsequence of the extension σ, we can just as well put the vertices to the left of this subsequence in the set L, and the ones to the right in the set R. For the fourth equality we then use that in $G[W]$, the vertices of $\bigcup_{1 \le j < i} U_j$ are not weakly connected to those in U_i. Thus, we can put them in the R-set, without changing the restricted partial scanwidth. We are now ready to prove the lemma.

($\le$) For all $U_i \lhd W$, we let $\sigma_i \in \Pi[U_i]$ be an optimal extension. In other words, $(U_i, V \setminus U_i)\text{-rpsw}(G) = (U_i, V \setminus U_i)\text{-rpsw}(\sigma_i, G)$. Now let $\sigma = \sigma_1 \circ \dots \circ \sigma_r \in \Pi[W]$. Using equation (3), we get that $\mathcal{Q}\text{-rpsw}(G) \le \max_{U_i \lhd W} \{(U_i, V \setminus U_i)\text{-rpsw}(G)\}$.

($\ge$) We first present a claim. As the claim is quite intuitive, we refer to [16] for the rather technical proof.

Claim: Let $\sigma \in \Pi[W]$ be such that for some $k \in \{1, \dots, |W| - 1\}$, $\sigma(k)$ and $\sigma(k+1)$ are not weakly connected in $G[W]$. Let π be obtained from σ by swapping $\sigma(k)$ and $\sigma(k+1)$. Then, $\pi \in \Pi[W]$ and $\mathcal{Q}\text{-rpsw}(\sigma, G) = \mathcal{Q}\text{-rpsw}(\pi, G)$.

Let $\sigma \in \Pi[W]$ be such that $\mathcal{P}\text{-rpsw}(G) = \mathcal{P}\text{-rpsw}(\sigma, G)$, which exists by definition. We can also assume that $\sigma = \sigma_1 \circ \dots \circ \sigma_r$, where for each i we have $\sigma_i \in \Pi[U_i]$. Such an extension exists since we can keep swapping consecutive vertices from different U_i until this condition holds, if σ does not have this property. By the claim, this is also an extension, and it will give the same restricted partial scanwidth. From equation (3) it then quickly follows that $\mathcal{Q}\text{-rpsw}(G) \ge \max_{U_i \lhd W} \{(U_i, V \setminus U_i)\text{-rpsw}(G)\}$. □

The previous two lemmas now result in the following corollary, which forms the core of our dynamic programming algorithm. In particular, part (a) is an immediate consequence of Lemma 4.7, while parts (b) and (c) follow from Lemma 4.6.

Corollary 4.8. *Let $G = (V, E)$ be a weakly connected DAG, $k \ge 1$ an integer and $\mathcal{Q} = (W, R)$ an ordered 2-partition of V such that $W \neq \emptyset$. Then,*

(a) $\mathcal{Q}\text{-rpsw}(G) > k$ if and only if $(U_i, V \setminus U_i)\text{-rpsw}(G) > k$ for some component $G[U_i]$ of $G[W]$.
(b) For $|W| = 1$, $\mathcal{Q}\text{-rpsw}(G) > k$ if and only if $\delta^{\text{in}}(W) > k$.
(c) For $|W| \ge 2$ such that $G[W]$ is weakly connected, $\mathcal{Q}\text{-rpsw}(G) > k$ if and only if $\delta^{\text{in}}(W) > k$ or $(W \setminus \{\rho\}, R \cup \{\rho\})\text{-rpsw}(G) > k$ for all roots ρ of $G[W]$.

With this corollary in mind, we can set up a dynamic programming procedure, described in Algorithm 3. The algorithm involves the use of a 'table', denoted by T, to store previously calculated results. At the cost of more space, the recursive procedure can first check if a result is already known, thereby saving time. See Fig. 10 for an example where $G[W]$ consists of two weakly connected components, each with a single root, so the algorithm recurses on two ordered 2-partitions (assuming k is not too small, which would prevent the recursive calls).

Before Theorem 4.11 formally states correctness of the algorithm, we need two lemmas that help to further bound the number of considered sets in the algorithm and consequently its time complexity.

An *antichain* is a subset of a (partially) ordered set, in which each pair of elements is incomparable to each other. In our context, the roots of a sinkset form an antichain when considering the natural order of a DAG. This is proved in the next lemma. Recall that we write $W \sqsubseteq U$ for any $U \subseteq V$ if both $W \subseteq U$ and W is a sinkset.

Lemma 4.9. *Let $G = (V, E)$ be a weakly connected DAG. Then for all $W \sqsubseteq V$, the roots of $G[W]$ form an antichain with respect to the partial order $<_G$. Moreover, there is a one-to-one correspondence between the sets $W \sqsubseteq V$ and the antichains of the partial order $<_G$, defined by the roots of $G[W]$.*

Proof. We will start with the first statement. Assume towards a contradiction that the roots of $G[W]$ are not an antichain. If we let $P(G[W])$ be the set of roots of $G[W]$, we must then have that for some $\rho_1, \rho_2 \in P(G[W])$ it holds that $\rho_1 <_G \rho_2$.

Algorithm 3: Dynamic programming algorithm to solve k-Scanwidth.

```
Input: Weakly connected DAG G = (V, E), integer k ≥ 1.
Output: If sw(G) ≤ k: scanwidth sw of G and an optimal extension σ_opt. If sw(G) > k: ∞ and an incomplete extension.
1  T ← empty table to tabulate results, indexed by all 2-partitions of V
2  sw, σ_opt ← R-PartialScanwidth(V, ∅, k)
3  return sw, σ_opt
   procedure R-PartialScanwidth(W, R, k)
1    if T(W, R) exists then                        // Look up result in global table, if available
2      return T(W, R)
3    initialize rpsw ← ∞ ; σ ← ()
4    for each weakly connected component G[U_i] of G[W], (i = 1, ..., r) do
5      initialize rpsw_i ← ∞ ; σ_i ← ()
6      if |U_i| = 1 with U_i = {v} and δ^in(U_i) ≤ k then
7        rpsw_i ← δ^in(v)
8        σ_i ← (v)
9      else if |U_i| > 1 and δ^in(U_i) ≤ k then
10       for each root ρ of G[U_i] do
11         rpsw'_1, σ'_1 ← R-PartialScanwidth(U_i \ {ρ}, V \ (U_i \ {ρ}), k)
12         rpsw' ← max{rpsw'_1, δ^in(U_i)}
13         if rpsw' < rpsw_i then
14           rpsw_i ← rpsw'
15           σ_i ← σ'_1 ∘ (ρ)
16   rpsw ← max{rpsw_i : i = 1, ..., r}
17   σ ← σ_1 ∘ ··· ∘ σ_r
18   T(W, R) ← rpsw, σ ;                           // Store result in global table
19   return rpsw, σ
```

Now let $v \in V$ be such that $\rho_1 <_G v \leq_G \rho_2$ and $(v, \rho_1) \in E$. Such a vertex exists, as otherwise ρ_1 and ρ_2 would not be comparable. But as $v \leq_G \rho_2$ and W is a sinkset, v must be in W. Thus, ρ_1 can not be a root of $G[W]$: a contradiction. This proves the first statement.

Let us denote the set of all $W \sqsubseteq V$ by $\mathcal{V}$ and the set of all vertex-antichains with respect to $<_G$ by $\mathcal{A}$. We can now define a function $f : \mathcal{V} \to \mathcal{A}$ by $f(W) = P(G[W])$ for all $W \in \mathcal{V}$. By the first statement, f indeed maps all sets W into $\mathcal{A}$. It can easily be shown that f is both surjective and injective (see [16] for the full proof), thus proving that f is a bijection from the sinksets to the vertex-antichains of G, as desired. □

With the characterization of the sinksets of a DAG by means of their roots, we can bound the number of sinksets that have bounded indegree.

Lemma 4.10. *Let $G = (V, E)$ be a weakly connected DAG of n vertices and r roots, and let $k \geq 1$ be an integer. Then, the number of sets $W \sqsubseteq V$ such that $\delta^{in}(W) \leq k$, is bounded from above by n^{k+r-1}.*

Proof. Let $k \geq 1$ be arbitrary and $\mathcal{V}_k = \{W \sqsubseteq V : \delta^{in}(W) \leq k\}$. We first prove the claim that for all $W \in \mathcal{V}_k$, $G[W]$ has at most $k + r - 1$ roots.

Proof of claim. Let $W \in \mathcal{V}_k$ be arbitrary. We now consider two cases.

Case 1: each root of G is a root of W. As W is a sinkset, this means that $W = V$, and so $G[W] = G$. Therefore, $G[W]$ has exactly r roots. Because $k \geq 1$, the bound then follows.

Case 2: there exists a root of G that is not a root of $G[W]$. Then, at most $r - 1$ of the roots of $G[W]$, are also a root of G. Therefore, $G[W]$ has at most $r - 1$ roots with indegree 0 in G. Furthermore, $G[W]$ has at most k roots with an indegree of at least 1 in G (otherwise, the indegree of W would be larger than k, and then $W \notin \mathcal{V}_k$). Together, this gives that $G[W]$ has at most $k + r - 1$ roots. △

Together with Lemma 4.9, this claim shows that for all $W \in \mathcal{V}_k$, the roots of $G[W]$ form a vertex-antichain of G, with a size of at most $k + r - 1$. Now let $\mathcal{A}_\ell$ denote the set of vertex-antichains of G of size at most ℓ. Then we can define a function $h : \mathcal{V}_k \to \mathcal{A}_{k+r-1}$ by $h(W) = P(G[W])$ for all $W \in \mathcal{V}_k$ (here, $P(G[W])$ indicates the set of roots of $G[W]$ again). But then, h is actually a restriction of the function f from the proof of Lemma 4.9, to the domain $\mathcal{V}_k$ (and with a smaller co-domain). Using that this function f was bijective, we must have that $h : \mathcal{V}_k \to \mathcal{A}_{k+r-1}$ is injective, otherwise this would contradict the injectivity of f. Therefore, we naturally have that $|\mathcal{V}_k| \leq |\mathcal{A}_{k+r-1}|$. So, it suffices to count the number of

vertex-antichains in G of size at most $k+r-1$, to get an upper bound for our lemma. We will now show by induction on ℓ that $|\mathcal{A}_\ell| \leq n^\ell$ for all integers $\ell \geq 1$, which will then prove the lemma.

The base case where $\ell = 1$ is trivial. Now assume that the induction hypothesis holds for ℓ, so $|\mathcal{A}_\ell| \leq n^\ell$. Note that all vertex-antichains of size exactly $\ell+1$ can be created from the antichains of size ℓ, by the addition of one incomparable vertex. As there are only $n-\ell$ vertices left to be added for each antichain of size ℓ, we obtain that $|\mathcal{A}_{\ell+1}| \leq n^\ell + n^\ell \cdot (n-\ell) \leq n^{\ell+1}$. □

The previous lemma allows us to bound the number of sets that are considered during the execution of the algorithm. This sets up the stage for proving the time complexity of Algorithm 3 in the following theorem.

Theorem 4.11. *Let $G = (V, E)$ be a weakly connected DAG of n vertices, m arcs and r roots, and let $k \geq 1$ be an integer. Then, Algorithm 3 solves k-*Scanwidth *in $O((k+r-1) \cdot n^{k+r-1} \cdot m)$ time and $O((k+r-1) \cdot n^{k+r})$ space.*

Proof. *Correctness:* In the algorithm we use ∞ as a placeholder value whenever the restricted partial scanwidth is larger than k, and in that case, we do not care about the corresponding extension. The correctness of the subroutine `R-PartialScanwidth` now follows immediately from Corollary 4.8, which precisely defines when ∞ should be returned. The correct extensions are returned due to the constructive nature of Lemmas 4.6 and 4.7, on which Corollary 4.8 is based.

Time complexity: Within each subroutine call, it takes $O(m)$ time to create the components $G[U_i]$ and to compute all indegrees $\delta^{\text{in}}(U_i)$. Given these indegrees, only $O(|U_i|)$ extra time is spent in the outer for-loop (without the recursive call). But since the U_i are disjoint, the complete time spend per subroutine call must then be dominated by $O(m)$.

Due to the tabulation, we execute the subroutine at most once for each of the (at most 2^n) sinksets of G. The subroutine is only executed for sinksets U_i that are created by the deletion of a root of a weakly connected sinkset W with indegree at most k. According to Lemma 4.10, there are at most n^{k+r-1} such sinksets W. As stated in the claim of the proof of Lemma 4.10 these sinksets W have at most $k+r-1$ roots, yielding $O((k+r-1) \cdot n^{k+r-1})$ recursive subroutine calls. This results in a total time complexity of $O((k+r-1) \cdot n^{k+r-1} \cdot m)$.

Space complexity: We store an extension of size $O(n)$ and a value rpsw for each considered sinkset. This requires $O((k+r-1) \cdot n^{k+r})$ space. The space needed for the graph, and to run the subroutine, is also surely bounded by this function. □

From this theorem, we can deduce a nice complexity result for DAGs with a fixed number of roots. Our algorithm then functions as an XP algorithm when considering scanwidth as the parameter.

Corollary 4.12. *Let $G = (V, E)$ be a weakly connected rooted DAG with n vertices, m arcs, r roots, and a scanwidth of k. Then, there exists an algorithm that solves* Scanwidth *in $O((k+r-1) \cdot m \cdot n^{k+r-1})$ time and $O((k+r-1) \cdot n^{k+r})$ space. Thus, for DAGs with a fixed number of roots* Scanwidth *is in* XP *when considering the scanwidth as the parameter.*

Proof. By repeatedly running Algorithm 3 we can solve i-Scanwidth for an increasing value of i. When we eventually reach the value k (which equals the scanwidth), we will find an optimal extension. If we keep the intermediate results of the previous algorithm runs in the table T, we consider the same amount of sinksets as we would have considered by directly solving k-Scanwidth. Thus, the time and space complexity is the same as in Theorem 4.11. Since we consider the number of roots r to be fixed, this directly proves the XP result stated in the corollary. □

In Subsection 3.3 we introduced a decomposition algorithm aimed at reducing the size of an instance. We proved that, in the case of networks, these reduced instances have their size bounded by a linear function of the level. If we apply this to the algorithm described in the previous corollary, we can formulate another complexity result, proving that for networks Scanwidth is FPT when considering the level as a parameter.

Corollary 4.13. *Let $G = (V, E)$ be a level-ℓ network of n vertices and m arcs. Then, there exists an algorithm that solves* Scanwidth *in $O(2^{4\ell-1} \cdot \ell \cdot n + n^2)$ time. Thus, for networks* Scanwidth *is in* FPT *when considering the level as the parameter.*

Proof. The algorithm in the previous corollary visits each sinkset at most once and spents at most $O(m)$ time on each such set. Thus it also has its time complexity bounded by $O(2^n \cdot m)$. When combined with decomposition Algorithm 1, we can then solve Scanwidth in $O(n^2 + n \cdot m' \cdot 2^{n'})$ time, according to Lemma 3.5. Here, n' (resp. m') is the maximum number of vertices (resp. arcs) of any of the subproblems created by the decomposition algorithm. G is assumed to be a network, and thus the graphs of the different subproblems created by the decomposition algorithm have at most $4\ell - 1$ vertices, and at most $5\ell - 2$ arcs, according to Lemma 3.6. Substituting these numbers for n' and m' gives the desired result. □

5. Heuristics

Motivated by the successful use of sub-optimal solution methods for other width parameters [13], we now divert our attention to heuristics. We observe that the sets SW_i^σ of an extension σ of a DAG correspond to arc-cuts of the DAG. In the

(a) Weakly connected DAG G

(b) Weighted auxiliary graph H

Fig. 11. (a): Weakly connected DAG G with unit weights on all arcs. C_1 is a non-trivial DAG-cut of weight 5; C_2 is a smallest non-trivial DAG-cut of weight 4; C_3 is a trivial DAG-cut; C_4 is a directed cut that is not a DAG-cut, because it does not cut off a sinkset; C_5 is a directed cut that is not a DAG-cut, because the cut is not minimal (in particular, C_3 is a cut with a cut-set contained in the cut-set of C_5). (b): The corresponding weighted auxiliary graph H, where all dashed arcs have weight ∞ and the black arcs have unit weights. The grey vertices are in the set U and the black vertices are in the set W. The smallest DAG-cut C_2 of G corresponds to a minimum directed s-t cut in H with the same weight, where $s \in U$ and $t \in W$.

next two subsections, we leverage this observation to develop a heuristic. Instinctively, it makes sense to search for a small arc-cut in the graph and then use that cut to split into two subproblems. Subsection 5.1 takes a closer look at these *DAG-cuts* that appear in a (tree) extension and thereafter develops the corresponding heuristic idea. We reserve Subsection 5.2 for a brief discussion on how to apply two other already existing algorithmic frameworks to scanwidth.

5.1. Repeated DAG-cut-splitting heuristic

First recall some standard terminology for cuts. A *(directed) cut* in a directed graph $G = (V, E)$ is a partition $C = (S, T)$ of V (with $|S|, |T| > 0$). The corresponding *cut-set* is the set $\{uv \in E : u \in S, v \in T\}$. A directed cut C is *minimal* if no other cut exists that has a cut-set that is contained in the cut-set of C. For two distinct vertices $s, t \in V$, an *s-t cut* is a directed cut (S, T) such that $s \in S$ and $t \in T$. The size (resp. weight) of the cut refers to the size (resp. sum of weights) of the cut-set and we denote it by $|C|$ (resp. $w(C)$, where w is the weight function of the graph).

We can now introduce a certain type of cut, for which we will show shortly that they are the exact cuts that appear in extensions. To illustrate the definition, see Fig. 11a.

Definition 5.1 *(DAG-cut).* Let $G = (V, E)$ be a weakly connected DAG, then we call a directed cut $C = (S, T)$ a *DAG-cut* if C is minimal and T is a sinkset. If $|S| = 1$ or $|T| = 1$, C is a *trivial* DAG-cut.

It might not immediately be clear that the sets SW_i^σ of an extension σ correspond to the cut-sets of DAG-cuts, and vice versa. The next lemma formally proves this, thus solidifying the idea of using DAG-cuts to split a graph.

Lemma 5.2. *Let $G = (V, E)$ be a weakly connected DAG with $n \geq 2$ vertices. Then, a set $F \subseteq E$ is the cut-set of some DAG-cut C if and only if $F = \mathrm{SW}_i^\sigma$ for some position $i < n$ of an extension σ of G.*

Proof. ($\Rightarrow$) Let F be the cut-set of some DAG-cut $C = (S, T)$. By Definition 5.1, C is now a minimal cut. This means that no other cut exists with a cut-set contained in F. This implies that $G[T]$ is a weakly connected graph. Now let $\sigma_1 \in \Pi[T]$ and $\sigma_2 \in \Pi[S]$. Since T is a sinkset, $\sigma = \sigma_1 \circ \sigma_2$ is then an extension of G. Using that $G[T]$ was weakly connected, we have that $\mathrm{SW}_{|T|}^\sigma = F$, where $|T| < n$ because $|S| > 0$.

($\Leftarrow$) Let σ be an extension of G and $i < n$ a position of σ. Now let T be the vertex set of the weakly connected component of $G[1 \ldots i]$ that contains $\sigma(i)$. We let $S = V \setminus T$. It should come as no surprise that $\mathrm{SW}_i^\sigma = \{uv \in E : u \in S, v \in T\}$. In other words, SW_i^σ is the cut-set of $C = (S, T)$, with $|S| > 0$ because $|T| \leq i < n$. Since T is weakly connected, C must be minimal. As T was a component of $G[1 \ldots i]$ and σ was an extension, it must also be a sinkset. Thus, C is a DAG-cut. □

Our main focus will be to find non-trivial DAG-cuts, specifically the smallest such cut(s). Finding these minimum-weight non-trivial cuts is not obvious, but the following lemma, illustrated in Fig. 11b, will clarify the approach. In particular, we prove that finding non-trivial DAG-cuts of finite weight is equivalent to finding certain s-t cuts in an auxiliary graph H. This is helpful, as several algorithms are known that find minimum s-t cuts. The idea to use reverse arcs of infinite weight in the auxiliary graph is inspired by Ravi, Agrawal, and Klein [24], who employed this technique for the closely related *DAG edge-separators*.

Lemma 5.3. *Let $G = (V, E, w)$ be a weighted, weakly connected DAG with weight function $w : E \to \mathbb{R}_{>0}$. Let H be the weighted directed graph obtained from G, by adding for each arc a reverse arc with infinite weight. Denote by U (resp. W) the set of children (resp. parents) of the roots (resp. leaves) of G. Then,*

(a) $C = (S, T)$ is a non-trivial DAG-cut in G with weight $k < \infty$ if and only if for some $s \in U$ and $t \in W \setminus \{s\}$, C is a minimal directed s-t cut in H with weight $k < \infty$.

(b) No non-trivial DAG-cut in G exists if and only if for all $s \in U$ and $t \in W \setminus \{s\}$ no minimal directed s-t cut in H with finite weight exists.

Proof. $(a, \Rightarrow)$ By definition, C is a minimal cut in G, T is a sinkset, and $|S|, |T| \geq 2$. We must then have that S (resp. T) contains at least one root (resp. leaf) of G and a child s (resp. parent t) of this vertex. (Note that they can not be the same.) If this were not the case, either T would not be a sinkset, or the cut would not be minimal (e.g. if T consists of just two leaves). We now have that $s \in U$ and $t \in W \setminus \{s\}$. Thus, C is clearly an s-t cut in H. Furthermore, it has exactly weight k, because the arcs going from S to T in H are exactly the arcs in the original cut-set in G. We have no infinite weight arcs going from S to T in H, since T was a sinkset in G. Lastly, C is also minimal in H, since it was minimal in G.

$(a, \Leftarrow)$ Because C has finite weight in H, it does not have any of the infinite weight arcs in its cut-set. This must mean that no infinite weight arc goes from S to T in H, or equivalently no arc in G goes from T to S. Thus T must be a sinkset in G. This also means that the parent of s in G (which by definition is a root of G) is in the set S, otherwise T would be no sinkset in G any more. Similarly, the child of t in G (which is a leaf of G) must be in the set T. Thus, we have that $|S|, |T| \geq 2$. Again, the only arcs going from S to T in H are exactly the arcs in the cut-set of C in G. So, the weights also coincide. Lastly, C is minimal in G, because it was minimal in H.

(b) This follows directly from (a), and the fact that G only has finite weight cuts. □

By the previous lemma, we can find a minimum-weight non-trivial DAG-cut by finding a minimum directed s-t cut in the auxiliary graph H for all $s \in U$ and $t \in W \setminus \{s\}$. A minimum weight directed s-t cut in a directed graph with n vertices and m arcs can be found in $O(n \cdot m)$ time with the max-flow algorithm from [25]. By repeating this algorithm for $O(n^2)$ choices of s and t, we can therefore find a minimum-weight non-trivial DAG-cut in $O(n^3 \cdot m)$ time.

The idea behind our scanwidth-heuristic is to recursively split the graph at a smallest non-trivial DAG-cut. Consequently, we obtain an upper and a lower subgraph. However, when considering scanwidth we can not just 'forget' about the arcs in the cut, as they might also be counted at vertices lower or higher in the graph. Thus, for both created graphs, we merge the other part of the graph into one 'supervertex'. This ensures the arcs in the DAG-cut are still accounted for. It also explains why we look for non-trivial DAG-cuts: otherwise, the merging operation will not decrease the size of our graph. Whenever no non-trivial DAG-cut exists, the graph is very small, and we simply take an arbitrary extension. This leads to the heuristic described in Algorithm 4.

Algorithm 4: Repeated DAG-cut-splitting heuristic to find an extension.

```
Input: Weakly connected DAG G = (V, E).
Output: Extension σ_G.
1  G' ← weighted version of G with unit weights
2  σ_G ← MinDAGCutSplit(G')
3  return σ_G
   procedure MinDAGCutSplit(H)                          // H is a weighted graph.
1      if H has no non-trivial DAG-cut then
2          σ ← arbitrary extension of H ;               // Use reverse BFS traversal.
3      else
4          C = (S, T) ← minimum-weight non-trivial DAG-cut of H
5          H_1 ← H[S]; H_2 ← H[T]
6          add a vertex x to H_1 and a vertex y to H_2
7          for each uv ∈ E(H) : u ∈ S, v ∈ T do
8              add an arc ux to H_1 (if it already exists increase the weight by 1)
9              add an arc yv to H_2 (if it already exists increase the weight by 1)
10         σ_1 ← MinDAGCutSplit(H_1)
11         σ_2 ← MinDAGCutSplit(H_2)
12         σ ← σ_2[T] ∘ σ_1[S]
13     return σ
```

In Figs. 12a and 12b the first iteration of the algorithm is visualized. Fig. 12c shows the canonical tree extension corresponding to the extension resulting from the algorithm. We indeed see the cut C reappearing. Fig. 12d shows the optimal tree extension which has a smaller scanwidth than the one in Fig. 12c. Thus, in general, it is not necessarily true that

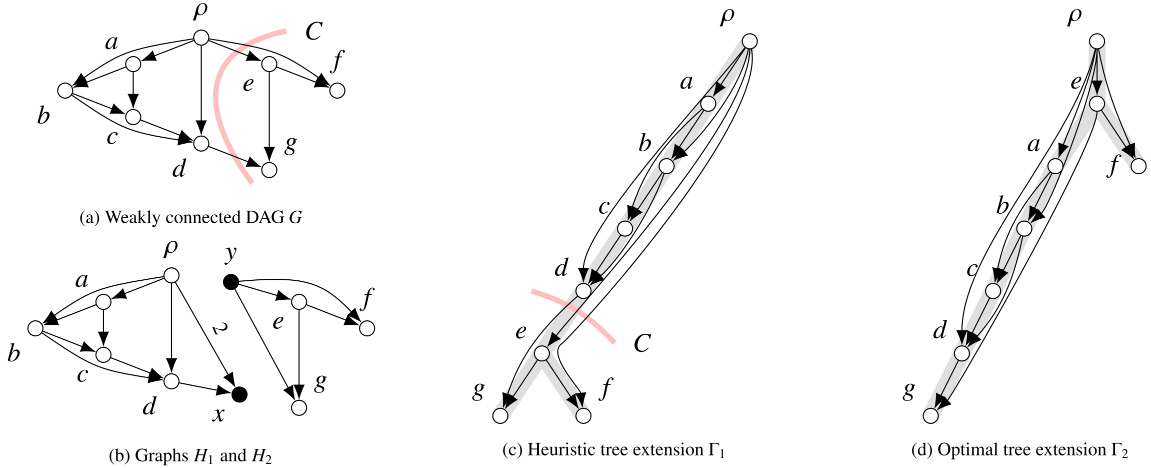


Fig. 12. (a): Weakly connected DAG G, with the unique minimum non-trivial DAG-cut C. (b): Weighted graphs H_1 and H_2 created after one iteration of Algorithm 4. The arc ρx has a weight of 2, while the other arcs have unit weights. The two 'supervertices' x and y are in black. (c): Canonical tree extension Γ_1 corresponding to the extension obtained by Algorithm 4. The cut C appears again. The tree extension is not optimal, and has a scanwidth of 6. (d): Optimal tree extension Γ_2 (of scanwidth 5) which does not contain the cut C.

the smallest non-trivial DAG-cut appears in an optimal extension. Consequently, the algorithm is not necessarily optimal. Nonetheless, the algorithm remains practical, running in polynomial time, as formalized in the following theorem.

Theorem 5.4. *Let $G = (V, E)$ be a weakly connected DAG of n vertices and m arcs. Then, Algorithm 4 returns an extension of G and runs in $O(n^4 \cdot m)$ time.*

Proof. *Correctness:* We will show that the subroutine `MinDAGCutSplit` always returns an extension of the graph H, which will imply that the complete algorithm returns an extension of G. We will do this by strong induction on the number of vertices k of H.

Base case: Whenever $k \in \{1, 2, 3\}$, H never has a non-trivial DAG-cut, because such a cut needs at least 2 vertices on either side. Then, the procedure returns an extension by reversing a BFS traversal.

Induction step: Let $k \geq 4$ be arbitrary, and assume that the statement holds for all $1 \leq \ell \leq k-1$. We can furthermore assume that H has a non-trivial DAG-cut, otherwise the procedure will automatically return an extension. We have $|S|, |T| \leq k-2$, since the DAG-cut is non-trivial and cuts off at least 2 vertices on either side. After adding the 'supervertices', we thus have that H_1 and H_2 both have at most $k-1$ vertices. By the induction hypothesis, σ_1 is then an extension of H_1. Because $H[S]$ is a subgraph of H_1, $\sigma_1[S]$ (which restricts σ_1 to S) is an extension of $H[S]$. Analogously, we can obtain that $\sigma_2[T]$ is an extension of $H[T]$. Since C was a DAG-cut, T is a sinkset of G. This then means that $\sigma = \sigma_2[T] \circ \sigma_1[S]$ is an extension of H.

Time complexity: Let $T(n, m)$ be the time the algorithm takes to run on a graph with n vertices and m arcs. According to the discussion after Lemma 5.3, finding the minimum non-trivial DAG-cut of such a graph (or showing that no non-trivial DAG-cut exists) can be done in $O(n^3 \cdot m)$ time. As the other non-recursive parts in the procedure are also dominated by this complexity and graphs with less than four vertices are not split any further, we obtain the following recurrence relation for some constant $c > 0$:

$$\begin{cases} T(3, m) \leq c, & \text{if } n = 3 \text{ (and so } m \leq 3); \\ T(n, m) \leq \max\limits_{2 \leq k \leq n-2} T(k+1, m) + T(n-k+1, m) + c \cdot n^3 \cdot m, & \text{if } n \geq 4. \end{cases}$$

We prove by strong induction on $n \geq 1$ that $T(n, m) \leq d \cdot n^4 \cdot m$ for some constant $d > 0$. The base cases for $n \leq 3$ are trivial. Now assume that the induction hypothesis holds for $1 \leq \ell < n$. Then, using the recurrence relation, we get

$$\begin{aligned} T(n, m) &\leq \max_{2 \leq k \leq n-2} \left\{ d \cdot (k+1)^4 \cdot m + d \cdot (n-k+1)^4 \cdot m \right\} + c \cdot n^3 \cdot m \\ &= d \cdot 3^4 \cdot m + d \cdot (n-2)^4 \cdot m + c \cdot n^3 \cdot m. \end{aligned}$$

In the last equality we used that the above maximum is attained at the extreme value of $k = 2$ (or equivalently, $k = n-2$) for fixed m and n, which can easily be shown with a concavity argument. We can now choose d large enough (and independent of m and n) such that this expression is at most $d \cdot n^4 \cdot m$. This proves the theorem. □

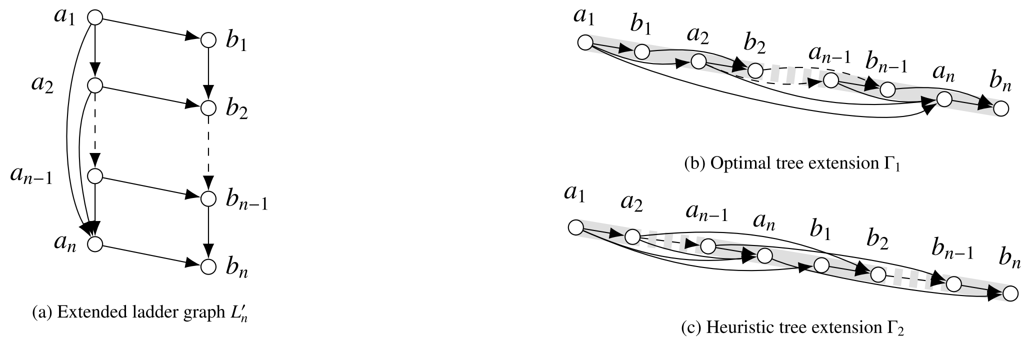


Fig. 13. (a): The *extended ladder-graph* L'_n (with $n \geq 3$), which is a weakly connected DAG. (b): An optimal tree extension Γ_1 of L'_n with scanwidth 5. (c): The worst-case tree extension Γ_2 of L'_n with scanwidth n. This is the canonical tree extension of the extension returned by the greedy heuristic.

5.2. Greedy heuristic and simulated annealing

In this subsection we discuss two other heuristic methods. Since simulated annealing and greedy algorithms are well known and intuitive, we will only briefly describe the general ideas and refer the interested reader to [16, Ch. 5] for the (algorithmic) details and (complexity) analyses.

Greedy heuristic The idea behind our *greedy algorithm* is to create an extension of seemingly small scanwidth by adding vertices one by one, each time adding the vertex that increases the scanwidth the least. This is achieved by maintaining a set S that keeps track of the vertices that still need to be added. In each iteration, the algorithm chooses the leaf of $G[S]$ that increases the scanwidth the least. The procedure is repeated until the set S is empty (or equivalently, σ contains all vertices of G). This consecutive 'picking' of leaves could lead to an optimal extension if the correct leaf would be chosen in each iteration. This is because we can create any extension by consecutively picking leaves of a graph.

The greedy rule will not necessarily pick the correct leaf each time. One can construct instances where the algorithm will perform very badly. An example is depicted in Fig. 13. The figure shows a class of graphs that all have scanwidth 5, yet the greedy algorithm will construct a solution of scanwidth n, with n half the number of vertices in the graph.

Simulated annealing A different approach to generate an extension with small scanwidth is to use *simulated annealing*: a metaheuristic that efficiently guides itself through the space of possible solutions. The algorithm starts with some initial extension, either created at random, or utilizing another heuristic. It then randomly selects a neighboring extension by swapping two vertices that are adjacent in the extension but that have no arc connecting them in the graph. This extension is accepted as the new solution if it has a smaller scanwidth. If the neighbor has a larger scanwidth, we still have a probability of accepting it depending on the decreasing *temperature* parameter: the lower the temperature, the less likely it becomes to accept a worse solution.

As a result of the high starting temperature, the algorithm will initially perform a fairly global and random search. Consequently, almost any solution will be accepted as the next state. Due to the 'cooling' of the temperature, the heuristic will slowly start to limit its choices to better solutions. Thus, it nudges itself to a (hopefully global) minimum.

6. Experimental results

In this section we conduct an experimental study to evaluate the performance of our exact algorithms, heuristics and reduction rules on phylogenetic networks. In Subsection 6.1 we will describe the networks that are used in the experiments and compare their level, scanwidth and treewidth. The subsequent subsections offer analyses of the reduction scheme from Section 3, the exact methods from Section 4, and the heuristics from Section 5, respectively.

All algorithms are implemented in Python and are publicly available as an easily installable package at

https://github.com/nholtgrefe/scanwidth.

This repository also provides the used networks and complete numerical results. All experiments in this section were conducted on an Intel Core i7-8750H CPU @ 2.20 GHz with 16 GB RAM.

6.1. Network generation

Closely following the experimental study from [26], we utilize both a dataset of real-world networks and a synthetically created dataset. The real data is made up of 27 real phylogenetic networks found in the literature, collected on [27]. Among these *real* networks, 15 are binary, while the remaining 12 are non-binary. The number of leaves ranges from 6 to 39, while the number of reticulations ranges from 1 to 9, except for one outlier with 32 reticulations.

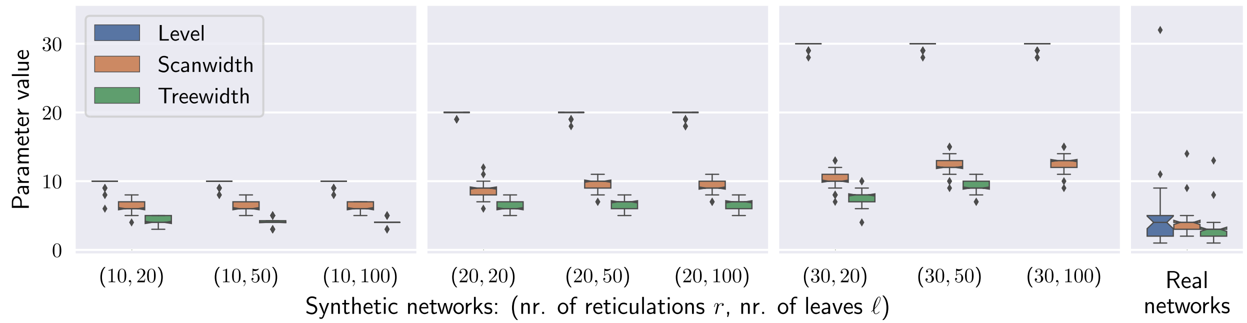


Fig. 14. Variation in level, scanwidth and treewidth within each dataset. The figure displays a boxplot for each of the nine subsets of the synthetic data and one for the real dataset. The boxplots show the quartiles of the data and its outliers. Different colors indicate the three different parameters. The treewidth values of the last synthetic dataset are not presented due to computational constraints, since obtaining these values required excessive computation times surpassing 19 hours for some networks.[5]

To augment our dataset we use the birth-hybridization network generator from [28], often called the *ZODS generator*. The method takes two input parameters: the speciation rate λ, and the hybridization rate ν. Following the computational experiments from [7,26,29], we set $\lambda = 1$ and sample ν uniformly at random from the interval $[0.0001, 0.4]$ for each individual network. We adapt the implementation from [26] to generate 100 binary networks for each pair of (r, ℓ), where $r \in \{10, 20, 30\}$ denotes the number of reticulations, and $\ell \in \{20, 50, 100\}$ the number of leaves. In total, this gives rise to a dataset comprising 900 *synthetic* networks.

Scornavacca and Weller [6] requested a comparison of the reticulation number, level, scanwidth and treewidth for different network classes. Fig. 14 functions as a partial answer to this call. It depicts boxplots that show the spread of the level, the treewidth and the scanwidth within each dataset.

For the synthetic data, we fixed the number of reticulations, explaining why the level has a sharp cutoff at those values in the figure. Moreover, it is evident that the ZODS generator favors networks with levels very close to the reticulation number. Notably, we observe that the level + 1 is greater than or equal to the scanwidth, which in turn is greater than or equal to the treewidth. This aligns with the bounds from Lemmas 2.7 and 2.8. As mentioned in the introductory section, scanwidth was proposed as an alternative for treewidth in parametrized algorithms. Although the level and the scanwidth exhibit a considerable difference in value, the treewidth is not much smaller than the scanwidth. This observation strengthens our belief in the practical value of scanwidth as a parameter.

Regarding the real networks, we see a somewhat different trend. The values of the three parameters are closer together in this case. We attribute this to the fact that most levels of the real networks are fairly small. As a consequence, there is limited 'room' for the scanwidth and the treewidth. The scanwidth, therefore, takes on predominantly small values, which further suggests its practicality. This is particularly promising since we have an algorithm capable of computing the scanwidth in polynomial time for fixed scanwidth, which becomes efficient when the scanwidth is small.

6.2. Reductions

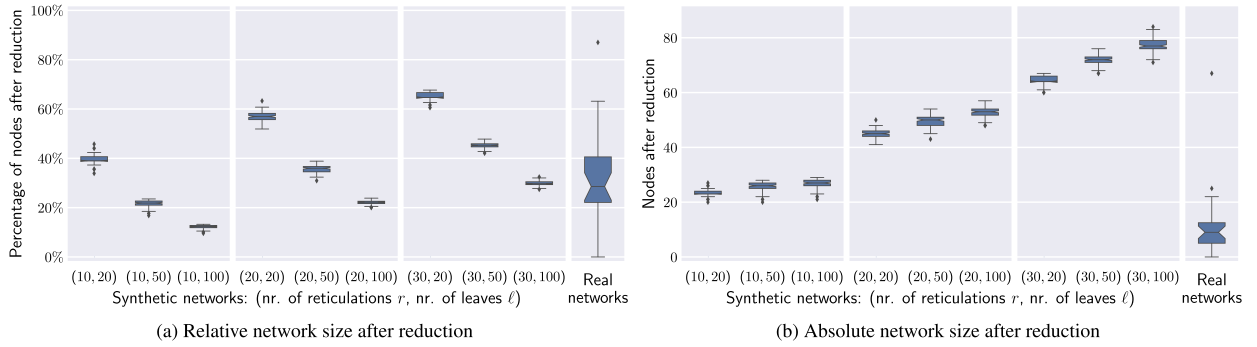
To test the effect the reduction rules from Section 3 have on the size of the networks, we employed the decomposition method (Algorithm 1) on each network. Fig. 15a showcases the percentage of the original vertices that remain after decomposition. Networks with fewer reticulations demonstrate greater potential for reduction. This is to be expected, as such networks are more tree-like, and many of their blocks thus have scanwidth 1 or 2. The decomposition algorithm effectively 'deletes' these blocks, leading to a significant reduction in the overall network size.

Regarding the number of leaves, we see a different relationship: a larger number of leaves corresponds to a greater reduction in size. This can be attributed to the fact that our reduction rules delete leaves of networks. Since the real networks vary in terms of reticulation numbers and number of leaves, their reduction percentages exhibit a wider range of values.

Fig. 15b shows the absolute number of vertices after the decomposition of the networks. Logically, networks with more leaves and more reticulations are larger, even after decomposing them. We also see that most of the real networks are relatively small after decomposition.

The time to reduce the networks is extremely small. This outcome is not surprising given that Lemma 3.5 proved the quadratic time complexity of the decomposition algorithm. The largest computation time for any of the instances was 0.056 seconds, while the average computation time remained below 0.005 seconds. All in all, the reduction rules prove to be beneficial without imposing substantial computational overhead. Therefore, we certainly recommend incorporating these reduction rules in practice.

[5] The values of the scanwidth are calculated using our exact algorithms, whose performance is discussed in Subsection 6.3. The values of the treewidth are calculated on a different CPU using a Java implementation by Tamaki of one of the fastest known exact algorithms [30,31].



(a) Relative network size after reduction

(b) Absolute network size after reduction

Fig. 15. Performance of the decomposition method (Algorithm 1). Figure (a) depicts how many vertices remain after the decomposition, as a percentage of the number of vertices of the original network. Figure (b) shows the absolute number of vertices after the decomposition. Both subfigures contain boxplots - showing quartiles and outliers of the data - for each of the nine subsets of the synthetic data and one for the real dataset.

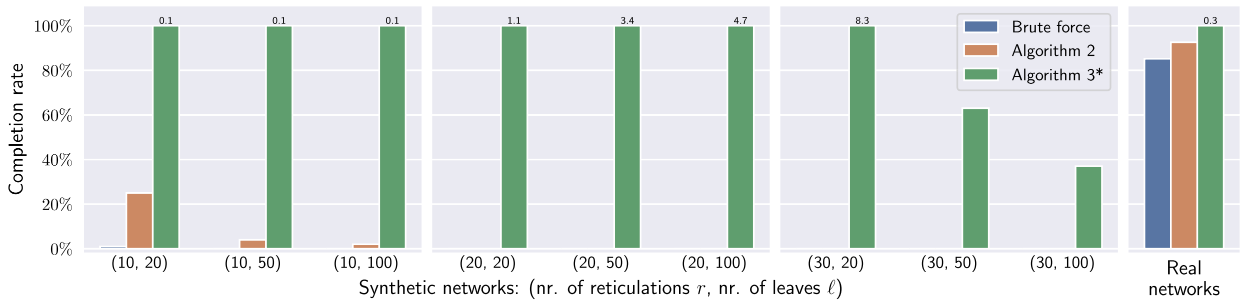


Fig. 16. Performance of the exact algorithms. For each algorithm, the completion rate, calculated as the number of instances per data set where the algorithm successfully computed the scanwidth within 60 seconds, is shown. If the rate is 100%, the average running etime in seconds is also depicted. The figure contains results for each of the nine subsets of the synthetic data and one for the real dataset. The different colors indicate the different algorithms. In all cases, the decomposition algorithm was also applied. The asterisk in the legend indicates that we repeatedly applied Algorithm 3 as outlined in the proof of Corollary 4.12, since the algorithm itself only solves the fixed-parameter version of the problem.

6.3. Exact algorithms

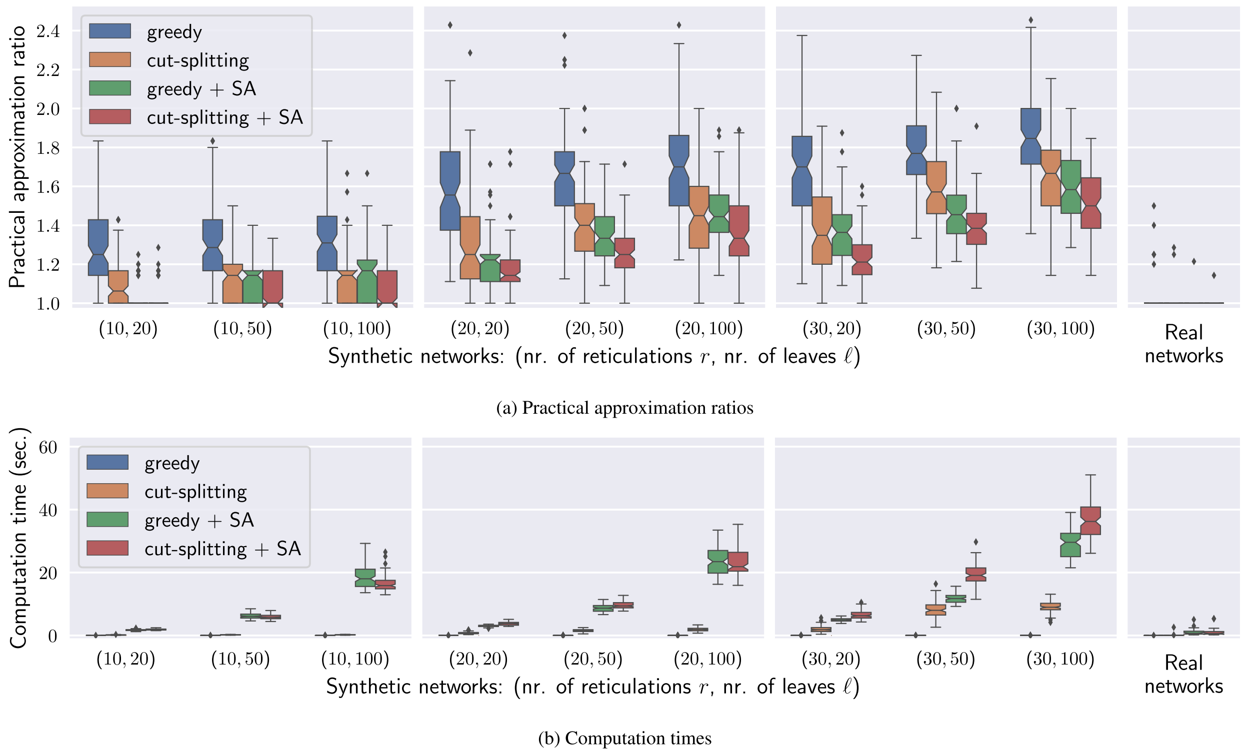
In Section 4 we explored multiple exact algorithms and their respective time complexities. A brute force solution runs in $\tilde{O}(n!)$ time, while the recursive Algorithm 2 runs in $\tilde{O}(4^n)$ time. The theoretically superior algorithm repeatedly applies the fixed-parameter Algorithm 3, as outlined in the proof of Corollary 4.12. Since phylogenetic networks have a single root by definition (and thus $r = 1$), the time complexity from Corollary 4.12 simplifies to $O(k \cdot m \cdot n^k)$, where k represents the scanwidth.

While we have proved that these algorithms yield optimal solutions, it is interesting to assess how fast they run in practice. For each of the above-described algorithms, Fig. 16 provides insight into the percentage of networks for which the scanwidth can be determined within 60 seconds, after first applying the decomposition algorithm. Throughout the different data (sub)sets, the order of the algorithms concerning their completion rates aligns with the theoretical time complexities of the algorithms. The brute-force solution has the smallest completion rate, while (the repeated application of) Algorithm 3 achieves the highest.

Interestingly, for the real networks, (the repeated application of) Algorithm 3 achieves a 100% completion rate and an average computation time of just 0.3 seconds. Thus, on the real dataset the algorithms perform extremely well. However, the brute-force solution also attains a significant completion rate of 85%, indicating that most of these real instances are not too hard after applying the decomposition algorithm.

In general, the completion rates drop as the number of leaves and reticulations increases. This is in line with our discussion from the previous two subsections, where it is noted that these networks are larger and have a higher scanwidth and level.

Looking at the complete synthetic dataset, the best-performing algorithm (i.e. repeated application of Algorithm 3) had a completion rate of 88.9% within 60 seconds. Additionally, we allowed this algorithm to run indefinitely to get the scanwidth values for all networks. After 300 seconds, the overall completion rate increased to 98.9%. The maximum computation time of any of the networks using this algorithm turned out to be 453.52 seconds. Hence, for all generated networks, with up to 30 reticulations and 100 leaves, the scanwidth could be computed exactly within 8 minutes.



(a) Practical approximation ratios

(b) Computation times

Fig. 17. Performance of the different heuristics. Boxplots are shown for each of the nine subsets of the synthetic data and the real dataset. The boxplots show the quartiles of the data and its outliers. Figure (a) displays the practical approximation ratios, while Figure (b) depicts the computation times. The computation times for simulated annealing (SA) include the computation time to obtain the initial tree extension. We also applied the decomposition algorithm in all cases.

6.4. Heuristics

In the previous section, we observed that our fastest algorithm is able to find all optimal tree extensions within 500 seconds. If less time is allowed, we need to turn our attention to heuristics. We evaluate the performance of the greedy heuristic (see Subsection 5.2) and the cut-splitting heuristic (Algorithm 4) developed in Section 5. Additionally, we apply simulated annealing (see Subsection 5.2) to both results and compare the performances. Note that we have not proved any theoretical bounds on the approximation ratios of the heuristics. We refer to [16] for the used parameter-settings of the simulated annealing algorithm.

Fig. 17 presents the results of the experiment. In Fig. 17a the practical approximation ratios obtained by the heuristics are shown for the different datasets. First of all, we observe that for the real networks, the practical approximation ratios are very close to 1. In fact, the whiskers of the boxplots are not even shown, indicating that for most networks all heuristics can find an optimal (tree) extension. Applying the cut-splitting heuristic together with simulated annealing even solves all but one network to optimality.

For the synthetic data, the approximation ratios increase with the number of reticulations and leaves. This is attributed to the fact that such networks are inherently more complex and thus more challenging. It is fairly clear that the greedy heuristic performs the worst, although the practical approximation ratio stays below 2.5. In all cases, the cut-splitting heuristic consistently shows significant improvement over the greedy algorithm. Lastly, applying simulated annealing indeed improves the solution quality even more, albeit at the cost of more computation time.

The computation times of the heuristics are visualized in Fig. 17b. It is apparent that simulated annealing takes considerably more computation time compared to just applying the heuristics on their own. Furthermore, we see that the better-performing heuristics require more time than the less effective ones.

7. Conclusions

The main contribution of this paper is threefold: we have presented and implemented a relatively fast exact algorithm to compute the scanwidth, we have provided insight into the parametrized complexity of the Scanwidth problem, and we developed a heuristic to compute the scanwidth for instances that are too large for the exact method.

Regarding the first topic, we first improve upon a brute-force approach by a recursive exponential time algorithm. The recursive relations in this method form the foundation of an exact algorithm that can compute the scanwidth of DAGs

with a fixed number of roots in slicewise polynomial time for fixed scanwidth. This algorithm iterates from top-to-bottom through different vertex subsets of a graph while storing intermediate results. The time complexity can be bounded by $O((k+r-1)\cdot m\cdot n^{k+r-1})$ for DAGs with r roots, where k is the scanwidth. Furthermore, in combination with a decomposition algorithm, the time complexity of the algorithm can be bounded by $O(2^{4\ell-1}\cdot \ell\cdot n+n^2)$ for level-ℓ networks.

The worst-case time complexity of this algorithm shows that for DAGs with a fixed number of roots Scanwidth is part of the complexity class XP with respect to its natural parametrization (i.e. with respect to the parameter scanwidth itself). Moreover, the above time complexity for level-ℓ networks shows that Scanwidth is fixed-parameter tractable with respect to the level of a network. Specifically, it takes quadratic time to calculate the scanwidth of a network when the level is fixed. It remains open whether Scanwidth lies in XP with respect to its natural parametrization when the number of roots of the DAG is unbounded. An even stronger open question is whether the problem is FPT with respect to its natural parametrization (with, or possibly without, a bounded number of roots), as is known to be the case for cutwidth [12] and treewidth [32].

We observe that the cuts in a tree extension are of a specific type: DAG-cuts. Using the fact that one can find a smallest (non-trivial) DAG-cut in polynomial time, we are able to efficiently keep splitting a graph into smaller subgraphs at these cuts. Although not necessarily optimal, we showed that this heuristic performs great in practice, especially when enhanced with simulated annealing.

Tested on a set of 27 real-world rooted phylogenetic networks, the exact XP algorithm performs best. The algorithm is able to compute the scanwidth of any network within 7.86 seconds, averaging a computation time of just 0.30 seconds. On a synthetic dataset of networks, the algorithm struggles with the harder instances, albeit still able to compute any scanwidth within 500 seconds. On these fairly hard instances—with 30 reticulations and 100 leaves—the earlier described heuristic attains an average practical approximation ratio of 1.5.

These experimental results show that computing the scanwidth exactly or finding a near-optimal solution can surely be done in a reasonable time. Additionally, we show that in practice, the treewidth—scanwidth's main competitor when it comes to parametrized algorithms—is not much smaller for networks. Scanwidth is also more intuitive for phylogenetic networks than treewidth. Together, these observations motivate the use of scanwidth (and the corresponding tree extensions) when designing parametrized algorithms in phylogenetics. No treewidth algorithm is known for Hybridization Number [33], thus making it a possible candidate for a scanwidth-based approach. On the other hand, Tree Containment [4] and Small Parsimony [6] can be parametrized by treewidth, but might be solved faster in practice when parametrized by scanwidth.

Another direction of possible further research would be the transferability of some of our results to edge-treewidth, introduced in [11]. As this parameter is very closely related to scanwidth, it seems that translating our algorithms to edge-treewidth is not far-fetched. Perhaps, it is also possible to adapt our algorithms to *node-scanwidth*, where instead of arcs we are interested in the tails of the arcs (similar to Definition 2.6 of treewidth). A possibly simpler generalization would be the translation of our results to arc-weighted DAGs or multigraphs.

Furthermore, it is not known whether scanwidth can be approximated efficiently. Recent research into the inapproximability of width parameters suggests that treewidth and undirected cutwidth are inapproximable up to a constant factor within polynomial time [34]. It is not unthinkable that these inapproximability results translate to scanwidth. On the positive side, treewidth can be approximated within a ratio only linearly dependent on the treewidth itself (see [35] for the state-of-the-art and an overview of other approximation algorithms).

Lastly, we address our experimental study. Our synthetic networks were created with a single generating method [28]. A sensible next step is then to extend the study to other network generators (see [29]). Moreover, we could look into specific types of networks, both from a computational and a theoretical standpoint. We already considered networks with fixed reticulation numbers and levels, but other classes also exist (see [36] for a recent survey). During our experiments, we observed that scanwidth was a lot faster to compute than treewidth. We would thus welcome efforts towards a thorough comparison of the practical computability of the two parameters, both from an exact and a heuristic point of view.

CRediT authorship contribution statement

Niels Holtgrefe: Writing – review & editing, Writing – original draft, Visualization, Software, Methodology, Conceptualization. **Leo van Iersel:** Writing – review & editing, Visualization, Methodology, Conceptualization. **Mark Jones:** Writing – review & editing, Visualization, Methodology, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We extend our sincere appreciation to Mathias Weller and Norbert Zeh for insightful discussions on the topic of this paper.

Appendix A. Omitted proofs

Lemma A.1. *Let $G = (V, E)$ be a weakly connected DAG, and let Γ and Ω be two tree extensions of G. Then, $\mathrm{GW}_v^\Gamma = \mathrm{GW}_v^\Omega$ for all $v \in V$ if and only if $\Gamma = \Omega$. Therefore, a tree extension is uniquely determined by the sets* GW.

Proof. The 'if direction' is trivial, so it remains to prove the 'only-if direction'. Assume that Γ and Ω are two different tree extensions of the graph G. For any $v \in V$, we write Γ_v (resp. Ω_v) for the subtree of Γ (resp. Ω) rooted at v. As $\Gamma \neq \Omega$, we can assume without loss of generality that $V(\Gamma_u) \not\subseteq V(\Omega_u)$ for some $u \in V$.

Let y be a vertex in $V(\Gamma_u) \setminus V(\Omega_u)$ that is highest in Γ. Assuming that G is non-trivial, we get three cases. (i): y has a parent x in G. Then $x \notin V(\Omega_u)$ and $x \notin V(\Gamma_u)$ (otherwise, x would be a higher vertex in $V(\Gamma_u) \setminus V(\Omega_u)$). Then, $xy \in \mathrm{GW}_u^\Gamma \setminus \mathrm{GW}_u^\Omega$. (ii): y does not have a parent in G and some successor of y in G is in $V(\Omega_u)$. (A *successor* of y is a vertex $z <_G y$.) Let z be the highest such successor in G, then there exist some arc $xz \in E$ and $xz \in \mathrm{GW}_u^\Omega \setminus \mathrm{GW}_u^\Gamma$. (iii): y does not have a parent in G and all successors of y in G are not in $V(\Omega_u)$. Since $V(\Gamma_u) \neq V$, there must be some arc xz entering $V(\Gamma_u)$. Using that y is a root in G, we can choose xz such that z has a successor that is also a successor of y. But then, x and z are not in $V(\Omega_u)$, and so $xz \in \mathrm{GW}_u^\Gamma \setminus \mathrm{GW}_u^\Omega$.

In all cases, $\mathrm{GW}_u^\Gamma \neq \mathrm{GW}_u^\Omega$. This proves the lemma, and consequently, the fact that the sets GW uniquely determine a tree extension. $\square$

Proposition 2.5. *Let $G = (V, E)$ be a weakly connected DAG, Γ a tree extension of G, and σ an extension of Γ. For each $v \in V$, let Γ_v be the subtree of Γ rooted at v. Then, Γ is the canonical tree extension for σ if and only if $G[V(\Gamma_v)]$ is weakly connected for all $v \in V$.*

Proof. The 'only-if direction' follows directly from [2, Lem. 5f], thus it remains to prove the 'if direction'. Let Γ be a tree extension that is not canonical for σ. We now claim that there exists a $v \in V$ such that $\mathrm{SW}_v^\sigma \subset \mathrm{GW}_v^\Gamma$.

Proof of claim. Let $v \in V$ and $xy \in \mathrm{SW}_v^\sigma$ be arbitrary. By definition, $x >_\sigma v \geq_\sigma y$ and $y \overset{G[1\dots v]}{\leftrightsquigarrow} v$. As σ is an extension of Γ, together this gives that $v \geq_\Gamma y$. Furthermore, x and v can not be incomparable in Γ, since $y \overset{G[1\dots v]}{\leftrightsquigarrow} v$ and xy is an arc of G. Combining with the fact that $x >_\sigma v$ and that σ is an extension of Γ, we must then have $x >_\Gamma v$. This means that $xy \in \mathrm{GW}_v^\Gamma$. So, $\mathrm{SW}_v^\sigma \subseteq \mathrm{GW}_v^\Gamma$.

According to Lemma A.1, there exists some $v \in V$ such that $\mathrm{GW}_v^\Gamma \neq \mathrm{GW}_v^{\Gamma^\sigma}$. By [2, Lem. 5i], we then get that $\mathrm{SW}_v^\sigma = \mathrm{GW}_v^{\Gamma^\sigma} \neq \mathrm{GW}_v^\Gamma$. We already had that $\mathrm{SW}_v^\sigma \subseteq \mathrm{GW}_v^\Gamma$, so $\mathrm{SW}_v^\sigma \subset \mathrm{GW}_v^\Gamma$. $\triangle$

Let v be as in the claim, then there exists an arc $xy \in \mathrm{GW}_v^\Gamma$ that is not in SW_v^σ. But since σ is an extension of Γ, we must have that $x >_\sigma v \geq_\sigma y$. Then, $xy \notin \mathrm{SW}_v^\sigma$ implies that y is not weakly connected to v in $G[\sigma[1\dots v]]$. Clearly, $V(\Gamma_v) \subseteq \sigma[1\dots v]$, as σ is an extension of Γ. But then, y is also not weakly connected to v in $G[V(\Gamma_v)]$. This means that $G[V(\Gamma_v)]$ is not weakly connected. $\square$

Lemma 2.7. *Let $G = (V, E)$ be a weakly connected DAG and $\tilde{G}$ its underlying undirected graph, then*

$$\mathrm{tw}(\tilde{G}) \leq \mathrm{sw}(G).$$

Proof. Let Γ be some tree extension of G and $v \in V$ arbitrary. Clearly, Γ is also a tree layout for $\tilde{G}$. By definition, $\mathrm{GW}_v^\Gamma(G) = \{xy \in E : x >_\Gamma v \geq_\Gamma y\}$. We now define the mapping $\phi : \mathrm{GW}_v^\Gamma(G) \to \mathrm{TW}_v^\Gamma(\tilde{G})$ as $\phi(xy) = x$. We first show that ϕ indeed maps all elements of $\mathrm{GW}_v^\Gamma(G)$ to $\mathrm{TW}_v^\Gamma(\tilde{G})$. For any $xy \in \mathrm{GW}_v^\Gamma(G)$, we surely have that $x \in V$ and that $x >_\Gamma v$. If we now set $y = w$, we immediately find the w with $w \leq_\Gamma v$, such that $xw \in E$. Thus, x is indeed in $\mathrm{TW}_v^\Gamma(\tilde{G})$.

Secondly, we prove that ϕ is surjective. To this end, let $u \in \mathrm{TW}_v^\Gamma(\tilde{G})$ be arbitrary. Then, there exists at least one w such that $u >_\Gamma v \geq_\Gamma w$ and $uw \in E$. But then, it holds that $uw \in \mathrm{GW}_v^\Gamma(G)$, and so $\phi(uw) = u$, proving that ϕ is surjective.

Since ϕ is a surjective mapping from $\mathrm{GW}_v^\Gamma(G)$ to $\mathrm{TW}_v^\Gamma(\tilde{G})$, it must hold that $|\mathrm{GW}_v^\Gamma(G)| \geq |\mathrm{TW}_v^\Gamma(\tilde{G})|$. Then also, $\mathrm{tw}(\Gamma, G) = \max_v |\mathrm{TW}_v^\Gamma(\tilde{G})| \leq \max_v |\mathrm{GW}_v^\Gamma(G)| = \mathrm{sw}(\Gamma, G)$. Finally, letting Γ be such that $\mathrm{sw}(G) = \mathrm{sw}(\Gamma, G)$ proves the bound. $\square$

Recall that we write $W \sqsubseteq V$ to indicate that W is a sinkset of a DAG $G = (V, E)$.

Lemma A.2. *Let $G = (V, E)$ be a network with reticulation number k. Then, for all $W \sqsubseteq V$ such that $G[W]$ is weakly connected, we have that $\delta^{\mathrm{in}}(W) \leq k + 1$.*

Proof. For any DAG H and $W \subseteq V(H)$, let $r_W(H) = \sum_{v \in W : \delta^{\mathrm{in}}(v) \geq 2} (\delta^{\mathrm{in}}(v) - 1)$. We will now prove the stronger statement that for any DAG $G = (V, E)$ and for all $W \sqsubseteq V$ with $G[W]$ weakly connected, $\delta^{\mathrm{in}}(W) \leq r_W(G) + 1$. For any such set W, we will prove this by induction on $r_W(G)$. This will immediately imply the lemma, since any network is a DAG and $r_W(G)$ is at most the reticulation number k of a network G.

Base case: ($r_W(G) = 0$). Let G be a DAG with $W \sqsubseteq V$ such that $G[W]$ is weakly connected and $r_W(G) = 0$. Then, W contains no reticulation vertices, otherwise $r_W(G) > 0$. As W is a weakly connected sinkset, $G[W]$ must then either be a pendent tree of G, or $G[W] = G$. In both cases, $\delta^{\text{in}}(W) \le 1 = r_W(G) + 1$.

Induction step: ($r_W(G) \ge 1$). Let G be a DAG with $W \sqsubseteq V$ such that $G[W]$ is weakly connected and $r_W(G) \ge 1$. Assume that the induction hypothesis holds for all DAGs H and sets $W' \sqsubseteq V(H)$ such that $r_{W'}(H) < r_W(G)$. As $r_W(G) \ge 1$, we know that W contains a reticulation vertex. We can then pick an arc $uv \in E$ such that v is a reticulation vertex in W. It is easy to see that W is still a sinkset of $G' = G - \{uv\}$, and that $r_W(G') = r_W(G) - 1$. We now consider two cases and show that $\delta^{\text{in}}_G(W) \le r_W(G) + 1$, which will conclude the proof.

Case 1: If $G'[W]$ is weakly connected, the induction hypothesis shows that $\delta^{\text{in}}_{G'}(W) \le r_W(G') + 1$. But then, if we add uv back to G' to obtain G again, we can only increase the indegree of W by 1, and it follows that $\delta^{\text{in}}_G(W) \le \delta^{\text{in}}_{G'}(W) + 1 \le r_W(G') + 2 = r_W(G) + 1$.

Case 2: If $G'[W]$ is not weakly connected, the deletion of uv must have disconnected $G[W]$. Deletion of one arc can only increase the number of weakly connected components by one. Therefore, $G'[W]$ consists of two weakly connected components: $G'[W_1]$ and $G'[W_2]$. Because $G'[W_1]$ and $G'[W_2]$ are disconnected in G', and W was a sinkset in G', we must have that W_1 and W_2 are sinksets of G'. Since W_1 and W_2 partition W, we have $r_W(G') = r_{W_1}(G') + r_{W_2}(G')$.

For both $i = 1$ and $i = 2$, we now get that $r_{W_i}(G') \le r_W(G') = r_W(G) - 1$. Furthermore, $G'[W_i]$ is a weakly connected sinkset, as discussed above. Thus, we can apply the induction hypothesis, and get $\delta^{\text{in}}_{G'}(W_i) \le r_{W_i}(G') + 1$ for $i = 1, 2$. Using this and the fact that no arc exists between W_1 and W_2 in G', we get $\delta^{\text{in}}_{G'}(W) = \delta^{\text{in}}_{G'}(W_1) + \delta^{\text{in}}_{G'}(W_2) \le (r_{W_1}(G') + 1) + (r_{W_2}(G') + 1) = r_W(G') + 2$. Since uv disconnected $G[W]$, u and v must both be in W. Consequently, the arc uv can not count towards the indegree of W. From this, it follows that $\delta^{\text{in}}_G(W) = \delta^{\text{in}}_{G'}(W) \le r_W(G') + 2 = r_W(G) + 1$. □

Lemma 2.8. *Let $G = (V, E)$ be a level-k network, then*

$$\text{sw}(G) \le k + 1.$$

Proof. Let r be the reticulation number of G. We will prove that the scanwidth of G is at most $r + 1$. The result then follows from the definition of the level and from Corollary 3.3, which shows that the scanwidth of a rooted weakly connected DAG is equal to the maximum scanwidth of its blocks.

Let σ be an optimal extension of G. Let $v \in V$ be arbitrary, and consider the set SW^σ_v. We can write $|\text{SW}^\sigma_v| = \delta^{\text{in}}(S)$, where $S \subseteq \sigma[1 \dots v]$ contains all vertices that are weakly connected to v in $G[1 \dots v]$. Clearly, S is also a sinkset (as σ is an extension of G). According to Lemma A.2, we then have that $|\text{SW}^\sigma_v| = \delta^{\text{in}}(S) \le r + 1$. Since v was arbitrary, $\text{sw}(G) = \max_{v \in V} |\text{SW}^\sigma_v| \le r + 1$. □

Data availability

All algorithms are implemented in Python and are publicly available at https://github.com/nholtgrefe/scanwidth. This repository also provides the used networks and complete numerical results.

References

[1] R. Diestel, Graph Theory, Springer Berlin Heidelberg, 2017.

[2] V. Berry, C. Scornavacca, M. Weller, Scanning phylogenetic networks is NP-hard, in: 46th International Conference on Current Trends in Theory and Practice of Informatics (SOFSEM 2020), 2020, pp. 519–530.

[3] C.-E. Rabier, V. Berry, M. Stoltz, J.D. Santos, W. Wang, J.-C. Glaszmann, F. Pardi, C. Scornavacca, On the inference of complex phylogenetic networks by Markov chain Monte-Carlo, PLoS Comput. Biol. 17 (9) (2021), https://doi.org/10.1371/journal.pcbi.1008380.

[4] L. van Iersel, M. Jones, M. Weller, Embedding phylogenetic trees in networks of low treewidth, Discrete Math. Theor. Comput. Sci. 25 (4) (2023) 2, https://doi.org/10.46298/dmtcs.10116.

[5] L. van Iersel, S. Kelk, G. Stamoulis, L. Stougie, O. Boes, On unrooted and root-uncertain variants of several well-known phylogenetic network problems, Algorithmica 80 (11) (2017) 2993–3022, https://doi.org/10.1007/s00453-017-0366-5.

[6] C. Scornavacca, M. Weller, Treewidth-based algorithms for the small parsimony problem on networks, Algorithms Mol. Biol. 17 (1) (2022), https://doi.org/10.1186/s13015-022-00216-w.

[7] G. Bernardini, L. van Iersel, E. Julien, L. Stougie, Constructing phylogenetic networks via cherry picking and machine learning, Algorithms Mol. Biol. 18 (1) (2023) 13, https://doi.org/10.1186/s13015-023-00233-3.

[8] S. Borst, L. van Iersel, M. Jones, S. Kelk, New FPT algorithms for finding the temporal hybridization number for sets of phylogenetic trees, Algorithmica 84 (7) (2022) 2050–2087, https://doi.org/10.1007/s00453-022-00946-8.

[9] L. van Iersel, S. Kelk, R. Rupp, D. Huson, Phylogenetic networks do not need to be complex: using fewer reticulations to represent conflicting clusters, Bioinformatics 26 (12) (2010) i124–i131, https://doi.org/10.1093/bioinformatics/btq202.

[10] L. Bulteau, M. Weller, Parameterized algorithms in bioinformatics: an overview, Algorithms 12 (12) (2019) 256, https://doi.org/10.3390/a12120256.

[11] L. Magne, C. Paul, A. Sharma, D.M. Thilikos, Edge-treewidth: algorithmic and combinatorial properties, Discrete Appl. Math. 341 (2023) 40–54, https://doi.org/10.1016/j.dam.2023.07.023.

[12] H.L. Bodlaender, M.R. Fellows, D.M. Thilikos, Derivation of algorithms for cutwidth and related graph layout parameters, J. Comput. Syst. Sci. 75 (4) (2009) 231–244, https://doi.org/10.1016/j.jcss.2008.10.003.

[13] J. Díaz, J. Petit, M. Serna, A survey of graph layout problems, ACM Comput. Surv. 34 (3) (2002) 313–356, https://doi.org/10.1145/568522.568523.

[14] J. Petit, Addenda to the survey of layout problems, Bull. Eur. Assoc. Theor. Comput. Sci. 3 (105) (2013).

[15] H.L. Bodlaender, F.V. Fomin, A.M.C.A. Koster, D. Kratsch, D.M. Thilikos, A note on exact algorithms for vertex ordering problems on graphs, Theory Comput. Syst. 50 (3) (2011) 420–432, https://doi.org/10.1007/s00224-011-9312-0.

[16] N. Holtgrefe, Computing the scanwidth of directed acyclic graphs, Master's thesis, Delft University of Technology, Delft, the Netherlands, July 2023, available at http://resolver.tudelft.nl/uuid:9c82fd2a-5841-4aac-8e40-d4d22542cdf5.

[17] J. Nešetřil, P.O. De Mendez, Tree-depth, subgraph coloring and homomorphism bounds, Eur. J. Comb. 27 (6) (2006) 1022–1041, https://doi.org/10.1016/j.ejc.2005.01.010.

[18] J. Hopcroft, R. Tarjan, Algorithm 447: efficient algorithms for graph manipulation, Commun. ACM 16 (6) (1973) 372–378, https://doi.org/10.1145/362248.362272.

[19] L. van Iersel, Algorithms, haplotypes and phylogenetic networks, Ph.D. thesis, Eindhoven University of Technology, Eindhoven, the Netherlands, January 2009, available at https://ir.cwi.nl/pub/17418/17418B.pdf.

[20] L. van Iersel, J. Keijsper, S. Kelk, L. Stougie, F. Hagen, T. Boekhout, Constructing level-2 phylogenetic networks from triplets, IEEE/ACM Trans. Comput. Biol. Bioinform. 6 (4) (2009) 667–681, https://doi.org/10.1109/TCBB.2009.22.

[21] Y. Gurevich, S. Shelah, Expected computation time for hamiltonian path problem, SIAM J. Comput. 16 (3) (1987) 486–502, https://doi.org/10.1137/0216034.

[22] H.L. Bodlaender, F.V. Fomin, A.M.C.A. Koster, D. Kratsch, D.M. Thilikos, On exact algorithms for treewidth, ACM Trans. Algorithms 9 (1) (2012), https://doi.org/10.1145/2390176.2390188.

[23] M. Held, R.M. Karp, A dynamic programming approach to sequencing problems, J. Soc. Ind. Appl. Math. 10 (1) (1962) 196–210, https://doi.org/10.1137/0110015.

[24] R. Ravi, A. Agrawal, P. Klein, Ordering problems approximated: single-processor scheduling and interval graph completion, in: International Colloquium on Automata, Languages, and Programming (ICALP 1991), 1991, pp. 751–762.

[25] J.B. Orlin, Max flows in $O(nm)$ time, or better, in: 45th Annual ACM Symposium on Theory of Computing (STOC 2013), 2013, pp. 765–774.

[26] L. van Iersel, M. Jones, E. Julien, Y. Murakami, Making a network orchard by adding leaves, in: 23rd International Workshop on Algorithms in Bioinformatics (WABI 2023), 2023, pp. 7:1–7:20.

[27] T. Agarwal, P. Gambette, D. Morrison, Who is who in phylogenetic networks: articles, authors and programs, http://phylnet.univ-mlv.fr/, 2016. (Accessed 26 March 2023).

[28] C. Zhang, H.A. Ogilvie, A.J. Drummond, T. Stadler, Bayesian inference of species networks from multilocus sequence data, Mol. Biol. Evol. 35 (2) (2017) 504–517, https://doi.org/10.1093/molbev/msx307.

[29] R. Janssen, P. Liu, Comparing the topology of phylogenetic network generators, J. Bioinform. Comput. Biol. 19 (06) (2021), https://doi.org/10.1142/s0219720021400126.

[30] H. Tamaki, Heuristic computation of exact treewidth, arXiv:2202.07793, 2022.

[31] H. Tamaki, tw, https://github.com/twalgor/tw, 2022, repository.

[32] H.L. Bodlaender, A linear-time algorithm for finding tree-decompositions of small treewidth, SIAM J. Comput. 25 (6) (1996) 1305–1317, https://doi.org/10.1137/S0097539793251219.

[33] M. Bordewich, C. Semple, Computing the minimum number of hybridization events for a consistent evolutionary history, Discrete Appl. Math. 155 (8) (2007) 914–928, https://doi.org/10.1016/j.dam.2006.08.008.

[34] Y. Wu, P. Austrin, T. Pitassi, D. Liu, Inapproximability of treewidth and related problems, J. Artif. Intell. Res. 49 (2014) 569–600, https://doi.org/10.1613/jair.4030.

[35] T. Korhonen, A single-exponential time 2-approximation algorithm for treewidth, in: IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS 2021), 2022, pp. 184–192.

[36] S. Kong, J.C. Pons, L. Kubatko, K. Wicke, Classes of explicit phylogenetic networks and their biological and mathematical significance, J. Math. Biol. 84 (6) (2022) 47, https://doi.org/10.1007/s00285-022-01746-y.