Computing the Scanwidth of Directed Acyclic Graphs

Niels Holtgrefe

Delft University of Technology

TU Delft

Computing the Scanwidth of Directed Acyclic Graphs

by

Niels Holtgrefe

to obtain the degree of Master of Science in Applied Mathematics
at the Delft University of Technology,
to be defended publicly on Wednesday, 12 July 2023 at 10:15.

Student number:	4954181	
Project duration:	1 December 2022 – 12 July 2023	
Thesis committee:	Dr. ir. L. J. J. van Iersel,	TU Delft, supervisor
	Dr. M. E. L. Jones,	TU Delft, supervisor
	Dr. M. T. J. Spaan,	TU Delft

A digital version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Abstract

Phylogenetic networks are a specific type of directed acyclic graph (DAG), used to depict evolutionary relationships among, for example, species or other groups of organisms. To solve computationally hard problems, treewidth has been used to parametrize algorithms in phylogenetics. In the hope of simplifying the algorithmic design process, Berry, Scornavacca and Weller [BSW20] recently proposed a new measure of tree-likeness that takes into account the directions of the arcs: *scanwidth*. They showed that the corresponding decision problem of this parameter - which can be seen as a variant of directed cutwidth, using a tree instead of a linear ordering - is NP-complete. This thesis aims to widen the structural knowledge of scanwidth and to find efficient ways of computing it on general DAGs, both by exact and heuristic algorithms.

With the help of reduction rules, we construct an explicit dynamic programming algorithm that computes scanwidth exactly, along with its corresponding *tree extension*, in $O(k \cdot n^k \cdot m)$ time for rooted DAGs of scanwidth k. This slicewise polynomial algorithm proves that computing the scanwidth is in the complexity class XP. The algorithm also functions as an FPT algorithm for networks of level-ℓ, with the complexity bounded by $O(2^{4\ell-1} \cdot \ell \cdot n + n^2)$. It performs well in practice, being able to compute the scanwidth of networks up to 30 reticulations and 100 leaves within 500 seconds.

On the heuristic side, an algorithm that repeatedly splits at a specific type of smallest cut is proposed. Enhanced with simulated annealing, this heuristic shows promising results, obtaining an average approximation ratio of 1.5 for large synthetic networks of 30 reticulations and 100 leaves. Applied to a real-world dataset of networks, the heuristic performs near-optimal. Although we prove that the scanwidth is always greater than or equal to the treewidth, experiments show that they are close to each other in practice. This further motivates the use of scanwidth over treewidth as a parameter in algorithms.

Preface

Since 2018, I have been studying the wonders of mathematics at TU Delft. First as part of my bachelor's degree, and currently to obtain my master's degree in applied mathematics. This thesis project, which I started in December 2022, marks the end of my period as a graduate student. I thoroughly enjoyed doing research and writing this report.

I owe lots of gratitude to my two supervisors Leo van Iersel and Mark Jones. Our weekly meetings were extremely fruitful and also very fun. I appreciate the time and effort they put in to help me during this project. I thank them for allowing me to focus on the topics I found interesting and for supporting me throughout the project.

I would also like to thank Matthijs Spaan for completing my thesis committee, attending my thesis defence, and assessing this report.

Furthermore, I would like to thank Matthias Weller from Université Gustave Eiffel in Champs-sur-Marne, France. We had two very interesting online meetings on the topic of this thesis, which contributed to the algorithm in Section 5.2.

Lastly, I want to thank my family and friends for supporting me during this project. Special thanks go towards my younger brother Tim for proofreading parts of the thesis and reviewing the many adjustments I made to the figures.

Niels Holtgrefe,
June 2023

Contents

Abstract ii

Preface iii

1 Introduction 1
- 1.1 Phylogenetics . . . 1
- 1.2 Scanning a network . . . 1
- 1.3 Main contributions and thesis outline . . . 3

2 Preliminaries 5
- 2.1 Complexity theory . . . 5
- 2.2 Graph theory . . . 6
  - 2.2.1 Standard graph terminology . . . 6
  - 2.2.2 Phylogenetic networks . . . 8
  - 2.2.3 Graph layouts . . . 9
  - 2.2.4 Cutwidth . . . 10
  - 2.2.5 Scanwidth . . . 11

3 Structural results 13
- 3.1 On the equivalent definitions of scanwidth . . . 13
  - 3.1.1 Canonical tree extensions . . . 13
  - 3.1.2 From canonical tree extension to extension . . . 15
  - 3.1.3 From extension to canonical tree extension . . . 16
- 3.2 Bounds and relations to other parameters . . . 19
  - 3.2.1 Width parameters . . . 19
  - 3.2.2 Reticulation number and level of a network . . . 22
  - 3.2.3 Weak-scanwidth: a lower bound . . . 25
- 3.3 Reduction rules . . . 27
  - 3.3.1 (S-)blocks of a graph . . . 27
  - 3.3.2 Arc contractions . . . 30
  - 3.3.3 Complete decomposition scheme . . . 31
- 3.4 Scanwidth-1 and scanwidth-2 characterizations . . . 34

4 Exact algorithms 35
- 4.1 Brute force solution . . . 35
- 4.2 Recursive algorithm . . . 36
- 4.3 Dynamic programming . . . 40
  - 4.3.1 Basic algorithm . . . 40
  - 4.3.2 Algorithm with component splitting . . . 43
  - 4.3.3 Algorithm for fixed scanwidth . . . 46

5 Heuristics 52
- 5.1 Greedy heuristic . . . 52
- 5.2 Cut-splitting heuristic . . . 54
  - 5.2.1 DAG-cuts . . . 54
  - 5.2.2 Repeated DAG-cut-splitting heuristic . . . 57

5.3 Simulated annealing 59
5.3.1 Neighbourhood of (tree) extensions 59
5.3.2 Description of algorithm 62
5.3.3 Asymptotic convergence and cooling schedule 63

6 Experimental results **66**
6.1 Network generation 66
6.2 Reductions 67
6.3 Exact algorithms 68
6.4 Heuristics 69

7 Conclusion and outlook **71**
7.1 Conclusion 71
7.2 Further research 72

References **74**

A Integer linear program **78**

B Appendix to the experimental study **80**
B.1 Parameter tuning for simulated annealing 80
B.2 Table with results for the real networks 81

C Omitted proofs **83**

Chapter 1

Introduction

1.1. Phylogenetics

In the early 1800s, biologists began contemplating the idea of evolution and its depiction by means of a tree (see [Arc14] for an overview). However, it was not until 1859 that Charles Darwin popularized the concept of evolution among the scientific community with his fabled work 'On the Origin of Species'. A famous quote in this manuscript perfectly captures the idea of such a 'tree of life':

> "The affinities of all the beings of the same class have sometimes been represented by a great tree. I believe this simile largely speaks the truth. The green and budding twigs may represent existing species; and those produced during each former year may represent the long succession of extinct species." [Dar59, p. 129]

It speaks to the imagination that the only illustration in Darwin's 502 pages long book, was that of such an evolutionary tree. Although not the first, this diagram can be considered a very early example of a phylogenetic tree.

Phylogenetic trees are important structures in phylogenetics, the study of evolutionary relationships among different genes, species, or populations. These trees capture the idea of a common ancestor that diverges into different species, as time progresses. To depict more complex evolutionary scenarios, one needs phylogenetic networks. Such networks are not completely bifurcating but also contain reticulate events: scenarios where two lineages converge again and combine genetic material to create a new (variant of a) species. Biologically speaking, this models concepts such as hybridization, introgression, horizontal gene transfer, and recombination. An example of a phylogenetic network for some species of cattle is shown in Figure 1.1.

From a mathematical point of view, phylogenetic networks can be represented by directed acyclic graphs with a single root. The leaves of such a graph represent the studied set of species (sometimes referred to as *taxa*), whereas the root is their most recent 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).

1.2. Scanning a network

Phylogenetics is a research area with many computationally hard problems. Current research includes but is not limited to NETWORK INFERENCE [Rab+21], TREE CONTAINMENT [IJW23],

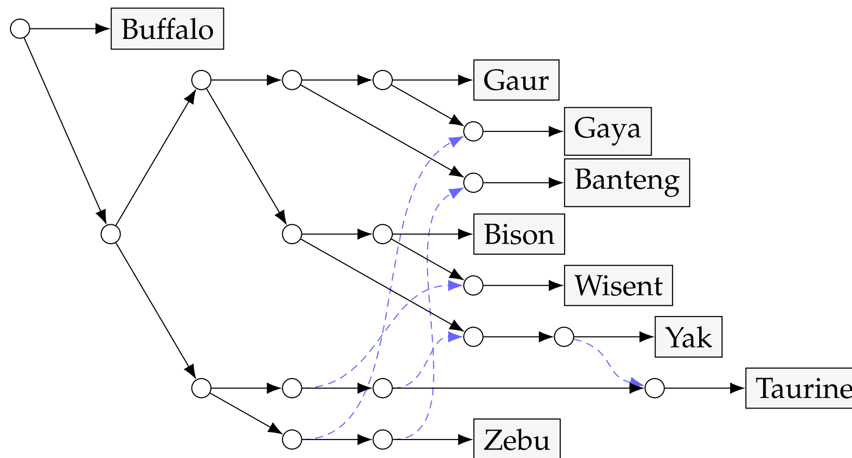
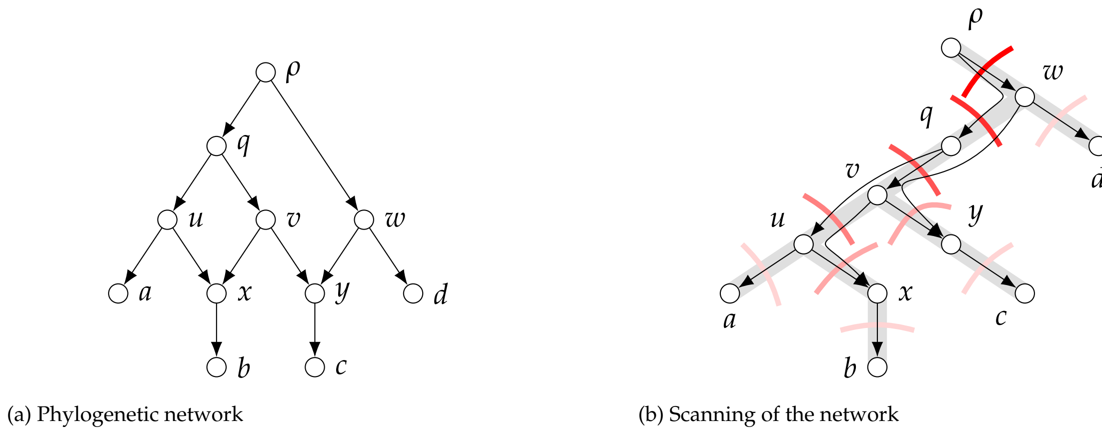


Figure 1.1: Phylogenetic network for nine species of the *Bos* genus, which contains wild and domesticated cattle. The dashed blue arcs depict the transfer of genetic material from one species to another, making this a phylogenetic network and not a phylogenetic tree. This network is a result of a study by Wu et al. [Wu+18].

SMALL PARSIMONY [SW22] and HYBRIDIZATION NUMBER [Ber+23]. A common way to create algorithms to solve these problems is by exploiting 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 are thus still somewhat 'tree-like'. The most well-known way to measure this tree-likeness is through the *treewidth*. The treewidth comes with a corresponding tree decomposition, which represents the graph in a tree-like way (see e.g. [Die17]). This decomposition can be used to guide an algorithm through the graph in an efficient way, while simultaneously bounding its running time. Treewidth has already successfully been used in phylogenetics [IJW23; SW22]. One major caveat of this approach, however, is that treewidth does not consider the directions of the arcs. To this end, Berry, Scornavacca and Weller [BSW20] recently developed an alternative measure for tree-likeness: *scanwidth*.

Contrary to treewidth, scanwidth is not agnostic to the directions of the arcs. Thus, it allows for a more intuitive algorithmic design in phylogenetics. Promising results regarding the use of scanwidth in algorithms have already appeared [Rab+21]. Berry, Scornavacca and Weller named scanwidth after the informal concept of 'scanning' a network. Imagine a 'scanner line' for each leaf of a network. These scanner lines then scan the arcs as they move up through the network. 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 network, with the constraint that it maintains the natural ordering of the network. 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 then referred to as the scanwidth of the network. Figure 1.2 provides an illustration of the concept of scanning a network.

Apart from an NP-completeness proof for the decision problem and some preliminary results in [BSW20], scanwidth has only very briefly been mentioned in [Rab+21] and [SW22]. Magne et al. [Mag+21] 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 extensively researched *directed cutwidth*. It is more restrictive than scanwidth since it only allows linear orderings instead of tree extensions. Consequently, most results for directed cutwidth are not



(a) Phylogenetic network (b) Scanning of the network

Figure 1.2: A phylogenetic network (a), and a tree extension of the network (b), functioning as a route for the scanner lines. The tree extension is indicated by the grey edges, while the network arcs 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 network is 3: the maximum number of network arcs that are cut by one of the scanner lines.

immediately transferable to scanwidth.

1.3. Main contributions and thesis outline

This thesis aims to widen the knowledge of the parameter scanwidth, with a focus on both theoretical results as well as exact and heuristic algorithms to compute it. The tree extensions returned by these algorithms can then be used by future algorithms relying on scanwidth as a parameter.

Chapter 2 covers some preliminary graph theory and complexity theory, as well as the formal introduction to scanwidth. In Chapter 3 we focus on structural results regarding scanwidth. We prove that the scanwidth is bounded from below by the treewidth and from above by the directed cutwidth. Although these two bounds were already mentioned in [BSW20], they had not yet been formally proved. Additionally, we give an example showing that in general, the scanwidth is incomparable to the pathwidth. We also generalize the result from [Rab+21] that the scanwidth of a binary level-k network is at most $k+1$ to non-binary networks, where the level is a well-known measure of tree-likeness for phylogenetic networks. Furthermore, we provide characterizations of the graphs with scanwidth 1 and 2 that have a unique root. Lastly, the chapter discusses some reduction rules to decrease the size of instances.

Chapter 4 is solely focused on exact algorithms that compute the scanwidth. A naive approach to find the scanwidth would run in $O(n! \cdot n \cdot m)$. We are able to improve upon this by providing different exact algorithms, culminating in an algorithm with a slicewise polynomial running time of $O(k \cdot m \cdot n^k)$ for rooted DAGs, with k the scanwidth. This proves that the fixed parameter version of the scanwidth problem for rooted DAGs falls within the complexity class XP, containing problems solvable in $n^{f(k)}$ 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.

Chapter 5 contains different heuristics that are explored, with a cut-splitting heuristic showing quite good results. This heuristic is enhanced by simulated annealing on a suitably defined neighbourhood, allowing for very fast computation. Although the heuristic is

proved to be non-optimal in general, computational experiments in Chapter 6 are promising, showing an average practical approximation ratio of 1.5 for networks of 30 reticulations and 100 leaves. For a dataset of real-world networks, the heuristic even performs optimally in all but one instance. 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.

The thesis is concluded in Chapter 7, which also contains an outlook on further research.

Chapter 2

Preliminaries

This chapter covers some necessary theory and notational conventions for this thesis. In Section 2.1 we briefly touch upon the topic of (parametrized) complexity theory. If the reader is familiar with this topic, this section may be skipped. Section 2.2 contains graph theoretical preliminaries, including the definition of scanwidth.

2.1. Complexity theory

Complexity theory is the study of the computational complexity of algorithms and optimization problems. We provide a short overview of the concepts that are relevant to this thesis. Rather than overwhelming the reader with formal definitions, we aim to present the main ideas in an intuitive manner. For a more detailed and mathematically rigorous description of these topics, see for example [Sip13] for standard complexity theory, and [DF13] for parametrized complexity.

The most common way to formally describe how long an algorithm runs is by means of the *big-Oh notation*. This notation is used to express the asymptotic behaviour of an algorithm's running time as the instance size grows. For two (possibly multivariate) functions $f(x)$ and $g(x)$ one writes $f(x) = O(g(x))$, if there exist values x_0 and $c > 0$ such that $f(x) \leq c \cdot g(x)$ for all $x \geq x_0$. Essentially, this means that in the limit the function f grows no faster than the function g, up to a constant factor c.

Commonly, the size of a problem instance is denoted by n. The above notation then gives rise to *linear time* algorithms that run in $O(n)$ time, *quadratic time* algorithms that run in $O(n^2)$ time, or *exponential time* algorithms running in, for example, $O(2^n \cdot n^3)$ time. This is referred to as the *time complexity* of an algorithm. Although other variations exist, we exclusively consider the so-called *worst-case time complexity*, which considers the worst possible time complexity an algorithm can have. Sometimes, we write $\tilde{O}$ to suppress the polynomial and logarithmic factors in the time complexity, resulting in, for example, $O(2^n \cdot n^3 \cdot \log n) = \tilde{O}(2^n)$.

When considering the complexity of a computational problem, it is common to consider the *decision version* of the problem. Then, instead of aiming to minimize or maximize a quantity, we are just interested in whether the optimum is below or above a certain value. The computational complexity of decision problems is typically described by their *complexity class*. We will informally describe some of the most important while avoiding the technicalities of the definitions.

The class P contains all problems that can be solved in polynomial time, while the 'harder' class NP contains problems that are only verifiable (i.e. a given solution can be verified) in polynomial time. It is widely believed that these two classes do not coincide, meaning that

there exist problems in NP that are not efficiently solvable. A problem is NP-*hard* if any NP problem can be reduced to it in polynomial time. Thus, this class captures problems that are at least as hard as all other NP problems. If an NP-hard problem is still within the class NP, it is called NP-*complete*. Therefore, this class contains the 'hardest' problems that are in NP.

For *parametrized complexity*, we do not only consider the size of the instance but also take into account some other parameter of the problem that can influence its complexity. This research area also contains a fair amount of complexity classes. We will describe two common ones that appear in this thesis.

The class XP contains problems that can be solved by a *slicewise polynomial* algorithm. Such XP algorithms take polynomial time when we fix the parameter of interest k. In other words, these algorithms run in $n^{f(k)}$ time for some computable function f. Hence, the term 'slicewise' is used: the running time is polynomial for each 'slice' of k, although the degree may depend on the value that k takes. This is not allowed for the class of *fixed parameter tractable* (FPT) problems. For a problem to be in FPT, it should be solvable by an algorithm that takes polynomial time for fixed k, but the degree of the polynomial must be the same for each k. Thus, such algorithms run in $n^c \cdot f(k)$ time for some computable function f and a constant c. Clearly, any problem that is in FPT, is also in XP.

Additionally, when analyzing algorithms, it is important to consider the *space complexity* of an algorithm. Then, instead of focusing on running times, one examines the amount of (computer) memory required to execute an algorithm.

Lastly, it is worth noting that when solving problems concerning graphs, we normally measure the size of an instance by the number of vertices n and the number of edges or arcs m.

2.2. Graph theory

In Subsection 2.2.1 we cover some standard graph theoretical concepts and notation, most of which the reader may already know. Subsection 2.2.2 formally introduces phylogenetic networks, and Subsection 2.2.3 handles the notion of graph layouts. We are then ready to define the well-known cutwidth in Subsection 2.2.4, while Subsection 2.2.5 is reserved for the formal introduction of scanwidth.

2.2.1. Standard graph terminology

In this thesis, we mostly follow the standard graph theoretical notation as presented in [Die17]. Below we give an overview of all the relevant terminology.

Undirected graphs

An *(undirected) graph* is a pair $G = (V, E)$, where V is the set of *vertices* (or *nodes*) and $E \subseteq V \times V$ the set of *edges*. In general, we refer to the vertices (resp. edges) of a graph H as $V(H)$ (resp. $E(H)$). Unless explicitly stated otherwise, we will consider our graphs to have no self-loops (i.e. edges from u to u) and not to be *multigraphs* (i.e. graphs with multiple edges that have the same endpoints). A graph is *weighted* if each edge e has an assigned value $w(e)$, and we then write $G = (V, E, w)$.

An edge $e \in E$ between two vertices $u, v \in V$ is denoted by $\{u, v\}$ or simply uv. In this case, u and v are *adjacent*, and both serve as *endpoints* of e. The *degree* of a vertex v, denoted as $\delta(v)$, refers to the number of edges in E that have v as one of their endpoints. Similarly, for a set $W \subseteq V$, we write $\delta(W)$ for the degree of W, which counts the number of edges with one endpoint in W and one endpoint in $V \setminus W$.

A graph $H = (V', E')$ is a *subgraph* of G (or equivalently, G *contains* H) if $V' \subseteq V$ and $E' \subseteq E$. If E' contains all edges $uv \in E$ with $u, v \in V'$, we say that H is an *induced subgraph*

of G and write $H = G[V']$. The notation $G - F$, for some set $F \subseteq E$, represents the graph $(V, E \setminus F)$. Similarly, we sometimes write $G - W$ to indicate the graph $G[V \setminus W]$, for a vertex set $W \subseteq V$.

A *path* in a graph is a sequence of distinct vertices such that consecutive vertices in the sequence are adjacent in the graph. Two vertices u and v of a graph are *connected*, if there exists a path between them. For a graph G, we then write $u \overset{G}{\leftrightsquigarrow} v$. A graph is considered to be connected if there exists a path between any two vertices.

If the first and last vertex in a path coincide and the path has a length of at least 4 (i.e. it contains at least 3 distinct vertices), we have a *cycle*. A graph that contains no cycles is called a *forest*, and if it is also connected a *tree*. A connected graph in which each edge is part of at most one cycle is known as a *cactus (graph)*. A *(connected) component* of a graph is a maximal connected induced subgraph. A *cut vertex* is a vertex whose removal increases the number of connected components. A *block* (or 'biconnected component') is a maximal connected induced subgraph without any cut vertices.

Directed graphs

If we assign the edges in a graph $G = (V, E)$ directions, we speak of a *directed graph*. In this case, each edge $(u, v) \in E$ is directed from its *tail* u to its *head* v, and we use the term *arc* to describe such edges. The *underlying undirected graph* of G is the undirected graph $\tilde{G} = (V, E)$, where we disregard the directions.

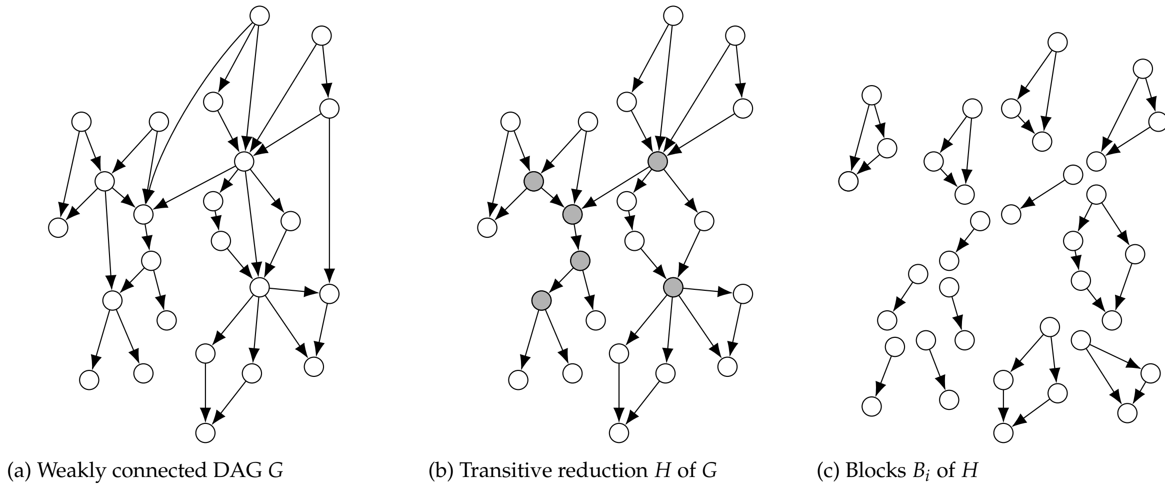
The *indegree* (resp. *outdegree*) of a vertex v is the number of arcs with v as its head (resp. tail), and we denote it by $\delta^{\text{in}}(v)$ (resp. $\delta^{\text{out}}(v)$). Likewise, for each set $W \subseteq V$, the number of arcs with its head (resp. tail) in W and its tail (resp. head) in $V \setminus W$ is called the indegree (resp. outdegree) of W, and this number is denoted by $\delta^{\text{in}}(W)$ (resp. $\delta^{\text{out}}(W)$).

We define a *directed path* (resp. *directed cycle*) as a path (resp. cycle) where all the arcs are directed in the direction of the path (resp. cycle). A directed graph G is said to be *strongly connected* if there exists a directed path in both directions between any two distinct vertices. This is in contrast with a *weakly connected* directed graph: a graph whose underlying undirected graph is connected. (As an example, consider the graph G in Figure 2.1a which is weakly connected but not strongly 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.

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 directed acyclic graph (which will be introduced shortly) 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. For an example of cut vertices, blocks, and the transitive reduction of a graph, we refer to Figure 2.1.

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 labelled 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 $\delta^{\text{out}}(W) = 0$, and we then write $W \sqsubseteq V$. Sometimes, we extend this notation to $W \sqsubseteq U$ for



(a) Weakly connected DAG G (b) Transitive reduction H of G (c) Blocks B_i of H

Figure 2.1: A multi-rooted, weakly connected, directed acyclic graph G (a), the transitive reduction H of G with its cut vertices coloured in grey (b), and the blocks B_i of the graph H (c). H is also a multi-rooted directed cactus.

some $U \subseteq V$. This means that both $W \subseteq U$ and W is a sinkset. If the underlying undirected graph of a DAG G is a tree, we call G a *directed tree*. Similarly, G is a *directed cactus (graph)* if its underlying undirected graph is a cactus. The DAG H in Figure 2.1b is an example of a multi-rooted directed cactus.

Since a DAG contains no directed cycles, it naturally exhibits a top-to-bottom structure. More formally, it defines a partial order[1] on its vertices: we write $v <_G u$ if there exists a directed path from u to v.

Unless otherwise specified, this thesis will consider each graph G to be a weakly connected, directed acyclic graph. If it is clear from the context, we sometimes drop the adjective 'directed' from the notions defined above.

2.2.2. 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. That is,

$$r(G) = \sum_{v \in V : \delta^{\text{in}}(v) \geq 2} (\delta^{\text{in}}(v) - 1).$$

It follows that for a binary network, the reticulation number equals the number of reticulations. 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 [Ier+10].

N is a *phylogenetic network* on a set of labels X, if N is a network and its leaves are bijectively labelled by the elements of X. As an example, consider the phylogenetic network on a set of nine species of cattle from Figure 1.1. Similarly, the graph from Figure 1.2a serves as a phylogenetic network on the set of labels $\{a, b, c, d\}$.

[1] A partial order on a set defines for pairs of elements whether one precedes the other. In contrast with a total order, elements are allowed to be incomparable.

2.2.3. Graph layouts

A *(linear) layout* σ (also known as a 'linear ordering', 'linear arrangement', 'numbering' or 'labelling') 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 layout, the arcs point away from the root.

Figure 2.2 serves as a visualization of the above concepts. In Figure 2.2b an extension is drawn, while Figure 2.2c depicts a tree extension. As a convention, we will always draw (tree) extensions as presented in this figure. To avoid confusion, Figure 2.2d shows a graph H that may mistakenly be interpreted as a tree extension. Although all arcs seem to point downwards, the red arc 'crosses' two branches of the tree H. This violates the constraint that the tree H must be G-respecting. When disregarding the directions of the arcs, the same figure also serves as an illustration of (tree) layouts.

Note that extensions of a DAG G also function as tree extensions. This is true, because an extension σ of G is by definition G-respecting, and its total order has a unique largest element which corresponds to the last vertex in the sequence. Furthermore, any $u, v \in V(G)$ are comparable in σ (i.e. $u <_\sigma v$ or $u >_\sigma v$).

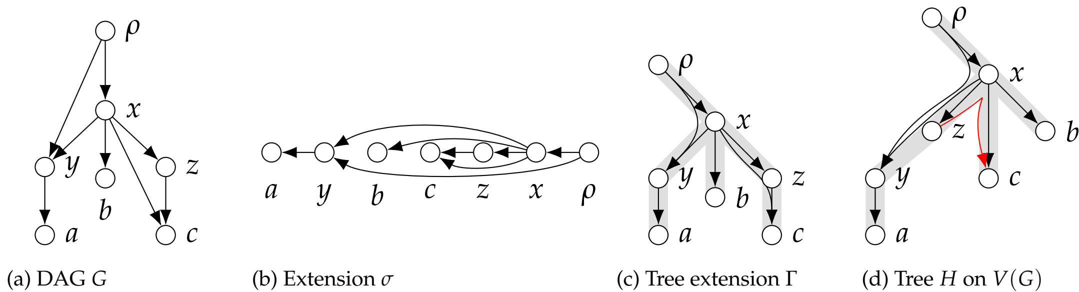


Figure 2.2: (a): Weakly connected DAG G. (b): An extension σ of G with the arcs of G also drawn. (c): A tree extension Γ of G indicated by the grey arcs, whose direction is downwards. The arcs of G are also drawn in Γ and are made to follow the grey arcs. (d): A tree H on $V(G)$ that is not a tree extension, because z and c are not comparable in H, while they are adjacent in G. Visually this means that the corresponding red arc zc of G 'crosses' two branches of the tree.

We will now cover some notation, partially adopted from [BSW20], on the above notions. Throughout this thesis, 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 and $U \subseteq V(G)$, we use $\Pi[U]$ to denote the set of extensions of $G[U]$. Thus, $\Pi[V(G)]$ is the set of all extensions of G. Similarly, we use $\mathcal{T}[U]$ to denote the set of tree extensions of $G[U]$. If G is undirected, we instead mean the sets of linear layouts (resp. tree layouts).

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 $\sigma[i \ldots j]$, while the order starting at i till the last vertex is $\sigma[i \ldots]$. 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 sometimes use vertices to represent a position. For example, $\sigma[v \ldots]$ denotes the suborder of σ starting at the vertex v, where we should have written $\sigma[\sigma^{-1}(v) \ldots]$. For a vertex set V, we sometimes use the notation $[V] = \{1, \ldots, |V|\}$ to denote the set of possible positions a vertex can have in a layout. Finally, we mention that subgraphs of the type $G[\sigma[1 \ldots i]]$ will appear throughout this thesis. We often denote such a subgraph as $G[1 \ldots i]$, if the extension σ is clear from the context.

2.2.4. Cutwidth

Following [BSW20], we will first introduce what cutwidth is, hopefully giving the reader a soft landing regarding the more involved scanwidth.

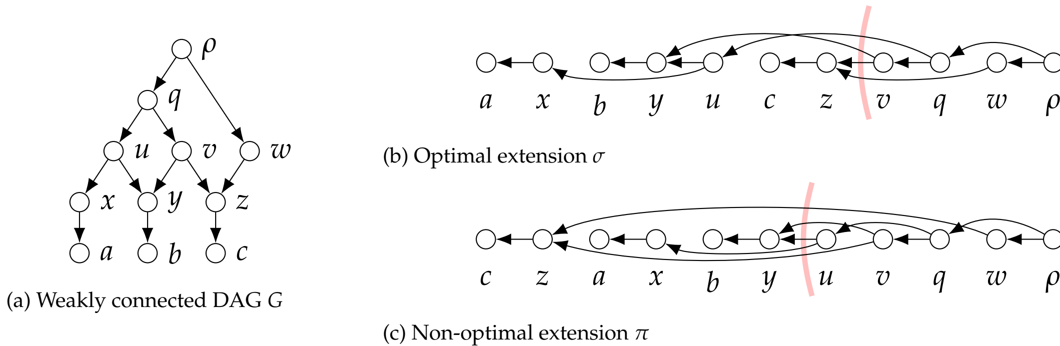
Cutwidth is a width parameter for undirected graphs that has seen a lot of attention since the 1970s (see the survey [DPS02] and its addendum [Pet13]). Multiple variants exist, but we focus on the specific version of the parameter for DAGs. For the sake of brevity, we will refer to it simply as *cutwidth*.[2]

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 $\mathrm{CW}_i^\sigma = \{uv \in E : u \in \sigma[i+1 \ldots], v \in \sigma[1 \ldots i]\}$. Then the* cutwidth *of G is*

$$\mathrm{cw}(G) = \min_{\sigma \in \Pi[V]} \max_{i \in [V]} |\mathrm{CW}_i^\sigma|.$$

Furthermore, we let $\mathrm{cw}(\sigma) = \max_{i \in [V]} \mathrm{cw}_i^\sigma$ be the cutwidth of σ, where $\mathrm{cw}_i^\sigma = |\mathrm{CW}_i^\sigma|$ 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.[3] An example of an optimal and a non-optimal extension in terms of cutwidth is shown in Figure 2.3.



(a) Weakly connected DAG G

(b) Optimal extension σ

(c) Non-optimal extension π

Figure 2.3: (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.

One can compute the cutwidth of a DAG in $\tilde{O}(2^n)$ time [Bod+11]. This exponential time complexity is not surprising, since the corresponding decision problem has been proved to be NP-complete [ES75]. In the context of parametrized complexity for a fixed cutwidth of k, an algorithm was proposed in 1984 for the undirected cutwidth, running in $O(n^k)$ time [GS84].

[2] There is no consensus on the naming of this DAG-variant of cutwidth. In [Bod+11] it is referred to as 'minimum cutwidth for directed acyclic graphs'. Other authors call it 'directed cutwidth' [BFT09; BSW20]. However, sometimes this name is reserved for a different variation on general directed graphs [GR19].

[3] The undirected cutwidth is commonly defined in a similar way, but it minimizes over the space of all linear layouts, instead of extensions.

A few years later, this approach was improved to a time complexity of $O(n^{k-1})$ [MS89]. The theoretically fastest parametrized algorithm for the undirected cutwidth was introduced in [TSB05], and it runs in linear time for fixed k. This implies that the undirected cutwidth can be computed in FPT time. Similarly, it is also known that computing the cutwidth of a DAG is possible in linear FPT time [BFT09].

2.2.5. Scanwidth

In Section 1.2 of the introductory chapter, we provided some intuition on the idea of 'scanning' a phylogenetic network. This concept can be extended to all weakly connected DAGs. Throughout this thesis we aim to provide results for this broader class of graphs.[4] With cutwidth in mind, we are ready to define scanwidth. The scanwidth of a DAG was formally introduced by Berry, Scornavacca and Weller [BSW20] 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 $\mathrm{SW}_i^\sigma = \{uv \in E : u \in \sigma[i+1\dots], v \in \sigma[1\dots i], v \overset{G[1\dots i]}{\leftrightsquigarrow} \sigma(i)\}$. Then the* scanwidth *of G is*

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

Furthermore, we let $\mathrm{sw}(\sigma) = \max_{i \in [V]} \mathrm{sw}_i^\sigma$ be the scanwidth of σ, where $\mathrm{sw}_i^\sigma = |\mathrm{SW}_i^\sigma|$ is the scanwidth of σ at position i.

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 that enter a vertex v that is weakly connected to $\sigma(i)$ in the graph $G[\sigma[1\dots i]]$. Recall from Section 2.2 that we write this as $v \overset{G[1\dots i]}{\leftrightsquigarrow} \sigma(i)$. Before explaining this definition with Figure 2.4, we introduce an alternative characterization of scanwidth.

In [BSW20, Prop. 1] it is shown that the following definition equivalently defines scanwidth.[5] Whereas the previous definition involves extensions, 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 $\mathrm{GW}_v^\Gamma = \{xy \in E : x >_\Gamma v \geq_\Gamma y\}$. Then the* scanwidth *of G is*

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

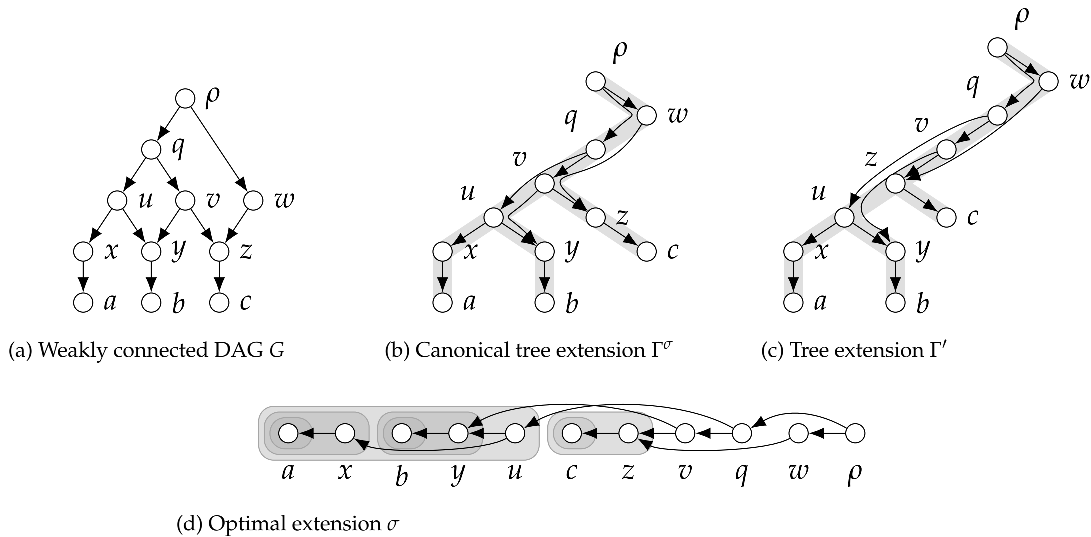
Furthermore, we let $\mathrm{sw}(\Gamma) = \max_{v \in V} \mathrm{sw}_v^\Gamma$ be the scanwidth of Γ, where $\mathrm{sw}_v^\Gamma = |\mathrm{GW}_v^\Gamma|$ is the scanwidth of Γ at vertex v.

We now have two equivalent ways to define scanwidth at hand. Definition 2.3 more closely resembles the intuition of 'scanning', allowing for more straightforward application in a parametrized algorithm. On the other hand, Definition 2.2 will turn out the be more convenient in proofs, as extensions are less complicated mathematical objects than tree extensions. For example, induction-based proofs are often easier when iterating over the positions of an extension.

[4] Most results in this thesis, if not all, generalize to disconnected graphs by considering a *forest layout*, which contains a separate tree layout for each component of the graph. However, we will only consider weakly connected graphs and thus leave this generalization to the reader.

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

To illustrate the two definitions and their relation, we take a look at Figure 2.4. We first explore the tree-based definition, as it is visually more intuitive. Figure 2.4a depicts the same weakly connected DAG as in Figure 2.3, while Figure 2.4b shows a tree extension Γ^σ (the fat grey arcs, whose direction is downwards) with the arcs of the original DAG drawn in it. For each vertex in the graph, the set GW 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 the tree extension Γ^σ equals 3, which is attained at the vertex v, where we have that $\mathrm{GW}_v^{\Gamma^\sigma} = \{qv, qu, wz\}$. It turns out that this is an optimal tree extension for the given graph. In a similar fashion, Figure 2.4c shows a tree extension of the same graph, with a non-optimal scanwidth of 4, attained at the vertex z.



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

(d) Optimal extension σ

Figure 2.4: (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 4, attained at the vertex z. For each $i \leq 7$, the outermost grey shaded areas containing only vertices belonging to $\sigma[1 \ldots i]$ depict the weakly connected components of $G[1 \ldots i]$. For $i > 7$, $G[1 \ldots i]$ is weakly connected and therefore consists of just one component.

We have already stated that the scanwidth of the graph G in Figure 2.4a equals 3. This must mean that there also exists an extension with a scanwidth of 3 when considering Definition 2.2. One such extension σ is shown in Figure 2.4d. Contrary to tree extensions, arcs in the sets SW are a bit more involved to calculate. It requires knowledge of the weak connectivity relations within subgraphs of the graph. In Figure 2.4d 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, $\mathrm{SW}_z^\sigma = \{vz, wz\}$, 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 Figure 2.3b).

Chapter 3

Structural results

In this chapter we present new theoretical results regarding scanwidth. In Section 3.1 we take a more in-depth look into the two equivalent definitions of scanwidth. Section 3.2 will focus on bounding the scanwidth of a DAG by other (width) parameters. Section 3.3 is aimed at the creation of reduction rules for the scanwidth problem, which will prove valuable in subsequent chapters. Lastly, we characterize the rooted DAGs of scanwidth 1 and 2 in Section 3.4.

3.1. On the equivalent definitions of scanwidth

This section is divided into three subsections. After exploring the relation between the two equivalent scanwidth definitions in Subsection 3.1.1, we will discuss how to move between the two definitions in Subsections 3.1.2 and 3.1.3.

3.1.1. Canonical tree extensions

Upon further examination of the example in Figure 2.4, an interesting observation emerges. We notice 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 [BSW20] used to prove the equivalence between the two scanwidth definitions. Formally, it is defined as follows:

Definition 3.1 (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*

$$H = (V, \{uv : u >_\sigma v, u \overset{G[1\ldots u]}{\leftrightsquigarrow} v\}).$$

Lemma 3.2, which is proved in [BSW20], establishes the relation between canonical tree extensions and extensions. It states that Γ^σ is indeed a tree extension of G (by b) that has the same scanwidth as σ (by a and c). Moreover, any extension of the canonical tree extension has the same scanwidth again (by c).

Lemma 3.2 (Berry, Scornavacca and Weller [BSW20, Lem. 5])**.** *Let $G = (V, E)$ be a weakly connected DAG, σ an extension of G, and Γ^σ the canonical tree extension for σ. Then,*

(a) σ is an extension of Γ^σ;

(b) Γ^σ is a tree extension of G;

(c) for each extension π of Γ^σ, and for each $v \in V$, we have $\mathrm{SW}_v^\pi = \mathrm{GW}_v^{\Gamma^\sigma}$;

(d) for each $v \in V$, the subgraph $G[V(\Gamma_v^\sigma)]$ is a weakly connected component of $G - \mathrm{GW}_v^{\Gamma^\sigma}$, where Γ_v^σ is the subtree of Γ^σ rooted at v.

By the lemma, we are able to construct for each extension, a tree extension with the same scanwidth at every vertex. For canonical tree extensions, the converse is also true. However, this converse does not need to hold in general. As an example, the tree extension Γ' in Figure 2.4c is not canonical (which will be proved later), and indeed no extension of Γ' has the property that $\mathrm{SW}_z^\sigma = \mathrm{GW}_z^{\Gamma'}$.

As a side result of the lemma, we can now easily show that there always exists a canonical tree extension that is optimal. This implies that it is sufficient to minimize over all canonical tree extensions when computing the scanwidth.

Corollary 3.3. *Let $G = (V, E)$ be a weakly connected DAG, then there exists a canonical tree extension Γ such that* $\mathrm{sw}(G) = \mathrm{sw}(\Gamma)$.

Proof. Let σ be an optimal extension with respect to the scanwidth. By Lemma 3.2a, σ is an extension of the canonical tree extension Γ^σ. According to Lemma 3.2c, we then have that $\mathrm{sw}(\Gamma^\sigma) = \mathrm{sw}(\sigma)$. Since σ was optimal, we have that $\mathrm{sw}(G) = \mathrm{sw}(\sigma) = \mathrm{sw}(\Gamma^\sigma)$. This proves that the canonical tree extension Γ^σ is optimal. □

Lemma 3.2d gives a necessary condition for a tree extension to be canonical. The previous corollary raises the question whether a sufficient condition exists. As the definition of canonical tree extensions is rather involved, an easy-to-check condition could be helpful in algorithmic design.

It turns out that it is indeed possible to completely characterize the canonical tree extensions by a quickly checkable condition. We first need the following uniqueness result which proves that the sets GW are enough to uniquely describe a tree extension.

Lemma 3.4. *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 $\Gamma \neq \Omega$, there must exist some $u \in V$ such that $V(\Gamma_u) \neq V(\Omega_u)$. Clearly, the sets GW_u contain exactly the arcs that enter the above two vertex sets. In other words, $\mathrm{GW}_u^\Gamma = \{xy \in E(G) : x \notin V(\Gamma_u), y \in V(\Gamma_u)\}$ and $\mathrm{GW}_u^\Omega = \{xy \in E(G) : x \notin V(\Omega_u), y \in V(\Omega_u)\}$. But since $V(\Gamma_u) \neq V(\Omega_u)$, we must then have $\mathrm{GW}_u^\Gamma \neq \mathrm{GW}_u^\Omega$. This proves the lemma, and consequently, the fact that the sets GW uniquely determine a tree extension. □

With this lemma at our disposal, we are now ready to characterize the canonical tree extensions in the following proposition.

Proposition 3.5. *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. ($\Rightarrow$) Let Γ be the canonical tree extension for σ, i.e. $\Gamma = \Gamma^\sigma$. From Lemma 3.2d we know that for each $v \in V$, it holds that $G[V(\Gamma_v)]$ is a weakly connected component of $G - \mathrm{GW}_v^{\Gamma^\sigma}$. Thus, $G[V(\Gamma_v)]$ is weakly connected for all $v \in V$.

($\Leftarrow$) Let Γ be a tree extension that is not canonical for σ, i.e. $\Gamma \neq \Gamma^\sigma$. 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 $y \overset{G[1...v]}{\leftrightsquigarrow} v$. As σ is an extension of Γ, together this gives that $v \geq_\Gamma y$. Using that Γ is a tree extension and that xy is an arc of G, we also have that $x >_\Gamma y$. Combining with the fact that $x >_\sigma v$, we must then have $x >_\Gamma v \geq_\Gamma y$. This means that $xy \in \mathrm{GW}_v^\Gamma$. So, $\mathrm{SW}_v^\sigma \subseteq \mathrm{GW}_v^\Gamma$.
>
> According to Lemma 3.4, there exists some $v \in V$ such that $\mathrm{GW}_v^\Gamma \neq \mathrm{GW}_v^{\Gamma^\sigma}$. By Lemma 3.2c, 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 $\mathrm{SW}_v^\sigma \subset \mathrm{GW}_v^\Gamma$. △

Let v be as in the claim. By the claim, 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. □

As Lemma 3.2a tells us that a canonical tree extension can only be canonical for an extension of itself, the previous proposition gives us an easy-to-check characterization of the canonical tree extensions. We only need to check whether the induced subgraphs corresponding to the vertices of each subtree of the tree extension are weakly connected. Recalling the examples from Figure 2.4, we can quickly see that Γ^σ in Figure 2.4b is indeed canonical for σ, since σ is an extension of Γ^σ, and since the above connectivity condition holds. On the other hand, the tree extension Γ' from Figure 2.4c can now be shown to not be canonical for any extension of Γ' (and consequently it is not canonical for any extension of the graph), since $G[V(\Gamma'_z)]$ is not weakly connected.

3.1.2. From canonical tree extension to extension

Depending on the application, one might be interested in an extension, while only a tree extension is at hand. If this tree extension is canonical, we can find an extension with the same scanwidth. Algorithm 1 accomplishes this.

Algorithm 1: Verify whether a tree extension is canonical, and if it is, create an extension with the same scanwidth.

Input: Weakly connected DAG $G = (V, E)$, tree extension Γ of G.
Output: If Γ is canonical, an extension σ such that $\mathrm{sw}(\sigma) = \mathrm{sw}(\Gamma)$. Else, `None`.

```
1 for each v ∈ V do
2     Γ_v ← the subtree of Γ rooted at v
3     if G[V(Γ_v)] is not weakly connected then
4         return None
5 σ ← reverse order of a BFS traversal of Γ
6 return σ
```

The workings of the algorithm are fairly obvious. First, it uses Proposition 3.5 to check if the tree extension is canonical. If it is, it applies a *breadth-first search* (BFS) to get an extension of the tree extension. We summarize this, together with the algorithm's time complexity, in the following theorem.

Theorem 3.6. *Let $G = (V, E)$ be a weakly connected DAG of n vertices and m arcs, and let Γ be a tree extension of G. Then, Algorithm 1 verifies whether Γ is canonical in $O(n \cdot m)$ time. If this is the case, the algorithm returns an extension σ such that $\text{sw}(\sigma) = \text{sw}(\Gamma)$ in $O(n)$ time.*

Proof. Correctness: By Lemma 3.2a, a tree extension can only be canonical for an extension σ of G if σ is also an extension of Γ. So, to determine whether Γ is canonical, it suffices to know whether Γ is canonical for some extension of Γ. According to Proposition 3.5, this is equivalent to the fact that for all $v \in V$, the subgraph $G[V(\Gamma_v)]$ is weakly connected. This proves that the algorithm returns `None`, if and only if Γ is not canonical.

If Γ is canonical, Lemma 3.2c tells us that each extension σ of Γ has the property that $\text{SW}_v^\sigma = \text{GW}_v^\Gamma$. Thus, $\text{sw}(\sigma) = \text{sw}(\Gamma)$. Clearly, a BFS traversal of Γ parses the vertices in some topological ordering of Γ, if started at the root of Γ. But then, reversing this ordering gives an extension of Γ, which proves correctness.

Time complexity: A BFS parses each arc and node of a graph once. Because Γ has n nodes and $n-1$ arcs, a BFS of Γ takes $O(n)$ time. For each vertex v, we can create the set of descendants in Γ by doing a BFS of Γ in $O(n)$ time. Creating the subgraph $G[\Gamma_v]$ then takes at most $O(n+m)$ time, while checking whether it is connected can be done by a BFS on this subgraph in at most $O(n+m)$ time. This process is repeated for each vertex, resulting in a time complexity of $O(n \cdot m)$ to check whether Γ is canonical.

Creating the extension by a BFS then takes $O(n)$ time. □

Note that if we already know that Γ is canonical, we can reduce the time complexity to just $O(n)$. Furthermore, alternative traversal algorithms such as *depth-first search* (DFS) can be employed in the algorithm, as long as they guarantee a linear time complexity.

3.1.3. From extension to canonical tree extension

It is not entirely straightforward from Definition 3.1 how to create Γ^σ from an extension σ. A naive approach would be to first create the graph H appearing in this definition. To this end, we need the connectivity relations of $G[1 \ldots v]$ for all $v \in V$. It takes $O(n+m)$ time to create such a subgraph, and one can use a BFS in $O(n+m)$ time to find the weakly connected components within it. Then, checking whether two vertices are weakly connected, amounts to checking if they are in the same component. We repeat this n times, and thus it would take us $O(n^2 + nm)$ time to create H. Finding the transitive reduction of H can be implemented in $O(n^{2.37188})$ time[1], resulting in an overall time complexity of $O(n^{2.37188} + nm)$ to create the canonical tree extension from an extension.

We opt for a structured and more intuitive approach that slowly builds up the tree extension from its leaves to the root. As a nice side effect, it runs faster than the above-described solution would, having a time complexity of $O(n^2)$. Our algorithm relies on Proposition 3.5, which implies that creating the canonical tree extension amounts to creating a tree extension Γ of G, such that σ is an extension of Γ and $G[\Gamma_v]$ is weakly connected for all $v \in V(G)$. The method is formally described in Algorithm 2.

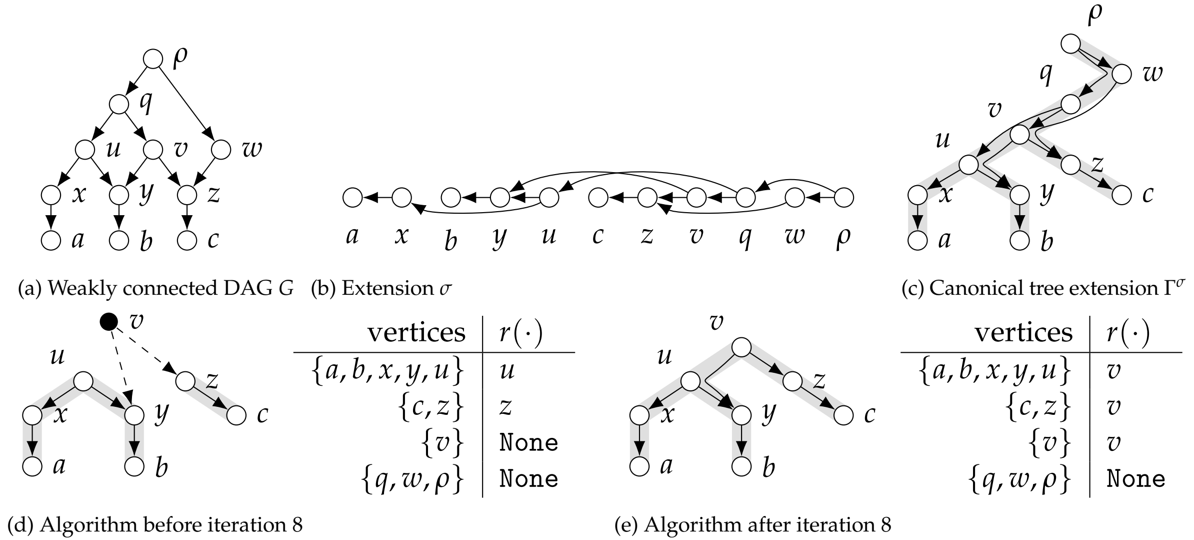
[1] This follows from a result in [AGU72], which states that finding the transitive reduction is equivalent to matrix multiplication. Currently, the best algorithm for matrix multiplication runs in $O(n^{2.37188})$, although it has an extremely large hidden constant [DWZ23].

Algorithm 2: Create the canonical tree extension Γ^σ for an extension σ.

Input: Weakly connected DAG $G = (V, E)$, extension σ of G.
Output: Canonical tree extension Γ^σ corresponding to σ.

1 **initialize**
2 $\quad r(v) \leftarrow \texttt{None}, \forall v \in V(G)$
3 $\quad \Gamma^\sigma \leftarrow$ empty directed tree
4 **for each** $i \in \{1, \ldots, |V|\}$ **do**
5 $\quad v \leftarrow \sigma(i)$
6 $\quad C \leftarrow$ set of children of v in G
7 $\quad R \leftarrow \{r(c) : c \in C\}$
8 $\quad$ add the vertex v to Γ^σ
9 $\quad$ add an arc vw to $\Gamma^\sigma, \forall w \in R$
10 $\quad$ **for each** $u \in V(\Gamma^\sigma)$ **do**
11 $\quad\quad$ **if** $r(u) \in R$ **then**
12 $\quad\quad\quad r(u) \leftarrow v$
13 $\quad r(v) \leftarrow v$
14 **return** Γ^σ

In the algorithm we iterate over the vertices in the extension σ. For each vertex, we keep track of the root-vertex of its component (in the thus far constructed tree extension) by means of the mapping r. We then add the vertices one by one, each time creating arcs to the root vertices of the components that contain the children of the current vertex in G. Then, we update the r-assignment of the vertices in those components. Figure 3.1 shows an example run of an iteration of this algorithm applied to the graph from Figure 2.4.



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

vertices	$r(\cdot)$
$\{a,b,x,y,u\}$	u
$\{c,z\}$	z
$\{v\}$	None
$\{q,w,\rho\}$	None

(d) Algorithm before iteration 8

vertices	$r(\cdot)$
$\{a,b,x,y,u\}$	v
$\{c,z\}$	v
$\{v\}$	v
$\{q,w,\rho\}$	None

(e) Algorithm after iteration 8

Figure 3.1: Illustration of an iteration of Algorithm 2. (a): Weakly connected DAG G. (b): Extension σ of G. (c): Canonical tree extension Γ^σ of G. (d): Partial tree extension that was built in the first seven iterations of Algorithm 2 applied to the graph G and extension σ. The corresponding r-assignments are also shown. We already drew in the vertex $v = \sigma(8)$ and its corresponding outgoing (dashed) arcs of G. (e): Partial tree extension and new r-assignments after the eighth iteration of Algorithm 2. The children of v in G are y and z, thus $C = \{y, z\}$. Since the roots of the components that contained these vertices in subfigure (d) are in the set $R = \{u, z\}$, we attached v to u and to z. When the algorithm terminates, it will have constructed the canonical tree extension Γ^σ from subfigure (c).

To prove correctness, we first state the following technical lemma about the algorithm. As

the result is rather intuitive, we delay its induction-based proof to Appendix C.

Lemma 3.7. *Let $G = (V, E)$ be a weakly connected DAG, σ an extension of G, and Γ the graph returned by Algorithm 2 applied to σ. Then,*

(a) *Γ is a tree extension of G;*

(b) *σ is an extension of Γ;*

(c) *$G[V(\Gamma_v)]$ is weakly connected for each $v \in V$, where Γ_v is the subtree of Γ rooted at v.*

We are now ready to prove the correctness of the algorithm and show that it runs in quadratic time.

Theorem 3.8. *Let $G = (V, E)$ be a weakly connected DAG of n vertices and m arcs, and let σ be an extension of G. Then, Algorithm 2 applied to σ returns the canonical tree extension Γ^σ in $O(n^2)$ time.*

Proof. *Correctness:* Let Γ be the graph returned from Algorithm 2 applied to σ. The three statements from Lemma 3.7 together are now equivalent to the fact that Γ is the canonical tree extension for σ, according to Proposition 3.5. This proves correctness.

Time complexity: The outer for loop is executed exactly n times. Within each for loop, we first create the set of children of v. This can be done in $O(n)$ time (by saving the graph as an adjacency matrix). The other parts of the loop are also dominated by $O(n)$. The inner for loop is executed at most n times. All in all, we thus have a time complexity of $O(n^2)$. □

For ease of later reference, we can now formulate a corollary stating the time it takes to calculate the scanwidth of a given (tree) extension.

Corollary 3.9. *Let $G = (V, E)$ be a weakly connected DAG of n vertices and m arcs, then the scanwidth of any extension or tree extension of G can be calculated in $O(n \cdot m)$ time.*

Proof. Let Γ be any tree extension of G. With a BFS one can find the partial order $<_\Gamma$ of the vertices in the tree extension. As mentioned in the proof of Theorem 3.6, such a BFS takes $O(n)$ time. For each of the n vertices, we can then find the set GW_v^Γ in $O(m)$ time, by iterating over all m arcs and checking the partial order condition (as defined in Definition 2.3). Taking the maximum takes constant time, so we can calculate $\mathrm{sw}(\Gamma)$ in $O(nm)$ time.

Now let σ be any extension of G. According to Theorem 3.8, we can transform σ in Γ^σ in $O(n^2)$ time. By the first part of the current proof, it takes $O(nm)$ time to calculate the scanwidth of Γ^σ. By Lemma 3.2c this value equals $\mathrm{sw}(\sigma)$, which shows that it takes $O(nm)$ time to calculate $\mathrm{sw}(\sigma)$. □

3.2. Bounds and relations to other parameters

In this section, we will discuss how scanwidth relates to a range of other parameters. Subsection 3.2.1 will focus on width parameters, while Subsection 3.2.2 concentrates on two network-specific notions which are commonly used in the phylogenetic network literature: reticulation number and level. We reserve Subsection 3.2.3 for a lower bound of the scanwidth in terms of a newly defined parameter.

Of course, one can also construct simple, yet useful bounds for the scanwidth. We gather three of them in the following lemma:

Lemma 3.10. *Let $G = (V, E)$ be a weakly connected DAG, then (a) $\mathrm{sw}(G) \geq \max_{v \in V} \delta^{\mathrm{in}}(v)$; (b) $\mathrm{sw}(G) \leq |E|$; (c) $\mathrm{sw}(G) \geq \left\lfloor \frac{|E|}{|V|-1} \right\rfloor$.*

Proof. (a): For every extension σ and $v \in V$, we have by definition that SW^{σ}_{v} contains at least all arcs coming into v. The bound then follows.

(b): For every extension σ and $v \in V$, we have by definition that SW^{σ}_{v} contains at most all arcs of the graph. The bound then follows.

(c): For every extension σ, every arc must be in atleast one set SW^{σ}_{v}. The set $\mathrm{SW}^{\sigma}_{\sigma(|V|)}$ is always empty, since $\sigma(|V|)$ is the last vertex in the extension. Thus, we have at most $|V| - 1$ non-empty sets. Then, there always exists a set SW^{σ}_{v}, such that $|\mathrm{SW}^{\sigma}_{v}| \geq \left\lfloor \frac{|E|}{|V|-1} \right\rfloor$. This proves the bound. □

3.2.1. Width parameters

In this subsection, we will turn our attention to an undirected analogue of scanwidth: edge-treewidth. We also cover the relation of scanwidth to three of the most common width parameters: cutwidth, pathwidth and treewidth.

Edge-treewidth

The first width parameter we will examine is *edge-treewidth*, recently introduced by Magne et al. [Mag+21]. In the introductory chapter, we already mentioned that edge-treewidth can be viewed as the undirected analogue of scanwidth. The following definition is given in [Mag+21], although we adapt the notation to our standard. They also provide an equivalent tree layout definition which resembles Definition 2.3 of the scanwidth.

Definition 3.11 (Edge-treewidth). *Let $G = (V, E)$ be a connected undirected graph. For a linear layout σ and a position i of σ, we will denote $\mathrm{ETW}^{\sigma}_{i} = \{uv \in E : u \in \sigma[i+1\ldots], v \in \sigma[1\ldots i], u \overset{G[1\ldots i]}{\leftrightsquigarrow} v\}$. Then the* edge-treewidth *of G is*

$$\mathrm{etw}(G) = \min_{\sigma \in \Pi[V]} \max_{i \in [V]} |\mathrm{ETW}^{\sigma}_{i}|.$$

Furthermore, we denote $\mathrm{etw}(\sigma) = \max_{i \in [V]} \mathrm{etw}^{\sigma}_{i}$, where $\mathrm{etw}^{\sigma}_{i} = |\mathrm{ETW}^{\sigma}_{i}|$.

Recall, that $\Pi[V]$ denotes the set of all linear layouts of G, since G is an undirected graph. It is not hard to see that for any DAG the edge-treewidth of its underlying undirected graph is at most the scanwidth of the DAG. This follows from the fact that for the scanwidth we minimize over all extensions, which are by definition linear layouts of the underlying undirected graph. We thus get the following lemma:

Lemma 3.12. *Let $G = (V, E)$ be a weakly connected DAG and $\tilde{G}$ its underlying undirected graph, then*

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

Proof. Let σ be an extension of G. Clearly, σ is also a linear layout of $\tilde{G}$. Furthermore, by definition, $\mathrm{ETW}_v^\sigma = \mathrm{SW}_v^\sigma$ for all $v \in V$. But then, $\mathrm{etw}(\sigma) = \mathrm{sw}(\sigma)$. As σ was an arbitrary extension, we thus get $\mathrm{etw}(\tilde{G}) \leq \mathrm{sw}(G)$. □

Although edge-treewidth is extremely similar to scanwidth, most results presented in [Mag+21] are not of immediate use to us. Their results are of pure structural purpose and are not directly aimed at computing the parameter. We will however use one of their results in Section 3.4 to characterize DAGs of scanwidth 1 and 2.

Cutwidth

In Subsection 2.2.4 we covered the well-known cutwidth and already mentioned the strong relation between cutwidth and scanwidth. In fact, if a graph has just one leaf, it has been shown that its scanwidth and its cutwidth coincide [BSW20]. It should come as no surprise that in general the cutwidth (as introduced in Definition 2.1) bounds the scanwidth from above. This obvious fact was already mentioned in [BSW20], but for completeness, we will formally prove it in the next lemma. At the heart of the proof lies the fact that the sets SW are always subsets of the sets CW.

Lemma 3.13. *Let $G = (V, E)$ be a weakly connected DAG, then*

$$\mathrm{sw}(G) \leq \mathrm{cw}(G).$$

Proof. Let σ be an extension of G, and let $v \in V$ be arbitrary. We have that $\mathrm{sw}_v^\sigma = |\mathrm{SW}_v^\sigma| \leq |\mathrm{CW}_v^\sigma| = \mathrm{cw}_v^\sigma$. This holds, since $\mathrm{SW}_v^\sigma \subseteq \mathrm{CW}_v^\sigma$, which follows immediately from the definitions. Now it follows that $\mathrm{sw}(\sigma) = \max_{v \in V} \mathrm{sw}_v^\sigma \leq \max_{v \in V} \mathrm{cw}_v^\sigma = \mathrm{cw}(\sigma)$. Therefore, $\mathrm{sw}(G) = \min_{\sigma \in \Pi[V]} \mathrm{sw}(\sigma) \leq \min_{\sigma \in \Pi[V]} \mathrm{cw}(\sigma) = \mathrm{cw}(G)$. □

Pathwidth

An often-used width parameter for undirected graphs is *pathwidth*. The pathwidth can be viewed as a measure of how closely an undirected graph resembles a path. Typically, the pathwidth is defined by means of a *path-decomposition* (see for example [DPS02]). We will not do so here. In [Kin92] it is shown that the pathwidth of an undirected graph G is equal to its 'vertex separation number'. This provides us with the following equivalent definition of pathwidth:

Definition 3.14 (Pathwidth). *Let $G = (V, E)$ be a connected undirected graph. For a linear layout σ and a vertex v of V, we will denote $\mathrm{PW}_i^\sigma = \{v \in \sigma[i+1\dots] : \exists u \in \sigma[1\dots i] \text{ s.t. } uv \in E\}$. Then the* pathwidth *of G is*

$$\mathrm{pw}(G) = \min_{\sigma \in \Pi[V]} \max_{i \in [V]} |\mathrm{PW}_i^\sigma|.$$

Furthermore, we denote $\mathrm{pw}(\sigma) = \max_{i \in [V]} \mathrm{pw}_i^\sigma$, where $\mathrm{pw}_i^\sigma = |\mathrm{PW}_i^\sigma|$.

Compared to cutwidth, pathwidth differs in two places. First of all, we minimize over all linear layouts of the undirected graph, instead of extensions of a DAG. Secondly, we do not consider the number of edges that cross each gap of the extension. Instead, we only care about the endpoints of these edges that are higher up in the ordering. As an example, we revisit the extension σ in Figure 2.3b. The red cut in this image indicates the set $\mathrm{CW}_z^\sigma = \{qy, vu, vz, wz\}$ for the cutwidth. On the other hand, for the pathwidth of this extension (where we consider the underlying undirected graph of the DAG), we just have $\mathrm{PW}_z^\sigma = \{v, q, w\}$, since those vertices are the endpoints to the right of the cut.

Berry, Scornavacca and Weller [BSW20] do not mention the relation of scanwidth to pathwidth. With two examples, we show that in general, neither of the parameters bounds the other. To this end, consider the graph G in Figure 3.2a. As G is a tree, we can take G as a tree extension. It is immediately clear that we then have that $\mathrm{sw}(G) = 1$. However, the underlying undirected graph $\tilde{G}$ has a pathwidth of 2, which should be clear by inspection.[2] As a consequence, the pathwidth (of the underlying undirected graph) is larger than the scanwidth of this DAG.

Now consider the graph H in Figure 3.2b. For this graph, only one extension is possible: $\sigma = (c, b, a, \rho)$. Then we can just calculate the scanwidth as $\mathrm{sw}(H) = \mathrm{sw}(\sigma) = 3$. This same layout gives a pathwidth of 2 for the underlying undirected graph, and it turns out that this is optimal. Thus, in this case, the pathwidth (of the underlying undirected graph) is smaller than the scanwidth. In general, the scan- and pathwidth are therefore not comparable to each other.

(a) Weakly connected DAG G (b) Weakly connected DAG H

Figure 3.2: (a): Weakly connected DAG G, with $\mathrm{sw}(G) = 1 < 2 = \mathrm{pw}(\tilde{G})$. (b): Weakly connected DAG H, with $\mathrm{sw}(H) = 3 > 2 = \mathrm{pw}(\tilde{H})$.

Pathwidth also has a directed counterpart: *directed pathwidth* [Bar06]. Berry, Scornavacca and Weller [BSW20] argue that this parameter is always 0 for DAGs. They also present *register width*, which can be viewed as a DAG-variant of pathwidth. This parameter is also incomparable to scanwidth by the above counterexamples.

Treewidth

In the introduction, the use of scanwidth as a parameter in algorithms in phylogenetics was motivated by the successful applicability of another tree measure: *treewidth*. This parameter has seen a vast amount of research efforts (see [Bod12] for a survey). A fabled result by Bodlaender [Bod93] is that the treewidth can be calculated in linear FPT time. Although normally defined by a so-called *tree-decomposition* (see e.g. [Die17]), 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 from [SW22].

Definition 3.15 (Treewidth). *Let $G = (V, E)$ be a connected undirected graph. For a tree layout Γ and a vertex v of V, we will denote $\mathrm{TW}_v^\Gamma = \{u \in V : u >_\Gamma v, \exists w \leq_\Gamma v \text{ s.t. } uw \in E\}$. Then the* treewidth *of G is*

$$\mathrm{tw}(G) = \min_{\Gamma \in \mathcal{T}(V)} \max_{v \in V} |\mathrm{TW}_v^\Gamma|.$$

Furthermore, we denote $\mathrm{tw}(\Gamma) = \max_{v \in V} \mathrm{tw}_v^\Gamma$, where $\mathrm{tw}_v^\Gamma = |\mathrm{TW}_v^\Gamma|$.

Recall, that $\mathcal{T}(V)$ is the set of all tree layouts of G, since G is an undirected graph. Essentially, treewidth is to pathwidth, what scanwidth is to cutwidth. Scanwidth extends the search space of cutwidth to tree extensions, while still looking at arc-cuts. Similarly, treewidth

[2]This also follows more formally from [Sch90, Thm. 4]. There it is proved that a tree has pathwidth $\geq p$, if and only if it contains a vertex with three subtrees, each having pathwidth $\geq p - 1$.

extends the search space of pathwidth to tree layouts but still looks at endpoints of edges in the cuts such that the endpoints are higher up in the tree.

To exemplify the relation between treewidth and scanwidth, consider the canonical tree extension Γ^σ in Figure 2.4b, where $\mathrm{GW}_v^{\Gamma^\sigma} = \{wz, qu, qv\}$. For the treewidth, we can disregard the directions, and the set $\mathrm{TW}_v^{\Gamma^\sigma}$ now contains only the endpoints in $\mathrm{GW}_v^{\Gamma^\sigma}$ that are higher up in the tree than v. Therefore, $\mathrm{TW}_v^{\Gamma^\sigma} = \{w, q\}$.

In [BSW20] 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 [BSW20], we feel the need to formally prove the bound in the next lemma.

Lemma 3.16. *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. Clearly, Γ is then also a tree layout for $\tilde{G}$. (Note that the converse does not need to hold.) Let $v \in V$ be arbitrary. By definition, $\mathrm{GW}_v^\Gamma = \{xy \in E(G) : x >_\Gamma v \geq_\Gamma y\}$. We now define the mapping $\phi : \mathrm{GW}_v^\Gamma \to \mathrm{TW}_v^\Gamma$ as $\phi(xy) = x$. We first show that we have specified the correct codomain. In other words, ϕ indeed maps all elements of GW_v^Γ to TW_v^Γ. For any $xy \in \mathrm{GW}_v^\Gamma$, 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 TW_v^Γ.

Secondly, we prove that ϕ is surjective. To this end, let $u \in \mathrm{TW}_v^\Gamma$ 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 $uw \in \mathrm{GW}_v^\Gamma$, and so $\phi(uw) = u$. This shows that ϕ is surjective.

We have now shown that ϕ is a surjective mapping from GW_v^Γ to TW_v^Γ. It must then hold that $|\mathrm{GW}_v^\Gamma| \geq |\mathrm{TW}_v^\Gamma|$, or equivalently, $\mathrm{tw}_v^\Gamma \leq \mathrm{sw}_v^\Gamma$. Then also, $\mathrm{tw}(\Gamma) = \max_v \mathrm{tw}_v^\Gamma \leq \max_v \mathrm{sw}_v^\Gamma = \mathrm{sw}(\Gamma)$. Finally, letting Γ be such that $\mathrm{sw}(G) = \mathrm{sw}(\Gamma)$, proves the bound. □

3.2.2. Reticulation number and level of a network

In this subsection we show that the scanwidth is at most the reticulation number of a network + 1, and consequently at most the level + 1. Although we will make use of a result that will be proved in a later section, we feel that the bound better fits within this section and thus already present it here.

The fact that the scanwidth is at most the level + 1 for a binary network has already been stated without proof in [BSW20] and is proved in the appendix of [Rab+21]. Although a very interesting result, this proof heavily relies on notions that are introduced for the specific context of *vectors of population interfaces*, making it less accessible without understanding the complete context of the paper. Thus, we tackle the bound from a more graph theoretical point of view, making the proof (hopefully) more approachable. Furthermore, we generalize the result to non-binary networks.

As a starting point, we prove a bound concerning the indegrees of sinksets of a network. Recall that we write $W \sqsubseteq V$, to indicate that W is a sinkset of a DAG $G = (V, E)$.

Lemma 3.17. *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*

$$\delta^{\mathrm{in}}(W) \leq k + 1.$$

[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 [Wel23].

Proof. Recall from the preliminary chapter that we can formally write the reticulation number as

$$k = \sum_{v \in V : \delta^{\text{in}}(v) \geq 2} (\delta^{\text{in}}(v) - 1).$$

Now define

$$r_W = \sum_{v \in W : \delta^{\text{in}}(v) \geq 2} (\delta^{\text{in}}(v) - 1),$$

for all $W \subseteq V$. Thus, r_W takes only the reticulations in the set W into account.

We will now prove the stronger statement that for all $W \sqsubseteq V$ with $G[W]$ weakly connected, $\delta^{\text{in}}(W) \leq r_W + 1$. This will immediately imply the lemma, since $r_W \leq k$. For any such set W, we will prove this by induction on r_W.

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

Induction step: ($r_W \geq 1$). Let G be a network with $W \sqsubseteq V$ such that $G[W]$ is weakly connected and $r_W \geq 1$. Assume that the induction hypothesis holds for $r_W - 1$. As $r_W \geq 1$, this means that W contains a reticulation vertex. We can then pick an arc $uv \in E$ such that v is a reticulation vertex in W. Let $G' = G - \{uv\}$. It is easy to see that W is still a sinkset of G', and that $r_W(G') = r_W(G) - 1$. Now consider two cases.

Case 1: If $G'[W]$ is weakly connected, the induction hypothesis shows that $\delta^{\text{in}}_{G'}(W) \leq r_W(G') + 1$. 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 $\delta^{\text{in}}_G(W) \leq \delta^{\text{in}}_{G'}(W) + 1 \leq r_W(G') + 2 = r_W(G) + 1$.

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 1. Therefore, $G'[W]$ consists of two weakly connected components: $G'[W_1]$ and $G'[W_2]$, with $W_1 \cup W_2 = W$ and $W_1 \cap W_2 = \emptyset$. 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

$$\begin{aligned} r_W(G') &= \sum_{v \in W : \delta^{\text{in}}_{G'}(v) \geq 2} (\delta^{\text{in}}_{G'}(v) - 1) \\ &= \sum_{v \in W_1 : \delta^{\text{in}}_{G'}(v) \geq 2} (\delta^{\text{in}}_{G'}(v) - 1) + \sum_{v \in W_2 : \delta^{\text{in}}_{G'}(v) \geq 2} (\delta^{\text{in}}_{G'}(v) - 1) = r_{W_1}(G') + r_{W_2}(G'). \end{aligned}$$

For both $i = 1$ and $i = 2$, we now get that $r_{W_i}(G') \leq r_W(G') = r_W(G) - 1$. Furthermore, $G'[W_i]$ is a weakly connected sinkset, as discussed above. Thus, we can apply the induction hypothesis, and get $\delta^{\text{in}}_{G'}(W_i) \leq r_{W_i}(G') + 1$ for $i = 1, 2$. Using this and the fact that no arc exists between W_1 and W_2 in G', we get

$$\delta^{\text{in}}_{G'}(W) = \delta^{\text{in}}_{G'}(W_1) + \delta^{\text{in}}_{G'}(W_2) \leq (r_{W_1}(G') + 1) + (r_{W_2}(G') + 1) = r_W(G') + 2.$$

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 $\delta^{\text{in}}_G(W) = \delta^{\text{in}}_{G'}(W) \leq r_W(G') + 2 = r_W(G) + 1$.

In both cases, we have shown that $\delta^{\text{in}}(W) \leq r_W + 1$, which concludes the proof. □

Since indegrees of sinksets correspond to cuts in a tree extension, we can now prove that the scanwidth of a network is at most the reticulation number + 1.

Lemma 3.18. *Let $G = (V, E)$ be a network with reticulation number k. Then,*

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

Proof. Let Γ be an optimal tree extension of G. By Corollary 3.3, we can choose Γ to be canonical. Let $v \in V$ be arbitrary, and consider the set GW_v^Γ. We can write $\mathrm{sw}_v^\Gamma = |\mathrm{GW}_v^\Gamma| = \delta^{\mathrm{in}}(V(\Gamma_v))$, where Γ_v is the subtree of Γ rooted at v. But as Γ is canonical, Proposition 3.5 implies that $G[V(\Gamma_v)]$ is weakly connected. Clearly, $V(\Gamma_v)$ is also a sinkset (as Γ is a tree extension of G). According to Lemma 3.17, we then have that $\mathrm{sw}_v^\Gamma = \delta^{\mathrm{in}}(V(\Gamma_v)) \leq k + 1$.

Since v was arbitrary,

$$\mathrm{sw}(G) = \mathrm{sw}(\Gamma) = \max_{v \in V} \mathrm{sw}_v^\Gamma \leq k + 1.$$

□

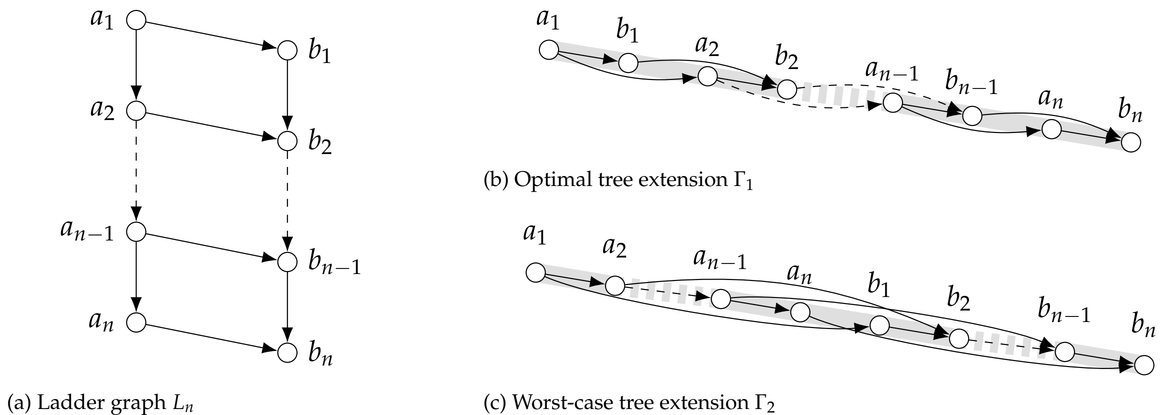
The following result leans on a result in a later section but should be quite intuitive in its own right.

Corollary 3.19. *Let $G = (V, E)$ be a rooted level-k network, then*

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

Proof. By definition, each block of G has a reticulation number of at most k. As any block of a network is a network in itself, we can apply Lemma 3.18 here. Therefore, each block of G has a scanwidth of at most $k + 1$. In Corollary 3.26 in the next section we will show that the scanwidth of a rooted, weakly connected DAG is equal to the maximum scanwidth of its blocks. As a consequence, we have that the scanwidth of G is at most $k + 1$. □

The above bound is certainly not tight in general. Consider for example the network in Figure 3.3, which is a variation of a network from [Rab+21]. This network always has a scanwidth of 3 but can be extended to have an arbitrarily large level.



(a) Ladder graph L_n

(b) Optimal tree extension Γ_1

(c) Worst-case tree extension Γ_2

Figure 3.3: (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.

3.2.3. Weak-scanwidth: a lower bound

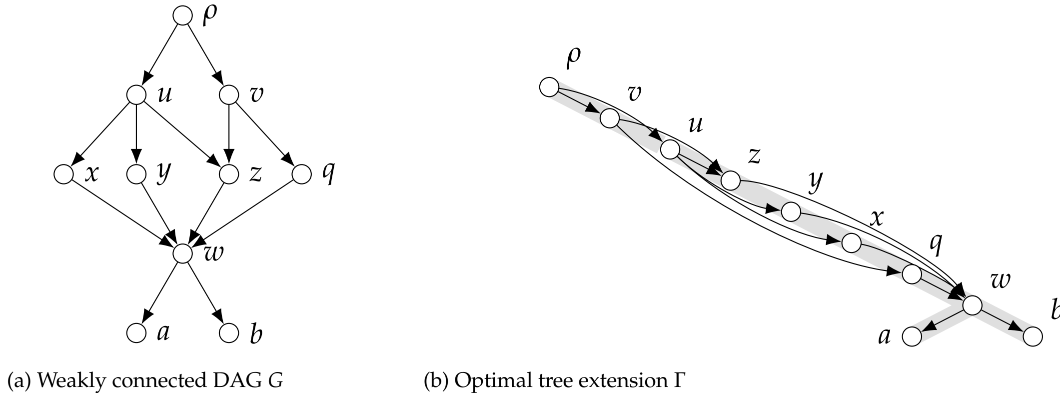
In [Bez99] a lower bound of the undirected cutwidth is mentioned. This bound has its origin in *edge isoperimetric theory*: the study of edge-cuts obtained by fixed-sized bipartitions of a graph. We introduce a related bound for scanwidth: *weak scanwidth*.

Definition 3.20 (Weak scanwidth). *Let $G = (V, E)$ be a weakly connected DAG, and for all $W \sqsubseteq V$, denote by $\overline{\delta^{\text{in}}}(W)$ the largest indegree in G of the vertex sets of any of the weakly connected components of $G[W]$. The* weak scanwidth *of G is then*

$$\text{zw}(G) = \max_{i \in [V]} \min_{W \sqsubseteq V : |W| = i} \overline{\delta^{\text{in}}}(W).$$

Furthermore, we let $\text{zw}_i(G) = \min_{W \sqsubseteq V:|W|=i} \overline{\delta^{\text{in}}}(W)$, for any $i \in [V]$.

To illustrate this definition, consider the graph G in Figure 3.4a. G has only one sink-set of size 2: $W = \{a, b\}$. Since $G[W]$ is not weakly connected, $\overline{\delta^{\text{in}}}(W)$ only counts the largest indegree of any of its components. In this case, both components have an indegree of 1, thus we have that $\text{zw}_2(G) = 1$. On the other hand, G has two sinksets of size 8: $W_1 = \{a, b, w, x, y, z, q, u\}$ and $W_2 = \{a, b, w, x, y, z, q, v\}$. Both of them induce subgraphs that are weakly connected, implying that $\overline{\delta^{\text{in}}}$ equals their indegrees, which are $\delta^{\text{in}}(W_1) = 3$ and $\delta^{\text{in}}(W_2) = 4$. We then get that $\text{zw}_8(G) = \min\{\delta^{\text{in}}(W_1), \delta^{\text{in}}(W_2)\} = 3$. If one repeats this for every i, it will turn out that $\text{zw}(G) = 4$.



(a) Weakly connected DAG G (b) Optimal tree extension Γ

Figure 3.4: (a): Weakly connected DAG G with $\text{sw}(G) = 5$ and $\text{zw}(G) = 4$. (b): Optimal tree extension Γ of G.

The scanwidth of the graph G equals 5 (see the optimal tree extension in Figure 3.4b). It is no coincidence that this value is larger than the weak scanwidth of the graph. As we will prove in Lemma 3.21, the weak scanwidth is always a lower bound on the scanwidth. Moreover, any solution to the weak scanwidth problem (i.e. any zw_i) serves as a lower bound for the scanwidth.[4]

Lemma 3.21. *Let $G = (V, E)$ be a weakly connected DAG, then*

$$\text{zw}(G) \leq \text{sw}(G).$$

[4]This is the reason for the name 'weak' scanwidth. It mimics the concept of 'weak duality': we have a maximization problem where any solution functions as a lower bound for a related minimization problem.

Proof. We claim that for any $i \in [V]$, there exists an optimal extension σ such that $|\mathrm{SW}_i^\sigma| = \overline{\delta^{\mathrm{in}}}(\sigma[1\dots i])$.

Proof of claim: For any extension σ, the set SW_i^σ contains all arcs entering the component $G[U]$ of $G[1\dots i]$ that contains $\sigma(i)$, with $U \sqsubseteq \sigma[1\dots i]$. Clearly, $\delta^{\mathrm{in}}(U) \leq \overline{\delta^{\mathrm{in}}}(\sigma[1\dots i])$. We then get that $\mathrm{SW}_i^\sigma = \delta^{\mathrm{in}}(U) \leq \overline{\delta^{\mathrm{in}}}(\sigma[1\dots i])$.

Now assume that an optimal σ does not have the desired property, i.e. $|\mathrm{SW}_i^\sigma| < \overline{\delta^{\mathrm{in}}}(\sigma[1\dots i])$. Then this maximum indegree must be attained by some other sinkset. Let $W \sqsubseteq \sigma[1\dots i]$ be such that $\delta^{\mathrm{in}}(W) = \overline{\delta^{\mathrm{in}}}(\sigma[1\dots i])$. We will construct an optimal extension π that has the desired property. Let $\sigma(j)$ be the vertex in W that is the highest in the extension (note that $j < i$, as if $j = i$, we would already have the property). The extension where we move $\sigma(j)$ upwards to above $\sigma(i)$ will be our extension π. We have that π is still an extension of the canonical tree extension Γ^σ, because all vertices between positions i and j are not weakly connected to $\sigma(j)$ in $\sigma[1\dots i]$. By Proposition 3.5, the optimal tree extension Γ^σ is then also canonical for π, and so π is also optimal.

As now $|\mathrm{SW}_i^\pi| = \delta^{\mathrm{in}}(W) = \overline{\delta^{\mathrm{in}}}(\sigma[1\dots i]) = \overline{\delta^{\mathrm{in}}}(\pi[1\dots i])$, the claim is proved. △

Let $i \in \{1,\dots,|V|\}$ be arbitrary, and let σ be an optimal extension as in the claim. Due to the property of the claim, we have that $|\mathrm{SW}_i^\sigma| = \overline{\delta^{\mathrm{in}}}(\sigma[1\dots i]) \geq \min_{W \sqsubseteq V : |W| = i} \overline{\delta^{\mathrm{in}}}(W)$. Here, we used that $\sigma[1\dots i] \sqsubseteq V$ (else, σ would not be G-respecting) and $|\sigma[1\dots i]| = i$. From this inequality, we obtain that

$$\mathrm{sw}(G) = \mathrm{sw}(\sigma) = \max_{j \in [V]} |\mathrm{SW}_j^\sigma| \geq |\mathrm{SW}_i^\sigma| \geq \min_{W \sqsubseteq V : |W| = i} \overline{\delta^{\mathrm{in}}} = \mathrm{zw}_i(G).$$

Since i was arbitrary, it now follows immediately that

$$\mathrm{zw}(G) = \max_{i \in [V]} \mathrm{zw}_i(G) \leq \mathrm{sw}(G).$$

□

Intuitively, the above proof uses the observation that one can relate the sets W of size i, to the first i vertices of an extension. This brings us to the reason why equality does not hold in the above lemma: the *nested solutions property* [Bez99] does not always hold. Formally, this means that there does not need to exist a nested sequence of sinksets $W_1 \sqsubset W_2 \sqsubset \cdots \sqsubset W_{|V|}$ with $|W_i| = i$ and $\overline{\delta^{\mathrm{in}}}(W_i) = \mathrm{zw}_i(G)$, for all $i \in [V]$. As a consequence, we are not always able to construct an extension σ from the sequence of nested sinksets, with the property that $\mathrm{sw}_i^\sigma = \mathrm{zw}_i(G)$ for each i.

For an example where this property does not hold, we revert our attention back to the DAG G from Figure 3.4, for which we have that $\mathrm{zw}_6(G) = 4$. The two sinksets of size 6 for which this value is attained, both contain the vertex q. However, the sinkset $\{a,b,x,y,z,u\}$, which uniquely determines $\mathrm{zw}_7(G)$, does not contain q. So indeed, the nested solutions property does not hold for this graph G.

Whenever the optimal values do coincide, a solution to the weak scanwidth problem provides a certificate of optimality for the scanwidth problem. Consider for example Figure 3.3. Say we are given the extension of scanwidth 3 from that figure. It can then be shown to be an optimal solution, by using that $\mathrm{zw}(G) \geq \mathrm{zw}_3(G) = 3$. Furthermore, this value is easily found. It requires us to calculate $\overline{\delta^{\mathrm{in}}}$ only for the two sinksets of size 3: $\{a_n, b_{n-1}, b_n\}$ and $\{b_{n-2}, b_{n-1}, b_n\}$.

3.3. Reduction rules

Before discussing algorithms that compute the scanwidth and find its corresponding (tree) extension, it is useful to create some reduction rules. In Subsection 3.3.1 we will explain that one can split the scanwidth problem over the blocks of a network. Subsection 3.3.2 will cover an arc-contraction rule, while Subsection 3.3.3 summarizes the complete reduction scheme and provides a nice bound on the size of a reduced instance.

3.3.1. (S-)blocks of a graph

Magne et al. [Mag+21] mention that the edge-treewidth can be split into subproblems for each block of a graph. It is rather intuitive that a similar result is true for the scanwidth of a rooted DAG. One needs to be careful, however, as this is not the case for DAGs with multiple roots. Think for example of a wide upside-down tree. That is, each leaf of the tree becomes a root, and its only root becomes the sole leaf. Such an upside-down tree will have a scanwidth that is larger than 1. However, each block of such a DAG will be a single arc of scanwidth 1.

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*.

Definition 3.22 (S-block)**.** *Let $G = (V, E)$ be a weakly connected DAG and v a cut-vertex of G. If at least one of the connected components of $G - \{v\}$ does not contain any root of G, we call v a* directed cut-vertex. *An* s-block *of G is a maximal weakly connected induced subgraph of G without any directed cut-vertices.*

Intuitively, this definition prevents us from splitting a graph at a cut-vertex such that all resulting subgraphs will contain a root of the DAG. In such a case, we can never be sure that one such block will be above the other. Although this intuitively makes sense, it will be more convenient to characterize the s-blocks using the blocks of an auxiliary graph. Proposition 3.23 formally describes this idea, while Figure 3.5 illustrates this characterization and the above definition of s-blocks.

Proposition 3.23. *Let $G = (V, E)$ be a weakly connected DAG, and let H be the underlying undirected graph of G where we add edges between all roots of G. Then for all $W \subseteq V$, the subgraph $G[W]$ is an s-block of G, if and only if $H[W]$ is a block of H.*

Proof. We will prove that a vertex $v \in V$ is a directed cut-vertex in G, if and only if it is a cut-vertex in H. Since G and H have the same vertex set, this then immediately implies that the s-blocks of G coincide with the blocks of H.

($\Rightarrow$) Let v be a directed cut-vertex in G. By definition, v is a cut-vertex in G, and at least one of the components $G[U]$ (with $|U| < |V|$) of $G - \{v\}$ does not contain a root of G. Then, $H[U]$ must also be a component of $H - \{v\}$. This is true, since the only additional edges that H has compared to G are between roots, and those edges have no endpoints in U. As $|U| < |V|$, the graph $H - \{v\}$ contains multiple components. Thus, v is a cut-vertex in H.

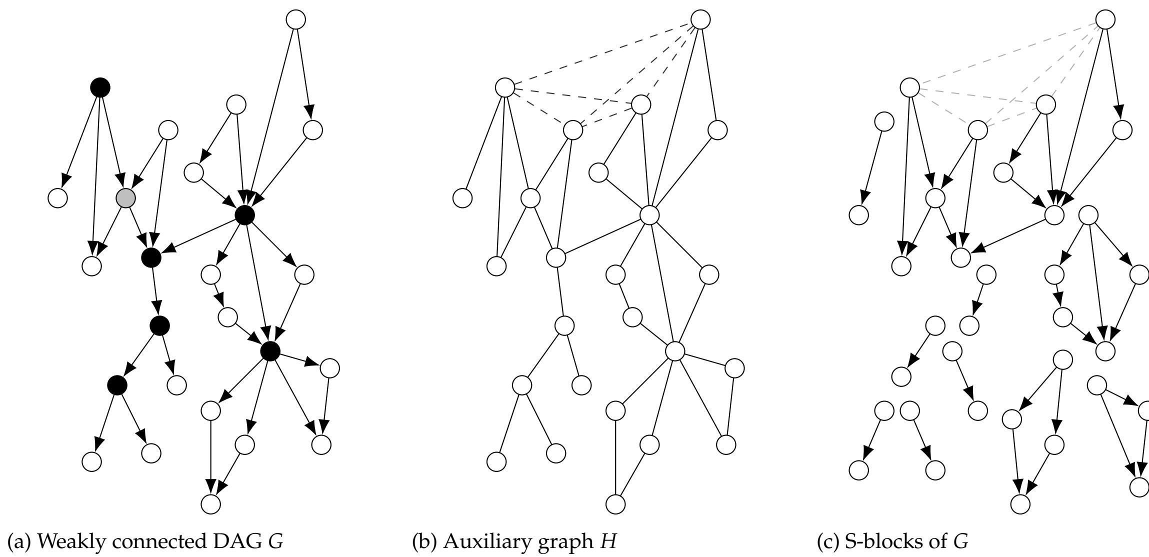
($\Leftarrow$) Let v be a cut-vertex in H. By definition, $H - \{v\}$ contains multiple components. Now assume towards a contradiction that all components of $H - \{v\}$ contain a root of G. As H contains an edge from each root of G to any other root of G, those components must be connected: a contradiction. Thus, there exists at least one component $H[U]$ (with $|U| < |V|$) of $H - \{v\}$ that does not contain a root of G. Clearly, v is also a cut-vertex of G, because G contains fewer arcs than the edges of H. Furthermore, $G[U]$ must be a

component of $G - \{v\}$. Therefore, $G - \{v\}$ contains a component without a root, and so we must have that v is a directed cut-vertex in G. □

A direct consequence of this proposition is that if G is rooted, s-blocks are equivalent to blocks. We formalize this in the following corollary.

Corollary 3.24. *Let $G = (V, E)$ be a weakly connected rooted DAG. Then, for all $W \subseteq V$, the subgraph $G[W]$ is an s-block of G, if and only if it is a block of G.*

Proof. Let H be the auxiliary graph as defined in Proposition 3.23. The proposition says that the s-blocks of G are exactly the blocks of H. But as G has just one root, H will just be the underlying undirected graph of G. Thus, its blocks are by definition the blocks of G. This proves the result. □



(a) Weakly connected DAG G (b) Auxiliary graph H (c) S-blocks of G

Figure 3.5: (a): A multi-rooted weakly connected DAG G, with its directed cut-vertices coloured in black and one cut-vertex that is not a directed cut-vertex coloured in grey. (b): The auxiliary undirected graph H (as defined in Proposition 3.23) where the newly added edges are dashed. (c): The s-blocks of the DAG G.

The proof of the following theorem is rather technical, yet the idea 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 3.25. *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 Proposition 3.23, these s-blocks are exactly the blocks of the auxiliary graph H defined in that proposition. 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 Figure 3.5c). Furthermore, any other block in H

(and thus s-block in G) can have at most one root in it, since blocks can have at most one vertex in common with another block. Thus, all vertices in such other blocks are 'below' this root-block. Because we have more than one s-block, we can indeed find such an s-block that is below the root-block. It should not be surprising that we can move down in the graph, and even find an s-block $G_1 = G[V_1]$ (with $V_1 \subset V$) that is *pendant*. That is, G_1 is attached to the rest of G by a directed cut-vertex v, all vertices of V_1 are below v in G, and no other s-blocks are 'below' G_1.[a]

Now define $G_2 = G[V \backslash V_1 \cup \{v\}] = (V_2, E_2)$. Then, G_1 and G_2 are subgraphs that only have v in common. Let $\sigma_1 \in \Pi[V_1]$ and $\sigma_2 \in \Pi[V_2]$. As v was above all vertices of G_1, it is the root of G_1. Thus, 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 pendant 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:

$$\begin{aligned} \mathrm{sw}(\sigma, G) = \max_{w \in V} \mathrm{sw}_w^{\sigma} &= \max\left\{ \max_{w \in V_1 \setminus \{v\}} \mathrm{sw}_w^{\sigma}, \max_{w \in V_2} \mathrm{sw}_w^{\sigma} \right\} \\ &= \max\left\{ \max_{w \in V_1 \setminus \{v\}} \mathrm{sw}_w^{\sigma_1}, \max_{w \in V_2} \mathrm{sw}_w^{\sigma_2} \right\} \\ &= \max\left\{ \max_{w \in V_1} \mathrm{sw}_w^{\sigma_1}, \max_{w \in V_2} \mathrm{sw}_w^{\sigma_2} \right\} \\ &= \max\{\mathrm{sw}(\sigma_1, G_1), \mathrm{sw}(\sigma_2, G_2)\}. \end{aligned} \tag{3.1}$$

Here, we used that $\mathrm{sw}_v^{\sigma_1} = 0$ for the graph G_1, 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, it then follows by the induction hypothesis that

$$\mathrm{sw}(G) = \max\{\mathrm{sw}(G_1), \mathrm{sw}(G_2)\} = \max\{\mathrm{sw}(G_1), \max_{H \in \mathcal{S}(G_2)} \mathrm{sw}(H)\} = \max_{H \in \mathcal{S}(G)} \mathrm{sw}(H),$$

which proves 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 (3.1),

$$\mathrm{sw}(G) \leq \mathrm{sw}(\sigma, G) = \max\{\mathrm{sw}(\sigma_1, G_1), \mathrm{sw}(\sigma_2, G_2)\} = \max\{\mathrm{sw}(G_1), \mathrm{sw}(G_2)\}.$$

($\geq$) Let Γ be an optimal canonical tree extension of G, which exists by Corollary 3.3. Because the vertices of $V_1 \setminus \{v\}$ are not above any of the vertices in V_2, Γ must have an extension of the form $\sigma = \sigma_1' \circ \sigma_2$, with $\sigma_1' \in \Pi[V_1 \setminus \{v\}]$ and $\sigma_2 \in \Pi[V_2]$. By Proposition 3.5, Γ is then canonical for σ, and we thus have that $\mathrm{sw}(\sigma, G) = \mathrm{sw}(\sigma)$. Using equation (3.1), we then have that

$$\mathrm{sw}(G) = \mathrm{sw}(\sigma, G) = \max\{\mathrm{sw}(\sigma_1, G_1), \mathrm{sw}(\sigma_2, G_2)\} \geq \max\{\mathrm{sw}(G_1), \mathrm{sw}(G_2)\}.$$

□

[a] This is the reason we introduced the s-blocks. Else, we can not ensure that there exists such a pendant s-block. A graph could consist of two blocks, both containing a root, attached to a leaf of the graph. Then, neither of the blocks is completely below the other. However, both blocks form one s-block (namely the root-block), and therefore such a graph would consist of just one s-block.

An immediate corollary is now that one can split over the blocks to compute the scanwidth if G has a single root. Note that we already used this result in Subsection 3.2.2 to show that the scanwidth is less than $k+1$ for level-k networks.

Corollary 3.26. *Let G be a weakly connected rooted DAG and $\mathcal{B}(G)$ the set of blocks of G. Then, we have that*

$$\mathrm{sw}(G) = \max_{H \in \mathcal{B}(G)} \mathrm{sw}(H).$$

Proof. This follows directly from the equivalence of s-blocks and blocks in rooted DAGs, as shown in Corollary 3.24, and from Theorem 3.25. □

3.3.2. Arc contractions

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 a sense, we 'contract' arcs of indegree-1 and outdegree-1, hence the name of this subsection.

Lemma 3.27. *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 $uw \notin E$. Let H be the same graph, where the vertex v is deleted, the arcs uv and vw are deleted, and the arc uw is added. Then, $\mathrm{sw}(G) = \mathrm{sw}(H)$.*

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 $a \overset{G[\sigma[1\dots b]]}{\leftrightsquigarrow} b$, if and only if $a \overset{H[\pi[1\dots b]]}{\leftrightsquigarrow} b$. Furthermore, $a >_G b$, if and only if $a >_H b$.

> *Proof of claim:* First note that π indeed is an extension of H, as $u >_H w$. The second part of the claim is trivial and needs no proof.
>
> ($\Rightarrow$) Let $a, b \in V \setminus \{v\}$, and assume that $a \overset{G[\sigma[1\dots b]]}{\leftrightsquigarrow} a$. Then, there exists an undirected a-b path in $G[\sigma[1 \dots b]]$. If this path does not use uv and vw, the same path connects a and b in $H[\pi[1 \dots b]]$. If it does use uv or vw, both must be used, as a nor b can equal v (by assumption), and v has no other adjacent vertices. But then, we can replace these two arcs with uw to get a path in H. This proves the first direction.
>
> ($\Leftarrow$) Let $a, b \in V \setminus \{v\}$, and assume that $a \overset{H[\pi[1\dots b]]}{\leftrightsquigarrow} b$. Then, there exists an undirected a-b path in $H[\pi[1 \dots b]]$. If this path does not use uw, the same path connects a and b in $G[\sigma[1 \dots b]]$. If it does use uw, we can replace the arc with uv and vw in G. Consequently, there is an a-b path in G. This proves the claim. △

($\mathrm{sw}(G) \geq \mathrm{sw}(H)$) 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 $\mathrm{sw}_z^\pi(H) = |\mathrm{SW}_z^\pi(H)| \leq |\mathrm{SW}_z^\sigma(G)| = \mathrm{sw}_z^\sigma(G)$. 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 $\sigma = \pi[1 \dots i] \circ (v) \circ \pi[i+1 \dots]$ 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 $\mathrm{sw}_z^\sigma(G) = |\mathrm{SW}_z^\sigma(H)| \leq |\mathrm{SW}_z^\pi(H)| = \mathrm{sw}_z^\pi(H)$.

By similar arguments, one finds that $\mathrm{sw}_v^\sigma(G) = \mathrm{sw}_w^\pi(H)$. Therefore,

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

□

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.

3.3.3. Complete decomposition scheme

In combination with any exact algorithm, the reduction rules from the previous two subsections can be used to compute the scanwidth. This scheme is gathered in the following decomposition algorithm:

Algorithm 3: 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 $\leftarrow \max\{\mathrm{sw}, 1\}$
7 **else if** *H is a cycle with a unique root* **then**
8 sw $\leftarrow \max\{\mathrm{sw}, 2\}$
9 **else**
10 $H' \leftarrow H$ after exhaustively contracting arcs using Lemma 3.27
11 sw $\leftarrow \max\{\mathrm{sw}, \texttt{ExactSW}(H')\}$
12 **return** sw

To summarize, the algorithm first decomposes a DAG into its s-blocks, using the auxiliary graph G'. 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 arc contraction rule to decrease their size. We then only need to exactly calculate the scanwidth of those s-blocks with some exact algorithm. Figure 3.6 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 3.25 and Lemma 3.27 can be used to make the algorithm constructive, if `ExactSW` also returns an optimal extension.

Lemma 3.28. *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 3 returns $\mathrm{sw}(G)$ 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 3.*

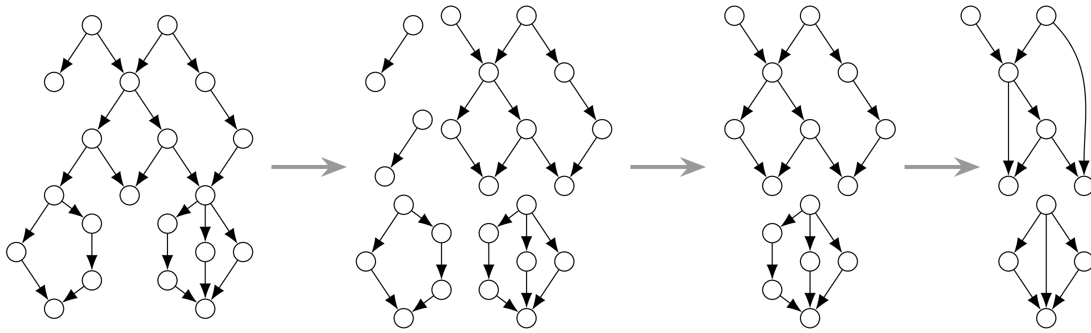


Figure 3.6: Illustration of the decomposition algorithm. The first graph shows the original graph. Then, the graph is split into its s-blocks. Thereafter, the single arcs and the cycle with a unique root are 'deleted', since their scanwidths are known. Note that we then also know that the scanwidth of the complete graph is at least 2. Lastly, we exhaustively contract indegree-1 outdegree-1 arcs in the remaining s-blocks, until it will create a multigraph.

Proof. Correctness: By Proposition 3.23, $\mathcal{S}$ will be the set of s-blocks of G. From Theorem 3.25 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 3.27, 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 [HT73]. 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.[a] Exhaustively contracting arcs also takes $O(n)$ time, since we check for each vertex if it has indegree-1 and outdegree-1, together with some other constant operations. 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 is $O(n^2 + n \cdot (n + f(n', m'))) = O(n^2 + n \cdot f(n', m'))$. □

[a] Checking whether H is a cycle with a unique root can for example be done by checking if H has one vertex with indegree-0 and outdegree-2, one vertex with indegree-2 and outdegree-0, and then checking whether the remaining vertices have indegree-1 and outdegree-1.

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*. These building blocks of binary networks were introduced in [Ier09; Ier+09]. We mention that for binary networks the only difference between these generators and our graphs H' is that to obtain level-k generators we always contract arcs, even if they lead to a multigraph. We will not prove this statement, as the exact relation to level-k generators is not of immediate interest to us. What is however useful, is a known bound on the size of the level-k generators.

In [Ier09, Lem 4.2] it is shown that the number of nodes 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 extend these bounds to non-binary networks. As we do not allow multigraphs, we do need a small adjustment to the bounds. However, they remain linear in k, so we feel this is justified for the sake of simplicity. We prove these bounds directly, thus eliminating the need to formally introduce generators.

Lemma 3.29. *Let $G = (V, E)$ be a level-k network, and let H be a block of G. Let H' be the block H after exhaustively applying the arc-contraction reduction rule from Lemma 3.27. Then, $|V(H')| \leq 4k-1$ and $|E(H')| \leq 5k-2$.*

Proof. We can assume that H' is not a single arc, as then the lemma follows trivially. We now first prove the following claim on the types of vertices that can occur in H'.

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 $\delta^{\text{out}}(v) \leq 1$.

> *Proof of claim:* Because G has a unique root, and H' originated from a block H of G, the subgraph H' must have a unique root. The outdegree of this root must be at least 2, else the outgoing arc would be a block on its own: a case we excluded. Now let $v \in V(H')$ be an arbitrary vertex that is not the unique root of H'. Then, $\delta^{\text{in}}(v) > 0$, and we can consider two cases. (*Case 1*: $\delta^{\text{in}}(v) = 1$). In this case, the outdegree of v can not be 0, else v would be a leaf with indegree 1. This would mean that the incoming arc is a block on its own, thus H' must again be a single arc: a case we already excluded. Thus, v has an outdegree of at least 1 and must either be a flow vertex or a tree-vertex. (*Case 1*: $\delta^{\text{in}}(v) \geq 2$). Then, v can not have an outdegree that is greater than 1 since G is a network, and it does not have vertices with both in- and outdegree larger than 1. Furthermore, the splitting into blocks and the contraction of indegree-1 outdegree-1 arcs will never create such vertices. Therefore, v must be a reticulation vertex, which proves the claim. △

Now suppose that the unique root of H' has outdegree $\alpha \geq 2$, 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'. Note that the degrees that are not mentioned, are fixed by the claim.

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 enter a reticulation, else we would be able to contract it. Furthermore, each reticulation v can have at most $\delta^{\text{in}}(v) - 1$ flow vertices as its parents. For if all its parents were flow vertices, we would be able to contract 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:

$$|V(H')| = 1 + f + t + r \leq 1 + (\gamma_1 - 1) \cdot r + \gamma_1 r - \alpha + r \leq 4 \cdot (\gamma_1 - 1) \cdot r - 1.$$

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

$$|E(H')| = f + t + \gamma_1 r \leq (\gamma_1 - 1) \cdot r + \gamma_1 r - \alpha + \gamma_1 r \leq +5 \cdot (\gamma_1 - 1) \cdot r - 2.$$

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. Contracting some indegree-1 outdegree-1 arcs in H will not increase this number. By the definition of the reticulation number from Subsection 2.2.2, 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 $(\gamma_1 - 1) \cdot r \leq k$. Filling this in into the two upper bounds then proves the lemma. □

3.4. Scanwidth-1 and scanwidth-2 characterizations

Using results from the previous section and characterizations for the edge-treewidth proved in [Mag+21], we are able to provide characterizations for rooted DAGs of scanwidth 1 and 2.

Proposition 3.30. *Let $G = (V, E)$ be a weakly connected rooted DAG. Then the following are equivalent:*

1. $\mathrm{sw}(G) \leq 1$*;*
2. *G is a directed tree.*

Proof. (1) $\Rightarrow$ (2): Denote by $\tilde{G}$ the underlying undirected graph of G. According to Lemma 3.12, we then have that $\mathrm{etw}(\tilde{G}) \leq \mathrm{sw}(G) \leq 1$. In [Mag+21, Thm. 4] it is shown that $\tilde{G}$ is a forest if its edge-treewidth is at most 1. Since G is also weakly connected, $\tilde{G}$ must then be a tree. By definition, this means that G is a directed tree.

(2) $\Rightarrow$ (1): Just taking G itself as the tree extension, automatically gives that its scanwidth is at most 1. □

This result does not need to hold for multi-rooted DAGs. Take for example a DAG with two roots, connected at a single leaf. This is still a directed tree, but it has a scanwidth of 2.

Although it is fairly obvious that rooted cycles (i.e. cycles with a single root and a single leaf), and thus (non-trivial) directed cactuses, have a scanwidth of 2, it is not immediately clear that these are the only rooted DAGs with this property. In the following proposition, we prove that this is indeed the case.

Proposition 3.31. *Let $G = (V, E)$ be a weakly connected rooted DAG. Then the following are equivalent:*

1. $\mathrm{sw}(G) \leq 2$*;*
2. *G is a directed cactus.*

Proof. (1) $\Rightarrow$ (2): We let $\tilde{G}$ be the underlying undirected graph of G. We have that $\mathrm{etw}(\tilde{G}) \leq \mathrm{sw}(G) \leq 2$ by Lemma 3.12. In [Mag+21, Thm 5.] it is shown that $\tilde{G}$ is a cactus graph if its edge-treewidth is at most 2. By definition, G is thus a directed cactus.

(2) $\Rightarrow$ (1): By definition, the underlying undirected graph $\tilde{G}$ is a cactus graph. Assuming we are not in the trivial case where $|V| = 1$, all blocks of $\tilde{G}$ must then be a single edge or a cycle. As G has a single root, each block of G must then either be a single arc of scanwidth 1 or a rooted cycle of scanwidth 2. By Corollary 3.26, it follows that $\mathrm{sw}(G) \leq 2$. □

Similar to the first proposition, this result also does not extend to DAGs with multiple roots. As an example, consider two rooted cycles that meet at a single leaf. Such a graph is a cactus but has scanwidth 4 (which is the indegree of the leaf).

Note that the only rooted DAGs with scanwidth 0 are the single vertices. Then, the previous two propositions imply that the rooted DAGs with scanwidth equal to 1 are the directed trees that have at least 2 vertices, while the rooted DAGs with scanwidth equal to 2 are the directed cactus graphs that are not directed trees.

Chapter 4

Exact algorithms

In this chapter, our focus lies on methods that exactly compute the scanwidth. Section 4.1 covers a naive brute force solution. In Section 4.2 we look at a recursive algorithm, while the subsequent section focuses on a dynamic programming solution. 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 (as discussed in Subsection 2.2.5), but in this chapter we only consider solution methods that construct optimal extensions. With Algorithm 2 it is possible to create an optimal tree extension from an optimal extension in quadratic time.

We also mention that Appendix A contains an integer linear programming formulation as a solution method, which we provide as a starting point for possible future research.

4.1. Brute force solution

As a benchmark, we will first shortly discuss 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. It is important to note that the number of permutations for a graph with n vertices is $n!$. This results in a combinatorial explosion of the search space for a growing input size. For later reference, we summarize this approach in the following proposition.

Proposition 4.1. *Let $G = (V, E)$ be a weakly connected DAG of n vertices and m arcs, then a brute-force algorithm will solve* SCANWIDTH *in $O(n! \cdot n \cdot m)$ time.*

Proof. G has at most $n!$ extensions, and by Corollary 3.9, it takes $O(n \cdot m)$ time to calculate the scanwidth of each of those. The statement then follows. □

It seems beneficial to use an algorithm that directly creates all extensions, instead of checking for each permutation whether it is an extension. The standard algorithm for this problem

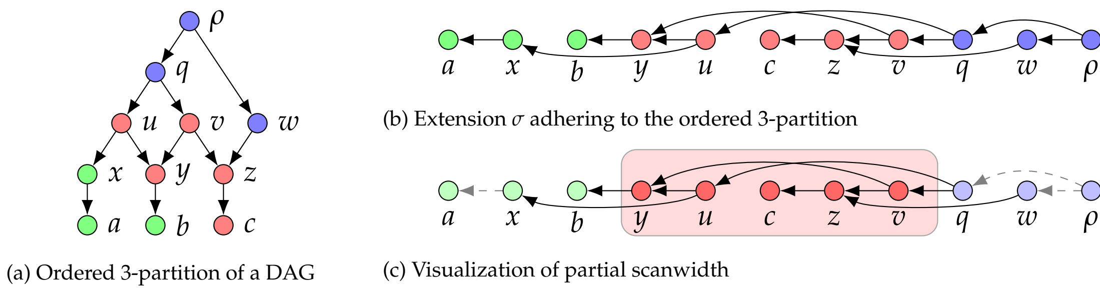
was proposed by Knuth and Szwarcfiter [KS74]. Although the worst-case time complexity of their algorithm is still $O(n!)$, the number of extensions can be significantly smaller than $n!$ in practice.[1] Therefore, using their algorithm will possibly lead to improved efficiency in practice.

4.2. Recursive algorithm

In this section, 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. [Bod+11]. They present a solution method for a range of ordering problems on undirected graphs, using a technique by Gurevich and Shelah [GS87] 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 concept of an ordered partition.

Definition 4.2 (Ordered partition)**.** *Let $G = (V, E)$ be a weakly connected graph and $(A_1, \dots, A_r)$ an r-partition of V, with $r \geq 2$ (and some A_i possibly empty). If for all $1 \leq i < r$, no arcs in E are directed from $\bigcup_{j \leq i} A_j$ towards $\bigcup_{j > i} A_j$, then a $(A_1, \dots, A_r)$ is an* ordered r-partition *of V.*

Intuitively, an ordered partition preserves the natural ordering of a DAG. 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 Figures 4.1a and 4.1b for an illustration of this concept.



(a) Ordered 3-partition of a DAG

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

(c) Visualization of partial scanwidth

Figure 4.1: (a): The weakly connected DAG G from Figure 2.4a, with the colors indicating an ordered 3-partition (L, W, R) of $V(G)$. L is in green, W in red and R in blue. (b): The extension σ of G from Figure 2.4d which is a concatenation of extensions of the three subgraphs induced by the partition (L, W, R). (c): The same extension σ but with a 'window' drawn around the red vertices, aiding the interpretation of partial scanwidth. The arcs between two green (resp. blue) vertices are grey and dashed because they never count towards the partial scanwidth for this ordered 3-partition.

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 4.3 (Partial scanwidth)**.** *Let $G = (V, E)$ be a weakly connected DAG and $\mathcal{Q} = (L, W, R)$ 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*

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

[1] In [KS74] it is explained that their algorithm can be made iterative with a running time of $O(2^n \cdot n)$, at the cost of its space complexity. However, it does so by not storing all extensions. To find the extension with the smallest scanwidth we have to iterate over all generated solutions anyway, rendering this improvement useless for our cause.

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

$$\mathrm{psw}_G(\mathcal{Q}) = \min_{\sigma\in\Pi[W]} \max_{i\in W} |\mathrm{PSW}_i^\sigma(\mathcal{Q})|.$$

Furthermore, we let $\mathrm{psw}(\sigma, \mathcal{Q}) = \max_{i\in W} \mathrm{psw}_i^\sigma(\mathcal{Q})$ *be the partial scanwidth of σ for $\mathcal{Q}$, where* $\mathrm{psw}_i^\sigma(\mathcal{Q}) = |\mathrm{PSW}_i^\sigma(\mathcal{Q})|$ *is the partial scanwidth of σ at position i for $\mathcal{Q}$.*

Essentially, partial scanwidth only considers 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.

Figure 4.1c clarifies this concept. Whereas the standard scanwidth of an extension looks at the scanwidth at each vertex, the partial scanwidth only takes into account the scanwidth of the red vertices. Although the specific ordering of the green (resp. blue) vertices is irrelevant, the green (resp. blue) vertices are assumed to be left (resp. right) of the red vertices. Note that only the arcs that appear in the red 'window' are important. The other (grey and dashed) arcs, which connect two green (resp. blue) vertices, are never counted towards the partial scanwidth.

We can now proceed with the main recursive idea of the algorithm. It builds upon Lemma 4 from [Bod+11], 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.

Lemma 4.4. *Let $G = (V, E)$ be a weakly connected DAG and $\mathcal{Q} = (L, W, R)$ an ordered 3-partition of V, with $W \neq \emptyset$.*

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

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

$$\mathrm{psw}_G(\mathcal{Q}) = \min_{W'\in\mathcal{W}_k} \max\left\{\mathrm{psw}_G\big((L, W', R\cup(W\backslash W'))\big), \mathrm{psw}_G\big((L\cup W', W\backslash W', R)\big)\right\},$$

where $\mathcal{W}_k = \{W' \subseteq W : |W'| = k$ and no arc in E is directed from W' to $W \setminus W'\}$.

Proof. (a): By Definition 4.3 and the fact that $G[W]$ only has a single extension (w), we get

$$\begin{aligned}\mathrm{psw}_G(\mathcal{Q}) &= \min_{\sigma\in\Pi[W]} \max_{i\in W} |\mathrm{PSW}_i^\sigma(\mathcal{Q})| = |\mathrm{PSW}_w^{(w)}(\mathcal{Q})| \\ &= |\{uv \in E(G) : u \in \emptyset \cup R, v \overset{G[\{w\}\cup L]}{\leftrightsquigarrow} w\}| \\ &= |\{uv \in E(G) : u \in R, v \overset{G[L\cup W]}{\leftrightsquigarrow} w\}|.\end{aligned}$$

(b): Let $k \in \{1, \ldots, |W| - 1\}$ be arbitrary. Throughout the proof, we write $\mathcal{Q}_1(W') = (L, W', R \cup (W\backslash W'))$ and $\mathcal{Q}_2(W') = (L \cup W', W\backslash W', R)$ for any $W' \in \mathcal{W}_k$. Note that both are also ordered 3-partitions of V with the middle set non-empty. This follows from the fact that $\mathcal{W}_k$ contains only sets W' such that no arcs go from W' to $W\backslash W'$ and because $k \geq 1$.

First, 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 $\sigma_1 \circ \sigma_2 = \sigma \in \Pi[W]$. The ordering σ is indeed an

extension of $G[W]$ by the definition of $\mathcal{W}_k$. We then have that

$$
\begin{aligned}
\operatorname{psw}(\sigma, \mathcal{Q}) &= \max_{w \in W} \operatorname{psw}_w^\sigma(\mathcal{Q}) \\
&= \max \left\{ \max_{w \in W'} \operatorname{psw}_w^\sigma(\mathcal{Q}), \max_{w \in W \setminus W'} \operatorname{psw}_w^\sigma(\mathcal{Q}) \right\} \\
&= \max \left\{ \max_{w \in W'} \operatorname{psw}_w^{\sigma_1}(\mathcal{Q}_1(W')), \max_{w \in W \setminus W'} \operatorname{psw}_w^{\sigma_2}(\mathcal{Q}_2(W')) \right\} \\
&= \max \left\{ \operatorname{psw}(\sigma_1, \mathcal{Q}_1(W')), \operatorname{psw}(\sigma_2, \mathcal{Q}_2(W')) \right\}. \qquad (4.1)
\end{aligned}
$$

Here, the third 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 $\operatorname{psw}(\sigma_1, \mathcal{Q}_1) = \operatorname{psw}_G(\mathcal{Q}_1)$ and $\sigma_2 \in \Pi[W \setminus W']$ be such that $\operatorname{psw}(\sigma_2, \mathcal{Q}_2) = \operatorname{psw}_G(\mathcal{Q}_2)$. Both exist by definition of the partial scanwidth. We now define $\sigma = \sigma_1 \circ \sigma_2 \in \Pi[W]$. Using equation (4.1), we then have that

$$
\begin{aligned}
\operatorname{psw}_G(\mathcal{Q}) \leq \operatorname{psw}(\sigma, \mathcal{Q}) &= \max \left\{ \operatorname{psw}(\sigma_1, \mathcal{Q}_1(W')), \operatorname{psw}(\sigma_2, \mathcal{Q}_2(W')) \right\} \\
&= \max \left\{ \operatorname{psw}_G(\mathcal{Q}_1(W')), \operatorname{psw}_G(\mathcal{Q}_2(W')) \right\}.
\end{aligned}
$$

Because $W' \in \mathcal{W}_k$ was arbitrary, we obtain

$$
\operatorname{psw}_G(\mathcal{Q}) \leq \min_{W' \in \mathcal{W}_k} \max \left\{ \operatorname{psw}_G(\mathcal{Q}_1(W')), \operatorname{psw}_G(\mathcal{Q}_2(W')) \right\}.
$$

($\geq$) Let $\sigma \in \Pi[W]$ be such that $\operatorname{psw}_G(\mathcal{Q}) = \operatorname{psw}(\sigma, \mathcal{Q})$, which exists by Definition 4.3. Now choose W'' to be the set of the first k vertices of σ (clearly $W'' \in \mathcal{W}_k$). We now 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 (4.1),

$$
\begin{aligned}
\operatorname{psw}_G(\mathcal{Q}) = \operatorname{psw}(\sigma, \mathcal{Q}) &= \max \left\{ \operatorname{psw}(\sigma_1, \mathcal{Q}_1(W'')), \operatorname{psw}(\sigma_2, \mathcal{Q}_1(W'')) \right\} \\
&\geq \max \left\{ \operatorname{psw}_G(\mathcal{Q}_1(W'')), \operatorname{psw}_G(\mathcal{Q}_1(W'')) \right\}.
\end{aligned}
$$

As W'' was an element of $\mathcal{W}_k$, we can minimize over all $W' \in \mathcal{W}_k$ to obtain

$$
\operatorname{psw}_G(\mathcal{Q}) \geq \min_{W' \in \mathcal{W}_k} \max \left\{ \operatorname{psw}_G(\mathcal{Q}_1(W')), \operatorname{psw}_G(\mathcal{Q}_2(W')) \right\},
$$

which proves the lemma. □

With this at first glance quite complicated recursive relation, we can formulate a relatively concise and elegant algorithm that solves SCANWIDTH optimally. We do so by using the recursive relation from the previous lemma, with 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. Note that in a practical implementation of the algorithm (and also in subsequent algorithms), we can replace the value ∞ with the trivial upper bound of the scanwidth, which is $|E| + 1$ (as indicated in Lemma 3.10b).

Algorithm 4: Recursive algorithm to solve SCANWIDTH.

Input: Weakly connected DAG $G = (V, E)$.
Output: Scanwidth sw of G, optimal extension σ_{opt}.

1 $\text{sw}, \sigma_{\text{opt}} \leftarrow$ `PartialScanwidth`$(\emptyset, V, \emptyset)$
2 **return** $\text{sw}, \sigma_{\text{opt}}$
procedure `PartialScanwidth`(L, W, R)
1 **initialize** $\text{psw} \leftarrow \infty$; $\sigma \leftarrow ()$
2 **if** $|W| = 1$ *with* $W = \{w\}$ **then**
3 $\quad \text{psw} \leftarrow |\{uv \in E : u \in R, v \overset{G[L \cup W]}{\rightsquigarrow} w\}|$
4 $\quad \sigma \leftarrow (v)$
5 **else if** $|W| > 1$ **then**
6 $\quad$ **for each** $W' \subseteq W : |W'| = \left\lfloor \frac{|W|}{2} \right\rfloor$ *and no arc in E is directed from W' to $W \setminus W'$* **do**
7 $\quad\quad \text{psw}'_1, \sigma'_1 \leftarrow$ `PartialScanwidth`$(L, W', R \cup (W \setminus W'))$
8 $\quad\quad \text{psw}'_2, \sigma'_2 \leftarrow$ `PartialScanwidth`$(L \cup W', W \setminus W', R)$
9 $\quad\quad \text{psw}' \leftarrow \max\{\text{psw}'_1, \text{psw}'_2\}$
10 $\quad\quad$ **if** $\text{psw}' < \text{psw}$ **then**
11 $\quad\quad\quad \text{psw} \leftarrow \text{psw}'$
12 $\quad\quad\quad \sigma \leftarrow \sigma'_1 \circ \sigma'_2$
13 **return** psw, σ

It turns out that Algorithm 4 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 4.5. *Let $G = (V, E)$ be a weakly connected DAG of n vertices and m arcs, then Algorithm 4 solves* SCANWIDTH *in $O(4^n \cdot \log n \cdot m)$ time and polynomial space.*

Proof. Correctness: The correctness of the partial scanwidth that is returned by the subroutine `PartialScanwidth` in the algorithm follows directly from Lemma 4.4. In particular, the case where W is only one vertex is covered in part (a) of the lemma, while the other case uses part (b) with k chosen as half the size of W. The correctness of the corresponding extension is a direct consequence of the constructive nature of the proof of Lemma 4.4. Lastly, the definition of the partial scanwidth immediately implies that $\text{psw}_G(\emptyset, V, \emptyset) = \text{sw}(G)$, and the corresponding extension is also optimal for the scanwidth. The algorithm terminates, as each recursive call is made for a strictly smaller set W. This establishes the correctness of the algorithm.

Time complexity: The following analysis partially builds upon the analysis of the recursive algorithm in [Bod+11]. 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, the stuff within each iteration of the for-loop can also be done in $O(m)$ time per recursive call.[a] 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:

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

We now claim that $T(m,k) \leq b \cdot 4^k \cdot m \cdot \log k$, for all $m \geq 1$, $k \geq 2$, and for some constant $b \geq 0$. To not lengthen the analysis too much, we delay its induction-based proof to Lemma C.1 in Appendix C. From this claim - and the fact that the algorithm runs in $T(m,n)$ time - it follows that the algorithm has a time complexity of $O(4^n \cdot \log n \cdot m)$.

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 [Bod+12], where the same explanation is given for a specific case of the algorithm from [Bod+11], applied to treewidth.) □

[a] Note that checking whether no arcs go from W' to $W \setminus W'$, also takes $O(m)$ time.

We can reduce this time complexity to $\tilde{O}(3^n)$ by storing and reusing intermediate results. By doing so, we make at most one recursive call per 3-partition. However, this improvement comes at the cost of exponential space. If we are willing to allocate such space, the algorithm presented in the next section will outperform the current algorithm.

4.3. Dynamic programming

In this section, we follow a different approach to solve SCANWIDTH. In Subsection 4.3.1 we introduce the basic version of this algorithm, which will employ *dynamic programming*: a technique where an algorithm saves the results of subproblems, such that they need not be calculated again further down the line. Subsection 4.3.2 introduces a practical improvement over this basic algorithm, while Subsection 4.3.3 leverages this improvement to efficiently solve SCANWIDTH in polynomial time when the value of the scanwidth is bounded by a constant.

4.3.1. Basic algorithm

The basic version of the algorithm shows some similarities with a dynamic programming algorithm from [Bod+11]. The authors adapt a classical technique by Held and Karp [HK62] for the TRAVELLING SALESMAN PROBLEM to address general ordering problems on undirected graphs. Although the exact formulation of our basic algorithm differs considerably, the algorithms structurally have a close resemblance. Specifically, both algorithms consider each subset (or equivalently, each 2-partition) of the vertex set of a graph at most once, and recursively decrease the size of the considered sets by one. We do feel our approach has a slightly more intuitive interpretation, due to the directions of the arcs.

Our new algorithm neatly fits within the framework of ordered partitions and the partial scanwidth from before. Particularly, 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 4.6 (Restricted partial scanwidth)**.** *Let G be a weakly connected DAG and $\mathcal{Q} = (W, R)$ an ordered 2-partition of V such that $W \neq \emptyset$. Then the* restricted partial scanwidth *of G for $\mathcal{Q}$ is*

$$\mathrm{rpsw}_G(\mathcal{Q}) = \mathrm{psw}_G((\emptyset, W, R)).$$

Similarly, we define $\mathrm{rpsw}(\sigma, \mathcal{Q}) = \mathrm{psw}(\sigma, (\emptyset, W, R))$, $\mathrm{rpsw}_i^\sigma(\mathcal{Q}) = \mathrm{psw}_i^\sigma((\emptyset, W, R))$ *and* $\mathrm{RPSW}_i^\sigma(\mathcal{Q}) = \mathrm{PSW}_i^\sigma((\emptyset, W, R))$.

Note that if (W, R) is an ordered 2-partition, then $(\emptyset, W, R)$ is indeed an ordered 3-partition of a graph, making this is a valid definition. 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.

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 very useful by-product of this inherent relation to the partial scanwidth is that we can formulate the main recursive idea of the dynamic programming algorithm as a specific case of Lemma 4.4.

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

(a) If $|W| = 1$ (with $W = \{w\}$), then $\mathrm{rpsw}_G(\mathcal{Q}) = \delta^{\mathrm{in}}(w)$.

(b) If $|W| \geq 2$, then

$$\mathrm{rpsw}_G(\mathcal{Q}) = \min_{\rho \in P(G[W])} \max \left\{ \mathrm{rpsw}_G\big((W \setminus \{\rho\}, R \cup \{\rho\})\big), |\{uv \in E : u \in R, v \overset{G[W]}{\leftrightsquigarrow} \rho\}| \right\},$$

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

Remark. If $G[W]$ is weakly connected, we have $|\{uv \in E : u \in R, v \overset{G[W]}{\leftrightsquigarrow} \rho\}| = \delta^{\mathrm{in}}(W)$ for all $\rho \in P(G[W])$.

Proof. (a): First note that $R = V \setminus \{w\}$ in this case. Then from Lemma 4.4a and Definitions 4.3 and 4.6 we obtain

$$\begin{aligned} \mathrm{rpsw}_G(\mathcal{Q}) = \mathrm{psw}_G((\emptyset, W, V \setminus \{w\})) &= |\{uv \in E : u \in V \setminus \{w\}, v \overset{G[\emptyset \cup \{w\}]}{\leftrightsquigarrow} w\}| \\ &= |\{uv \in E : u \neq w, v = w\}| \\ &= \delta^{\mathrm{in}}(w). \end{aligned}$$

(b): Recall that in Lemma 4.4b, 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 $k \in \{1, \dots, |W| - 1\}$. We now use this lemma with $k = |W| - 1$, to get

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

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 $G[W]$. It is evident that any subset of W of size $|W| - 1$ can be obtained by removing one vertex from W. Having the constraint that no arc in E points from W' to $W \setminus W'$ then implies that we can only remove the roots of $G[W]$ to create these sets W'.

Now using Lemma 4.4a, we immediately have that

$$\mathrm{psw}_G\big((W \setminus \{\rho\}, \{\rho\}, R)\big) = |\{uv \in E(G) : u \in R, v \overset{G[W]}{\leftrightsquigarrow} \rho\}|,$$

which proves the equality.

(Remark): For the remark, observe that if $G[W]$ is weakly connected, the vertices v with the property that $v \overset{G[W]}{\leftrightsquigarrow} \rho$, are exactly the vertices of W (because $\rho \in W$). As (W, R) is a partition of V, the equality follows. □

With this lemma in mind, we can set up the dynamic programming algorithm. The algorithm involves the use of a 'table', denoted as T, to store previously calculated results. In this way, the recursive procedure can first check if a result is already known, thereby saving time. However, this comes at the cost of exponential space usage.

Algorithm 5: Dynamic programming algorithm to solve SCANWIDTH.

Input: Weakly connected DAG $G = (V, E)$.
Output: Scanwidth sw of G, optimal extension σ_{opt}.

```
1  T ← empty table to tabulate results, indexed by all 2-partitions of V
2  sw, σ_opt ← R-PartialScanwidth(V, ∅)
3  return sw, σ_opt
   procedure R-PartialScanwidth(W, R)
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      if |W| = 1 with W = {v} then
5          rpsw ← δ^in(v)
6          σ ← (v)
7      else if |W| > 1 then
8          for each root ρ of G[W] do
9              rpsw'_1, σ' ← R-PartialScanwidth(W \ {ρ}, R ∪ {ρ})
10             rpsw'_2 ← |{uv ∈ E : u ∈ R, v ⇝^{G[W]} ρ}|
11             rpsw' ← max{rpsw'_1, rpsw'_2}
12             if rpsw' < rpsw then
13                 rpsw ← rpsw'
14                 σ ← σ' ∘ (ρ)
15     T(W, R) ← rpsw, σ ;                   // Store result in global table
16     return rpsw, σ
```

We now prove that this algorithm indeed solves SCANWIDTH and improves the time complexity of our recursive algorithm.

Theorem 4.8. *Let $G = (V, E)$ be a weakly connected DAG of n vertices and m arcs, then Algorithm 5 solves* SCANWIDTH *in $O(2^n \cdot m)$ time and exponential space.*

Proof. Correctness: The correctness of the restricted partial scanwidth that is returned by `R-PartialScanwidth` in the algorithm follows directly from Lemma 4.7. The correctness of the corresponding extension is a direct consequence of the constructive nature of the proof of Lemma 4.4, which forms the basis of Lemma 4.7. Definition 4.6 immediately implies that $\text{rpsw}_G(V, \emptyset) = \text{psw}_G(\emptyset, V, \emptyset) = \text{sw}(G)$, and the corresponding extension is also optimal for the scanwidth. Finally, it can easily be seen that the algorithm terminates, as each recursive call is made for a strictly smaller set W. This proves the correctness of the algorithm.

Time complexity: One can implement the algorithm such that the work done in the

subroutine `R-PartialScanwidth`, apart from the recursive call, is bounded by $O(m)$. If $|W| = 1$, this clearly holds. Otherwise, if $|W| > 1$, we employ a smart trick to save time. We first find the components of $G[W]$ in $O(m)$ time (e.g. by a BFS, which takes $O(m+n) = O(m)$ time). Subsequently, we compute the indegrees of the vertex sets of these components, which can also be done in $O(m)$ time. Afterwards, we loop over $O(n) = O(m)$ roots, which can be found in $O(m)$ time by a BFS. For all of these roots, the computation of rpsw'_2 then becomes unnecessary, since this equals the already calculated indegree of the component containing the current root. Hence, this trick prevents us from doing redundant work in the for-loop. Lastly, maximization and appending a vertex to σ are constant-time operations. Overall, we indeed spend $O(m)$ time per recursive call.

Whenever we encounter a 2-partition that has not been calculated, we tabulate the result in the table T. Thus, we enter the subroutine at most once for each ordered 2-partition.[a] There are at most 2^n different (ordered) 2-partitions (each one corresponds to a unique subset W, of which there are at most 2^n), which makes the total time complexity $O(2^n \cdot m)$.

Space complexity: We save a return value and the corresponding extension for all different 2-partitions. Therefore, we surely need exponential space of $\tilde{O}(2^n)$ in the worst case. □

[a] In this implementation we do enter the subroutine but immediately return the result when it already has been calculated.

In the worst case, we can get close to 2^n ordered 2-partitions. Consider for example the graph with one root ρ, one leaf l, and n vertices v_i in between them such that we have arcs (ρ, v_i) and (v_i, l) for each i. In practical instances, however, the number of ordered partitions might be significantly less than 2^n. In the next subsection, we reduce the number of considered partitions even more, although the worst-case scenario remains the same.

4.3.2. Algorithm with component splitting

In this subsection, we will improve upon the basic algorithm by incorporating the notion of *component splitting*. This technique is especially useful when the DAG branches into disconnected parts close to the root. Intuitively, it exploits the fact that when we arrive at some set W, and $G[W]$ is not weakly connected, we can consider the components of $G[W]$ separately. This is in contrast with the basic version of the algorithm, where we unnecessarily would consider all different ways to interleave extensions of the two parts. This idea is formalized in the next lemma.

Lemma 4.9. *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*

$$\text{rpsw}_G(\mathcal{Q}) = \max_{U_i \lhd W} \left\{ \text{rpsw}_G\big((U_i, V \setminus U_i)\big) \right\},$$

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 $i \in \{1, \ldots, r\}$, let $\sigma_i \in \Pi[U_i]$ be arbitrary, and define $\sigma_1 \circ \cdots \circ \sigma_r = \sigma \in \Pi[W]$. This is indeed an extension because the different σ_i are weakly disconnected

in $G[W]$. Using Definitions 4.3 and 4.6, we have:

$$\begin{aligned}
\operatorname{rpsw}(\sigma, \mathcal{Q}) = \max_{w\in W} \operatorname{rpsw}_w^\sigma(\mathcal{Q}) &= \max_{U_i \lhd W}\left\{\max_{w\in U_i} \operatorname{psw}_w^\sigma((\emptyset, W, R))\right\} \\
&= \max_{U_i \lhd W}\left\{\max_{w\in U_i} \operatorname{psw}_w^{\sigma_i}\left(\left(\bigcup\nolimits_{1\le j<i} U_j, U_i, R \cup \bigcup\nolimits_{j>i} U_j\right)\right)\right\} \\
&= \max_{U_i \lhd W}\left\{\max_{w\in U_i} \operatorname{psw}_w^{\sigma_i}\left(\left(\emptyset, U_i, R \cup \bigcup\nolimits_{j\ne i} U_j\right)\right)\right\} \\
&= \max_{U_i \lhd W}\left\{\max_{w\in U_i} \operatorname{psw}_w^{\sigma_i}((\emptyset, U_i, V\setminus U_i))\right\} \\
&= \max_{U_i \lhd W}\left\{\operatorname{rpsw}(\sigma_i, (U_i, V\setminus U_i))\right\}. \qquad (4.2)
\end{aligned}$$

The third equality is similar to the third equality of equation (4.1) that appears in the proof of Lemma 4.4b. 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, $\operatorname{rpsw}_G((U_i, V\setminus U_i)) = \operatorname{rpsw}(\sigma_i, (U_i, V\setminus U_i))$. Now let $\sigma = \sigma_1 \circ \cdots \circ \sigma_r \in \Pi[W]$. Using equation (4.2), we get that

$$\operatorname{rpsw}_G(\mathcal{Q}) \le \operatorname{rpsw}(\sigma, \mathcal{Q}) = \max_{U_i \lhd W}\{\operatorname{rpsw}(\sigma_i, (U_i, V\setminus U_i))\} = \max_{U_i \lhd W}\{\operatorname{rpsw}_G((U_i, V\setminus U_i))\}.$$

($\ge$) We first present a claim. It generalizes the fact that weakly connected vertices in an extension can be swapped without changing the scanwidth. As the claim is quite intuitive, we delay its rather technical proof to Lemma C.2 in Appendix C.

Claim: Let $\sigma \in \Pi[W]$ be such that for some $k \in \{1, \ldots, |W|-1\}$, $\sigma(k)$ 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 $\operatorname{rpsw}(\sigma, \mathcal{Q}) = \operatorname{rpsw}(\pi, \mathcal{Q})$.

Let $\sigma \in \Pi[W]$ be such that $\operatorname{rpsw}_G(\mathcal{Q}) = \operatorname{rpsw}(\sigma, \mathcal{Q})$, which exists by definition. We can also assume that $\sigma = \sigma_1 \circ \cdots \circ \sigma_r$, 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. The inequality now quickly follows from equation (4.2):

$$\operatorname{rpsw}_G(\mathcal{Q}) = \operatorname{rpsw}(\sigma, \mathcal{Q}) = \max_{U_i \lhd W}\{\operatorname{rpsw}(\sigma_i, (U_i, V\setminus U_i))\} \ge \max_{U_i \lhd W}\{\operatorname{rpsw}_G((U_i, V\setminus U_i))\}.$$

□

With this lemma, we can integrate the component splitting in the algorithm. We achieve this by first checking if the subgraph $G[W]$ is weakly connected. If it is, we make a recursive call for each of its components. This modification is reflected in the updated Algorithm 6.

Algorithm 6: Dynamic programming algorithm with component splitting to solve SCANWIDTH.

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

The worst-case time complexity of the algorithm remains $O(2^n \cdot m)$, as it depends on the exact graph how often the component-splitting is used. For a graph with a single leaf, this new algorithm is no better than before, as there are no components to split. On the other hand, on a very tree-like graph, the component splitting makes sure that the number of sets W to be considered is a lot less. Thus, in practice, this technique is likely to decrease the algorithm's running time. The following theorem formalizes the complexity and correctness of the algorithm.

Theorem 4.10. *Let $G = (V, E)$ be a weakly connected DAG of n vertices and m arcs, then Algorithm 6 solves* SCANWIDTH *in $O(2^n \cdot m)$ time and exponential space.*

Proof. Correctness: The proof of correctness is analogous to the proof of Theorem 4.8, with the addition of Lemma 4.9. The lemma ensures the correct restricted partial scanwidth (and corresponding extension) is returned in the first if-statement. It should be noted that the algorithm still terminates, as we only enter the first if-statement whenever $G[W]$ is not weakly connected. This prevents the algorithm from looping through this if-statement indefinitely.

We only enter the second 'else if'-statement for sets W with $G[W]$ weakly connected.

The remark from Lemma 4.7 then allows us to write $\delta^{\text{in}}(W)$ on line 15 of the algorithm, instead of the more elaborate notation used on line 10 of Algorithm 5.

Time complexity: With a BFS, we can check in $O(m)$ time whether $G[W]$ is weakly connected. If it is, the work done in `R-PartialScanwidthCS`, apart from the recursive calls, takes $O(m)$ time. In this case, the subroutine is identical to the one in Algorithm 5, which also took $O(m)$ time (see proof of Theorem 4.8). If $G[W]$ is not weakly connected, we can find its components by a BFS in $O(m)$ time. The number of components is at most n, and the maximization and concatenation are also $O(n) = O(m)$ time operations. Thus, the non-recursive part of the subroutine then also takes $O(m)$ time.

We still only enter the subroutine at most once for each ordered 2-partition, because we tabulate the results. These partitions are still bounded by 2^n, which makes the total time complexity again $O(2^n \cdot m)$.

Space complexity: By the same reasoning as in Theorem 4.8 the space complexity remains exponential. □

Component splitting is not only of practical interest. In the next subsection, it turns out to be essential for further bounding the time complexity when computing the scanwidth of a DAG.

4.3.3. Algorithm for fixed scanwidth

In this subsection, we consider the fixed parameter version of SCANWIDTH. Formally, this can be written as:

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

The key idea to solve this problem is that we only need to consider sets W that are weakly connected (by the component splitting technique from the previous section) and that have an indegree of at most k. We now show a corollary derived from Lemmas 4.7 and 4.9, which acts as a formal foundation of this idea.

Corollary 4.11. *Let $G = (V, E)$ be a weakly connected DAG, $k \geq 1$ an integer and $\mathcal{Q} = (W, R)$ an ordered 2-partition of V such that $W \neq \emptyset$ and $G[W]$ is weakly connected. Then,*

(a) $\text{rpsw}_G(\mathcal{Q}) > k$*, if and only if* $\text{rpsw}_G((U_i, V \setminus U_i)) > k$ *for some component $G[U_i]$ of $G[W]$.*

Furthermore,

(b) For $|W| = 1$, $\text{rpsw}_G(\mathcal{Q}) > k$*, if and only if $\delta^{\text{in}}(W) > k$.*

(c) For $|W| \geq 2$, $\text{rpsw}_G(\mathcal{Q}) > k$*, if and only if $\delta^{\text{in}}(W) > k$ or* $\text{rpsw}_G((W \setminus \{\rho\}, R \cup \{\rho\})) > k$ *for all roots ρ of $G[W]$.*

Proof. Part (a) is an immediate consequence of Lemma 4.9, while part (b) follows from Lemma 4.7a.

Regarding part (c), we will use Lemma 4.7b. Note that because we assume $G[W]$ to be weakly connected, the remark from that lemma tells us that we can use $\delta^{\text{in}}(W)$ instead of the lengthier set notation. Furthermore, recall that $P(G[W])$ denotes the set of roots of

$G[W]$. We then get

$$\begin{aligned}\text{rpsw}_G(\mathcal{Q}) &= \min_{\rho \in P(G[W])} \max\left\{\text{rpsw}_G((W \setminus \{\rho\}, R \cup \{\rho\})), \delta^{\text{in}}(W)\right\} \\ &= \max\left\{\min_{\rho \in P(G[W])} \text{rpsw}_G((W \setminus \{\rho\}, R \cup \{\rho\})), \delta^{\text{in}}(W)\right\}.\end{aligned}$$

Here, we used that $\delta^{\text{in}}(W)$ is independent of the choice of ρ. Part (c) of the lemma now follows. □

Using this corollary, we can transform Algorithm 6 into a fixed parameter version: Algorithm 7. The core of the algorithm remains unchanged. We only incorporate checks whether the conditions of Corollary 4.11 are satisfied. This ensures that we return ∞ whenever the scanwidth is larger than k.

Algorithm 7: Dynamic programming algorithm with component splitting to solve k-SCANWIDTH.

Input: Weakly connected DAG $G = (V, E)$, integer $k \geq 1$.
Output: If $\text{sw}(G) \leq k$: scanwidth sw of G and an optimal extension σ_{opt}. If $\text{sw}(G) > k$: ∞ and an incomplete extension.

```
1  T ← empty table to tabulate results, indexed by all 2-partitions of V
2  sw, σ_opt ← k-R-PartialScanwidthCS(V, ∅, k)
3  return sw, σ_opt
   procedure k-R-PartialScanwidthCS(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      if G[W] is not weakly connected then
5          for each weakly connected component G[U_i] of G[W], (i = 1, ..., r) do
6              rpsw_i, σ_i ← k-R-PartialScanwidthCS(U_i, V \ U_i)
7          rpsw ← max{rpsw_i : i = 1, ..., r}
8          σ ← σ_1 ∘ ··· ∘ σ_r
9      else if |W| = 1 with W = {v} and δ^in(W) ≤ k then
10         rpsw ← δ^in(v)
11         σ ← (v)
12     else if |W| > 1 and δ^in(W) ≤ k then
13         for each root ρ of G[W] do
14             rpsw'_1, σ'_1 ← k-R-PartialScanwidthCS(W \ {ρ}, R ∪ {ρ})
15             rpsw' ← max{rpsw'_1, δ^in(W)}
16             if rpsw' < rpsw then
17                 rpsw ← rpsw'
18                 σ ← σ'_1 ∘ (ρ)
19     T(W, R) ← rpsw, σ ;                        // Store result in global table
20     return rpsw, σ
```

Before Theorem 4.14 formally proves 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.

For the first lemma, we define what antichains are. 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.

Lemma 4.12. *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 $\rho_1 <_G \rho_2$. 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 else 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 $\mathcal{A}$.

We now prove that f is surjective. Let S be an antichain in $\mathcal{A}$. Then define U as the set of all vertices v 'below' S or in S (i.e. $v \leq_G s$ for some $s \in S$). We then have that U is a sinkset of G, and the roots of $G[U]$ are exactly the set S. Thus $f(U) = S$ and f is surjective.

For the injectivity, let $W_1, W_2 \in \mathcal{V}$ be such that $f(W_1) = f(W_2)$, i.e. $P(G[W_1]) = P(G[W_2]) = P$. Let $v \in V$ be arbitrary. We will show by a case analysis that v is either in both W_1 and W_2, or in neither of them. This must then mean that $W_1 = W_2$, showing that f is injective. We now consider the four cases. *(i)* If $v \in P$, then by definition v is also in W_1 and W_2. *(ii)* If $v <_G r$ (for some $r \in P$), v must be in W_1 and W_2, else they would not be sinksets. *(iii)* If $v >_G r$ (for some $r \in P$), then v can not be in W_1 and W_2, because then either they are no sinksets any more, or r is not a root any more. *(iv)* If v is incomparable to all $r \in P$ (and $v \notin P$), then v is not in W_1 and W_2. Else, v would also be a root, and thus part of P.

All in all, f is surjective and injective. This proves 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 are able to bound the number of sinksets with a bounded indegree.

Lemma 4.13. *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 $\delta^{\text{in}}(W) \leq k$, is bounded from above by n^{k+r-1}.*

Proof. Let $k \geq 1$ be arbitrary and $\mathcal{V}_k = \{W \sqsubseteq V : \delta^{\text{in}}(W) \leq k\}$. We first prove a claim relating the indegrees of the sets W to the number of roots of $G[W]$.

Claim: 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 (else, 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 4.12, 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 4.12, 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, else this will contradict the injectivity of f.[a] 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.

Base case: ($\ell = 1$). We have that $|\mathcal{A}_1| \leq n$, as each such vertex-antichain contains one vertex.

Induction step: ($\ell \geq 1$). First, assume that the induction hypothesis holds for ℓ, so $|\mathcal{A}_\ell| \leq n^\ell$. Now note that all vertex-antichains of size exactly $\ell + 1$ can be created from the antichains of size ℓ (of which we have at most n^ℓ by the induction hypothesis), 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

$$|\mathcal{A}_{\ell+1}| \leq n^\ell + n^\ell \cdot (n - \ell) = n^{\ell+1} - (\ell - 1)n^\ell \leq n^{\ell+1}.$$

□

[a] Note that h is not necessarily surjective, as graphs can have sinksets with a larger indegree than their number of roots.

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 correctness and time complexity of Algorithm 7 in the following theorem. We emphasize that the $O(2^n \cdot m)$ bound on the time complexity from the previous versions of the algorithm is still applicable here.

Theorem 4.14. *Let $G = (V, E)$ a weakly connected DAG of n vertices, m arcs, r roots and maximum outdegree Δ^{out}, and let $k \geq 1$ be an integer. Then, Algorithm 7 solves k-SCANWIDTH in $O(\Delta^{\text{out}} \cdot (k + r - 1) \cdot n^{k+r-1} \cdot m)$ time and $O(\Delta^{\text{out}} \cdot (k + r - 1) \cdot n^{k+r})$ 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 `k-RestrictedPartialScanwidth` now follows from the correctness of the previous Algorithm 6 (see Theorem 4.10) and the new Corollary 4.11, which precisely defines when ∞ should be returned.

When $G[W]$ is not weakly connected and for some component $G[U_i]$ the restricted partial scanwidth is larger than k, the maximization on line 7 will ensure we correctly return

∞. This is the correct behaviour due to Corollary 4.11a. Similarly, the last if-statement will return ∞, whenever the rpsw is larger than k (i.e. equal to ∞) for all the roots appearing in the for-loop. This correctly mimics the second condition of Corollary 4.11c. Lastly, if $\delta^{\text{in}}(W) > k$ and $G[W]$ is weakly connected, no if-statement is entered. We then correctly return the initial value of ∞, in accordance with Corollary 4.11b and the first condition of Corollary 4.11c. The correctness of the complete algorithm is now an immediate consequence of the correctness of the subroutine.

Time complexity: Due to the tabulation, we still enter the subroutine at most once for each of the at most 2^n sinksets W. It can be checked that the subroutine is only called for sinksets W of the following types:

1. Weakly connected sinksets W with $\delta^{\text{in}}(W) \leq k$.
2. Weakly connected sinksets W with $\delta^{\text{in}}(W) > k$, created by deletion of one root of a type 1 sinkset. (These sinksets are created in the last for-loop, and they do not create new recursive calls.)
3. Weakly disconnected sinksets W, created by the deletion of one root of a type 1 sinkset. (These sinksets are created in the last for-loop.)
4. Weakly connected sinksets W with $\delta^{\text{in}}(W) > k$, created by splitting a type 2 sinkset. (These sinksets are created in the first if-statement, and they also do not create new recursive calls.)

Note that we sloppily refer to the roots of a sinkset W, where we mean the roots of $G[W]$.

According to Lemma 4.13, there are at most n^{k+r-1} type 1 sinksets. Type 2 sinksets are created from type 1 sinksets by deletion of a root. As stated in the claim of the proof of Lemma 4.13, type 1 sinksets have at most $k+r-1$ roots, yielding at most $(k+r-1) \cdot n^{k+r-1}$ type 2 sinksets. Similarly, there are at most $(k+r-1) \cdot n^{k+r-1}$ type 3 sinksets. Lastly, type 4 sinksets are created by splitting a type 2 sinkset into its components. Since the maximum outdegree of G is Δ^{out}, type 2 sinksets can exist of at most Δ^{out} components. This is because they are formed by deleting one root of a weakly connected sinkset, and this root had an outdegree of at most Δ^{out}. Consequently, there can be at most $\Delta^{\text{out}} \cdot (k+r-1) \cdot n^{k+r-1}$ type 4 sinksets.

We now consider the time that is spent within each subroutine call. In Theorem 4.10 we argued that this takes $O(m)$ time for Algorithm 6. Apart from checking the indegrees of the sinksets, which also takes $O(m)$ time, there have been no significant changes compared to that algorithm. Since the number of sinksets that is considered is bounded by $O(\Delta^{\text{out}} \cdot (k+r-1) \cdot n^{k+r-1})$, the time complexity of the algorithm becomes $O(\Delta^{\text{out}} \cdot (k+r-1) \cdot m \cdot n^{k+r-1})$.

Space complexity: Regarding the space complexity, we store an extension of size $O(n)$ and a value rpsw, for each considered sinkset. This requires $O(\Delta^{\text{out}} \cdot (k+r-1) \cdot n^{k+r})$ 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 rooted DAGs. Our algorithm then functions as a slicewise polynomial algorithm when considering scanwidth as the parameter.

Corollary 4.15. *Let $G = (V, E)$ be a weakly connected rooted DAG with n vertices, m arcs, fixed maximum outdegree, and a scanwidth of k. Then, there exists an algorithm that solves* SCANWIDTH *in $O(k \cdot m \cdot n^k)$ time and $O(n^{k+1})$ space. Thus, for rooted DAGs* SCANWIDTH *is in* XP *when considering*

the scanwidth as the parameter.

Remark. At the cost of a factor n in both the time and space complexity, the fixed maximum outdegree constraint can be released.

Proof. By repeatedly running Algorithm 7 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 of Theorem 4.14 is still applicable here. Substituting $r = 1$, and using that the maximum outdegree is fixed, we obtain the desired complexities from this theorem. This directly proves the XP result stated in the corollary. The remark follows from the fact that $\Delta^{\text{out}} \leq n$, and filling this in into Theorem 4.14. □

In Subsection 3.3.3 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 obtain an FPT algorithm when considering the level as a parameter. This allows us to formulate another complexity result, proving that for networks SCANWIDTH is in FPT when considering the level as a parameter.

Corollary 4.16. *Let $G = (V, E)$ be a level-k network of n vertices and m arcs. Then, there exists an algorithm that solves* SCANWIDTH *in $O(2^{4k-1} \cdot k \cdot n + n^2)$ time. Thus, for networks* SCANWIDTH *is in* FPT *when considering the level as the parameter.*

Proof. As mentioned earlier, the algorithm described in the previous corollary still has its time complexity bounded by $O(2^n \cdot m)$. This is true since it still considers at most 2^n sinksets and spends $O(m)$ time per such set. When combined with decomposition algorithm 3, we can then solve SCANWIDTH in $O(n^2 + n \cdot m' \cdot 2^{n'})$ time, according to Lemma 3.28.[a] Here, n' (resp. m') is the maximum number of nodes (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 $4k - 1$ vertices, and at most $5k - 2$ arcs, according to Lemma 3.29. Substituting these numbers for n' and m' gives the desired result. □

[a] Algorithms 5 and 6 can also be used for this result, as they too have their time complexity bounded by $O(2^n)$. Because we formulated Lemma 3.28 in a very general way, even the recursive algorithm 4 can function as an FPT algorithm (when combined with the decomposition algorithm), albeit with a worse time complexity.

Another interpretation of this corollary is that if we fix the level, SCANWIDTH can be solved in quadratic time on networks.

In combination with the decomposition algorithm, it is even possible to create an algorithm with a doubly parametrized time complexity for networks of fixed degree. As networks are by definition rooted, combining the proofs of the two previous corollaries would then result in an algorithm with a time complexity of $O(k \cdot \ell \cdot (4\ell - 1)^k)$. Here, k refers to the scanwidth, and ℓ to the level. It depends on the specific values of k and ℓ whether this is smaller than the previously stated bounds.

Chapter 5

Heuristics

Strengthened in our belief by the successful use of sub-optimal solution methods for other width parameters [DPS02], we now divert our attention to heuristics. Contrary to the exact methods of the previous chapter, heuristic algorithms are not guaranteed to find an optimal (tree) extension. Instead, these algorithms are meant to have faster computation times, while still producing reasonable solutions. In Section 5.1 we start with the simplest of heuristics: a *greedy algorithm*. Section 5.2 explores a different idea: the repeated cutting of a graph. Lastly, we attempt to enhance these heuristics with *simulated annealing* in Section 5.3.

5.1. Greedy heuristic

An intuitive idea to create an extension of seemingly small scanwidth is by adding vertices one by one, each time adding the vertex that increases the scanwidth the least. This approach of making locally optimal choices is widely known as a *greedy* approach.

To apply a greedy approach to our SCANWIDTH problem, it seems logical to keep track of the vertices we can add to the extension. We can then calculate what effect each vertex has on the scanwidth and choose the best option. We continue this process until we have a full extension of the graph. This gives rise to the following greedy heuristic:

Algorithm 8: Greedy heuristic to find an extension.

Input: Weakly connected DAG $G = (V, E)$.
Output: Extension σ and scanwidth sw of σ.

```
1  initialize
2  |  S ← V
3  |  σ ← (); sw ← 0
4  while |S| > 0 do
5  |  initialize x ← None; sw_x ← ∞
6  |  for each leaf ℓ of G[S] do
7  |  |  s ← |{uv ∈ E : u ∈ S \ {ℓ}, v ∈ σ ∪ {ℓ}, ℓ ⇝^{G[σ ∪ {ℓ}]} v}|;     // Scanwidth at ℓ
8  |  |  if s < sw_x then
9  |  |  |  x ← ℓ ; sw_x ← s
10 |  S ← S \ {x}
11 |  σ ← σ ∘ (x)
12 |  sw ← max{sw, sw_x}
13 return σ, sw
```

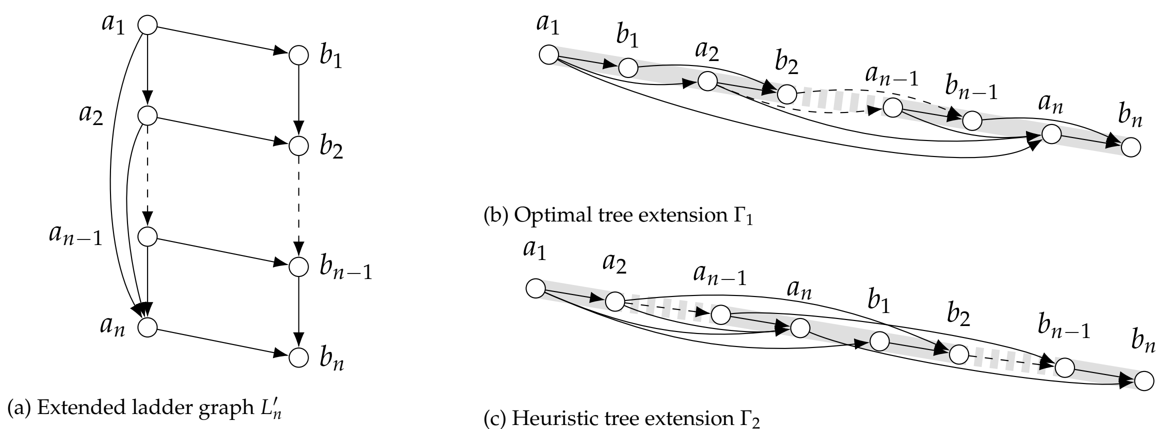
The algorithm maintains a set S to keep track of the vertices that still need to be added. In each iteration, it chooses a leaf of $G[S]$ as the next vertex in the extension. The choice of the leaf is not arbitrary, as the algorithm calculates the scanwidth that each of those leaves would have, and adds the best one to the extension. This procedure is repeated until the set S is empty (or equivalently, σ contains all vertices of G). Note that we slightly abuse notation in the algorithm by writing $\sigma \cup \{\ell\}$, where σ is not a set but an extension. The following theorem shows that this greedy algorithm always returns an extension and that it runs in polynomial time.

Theorem 5.1. *Let $G = (V, E)$ be a weakly connected DAG of n vertices and m arcs. Then, Algorithm 8 returns an extension and its corresponding scanwidth in $O(n^2 \cdot m)$ time.*

Proof. Correctness: Note that S keeps track of the vertices of which no order has been established yet, while σ is the intermediate extension. Clearly, if we only add leaves of the graph $G[S]$ in each iteration, we never get arcs that point the wrong way in the extension. Furthermore, the value of the scanwidth at each vertex is exactly as in Definition 2.2. This proves correctness.

Time complexity: The while loop adds a vertex to σ in each iteration, and thus runs n times. The for-loop certainly runs at most n times, while the other stuff in the while loop takes constant time. Calculating the value of s can be implemented to run in $O(m)$ time. To this end, we first keep track of the weakly connected components of $G[T \cup \{l\}]$, which can be done by a BFS in $O(n + m) = O(m)$ time. We then check for each of the m arcs, whether they contribute towards the value of s. All in all, the algorithm then runs in $O(n^2 \cdot m)$ time. □

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. However, the greedy rule will not necessarily pick the correct leaf. One can construct instances where the algorithm will perform very badly. An example is depicted in Figure 5.1. The figure shows a class of graphs that all have scanwidth 5, yet the greedy algorithm will construct a solution of scanwidth n. Here, n is half the number of nodes in the graph.



(a) Extended ladder graph L'_n

(b) Optimal tree extension Γ_1

(c) Heuristic tree extension Γ_2

Figure 5.1: (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. Cut-splitting heuristic

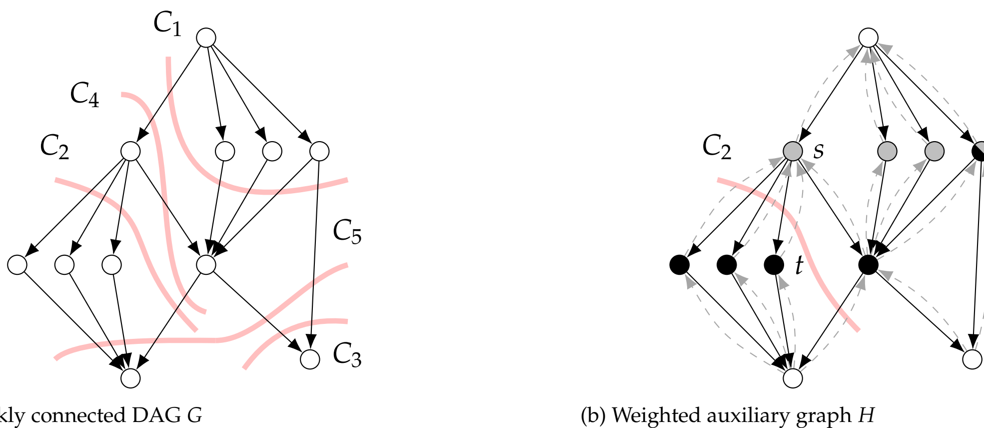
An important observation is that the scanwidth of an extension corresponds to the size of an arc-cut of the graph. In this section, we leverage this observation to develop another heuristic. Instinctively, it makes sense to search for a small arc-cut in the graph and then use that cut to split into two subproblems. Subsection 5.2.1 takes a closer look at the type of cuts that appear in a (tree) extension. In Subsection 5.2.2 we will use these *DAG-cuts*, and develop the corresponding heuristic idea.[1]

5.2.1. DAG-cuts

Let us first review some standard terminology of 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, which we will show shortly are the exact cuts that appear in extensions. To illustrate the definition, see Figure 5.2a.

Definition 5.2 (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.*



(a) Weakly connected DAG G

(b) Weighted auxiliary graph H

Figure 5.2: (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 (C_3 is namely a cut with a cut-set that is contained in the one 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 nodes are in the set U and the black nodes 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 $t \in W$.

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

[1] The algorithmic idea to solve SCANWIDTH by repeatedly cutting at DAG-cuts stems from an early thought by Mathias Weller [Wel23], although we later independently developed the framework and whole theory around it. Weller originally named DAG-cuts, *antichain edge-cuts*. We opt for a different name to not confuse them with the vertex-antichains used in Subsection 4.3.3.

Proposition 5.3. *Let $G = (V, E)$ be a weakly connected DAG. 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 of an extension σ of G.*

Proof. ($\Rightarrow$) Let F be the cut-set of some DAG-cut $C = (S, T)$. By Definition 5.2, 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. Else, F would contain a smaller cut-set. T is also a sinkset, thus we can create an extension σ that first has the vertices in T, and then those in S. Using that $G[T]$ was weakly connected, we have that $\mathrm{SW}^\sigma_{|T|} = F$.

($\Leftarrow$) Let σ be an extension of G and i 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 set $S = V \setminus T$.[a] 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 $C = (S, T)$. Because 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. □

[a]We implicitly exclude the trivial case that G is just a single vertex.

Although trivial DAG-cuts also appear as cuts in an extension, they are less interesting. As we will explain in more detail in the next subsection, our main interest lies in the non-trivial DAG-cuts, specifically in the smallest such cut(s). It is far from obvious how to find the minimum-weight non-trivial DAG-cuts, but the following lemma will enlighten us. 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 [RAK91], who employed this technique for the closely related *DAG edge-separators*.

Lemma 5.4. *Let $G = (V, E, w)$ be a weighted, weakly connected DAG with $w : E \to \mathbb{R}_{>0}$. 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 infinity 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$G$.

($a, \Leftarrow$) First note that 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, else T is 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 has a finite weight-function, implying that all its cuts have finite weight. □

To illustrate the sets U and W in this lemma, as well as the correspondence between a DAG and its auxiliary graph, we refer to Figure 5.2b. Note that instead of infinity, any general upper bound on the largest DAG-cut, such as $|E| + 1$, is sufficient.

Using this lemma, we are able to devise Algorithm 9 which finds a minimum-weight non-trivial DAG-cut in polynomial time.

Algorithm 9: Minimum-weight non-trivial DAG-cut.

Input: Weighted weakly connected DAG $G = (V, E, w)$ with finite weights.
Output: If a non-trivial DAG-cut in G exists: a minimum-weight non-trivial DAG-cut C. Else: `None`.

```
1  initialize C ← None
2  U ← set of children of the roots of G
3  W ← set of parents of the leaves of G
4  H ← G with reverse arcs of infinite weight added, resulting in weight-function w'
5  for each s ∈ U do
6      for each t ∈ W \ {s} do
7          C' ← minimum weight directed s-t cut in H ;      // Use any known algorithm.
8          if w'(C') < w'(C) then                          // If C is None, we let w'(C) = ∞.
9              C ← C'
10 return C
```

This algorithm is a direct application of Lemma 5.4. As a side note, we mention that the algorithm can be modified to find the smallest DAG-cut (including trivial cuts), by letting the set U (resp. W) consist of all roots (resp. leaves) of G. The previous lemma can then be trivially adapted. Correctness of the algorithm is now proved in the next theorem.

Theorem 5.5. *Let $G = (V, E, w)$ be a weighted, weakly connected DAG of n vertices and m arcs. Then, Algorithm 9 can certify whether a non-trivial DAG-cut of G exists, and if it does, it finds a minimum-weight non-trivial DAG-cut of G. The algorithm can be implemented to run in $O(n^3 \cdot m)$ time.*

Proof. Correctness: The correctness of the algorithm is a consequence of Lemma 5.4. The graph H that is constructed is identical to the graph in this lemma, and the sets U and W are also the same. The minimum-weight cuts in H that we find in the algorithm are always minimal, otherwise, there would be a cut of smaller weight. The algorithm returns `None`, if and only if no s-t cut exists with finite weight. According to part (b) of the lemma this is the case only when no non-trivial DAG-cut exists at all. Therefore, `None` is returned correctly.[a]

If there does exist some s-t cut with finite weight, part (a) of the lemma ensures correctness. We can then check for all combinations of s and t, what the minimum-weight s-t cut is in H. By the lemma, a smallest s-t cut (which must have finite weight) corresponds to a minimum-weight non-trivial DAG-cut in G. Thus, the algorithm correctly identifies a minimum-weight non-trivial DAG-cut when it exists.

Time complexity: The famous 'max-flow min-cut theorem' states that finding a minimum weight directed s-t cut in a directed graph is equivalent to finding a maximum

capacity s-t flow in the same graph [FF56]. The fastest known algorithm for general directed graphs, and only parametrized by the number of vertices n and the number of arcs m, is described by Orlin [Orl13]. He combines his own algorithm with one by King, Rao and Tarjan [KRT94] to obtain a time complexity of $O(n \cdot m)$. Therefore, finding the minimum weight directed s-t cut can be done in $O(n \cdot m)$ time.

The double for loop could result in $O(n^2)$ combinations of s and t, thus in total this part of the algorithm can be implemented to run in $O(n^3 \cdot m)$ time. The creation of the sets U, W, and the construction of the graph H, are also dominated by this time complexity, proving the result. □

[a] For readability the lemma inexplicitly assumes that we do not have $U = W = \{s\}$ and thus $W \setminus \{s\} = \emptyset$, but it is trivial to see that in that case `None` is also returned correctly.

It is not unthinkable that the above running time can be improved by analyzing the workings of minimum s-t cut algorithms. Most of these algorithms rely on flow operations, and since we iterate over $O(n^2)$ source-sink combinations, it could be possible that some computations are done twice. Even though further optimizations may thus be possible, the current algorithm provides a straightforward and effective approach to finding minimum-weight non-trivial DAG-cuts.

5.2.2. Repeated DAG-cut-splitting heuristic

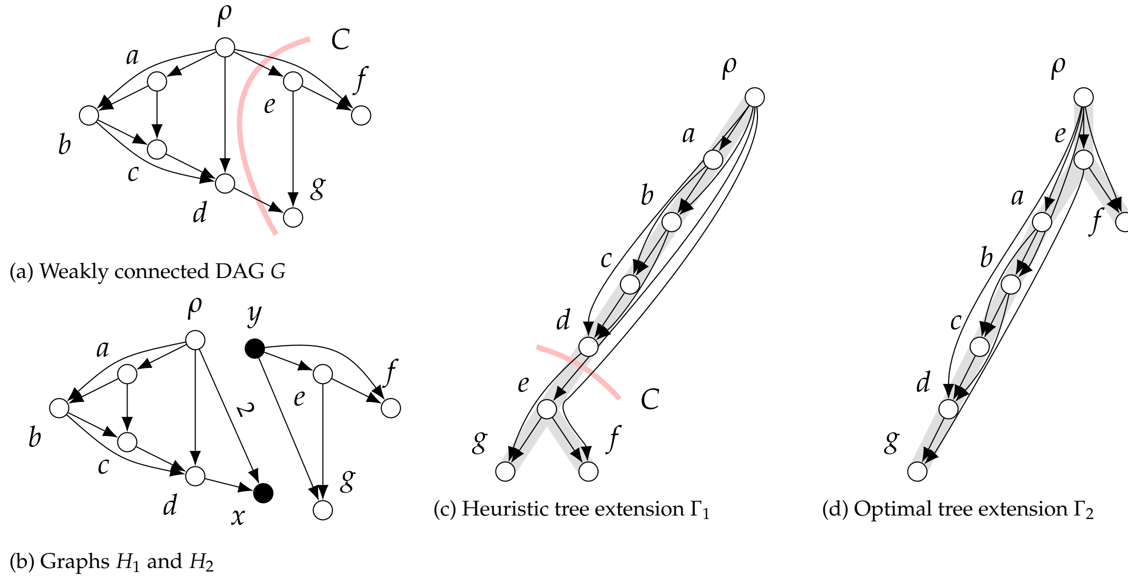
The idea behind our heuristic is to recursively split the graph at a smallest non-trivial DAG-cut. Consequently, we obtain an upper and a lower subgraph. However, when considering scanwidth we can not just 'forget' about the arcs in the cut, as they might also be counted at vertices lower or higher in 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: else, 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 just take any extension. This leads to the following heuristic:

Algorithm 10: 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     C = (S, T) ← minimum-weight non-trivial DAG-cut of H, using Algorithm 9
2     if C is None then
3        σ ← any extension of H ;        // Use e.g. reverse DFS or BFS traversal.
4     else
5        H_1 ← H[S]; H_2 ← H[T]
6        add a 'superleaf' x to H_1 and a 'superroot' 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_1[S] ∘ σ_2[T]
13    return σ
```

In Figures 5.3a and 5.3b the first iteration of the algorithm is visualized. Figure 5.3c shows the canonical tree extension corresponding to the extension resulting from the algorithm. We indeed see the cut C reappearing. The graph from this figure also demonstrates that the algorithm can indeed be sub-optimal. Specifically, Figure 5.3d shows the optimal tree extension, which has a smaller scanwidth than the one in Figure 5.3c. Thus, in general, it is not necessarily true that the smallest non-trivial DAG-cut appears in an optimal extension.



(a) Weakly connected DAG G

(b) Graphs H_1 and H_2

(c) Heuristic tree extension Γ_1

(d) Optimal tree extension Γ_2

Figure 5.3: (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 10. 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 10. 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.

Nonetheless, the algorithm could still be valuable in practice, since it can be applied efficiently to larger instances. This is formalized in the following theorem, where it is proved that the algorithm has a polynomial running time, and thus serves as a suitable candidate for a heuristic.

Theorem 5.6. *Let $G = (V, E)$ be a weakly connected DAG of n vertices and m arcs. Then, Algorithm 10 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 $k \in \{1, 2, 3\}$, we can never have 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 or DFS traversal. This proves the base case.

Induction step: Let $k \geq 4$ be arbitrary, and assume that the statement holds for all $1 \leq \ell \leq k-1$. We will show it then also holds for k. We can furthermore assume that H has a non-trivial DAG-cut, else the procedure will automatically return an extension. Note that 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 $\sigma = \sigma_1[S] \circ \sigma_2[T]$ must be an extension of H. This concludes the induction.

Time complexity: The algorithm terminates, since the recursive calls are made on strictly smaller graphs (see the reasoning in the above induction proof). According to Theorem 5.5, finding the minimum non-trivial DAG-cut can be done in $O(n^3 \cdot m)$ time with Algorithm 9. As the other non-recursive parts in the procedure are also dominated by this complexity, $O(n^3 \cdot m)$ time is spent per subroutine call.

It is fairly obvious to see that - as with other recursive split algorithms such as *quicksort* - the worst-case time complexity is obtained when the splits are far from balanced. In the absolute worst case, we repeatedly cut off a graph of size 2 and one of size $n - 2 + 1 = n - 1$. The graph of size 2 (which has size 3 after adding the supervertex), creates no further recursive calls. All in all, the non-recursive part is then executed $O(n)$ times. This results in a worst-case time complexity of $O(n^4 \cdot m)$. □

5.3. Simulated annealing

Following other successful attempts of simulated annealing algorithms applied to width-parameters [SC21] and phylogenetics [Esl+12], we will try to set up a simulated annealing framework for scanwidth. In Subsection 5.3.1 we introduce what a neighbourhood is and how one can apply this to the space of (tree) extensions. Afterwards, we are ready to state the algorithm in Subsection 5.3.2. In the last part of this chapter, Subsection 5.3.3, we will discuss a strategy to find values for the parameters of the heuristic, along with a convergence result.

5.3.1. Neighbourhood of (tree) extensions

Before we dive into the concept of simulated annealing, we introduce the general notion of a neighbourhood. One can think of a neighbourhood as a mapping from one element in a solution space to a set of other solutions, thus formalizing the idea of *neighbouring solutions*. To prove a convergence result in the next subsection, we will also cover a more restricted type of neighbourhood.

Definition 5.7 (Neighbourhood). *Let Σ be a set of elements. A mapping $\Phi : \Sigma \to 2^\Sigma$ that maps each element of Σ to a subset of Σ is a* neighbourhood *on Σ. For two elements $i, j \in \Sigma$ such that $i \in \Phi(j)$, we say that i is a neighbour of j. Furthermore, Φ is a* proper neighbourhood *on Σ, if:*

1. *for all $i \in \Sigma$ it holds that $i \notin \Phi(i)$;*
2. *for all $i \neq j \in \Sigma$ it holds that $i \in \Phi(j)$, if and only if $j \in \Phi(i)$;*
3. *for all $i \neq j \in \Sigma$ there exist a finite (possibly empty) sequence $i_1, \dots, i_\ell \in \Sigma$ such that $i \in \Phi(i_1), i_1 \in \Phi(i_2), \dots, i_\ell \in \Phi(j)$.*

The above definition (adapted from [CB00]) states that a neighbourhood is proper, if elements are not neighbours of themselves (criterion 1), if being a neighbour is a two-way relationship (criterion 2), and if it is possible to get from each element to any other element by moving through finitely many neighbours (criterion 3). This last criterion is especially important. It tells us that we can never get stuck in a set of solutions by moving only through neighbours. Therefore, a (proper) neighbourhood provides a structured way to explore and navigate a solution space. Hence, the notion of a (proper) neighbourhood is widely used in optimization algorithms.

In our context, we aim to create a proper neighbourhood in the space of extensions or tree extensions. A good initial candidate would be to let the neighbourhood of an extension

consist of all extensions that can be reached by swapping two consecutive vertices in the extension. One needs to be careful, however, since only two vertices that are not adjacent can be swapped. Else, the ordering is not an extension any more. We now prove that this approach indeed leads to a proper neighbourhood.

Lemma 5.8. *Let $G = (V, E)$ be a weakly connected DAG. For all extensions σ of G and all $v <_\sigma u$, denote by $\mathrm{move}_{v,u}(\sigma)$ the ordering we obtain from σ after moving the vertex v to directly after u in σ. Then, $\Phi_{\mathrm{swap}}(\sigma) = \{\mathrm{move}_{v,u}(\sigma) : uv \notin E, v = \sigma(i) \text{ and } u = \sigma(i+1) \text{ for some } i < |V|\}$ defines a proper neighbourhood on the set of extensions σ of G.*

Proof. Any extension where we swap two consecutive vertices in the extension such that there is no arc between them, is naturally an extension again. Therefore, Φ_{swap} is a neighbourhood on the set of extensions.

To establish that Φ_{swap} is a proper neighbourhood, we need to verify the tree criteria from Definition 5.7. It is obvious that the first two criteria hold. For the third criterion let σ and π be two distinct extensions of G. We will now prove by induction on the number of vertices n of G, that one can transform σ into π in a finite number of steps using Φ_{swap}. We can assume that $n \geq 3$, as all weakly connected DAGs with at most 2 vertices have only one extension.

Base case: For $n = 3$, there are only two possible scenarios when G has at least two unique extensions: either G consists of one root and two leaves, or it has two roots and one leaf. In both cases, G has exactly two possible extensions, which are direct swap-neighbours of each other. This immediately proves the base case.

Induction step: Let $n \geq 4$ be arbitrary and assume that the statement holds for a graph of $n-1$ vertices. Let $x = \sigma(1)$ and $y = \pi(1)$. As y is the first vertex in π, it can not be above any other vertex in G, showing that y is a leaf. But then, if $y \neq x$, we can swap y all the way down to the last position in σ to obtain an extension σ'. On the other hand, if $y = x$, it means that y is already in the last position, and we simply have $\sigma = \sigma'$. Either way, using a (possibly empty) sequence of swaps, we have now obtained an extension σ' with $\sigma'(1) = \pi(1) = y$.

Let $H = G[V \setminus \{y\}]$ which has $n-1$ vertices. It is evident that $\sigma'[2 \dots n]$ and $\pi[2 \dots n]$ are extensions of H. By the induction hypothesis, we can create $\pi[2 \dots n]$ from $\sigma'[2 \dots n]$ by a sequence of swaps.[a] As y was a leaf, its deletion does not affect any order relations between other vertices. Thus, these swaps are also valid in G, and turn $\sigma'[1 \dots n]$ into $\pi[1 \dots n]$. Altogether, we have turned σ into σ', and σ' into π, with a finite amount of swaps. This concludes the induction proof and thus proves the lemma. □

[a] If H is not weakly connected any more, one can do this for each component separately. As the components are not connected, one can then interleave the extensions of the components by only using swaps again. Thus, this poses no difficulty.

At first sight, this seems to be a good choice of neighbourhood. Upon closer inspection, however, the neighbourhood may result in a large number of neighbours of an extension that have the same canonical tree extension as the original extension. This occurs, since swapping two vertices that are not yet weakly connected in the extension, will not change the corresponding canonical tree extension. As an example, imagine a DAG that is a very wide directed tree with a single root. Here, the DAG has a plethora of extensions, while they all have the same canonical tree extension (namely the tree itself). This redundancy in neighbours can lead to unnecessary computations.

We circumvent this problem by opting for a neighbourhood on the space of canonical tree extensions. This way, we ensure that neighbouring solutions are indeed structurally different.

We build upon the previous result to prove that this more sophisticated neighbourhood also satisfies the properties of a proper neighbourhood.

Lemma 5.9. *Let $G = (V, E)$ be a weakly connected DAG. For all extensions σ of G and all $v <_\sigma u$, let* $\text{move}_{v,u}(\sigma)$ *be as in Lemma 5.8. Then,* $\Phi_{\text{move}}(\Gamma) = \{$*canonical tree extension of* $\text{move}_{v,u}(\sigma) : uv \in E(\Gamma), uv \notin E(G)\}$ *with σ an extension of Γ, defines a proper neighbourhood on the set of canonical tree extensions Γ of G.*

Proof. To prove that Φ_{move} is a neighbourhood, we will first need a claim.

Claim: Let Γ be a canonical tree extension of G, and σ an extension of Γ. If $uv \in E(\Gamma)$ and $uv \notin E(G)$, then $\text{move}_{v,u}(\sigma)$ is an extension of G.

> *Proof of claim:* Since σ is an extension of Γ, Proposition 3.5 tells us that Γ is canonical for σ. All vertices between v and u in σ are not above v in Γ, else such a vertex would have to be in between u and v in the canonical tree extension Γ. As uv is not an arc of G, we are thus able to move v up to directly after u, while still maintaining that we have an extension. △

Intuitively, it should be clear that $\Phi_{\text{move}}(\Gamma)$ does not depend on the choice of σ. This is because Γ is canonical for any extension of itself (by Proposition 3.5), and its extensions can thus all be seen as representations of the same tree extension. By the claim and the definition of Φ_{move}, we then have that Φ_{move} defines a neighbourhood for all canonical tree extensions of G.

To show that Φ_{move} is a proper neighbourhood, we first note that a canonical tree extension is never a neighbour of itself, and therefore criterion 1 of Definition 5.7 is satisfied. This is true, because for each neighbour Ω of Γ, there exists some u and v such that $u >_\Gamma v$ and $u \not>_\Omega v$. Thus, the neighbour Ω can never be equal to Γ.

Criterion 2 follows from the fact that we can choose σ in such a way that u and v are consecutive in σ. Therefore, when we move u to after v, we are essentially only swapping the two vertices. This new extension is an extension of the new canonical tree extension. We can then swap u and v again to move back to the original tree extension, fulfilling the criterion.

For the third criterion, let Γ and Ω be two different canonical tree extensions, and let σ and π be extensions of Γ and Ω, respectively. By Lemma 5.8, there exists a sequence of swapping moves of length ℓ that transforms σ into π. We will now prove the third criterion by induction on the length ℓ of this sequence.

Base case: If $\ell = 0$, it means that π and σ are direct neighbours. Then, π can be obtained from σ by swapping two consecutive vertices that are not adjacent in G. As the two tree extensions are different, those two vertices must form an arc in the tree extension. As a consequence, these two vertices can be used in our Φ_{move}-neighbourhood, proving the base case.

Induction step: Now assume that $\ell > 0$, and that the induction hypothesis holds for $\ell - 1$. We have two cases. Either, the first swap in the sequence does not change the canonical tree extension. Then, this swap is redundant, and we can just as well pick the extension obtained after this swap as our σ. This σ then needs one swap less in the sequence, and the claim follows from the induction hypothesis. In the other case, the first swap did change the canonical tree extension. But then these two vertices must form an arc in the canonical tree extension. Thus, swapping these two vertices is allowed in the Φ_{move}-neighbourhood. By the induction hypothesis, the statement then follows. □

5.3.2. Description of algorithm

With the newly defined neighbourhood in mind, we are ready to introduce the topic of simulated annealing. Throughout the 1980s numerous authors developed similar methods (see [LA87] for an overview) but Kirkpatrick, Gelatt and Vecchi [KGV83] are often credited as the creators of the algorithm. They successfully applied simulated annealing to the TRAVELLING SALESMAN PROBLEM and named the algorithm after the annealing process in metallurgy, where a material is heated and slowly cooled to reduce its defects and reach a more optimal state.

At its heart, the algorithm is a *metaheuristic*: a heuristic that efficiently guides us through the space of possible solutions. It does so with the help of a nicely chosen neighbourhood (see the previous subsection). The algorithm starts with some initial solution, either created at random or utilizing another heuristic. It then randomly selects a neighbour of the solution, accepting it as the new solution if it has a lower *energy*. Unsurprisingly, in our case, the energy of a solution (that is, an extension) will correspond to its scanwidth. If the neighbour has a larger (i.e. worse) energy, we still have a probability of accepting it. This probability is determined by the *temperature* parameter. The lower the temperature, the less likely it becomes to accept a worse solution. The algorithm starts with a high temperature and as it progresses, it will slowly decrease the temperature according to a pre-defined *cooling schedule*.

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, the heuristic will slowly start to only accept better solutions. Thus, it nudges itself to a (hopefully global) minimum. The success of the method of course heavily relies on a good choice of neighbourhood.

Throughout the literature, a wide range of different formulations exist. Although inherently the same, they differ, among others, in their specified cooling schedules. We will base our algorithm on a formulation from [CB00] (see also [Esl+12]), due to its nice convergence proof.

Algorithm 11: Simulated annealing heuristic to improve an initial tree extension.

Input: Weakly connected DAG $G = (V, E)$, initial canonical tree extension Γ_0 of G, stopping temperature $T_{\text{end}} > 0$, initial temperature $T_0 > T_{\text{end}}$, iteration-dependant cooling factor $\alpha_k \in (0, 1)$.
Output: Canonical tree extension Γ_{best} of G with lowest scanwidth that was found

```
1  initialize
2      T ← T_0; Γ ← Γ_0; Γ_best ← Γ_0; k ← 0
3  while T ≥ T_end do
4      repeat |V| times
5          Γ' ← random neighbour from Φ_move(Γ) ;          // See Lemma 5.9
6          Δsw ← sw(Γ') − sw(Γ)
7          if Δsw < 0 then
8              Γ ← Γ'
9              if sw(Γ) < sw(Γ_best) then
10                 Γ_best ← Γ
11         else if rand(0,1) < e^(−Δsw/T) then
12             Γ ← Γ'
13     k ← k + 1
14     T ← α_k · T
15 return Γ_best
```

In this formulation, we see that the algorithm runs until the temperature drops below some lower bound. The cooling of the temperature is done by multiplying the previous temperature with α_k, whose value may depend on what iteration of the while loop the algorithm is in. Before we cool down the temperature, we repeat the 'searching procedure' $|V|$ times. This polynomial dependence on the size of the problem is a standard choice [LA87], since the solution space also grows with the number of vertices in the graph.[2]

The searching procedure starts with selecting a random neighbour Γ' of the current solution Γ, as defined in Lemma 5.9. Improved solutions are then always accepted. If they improve the overall best solution Γ_{best}, it is updated. We accept a worse solution with a probability of $e^{-\Delta \text{sw}/T}$, the so-called *Metropolis criterion* [LA87]. In this case, Γ_{best} obviously never needs updating.

The algorithm always returns a valid canonical tree extension. This fact, together with the time complexity per iteration, is proved in the next theorem.

Theorem 5.10. *Let $G = (V, E)$ be a weakly connected DAG of n vertices and m arcs. Then, for any initial canonical tree extension Γ_0, stopping temperature $T_{\text{end}} > 0$, initial temperature $T_0 > T_{\text{end}}$, and iteration-dependant cooling factor $\alpha_k \in (0, 1)$, Algorithm 11 will return a canonical tree extension. Furthermore, each iteration of the for-loop takes $O(n^2 \cdot m)$ time.*

Proof. In Lemma 5.9 we proved that Φ_{move} is a neighbourhood for the canonical tree extensions of G. As the initial tree extension is also canonical, it follows that the algorithm always outputs a canonical tree extension of G.

Complexity: In each iteration of the algorithm we first select a random neighbour from $\Phi_{\text{move}}(\Gamma)$. To this end, we start by creating an extension σ of Γ in $O(n)$ time, which is possible by Theorem 3.6. To now efficiently pick a neighbour, we only need to randomly pick an arc uv from Γ such that uv is not an arc of G. As Γ has $n - 1$ arcs, this takes $O(n)$ time. Moving v to after u in σ is also surely bounded by this complexity while creating the canonical tree of this new extension takes $O(n^2)$ time with Algorithm 2 (by Theorem 3.8). Therefore, we can find a neighbour Γ' in $O(n^2)$ time.[a]

We also need to calculate the scanwidth, which takes $O(nm)$ time (see Corollary 3.9). All remaining operations take constant time, resulting in a time complexity of $O(nm)$ (using that $O(n^2) = O(nm)$). As we repeat this procedure n times per iteration, we end up with $O(n^2 \cdot m)$. □

[a] In practice, one can speed up the process of creating a neighbour of Γ by directly altering Γ. Since most parts of the tree extension Γ will remain unchanged, one only needs to locally modify Γ around the vertices u and v. This is a mere practical improvement, so we will not describe this tedious procedure in detail. We do mention that in essence, this comes down to a modification of Algorithm 2. Most of the two underlying extensions remain the same, so we can often reuse a lot of computations each time we run this algorithm.

5.3.3. Asymptotic convergence and cooling schedule

In this subsection, we focus on the *cooling schedule*, which captures all choices to be made regarding the cooling of the temperature. In our case, these are the cooling factor α_k, the initial temperature T_0, and the stopping temperature T_{end}.

Due to the use of a proper neighbourhood, the algorithm has a nice convergence property. Under certain theoretical conditions on the cooling schedule, it can be shown that the algorithm converges in probability to an optimal solution. That is, the probability that the returned solution is optimal will tend to 1 as we lower the stopping temperature to 0 (i.e.

[2] The linear dependence on $|V|$ seems to be a justified choice, since the size of our neighbourhood is upper bounded by $|V|$, because we have at most one neighbour per arc of the tree extension.

let the number of iterations grow to ∞). Instinctively, this makes sense, since criterion 3 of Definition 5.7 says that we will never get stuck in some set of solutions when we wander only through proper neighbours. Therefore, it seems logical that we find an optimal solution in the limit.

Convergence proofs for simulated annealing have been widely researched (see [LA87] for an overview). As our simulated annealing formulation fits the framework of [CB00], we will heavily rely on the convergence proof that is discussed there. The next theorem states the exact conditions that need to be satisfied for convergence.

Theorem 5.11. *Let $G = (V, E)$ be a weakly connected DAG, Γ_0 any initial canonical tree extension, and Δ the difference between the largest and smallest possible scanwidth of any canonical tree extension of G. Then, if $T_0 \geq \Delta$, $\alpha_1 \geq \frac{1}{\ln(2)}$, and $\alpha_k \geq \frac{\ln(k)}{\ln(k+1)}$ for all $k \geq 2$,*

$$\lim_{T_{\text{end}} \to 0} \mathbb{P}[\text{sw}(\Gamma_{\text{best}}) = \text{sw}(G)] = 1,$$

where Γ_{best} is the tree extension returned by Algorithm 11.

Proof. We first note that our used neighbourhood Φ_{move} is proper by Lemma 5.9. Clote and Backofen [CB00] state that what we define as a proper neighbourhood (see Definition 5.7), is in essence a *finite, aperiodic and irreducible Markov chain*.[a] They also reformulate the classic result that a *stationary distribution* exists for these types of Markov chains. Using a much-cited theorem from [GG84], they go on to show that - under some 'technical conditions' - the current solution of the simulated annealing algorithm converges in probability to this stationary distribution. Lastly, it is proved that this stationary distribution only has a positive probability for the optimal states. From this, we can deduce that the best solution Γ_{best} that was found within the algorithm, must be one of the optimal solutions with probability 1 in the limit.

It remains to show that the above 'technical conditions' of the result in [CB00] are met. The first condition they impose is that the initial temperature must be larger than the difference Δ between the maximum and minimum possible energies. As the scanwidth represents our energy value and we search through the space of tree extensions, this is satisfied by the assumptions of our theorem. The second necessary condition is that the temperature T_k in the k-th iteration of the algorithm must satisfy $T_k \geq \frac{\Delta}{\ln(k+1)}$. Using the conditions we impose in this theorem, we get that

$$T_k = T_0 \cdot \alpha_1 \cdot \cdots \cdot \alpha_k \geq T_0 \cdot \frac{1}{\ln(2)} \cdot \frac{\ln(2)}{\ln(3)} \cdot \cdots \cdot \frac{\ln(k)}{\ln(k+1)}.$$

This is a telescoping product, so we conclude that $T_k \geq T_0 \cdot \frac{1}{\ln(k+1)} \geq \frac{\Delta}{\ln(k+1)}$. These were the only conditions necessary for convergence, proving the theorem. □

[a] [CB00] additionally impose that each element has the same number of neighbours, but this is not necessary for the convergence (see e.g. [Esl+12]).

Although this theorem seems to give us some direction towards good parameter values, the result is of mere theoretical relevance. Several studies have shown that the convergence obtained by this so-called *logarithmic cooling* is excessively slow in practice (see e.g. [GG84] for a discussion). Hence, we explore alternative methods to determine the parameter values.

We opt for the most common cooling factor: a constant α. This leads to *exponential cooling*. Often, an α in the range of $[0.85, 0.95]$ is chosen [KGV83; KC92; CB00; Esl+12]. We use a

different option for more control over the running time of the algorithm. Following [SC21], we instead choose the number of iterations we want the algorithm to run and then determine which value of α fulfils this. As Theorem 5.11 provides us with a theoretical estimate of the running time per iteration, this makes it possible to make a somewhat informed decision on how long the algorithm will run. Formally, if we set the number of iterations at $M > 0$, we let

$$\alpha = (T_{\text{end}}/T_0)^{1-M}. \tag{5.1}$$

It is not hard to check that this leads to exactly M iterations.

The second part of our cooling schedule concerns the range of temperatures that will be used. Upon first look, it is far from clear what a suitable temperature range would be. Hence, we adopt an approach described by Johnson et al. [Joh+89] (more elaborately discussed in [LA87]), who instead choose an *initial acceptance probability* χ_0 and then determine the starting temperature accordingly. The reason for this approach is that this probability is a more interpretable quantity than the starting temperature. Furthermore, it is less prone to changes in the size of the graph, because probabilities are always in the interval $[0, 1]$.

In Algorithm 11 the actual initial acceptance probability equals $e^{-\Delta\text{sw}_0/T_0}$, where $\Delta\text{sw}_0 > 0$ denotes the increase in scanwidth, resulting from the first occasion where a worse solution is accepted. To determine T_0 such that this probability equals the predefined value of χ_0, we need the value of Δsw_0.

Since Δsw_0 depends on the specific neighbours that are chosen at random, we do not know its value in advance. Hence, Johnson et al. propose to perform a trial run of the algorithm (of e.g. 1 iteration). This allows us to determine the average of all $\Delta\text{sw} > 0$ encountered in the trial run, for which the corresponding worse solution was accepted. This value then functions as an estimate of the actual Δsw_0. As a convention, we set $\overline{\Delta\text{sw}}^+ = 1$, if no worse solution appeared during the trial run.

To approximately obtain an initial acceptance probability of χ_0, the initial temperature can now be determined as

$$T_0 = -\frac{\overline{\Delta\text{sw}}^+}{\chi_0}. \tag{5.2}$$

We extend this idea to also find the stopping temperature by choosing a *final acceptance probability* χ_{end}. Thus,

$$T_{\text{end}} = -\frac{\overline{\Delta\text{sw}}^+}{\chi_{\text{end}}}. \tag{5.3}$$

To summarize, one needs to choose the number of iterations M, the initial acceptance probability χ_0, and the final acceptance probability χ_{end}, when executing the algorithm. Intuitively, χ_0 needs to be relatively close to 1, while χ_{end} needs to be very small.

Chapter 6

Experimental results

In this chapter we conduct an experimental study to evaluate the performance of our exact algorithms, heuristics and reduction rules on networks. In Section 6.1 we will describe the networks that are used in the experiments and compare their level, scanwidth and treewidth. Section 6.2 focuses on the reduction scheme from Section 3.3. Moving forward, Section 6.3 covers the exact methods from Chapter 4, while Section 6.4 offers a comparative analysis of the heuristics from Chapter 5.

All experiments in this chapter are conducted on an Intel Core i7-8750H CPU @ 2.20 GHz with 16 GB RAM. The algorithms are implemented in Python 3.11.3 using the `NetworkX` graph-library. The implemented algorithms, used networks, and complete numerical results can be found at `https://github.com/nholtgrefe/scanwidth`.

6.1. Network generation

Closely following the experimental study from [Ier+23], 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 [AGM16]. These networks, referred to as the *real* networks, include the network from Figure 1.1. Among these 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. As these networks relate to real-world data, we have included Table B.1 in Appendix B.2, containing the exact results of most experiments for each of these real networks.

To augment our dataset, we use the birth-hybridization network generator from [Zha+17] to create 900 binary networks. This generator is often called the *ZODS generator*, named after the authors of the original paper. The method takes two input parameters: λ, the speciation rate, and ν, the hybridization rate. Just as in the computational experiments from [JL21; Ier+23; Ber+23], 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 [Ier+23] to generate 100 networks for each pair of (r, ℓ), where $r \in \{10, 20, 30\}$ denotes the number of reticulations[1], and $\ell \in \{20, 50, 100\}$ the number of leaves. In total, this gives rise to a dataset comprising 900 networks, which we will refer to as the *synthetic* networks.

Scornavacca and Weller [SW22] requested a comparison of the reticulation number, level, scanwidth and treewidth for different network classes. Figure 6.1 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.

[1] Since all networks are binary, the number of reticulations equals the reticulation number.

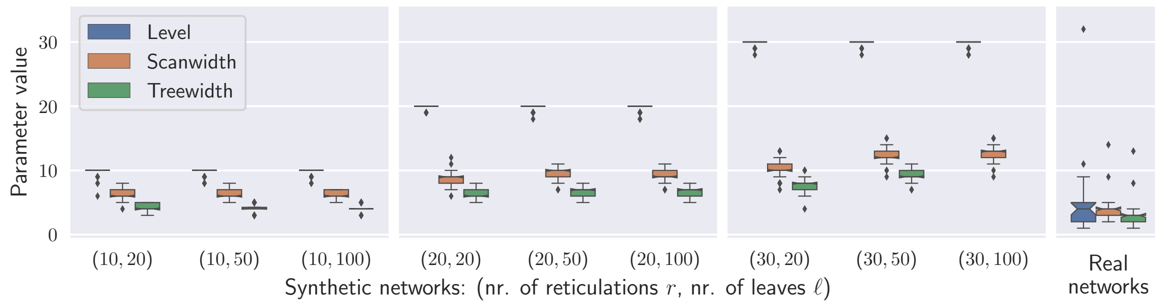


Figure 6.1: Variation in level, scanwidth and treewidth within each dataset. The figure displays a boxplot for each of the nine subsets of the synthetic data and one for the real dataset. The boxplots show the quartiles of the data and its outliers. Different colours indicate the three different parameters. 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.[2]

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 favours 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 Lemma 3.16 and Corollary 3.19. As mentioned in the introductory chapter, 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 a lot smaller than the scanwidth. This observation strengthens our belief in the practical value of scanwidth as a parameter.

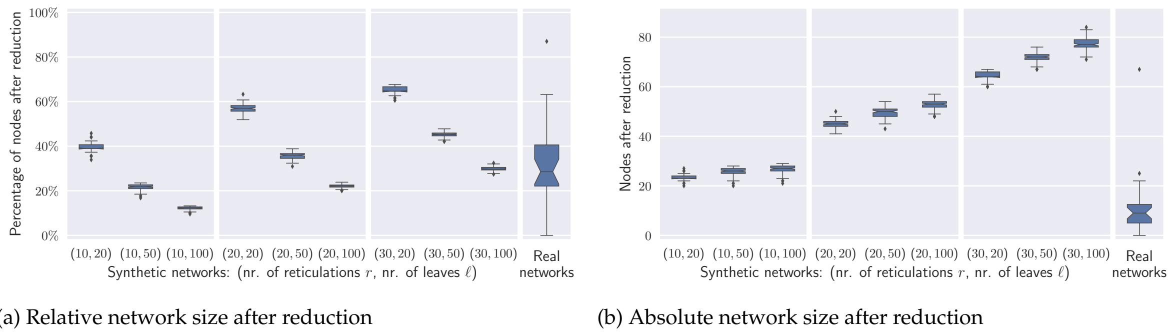
Regarding the real networks, we see a somewhat different trend. The values of the three parameters are closer together in this case (see also Table B.1 for a complete overview of the values). This is attributed to the fact that most levels of the real networks are fairly small. As a consequence, there 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 runs efficiently when the scanwidth is small.

6.2. Reductions

To test the effect the reduction rules from Section 3.3 have on the size of the networks, we employed the decomposition method (Algorithm 3) on each network. Figure 6.2a showcases the percentage of the original nodes 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. However, in most cases the reductions are effective, and reduce the size of the networks to less than half

[2]The values of the scanwidth are calculated using our exact algorithms, whose performance is discussed in Section 6.3. The values of the treewidth are calculated with one of the fastest known exact algorithms by Tamaki [Tam22]. We used his Java implementation of this algorithm - which was published on `https://github.com/twalgor/tw` - on a different CPU.



(a) Relative network size after reduction

(b) Absolute network size after reduction

Figure 6.2: Performance of the decomposition method (Algorithm 3). Figure (a) depicts how many nodes remain after the decomposition, as a percentage of the number of nodes of the original network. Figure (b) shows the absolute number of nodes 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.

of their original size.

Figure 6.2b shows the absolute number of nodes 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 3.28 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.

6.3. Exact algorithms

In Chapter 4 we explored multiple exact algorithms and their respective time complexities. A brute force solution runs in $\tilde{O}(n!)$ time as shown in Proposition 4.1. The recursive Algorithm 4 runs in $\tilde{O}(4^n)$ time. On the other hand, dynamic programming Algorithm 5 has a time complexity of $\tilde{O}(2^n)$ time. We also used a practical improvement called *component splitting* in Algorithm 6, resulting in the same time complexity. The theoretically superior algorithm repeatedly applies the fixed-parameter Algorithm 7, as outlined in the proof of Corollary 4.15. It runs in $O(k \cdot m \cdot n^k)$ time, where k represents the scanwidth. Combining this algorithm and the decomposition algorithm, Corollary 4.16 further establishes that SCANWIDTH is in FPT when considering the level as parameter.

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, Figure 6.3 provides insight into the percentage of networks for which the scanwidth can be determined within 60 seconds.

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 7 achieves the highest. The component splitting from Algorithm 6 also provides an improvement over the standard Algorithm 5.

Interestingly, for the real networks, three of the five algorithms achieve a 100% completion rate, with the fastest of them having 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

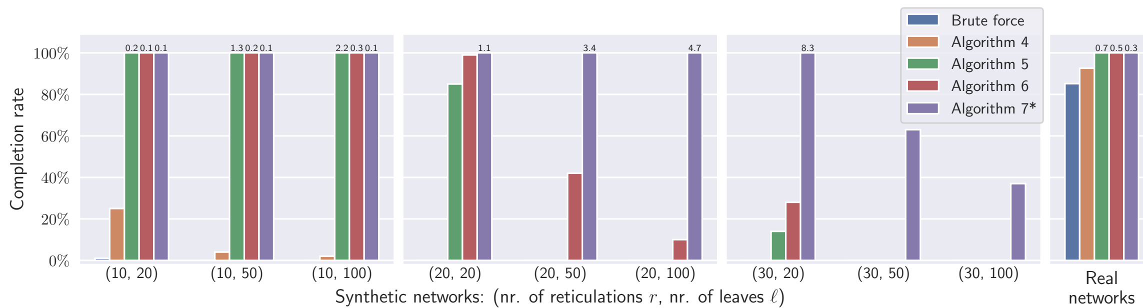


Figure 6.3: Performance of the exact algorithms. For each algorithm, the completion rate, calculated as the number of instances per data set where the algorithm successfully computed the scanwidth within 60 seconds, is shown. If the rate is 100%, the average running time in seconds is also depicted. The figure contains results for each of the nine subsets of the synthetic data and one for the real dataset. The different colours indicate the different algorithms. In all cases, the decomposition algorithm was also applied. The asterisk in the legend indicates that we repeatedly applied Algorithm 7 as outlined in the proof of Corollary 4.15, since the algorithm itself only solves the fixed-parameter version of the problem.

are not too hard.

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 7) 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.

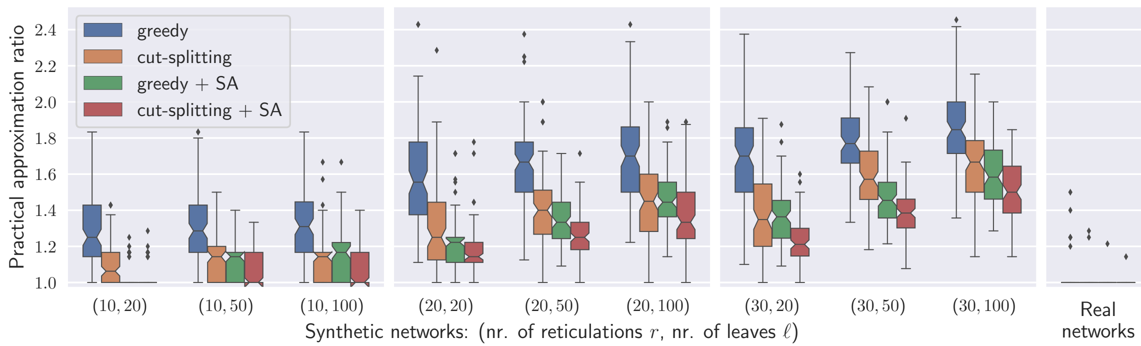
6.4. Heuristics

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 (Algorithm 8) and the cut-splitting heuristic (Algorithm 10) developed in Chapter 5. Additionally, we apply simulated annealing (Algorithm 11) to both results and compare the performances. We emphasize that we have not proved any theoretical bounds on the approximation ratios of the heuristics. For the greedy heuristic, we even provided an example showing that the greedy solution can be arbitrarily bad (see Figure 5.1).

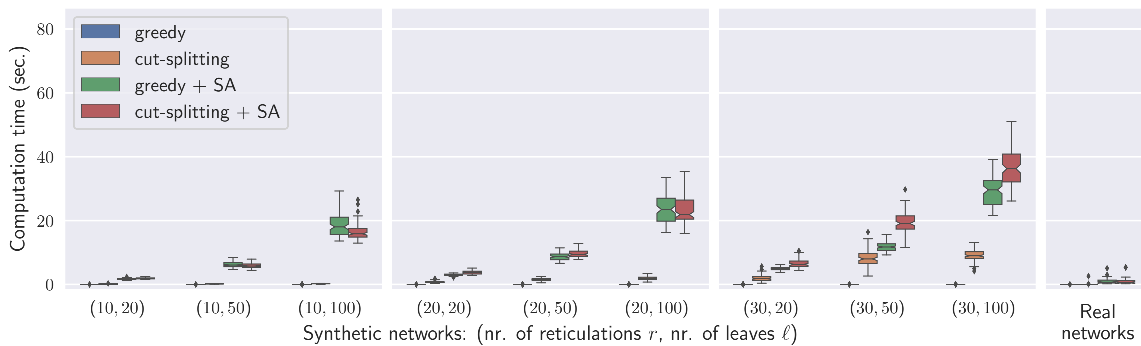
The greedy heuristic and the cut-splitting algorithm do not require any user-defined parameters. However, our simulated annealing algorithm does have three parameters. As proposed in Subsection 5.3.3, we use a cooling schedule that needs a specified number of iterations M, an initial acceptance probability χ_0, and a final acceptance probability χ_{end}. For our experiments, we set the number of iterations $M = 100$, as for all instances this allowed the running time to stay below 60 seconds, the cutoff time for the exact methods. Furthermore, we set $\chi_0 = 0.55$ and $\chi_{\text{end}} = 0.001$. These parameter settings were determined through parameter tuning, described in Appendix B.1.

Figure 6.4 presents the results of the experiment. In Figure 6.4a 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.



(a) Practical approximation ratios



(b) Computation times

Figure 6.4: 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.

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 to solve. 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 Figure 6.4b. 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. This is also in line with their respective theoretical time-complexities from Chapter 5. Finally, it is worth noting that the computation times also scale with the number of leaves and reticulations of the networks.

Chapter 7

Conclusion and outlook

7.1. Conclusion

In this thesis we have successfully attempted to find ways of efficiently computing the scanwidth of directed acyclic graphs. The main contributions of the thesis are three-fold: we have created 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 good-performing heuristic to compute the scanwidth for instances that are too large for the exact method.

Regarding the first topic, an exact algorithm is proposed that can compute the scanwidth in slicewise polynomial time for fixed scanwidth. This algorithm iterates from top-to-bottom through different subsets of the vertices of a graph, using dynamic programming to store intermediate results. With the help of a clever trick to bound the number of sets that need to be considered, the time complexity can be bounded by $O(k \cdot m \cdot n^k)$, with k 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 SCANWIDTH is part of the complexity class XP with respect to the parameter scanwidth. The class XP contains the problems that can be solved in polynomial time for a fixed parameter, allowing the degree of the polynomial to depend on this parameter. 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.

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 minimal (non-trivial) DAG-cuts. Although not necessarily optimal, we showed that this heuristic performs great in practice. When applying simulated annealing to the resulting tree extension, we are able to improve the quality of solutions even more.

Tested on a set of 27 real-world phylogenetic networks, our best-performing exact algorithm is able to compute the scanwidth 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 prac-

tice, 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.

7.2. Further research

One direction of possible further research would be the transferability of some of our results to edge-treewidth, introduced in [Mag+21]. 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 3.15 of treewidth). A possibly simpler generalization would be the translation of our results to weighted DAGs or multigraphs.

A major open question that persists on the topic of time complexity is whether computing the scanwidth is FPT when the parameter is the scanwidth. Resolving this question will possibly also solve the same open question for edge-treewidth, posed in [Mag+21]. Fixed parameter tractability of both treewidth [Bod93] and directed cutwidth [BFT09] would suggest that the same might hold for scanwidth. However, these existing FPT algorithms are far from easily transferable. It is of course to be seen whether such an FPT algorithm would improve our XP algorithm in practice. Nonetheless, we list three strategies that in our opinion might be valuable. Of course, a completely novel approach could prove even more successful.

- The FPT algorithm for cutwidth given in [BFT09] (also discussed in [DT11]) relies on *finite state automata*: a specific type of abstract machine. As explained in [Bod12], this theory can be extended to *finite state tree automata*, which allow for tree-like structures. We hope that it is possible to consider scanwidth from a similar point of view, although attempts have been unsuccessful thus far.

- Berthomé et al. [Ber+13] present a unified structure for *partition functions*. This framework, which also contains cutwidth and treewidth, aims to unify different types of width parameters. The authors also include an involved description of a general FPT algorithm. It might be worthwhile to fit scanwidth within this framework. The main difficulty seems to be the incorporation of directions, as the framework is formulated for undirected graphs. It should be noted that the given algorithm is not constructive, in the sense that it does not provide the optimal decomposition corresponding to the width parameter.

- The FPT algorithm for undirected cutwidth by Thilikos, Serna and Bodlaender [TSB05] makes use of the fact that the cutwidth is closed under taking *immersions*. In a similar fashion, Magne et al. [Mag+21] show that the edge-treewidth is closed under taking *weak topological minors*: a newly defined partial ordering relation on graphs. Due to the edge-treewidth's close relation to the scanwidth, this might be a valuable stepping stone for an FPT algorithm.

A different direction of further research is in the area of approximation algorithms, which are polynomial-time algorithms that are guaranteed to stay within a factor of the optimal solution. Recent research into the (in)approximability of width parameters suggests that treewidth and undirected cutwidth are inapproximable up to a constant factor within polynomial time [Wu+14].[1] It is not unthinkable that these inapproximability results translate

[1] To be exact, [Wu+14] prove that this holds under the *Small Set Expansion Conjecture*. This conjecture, introduced in [RS10], is a strengthened version of the famous P $\neq$ NP-conjecture.

to scanwidth. On the positive side, there does exist a polynomial $O(\log n)$-approximation algorithm for cutwidth [LR99]. Furthermore, treewidth can be approximated within a ratio only linearly dependent on the treewidth itself (see [Kor22] for the state-of-the-art and an overview of other approximation algorithms).

We would also like to address our experimental study, as it inherently has some limitations. Our synthetic networks were created with a single generating method [Zha+17]. A sensible next step is then to extend the study to other network generators (see [JL21]). 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 [Kon+22] for a recent survey). During our experiments, we observed that scanwidth was a lot easier 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.

A final recommendation is to apply scanwidth as a parameter in FPT algorithms for hard problems in phylogenetics. TREE CONTAINMENT [IJW23] and SMALL PARSIMONY [SW22] can be parametrized by treewidth. However, an algorithm based on scanwidth could possibly run faster in practice. On the other hand, no treewidth algorithm is known for HYBRIDIZATION NUMBER [BS07]. This makes it an excellent candidate for a scanwidth-based approach.

References

[AGM16] T. Agarwal, P. Gambette and D. Morrison. *Who is Who in Phylogenetic Networks: Articles, Authors and Programs*. 2016. URL: http://phylnet.univ-mlv.fr/. Accessed: 26-03-2023.

[AGU72] A. V. Aho, M. R. Garey and J. D. Ullman. 'The Transitive Reduction of a Directed Graph'. In: *SIAM Journal on Computing* 1.2 (1972), pp. 131–137. DOI: 10.1137/0201008.

[Arc14] J. D. Archibald. *Aristotle's Ladder, Darwin's Tree: The Evolution of Visual Metaphors for Biological Order*. Columbia University Press, Jan. 2014. DOI: 10.7312/arch16412.

[Bar06] J. Barát. 'Directed Path-width and Monotonicity in Digraph Searching'. In: *Graphs and Combinatorics* 22.2 (June 2006), pp. 161–172. DOI: 10.1007/s00373-005-0627-y.

[Ber+13] P. Berthomé, T. Bouvier, F. Mazoit, N. Nisse and R. Pardo Soares. *An Unified FPT Algorithm for Width of Partition Functions*. Research Report RR-8372. INRIA, Sept. 2013.

[Ber+23] G. Bernardini, L. van Iersel, E. Julien and L. Stougie. *Constructing Phylogenetic Networks via Cherry Picking and Machine Learning*. 2023. arXiv: 2304.02729.

[Bez99] S. L. Bezrukov. 'Edge isoperimetric problems on graphs'. In: *Graph theory and combinatorial biology* 7 (1999), pp. 157–197.

[BFT09] H. L. Bodlaender, M. R. Fellows and D. M. Thilikos. 'Derivation of algorithms for cutwidth and related graph layout parameters'. In: *Journal of Computer and System Sciences* 75.4 (2009), pp. 231–244. DOI: 10.1016/j.jcss.2008.10.003.

[Bod+11] H. L. Bodlaender, F. V. Fomin, A. M. C. A. Koster, D. Kratsch and D. M. Thilikos. 'A Note on Exact Algorithms for Vertex Ordering Problems on Graphs'. In: *Theory of Computing Systems* 50.3 (Jan. 2011), pp. 420–432. DOI: 10.1007/s00224-011-9312-0.

[Bod12] H. L. Bodlaender. 'Fixed-Parameter Tractability of Treewidth and Pathwidth'. In: *The Multivariate Algorithmic Revolution and Beyond: Essays Dedicated to Michael R. Fellows on the Occasion of His 60th Birthday*. Ed. by H. L. Bodlaender, R. Downey, F. V. Fomin and D. Marx. Springer Berlin Heidelberg, 2012, pp. 196–227. DOI: 10.1007/978-3-642-30891-8_12.

[Bod+12] H. L. Bodlaender, F. V. Fomin, A. M. C. A. Koster, D. Kratsch and D. M. Thilikos. 'On Exact Algorithms for Treewidth'. In: *ACM Trans. Algorithms* 9.1 (Dec. 2012). DOI: 10.1145/2390176.2390188.

[Bod93] H. L. Bodlaender. 'A linear time algorithm for finding tree-decompositions of small treewidth'. In: *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*. ACM Press, 1993. DOI: 10.1145/167088.167161.

[BS07] M. Bordewich and C. Semple. 'Computing the minimum number of hybridization events for a consistent evolutionary history'. In: *Discrete Applied Mathematics* 155.8 (2007), pp. 914–928. DOI: 10.1016/j.dam.2006.08.008.

[BSW20] V. Berry, C. Scornavacca and M. Weller. 'Scanning Phylogenetic Networks is NP-hard'. In: *SOFSEM 2020 - 46th International Conference on Current Trends in Theory and Practice of Informatics*. Springer International Publishing, 2020, pp. 519–530. DOI: 10.1007/978-3-030-38919-2_42.

[CB00] P. Clote and R. Backofen. *Computational Molecular Biology: An Introduction*. Wiley, 2000.

[Dar59] C. Darwin. *On the Origin of Species by Means of Natural Selection, or the Preservation of Favored Races in the Struggle for Life*. London: John Murray, 1859.

[DF13] R. G. Downey and M. R. Fellows. *Fundamentals of Parameterized Complexity*. Springer London, 2013. DOI: 10.1007/978-1-4471-5559-1.

[Die17] R. Diestel. *Graph Theory*. Springer Berlin Heidelberg, 2017. DOI: 10.1007/978-3-662-53622-3.

[DPS02] J. Díaz, J. Petit and M. Serna. 'A Survey of Graph Layout Problems'. In: *ACM Comput. Surv.* 34.3 (Sept. 2002), 313–356. DOI: 10.1145/568522.568523.

[DT11] R. G. Downey and D. M. Thilikos. 'Confronting intractability via parameters'. In: *Computer Science Review* 5.4 (2011), pp. 279–317. DOI: 10.1016/j.cosrev.2011.09.002.

[DWZ23] R. Duan, H. Wu and R. Zhou. *Faster Matrix Multiplication via Asymmetric Hashing*. 2023. arXiv: 2210.10173.

[ES75] S. Even and Y. Shiloach. *NP-completeness of several arrangement problems*. Tech. rep. TR-43. Department of Computer Science, Technion, Haifa, 1975.

[Esl+12] C. Eslahchi, R. Hassanzadeh, E. Mottaghi, M. Habibi, H. Pezeshk and M. Sadeghi. 'Constructing circular phylogenetic networks from weighted quartets using simulated annealing'. In: *Mathematical Biosciences* 235.2 (2012), pp. 123–127. DOI: 10.1016/j.mbs.2011.11.003.

[FF56] L. R. Ford and D. R. Fulkerson. 'Maximal Flow Through a Network'. In: *Canadian Journal of Mathematics* 8 (1956), 399–404. DOI: 10.4153/CJM-1956-045-5.

[GG84] S. Geman and D. Geman. 'Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images'. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-6.6 (1984), pp. 721–741. DOI: 10.1109/TPAMI.1984.4767596.

[GR19] F. Gurski and C. Rehs. 'Comparing Linear Width Parameters for Directed Graphs'. In: *Theory of Computing Systems* 63.6 (Apr. 2019), pp. 1358–1387. DOI: 10.1007/s00224-019-09919-x.

[GS84] E. M. Gurari and I. H. Sudborough. 'Improved dynamic programming algorithms for bandwidth minimization and the MinCut Linear Arrangement problem'. In: *Journal of Algorithms* 5.4 (1984), pp. 531–546. DOI: 10.1016/0196-6774(84)90006-3.

[GS87] Y. Gurevich and S. Shelah. 'Expected Computation Time for Hamiltonian Path problem'. In: *SIAM Journal on Computing* 16.3 (1987), pp. 486–502. DOI: 10.1137/0216034.

[HK62] M. Held and R. M. Karp. 'A Dynamic Programming Approach to Sequencing Problems'. In: *Journal of the Society for Industrial and Applied Mathematics* 10.1 (1962), pp. 196–210.

[HT73] J. Hopcroft and R. Tarjan. 'Algorithm 447: Efficient Algorithms for Graph Manipulation'. In: *Commun. ACM* 16.6 (June 1973), 372–378. DOI: 10.1145/362248.362272.

[Ier09] L. van Iersel. 'Algorithms, haplotypes and phylogenetic networks'. PhD thesis. Eindhoven, The Netherlands: Eindhoven University of Technology, 2009.

[Ier+09] L. van Iersel, J. Keijsper, S. Kelk, L. Stougie, F. Hagen and T. Boekhout. 'Constructing Level-2 Phylogenetic Networks from Triplets'. In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 6.4 (2009), pp. 667–681. DOI: 10.1109/TCBB.2009.22.

[Ier+10] L. van Iersel, S. Kelk, R. Rupp and D. Huson. 'Phylogenetic networks do not need to be complex: using fewer reticulations to represent conflicting clusters'. In: *Bioinformatics* 26.12 (June 2010), pp. i124–i131. DOI: 10.1093/bioinformatics/btq202.

[Ier+23] L. van Iersel, M. Jones, E. Julien and Y. Murakami. *Making a Network Orchard by Adding Leaves*. 2023. arXiv: 2305.03106.

[IJW23] L. van Iersel, M. Jones and M. Weller. *Embedding phylogenetic trees in networks of low treewidth*. 2023. arXiv: 2207.00574.

[JL21] R. Janssen and P. Liu. 'Comparing the topology of phylogenetic network generators'. In: *Journal of Bioinformatics and Computational Biology* 19.06 (Dec. 2021). DOI: 10.1142/s0219720021400126.

[Joh+89] D. S. Johnson, C. R. Aragon, L. A. McGeoch and C. Schevon. 'Optimization by Simulated Annealing: An Experimental Evaluation; Part I, Graph Partitioning'. In: *Operations Research* 37.6 (Dec. 1989), pp. 865–892. DOI: 10.1287/opre.37.6.865.

[KC92] P. Kouvelis and W.-C. Chiang. 'A simulated annealing procedure for single row layout problems in flexible manufacturing systems'. In: *International Journal of Production Research* 30.4 (Apr. 1992), pp. 717–732. DOI: 10.1080/00207543.1992.9728452.

[KGV83] S. Kirkpatrick, C. D. Gelatt and M. P. Vecchi. 'Optimization by Simulated Annealing'. In: *Science* 220.4598 (1983), pp. 671–680. DOI: 10.1126/science.220.4598.671.

[Kin92] N. G. Kinnersley. 'The vertex separation number of a graph equals its path-width'. In: *Information Processing Letters* 42.6 (1992), pp. 345–350. DOI: 10.1016/0020-0190(92)90234-M.

[Kon+22] S. Kong, J. C. Pons, L. Kubatko and K. Wicke. 'Classes of explicit phylogenetic networks and their biological and mathematical significance'. In: *Journal of Mathematical Biology* 84.6 (2022), p. 47. DOI: 10.1007/s00285-022-01746-y.

[Kor22] T. Korhonen. 'A Single-Exponential Time 2-Approximation Algorithm for Treewidth'. In: *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, Feb. 2022. DOI: 10.1109/focs52979.2021.00026.

[KRT94] V. King, S. Rao and R. Tarjan. 'A Faster Deterministic Maximum Flow Algorithm'. In: *Journal of Algorithms* 17.3 (1994), pp. 447–474. DOI: 10.1006/jagm.1994.1044.

[KS74] D. E. Knuth and J. L. Szwarcfiter. 'A structured program to generate all topological sorting arrangements'. In: *Information Processing Letters* 2.6 (Apr. 1974), pp. 153–157. DOI: 10.1016/0020-0190(74)90001-5.

[LA87] P. J. M. van Laarhoven and E. H. L. Aarts. *Simulated Annealing: Theory and Applications*. Springer Netherlands, 1987. DOI: 10.1007/978-94-015-7744-1.

[LR99] T. Leighton and S. Rao. 'Multicommodity Max-Flow Min-Cut Theorems and Their Use in Designing Approximation Algorithms'. In: *J. ACM* 46.6 (Nov. 1999), 787–832. DOI: 10.1145/331524.331526.

[Mag+21] L. Magne, C. Paul, A. Sharma and D. M. Thilikos. *Edge-treewidth: Algorithmic and combinatorial properties*. 2021. arXiv: 2112.07524.

[MS89] F. Makedon and I. H. Sudborough. 'On minimizing width in linear layouts'. In: *Discrete Applied Mathematics* 23.3 (1989), pp. 243–265. DOI: 10.1016/0166-218X(89)90016-4.

[Orl13] J. B. Orlin. 'Max Flows in O(nm) Time, or Better'. In: *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*. Association for Computing Machinery, 2013, pp. 765–774. DOI: 10.1145/2488608.2488705.

[Pet13] J. Petit. 'Addenda to the survey of layout problems'. In: *Bulletin of EATCS* 3.105 (2013).

[Rab+21] C.-E. Rabier, V. Berry, M. Stoltz, J. D. Santos, W. Wang, J.-C. Glaszmann, F. Pardi and C. Scornavacca. 'On the inference of complex phylogenetic networks by Markov Chain Monte-Carlo'. In: *PLOS Computational Biology* 17.9 (Sept. 2021). Ed. by S. L. K. Pond. DOI: 10.1371/journal.pcbi.1008380.

[RAK91] R. Ravi, A. Agrawal and P. Klein. 'Ordering problems approximated: single-processor scheduling and interval graph completion'. In: *Automata, Languages and Programming*. Springer Berlin Heidelberg, 1991, pp. 751–762. DOI: 10.1007/3-540-54233-7_180.

[RS10] P. Raghavendra and D. Steurer. 'Graph Expansion and the Unique Games Conjecture'. In: *Proceedings of the forty-second ACM symposium on Theory of computing*. Association for Computing Machinery, 2010, pp. 755–764. DOI: 10.1145/1806689.1806792.

[SC21] V. G. M. Santos and M. A. M. de Carvalho. 'Tailored heuristics in adaptive large neighborhood search applied to the cutwidth minimization problem'. In: *European Journal of Operational Research* 289.3 (2021), pp. 1056–1066. DOI: 10.1016/j.ejor.2019.07.013.

[Sch90] P. Scheffler. 'A Linear Algorithm for the Pathwidth of Trees'. In: *Topics in Combinatorics and Graph Theory*. Physica Heidelberg, 1990, pp. 613–620. DOI: 10.1007/978-3-642-46908-4_70.

[Sip13] M. Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 2013.

[SW22] C. Scornavacca and M. Weller. 'Treewidth-based algorithms for the small parsimony problem on networks'. In: *Algorithms for Molecular Biology* 17.1 (Aug. 2022). DOI: 10.1186/s13015-022-00216-w.

[Tam22] H. Tamaki. *Heuristic computation of exact treewidth*. 2022. arXiv: 2202.07793.

[TSB05] D. M. Thilikos, M. Serna and H. L. Bodlaender. 'Cutwidth I: A linear time fixed parameter algorithm'. In: *Journal of Algorithms* 56.1 (2005), pp. 1–24. DOI: 10.1016/j.jalgor.2004.12.001.

[Wel23] M. Weller. Personal communication. 2023.

[Wol20] L. A. Wolsey. *Integer Programming*. Wiley, Sept. 2020. DOI: 10.1002/9781119606475.

[Wu+14] Y. Wu, P. Austrin, T. Pitassi and D. Liu. 'Inapproximability of Treewidth and Related Problems'. In: *Journal of Artificial Intelligence Research* 49 (Apr. 2014), pp. 569–600. DOI: 10.1613/jair.4030.

[Wu+18] D.-D. Wu, X.-D. Ding, S. Wang, J. M. Wójcik, Y. Zhang, M. Tokarska, Y. Li, M.-S. Wang, O. Faruque, R. Nielsen, Q. Zhang and Y.-P. Zhang. 'Pervasive introgression facilitated domestication and adaptation in the Bos species complex'. In: *Nature Ecology & Evolution* 2.7 (May 2018), pp. 1139–1145. DOI: 10.1038/s41559-018-0562-y.

[Zha+17] C. Zhang, H. A. Ogilvie, A. J. Drummond and T. Stadler. 'Bayesian Inference of Species Networks from Multilocus Sequence Data'. In: *Molecular Biology and Evolution* 35.2 (Dec. 2017), pp. 504–517. DOI: 10.1093/molbev/msx307.

Chapter A

Integer linear program

In this appendix we construct an *integer linear program* (ILP) to solve SCANWIDTH. We will not prove correctness, and merely provide it as a useful building block for possible further research. The reader is assumed to be familiar with integer programming. Should this not be the case, then [Wol20] functions as an excellent reference.

For a weakly connected DAG $G = (V, E)$ of n vertices and m arcs, we introduce a parameter p that indicates the partial order $>_G$ of the graph. Formally,

$$p_{uv} = \begin{cases} 1 & \text{if } u >_G v, \\ 0 & \text{if } u \not>_G v, \end{cases} \quad \forall u, v \in V. \tag{A.1}$$

Our ILP formulation is based on the tree extension definition of scanwidth (Definition 2.3). The ILP uses three types of binary decision variables:

$$x_{uv} = \begin{cases} 1 & \text{if } uv \in E(\Gamma), \\ 0 & \text{if } uv \notin E(\Gamma), \end{cases} \quad \forall u, v \in V \tag{A.2}$$

$$y_{uv} = \begin{cases} 1 & \text{if } u >_\Gamma v, \\ 0 & \text{if } u \not>_\Gamma v, \end{cases} \quad \forall u, v \in V \tag{A.3}$$

$$z^w_{uv} = \begin{cases} 1 & \text{if } uv \in \mathrm{GW}^\Gamma_w, \\ 0 & \text{if } uv \notin \mathrm{GW}^\Gamma_w. \end{cases} \quad \forall uv \in E, \forall w \in V \tag{A.4}$$

Here, x is used to model the arcs of the tree extension and thus functions as the variable of interest. The variables y and z are secondary variables that model the partial order of the tree extension and the cut-sets in the tree extension, respectively. Furthermore, we have one integer variable sw that will capture the value of the scanwidth.

We are now ready to write down the ILP:

$$
\begin{aligned}
\text{minimize} \quad & \text{sw} && && \text{(A.5a)}\\
\text{subject to} \quad & \text{sw} \geq \sum_{uv \in E} z^w_{uv} && ,\forall w \in V, && \text{(A.5b)}\\
& \sum_{u \in V} \sum_{v \in V} x_{uv} = |V| - 1, && && \text{(A.5c)}\\
& \sum_{u \in S} \sum_{v \in S} x_{uv} \leq |S| - 1 && ,\forall S \subseteq V, && \text{(A.5d)}\\
& x_{uv} \leq p_{uv} && ,\forall u, v \in V, && \text{(A.5e)}\\
& x_{uv} \leq y_{uv} && ,\forall u, v \in V, && \text{(A.5f)}\\
& y_{uv} \geq y_{uw} + y_{wv} - 1 && ,\forall u, v, w \in V, && \text{(A.5g)}\\
& z^w_{uv} \geq y_{uw} + y_{wv} - 1 && ,\forall uv \in E, \forall w \in V, && \text{(A.5h)}\\
& z^v_{uv} \geq y_{uv} && ,\forall uv \in E, && \text{(A.5i)}\\
& x_{uv} \in \{0, 1\} && ,\forall u, v \in V, && \text{(A.5j)}\\
& y_{uv} \in \{0, 1\} && ,\forall u, v \in V, && \text{(A.5k)}\\
& z^w_{uv} \in \{0, 1\} && ,\forall uv \in E, \forall w \in V, && \text{(A.5l)}\\
& \text{sw} \in \mathbb{Z}_{\geq 0}. && && \text{(A.5m)}
\end{aligned}
$$

The objective function is trivial: we minimize the value of the scanwidth. The first constraint ensures that sw indeed takes on the value of the scanwidth, by using the variables z, which determine the elements of the sets GW. Together, the second and third constraint are a variant of the well-known *subtour elimination constraints*, used for example in the TRAVELLING SALESMAN PROBLEM. In our case, they ensure that the underlying undirected graph of Γ is a tree.

Constraint (A.5e) then makes sure that we can only have an arc uv in the tree if it respects the graph G. In this way, Γ becomes a tree extension. It also forces x_{vv} to be 0.

Constraints (A.5f) and (A.5g) imply that y has the correct interpretation. If we have an arc $uv \in E(\Gamma)$, then $x_{uv} = 1$, and thus y_{uv} is also forced to be 1. The other constraint ensures that y gets the correct interpretation for nodes that are not directly connected in Γ: if u is above w and w above v, then u must also be above v.

The constraints (A.5h) and (A.5i) force z to model the elements of the set GW. If $u >_\Gamma w >_\Gamma v$ and $uv \in E(G)$, then the constraint (A.5h) forces z^w_{uv} to be 1. This is desired, because we then have that $uv \in GW^\Gamma_w$. The constraint (A.5i) handles the case where $u >_\Gamma w = v$ and $uv \in E(G)$. Together we indeed get that for all $uv \in E(G)$, $uv \in GW^\Gamma_w$ if $u >_\Gamma w \geq_\Gamma v$. Lastly, we have the binarity and integrality constraints of the variables.

All in all, this formulation has a polynomial number of variables, namely $|E||V| + 2|V|^2 + 1 = O(n^3)$. The number of constraints is exponential because constraint (A.5d) appears for each subset of V. As the other constraints are bounded by a polynomial of n and m, we have $O(2^n)$ constraints.

Chapter B

Appendix to the experimental study

B.1. Parameter tuning for simulated annealing

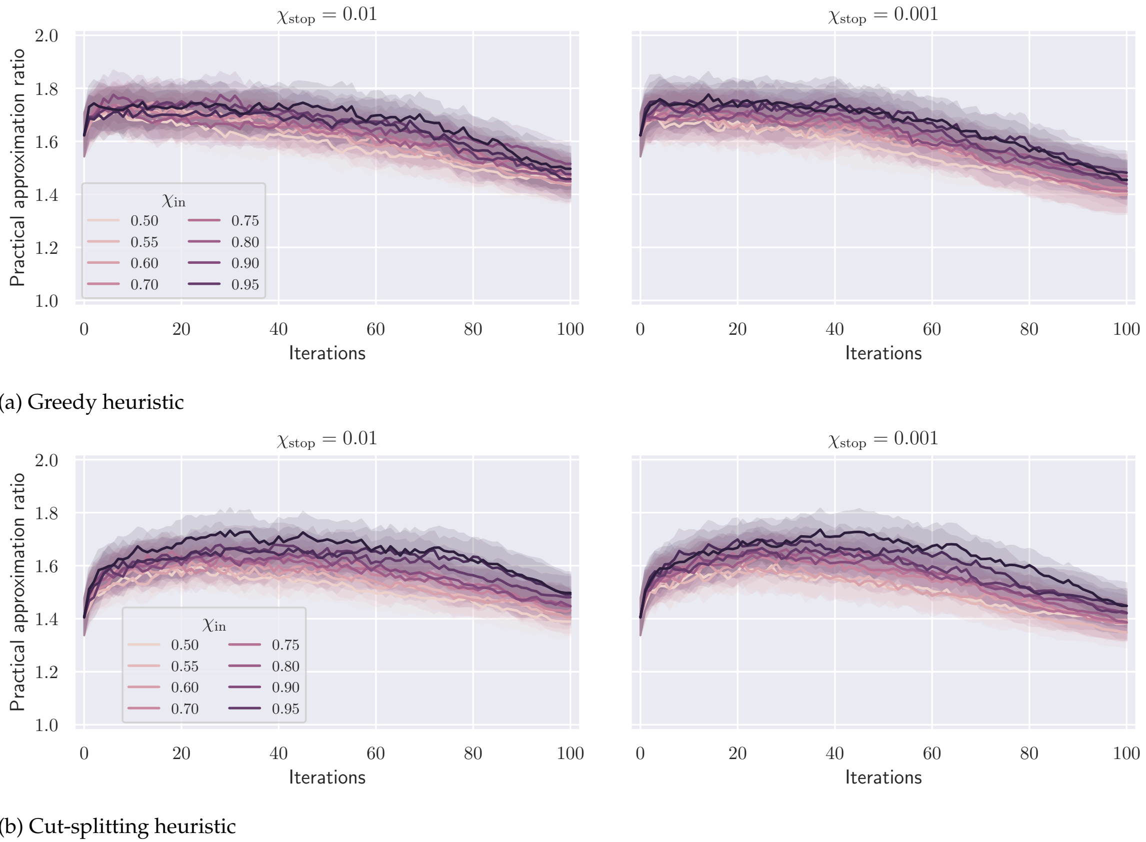
In this appendix we aim to find the best parameter settings for the simulated annealing algorithm. The number of iterations M of the algorithm has been set at 100, as discussed in Section 6.4. Of course, for better results, one can increase the number of iterations.

It remains to choose the initial acceptance probability χ_0 and the final acceptance probability χ_{end}, as defined in Subsection 5.3.3. Following [KC92], we will choose χ_0 from the range $[0.5, 0.95]$. Regarding χ_{end}, we try two different orders of magnitude, resulting in the two values 0.01 and 0.001.

We randomly select 45 difficult samples from our synthetic dataset. Here, 'difficult' indicates that both the greedy heuristic and the cut-splitting heuristic are not able to find the optimum. We then run the simulated annealing algorithm using both heuristics as the initial solution, keeping track of the current scanwidth value. This experiment is performed for each combination of $\chi_0 \in \{0.5, 0.55, 0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95\}$ and $\chi_{\text{end}} \in \{0.01, 0.001\}$.

Figure B.1 shows the average approximation ratios at each iteration (using the scanwidth value of the current solution in the algorithm), with the corresponding 95% confidence intervals. Although no huge differences are apparent, $\chi_{\text{end}} = 0.001$ seems to give better results. Thus, we choose this as our parameter value. We clearly see that the smaller values of χ_0 perform better. To be precise, a value of 0.55 gives the best results for both the heuristics when $\chi_{\text{end}} = 0.001$. All in all, we opt for the choice of $\chi_0 = 0.55$ and $\chi_{\text{end}} = 0.001$.

At first, the relatively low value χ_0 might seem weird. However, this can be explained by the fact that we apply the algorithm to initial heuristic solutions that are already somewhat good. Thus, there is no need to first do a very broad and global search, and we can start with an acceptance probability that is on the lower end of the interval.



(a) Greedy heuristic

(b) Cut-splitting heuristic

Figure B.1: Simulated annealing applied to 45 random samples for different values of χ_0 and χ_{end}. The situation with the greedy result as initial solution is shown in (a), while (b) shows the case with the result of the cut-splitting heuristic as initial solution. The plots show how the average approximation ratios of the solutions develop over the course of 100 iterations. The corresponding 95% confidence intervals are also depicted.

B.2. Table with results for the real networks

This appendix contains the large Table B.1 with information and computational results for the dataset of the 27 real networks. The networks are found throughout the literature and collected on [AGM16]. We use the file names provided on this website.

Apart from some structural information of the networks, we show the performance of four heuristics: the greedy heuristic 8, cut-splitting heuristic 10, greedy heuristic enhanced by simulated annealing (SA, Algorithm 11) and the cut-splitting heuristic enhanced by simulated annealing. Furthermore, the table covers the following exact methods: the brute force solution from Section 4.1, Algorithm 4, Algorithm 5, Algorithm 6, and the repeated application of Algorithm 7 as described in the proof of Corollary 4.15.

Table B.1: Table containing information and experimental results of the 27 real networks. The ‘Network characteristics’ segment contains in order: whether the network is binary, number of vertices $|V|$, number of arcs $|E|$, number of leaves ℓ, reticulation number r, level, scanwidth, and treewidth. The second segment shows the scanwidth values obtained by the different heuristics, together with the computation times. The last segment shows the computation times of the different exact methods, with a time-out of 60 seconds. The asterisk after Algorithm 7 indicates that we repeatedly applied this algorithm as outlined in the proof of Corollary 4.15, since the algorithm itself only solves the fixed-parameter version of the problem. In all cases, we first applied the decomposition method (Algorithm 3) before running the algorithms.

	Network characteristics								**Heuristic scanwidth (time in seconds)**				**Exact methods, time in seconds**				
file	binary	$\|V\|$	$\|E\|$	ℓ	r	lev.	sw	tw	greedy	cut-splitting	SA+ greedy	SA+cut-splitting	brute force	Alg. 4	Alg. 5	Alg. 6	Alg. 7*
n01.el	×	26	27	13	2	2	3	2	3 (0.001)	3 (0.007)	3 (0.561)	3 (0.549)	0.003	0.002	0.002	0.002	0.002
n02.el	×	97	107	39	11	11	9	8	9 (0.005)	9 (0.178)	9 (5.049)	9 (5.194)	60+	60+	0.160	0.043	0.082
n03.el	×	48	49	29	2	2	3	2	3 (0.002)	3 (0.006)	3 (1.529)	3 (1.495)	0.003	0.002	0.002	0.002	0.003
n04.el	✓	53	54	25	2	2	3	2	3 (0.002)	3 (0.007)	3 (1.901)	3 (1.841)	0.003	0.003	0.003	0.002	0.003
n05.el	×	26	29	9	4	4	4	3	5 (0.001)	5 (0.016)	4 (0.440)	4 (0.497)	0.026	0.007	0.004	0.004	0.005
n06.el	✓	77	108	7	32	32	14	13	21 (0.018)	18 (2.608)	17 (3.020)	16 (2.699)	60+	60+	19.572	14.651	7.859
n07.el	×	27	31	8	5	5	4	3	4 (0.001)	4 (0.014)	4 (0.603)	4 (0.602)	0.005	0.005	0.003	0.003	0.004
n08.el	✓	17	19	6	3	3	3	2	3 (0.001)	3 (0.008)	3 (0.280)	3 (0.285)	0.003	0.002	0.002	0.002	0.002
n09.el	×	19	23	4	5	5	5	4	5 (0.001)	5 (0.037)	5 (0.290)	5 (0.275)	1.477	0.046	0.008	0.007	0.011
n10.el	✓	19	22	6	4	4	4	3	5 (0.001)	4 (0.016)	4 (0.255)	4 (0.248)	0.012	0.005	0.003	0.003	0.004
n11.el	✓	53	58	21	6	5	4	3	4 (0.003)	4 (0.035)	4 (1.188)	4 (1.047)	60+	0.351	0.024	0.011	0.013
n12.el	×	16	17	6	2	2	3	2	3 (0.001)	3 (0.003)	3 (0.270)	3 (0.278)	0.002	0.001	0.002	0.002	0.002
n13.el	×	35	40	13	6	4	4	3	4 (0.002)	4 (0.025)	4 (0.698)	4 (0.629)	0.019	0.007	0.004	0.004	0.005
n14.el	×	21	24	7	4	4	4	3	4 (0.001)	4 (0.019)	4 (0.367)	4 (0.359)	0.017	0.006	0.003	0.003	0.004
n15.el	✓	21	23	8	3	3	4	3	4 (0.001)	4 (0.007)	4 (0.455)	4 (0.405)	0.003	0.002	0.002	0.002	0.003
n16.el	✓	23	25	9	3	3	3	2	3 (0.001)	3 (0.006)	3 (0.503)	3 (0.469)	0.002	0.002	0.002	0.002	0.003
n17.el	✓	37	38	17	2	2	3	2	3 (0.001)	3 (0.006)	3 (1.107)	3 (1.019)	0.003	0.002	0.002	0.002	0.003
n18.el	×	26	30	9	5	5	4	3	4 (0.002)	4 (0.023)	4 (0.609)	4 (0.511)	0.074	0.015	0.005	0.004	0.005
n19.el	✓	66	70	28	5	5	5	4	5 (0.002)	5 (0.023)	5 (2.444)	5 (2.285)	0.007	0.010	0.004	0.003	0.005
n20.el	×	14	18	5	5	2	3	2	3 (0.001)	3 (0.003)	3 (0.187)	3 (0.190)	0.002	0.002	0.002	0.002	0.002
n21.el	✓	31	35	11	5	5	4	3	5 (0.002)	4 (0.021)	4 (0.724)	4 (0.593)	0.168	0.026	0.005	0.005	0.006
n22.el	✓	27	27	13	1	1	2	1	2 (0.001)	2 (0.001)	2 (0.703)	2 (0.694)	0.001	0.001	0.001	0.001	0.001
n23.el	×	27	32	6	6	6	5	4	7 (0.002)	5 (0.048)	5 (0.513)	5 (0.501)	33.958	0.197	0.016	0.010	0.012
n24.el	✓	41	46	15	6	6	4	3	4 (0.002)	4 (0.023)	4 (0.974)	4 (0.991)	0.034	0.036	0.005	0.004	0.005
n25.el	✓	39	42	16	4	4	4	3	4 (0.002)	4 (0.031)	4 (1.023)	4 (0.970)	0.336	0.023	0.006	0.004	0.005
n26.el	✓	61	69	22	9	9	5	4	6 (0.004)	5 (0.094)	5 (2.217)	5 (1.591)	60+	51.737	0.160	0.043	0.030
n27.el	✓	51	52	24	2	1	2	1	2 (0.001)	2 (0.001)	2 (1.538)	2 (1.420)	0.002	0.002	0.002	0.002	0.002

Chapter C

Omitted proofs

This appendix contains less interesting or technical proofs of some results in this thesis. We start with the technical Lemma 3.7 used in the proof of correctness of Algorithm 2.

Lemma 3.7. *Let $G = (V, E)$ be a weakly connected DAG, σ an extension of G, and Γ the graph returned by Algorithm 2 applied to σ. Then,*

(a) Γ is a tree extension of G;

(b) σ is an extension of Γ;

(c) $G[V(\Gamma_v)]$ is weakly connected for each $v \in V$, where Γ_v is the subtree of Γ rooted at v.

Proof. Let Γ^i be the graph that is built after the i-th iteration of Algorithm 2. Similarly, denote by r_i the mapping r after iteration i. We will prove the following statements by induction on i:

1. $\sigma[1..i]$ is an extension of Γ^i;

2. Γ^i consists of a tree extension for each connected component of $G[1..i]$;

3. For each $v \in V(\Gamma_i)$, $r_i(v)$ maps to the root of the tree extension in Γ_i that contains v. For each $v \notin V(\Gamma_i)$, $r_i(v)$ equals `None`;

4. For each $v \in V(\Gamma_i)$, the subgraph $G[\Gamma^i_v]$ is weakly connected, where Γ^i_v is the subtree of Γ^i rooted at v.

Base case: ($i = 1$). After the first iteration of Algorithm 2, the vertex $\sigma(1)$ is added. Clearly, $\sigma(1)$ must be a leaf of G, as σ is an extension of G. Therefore, $C = \emptyset$ and no further arcs are added. Thus, Γ_1 is equal to the single vertex $\sigma(1)$. The base cases of the four statements then follow.

Induction step: ($1 \leq i < |V|$). Assume that the induction hypotheses (IH1-IH4) of statements 1-4 hold for i, we will prove that they then also hold for $i + 1$. We denote $v = \sigma(i + 1)$, and consider two cases, depending on whether v is a leaf of G, or not.

Case 1: v is a leaf of G. In the $(i + 1)^{\text{th}}$ iteration of the algorithm, only v is added, and no arcs are added, since v is a leaf of G. Thus, Γ_{i+1} is just Γ_i with the single new vertex v. As $\sigma[1..i]$ was an extension of Γ_i (by IH1), $\sigma[1 \ldots i + 1]$ is then a valid extension of Γ_{i+1}, which proves 1. As R is empty, we only change $r(v)$. This assignment is correctly changed to v,

and statement 3 then follows. Because σ is an extension and v is a leaf, all vertices adjacent to v in G appear after v in σ. Consequently, v is a separate component in $G[1 \ldots i+1]$. The other components are the same as in $G[1 \ldots i]$. From IH2, the second statement follows. For all $u \in V(\Gamma_i)$, we have that $\Gamma_u^i = \Gamma_u^{i+1}$. For v, we have that $\Gamma_v^{i+1} = \{v\}$. Together with IH4, this proves statement 4.

Case 2: v is not a leaf of G. In this case, v has a non-empty set of children C in G. In the $i+1^{\text{th}}$ iteration of the algorithm, we then connect the added vertex v to the roots of the components (in Γ_i) containing the elements of C. By IH3, r_i indeed captures these roots. Thus, v is never added below an already added vertex, and statement 1 now follows (by IH1). Compared to $G[1 \ldots i]$, the components not containing any vertices of C remain a component in $G[1 \ldots i+1]$. The components that do contain vertices from C are grouped together as one component in $G[1 \ldots i+1]$. Combined with the fact that we connect v only to roots of the tree extensions (by IH2 and IH3) and the fact that σ is already G-respecting, 2 follows. Furthermore, $r(v)$ is correctly changed to v, while the vertices that had a root in R are now set to have v as their root. By IH3, the other vertices had the correct root, proving statement 3. $G[\Gamma_v^{i+1}]$ is weakly connected, since we only connect v to components containing a child of v, and those components are weakly connected by IH4. Finally, the other subtrees are also weakly connected by IH4. This proves statement 4.

In both cases, we have proved all 4 statements. Part (a) of the lemma now follows from statement 2 with $i = |V|$. Part (b) is a consequence of statement 1 when $i = |V|$. Lastly, part (c) holds, due to statement 4 with $i = |V|$. □

The following lemma is used in the proof of Theorem 4.5 to bound the time complexity of Algorithm 4.

Lemma C.1. *Let $c > 0$ be a constant, and let $T : \mathbb{N}^2 \to \mathbb{R}_{\geq 0}$ be a function for which the following holds:*

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

Then, there exists a constant $b > 0$ such that $T(k,m) \leq b \cdot 4^k \cdot \log k \cdot m$ for all $m \geq 1$ and $k \geq 2$.

Proof. We will prove the stronger result that the upper bound holds if we have equality in the recurrence relations. It then immediately follows that the upper bound also holds if we have inequalities.

We start by proving the claim that T is non-decreasing with respect to k.

Proof of claim: We prove this by strong induction on k.

Base case: If $k = 2$, we immediately get

$$T(2,m) = \binom{2}{\lfloor 2/2 \rfloor}\left(T(\lfloor 2/2 \rfloor, m) + T(\lceil 2/2 \rceil, m)\right) + c \cdot m$$
$$= 4 \cdot T(1,m) + c \cdot m = 5 \cdot T(1,m) > T(1,m).$$

Induction step: Let $k \geq 2$ be arbitrary and assume that $T(\ell+1, m) \geq T(\ell, m)$ for all $\ell \in \{1, \ldots, k-1\}$. We now have:

$$\begin{aligned}
T(k+1,m) &= \binom{k+1}{\lfloor (k+1)/2 \rfloor}\left(T(\lfloor (k+1)/2 \rfloor, m) + T(\lceil (k+1)/2 \rceil, m)\right) + c \cdot m \\
&\geq \binom{k}{\lfloor k/2 \rfloor}\left(T(\lfloor (k+1)/2 \rfloor, m) + T(\lceil (k+1)/2 \rceil, m)\right) + c \cdot m \\
&\geq \binom{k}{\lfloor k/2 \rfloor}\left(T(\lfloor k/2 \rfloor, m) + T(\lceil k/2 \rceil, m)\right) + c \cdot m = T(k,m).
\end{aligned}$$

For the first inequality, we used that $\binom{x}{y} \geq \binom{x-1}{y-1}$ and $\binom{x}{y} \geq \binom{x-1}{y}$, which follows immediately from Pascal's Rule[a]. This covers the case where k is even and the case where it is odd. The second inequality is due to the induction hypothesis. △

[a]Pascal's Rule says that $\binom{x}{y} = \binom{x-1}{y-1} + \binom{x-1}{y}$.

We now prove a second claim: $T(k,m) \leq 2c \cdot 4^k \cdot \log k \cdot m$ for all $m \geq 1$ and for all $k = 2^\ell$ with $\ell \in \mathbb{N}_{\geq 1}$. We prove this by induction on ℓ.

Proof of claim:

Base case: Here, we take $\ell = 1$, i.e. $k = 2$. Similar to the proof of the previous claim, we have $T(2,m) = 5 \cdot c \cdot m < 9 \cdot c \cdot m < 2c \cdot 4^2 \cdot \log(2) \cdot m$, since $9 < 32 \cdot \log(2)$. This proves the base case.

Induction step: Let $\ell \in \mathbb{N}_{\geq 1}$ be arbitrary, and set $k = 2^\ell$. Assume that the induction hypothesis holds for this value of k. We will prove that it also holds for $2k = 2^{\ell+1}$.

$$\begin{aligned}
T(2k,m) &= \binom{2k}{\lfloor 2k/2 \rfloor}\left(T(\lfloor 2k/2 \rfloor, m) + T(\lceil 2k/2 \rceil, m)\right) + c \cdot m \\
&= \binom{2k}{k}\left(T(k,m) + T(k,m)\right) + c \cdot m \\
&\leq 2^{2k-1} \cdot 2 \cdot T(k,m) + c \cdot m \\
&\leq 2^{2k} \cdot 2c \cdot 4^k \cdot \log k \cdot m + c \cdot m \\
&= 2c \cdot 4^{2k} \cdot (\log(2k) - \log(2)) \cdot m + c \cdot m \\
&= 2c \cdot 4^{2k} \cdot \log(2k) \cdot m + c \cdot m \cdot (1 - 4^{2k} \cdot 2\log(2)) \\
&< 2c \cdot 4^{2k} \cdot \log(2k) \cdot m.
\end{aligned}$$

The first inequality is a consequence of the inequality $\binom{x}{y} \leq 2^{x-1}$ for $x \geq 1$.[a] The second inequality uses the induction hypothesis, while the last inequality holds because $4^{2k} \cdot 2\log(2) > 1$ for all $k \in \mathbb{N}$. This proves the claim. △

[a]Using Pascal's Rule, we get for $x \geq 1$ that $\binom{x}{y} = \binom{x-1}{y-1} + \binom{x-1}{y} \leq \sum_{i=0}^{x-1} \binom{x-1}{i} = 2^{x-1}$.

From the second claim, we know that the upper bound in the lemma is applicable if k is a positive power of 2 (and the constant b is then $2c$). The first claim says that T is non-decreasing with respect to k. Thus, T can not be arbitrarily large if k is in between two powers of 2. Therefore, there must exist some constant $b > 0$ such that $T(k,m) \leq b \cdot 4^k \cdot \log k \cdot m$ for all $m \geq 1$ and $k \geq 2$. $\square$

The next lemma appears as a claim in the proof of Lemma 4.9.

Lemma C.2. *Let $G = (V, E)$ be a weakly connected DAG and $\mathcal{Q} = (W, R)$ an ordered 2-partition of V with $W \neq \emptyset$. Let $\sigma \in \Pi[W]$ be such that for some $k \in \{1, \ldots, |W| - 1\}$, $\sigma(k)$ and $\sigma(k+1)$ are not weakly connected in $G[W]$. Furthermore, let π be obtained from σ by swapping $\sigma(k)$ and $\sigma(k+1)$. Then, $\pi \in \Pi[W]$ and* $\mathrm{rpsw}(\sigma, \mathcal{Q}) = \mathrm{rpsw}(\pi, \mathcal{Q})$.

Proof. First note that $\pi \in \Pi[W]$, since there can not be an arc between $\sigma(k)$ and $\sigma(k+1)$, because they are not weakly connected. We have that $\sigma[1 \ldots i] = \pi[1 \ldots i]$, and $\sigma[i + 1 \ldots] = \pi[i+1 \ldots]$ for all $i \neq k$. From this it can easily be checked that $\mathrm{RPSW}_i^\sigma(\mathcal{Q}) = \mathrm{RPSW}_i^\pi(\mathcal{Q})$ for all $i \notin \{k, k+1\}$. Therefore,

$$\begin{aligned}
\mathrm{RPSW}_k^\sigma(\mathcal{Q}) &= \{uv \in E : u \in \sigma[k+1 \ldots] \cup R, v \overset{G[\sigma[1\ldots k]]}{\leftrightsquigarrow} \sigma(k)\} \\
&= \{uv \in E : u \in \sigma[k+1 \ldots] \cup R, v \overset{G[\sigma[1\ldots k+1]]}{\leftrightsquigarrow} \sigma(k)\} \\
&= \{uv \in E : u \in \pi[k] \cup \pi[k+2 \ldots] \cup R, v \overset{G[\pi[1\ldots k+1]]}{\leftrightsquigarrow} \pi(k+1)\} \\
&= \{uv \in E : u \in \pi[k+2 \ldots] \cup R, v \overset{G[\pi[1\ldots k+1]]}{\leftrightsquigarrow} \pi(k+1)\} \\
&= \mathrm{RPSW}_{k+1}^\pi(\mathcal{Q}).
\end{aligned}$$

In the second equality we use that a vertex is weakly connected to $\sigma(k)$ in $G[\sigma[1 \ldots k]]$, if and only if it is weakly connected to $\sigma(k)$ in $G[\sigma[1 \ldots k+1]]$. This is because $\sigma(k)$ and $\sigma(k+1)$ are weakly disconnected in $G[\sigma[1 \ldots k+1]]$. In the fourth equality, we use that there is no arc directed from $\pi(k)$ towards a vertex that is weakly connected to $\pi(k+1)$ in $G[\pi[1 \ldots k+1]]$, because $\pi(k)$ and $\pi(k+1)$ are weakly disconnected in $G[\pi[1 \ldots k+1]]$.

By symmetry, we also obtain that $\mathrm{RPSW}_k^\pi(\mathcal{Q}) = \mathrm{RPSW}_{k+1}^\sigma(\mathcal{Q})$. All in all, when looking at the vertices and not the positions, we get $\mathrm{RPSW}_j^\sigma(\mathcal{Q}) = \mathrm{RPSW}_j^\pi(\mathcal{Q})$ for all $j \in [W]$. Thus, $\mathrm{rpsw}(\sigma, \mathcal{Q}) = \mathrm{rpsw}(\pi, \mathcal{Q})$. $\square$