*algorithms*

*Article*

# Width, Depth, and Space: Tradeoffs between Branching and Dynamic Programming

**Li-Hsuan Chen [1], Felix Reidl [2], Peter Rossmanith [3,*] and Fernando Sánchez Villaamil [3]**

[1] AROBOT Innovation, Taiwan; clh100p@cs.ccu.edu.tw
[2] Royal Holloway, University of London, TW20 0EX, UK; Felix.Reidl@rhul.ac.uk
[3] Department of Computer Science, RWTH Aachen University, 52062 Aachen, Germany; fernando.sanchez@cs.rwth-aachen.de
* Correspondence: rossmani@cs.rwth-aachen.de; Tel.: +49-241-80-21131

Received: 31 March 2018; Accepted: 28 June 2018; Published: 1 July 2018

**Abstract:** *Treedepth* is a well-established width measure which has recently seen a resurgence of interest. Since graphs of bounded treedepth are more restricted than graphs of bounded tree- or pathwidth, we are interested in the algorithmic utility of this additional structure. On the negative side, we show with a novel approach that the space consumption of any (single-pass) dynamic programming algorithm on treedepth decompositions of depth $d$ cannot be bounded by $(2-\epsilon)^d \cdot \log^{O(1)} n$ for Vertex Cover, $(3-\epsilon)^d \cdot \log^{O(1)} n$ for 3-Coloring and $(3-\epsilon)^d \cdot \log^{O(1)} n$ for Dominating Set for any $\epsilon > 0$. This formalizes the common intuition that dynamic programming algorithms on graph decompositions necessarily consume a lot of space and complements known results of the time-complexity of problems restricted to low-treewidth classes. We then show that treedepth lends itself to the design of branching algorithms. Specifically, we design two novel algorithms for Dominating Set on graphs of treedepth $d$: A pure branching algorithm that runs in time $d^{O(d^2)} \cdot n$ and uses space $O(d^3 \log d + d \log n)$ and a hybrid of branching and dynamic programming that achieves a running time of $O(3^d \log d \cdot n)$ while using $O(2^d d \log d + d \log n)$ space.

**Keywords:** treewidth; treedepth; dynamic programming; branching algorithm; space lower bound

## 1. Introduction

The notion of *treedepth* has been introduced several times in the literature under several different names. It was arguably first formalized under the name of elimination (tree) height in the context of Cholesky factorization [1–6]. The notion has also been studied under the names of 1-*partition tree* [7], *separation game* [8], *vertex ranking/ordered colorings* [9,10]. Recently, Ossona de Mendez and Nešetřil brought the same concept to the limelight in the guise of *treedepth* in their book *Sparsity* [11]. Here we use a definition of treedepth that we consider easiest to exploit algorithmically: The treedepth $\mathbf{td}(G)$ of a graph $G$ is the minimal height of a rooted forest $F$ such that $G$ is a subgraph of the closure of $F$. The closure of a forest is the graph resulting from adding edges between every node and its ancestors, i.e., making every path from root to leaf into a clique. A *treedepth decomposition* is a forest witnessing this fact.

Algorithmically, treedepth is interesting since it is structurally more restrictive than pathwidth. Treedepth bounds the pathwidth and treewidth of a graph, i.e., $\mathbf{tw}(G) \leq \mathbf{pw}(G) \leq \mathbf{td}(G) - 1 \leq \mathbf{tw}(G) \cdot \log n$, where $\mathbf{tw}(G)$ and $\mathbf{pw}(G)$ are the treewidth and pathwidth of an $n$-vertex graph $G$ respectively. Furthermore, a path decomposition can be easily computed from a treedepth decomposition. There are problems that are $W[1]$-hard or remain NP-hard when parameterized by treewidth or pathwidth, but become fixed parameter tractable (fpt) when parameterized by treedepth [12–15]; low treedepth can also be exploited to count the number of appearances of different substructures, such as matchings and small subgraphs, much more efficiently [16,17].

Lokshtanov, Marx and Saurabh showed—assuming the Strong Exponential Time Hypothesis (SETH)—that for 3-COLORING, VERTEX COVER and DOMINATING SET algorithms on a tree decomposition of width $w$ with running time $O(3^w \cdot n)$, $O(2^w \cdot n)$ and $O(3^w w^2 \cdot n)$, respectively, are basically optimal [18]. Their stated intent (as reflected in the title of the paper) was to substantiate the common belief that known dynamic programming algorithms (DP algorithms) that solve these problems where optimal. This is why we feel that a restriction to a certain type of algorithm is not necessarily inferior to a complexity-based approach. Indeed, most algorithms leveraging treewidth *are* dynamic programming algorithms or can be equivalently expressed as such [19–24]. Even before dynamic programming on tree-decompositions became an important subject in algorithm design, similar concepts were already used implicitly [25,26]. The sentiment that the table size is the crucial factor in the complexity of dynamic programming algorithms is certainly not new (see e.g., [27]), so it seems natural to provide lower bounds to formalize this intuition. Our tool of choice will be a family of boundaried graphs that are distinct under Myhill–Nerode equivalence. The perspective of viewing graph decompositions as an "algebraic" expression of boundaried graphs that allow such equivalences is well-established [21,23].

It can be noted that there have been previous formalizations of common algorithmic paradigms in an attempt to investigate what different kinds of algorithms can and cannot achieve, including dynamic programming [28–31]. This allowed to prove lower bounds for the number of operations required for certain specific problems when a certain algorithmic paradigm was applied. Other research shows that for certain problems such as STEINER TREE and SET COVER an improvement over the "naive" dynamic programming algorithm implies improving exhaustive $k$-SAT, which would have implications related to SETH [32,33].

To formalize the notion of a dynamic programming algorithm on tree, path and treedepth decompositions, we consider algorithms that take as input a tree-, path- or treedepth decomposition of width/depth $s$ and size $n$ and satisfy the following constraints:

1. They pass a *single time* over the decomposition in a bottom-up fashion;
2. they use $O(f(s) \cdot \log^{O(1)} n)$ space; and
3. they do not modify the decomposition, including re-arranging it.

While these three constraints might look stringent, they include pretty much all dynamic programming algorithm for hard optimization problems on tree or path decompositions. For that reason, we will refer to this type of algorithms simply as *DP algorithms* in the following.

To show the aforementioned space lower bounds, we introduce a simple machine model that models DP algorithms on treedepth decompositions and construct superexponentially large *Myhill–Nerode families* that imply lower bounds for DOMINATING SET, VERTEX COVER/INDEPENDENT SET and 3-COLORABILITY in this algorithmic model. These lower bounds hold as well for tree and path decompositions and align nicely with the space complexity of known DP algorithms: for every $\epsilon > 0$, no DP algorithm on such decomposition of width/depth $d$ can use space bounded by $(3-\epsilon)^d \cdot \log^{O(1)} n$ for 3-COLORING or DOMINATING SET nor $(2-\epsilon)^d \cdot \log^{O(1)} n$ for VERTEX COVER/INDEPENDENT SET. While probably not surprising, we consider a formal proof for what previously were just widely held assumptions valuable. The provided framework should easily extend to other problems.

Consequently, any algorithmic benefit of treedepth over pathwidth and treewidth must be obtained by non-DP means. We demonstrate that treedepth allows the design of branching algorithms whose space consumption grows only polynomially in the treedepth and logarithmic in the input size. Such space-efficient algorithms are quite easy to obtain for 3-COLORING and VERTEX COVER/INDEPENDENT SET with running time $O(3^d \cdot n)$ and $O(2^d \cdot n)$, respectively, and space complexity $O(d + \log n)$ and $O(d \cdot \log n)$. Our main contribution on the positive side here are two linear-fpt algorithms for DOMINATING SET which use more involved branching rules on treedepth decompositions. The first one runs in time $d^{O(d^2)} \cdot n$ and uses space $O(d^3 \log d + d \cdot \log n)$. Compared to simple dynamic programming, the space consumption is improved considerably, albeit at the cost of a much higher running time. For this reason, we design a second algorithm that uses a hybrid approach of branching and dynamic programming, resulting in a competitive running time of $O(3^d \log d \cdot n)$ and space consumption $O(2^d d \log d + d \log n)$. Both algorithms are amenable to heuristic improvements.

While applying branching to treedepth seems natural, it is unclear whether it could be applied to treewidth or pathwidth. Recent work by Drucker, Nederlof and Santhanam suggests that, relative to a collapse of the polynomial hierarchy, INDEPENDENT SET restricted to low-pathwidth graphs cannot be solved by a branching algorithm in fpt time [34].

The idea of using treedepth to improve space consumption is not novel. Fürer and Yu demonstrated that it is possible to count matchings using polynomial space in the size of the input [17] and a parameter closely related to the treedepth of the input. Their algorithm achieves a small memory footprint by using the algebraization framework developed by Loksthanov and Nederlof [35]. This technique was also used by Pilipczuk and Wrochna to develop an algorithm for DOMINATING SET which runs in time $3^d \cdot \text{poly}(n)$ (non-linear) and uses space $O(d \cdot \log n)$ [36]. Based on this last algorithm they showed that computations on treedepth decompositions correspond to a model of non-deterministic machines that work in polynomial time and logarithmic space, with access to an auxiliary stack of maximum height equal to the decomposition's depth.

In our opinion, algorithms based on algebraization have two disadvantages: On the theoretical side, the dependency of the running and space consumption on the input size is often at least $\Omega(n)$. On the practical side, using the *Discrete Fourier Transform* makes it hard to apply common algorithm engineering techniques, like *branch and bound*, which are available for branching algorithms.

## 2. Preliminaries

We write $N_G[x]$ to denote the closed neighbourhood of $x$ in $G$ and extends this notation to vertex sets via $N_G[S] := \bigcup_{x \in S} N_G[x]$. Otherwise we use standard graph-theoretic notation (see [37] for any undefined terminology). All our graphs are finite and simple and logarithms use base two. For sets $A, B, C$ we write $A \uplus B = C$ to express that $A, B$ partition $C$.

**Definition 1** (Treedepth). *A* treedepth decomposition *of a graph $G$ is a forest $F$ with vertex set $V(G)$, such that if $uv \in E(G)$ then either $u$ is an ancestor of $v$ in $F$ or vice versa. The* treedepth $\mathbf{td}(G)$ *of a graph $G$ is the minimum height of any treedepth decomposition of $G$.*

We assume that the input graphs are connected, which allows us to presume that the treedepth decomposition is always a tree. Furthermore, let $x$ be a node in some treedepth decomposition $T$. We denote by $T_x$ the complete subtree rooted at $x$ and by $P_x$ the set of ancestors of $x$ in $T$ (not including $x$). A *subtree of $x$* refers to a subtree rooted at some child of $x$. The treedepth of a graph $G$ bounds its *treewidth* $\mathbf{tw}(G)$ and *pathwidth* $\mathbf{pw}(G)$, i.e., $\mathbf{tw}(G) \leq \mathbf{pw}(G) \leq \mathbf{td}(G) - 1$ [11]. For a definition of treewidth and pathwidth see e.g., Bodlaender [19].

An *$s$-boundaried graph* $^\circ G$ is a graph $G$ with a set $bd(^\circ G) \subseteq V(G)$ of $s$ distinguished vertices labeled 1 through $s$, called the *boundary* of $^\circ G$. We will call vertices that are not in $bd(^\circ G)$ *internal*. By $^\circ \mathcal{G}_s$ we denote the class of all $s$-boundaried graphs. For $s$-boundaried graphs $^\circ G_1$ and $^\circ G_2$, we let the *gluing* operation $^\circ G_1 \oplus {^\circ G_2}$ denote the $s$-boundaried graph obtained by first taking the disjoint union of $G_1$ and $G_2$ and then unifying the boundary vertices that share the same label. In the literature the result of gluing is often an unboundaried graph. Our definition of gluing will be more convenient in the following.

## 3. Myhill–Nerode Families

In this section, we introduce the basic machinery to formalize the notion of dynamic programming algorithms and how we prove lower bounds based on this notion.

First of all, we need to establish what we mean by *dynamic programming (DP)*. DP algorithms on graph decompositions work by visiting the bags/nodes of the decomposition in a bottom-up fashion (a post-order depth-first traversal), maintaining tables to compute a solution. For decision problems, these algorithms only need to keep at most $\log n$ tables in memory at any given moment (achieved in the case of treewidth by always descending first into the part of the tree decomposition with the greatest number of leaves). We propose a machine model with a read-only tape for the input that can only be traversed once, which only accepts as input decompositions presented in a valid order. This model suffices to capture known dynamic programming algorithms on path, tree and treedepth decompositions. More specifically, given a decision problem on graphs $\Pi$ and some well-formed instance $(G, \xi)$ of $\Pi$ (where $\xi$ encodes the non-graph part of the input), let $T$ be a tree, path or treedepth decomposition of $G$ of width/depth $d$. We fix an encoding $\hat{T}$ of $T$ that lists the separators provided by the decomposition in the order they are normally visited in a dynamic programming algorithm (post-order depth-first traversal of the bag/nodes of a tree/path/treedepth decomposition) and additionally encodes the edges of $G$ contained in a separator using $O(d \log d)$ bits per bag or path. Then $(d, \hat{T}, \xi)$ is a well-formed instance of the *DP decision problem* $\Pi_{DP}$. Pairing DP decision problems with the following machine model provides us with a way to model DP computation over graph decompositions.

**Definition 2** (Dynamic programming TM)**.** *A* DPTM *$M$ is a Turing machine with an input read-only tape, whose head moves only in one direction and a separate working tape. It accepts as inputs only well-formed instances of some DP decision problem.*

Any single-pass dynamic programming algorithm that solves a DP decision problem on tree, path or treedepth decompositions of width/depth $d$ using tables of size $f(d)$ that does not re-arrange the decomposition can be translated into a DPTM with a working tape of size $O(f(d) \cdot \log n)$. This model does not suffice to rule out algebraic techniques, since this technique, like branching, requires to visit every part of the decomposition many times [17]; or algorithms that preprocess the decomposition first to find a suitable traversal strategy.

The theorem of Myhill and Nerode is best known from formal language theory where it is used to show that a language is not regular, but is more general in its original form. For example, if we look at the language $L$ consisting of all words $a^n b a^n$ we define the equivalence relation $\equiv$ by defining $u \equiv v$ if $uw \in L$ iff $vw \in L$ for all $w \in \{a, b\}^*$. Then $a^n \not\equiv a^m$ if $n \neq m$. This means that $\equiv$ has infinitely many equivalence classes which shows that $L$ is not regular. To show that an automaton would need infinitely many states is only the first step and we can furthermore investigate how much space we need when reading a word of length $n$. In the case of $L$ you would need $\Theta(n)$ states which translates in about $\log n$ bits of space. This is a typical use of the Myhill–Nerode theorem to show a space lower bound, but our goal is different: Let us assume we have a path decomposition of a graph which is read from left to right by dynamic programming. This is very similar to reading a word where the bags play the role of the characters. We want to show a lower bound in term of the size of the bags. Roughly, the size of a bag is related to the logarithm of the alphabet size because there are exponentially many graphs in the number of vertices. Our goal is to show a lower bound that is logarithmic in the length of the input times a *double exponential function* of the size of the bags. For this we need a family of graphs that simultaneously has several properties: (1) Each pair ${}^{\circ}G_1 \neq {}^{\circ}G_2$ from the family are not equivalent in the sense of a Myhill–Nerode equivalence, i.e., there must be a graph ${}^{\circ}H$ such that ${}^{\circ}G_1 \oplus {}^{\circ}H$ and ${}^{\circ}G_2 \oplus {}^{\circ}H$ give different results, e.g., one is three colorable and the other is not. (2) The family has to be very big because its size is a lower bound on the index of the Myhill–Nerode relation. To show that we need at least $c^k g(n)$ space, the size has to be at least $2^{c^k g(n)}$. If, for example, we want to show that $(3-\epsilon)^k \log^{O(1)}(n)$ space is not sufficient, the family has to have size at least $2^{(3-\epsilon)^k \log^{O(1)}(n)} < 2^{3^k}$. (3) All graphs in the family and $H$ have to be small. Otherwise $2^{(3-\epsilon)^k \log^{O(1)}(n)}$ could be bigger than $2^{3^k}$. The size $n$ is the size of a graph from the family glued to $H$.

The following notion of a *Myhill–Nerode family* will provide us with the machinery to prove space lower-bounds for DPTMs where the input instance is an unlabeled graph and hence for common dynamic programming algorithms on such instances. Recall that $^{\circ}\mathcal{G}_s$ denotes the class of all $s$-boundaried graphs.

**Definition 3** (Myhill–Nerode family)**.** *A set $\mathcal{H} \subseteq {^{\circ}\mathcal{G}_s} \times \mathbf{N}$ is an $s$-*Myhill–Nerode family *for a DP-decision problem $\Pi_{DP}$ if the following holds:*

1. *For every $(^{\circ}H, q) \in \mathcal{H}$ it holds that $|^{\circ}H| = |\mathcal{H}| \cdot \log^{O(1)}|\mathcal{H}|$ and $q = 2^{|\mathcal{H}| \cdot \log^{O(1)}|\mathcal{H}|}$.*
2. *For every subset $\mathcal{I} \subseteq \mathcal{H}$ there exists an $s$-boundaried graph $^{\circ}G_{\mathcal{I}} \in {^{\circ}\mathcal{G}_s}$ with $|^{\circ}G_{\mathcal{I}}| = |\mathcal{H}| \cdot \log^{O(1)}|\mathcal{H}|$ and an integer $p_{\mathcal{I}}$ such that for every $(^{\circ}H, q) \in \mathcal{H}$ it holds that*

$$(^{\circ}G_{\mathcal{I}} \oplus {^{\circ}H}, p_{\mathcal{I}} + q) \notin \Pi_{DP} \iff (^{\circ}H, q) \in \mathcal{I}.$$

Let $^{\circ}\mathbf{td}(^{\circ}G)$ be the minimal depth over all treedepth decompositions of $^{\circ}G \in \mathcal{G}_s$ where the boundary appears as a path starting at the root. We define the *size* of a Myhill–Nerode family $\mathcal{H}$ as $|\mathcal{H}|$, its *treedepth* as

$$\mathbf{td}(\mathcal{H}) = \max_{(^{\circ}H,\cdot) \in \mathcal{H}, \mathcal{I} \subseteq \mathcal{H}} {^{\circ}\mathbf{td}}(^{\circ}G_{\mathcal{I}} \oplus {^{\circ}H})$$

and its *treewidth* and *pathwidth* as the maximum tree/path decomposition of lowest width of any $(^{\circ}H, \cdot) \in \mathcal{H}$ where the boundary is contained in every bag.

The following lemma still holds if we replace "treedepth" by "pathwidth" or "treewidth".

**Lemma 1.** *Let $\epsilon > 0, c > 1$ and $\Pi$ be a DP decision problem such that for every $s$ there exists an $s$-Myhill–Nerode family $\mathcal{H}$ for $\Pi$ of size $c^s/f(s)$ where $f(s) = s^{O(1)}$ and depth $\mathbf{td}(\mathcal{H}) = s + o(s)$. Then no DPTM can decide $\Pi$ using space $O((c-\epsilon)^k \cdot \log^{O(1)} n)$, where $n$ is the size of the input instance and $k$ the depth of the treedepth decomposition given as input.*

Before proving Lemma 1 let us look at a simple example. The DP decision problem consists of graphs that have components of size $s$ and there are not two components that are isomorphic. Let $\mathcal{H}$ be a set of $c^s$ different $(^{\circ}G, 1)$ where $^{\circ}G$ is a graph of size $s$ with an empty boundary. Then $\mathcal{H}$ is an $s$-Myhill–Nerode family: The size constraints are fulfilled because $|\mathcal{H}| = c^s$ and $|^{\circ}H| = s$. For $\mathcal{I} \subseteq H$ let $^{\circ}G_{\mathcal{I}}$ be the graph whose components are exactly $\mathcal{I}$ and the boundary is empty. Obviously $(^{\circ}G_{\mathcal{I}} \oplus {^{\circ}H}, 1)$ is a no-instance iff $^{\circ}H \in \mathcal{I}$ and the treedepth is always bounded by $s$. Intuitively it is clear that a DPTM that reads a graph from $\mathcal{H}$ has to remember which components were read and which not and so it has to use $c^s$ space. That is exactly what Lemma 1 is saying.

**Proof.** (of Lemma 1) Assume, on the contrary, that such a DPTM $M$ exists. Fix $s = k$ and consider any subset $\mathcal{I} \subseteq \mathcal{H}$ of the $s$-Myhill–Nerode family $\mathcal{H}$ of $\Pi$. By definition, all graphs in $\mathcal{H}$ and the graph $^{\circ}G_{\mathcal{I}}$ have size at most

$$|\mathcal{H}| \cdot \log^{O(1)}|\mathcal{H}| = c^s \cdot s^{O(1)}.$$

By definition, for every $s$-boundaried graph $^{\circ}H$ contained in $\mathcal{H}$, there exist treedepth decompositions for $^{\circ}G_{\mathcal{I}} \oplus {^{\circ}H}$ of depth at most $s + o(s)$ such that the boundary vertices of $^{\circ}G_{\mathcal{I}}$ appear on a path of length $s$ starting at the root of the decomposition. Hence, we can fix a treedepth decomposition $T_{\mathcal{I}}$ of $G_{\mathcal{I}}$ with exactly these properties and choose a treedepth decomposition $T$ of $^{\circ}G_{\mathcal{I}} \oplus {^{\circ}H}$ such that $T_{\mathcal{I}}$ is a subgraph. Moreover, we choose an encoding of $T$ that lists the separators of $T_{\mathcal{I}}$ first.

Notice that $M$ only uses $(c-\epsilon)^{s+o(s)} \cdot s^{O(1)}$ space. There are $2^{|\mathcal{H}|} = 2^{c^s/f(s)}$ choices for $\mathcal{I}$. For there to be a different content on the working tape of $M$ for every choice of $\mathcal{I}$ we need at least $c^s/f(s)$ bits. We rewrite this as $(c-\epsilon)^s \cdot \alpha^s/f(s)$, where $\alpha = c/(c-\epsilon)$. Since $\alpha > 1$ it follows that $\alpha^s/f(s)$ grows exponentially faster than $(c-\epsilon)^{o(s)} \cdot s^{O(1)}$ and thus $c^s/f(s) \in \omega((c-\epsilon)^{s+o(s)} \cdot s^{O(1)})$. By the pigeonhole principle it follows that there exist graphs $^\circ G_{\mathcal{I}}$, $^\circ G_{\mathcal{J}}$ for sets $\mathcal{I} \neq \mathcal{J} \subseteq \mathcal{H}$ for sufficiently large $s$ for which $M$ is in the same state and has the same working tape content after reading the separators of the respective decompositions $T_{\mathcal{I}}$ and $T_{\mathcal{J}}$. Choose $(^\circ H, q) \in \mathcal{I} \triangle \mathcal{J}$. By definition

$$(^\circ G_{\mathcal{I}} \oplus {}^\circ H, p_{\mathcal{I}} + q) \notin \Pi \iff (^\circ G_{\mathcal{J}} \oplus {}^\circ H, p_{\mathcal{J}} + q) \in \Pi$$

but $M$ will either reject or accept both inputs. Contradiction. □

## 4. Space Lower Bounds for Dynamic Programming

In this section we prove space lower bounds for dynamic programming algorithms as defined in Section 3 for 3-Coloring, Vertex Cover and Dominating Set. These space lower bounds all follow the same basic construction. We define a problem-specific "state" for the vertices of a boundary set $X$ and construct two boundaried graphs for it: one graph that enforces this state in any (optimal) solution of the respective problem and one graph that "tests" for this state by either rendering the instance unsolvable or increasing the costs of an optimal solution. We begin by proving a lower bound for 3-Coloring.

**Theorem 1.** *For every $\epsilon > 0$, no DPTM solves* 3-Coloring *on a tree, path or treedepth decomposition of width/depth $k$ with space bounded by $O((3-\epsilon)^k \cdot \log^{O(1)} n)$.*

**Proof.** For any $s$ we construct an $s$-Myhill–Nerode family $\mathcal{H}$. Let $X$ be the $s$ vertices in the boundary of all the boundaried graphs in the following. Then for every three-partition $\mathcal{X} = \{R, G, B\}$ of $X$ we add a boundaried graph $^\circ H_{\mathcal{X}}$ to the family $\mathcal{H}$ by taking a single triangle $v_R, v_G, v_B$ and connecting the vertices $v_C$ to all vertices in $X \setminus C$ for $C \in \{R, G, B\}$. Notice that any 3-coloring of $^\circ H_{\mathcal{X}}$ induces the partition $\mathcal{X}$ on the nodes $X$. Since instances of three-coloring do not need any additional parameter, we ignore this part of the construction of $\mathcal{H}$ and implicitly assume that every graph in $\mathcal{H}$ is paired with zero.

To construct the graphs $G_{\mathcal{I}}$ for $\mathcal{I} \subset \mathcal{H}$, we will employ the *circuit gadget* $v_1, v_2, u$ highlighted in Figure 1. Please note that if $v_1, v_2$ receive the same color, then $u$ must be necessarily colored the same. In every other case, the color of $u$ is arbitrary. Now for every three-partition $\mathcal{X} = \{R, G, B\}$ of $X$ we construct a *testing gadget* $^\circ\mathrm{T}_{\mathcal{X}}$ as follows: For every $C \in \{R, G, B\}$ we arbitrarily pair the vertices in $C$ and connect them via the circuit gadget (as $v_1, v_2$). If $|C|$ is odd, we pair some vertex of $C$ with itself. We then repeat the construction with all the $u$-vertices of those gadgets, resulting in a hierarchical structure of depth $\sim \log |X|$ (c.f., Figure 1 for an example construction). Finally, we add a single vertex $a$ and connect it to the top vertex of the three circuits. Please note that by the properties of the circuit gadget, the graph $^\circ\mathrm{T}_{\mathcal{X}}$ is three-colorable iff the coloring of $X$ does *not* induce the partition $\mathcal{X}$. In particular, the graph $^\circ\mathrm{T}_{\mathcal{X}} \oplus {}^\circ H_{\mathcal{X}'}$ is three-colorable iff $\mathcal{X} \neq \mathcal{X}'$.

Now for every subset $\mathcal{I} \subseteq \mathcal{H}$ of graphs from the family, we define the graph $^\circ G_{\mathcal{I}} = \bigoplus_{^\circ H_{\mathcal{X}} \in \mathcal{I}} {}^\circ\mathrm{T}_{\mathcal{X}}$. By our previous observation, it follows that for every $^\circ H_{\mathcal{X}} \in \mathcal{H}$ the graph $^\circ G_{\mathcal{I}} \oplus {}^\circ H_{\mathcal{X}}$ is three-colorable iff $^\circ H_{\mathcal{X}} \notin \mathcal{I}$. Furthermore, every composite graph has treedepth at most $s + 3\lceil \log s \rceil + 1$ as witnessed by a decomposition whose top $s$ vertices are the boundary $X$ and the rest has the structure of the graph itself after every triangle is made into a path. The graphs $^\circ G_{\mathcal{I}}$ for every $\mathcal{I} \subseteq \mathcal{H}$ have size at most $3^s \cdot 6s$. We conclude that $\mathcal{H}$ is an $s$-Myhill–Nerode family of size $3^s/6$ (the factor $1/6$ accounts for the 3! permutations of the partitions) and the claim follows from Lemma 1. □

Surprisingly, the construction to prove a lower bound for Vertex Cover is very similar to the one for 3-coloring.

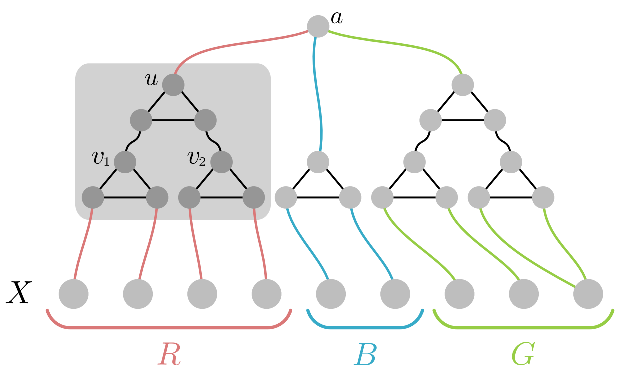

**Figure 1.** The gadget $^{\circ}\mathrm{T}_{\mathcal{X}}$ for $\mathcal{X} = \{R, G, B\}$.

**Theorem 2.** *For every $\epsilon > 0$, no DPTM solves* Vertex Cover *on a tree, path or treedepth decomposition of width/depth k with space bounded by $O((2-\epsilon)^k \cdot \log^{O(1)} n)$.*

**Proof.** For every $s$ we construct an $s$-Myhill–Nerode family $\mathcal{H}$. Let $X$ be the $s$ vertices in the boundary of all the boundaried graphs in the following. Assume for now that $s$ is even. For every subset $A \subseteq X$ such that $|A| = |X|/2$ we construct a graph $^{\circ}H_A$ which consists of the boundary as an independent set and a matching to $A$ and add $(^{\circ}H_A, s/2)$ to $\mathcal{H}$. Please note that any optimal vertex cover of any $^{\circ}H_A$ has size $s/2$ and that $A$ is such a vertex cover.

Consider $\mathcal{I} \subseteq \mathcal{H}$. We will again use the circuit gadget highlighted in Figure 1 to construct $^{\circ}G_{\mathcal{I}}$. Please note that if either $v_1$ or $v_2$ is in the vertex cover we can cover the rest of gadget with two vertices, one of them being the top vertex $u$. Otherwise, $u$ cannot be included in a vertex cover of size two. We still need two vertices, even if both $v_1$ and $v_2$ are already in the vertex cover. For a set $A \subseteq X$ such that $|A| = |X|/2$ we construct the testing gadget $^{\circ}\mathrm{T}_A$ by starting with the boundary $X$ as an independent set, connecting the vertices of $X \setminus A$ pairwise via the circuit gadget (using an arbitrary pairing and potentially pairing a leftover vertex with itself). As in the proof of Theorem 1, we repeat this construction on the respective $u$-vertices of the circuits just added and iterate until we have added a single circuit at the very top. Let us denote the topmost $u$-vertex in this construction by $u'$. Let $\lambda$ be the number of circuits added in this fashion. Any optimal vertex of $^{\circ}\mathrm{T}_A$ has size $2\lambda$ and does not include $u'$. Please note that if a node of $X \setminus A$ is in the vertex cover, we can cover the rest of the gadget with $2\lambda$ many vertices, such that $u'$ is part of the vertex cover.

We construct $^{\circ}G_{\mathcal{I}}$ by taking $\oplus_{^{\circ}H_A \in \mathcal{I}} {^{\circ}\mathrm{T}_A}$ and adding a single vertex $a$ that connects to all $u'$-vertices of the gadgets $\{^{\circ}\mathrm{T}_A\}_{^{\circ}H_A \in \mathcal{I}}$. Notice that, by the same reason $u'$ was not part of an optimal vertex cover of any gadget $^{\circ}\mathrm{T}_A$, the node $a$ must be part of any optimal vertex cover of $^{\circ}G_{\mathcal{I}}$ for $|\mathcal{I}| > 1$. For $|\mathcal{I}| = 1$ either the only $u'$ or $a$ must be contained besides the other vertices, but we will assume w.l.o.g. that it is $a$. Let $\ell$ be the biggest optimal vertex cover for any such $^{\circ}G_{\mathcal{I}}$. Let $\ell_{\mathcal{I}}$ be the size of an optimal vertex cover for a specific $^{\circ}G_{\mathcal{I}}$. For simplicity, we pad $^{\circ}G_{\mathcal{I}}$ with $\ell - \ell_{\mathcal{I}}$ isolated $K_2$ subgraphs to ensure that the size of an optimal vertex cover is $\ell$.

We claim that $^{\circ}G_{\mathcal{I}} \oplus {^{\circ}H_A}$ has a vertex cover of size $\ell + s/2 - 1$ iff $^{\circ}H_A \notin \mathcal{I}$. If $H_A \notin \mathcal{I}$, then for every gadget $^{\circ}\mathrm{T}_{A'}$ that comprises $^{\circ}G_{\mathcal{I}}$ it holds that $A' \neq A$. Since $|A'| = |A|$ it follows that $(X \setminus A') \cap A \neq \emptyset$. Since $^{\circ}G_{\mathcal{I}} \oplus {^{\circ}H_A}$ has $s/2$ vertices of degree one whose neighborhood is $A$, we can assume that an optimal vertex cover contains $A$. From the previous arguments about the possible vertex covers for the $^{\circ}\mathrm{T}_{A'}$ gadgets it follows that the solution still needs two nodes for every circuit gadget of $^{\circ}\mathrm{T}_{A'}$, but now this part of the vertex cover can include $u'$. Since this is true for every $^{\circ}\mathrm{T}_{A'}$ it follows that $a$ does not need to be part of the vertex cover. Thus, the size of an optimal vertex cover is precisely $\ell + s/2 - 1$. If $^{\circ}H_A \in \mathcal{I}$ then the gadget $^{\circ}\mathrm{T}_A$ that has no vertex cover using two nodes per circuit gadget that contains its node $u'$. It follows that any optimal vertex cover of $^{\circ}G_{\mathcal{I}} \oplus {^{\circ}H_A}$ must contain either $a$ or its $u'$. Thus the size of the solution is at least $\ell + s/2$. We thus set $p_{\mathcal{I}}$ to be $\ell - 1$.

The size of the family $\mathcal{H}$ is, using Stirling's approximation, bounded from below by

$$\binom{s}{s/2} \geq \frac{2^{s-1}}{\sqrt{s/2}}$$

and is smaller than $2^s$. All numbers involved describe subsets of graphs and thus must be smaller than the sizes of those graphs. All graphs in the family have size $s$. The graphs $^{\circ}G_{\mathcal{I}}$ as described, are constructed by adding a polylogarithmic number of nodes to the boundary per gadget $^{\circ}\mathrm{T}_A$ and thus their size is bounded by $|\mathcal{H}| \cdot \log^{O(1)}|\mathcal{H}|$. For odd $s$ we take $s' = s - 1$ and use the $s'$-family as the $s$-family. The treedepth is $\mathbf{td}(\mathcal{H}) = s + o(s)$ by the same argument as for the construction for Theorem 1. We conclude that $\mathcal{H}$ is an $s$-Myhill–Nerode family of size $2^s/f(s)$ for $f(s) = s^{O(1)} \cap \Omega(1)$ and depth $\mathbf{td}(\mathcal{H}) = s + o(s)$ and thus by Lemma 1 the theorem follows. □

**Theorem 3.** *For every $\epsilon > 0$, no DPTM solves* Dominating Set *on a tree, path or treedepth decomposition of width/depth $k$ with space bounded by $O\big((3-\epsilon)^k \cdot \log^{O(1)} n\big)$.*

**Proof.** For any $s$ divisible by three we construct an $s$-Myhill–Nerode family $\mathcal{H}$ as follows. Let $X$ be the $s$ boundary vertices of all the boundaried graphs in the following. Then for every three-partition $\mathcal{X} = (B, D, W)$ of $X$ into sets of size $s/3$, we construct a graph $H_{\mathcal{X}}$ by connecting two new pendant vertices (with degree one) to every vertex in $B$, connecting every vertex in $D$ to a vertex which itself is connected to two pendant vertices and leaving $W$ untouched and thus isolated. Intuitively, we want the vertices of $B$ to be in any minimal dominating set, the vertices in $D$ to be dominated from a vertex in $H_{\mathcal{X}}$ not in the boundary and the vertices in $W$ to be dominated from elsewhere. We add every pair $(H_{\mathcal{X}}, 2s/3)$ to $\mathcal{H}$. Notice that the size of an optimal dominating set of $H_{\mathcal{X}}[B \cup D]$ is $2s/3$ and there is only one such optimal dominating set, namely $B \cup N(D)$.

For a subset $\mathcal{I} \subseteq \mathcal{H}$ let $\mathcal{D}_W = \{D \mid H_{(X\setminus(D\cup W)),D,W)} \in \mathcal{I}\}$ be a set defined for every $W \subset X$. We construct the graph $^{\circ}G_{\mathcal{I}}$ using the *circuit gadget* with nodes $v_1, v_2, u$ highlighted in Figure 2: Please note that if $v_1, v_2$ need to be dominated, then there is no dominating set of the gadget of size two that contains $u$. If one of $v_1, v_2$ does not need to be dominated (but is not in the dominating set) then a dominating set of size two of the circuit gadget containing $u$ exists. For every $W \subset X$ with $|W| = s/3$ construct a *testing gadget* $^{\circ}\mathrm{T}_W$ as follows. Assume first that $\mathcal{D}_W$ is non-empty. For every set $D \in \mathcal{D}_W$ we construct the gadget $^{\circ}\Lambda_D$ by arbitrarily pairing the vertices in $D$ and connecting them via the circuit gadget as exemplified in Figure 2. This closely parallels the constructions we have seen in the proofs for Theorem 1 and 2: If $|D|$ is odd, we pair some vertex of $D$ with itself. We then repeat the construction with all the $u$-vertices of those gadgets, resulting in a hierarchical structure of depth $\sim \log|D|$. To finalize the construction of $^{\circ}\Lambda_D$ we take the $u$-vertex of the last layer and connect it to a new vertex $u'$. This concludes the construction of $^{\circ}\Lambda_D$. Let in the following $\lambda$ be the number of circuits we used to construct such a $^{\circ}\Lambda_D$ gadget (this quantity only depends on $s$ and is the same for any $^{\circ}\Lambda_D$). If $\mathcal{D}_W$ is empty, then $^{\circ}\mathrm{T}_W$ is the boundary and a $K_2$ with one of its vertices connected to all vertices in $W$ plus $\binom{2s/3}{s/3}2\lambda$ isolated padding-vertices. Otherwise we obtain $^{\circ}\mathrm{T}_W$ by taking the graph $\bigoplus_{D\in\mathcal{D}} {}^{\circ}\Lambda_D$ and adding two additional vertices $a, b$ as well as $\big(\binom{2s/3}{s/3} - |\mathcal{D}_W|\big)2\lambda$ isolated vertices for padding. The vertex $a$ is connected to all $u'$ vertices of all the gadgets $\{^{\circ}\Lambda_D\}_{D\in\mathcal{D}_W}$ and the vertex $b$ is connected to $\{a\} \cup W$ (c.f., again Figure 2 for an example). Finally we define for every $\mathcal{I} \subseteq \mathcal{H}$ the graph $^{\circ}G_{\mathcal{I}} = \bigoplus_{W\subset X, |W|=s/3} {}^{\circ}\mathrm{T}_W$.

Let $\alpha = \binom{2s/3}{s/3}2\lambda + 1$. Consider some $^{\circ}\mathrm{T}_W$, for a $W$ such that $\mathcal{D}_W \neq \varnothing$. Assume we start with a dominating set $S$ such that $S \cap D = \varnothing$ for at least one $D \in D_W$. We want to show that extending $S$ to dominate $V((^{\circ}\mathrm{T}_W) \setminus X) \cup W$ requires at least $\alpha + 1$ many vertices. We can assume that $b$ must be added to the dominating set. All $\big(\binom{2s/3}{s/3} - |\mathcal{D}_W|\big)2\lambda$ padding vertices must also be added. Since we need at least two vertices per circuit gadget, at least $2\lambda$ vertices will always be necessary to dominate each $^{\circ}\Lambda_D$ subgraph of $^{\circ}\mathrm{T}_W$ (of which there are $|\mathcal{D}_W|$ many). For $^{\circ}\Lambda_D$ where $S \cap D = \varnothing$ no dominating set of the circuit gadgets of size $2\lambda$ can also dominate $u'$. Thus we also need to take $a$ or $u'$ into the dominating set and we need at least $\alpha + 1$ many vertices.

Now assume that we start with a dominating set $S$ that contains at least one node of every $D \in \mathcal{D}_W \neq \emptyset$. In this case we can dominate all the circuit gadgets and $u'$ with $2\lambda$ many nodes in every ${}^{\circ}\Lambda_D$. Thus, there is a set in ${}^{\circ}\mathrm{T}_W$ that dominates $V(({}^{\circ}\mathrm{T}_W) \setminus X) \cup W$ of size $\alpha$, since neither $a$ nor any $u'$ needs to be in the dominating set.

Let us now show that our boundaried graphs work as intended and calculate the appropriate parameters $p_{\mathcal{I}}$. Consider any graph ${}^{\circ}H_{(B_0,D_0,W_0)} \in \mathcal{H}$ and the graph ${}^{\circ}G_{\mathcal{I}}$ for any $\mathcal{I} \subseteq \mathcal{H}$. We show that ${}^{\circ}H_{(B_0,D_0,W_0)} \oplus G_{\mathcal{I}}$ has an optimal dominating set of size at most $\binom{s}{s/3}\alpha + 2s/3$ iff ${}^{\circ}H_{(B_0,D_0,W_0)} \notin \mathcal{I}$. We need to include the $s/3$ vertices of $B_0$ and the $s/3$ vertices of $N(D) \cap V({}^{\circ}H_{(B_0,D_0,W_0)})$. We use the sets $\mathcal{D}_W$ as defined previously. First, assume that $\mathcal{D}_{W_0} = \emptyset$, that is, for every set $B', D'$ we have that $H_{(B',D',W_0)} \notin \mathcal{I}$ and in particular $H_{(B_0,D_0,W_0)} \notin \mathcal{I}$. It is easy to see that the simple version of the gadget ${}^{\circ}\mathrm{T}_{W_0}$ for the case where $\mathcal{D}_{W_0} = \emptyset$ can dominate its non-boundary nodes and $W_0$ with $\alpha$ nodes. All other gadgets ${}^{\circ}\mathrm{T}_{W'}$ for $W' \neq W_0$ do not need to dominate their respective $W'$-sets and can therefore include their $a$-vertices and not include their $b$-vertices. Accordingly, they can all dominate their internal vertices with $\alpha$ many nodes. This all adds up to a dominating set of size $\binom{s}{s/3}\alpha + 2s/3$. Next, assume that $\mathcal{D}_{W_0} \neq \emptyset$ and $D_0 \notin \mathcal{D}_{W_0}$, i.e., again $H_{(B_0,D_0,W_0)} \notin \mathcal{I}$. Therefore, for every set $D' \in D_{W_0}$ we have that $D' \cap B_0 \neq \emptyset$. Since we can assume $B_0$ is part of our dominating set, we only need to add $\alpha$ vertices to the dominating from ${}^{\circ}\mathrm{T}_{W_0}$ to dominate $V(({}^{\circ}\mathrm{T}_{W_0}) \setminus X) \cup W_0$. All gadgets ${}^{\circ}\mathrm{T}_{W'}, W' \neq W_0$ also need $\alpha$ vertices, as observed above. We obtain in total a dominating set of size $\binom{s}{s/3}\alpha + 2s/3$. Finally, consider the case that $D_0 \in \mathcal{D}_{W_0}$, i.e., $H_{(B_0,D_0,W_0)} \in \mathcal{I}$. Since $B_0 \cap D_0 = \emptyset$ the gadget ${}^{\circ}\Lambda_{D_0}$ needs $\alpha + 1$ vertices to dominate $V(({}^{\circ}\mathrm{T}_{W_0}) \setminus X) \cup W_0$. Dominating $W_0$ with nodes different from the $b$-vertex of ${}^{\circ}\mathrm{T}_{W_0}$ does not help. Thus, we need at least $\binom{s}{s/3}\alpha + 2s/3 + 1$ vertices to dominate ${}^{\circ}H_{(B_0,D_0,W_0)} \oplus G_{\mathcal{I}}$.

Choosing $p_{\mathcal{I}} = \binom{s}{s/3}\alpha$ completes the construction of $({}^{\circ}G_{\mathcal{I}}, p_{\mathcal{I}})$. The size of all these graphs is bounded by $O(\binom{s}{s/3}\binom{2s/3}{s/3}\log s) = O(3^s \log s)$. We conclude that $\mathcal{H}$ is an $s$-Myhill–Nerode family of size $\binom{s}{s/3}\binom{2s/3}{s/3}$ which is $\Omega(3^s/s)$ and $O(3^s)$. For $s$ indivisible by three we take the next smaller integer $s'$ divisible by three and use $s'$-family as the $s$-family. It is easy to confirm that the treedepth of $\mathcal{H}$ is $\mathbf{td}(\mathcal{H}) = s + o(s)$ and the theorem follows from Lemma 1. $\square$

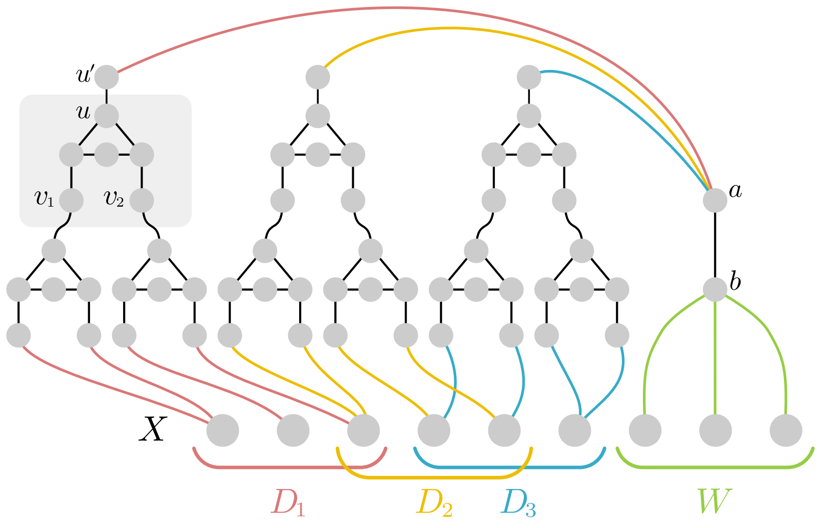

**Figure 2.** The gadget ${}^{\circ}\mathrm{T}_W$ for $\mathcal{D}_W = \{D_1, D_2, D_3\}$. Padding-vertices are not included.

## 5. Dominating Set Using $O(d^3 \log d + d \log n)$ Space

That branching might be a viable algorithmic design strategy for low-treedepth graphs can easily be demonstrated for problems like 3-Coloring and Vertex Cover: We simply branch on the topmost vertex of the decomposition and recur into (annotated) subinstances. For $q$-COLORING, this leads to an algorithm with running time $O(q^d \cdot n)$ and space complexity $O(d \log n)$. Since it is possible to perform a depth-first traversal of a given tree using only $O(\log n)$ space [38], the space consumption of this algorithm can be easily improved to $O(d + \log n)$. Similarly, branching solves VERTEX COVER in time $O(2^d \cdot n)$ and space $O(d \log n)$.

The task of designing a similar algorithm for DOMINATING SET is much more involved. Imagine branching on the topmost vertex of the decomposition: while the branch that includes the vertex into the dominating set produces a straightforward recurrence into annotated instances, the branch that excludes it from the dominating set needs to decide *how* that vertex should be dominated. The algorithm we present here proceeds as follows: We first guess whether the current node $x$ is in the dominating set or not. Recall that $P_x$ denotes the nodes of the decomposition that lie on the unique path from $x$ to the root of the decomposition (and $x \notin P_x$). We iterate over every possible partition $S_1 \uplus \cdots \uplus S_\ell = P_x \cup \{x\}$ into $\ell \leq d$ sets of $P_x \cup \{x\}$. The semantic of a block $S_i$ is that we want every element $S_i$ to be dominated exclusively by nodes from a specific subtree of $x$. A recursive call on a child $y$ of $x$, together with an element of the partition $S_i$, will return the size of a dominating set which dominates $V(T_y) \cup S_i$. The remaining issue is how these specific solutions for the subtrees of $x$ can be combined into a solution in a space-efficient manner. To that end, we first compute the size of a dominating set for $T_y$ itself and use this as baseline cost for a subtree $T_y$. For a block $S_i$ of a partition of $P_x$, we can now compare the cost of dominating $V(T_y) \cup S_i$ against this baseline to obtain overhead cost of dominating $S_i$ using vertices from $T_y$. Collecting these overhead costs in a table for subtrees of $x$ and the current partition, we are able to apply certain reduction rules on these tables to reduce their size to at most $d^2$ entries. Recursively choosing the best partition then yields the solution size using only polynomial space in $d$ and logarithmic in $n$. Formally, we prove the following:

**Theorem 4.** *Given a graph $G$ and a treedepth decomposition $T$ of $G$, Algorithm 1 finds the size of a minimum dominating set of $G$ in time $d^{O(d^2)} \cdot n$ using $O(d^3 \log d + d \log n)$ bits.*

We split the proof of Theorem 4 into lemmas for correctness, running time and used space.

**Lemma 2.** *Algorithm 1 called on a graph $G$, a treedepth decomposition $T$ of $G$, the root $r$ of $T$ and $P = D = \varnothing$ returns the size of a minimum dominating set of $G$.*

**Proof.** If we look at a minimal dominating set $S$ of $G$ we can charge every node in $V(G) \setminus S$ to a node from $S$ that dominates it. We are thus allowed to treat any node in $G$ as if it was dominated by a single node of $S$. We will prove this lemma by induction, the inductive hypothesis being that a call on a node $x$ with arguments $D = S \cap P_x$ and $P \subseteq P_x$ being the set of nodes dominated from nodes in $T_x$ by $S$ returns $|S \cap V(T_x)|$.

It is clear that the algorithm will call itself until a leaf is reached. Let $x$ be a leaf of $T$ on which the function was called. We first check the condition at line 2, which is true if either $x$ is not dominated by a node in $D$ or if some node in $P$ is not yet dominated. In this case we have no choice but to add $x$ to the dominating set. Three things can happen: $P$ is not fully dominated, which means that it was not possible under these conditions to dominate $P$, in which case we correctly return $\infty$, signifying that there is no valid solution. Otherwise we can assume $P$ is dominated and we return 1 if we had to take $x$ and 0 if we did not need to do so. Thus, the leaf case is correct.

We assume now $x$ is not a leaf and thus we reach line 7. We first add $x$ to $P$, since it can only be dominated either from a node in $D$ or a node in $T_x$. Nodes in $T_x$ can only be dominated by nodes from $V(T_x) \cup P_x$. We assume by induction that $D = S \cap P_x$ and that $P$ only contains nodes which are either in $S$ or dominated from nodes in $T_x$. Algorithm 1 executes the same computations for $D$ and $D \cup \{x\}$, representing not taking and taking $x$ into the dominating set respectively. We must show that the set $P$ for the recursive calls is correct. There exists a partition of the nodes of $P$ not dominated by $D$ (respectively $D \cup \{x\}$) such that the nodes of every element of the partition are dominated from a single subtree $T_y$ where $y$ is a child of $x$. The algorithm will eventually find this partition on line 9. The baseline value, i.e., the size of a dominating set of $T_y$ given that the nodes in $D$ (respectively $D \cup \{x\}$) are in the dominating set, gives a lower bound for any solution. In the lists in $L$ and $L'$ we keep the extra cost incurred by a subtree $T_y$ if it has to dominate an element of the partition. We only need to keep the best $d$ values for every $S_i$: Assume that it is optimal to dominate $S_i$ from $T_y$ and there are $d+1$ subtrees induced on children $y' \neq y$ of $x$ whose extra cost over the baseline to dominate $S_i$ is strictly smaller than the extra cost for $T_y$. At least one of these subtrees $T_{y'}$ is not being used to dominate an element of the partition. This means we could improve the solution by letting $T_y$ dominate itself and taking the solution of $T_{y'}$ that also dominates $S_i$. Keeping $d$ values for every element in the partition suffices to find a minimal solution, which is what *find_min_solution*$(L)$ does as follows: Create a bipartite graph $G = (A \uplus B, E)$ such that $A$ contains a node for every $S_i$ and $B$ contains a node for every $y$ for which there is an entry $(\cdot, y)$ in $L$. For every node $a$ representing $S_i$ we add an edge with weight $d - c$ to a node $b$ representing $y$ if $(c, y) \in L[i]$. Notice that the minimal number of nodes above the baseline needed to dominate an element of the partition is always less than $d$. A maximal matching in this bipartite graph tells us how many nodes above the baseline are required to dominate the elements of the partition from subtrees rooted at children of $x$. Since $L$ contains at most $d^2$ entries this can be computed in polynomial time in $d$. Since with lines 27 and 28 we take the minimum over all possible partitions and taking $x$ into the dominating set or not, we get that by inductive assumption the algorithm returns the correct value. The lemma follows since the first call to the algorithm with $D = P = \varnothing$ is obviously correct. □

**Lemma 3.** *Algorithm 1 runs in time $d^{O(d^2)} \cdot n$.*

**Proof.** The running time when $x$ is a leaf is bounded by $O(d^2)$, since all operations exclusively involve some subset of the $d$ nodes in $P_x \cup \{x\}$. Since $|P| \leq d$ the number of partitions of $P$ is bounded by $d^d$. When $x$ is not a leaf the only time spent on computations which are not recursive calls of the algorithm are all trivially bounded by $O(d)$, except the time spent on find_min_solution, which can be solved via a matching problem in polynomial time in $d$ (see proof of Lemma 2). The number of recursive calls that a single call on a node $x$ makes on a child $y$ is $O(d \cdot d^d)$ which bounds the total number of calls on a single node by $d^{O(d^2)}$. This proves the claim. □

**Lemma 4.** *Algorithm 1 uses $O(d^3 \log d + d \log n)$ bits of space.*

**Proof.** There are at most $d$ recursive calls on the stack at any point. We will show that the space used by one is bounded by $O(d^2 \log d + \log n)$. Each call uses $O(d)$ sets, all of which have size at most $d$. The elements contained in these sets can be represented by their position in the path to the root of $T$, thus they use at most $O(d^2 \log d)$ space. The arrays of ordered lists $L, L'$ contain at most $d^2$ elements and all entries are $\leq d$ or $\infty$: If the additional cost (compared to the baseline cost) of dominating a block $S_i$ of the current partition from some subtree $T_y$ exceeds $d$, we disregard this possibility—it would be cheaper to just take all vertices in $S_i$, a possibility explored in a different branch. To find a minimal solution from the table we need to avoid using the same subtree to dominate more than one element of the partition; however, at any given moment we only need to distinguish at most $d^2$ subtrees. Thus, the size of the arrays $L$ and $L'$ is bounded by $O(d^2 \log d)$. The only other space consumption is caused by a constant number of variables (*result*, *baseline*, *baseline*$'$, $b$, $b'$ and $x$) all of them $\leq n$. Thus, the space consumption of a single call is bounded by $O(d^2 \log d + \log n)$ and the lemma follows. □

**Algorithm 1:** Computing dominating sets with very little space.

```
Input: A graph $G$, a treedepth decomposition $T$ of $G$, a node $x$ of $T$ and sets $P, D \subseteq V(G)$.
Output: The size of a minimum Dominating Set.
if $x$ is a leaf in $T$ then
    if $x \notin N_G[D]$ or $P \not\subseteq N_G[D]$ then $D := D \cup \{x\}$ ;
    if $P \not\subseteq N_G[D]$ then return $\infty$ ;
    else if $x \in D$ then return 1 ;
    else return 0 ;
end
$result := \infty$;
$P := P \cup \{x\}$;
foreach partition $S_1 \uplus \cdots \uplus S_\ell$ of $P$ do
    $L :=$ $|P|$-element array of ordered lists;
    $L' :=$ $|P|$-element array of ordered lists;
    $baseline := 0$;
    $baseline' := 0$;
    foreach child $y$ of $x$ in $T$ do
        $b := domset(G, T, y, \varnothing, D)$;
        $baseline := baseline + b$;
        $b' := domset(G, T, y, \varnothing, D \cup \{x\})$;
        $baseline' := baseline + b'$;
        for $S_i \in \{S_1, \dots, S_\ell\}$ do
            $c := \mathrm{domset}(G, T, y, S_i, D) - b$;
            $c' := \mathrm{domset}(G, T, y, S_i, D \cup \{x\}) - b'$;
            Insert $(c, y)$ into ordered list $L[i]$ and keep only smallest $\ell$ elements;
            Insert $(c', y)$ into ordered list $L'[i]$ and keep only smallest $\ell$ elements;
        end
    end
    /* Find minimal cost of dominating $\{S_1, \dots, S_\ell\}$ from $L$ and $L'$ by solving
       appropriate matching problems (see proof of Lemma 2 for details).     */
    $result := min(result, \text{find\_min\_solution}(L) + baseline)$;
    $result := min(result, \text{find\_min\_solution}(L') + baseline' + 1)$;
end
return $result$;
```

## 6. Fast Dominating Set Using $O(2^d d \log d + d \log n)$ Space

We have seen that it is possible to solve Dominating Set on low-treedepth graphs in a space-efficient manner. However, we traded space exponential in the treedepth against superexponential running time in the treedepth and it is natural to ask whether there is some middle ground. We present Algorithm 2 to answer this question: its running time $O(3^d \log d \cdot n)$ is competitive with the default dynamic programming but its space complexity $O(2^d \log d + d \log n)$ is exponentially better. The basic idea is to again branch from the top deciding if the current node $x$ is in the dominating set or not. Intertwined in this branching we compute a function which for a subtree $T_x$ and a set $S \subseteq P_x$ gives the cost of dominating $V(T_x) \cup S$ from $T_x$. For each recursive call on a node we only need this function for subsets of $P_x$ which are not dominated. If $d'$ is the number of nodes of $P_x$ that are currently contained in $D$, the function only needs to be computed for $2^{d-d'}$ sets. This allows us to keep the running time of $O^*(3^d)$, since $\sum_{i=0}^{d} \binom{d}{i} \cdot 2^{d-i} = 3^d$, while only creating tables with at most $O(2^d)$

entries. By representing all values in these tables as $\leq d$ offsets from a base value, the space bound $O(2^d d \log d + d \log n)$ follows. Part of the algorithm will be *convolution* operations.

**Definition 4** (Convolution)**.** *For two functions $M_1, M_2$ with domain $2^U$ for some ground-set $U$ we use the notation $M_1 * M_2$ to denote the* convolution $(M_1 * M_2)[X] := \min_{A \uplus B = X} M_1[A] + M_2[B]$*, for all $X \subseteq U$.*

**Algorithm 2:** Computing dominating sets with $O^*(2^d)$ space.

```
Input: A graph $G$, a treedepth decomposition $T$ of $G$, a node $x$ of $T$ and a set $D \subseteq V(G)$.
Output: The size of a minimum Dominating Set if $x$ is the root of $T$ and $D = \emptyset$.
1  $M, M_1, M_2 :=$ are empty associative arrays. If a set is not in the array its value is $\infty$;
2  if $x$ is a leaf in $T$ then
3      foreach $S \subseteq (P_x \cap N_G[X]) \setminus D$ do $M[S] := 1$;
4      if $x \in N_G[D]$ then $M[\varnothing] := 0$ ;
5      return $M$;
6  end
7  /* Assume the children of $x$ are $\{y_1, \ldots, y_\ell\}$.                */
8  for $i \in \{1, \ldots, \ell\}$ do
9      $M' :=$ domset$(G, T, y_i, D)$;
10     $M_1 := M_1 * M'$;
11 end
12 /* $x$ is not in the dominating set. Discard entries where $x$ is undominated.  */
13 if $x \notin N_G[D]$ then delete all entries $S$ from $M_1$ where $x \notin S$ ;
14 for $i \in \{1, \ldots, \ell\}$ do
15     $M' :=$ domset$(G, T, y_i, D \cup \{x\})$;
16     $M_2 := M_2 * M'$;
17 end
18 foreach $S \in M_2$ do $M_2[S] := M_2[S] + 1$ ;
19 foreach $S \subseteq P_x$ do $M[S] := \min\{M_1[S], M_2[S]\}$;
20 /* Forget $x$.                                                               */
21 foreach $S \in M$ where $x \notin S$ do $M[S] := min(M[S], M[S \cup \{x\}])$ ;
22 Delete all entries $S$ from $M$ where $x \in S$;
23 if $x$ is the root of $T$ then return $M[\varnothing]$ ;
24 else return $M$ ;
```

**Theorem 5.** *For a graph G with treedepth decomposition T, Algorithm 2 finds the size of a minimum dominating set in time $O(3^d \log d \cdot n)$ using $O(2^d d \log d + d \log n)$ bits of space.*

We divide the proof into lemmas as before.

**Lemma 5.** *Algorithm 2 called on $G, T, r, \emptyset$, where T is a treedepth decomposition of G with root r, returns the size of a minimum dominating set of G.*

**Proof.** Notice that the associative array $M$ represents a function which maps subsets of $P_x \setminus D$ to integers and $\infty$. At the end of any recursive call, $M[S]$ for $S \subseteq P_x \setminus D$ should be the size of a minimal dominating set in $T_x$ which dominates $T_x$ and $S$ assuming that the nodes in $D$ are part of the dominating set. We will prove this inductively.

Assume $x$ is a leaf. We can always take $x$ into the dominating set at cost one. In case $x$ is already dominated we have the option of not taking it, dominating nothing at zero cost. This is exactly what is computed in lines 2–5.

Assume now that $x$ is an internal, non-root node of $T$. First, in lines 8–13 we assume that $x$ is not in the dominating set. By inductive assumption calling *domset* on a child $y$ of $x$ returns a table which contains the cost of dominating $T_y$ and some set $S \subseteq P_y \setminus D$. By convoluting them all together $M_1$ represents a function which gives the cost of dominating some set $S \subseteq (P_x \cup \{x\}) \setminus D$ and all subtrees rooted at children of $x$. We just need to take care that $x$ is dominated. If $x$ is not dominated by a node in $D$, then it must be dominated from one of the subtrees. Thus, we are only allowed to retain solutions which dominate $x$ from the subtrees. We take care of this on line 13. After this $M_1$ represents a function which gives the cost of dominating some set $S \subseteq (P_x \cup \{x\}) \setminus D$ and $T_x$ assuming $x$ is not in the dominating set. Then we compute a solution assuming $x$ is in the dominating set in lines 14–18. We first merge the results on calls to the children of $x$ via convolution. Since we took $x$ into the dominating set we increase the cost of all entries by one. After this $M_2$ represents the function which gives the cost of dominating some set $S \subseteq P_x \setminus D$ and $T_x$ assuming $x$ is in the dominating set. We finally merge $M_1$ and $M_2$ by taking the minimums. Since we have taken care that all solutions represented by entries in $M$ dominate $x$ we can remove all information about $x$. We do this in lines 21–22. Finally, $M$ represents the desired function and we return it. When $x$ is the root, instead of returning the table we return the value for the only entry in $M$, which is precisely the size of a minimum dominating set of $G$. □

To prove the running time of Algorithm 2 we will need the values $M$ to be all smaller or equal to the depth of $T$. Thus, we first prove the space upper bound. In the following we treat the associative arrays $M$, $M_1$ and $M_2$ as if the entries where values between 0 and $n$. We will show that we can represent all values as an offset $\leq d$ of a single single value between 0 and $n$.

**Lemma 6.** *Algorithm 2 uses $O(2^d d \log d + d \log n)$ bits of space.*

**Proof.** Let $d$ be the depth of the provided treedepth decomposition. It is clear that the depth of the recursion is at most $d$. Any call to the function keeps a constant number of associative arrays and nodes of the graph in memory. By construction these associative arrays have at most $2^d$ entries. For any of the computed arrays $M$ the value of $M[\varnothing]$ and $M[S]$ for any $S \neq \varnothing$ can only differ by at most $d$. We can thus represent every entry for such a set $S$ as an offset from $M[\varnothing]$ and use $O(2^d \log d + \log n)$ space for the tables. This together with the bound on the recursion depth gives the bound $O(2^d d \log d + d \log n)$. □

**Lemma 7.** *Algorithm 2 runs in time $O(3^d \log d \cdot n)$.*

**Proof.** On a call on which $d'$ nodes of $P_x$ are in the dominating set the associative arrays have at most $2^s$ entries for $s = d - d'$. As shown above the entries in the arrays are $\leq s$ (except one). Hence, we can use fast subset convolution to merge the arrays in time $O(2^s \log s)$ [39]. It follows that the total running time is bounded by

$$O\big(n \cdot \sum_{i=0}^{d} \binom{d}{i} \cdot 2^{d-i} \log(d-i)\big) = O(3^d \log d \cdot n)$$

and thus the lemma follows. □

## 7. Conclusions and Future Work

We have shown that single-pass dynamic programming algorithms on treedepth, tree or path decompositions without preprocessing of the input must use space exponential in the width/depth, confirming a common suspicion and proving it rigorously for the first time. This complements previous SETH-based arguments about the running time of arbitrary algorithms on low treewidth graphs. We further demonstrate that treedepth allows non-DP linear-time algorithms that only use polynomial space in the depth of the provided decomposition. Both our lower bounds and the provided algorithm for DOMINATING SET appear as if they could be special cases of a general theory to be developed in future work and we further ask whether our result can be extended to less stringent definitions of "dynamic programming algorithms".

It would be great to be able to characterize exactly which problems can be solved in linear-fpt time using $\text{poly}(d) \cdot \log n$ space. Tobias Oelschlägel proved as part of his master thesis [40] that the ideas presented here can be extended to the framework of Telle and Proskurowski for graph partitioning problems [41]. Mimicking the development for treewidth would point to extending this result to MSO. Sadly, a proven double exponential dependency on the run-time of model-checking MSO parameterized by the size of a vertex cover implies that no such result is possible [42]. Is there a characterization that better captures for which problems this is possible? Previous research that might be relevant to this endeavor has investigated the height of the tower in the running time for MSO model-checking on graphs of bounded treedepth [43].

Despite the less-than-ideal theoretical bounds of the presented DOMINATING SET algorithms, the opportunities for heuristic improvements are not to be slighted. Take the pure branching algorithm presented in Section 5. During the branching procedure, we generate all partitions from the root-path starting at the current vertex. However, we actually only have to partition those vertices that are not dominated yet (by virtue of being themselves in the dominating set or being dominated by another vertex on the root-path). A sensible heuristic as to which branch—including the current vertex in the dominating set or not—to explore first, together with a *branch and bound* routine should keep us from generating partitions of very large sets. A similar logic applies to the mixed dynamic programming/branching algorithm since the tables only have to contain information about sets that are not yet dominated. It might thus be possible to keep the tables a lot smaller than their theoretical bounds indicate.

Furthermore, it seems reasonable that in practical settings, the nodes near the root of treedepth decompositions are more likely to be part of a minimal dominating set. If this is true, computing a treedepth decomposition would serve as a form of smart preprocessing for the branching, a rough "plan of attack", if you will. How much such a *guided branching* improves upon known branching algorithms in practice is an interesting avenue for further research.

It is still an open question, proposed by Michał Pilipczuk during GROW 2015, whether DOMINATING SET can be solved in time $(3-\epsilon)^d \cdot \text{poly}(n)$ when parameterized by treedepth. Our lower bound result implies that if such an algorithm exists, it cannot be a straightforward dynamic programming algorithm.

**Author Contributions:** All authors' contributions are approximately of equal size.

**Acknowledgments:** Research funded by DFG-Project RO 927/13-1 "Pragmatic Parameterized Algorithms". Li-Hsuan Chen was supported in part by MOST-DAAD Sandwich Program under grant no. 103-2911-I-194-506.

**Conflicts of Interest:** The authors declare no conflict of interest.

## References

1. Duff, I. A multifrontal approach for solving sparse linear equations. In *Numerical Methods*; Springer: Berlin/Heidelberg, Germany, 1983; pp. 87–98.
2. Duff, I.S.; Reid, J.K. The multifrontal solution of indefinite sparse symmetric linear. *ACM Trans. Math. Softw.* **1983**, *9*, 302–325. [CrossRef]
3. Jess, J.A.G.; Kees, H.G.M. A data structure for parallel L/U decomposition. *IEEE Trans. Comput.* **1982**, *31*, 231–239. [CrossRef]
4. Kees, H.G.M. The organization of circuit analysis on array architectures. Ph.D. Thesis, Department of Electrical Engineering, Eindhoven University of Technology, Eindhoven, The Netherlands, 1982.
5. Pieck, J.T.M. *Formele Definitie Van Een E-Tree*; Technical Report; Technische Hogeschool Eindhoven: Eindhoven, The Netherlands, 1980.
6. Pothen, A. *The Complexity of Optimal Elimination Trees*; Technical Report; Pennsylvannia State University: State College, PA, USA, 1988.

7. Iyer, A.V.; Ratliff, H.D.; Vijayan, G. *On a Node Ranking Problem of Trees and Graphs. Technical Report*; DTIC Document; Georgia Inst of Tech Atlanta Production and Distribution Research Center: Atlanta, GA, USA, 1986.
8. Llewellyn, D.C.; Tovey, C.; Trick, M. Local optimization on graphs. *Discret. Appl. Math.* **1989**, *23*, 157–178. [CrossRef]
9. Bodlaender, H.; Deogun, J.; Jansen, K.; Kloks, T.; Kratsch, D.; Müller, H.; Tuza, Z. Rankings of Graphs. *SIAM J. Discret. Math.* **1998**, *11*, 168–181. [CrossRef]
10. Katchalski, M.; McCuaig, W.; Seager, S. Ordered colourings. *Discret. Math.* **1995**, *142*, 141–154. [CrossRef]
11. Nešetřil, J.; Ossona de Mendez, P. Sparsity: Graphs, Structures, and Algorithms. In *Algorithms and Combinatorics*; Springer: Berlin, Germany, 2012; Volume 28.
12. Bannister, M.J.; Cabello, S.; Eppstein, D. Parameterized complexity of 1-planarity. In *Algorithms and Data Structures, Proceedings of the 13th International Symposium, WADS 2013, London, ON, Canada, 12–14 August 2013*; Springer: Berlin/Heidelberg, Germany, 2013; pp. 97–108.
13. Gutin, G.Z.; Jones, M.; Wahlström, M. Structural Parameterizations of the Mixed Chinese Postman Problem. In Proceedings of the Algorithms—ESA 2015—23rd Annual European Symposium, Patras, Greece, 14–16 September 2015; pp. 668–679.
14. Jasik, K. Treewidth on Fire. Bachelor's Thesis, RWTH Aachen University, Aachen, Germany, 2015.
15. Wrochna, M. Reconfiguration in bounded bandwidth and treedepth. *arXiv* **2014**, arXiv:1405.0847.
16. Demaine, E.D.; Reidl, F.; Rossmanith, P.; Sánchez Villaamil, F.; Sikdar, S.; Sullivan, B.D. Structural Sparsity of Complex Networks: Bounded Expansion in Random Models and Real-World Graphs. *arXiv* **2014**, arXiv:1406.2587.
17. Fürer, M.; Yu, H. Space Saving by Dynamic Algebraization Based on Tree-Depth. *Theory Comput. Syst.* **2017**, *61*, 283–304. [CrossRef]
18. Lokshtanov, D.; Marx, D.; Saurabh, S. Known algorithms on graphs of bounded treewidth are probably optimal. In Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms, San Francisco, CA, USA, 23–25 January 2011; pp. 777–789.
19. Bodlaender, H. *Dynamic Programming on Graphs With Bounded Treewidth*; Springer: Berlin/Heidelberg, Germany, 1988.
20. Bodlaender, H. *Treewidth: Algorithmic Techniques and Results*; Springer: Berlin/Heidelberg, Germany, 1997.
21. Bodlaender, H. Fixed-parameter tractability of treewidth and pathwidth. In *The Multivariate Algorithmic Revolution and Beyond*; Springer: Berlin/Heidelberg, Germany, 2012; pp. 196–227.
22. Borie, R.; Parker, R.; Tovey, C. Automatic generation of linear-time algorithms from predicate calculus descriptions of problems on recursively constructed graph families. *Algorithmica* **1992**, *7*, 555–581. [CrossRef]
23. Borie, R.; Parker, R.; Tovey, C. Solving problems on recursively constructed graphs. *ACM Comput. Surv.* **2008**, *41*, 4. [CrossRef]
24. Scheffler, P. *Dynamic Programming Algorithms for Tree-Decomposition Problems*; Akad. d. Wissensch. d. DDR. Institut für Mathematik: Vienna, Austria, 1986.
25. Bern, M.W.; Lawler, E.L.; Wong, A.L. Linear-Time Computation of Optimal Subgraphs of Decomposable Graphs. *J. Algorithms* **1987**, *8*, 216–235. [CrossRef]
26. Bertele, U.; Brioschi, F. On non-serial dynamic programming. *J. Comb. Theory Ser. A* **1973**, *14*, 137–148. [CrossRef]
27. Rooij, J.V.; Bodlaender, H.; Rossmanith, P. Dynamic programming on tree decompositions using generalised fast subset convolution. In *Algorithms-ESA 2009, Proceedings of the 17th Annual European Symposium, Copenhagen, Denmark, 7–9 September 2009*; Number 5193 in LNCS; Springer: Berlin/Heidelberg, Germany, 2009; pp. 566–577.
28. Alekhnovich, M.; Borodin, A.; Buresh-Oppenheim, J.; Impagliazzo, R.; Magen, A.; Pitassi, T. Toward a model for backtracking and dynamic programming. *Comput. Complex.* **2011**, *20*, 679–740. [CrossRef]
29. Buresh-Oppenheim, J.; Davis, S.; Impagliazzo, R. A stronger model of dynamic programming algorithms. *Algorithmica* **2011**, *60*, 938–968. [CrossRef]
30. Helman, P. A common schema for dynamic programming and branch and bound algorithms. *J. ACM* **1989**, *36*, 97–128. [CrossRef]
31. Karp, R.M.; Held, M. Finite-state processes and dynamic programming. *SIAM J. Appl. Math.* **1967**, *15*, 693–718. [CrossRef]

32. Cygan, M.; Dell, H.; Lokshtanov, D.; Marx, D.; Nederlof, J.; Okamoto, Y.; Paturi, R.; Saurabh, S.; Wahlström, M. On problems as hard as CNF-SAT. *ACM Trans. Algorithms* **2016**, *12*, 41. [CrossRef]
33. Nederlof, J. Space and Time Efficient Structural Improvements of Dynamic Programming Algorithms. Ph.D. Thesis, University of Bergen, Bergen, Norway, 2011.
34. Drucker, A.; Nederlof, J.; Santhanam, R. Exponential Time Paradigms Through the Polynomial Time Lens. In Proceedings of the 24th Annual European Symposium on Algorithms (ESA 2016), Aarhus, Denmark, 22–24 August 2016; Volume 57.
35. Lokshtanov, D.; Nederlof, J. Saving space by algebraization. In Proceedings of the 42th ACM Symposium on Theory of Computing, Cambridge, MA, USA, 6–8 June 2010; pp. 321–330.
36. Pilipczuk, M.; Wrochna, M. On Space Efficiency of Algorithms Working on Structural Decompositions of Graphs. In Proceedings of the 33rd Symposium on Theoretical Aspects of Computer Science, Orléans, France, 17–20 February 2016; Ollinger, N., Vollmer, H., Eds.; Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik: Wadern, Germany, 2016; Volume 47, pp. 57:1–57:15.
37. Diestel, R. *Graph Theory*, 4th ed.; Springer: Heidelberg, Germany, 2010.
38. Lindell, S. A logspace algorithm for tree canonization. In Proceedings of the Twenty-Fourth Annual ACM Symposium on Theory of Computing, Victoria, BC, Canada, 4–6 May 1992; pp. 400–404.
39. Björklund, A.; Husfeldt, T.; Kaski, P.; Koivisto, M. Fourier meets Möbius: Fast subset convolution. In Proceedings of the Thirty-Ninth Annual ACM Symposium on Theory of Computing, San Diego, CA, USA, 11–13 June 2007; pp. 67–74.
40. Oelschlägel, T. Graph Partitioning Problems on Graphs of Bounded Treedepth. Master's Thesis, RWTH Aachen University, Aachen, Germany, 2016.
41. Telle, J.A.; Proskurowski, A. Algorithms for vertex partitioning problems on partial *k*-trees. *SIAM J. Discret. Math.* **1997**, *10*, 529–550. [CrossRef]
42. Lampis, M. Algorithmic meta-theorems for restrictions of treewidth. *Algorithmica* **2012**, *64*, 19–37. [CrossRef]
43. Gajarsky, J.; Hliněný, P. Faster deciding MSO properties of trees of fixed height, and some consequences. In Proceedings of the IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2012), Hyderabad, India, 15–17 December 2012; Volume 18.