algorithms

MDPI

Article

Recent Advances in Positive-Instance Driven Graph Searching [†]

Max Bannach [1,*] and Sebastian Berndt [2,*]

[1] Institute for Theoretical Computer Science, Universität zu Lübeck, 23562 Lübeck, Germany
[2] Institute for IT Security, Universität zu Lübeck, 23562 Lübeck, Germany
[*] Correspondence: bannach@tcs.uni-luebeck.de (M.B.); s.berndt@uni-luebeck.de (S.B.)

[†] This paper is an extended version of our paper published in WADS 2019, Edmonton, AB, Canada, 5–7 August 2019.

Abstract: Research on the similarity of a graph to being a tree—called the *treewidth* of the graph—has seen an enormous rise within the last decade, but a practically fast algorithm for this task has been discovered only recently by Tamaki (ESA 2017). It is based on dynamic programming and makes use of the fact that the number of positive subinstances is typically substantially smaller than the number of all subinstances. Algorithms producing only such subinstances are called *positive-instance driven* (PID). The parameter *treedepth* has a similar story. It was popularized through the graph sparsity project and is theoretically well understood—but the first practical algorithm was discovered only recently by Trimble (IPEC 2020) and is based on the same paradigm. We give an alternative and unifying view on such algorithms from the perspective of the corresponding configuration graphs in certain two-player games. This results in a single algorithm that can compute a wide range of important graph parameters such as treewidth, pathwidth, and treedepth. We complement this algorithm with a novel randomized data structure that accelerates the enumeration of subproblems in positive-instance driven algorithms.

Keywords: treewidth; pathwidth; treedepth; graph searching; positive-instance driven; color coding

check for updates

Citation: Bannach, M.; Berndt, S. Recent Advances in Positive-Instance Driven Graph Searching. *Algorithms* **2022**, *15*, 42. https://doi.org/10.3390/a15020042

Academic Editors: Frank Werner and Carla Piazza

Received: 15 November 2021
Accepted: 24 January 2022
Published: 27 January 2022

1. Introduction

Graph decompositions are an important tool in modern algorithmic graph theory that provide a structured representation of a graph. A graph decomposition comes along with a *width measure* that indicates how well a graph can be decomposed. Many problems that presumably *cannot* be solved in polynomial time on general graphs *can* be solved efficiently on graphs that admit a certain decomposition of small width [1].

The most prominent example of a width measure is *treewidth*, which (on an intuitive level) measures the similarity of a graph to a tree. This parameter is a cornerstone of parameterized algorithms [2] and its success has led to its integration into many different fields. For instance, treewidth has been studied in the context of machine learning [3–5], model-checking [6,7], SAT-solving [8–10], QBF-solving [11,12], CSP-solving [13,14], or ILPs [15–19]. Tools such as Jdrasil [20] that compute tree decompositions of minimum width or that try to find good heuristic solutions are actively used, for instance, in the analysis of large SPARQL query logs [21] or in propositional model counting [22–24]. A large-scale experimental study that classifies real-world data sets according to their treewidth was recently performed by Maniu, Senellart, and Jog [25].

However, treewidth is often too general and we require more restrictive width measures in order to obtain an algorithmic advantage [26,27]. Close relatives of treewidth are the width measures *pathwidth* and *treedepth*, which (again on an intuitive level) measure the similarity of the graph to a path or a star, respectively. These graph parameters can be naturally ordered in the sense that some of them are more restrictive than others, that is, a graph of bounded treedepth has bounded pathwidth as well, and a graph of bounded pathwidth has also bounded treewidth. Tools that compute treedepth decompositions of small width can be used to efficiently solve the mixed Chinese postman problem [26] or to design algorithms for mixed integer linear programs [28]. Pathwidth was recently used as structural parameter to prove that the well-known vertex cover problem can be solved in polynomial time with high probability on hyperbolic random graphs (which are frequently used to model real-world networks) [29].

No matter which of these width measures we wish to utilize for a task at hand, we have to be able to compute it quickly. More crucially, most algorithms also need a witness in the form of a *decomposition.* A lot of *theoretical* research has been performed in this direction [30,31] and width measures such as treewidth can even be computed in linear fpt-time by Bodlaender's famous algorithm [32]. Unfortunately, it is known that this algorithm does *not* work in practice due to huge constants [33] (the same holds for similar algorithms for the other width measures). When it comes to implementations, small progress has been made with classical exact exponential time algorithms, which are only able to solve graphs with about 50–100 vertices. See, for instance, the QuickBB algorithm for treewidth [34], the algorithm by Coudert, Mazauric, and Nisse for pathwidth [35], or a recent branch-and-bound algorithm for treedepth [36]. More progress was made with heuristics, see, for instance, [37–39] for treewidth, the algorithm by Kobayashi et al. for pathwidth [40], and Section 24 in [41] for treedepth. We may argue that, indeed, a heuristic is sufficient, as the attached solver will work correctly independently of the width of the provided decomposition—and the heuristic may produce a decomposition of "small enough" width. However, even a small error, something as "off by 5", may put the parameter to a computationally intractable range, as the dependency on the width is usually at least exponential. For instance, it was observed that small changes (even by just one or two) in the width of a tree decomposition can have a huge impact on the inference time in probabilistic networks [42]. A similar effect was observed in recent advances of implementing a lightweight model checker for a fragment of MSO [7], where it turned out to be beneficial to invest additional time in order to obtain an optimal tree decomposition rather than relying on faster heuristics. It is therefore a natural and important task to build practical fast algorithms to determine parameters such as the treewidth, pathwidth, or treedepth of a graph exactly.

To tackle this issue, the fpt-community came up with a contest, the Parameterized Algorithms and Computational Experiments (PACE) challenge [43–45], with the goal of finding new exact algorithms to determine the exact treewidth or treedepth of a graph. Besides many, one important result of the challenge was a new combinatorial algorithm due to Tamaki, which computes the treewidth of an input graph exactly and astonishingly fast on a wide range of instances. An implementation of this algorithm by Tamaki himself [46] won the corresponding track in the PACE challenge in 2016 [43] and an alternative implementation due to Larisch and Salfelder [47] won in 2017 [44]. In 2020, many participants adapted the original algorithm to treedepth [45,48,49].

Tamaki's original algorithm is based on a dynamic program by Arnborg, Corneil, and Proskurowski [50] for computing tree decompositions. This algorithm has a game theoretic characterization that can be used to generalize the algorithm to other width measures such as treedepth. Tamaki has improved his algorithm for the second iteration of the PACE by applying his framework to the algorithm by Bouchitté and Todinca [51,52], see, for instance, [53] for a recent evaluation of this version of the algorithm. While Bouchitté and Todinca's algorithm has a game theoretic characterization as well [54], it is still unclear how it can be generalized to other width measures.

We focus on Tamaki's first algorithm and characterize it in a unifying way that allows us to compute not just the treewidth of a graph but, with the same algorithm, the pathwidth, treedepth, q-branched treewidth, and dependency treewidth as well. Perhaps even more importantly, our description of the algorithm in a game theoretic framework is simple and intuitive and can be implemented quite directly. In fact, our treewidth and treedepth solvers Jdrasil and PID* are based on this characterization [20,49]. The detailed contributions of this paper are the following:

Contribution I: A Simple Description of Tamaki's First Algorithm.
We describe Tamaki's algorithm as a well-known graph searching game. This provides a link to known theory and allows us to analyze the algorithm in depth.
Contribution II. Extending Tamaki's Algorithm to Other Parameters
The game theoretic point of view allows us to extend the algorithm naturally to various other parameters—including pathwidth and treedepth.
Contribution III: A Novel Randomized Data Structure.
The bottleneck in positive-instance driven algorithms is the enumeration of already computed solutions. We present a lazily constructed randomized data structure that, in contrast to existing data structures for this task, provides a guarantee that certain useless solutions are not enumerated with high probability.

1.1. Related Work

The concepts of pathwidth and treewidth were rediscovered several times in the literature [55–57]. Treedepth was discovered and analyzed by Nešetřil and de Mendez in their study of sparsity [58]. The game theoretic characterization of treewidth goes back to Seymour and Thomas [59]. Kirousis and Papadimitriou have studied a similar game for pathwidth [60], and Giannopoulou, Hunter and Thilikos have studied game theoretic approaches for treedepth [61]. The generalized version of this game, which we will use in this paper, was introduced by Fomin, Fraigniaud, and Nisse [62].

The potential success of positive-instance driven algorithms was demonstrated by Tamaki, who implemented a positive-instance driven version of Arnborg, Corneil, and Proskurowski's treewidth algorithm for the PACE 2016 [46]. This algorithm outperformed all other submissions (including one by the authors) by far [43]. The winning algorithm of the PACE 2017 also "is basically Hisao Tamakis implementation" [44,47]. Tamaki later adapted his algorithm to the algorithm by Bouchitté and Todinca in his pioneering work introducing positive-instance driven dynamic programming [52]. Ongoing research constantly improves the performance of this version of the algorithm [39,53,63].

Positive-instance driven dynamic programming also played a key role in many submissions for the PACE 2020, where the goal was to compute the treedepth of a graph rather than the treewidth [45], for instance, in the winning submission by Trimble [48]. However, it should be noted that in contrast to treewidth, the dominance of positive-instance driven algorithms compared to other strategies (such as enumerating minimal separators [64,65]) was less dramatic for the computation of treedepth [45].

Besides the development of dedicated algorithms for these graph parameters, there is also ongoing research in implementing a unifying way of computing all these parameters using SAT or MAXSAT solvers. See, for instance, [20,66–71].

1.2. Organization of This Paper

We provide some preliminaries about graphs and their decompositions in Section 2. We then invite the reader, in Section 3, to a gentle introduction to positive-instance driven graph searching with the example of computing the treedepth of a graph—while not directly necessary for the remaining paper, this section should make it easier to follow the more general approach presented later on. Section 4 lifts the game theoretic approach to a unifying algorithm that allows us to compute not just the treewidth or treedepth of a given graph but also its pathwidth and dependency treewidth. The somewhat technical proof of Theorem 2 is only sketched within the main text and we dedicate Appendix A to fill in the details.

After the main part of the paper, in Section 5 we identify a bottleneck of the positive-instance driven graph searching approach that was observed experimentally in various solvers. We extend a known data structure used by such algorithms, called *block sieve*, with a randomized component based on the well-known color coding technique. We illustrate the gained performance exemplarily in our treedepth solver PID* in Section 6.

We conclude our findings and discuss further research directions in Section 7.

1.3. Difference to the Conference Paper

This paper is an extended version of our paper *Positive-Instance Driven Dynamic Programming for Graph Searching* presented at WADS 2019 [72]. In the conference version, we already presented the unifying approach now discussed in Section 4. Because of this, the structure and content of this section have barely changed. In the present work, we improve the presentation of the algorithm and extend the approach further to directed treewidth (Section 4.6).

A new contribution, compared to [72], is the gentle introduction to positive-instance driven graph searching in Section 3. It is based on insights we gained during the development of our treedepth solver PID* [49] and its aim is to help the reader in following the more general algorithm presented later.

A new technical contribution is the lazily constructed randomized data structure (dubbed *color coding sieve*) presented in Section 5. The most expensive part of a positive-instance driven algorithm (be it for treewidth, treedepth, or any other width measure) is the enumeration of already computed positive subproblems that are *compatible* with each other [20,48,49,52,63]. Previous solvers use a data structure called *block sieves* to accelerate this process, which are basically set tries that store the positive subproblems and that can prune *some* non-compatible elements during the enumeration process. However, set tries are cumbersome and do *not* provide any guarantees on the number of non-compatible elements that actually get pruned. Our color coding sieves, in contrast, are (i) much more compact, as they are lazily constructed, (ii) use randomization to provide a guarantee that non-compatible elements are pruned with high probability, and (iii) allow a trade-off between the time spend in the data structure and the amount of elements that will be pruned. We extend this theoretical analysis by an implementation of the data structure in our treedepth solver PID* and an experimental evaluation (Sections 6).

2. Preliminaries: Graphs and Their Decompositions

A *digraph* $G = (V(G), E(G))$ is a tuple containing a set $V(G)$ of *vertices* and a binary *edge* relation $E(G) \subseteq \{\, (v,w) \mid v \neq w \wedge v,w \in V \,\}$. The *neighborhood* of a vertex $v \in V(G)$ is defined as $N_G(v) = \{\, w \mid (v,w) \in E(G) \,\}$ and we define $N_G[v] = N_G(v) \cup \{v\}$. For a set $X \subseteq V$, we write $N_G(X) = \cup_{x \in X} N(x)$ and $G \setminus X = (V(G) \setminus X, E(G) \setminus \{\, e \in E(G) \mid X \cap e \neq \emptyset \,\})$ for the graph obtained by *deleting* the vertices in X from G. The subgraph *induced* by X is defined as $G[X] = G \setminus (V(G) \setminus X)$. If G is clear from the context, we may simply refer to $V(G)$ and $E(G)$ as V and E, respectively, and we may drop the subscript "G" in the other definitions. A graph is *undirected* if its edge relation is symmetric and we may write abbreviations like $\{u,v\} \in E$ instead of $(u,v),(v,u) \in E$ for such graphs.

Graph Decompositions

Let $G = (V,E)$ be an undirected graph. A *tree decomposition* (T, ι) of G is a tree T and a mapping ι from the nodes of T to subsets of V (which we call *bags*) such that (i) for every $v \in V$, the set $\{\, x \mid v \in \iota(x) \,\}$ is non-empty and connected in T, and (ii) for every edge $\{v,w\} \in E$ there is a node y in T with $\{v,w\} \subseteq \iota(y)$. The *width* of a tree decomposition is the maximum size of one of its bags minus one. The *treewidth* $\mathrm{tw}(G)$ is the minimum width over all tree decompositions of G and its *pathwidth* $\mathrm{pw}(G)$ is the minimum width over all tree decompositions of G in which T is a path. Furthermore, its *treedepth* $\mathrm{td}(G)$ is the minimum width over all tree decompositions in which T can be rooted in such a way that for all nodes x, y of T we have $\iota(x) \subsetneq \iota(y)$ if y is a descendant of x. Finally, its *q-branched treewidth* $\mathrm{tw}_{\mathrm{q}}(G)$ is the minimum width over all tree decompositions of G that can be rooted such that there are at most q vertices with more than one descendant on any root-leaf path. Intuitively, the treewidth of a graph measures how similar the graph is "to being a tree", in the same sense pathwidth measures the similarity to a path and treedepth the similarity to a star. The q-branched treewidth allows to study the phase transition between the pathwidth and the treewidth of a graph. Loosely speaking, we can study the trade-off between the *width* of a decomposition and its "complexity". A path decomposition has the highest width, but it is not "complex". In contrast, a tree decomposition has the smallest width but may be arbitrary "complex", as it may contain a large number of nodes of high degree. It is well known that we have $\mathrm{tw}(G) \leq \mathrm{pw}(G) \leq \mathrm{td}(G) \leq \mathrm{tw}(G) \cdot \log_2 n$ [58] and, by definition, we have $\mathrm{tw}(G) = \mathrm{tw}_\infty(G)$ and $\mathrm{pw}(G) = \mathrm{tw}_0(G)$. Figure 1 shows an example graph with multiple tree decompositions that minimize the various invariants defined above.

Another important variant of these parameters is *dependency treewidth*, which is used primarily in the context of quantified Boolean formulas [12]. Intuitively, this notion captures the idea that vertices in the graph are dependent on each other and thus need to be processed in a certain order. More formally, for a graph $G = (V, E)$ and a partial order $\lessdot$ of V, the dependency treewidth $\mathrm{dtw}(G)$ is the minimum width of any tree decomposition (T, ι) with the following property: Consider the natural partial order $\leq_T$ that T induces on its nodes, where the root is the smallest elements and the leaves form the maximal elements; define for any $v \in V$ the node $F_v(T)$ that is the $\leq_T$-minimal node t with $v \in \iota(t)$ (which is well defined); then define a partial order $<_{\mathcal{T}}$ on V such that $u <_{\mathcal{T}} v$ if, and only if, $F_u(T) \leq_T F_v(T)$; finally, for all $u, v \in V$, it must hold that $F_u(T) <_T F_v(T)$ implies that $u \lessdot v$ does not hold.

There are multiple generalizations of treewidth for *digraphs*. We will use the so-called *D-width*, which was introduced by Safari [73] and is based on *D-decompositions*. Such a decomposition is a tuple (T, ι) as above, but with the conditions that (i) for every strongly connected subset $S \subseteq V$ there is at least one x in T with $\iota(x) \cap S \neq \emptyset$, and (ii) the subgraph of T induced by $\{\, \{x, y\} \mid \iota(x) \cap \iota(y) \cap S \neq \emptyset \,\}$ is connected. The *D-width* of a digraph G, denoted by $\overrightarrow{\mathrm{tw}}(G)$, is the minimal width of any D-decomposition of G.

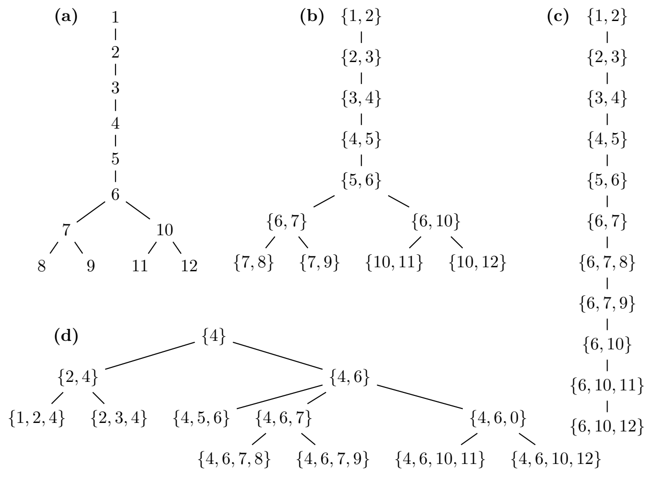


Figure 1. Various tree decompositions of an undirected graph $G = (V, E)$ shown at (**a**). The decompositions justify (**b**) $\mathrm{tw}(G) \leq 1$, (**c**) $\mathrm{pw}(G) \leq 2$, and (**d**) $\mathrm{td}(G) \leq 3$. With respect to q-branched treewidth, the decompositions also justify (**b**) $\mathrm{tw}_2(G) \leq 1$ and (**c**) $\mathrm{tw}_0(G) \leq 2$.

3. A Gentle Introduction to Positive-Instance Driven Graph Searching

Before we describe a unifying algorithm that can compute all the graph decompositions shown in Figure 1, we will illustrate the positive-instance driven graph searching technique in a simpler example. Our goal in this section is to describe a game theoretic positive-instance driven algorithm for the width measure *treedepth* alone. For that end, let us use the following equivalent definition of this parameter: the treedepth of a graph $G = (V, E)$ is the minimum height of a rooted forest F such that G is a subgraph of the closure of F. This forest is called a *treedepth decomposition* of G. Since the treedepth of a graph is the maximum treedepth of its connected components, we can restrict ourselves to connected graphs. In this case, a treedepth decomposition of G is an *elimination tree.*

3.1. Graph Searching

Many graph decompositions have game theoretic characterizations in the form of *vertex pursuit-evasion games* [74]. Such games, which are also known as *graph searching* or *cops and robber,* are played by two players on an undirected graph $G = (V, E)$. In the game for treedepth, the first player places a team of k *searchers* iteratively on the vertices of G, while the second player controls a single *fugitive* that hides in a connected component. The game is played in rounds as follows [61,75]: Initially, the fugitive hides in the vertex set $C = V$ (which we assume to be connected). The vertices in C are said to be *contaminated.* In each round, both players perform one action:

1. The searchers pick a vertex $v \in C$ on which they want to place the next searcher. We say they *clean* the vertex v.
2. The fugitive responds by picking a component C' of $G[C \setminus \{v\}]$. The contaminated area is reduced to C' and the game proceeds only on this subgraph.

The game ends when the contaminated area shrinks to the empty set or if the searchers have placed all k members of their team and C is still non-empty. In the first case, the graph was *cleaned* and the fugitive was *caught;* in the second case, the fugitive *escaped*. The searchers win if they catch the fugitive, otherwise the fugitive wins. Note that in this version of the game, the searchers are not allowed to remove an already placed searcher from the graph. The game is therefore monotone and always ends after, at most, k rounds. Further observe that the fugitive is *visible* in the sense that the searchers know in which connected component she hides—in contrast, an invisible fugitive could hide in subgraphs that are not connected (which notably complicates the arguments). The game is illustrated in Figure 2 on a small graph with eight vertices.

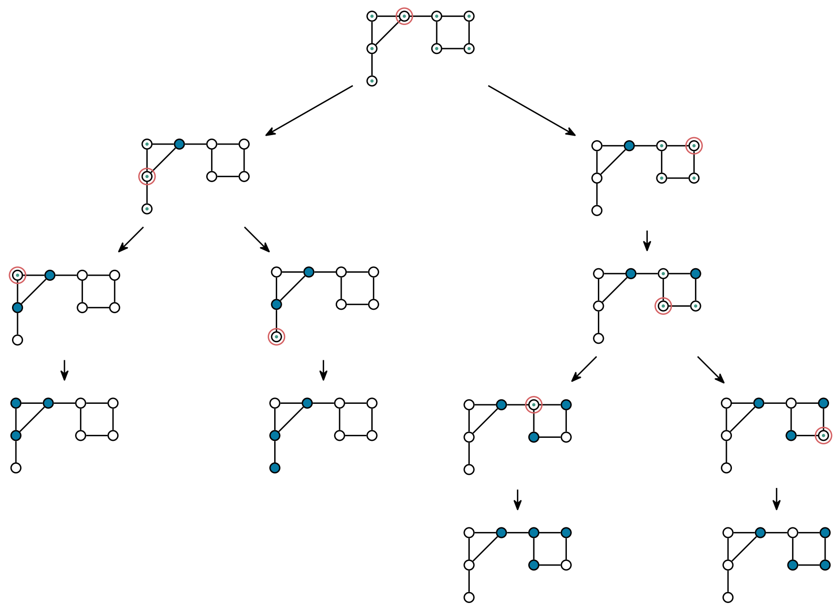


Figure 2. An illustration of the graph searching game for treedepth on the 8 vertex graph shown at the very top. Vertices that contain a green dot are currently *contaminated*. The searchers will *place* a searcher on the vertex with a red circle in the next round. The *cleaned* vertices (on which a searcher stands) are filled with blue. The arrows indicate the various choices of the fugitive. The diagram proves that 4 searchers have a winning strategy on this specific graph.

The configurations of this game are *blocks,* which are tuple (C, ρ) with $\rho \in \mathbb{N}$ and $C \subseteq V$ being a connected subgraph with $|N(C)| + \rho \leq k$. Informally, C is the (connected) contaminated area, and ρ is the number of remaining searchers. We require $|N(C)| + \rho \leq k$ as the neighborhood of C has to be cleaned in order to have C as contaminated area. We denote the set of blocks of the game played on a graph G with k searchers by $\mathcal{B}(G, k)$. Two blocks (C_1, ρ_1) and (C_2, ρ_2) *intersect* if $C_1 \cap C_2 \neq \emptyset$, $N_G(C_1) \cap C_2 \neq \emptyset$ or $C_1 \cap N_G(C_2) \neq \emptyset$. The *start configuration* of the game is (V, k) and the *winning configurations* for the searchers are $(\emptyset, \rho \geq 0)$. We say the searchers have a *winning strategy* on a block (C, ρ) if, starting with configuration (C, ρ), they can ensure to reach a winning configuration no matter how the fugitive acts. The set of blocks that guarantee such a strategy is the *winning region*, which we denote by $\mathcal{R}(G, k) \subseteq \mathcal{B}(G, k)$—the elements in this region are said to be *positive*. The connection between the game and the width measure treedepth is established by the following fact:

Fact 1 ([61]). *Let $G = (V, E)$ be a graph and $k \in \mathbb{N}$. Then $(V, k) \in \mathcal{R}(G, k) \iff \mathrm{td}(G) \leq k$.*

3.2. Simple Positive-Instance Driven Graph Searching

By Fact 1, it is sufficient to compute the set $\mathcal{R}(G, k)$ in order to test whether the treedepth of G is at most k. One way of doing so would be to first compute $\mathcal{B}(G, k)$, then build an auxiliary graph on top of this set, and finally compute $\mathcal{R}(G, k)$ by solving reachability queries on this auxiliary graph. We can estimate the number of configurations with $|\mathcal{B}(G, k)| \leq (k+1) \cdot n^{k+1}$, as there are n^k possible ways of placing k searchers on an n-vertex graph—at most n connected components adjacent to a separator—and we have $\rho \in \{0, \ldots, k\}$. Therefore, the sketched algorithm achieves a run time of $n^{O(k)}$, which is not feasible in practice for even moderate values of k.

To make the game theoretic approach feasible, we present an *output-sensitive* algorithm that computes just $\mathcal{R}(G, k)$— without "touching" the rest of $\mathcal{B}(G, k)$. Figure 3 illustrates why we may hope that $\mathcal{R}(G, k)$ is smaller than $\mathcal{B}(G, k)$. In the remainder of this section, we will develop some intuition about how to compute the set $\mathcal{R}(G, k)$. The formal details are postponed to the unifying version of the algorithm in the next section.

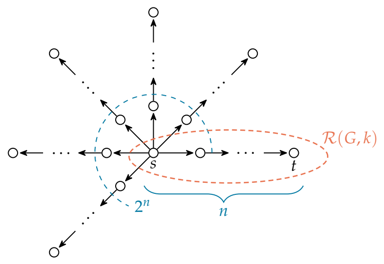


Figure 3. A directed spider with 2^n legs each of length n that could be the auxiliary graph of a graph searching game—i. e., the vertex set is $\mathcal{B}(G, k)$. Assume s is the start configuration and, for the sake of argument, that t is the sole winning configuration. Then $\mathcal{R}(G, k)$ contains only n elements, while the whole game has $2^n \cdot n$ configurations.

Surely, we cannot start at some block, say (V, k), and just simulate the game in a top-down fashion—we could touch a lot of blocks in $\mathcal{B}(G, k) \setminus \mathcal{R}(G, k)$ without even noticing it (in Figure 3, that would mean starting a graph traversal from s, which could explore any leg of the spider). After all, we do not know whether $(V, k) \in \mathcal{R}(G, k)$. We do know, however, that $(\emptyset, 0)$ is a winning configuration (vertex t in Figure 3). So, let us start with the set $\mathcal{R} = \{\, (\emptyset, 0) \,\}$ and then try to grow it to $\mathcal{R}(G, k)$ in a bottom-up fashion. We can first ask which configurations of the game lead to $(\emptyset, 0)$, i. e., what are configurations in which the searchers immediately win in the next round? These are the configurations $(\{v\}, 1)$ with $|N(v)| < k$, as in these the searchers can surround the fugitive and have a searcher left to place it on top of her (in Figure 3, this corresponds to the predecessor of t).

Now, assume that we have currently a set $\mathcal{R} \subseteq \mathcal{R}(G, k)$ that had already grown a little. What does a configuration $(C, \rho) \in \mathcal{R}(G, k) \setminus \mathcal{R}$ that is "close to" $\mathcal{R}$ look like (that is, we wish to traverse the s-t-path of Figure 3 in reverse direction)? The set C is connected by definition and, since the searchers have a winning strategy from (C, ρ), there is a vertex $v \in C$ such that $G[C \setminus \{v\}]$ has connected components $C_1, \ldots, C_q$ ($q = 1$ is possible) with $(C_i, \rho - 1) \in \mathcal{R}$ for all $i \in \{1, \ldots, q\}$. To find (C, ρ), we first guess the vertex v, scan the sets of pairwise non-intersecting blocks $X \subseteq \{\, (C',\rho') \in \mathcal{R} \mid v \in N_G(C') \wedge \rho' < \rho \,\}$, and generate the new blocks $\big(\bigcup_{(C',\rho')\in X} C' \cup \{v\}, 1 + \max_{(C',\rho')\in X} \rho'\big)$. Note that we can prune a set X if $\big|\big(N_G(\bigcup_{(C',\rho')\in X} C')\big| > k - \max_{(C',\rho')\in X} \rho'$, as this neighborhood has to be cleaned by the searchers before they clean one of the components C'. This will be utilized by the randomized data structure that we develop later in Section 5. Figure 4 illustrates this step of the algorithm, which we call a *glue operation.* We refer the reader who is interested in the details of implementing this algorithm for treedepth efficiently to the description of our solver PID* [49]. In the next section, we present a formal and more general version of the sketched strategy—which does not just work for treedepth, but for treewidth, pathwidth and many other width measures, too.

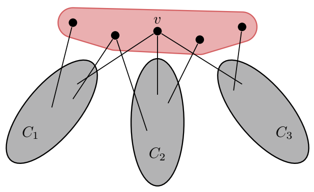


Figure 4. The *glue operation:* We have already guessed the vertex v and currently consider the set $X = \{\, (C_1,\rho_1),(C_2,\rho_2),(C_3,\rho_3) \,\}$. Note that all blocks are adjacent to v and are pairwise non-intersecting. The combined neighborhood is highlighted. This area is not allowed to be larger than $k - \max\{\rho_1,\rho_2,\rho_3\}$, as the searchers must clean it before they can proceed the search on one of the blocks. From this situation, we generate the block $(C_1 \cup C_2 \cup C_3 \cup \{v\}, 1 + \max\{\rho_1,\rho_2,\rho_3\})$.

3.3. Alternative Characterization

Let us briefly sketch an interpretation of the algorithm without the game theoretic point of view. One can think of it as a procedure that, given a connected graph $G = (V,E)$, lists ever growing treedepth decompositions of subgraphs of G. In detail, a collection $\mathcal{T}_d$ of trees of depth at most d for some $d \geq 1$ is managed—starting with the trivial trees of depth 1 (i. e., single vertices) we have $\mathcal{T}_1 \subseteq V$. Then, for ever larger d, the set $\mathcal{T}_d$ is computed by picking a new root $r \in V$ and a collection of previously computed trees $\mathcal{S} \subseteq \bigcup_{i=1}^{d-1} \mathcal{T}_i$, and by arranging the trees of $\mathcal{S}$ to a new tree by connecting their roots to r. Of course, in order to obtain a valid treedepth decomposition, the elements of $\mathcal{S}$ must be pairwise disjoint and non-adjacent. The selection of the set $\mathcal{S}$ is what was the glue operation and what will be a *universal step* in the algorithm presented in the next section, and the value d corresponds to $k - \rho$ and will be the *distance* that we will compute. Let us stress out that, in this characterization, the trees in $\mathcal{S}$ are, indeed, trees and thus connected. This is an invariant that the following algorithm cannot guarantee for all parameters but which improves the performance of the algorithm if met.

3.4. Execution Modes

It is worth noting that positive-instance driven algorithms only solve the problem for a fixed k. Hence, to obtain an *optimal* treedepth decomposition (or any other decomposition), one has to run the algorithm for various values of k. The order in which these values are tested is called the *execution mode* of the algorithm. Positive-instance driven algorithms differ from other strategies with respect to values that are "easy". For instance, branch-and-bound algorithms usually solve the problem quickly if k overestimates the optimal value. However, overestimating k means *more* positive subproblems, making these instances hard for positive-instance driven algorithms. On the other hand, underestimating the optimal value yields instances that are usually hard but that are solved quickly by positive-instance driven algorithms as there are only few positive instances. Positive-instance driven graph searching is, thus, especially suited to compute lower bounds [63]. Therefore, the typical execution mode is to start with $k = 1$ and to increase this lower bound until either an optimal solution was found or a heuristically obtained upper bound is met [48,49].

4. A Unifying Take on Positive-Instance Driven Graph Searching

Our goal is to generalize the algorithm that we just sketched in Section 3.2 to other graph measures such as treewidth and pathwidth. When we generalize from treedepth to other parameters, there are two main obstacles that we have to face. First, the sketched algorithm for treedepth "guessed" a root vertex in every iteration and glued already computed blocks to it. In more flexible graph decompositions such as tree decompositions, the root no longer is a single vertex but a *set of up to k vertices.* Clearly, we cannot guess such a set and, hence, have to compute it implicitly. Second, the treedepth algorithm explicitly carried the number of remaining searchers ρ around (they were part of the blocks). In more flexible graph measures, we may remove and reuse already placed searchers and, thus, have to encode this information in a different way. We will handle this issue by splitting the computation into two phases, where the first phase mainly computes a configuration graph (intuitively, this graph contains blocks for various values of ρ) and where the second phase then computes distances in this configuration graph (hence, computes best possible ρ-values for the blocks). However, let us postpone the fiddling with distance queries to Section 4.5 and focus solely on the more general search game in the first part of this section.

We study graph searching in a setting proposed by Fomin, Fraigniaud, and Nisse [62]. The input is again an undirected graph $G = (V, E)$ and a number $k \in \mathbb{N}$, and the question is whether a team of k searchers can catch an *invisible* fugitive on G by the following set of rules: At the beginning, the fugitive picks a connected component C of G in which she hides—since the fugitive is invisible, in contrast to Section 3.2, the game is continued on G and *not* on $G[C]$ and we say that the set V is *contaminated*. In each round, the player now follows a similar procedure as in the previous game, but the searchers have a larger set of possible moves:

1. The searchers perform one of the following:
   - *Place* a searcher on a contaminated vertex;
   - *Remove* a searcher from a vertex;
   - *Reveal* the current position of the fugitive.
2. The fugitive responds as follows:
   - If the searchers *place* or *remove* a searcher, the fugitive adapts her connected component by adding or removing the vertex, respectively. (This may join multiple components or disconnect the current component, in which case the fugitive selects one of the resulting connected components).
   - If the searchers perform a *reveal*, the fugitive responds by uncovering her current connected component C. The contaminated area is reduced to C.

We follow the same terminology as before, i. e, we say a contaminated vertex becomes *clean* if a searcher is placed on it. In contrast to the previous game, a vertex v may now become *recontaminated* if a searcher is removed from it and there is a contaminated vertex adjacent to v. The searchers win the game if they manage to clean all vertices, i. e., if they catch the fugitive; the fugitive wins if, at any point, a recontamination occurs or if she can escape infinitely long. Note that this implies that the searchers have to catch the fugitive in a *monotone* way. A priori, one could assume that the later condition gives the fugitive an advantage (recontamination could be necessary for the cleaning strategy); however, a crucial result in graph searching is that "recontamination does not help" in all variants of the game that we consider [59,61,76–78].

4.1. *Entering the Arena and the Colosseum*

Our primary goal is to determine whether the searchers have a winning strategy. A folklore algorithm for this task is to construct an alternating graph called *the arena*: $\mathrm{arena}(G,k) = ((V_s \cup V_f), E_{\mathrm{arena}})$ that contains for each position of the searchers ($S \subseteq V$ with $|S| \leq k$) and each position of the fugitive ($f \in V$) two copies of the vertex (S, f), one in V_s and one in V_f (see, for instance, Section 7.4 in [2]). Vertices in V_s correspond to a configuration in which the searchers perform the next move (they are existential) and vertices in V_f correspond to fugitive moves (they are universal). The edges E_{arena} are constructed according to all possible moves. The question is now whether there is an alternating path from a start configuration to some configuration in which the fugitive is caught. Since alternating paths can be computed in linear time ([79], Section 3.4), we immediately obtain an $O(n^{k+1})$ algorithm.

Modeling a configuration of the game as tuple (S, f) comes, however, with a major drawback: The size of the arena does directly depend on n and k and does *not* depend on some further structure of the input. For instance, the arena of a path of length n and any other graph on n vertices will have the same size for any fixed value k. As the major goal of parameterized complexity is to gain insight into structural parameters beyond the input size n, such a fixed-size approach is not what we are looking for.

A bit counter intuitive, we will tackle this problem by, firstly, defining an alternating graph that is *larger* than the arena: *the colosseum.* As it befits for any good colosseum, it is not only larger but, in particular, "prettier" than the arena (which here means that it adapts to the input structure of the graph).

Once we can capture the structure in the colosseum, we will introduce yet another alternating graph that, finally, is actually *small*. This graph, which will be a subset of the colosseum, is called *the pit*—where only true champions can survive!

4.2. Simplifying the Game

Before we define all the locations of potential gladiator fights in graph theoretic terms, let us start with some simplifications of the game. We restrict the fugitive as follows: Since she is invisible, there is no need for her to take regular actions. Instead, the only moment when she is actually active is when the searchers perform a reveal. If C is the set of contaminated vertices, consisting of the induced components $C_1, \dots, C_\ell$, a reveal will uncover the component in which the fugitive hides and, as a result, reduce C to C_i for some $1 \leq i \leq \ell$. The only task of the fugitive is to answer a reveal with such a number i. The complete process of the searcher performing a reveal, the fugitive answering it, and finally of reducing C to C_i is called a *reveal-move*.

We also restrict the searchers by the concept of *implicit searcher removal.* Let $S \subseteq V(G)$ be the set of vertices currently occupied by the searchers, and let $C \subseteq V(G)$ be the set of contaminated vertices. We call a vertex $v \in S$ *covered* if every path between v and C contains a vertex $w \in S$ with $w \neq v$.

Lemma 1. *A covered searcher can be removed safely.*

Proof. As we have $N(v) \cap C = \emptyset$, the removal of v will not increase the contaminated area. Furthermore, v cannot be recontaminated at a later point, unless a neighbor of v becomes recontaminated as well (in which case the game would already be over). □

Lemma 2. *Only covered searchers can be removed safely.*

Proof. Since for any other vertex $w \in S$ we have $N(w) \cap C \neq \emptyset$, the removal of w would recontaminate w and, hence, would result in a defeat of the searchers. □

Both lemmas together imply that the searchers never have to decide to remove a searcher but rather can do it *implicitly*. We restrict the possible moves of the searchers to a combined move of placing a searcher and *immediately* removing searchers from all covered vertices. This is called a *fly-move*. Observe that the sequence of original moves mimicked by a fly-move does not contain a reveal and, thus, may be performed independently of any action of the fugitive.

4.3. Building the Colosseum

We are now ready to define the colosseum. As for the arena, we could define it as an alternating graph. However, as only the searchers perform actions in the simplified game, we find it more natural to express this game as an *edge-alternating graph*—a generalization of alternating graphs. An edge-alternating graph is a triple $H = (V, E, A)$ consisting of a *vertex set* V, an existential edge relation $E \subseteq V \times V$, and a universal edge relation $A \subseteq V \times V$. The neighborhoods of a vertex v are the existential neighborhood $N_\exists(v) = \{\, w \mid (v, w) \in E \,\}$, the universal neighborhood $N_\forall(v) = \{\, w \mid (v, w) \in A \,\}$, and the complete neighborhood $N_H(v) = N_\exists(v) \cup N_\forall(v)$. An *edge-alternating* s-t-path is a set $P \subseteq V$ such that (i) $s, t \in P$ and (ii) for all $v \in P$ with $v \neq t$ we have either $N_\exists(v) \cap P \neq \emptyset$ or $\emptyset \neq N_\forall(v) \subseteq P$ or both. We write $s \prec t$ if such a path exists and define $\mathcal{R}(Q) = \{\, v \mid v \in Q \vee (\exists w \in Q \colon v \prec w) \,\}$ for $Q \subseteq V$ as the set of vertices on edge-alternating paths leading to Q. We say that an edge-alternating s-t-path P is *q-branched* if (i) H is acyclic and (ii) every (classical) directed path π from s to t in H with $\pi \subseteq P$ uses at most q universal edges.

For $G = \big(V(G), E(G)\big)$ and $k \in \mathbb{N}$, the colosseum(G, k) is the edge-alternating graph H with $V(H) = \{\, C \mid \emptyset \neq C \subseteq V(G)$ and $|N_G(C)| \leq k \,\}$ and the following edge sets: for all pairs $C, C' \in V(H)$ there is an existential edge $e = (C, C') \in E(H)$ if, and only if, $C \setminus \{v\} = C'$ for some $v \in C$ and $|N_G(C)| < k$; furthermore, for all $C \in V(H)$ with at least two components $C_1, \dots, C_\ell$ we have universal edges $(C, C_i) \in A(H)$.

The nodes of the colosseum are called *blocks* and, thinking of Section 3.2, we have $V(H) = \mathcal{B}(G, k)$ (but note that the definition of "block" has slightly changed). The *start configuration* of the game is the block $C = V(G)$, i. e., all vertices are contaminated. We define $Q = \{\, \{v\} \subseteq V(G) \colon |N_G(\{v\})| < k \,\}$ to be the set of *winning configurations*, as at least one searcher is available to catch the fugitive. Therefore, the searchers have a winning strategy if, and only if, $V(G) \in \mathcal{R}(Q)$ and we will therefore refer to $\mathcal{R}(Q) = \mathcal{R}(G, k)$ as the *winning region* (we did so similarly in Section 3.2, but we did not have the notation of edge-alternating graphs then). Observe that the colosseum is acyclic (that is, the digraph $(V(H), E(H) \cup A(H))$ is acyclic) as we have for every edge (C, C') that $|C| > |C'|$, and observe that Q is a subset of the sinks of this graph. Hence, we can test if $V(G) \in \mathcal{R}(Q)$ in time $O(|\operatorname{colosseum}(G, k)|)$. Finally, note that the size of colosseum(G, k) may be of order 2^n rather than n^{k+1}, giving us a slightly worse overall runtime. This larger structure is required to encode that the fugitive is invisible.

4.4. Fighting in the Pit

The sketched algorithms have running time proportional to the size of the arena and the colosseum. Both of these auxiliary graphs might be large, as the arena has fixed size of order $O(n^{k+1})$ while the colosseum may even have size $O(2^n)$. Additionally, both graphs can contain unnecessary configurations, that is, configurations not part of the winning region. In the light of dynamic programming, this is the same as listing all possible configurations, and in the light of positive-instance driven dynamic programming, we would like to list only the positive instances—which is exactly the winning region $\mathcal{R}(Q)$.

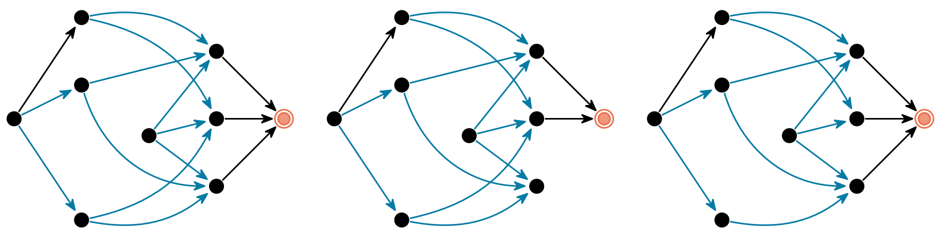
The *pit* pit(G, k) inside the colosseum is now formally defined as the subgraph of colosseum(G, k) induced by $\mathcal{R}(Q)$, that is, as the induced subgraph on the winning region. The key insight is that $|\operatorname{pit}(G, k)|$ may be smaller than $|\operatorname{colosseum}(G, k)|$ or even $|\operatorname{arena}(G, k)|$ on various graph classes. Our primary goal for this section will therefore be the development of an algorithm that computes the pit in time depending only on *the size of the pit* (rather the size of the arena or the colosseum).

The algorithm traverses the colosseum "backwards" starting from the winning configurations Q and uncovering $\mathcal{R}(Q)$ layer by layer. In order to achieve this, we need to compute the predecessors of a block C. This is easy if C was reached by a fly-move as we can simply enumerate the n possible predecessors. Reversing a reveal-move, that is, finding the universal predecessors, is significantly more involved. A simple approach is to test for every subset of already explored configurations if we can "glue" them together—as we did in the sketched algorithm in Section 3.2. However, this results in an even worse runtime of $2^{|\operatorname{pit}(G,k)|}$. To avoid this exponential blow-up, we require the following structural property of the colosseum.

Definition 1 (Universal Consistent)**.** *We say that an edge-alternating graph $H = (V, E, A)$ is* universal consistent *with respect to a set $Q \subseteq V$ if for all $v \in V \setminus Q$ with $v \in \mathcal{R}(Q)$ and $N_\forall(v) = \{w_1, \dots, w_r\}$ we have (1) $N_\forall(v) \subseteq \mathcal{R}(Q)$ and (2) for every $I \subseteq \{w_1, \dots, w_r\}$ with $|I| \geq 2$ there is a vertex $v' \in V$ with $N_\forall(v') = I$ and $v' \in \mathcal{R}(Q)$.*

Intuitively, Definition 1 implies that for every vertex with high universal-degree, there is a set of vertices that can be arranged in a tree-like fashion to realize the same adjacency relation. This allows us to glue only two configurations at a time and, thus, removes the exponential dependency. The definition is illustrated in Example 1.

Example 1. *Consider the following three edge-alternating graphs, where black edges are existential and the blue edges are universal. The set Q contains a single vertex that is highlighted. From left to right: the first graph is universal consistent, the second and third one are not. The second graph conflicts the condition that $v \in \mathcal{R}(Q)$ implies $N_\forall(v) \subseteq \mathcal{R}(Q)$, as the vertex on the very left is contained in $\mathcal{R}(Q)$ by the top path, while its universal neighbor on the bottom path is not contained in $\mathcal{R}(Q)$. The third graph conflicts the condition that $N_\forall(v) = \{w_1, \dots, w_r\}$ implies that for every $I \subseteq \{w_1, \dots, w_r\}$ with $|I| \geq 2$ there is a vertex $v' \in V$ with $N_\forall(v') = I$ and $v' \in \mathcal{R}(Q)$ as witnessed by the vertex with three outgoing universal edges.*



Lemma 3. *For every graph G and number k, the edge-alternating graph* $\text{colosseum}(G, k)$ *is universal consistent.*

Proof. For the first property, observe that "reveals do not harm": Searchers that can catch the fugitive without knowing where she hides, certainly can do so if they know.

For the second property, consider any configuration $C \in V(H)$ that has universal edges to $C_1, \dots, C_\ell$. By definition, we have $|N_G(C)| \leq k$ and $N_G(C_i) \subseteq N_G(C)$ for all $1 \leq i \leq \ell$. Therefore, we have for every $I \subseteq \{1, \dots, \ell\}$ and $C' = \cup_{i \in I} C_i$ that $N_G(C') \subseteq N_G(C)$ and $|N_G(C')| \leq k$ and, thus, $C' \in V(H)$. □

The algorithm for computing the pit, see Figures 5 and 6, runs in three phases: it first computes the set Q of winning configurations; then the winning region $\mathcal{R}(Q)$ (the vertices of $\text{pit}(G, k)$); and finally, it computes the edges of $\text{pit}(G, k)$.

Theorem 1. *The algorithm discover(G,k) finishes in at most $O\big(|\mathcal{R}(Q)|^2 \cdot |V|^2\big)$ steps and correctly outputs* $\text{pit}(G, k)$.

Proof. The algorithm computes Q in phase I, the winning region $\mathcal{R}(Q)$ in phase II, and the edges of $\text{colosseum}(G, k)[\mathcal{R}(Q)]$ in phase III. First observe that Q is correctly computed in phase I by the definition of Q.

For the correctness of the second phase, we show that the computed set $V(\text{pit}(G, k))$ equals $\mathcal{R}(Q)$. Let us refer to the set $V(\text{pit}(G, k))$ during the computation as K and observe that this is exactly the set of vertices inserted into the queue. We first show $K \subseteq \mathcal{R}(Q)$ by induction over the ith inserted vertex. The first vertex C_1 is in $\mathcal{R}(Q)$ as $C_1 \in Q$. Now, consider a $C_i \in K$. It was either added in Line 18 or Line 24. In the first case, there was a vertex $\tilde{C}_i \in K$ such that $C_i = \tilde{C}_i \cup \{v\}$ for some $v \in N(\tilde{C}_i)$. By the induction hypothesis we have $\tilde{C}_i \in \mathcal{R}(Q)$ and by the definition of the colosseum we have $(C_i, \tilde{C}_i) \in E(H)$ and, thus, $C_i \in \mathcal{R}(Q)$. In the second case, there were vertices $\tilde{C}_i, \hat{C}_i \in K$ with $C_i = \tilde{C}_i \cup \hat{C}_i$.

By the induction hypothesis, we have again $\tilde{C}_i, \hat{C}_i \in \mathcal{R}(Q)$. Let $t_1, \ldots, t_\ell$ be the connected components of $\tilde{C}_i$ and $\hat{C}_i$. Since the colosseum is universal consistent with respect to Q by Lemma 3, we have $t_1, \ldots, t_\ell \in \mathcal{R}(Q)$. By the definition of the colosseum, we have $N_\forall(C_i) = t_1, \ldots, t_\ell$ and, thus, $C_i \in \mathcal{R}(Q)$.

To see $\mathcal{R}(Q) \subseteq K$, consider for a contradiction the vertices of $\mathcal{R}(Q)$ in reversed topological order (recall that the colosseum is acyclic) and let C be the first vertex in this order with $C \in \mathcal{R}(Q)$ and $C \notin K$. If $C \in Q$, we have $C \in K$ by phase I and are complete, so assume otherwise. Since $C \in \mathcal{R}(Q)$, we have either $N_\exists(C) \cap \mathcal{R}(Q) \neq \emptyset$ or $\emptyset \neq N_\forall(C) \subseteq \mathcal{R}(Q)$. In the first case, there is a block $\tilde{C} \in \mathcal{R}(Q)$ with $(C, \tilde{C}) \in E(H)$. Block $\tilde{C}$, thus, precedes C in the reversed topological order and, by the choice of C, we have $\tilde{C} \in K$. Therefore, at some point, $\tilde{C}$ is extracted from the queue and, in Line 18, C would be added to K, a contradiction.

In the second case, there are vertices $t_1, \ldots, t_\ell \in \mathcal{R}(Q)$ with $N_\forall(C) = \{t_1, \ldots, t_\ell\}$. By the choice of C, we have again $t_1, \ldots, t_\ell \in K$. Since H is universal consistent with respect to Q, we have for every $I \subseteq \{1, \ldots, \ell\}$ that $\bigcup_{i \in I} t_i$ is contained in $\mathcal{R}(Q)$. In particular, the vertices $t_1 \cup t_2$, $t_3 \cup t_4$, $\ldots$, $t_{\ell-1} \cup t_\ell$ are contained in $\mathcal{R}(Q)$, and these elements are added to K whenever the t_i are processed (for simplicity, assume here that ℓ is a power of 2). Once these elements are processed, Line 24 will also add their union, that is, vertices of the form $(t_1 \cup t_2) \cup (t_3 \cup t_4)$. In this way, the process will add vertices that correspond to increasing subgraphs of G to K, resulting ultimately in adding $\bigcup_{i=1}^{\ell} t_i = C$ into K, which is the contradiction we have been looking for.

Once $\mathcal{R}(Q)$ is known, it is easy to compute the subgraph colosseum$(G, k)[\mathcal{R}(Q)]$, that is, to compute the edges of the subgraph induced by $\mathcal{R}(Q)$. Phase III essentially iterates over all vertices and adds edges according to the definition of the colosseum.

For the runtime, observe that the queue will contain exactly the set $\mathcal{R}(Q)$ and, for every element extracted, we search through the current $K' \subseteq \mathcal{R}(Q)$, which leads to the quadratic timebound of $|\mathcal{R}(Q)|^2$. Furthermore, we have to compute the neighborhood of every extracted element, and we have to test whether two such configurations intersect—both can easily be achieved in time $O(|V|^2)$. Finally, in phase III, we have to compute connected components of the elements in $\mathcal{R}(Q)$, but since this is possible in linear time per element, it is clearly possible in time $O\big(|\mathcal{R}(Q)| \cdot |V|^2\big)$ for the whole graph. $\square$

4.5. Distance Queries in Edge-Alternating Graphs

We have just discussed how to compute the pit for a given graph and $k \in \mathbb{N}$. The computation of graph measures such as treewidth now boils down to simple distance queries to this pit. To obtain an intuition of "distance" in edge-alternating graphs, think about such a graph as in our game and consider some vertex v. There is always one active player that may decide to take *one* existential edge (a fly-move in our game) or the player may decide to ask the opponent to make a move and, thus, has to handle *all* universal edges (a reveal-move in our game). From the point of view of the active player, the distance is thus the *minimum* over the *minimum* of the distances of the existential edges and the *maximum* of the universal edges.

```
procedure discover(G, k)
  $V(\text{pit}(G,k)) := \emptyset$
  $E(\text{pit}(G,k)) := \emptyset$
  $A(\text{pit}(G,k)) := \emptyset$
  initialize empty~queue

  // Phase I: compute Q
  for $v \in V(G)$ do
    offer({$v$}, $k-1$)
  end

  // Phase II: compute $\mathcal{R}(Q) = V(\text{pit}(G,k))$
  while queue not empty do
    extract C from~queue

    // reverse fly-moves
    for $v \in N(C)$ do
      offer($C \cup \{v\}$, $k-1$)
    end

    // reverse reveal-moves
    for $C' \in V(\text{pit}(G,k))$ do
      if not intersect(C, $C'$) then
        offer($C \cup C'$, k)
      end
    end
  end

  // Phase III: compute E and A
  discoverEdges($V(\text{pit}(G,k))$, $E(\text{pit}(G,k))$, $A(\text{pit}(G,k))$)

  return $(V(\text{pit}(G,k)), E(\text{pit}(G,k)), A(\text{pit}(G,k)))$
end
```

Figure 5. The discover algorithm computes, given a graph $G = (V, E)$ and an integer $k \in \mathbb{N}$, the auxiliary graph $\text{pit}(G, k)$. Using the positive-instance driven paradigm, only the elements of the pit are explored during this process. The executed subprocedures can be found in Figure 6.

```
procedure offer(C, t)
  if $C \in V(\text{pit}(G,k))$ then
    return
  end
  if $|N_G(C)| \leq t$ then
    add C to $V(\text{pit}(G,k))$
    insert C into queue
  end
end
```

```
procedure intersect(C, $C'$)
  if $C \cap C' \neq \emptyset$ then
    return true
  end
  if $N_G(C) \cap C' \neq \emptyset$ then
    return true
  end
  if $C \cap N_G(C') \neq \emptyset$ then
    return true
  end
  return false
end
```

```
procedure discoverEdges(V, E, A)
  for $C \in V$ do
    // add fly-move edges
    for $v \in C$ do
      if $C \setminus \{v\} \in V$ then
        add $(C, C \setminus \{v\})$ to E
      end
    end

    // add reveal-move edges
    let $C_1, \ldots, C_\ell$ be the
    components of $G[C]$
    if $C_1, \ldots, C_\ell \in K$ then
      for $i = 1$ to $\ell$ do
        add $(C, C_i)$ to A
      end
    end
  end
end
```

Figure 6. Subprocedures used by the discover algorithm. The offer procedure adds a block to the queue if t is not too large. The intersect procedure simply checks if two blocks are compatible (i. e., that they can be glued together), and the discoverEdges procedure identifies the edges of the pit.

Definition 2 (Edge-Alternating Distance)**.** *Let $H = (V, E, A)$ be an edge-alternating graph with $v \in V$ and $Q \subseteq V$, let further $c_0 \in \mathbb{N}$ be a constant, and $\omega_E \colon E \to \mathbb{N}$ and $\omega_A \colon A \to \mathbb{N}$ be weight functions. The distance $d(v, Q)$ from v to Q is inductively defined as $d(v, Q) = c_0$ for $v \in Q$ and otherwise:*

$$d(v,Q) = \min\Big(\min_{w \in N_\exists(v)} \big(d(w,Q) + \omega_E(v,w)\big),\ \max_{w \in N_\forall(v)} \big(d(w,Q) + \omega_A(v,w)\big) \Big).$$

Lemma 4. *Given an acyclic edge-alternating graph $H = (V, E, A)$, two weight functions $\omega_E \colon E \to \mathbb{N}$ and $\omega_A \colon A \to \mathbb{N}$, a source vertex $s \in V$, a subset of the sinks Q, and a constant $c_0 \in \mathbb{N}$. The value $d(s, Q)$ can be computed in time $O(|V| + |E| + |A|)$ and a corresponding edge-alternating path can be output in the same time.*

Proof of Lemma 4. Since H is acyclic, we can compute a topological order of $(V, E \cup A)$ using the algorithm from [80]. We iterate over the vertices v in reversed order and compute the distance as follows. If v is a sink, we set:

$$d(v,Q) = \begin{cases} c_0 & \text{if } v \in Q; \\ \infty & \text{otherwise.} \end{cases}$$

If v is not a sink, we have already computed $d(w, Q)$ for all $w \in N(v)$ and, hence, can compute $d(v, Q)$ by the formula of the definition. Since this algorithm has to consider every edge once, the whole algorithm runs in time $O(|V| + |E| + |A|)$. A path from s to Q of length $d(s, Q)$ can be found by backtracking the labels starting at s. □

Being able to answer distance queries in the pit yields an easy way of computing graph measures that have game theoretic characterizations [61,62,76]. For instance, if we wish to compute a tree decomposition (there is no bound on the number of branches nor on the depth), we seek a winning strategy that may use an unbounded number of reveals and fly-moves—hence, we just can look for *any* path from the start configuration to Q. If we, instead, look for a path decomposition, we seek a tree decomposition without branches—hence, we need a winning strategy that does not use reveals and we can find one by introducing heavy weights on the universal edges. The following theorem collects graph measures that can be computed with similar arguments. We provide a sketch that illustrates how the weights have to be chosen within the main text; the interested reader finds the full (unfortunately somewhat technical) proof in Appendix A.

Theorem 2. *Given a graph G and an integer k, we can decide in time $O(|\operatorname{pit}(G, k+1)|^2 \cdot |V|^2)$ whether G has $\{$ treewidth, pathwidth, treedepth, q-branched treewidth, dependency treewidth $\}$ at most k.*

Proof. All five problems have game theoretic characterizations in terms of the same search game with the same configuration set [61,62,76]. More precisely, they condense to various distance questions within the colosseum by assigning appropriate weights to the edges.

treewidth: To solve treewidth, it is sufficient to find *any* edge-alternating path from the vertex $C_s = V(G)$ to a vertex in Q. We can find a path by choosing ω_E and ω_A as $(x, y) \mapsto 0$, and by setting $c_0 = 0$.

pathwidth: In the pathwidth game, the searchers are not allowed to perform any reveal [76]. Hence, universal edges cannot be used and we set ω_A to $(x, y) \mapsto \infty$. By setting ω_E to $(x, y) \mapsto 0$ and $c_0 = 0$, we again only need to find some path from $V(G)$ to Q with weight less than ∞.

treedepth: In the game for treedepth, the searchers are not allowed to remove a placed searcher again [61]. Hence, the searchers can only use k existential edges. Choosing ω_E as $(x, y) \mapsto 1$, ω_A as $(x, y) \mapsto 0$, and $c_0 = 1$ is sufficient. We have to search a path of weight at most k.

q-branched treewidth: For q-branched treewidth, we wish to use at most q reveals [62]. By choosing ω_E as $(x, y) \mapsto 0$, ω_A as $(x, y) \mapsto 1$, and $c_0 = 0$, we have to search for a path of weight at most q.
dependency treewidth: This parameter is, in essence, defined via graph searching game that is equal to the game we study with some fly- and reveal-moves forbidden. Forbidding a move can be achieved by setting the weight of the corresponding edge to ∞ and by searching for an edge-alternating path of weight less than ∞. $\square$

4.6. Extending the Algorithm to Directed Treewidth

D-width has a game theoretic characterization based on the following version of the game: The searchers and the fugitive now play on a digraph G, the searchers have unlimited reveals ($q = \infty$), and the fugitive is restricted to move inside strongly connected components (i. e., she can only reach vertices from which there is path back to her current position). Let D-search(G) be the minimal number of searchers required to catch the fugitive in a *monotone* way. In contrast to the games we have considered previously, the number of required searchers may be reduced if we allow non-monotone strategies [81]. However, this is not relevant for the D-width, as we have:

Fact 2 ([81])**.** D-search$(G) = \overrightarrow{\text{tw}}(G) + 1$.

Similar to the undirected games, we can construct a colosseum H, where each vertex $C \in V(H)$ corresponds to a configuration of the game. We put an *existential edge* from C_1 to C_2 when we can transform C_1 to C_2 by placing a searcher in C_1 and by removing all covered searchers (i. e., all searchers that stand on vertices v with $v \notin N_G(C_2)$). A *universal edge* is put from C to all its strongly connected components $C_1, \ldots, C_r$.

Observation 1. *Apart from G now being directed, not much changes for the colosseum. In particular, it is easy to see that it is still an acyclic edge-alternating graph. Thus, we can compute the pit in it using Theorem 1.*

Corollary 1. *Given a digraph $G = (V, E)$ and an integer k, we can test whether or not we have $\overrightarrow{\text{tw}}(G) \leq k$ in time $O(|\operatorname{pit}(G, k)|^2 \cdot |V|^2)$.*

Proof. Compute the pit for the directed graph using Theorem 1; weight the edges with ω_E and ω_A as $(x, y) \mapsto 0$, and set $c_0 = 0$ (as for undirected treewidth in Theorem 2). Find a shortest edge-alternating path from V to the sinks of the pit using Lemma 4. $\square$

5. Color Coding Sieves

The bottleneck in practical implementations of the presented algorithm is the enumeration of compatible blocks that can be glued (Line 22 and 23 in Figure 5). The same problem was observed in the positive-instance driven algorithms for treewidth [52] and treedepth [48]. Both used so called *block sieves* to tackle this problem.

These sieves are data structures based on set tries [82] allowing for an efficient enumeration over blocks that are *possible* candidates. More precisely, if K is the set of already computed blocks and the algorithm from Figure 5 reaches Line 22 with a block C, a block sieve shall efficiently enumerate a set $K' \subseteq K$ such that all $C' \in K$ for which the algorithm reaches Line 24 are contained in K' while $|K \setminus K'|$ is maximized. Trivially, one could store K as list and just test for every element whether it is compatible (this is exactly what Figure 5 does). A block sieve improves this naive idea by storing the sets in a tree that represents intersections. Hence, while enumerating the output, one can prune some parts of the tree (of K) and, thus, has to consider only a subset of $K' \subseteq K$. Tamaki presented this idea for the first time [52] and a more involved version was used by Trimble [48]. However, both implementations came with the drawback that they do *not* provide any guarantee on how well they sieve. We provide a randomized data structure for which we can provide tight bounds for the probability that certain unnecessary blocks are pruned. Furthermore, we show how to derandomize this data structure using the well-known color coding technique and obtain an implementation of the block sieve data structure that correctly prunes *all* of these blocks.

Another disadvantage of set tries is the overhead introduced by managing a tree structure of sets compared to, say, storing the blocks in a simple list or array. During the development of our treedepth solver PID* for PACE 2020, we actually observed that block sieves can negatively impact the performance of positive-instance driven solvers on instances of medium size [49].

In this section, we develop a randomized data structure that gives us the performance advantage of block sieves, reduces the overhead introduced by the set tries, and provides provable guarantees on the sieve quality. *Color coding sieves* apply different filter strategies successively to divide the set of all blocks into smaller and smaller sets. These sets often become small enough to be stored in simple lists. Only if these lists become too large, we divide them into multiple lists via a random choice. If such a separation happens frequently, the data structure converges into a classical set trie.

We assume a total order $<$ on V, that is, $V = \{1, 2, \dots, n\}$. Furthermore, we denote the smallest vertex in a set $C \subseteq V$ by $\min(C)$. Observe that $<$ implies a partial order $\lessdot$ on subgraphs where $C_1 \lessdot C_2$ if, and only if, $\min(C_1) < \min(C_2)$. The *color coding sieve* will be used to store the set $\mathcal{R}(Q) = V(\mathrm{pit}(G, k))$, which, as before, consists of blocks C. The first operation supported by our data structure is the *insertion* operation that inserts a block C to the set represented by the data structure and is denoted by insert(C).

The second operation is used to speed up Line 22 in Listing 5. For a fixed block C, we have to enumerate all blocks C' that (i) do *not* intersect C (procedure intersect(C, C') in Figure 6) and that (ii) satisfy $|N_G(C \cup C')| \leq k$ (procedure offer(C, t) in Figure 6). Recall that two blocks intersect if $C \cap C' \neq \emptyset$, $N_G(C) \cap C' \neq \emptyset$ or $C \cap N_G(C') \neq \emptyset$.

Definition 3. *A block C' is* compatible *with respect to a block C if all of the following holds:*
1. $C \lessdot C'$;
2. $|N_G(C \cup C')| \leq k$;
3. $C \cap C' = \emptyset$, $N_G(C) \cap C' = \emptyset$, *and* $C \cap N_G(C') = \emptyset$.

The set of all blocks that are compatible to C is denoted by $\mathrm{comp}(C)$.

In light of this definition, *query* operation query(C) should return a super set of $\mathrm{comp}(C)$. To support efficient implementation of such queries, a color coding sieve stores a set of blocks in *three levels*. Each level filters the blocks by making use of one of the three items of Definition 3.

The level-1 sieve partitions the blocks into sets $S_i = \{\, C \in \mathcal{R} \mid 2^i \leq \min(C) < 2^{i+1} \,\}$ for $i \in \{0, \dots, \log n - 1\}$ to make use of the partial ordering $\lessdot$. We choose this partition because there are in general many more blocks with $\min(C) = 1$ than with $\min(C) = n$.

Each set S_i is stored as a *level-2 sieve of depth* γ. These sieves are the eponym for the data structure and partition S_i by making use of γ colorings $\mathsf{color}_1, \dots, \mathsf{color}_\gamma$. Each coloring assigns two colors, say blue and orange, to the vertices of G. Let orange_j and blue_j be the set of orange and blue vertices according to color_j, respectively. The set S_i is partitioned into sets $S_i[\boldsymbol{\ell}]$, where $\boldsymbol{\ell} = (\ell_1, \dots, \ell_\gamma) \in \{0, \dots, k\}^\gamma$. A block C belongs to $S_i[\boldsymbol{\ell}]$ if, and only if, the number of orange neighbors w. r. t. color_j is exactly ℓ_j, that is,

$$S_i[\boldsymbol{\ell}] = \{\, C \in S_i : |N_G(C) \cap \mathrm{orange}_j| = \ell_j \text{ for all } j \in \{1 \dots, \gamma\} \,\}.$$

The idea is that whenever we query a block C with $[r_1, r_2, \dots, r_\gamma]$ blue neighbors, we only have to search for compatible blocks in those sets $S_i[\ell_1, \dots, \ell_\gamma]$ with $\ell_j + r_j \leq k$.

Finally, the sets $S_i[\boldsymbol{\ell}]$ are stored as *level-3 sieves*, which are lazily built random set tries. Initially, we store the set $S_i[\boldsymbol{\ell}]$ as a list. If the size of the list grows too large, i. e., $|S_i[\boldsymbol{\ell}]| > \Theta$ for a threshold Θ, we pick a random vertex $x \in V$ and divide $S_i[\boldsymbol{\ell}]$ into $\{\, C \in S_i[\boldsymbol{\ell}] \mid x \in C \,\}$ and $\{\, C \in S_i[\boldsymbol{\ell}] \mid x \notin C \,\}$. These sets are then recursively managed by new level-3 sieves.

Observe that a level-3 sieve degenerates into a set trie with random order after n splits. However, because of the other sieve levels, we hope for far fewer recursive steps in the level-3 sieves.

5.1. Insert a Block to the Color Coding Sieve

Inserting a block C is straight forward: Every sieve adds it to the corresponding sieves of the next level. In detail, the level-1 sieve would add C to the sieve S_i with $2^i \leq \min(C) < 2^{i+1}$. The level-2 sieve, on the other hand, counts, for $j = 1, \dots, \gamma$, the number ℓ_j of orange neighbors of C and adds the block to $S_i[\boldsymbol{\ell}]$. Finally, this level-3 sieve follows a path from the root in the trie to a leaf by checking for vertices on that path whether they are in C. Then C is added to the list stored at the leaf, which eventually is split if it grows too large.

5.2. Query the Color Coding Sieve

To answer query(C), we need to iterate through the sieves. For the level-1 sieve, let q be such that $2^q \leq \min(C) < 2^{q+1}$. As we only need to find blocks C' with $C \lessdot C'$, we can focus our search on the sets S_i with $i \geq q$. In order to sieve out elements on level 2, let C have r_j blue neighbors (w. r. t. color_j). We only need to consider sets $S_i[\ell_1, \dots, \ell_\gamma]$ with $\ell_j + r_j \leq k$ for all $j \in \{1, \dots, \gamma\}$.

Finally, for level-3 sieves, if $S_i[\boldsymbol{\ell}]$ is a list, we simply output that list. If $S_i[\boldsymbol{\ell}]$ is split into $X = \{\, C' \in S_i[\boldsymbol{\ell}] \mid x \in C \,\}$ and $\overline{X} = \{\, C' \in S_i[\boldsymbol{\ell}] \mid x \notin C \,\}$ for a vertex $x \in V$, we have to investigate these sets recursively. If $x \in C$, we only have to recur in $\overline{X}$, as we seek blocks that are disjoint from C. Otherwise, we have to search in both sets.

The data structure is illustrated in Figure 7 together with the steps corresponding to an insert and a query. The following lemma states that the color coding sieve is, in fact, a block sieve: the output of query(C) contains all blocks that are compatible to C.

Lemma 5. *After a sequence of insertions has inserted a set K of blocks to a color coding sieve that corresponds to a graph $G = (V, E)$ and $k \in \mathbb{N}$, we have* $\mathrm{comp}(C) \subseteq$ *query(C) for all possible blocks C.*

Proof. All three levels of the color coding sieve manage a partition of the set of blocks they contain and, thus, of K. We argue that any $C' \in \mathrm{comp}(C)$ is output on query(C). The proof is by contradiction, so assume C' is pruned by one of the sieves.

If C' is pruned on level 1, it must have been in a set S_i with $2^i \leq \min(C') \leq 2^{i+1}$ and $i < q$ for $2^q \leq \min(C) \leq 2^{q+1}$. Thus, $C' \lessdot C$ and, hence, C' is not compatible with respect to C by property 1 of being compatible.

Now, assume C' was pruned by the level-2 sieve. Then C' was stored in a set $S_i[\ell_1, \dots, \ell_\gamma]$ that was not explored. Hence, there must be an index $j \in \{1, \dots, \gamma\}$ and a number $r_j \geq 0$ with $\ell_j + r_j > k$ such that $N_G(C \cup C')$ contains at least ℓ_j orange and r_j blue vertices with respect to color_j. This is a witness for $|N_G(C \cup C')| > k$ and, thus, a contradiction to property 2 of being compatible.

Finally, if C' was pruned by the level-3 sieve, then clearly the sieve contains multiple layers (otherwise the whole list is output). The only way the sieve prunes is that it had split its list with a vertex $x \in V$ and $x \in C$. Since C' was pruned, we have $x \in C'$ and, hence, $C \cap C' \neq \emptyset$—a contradiction to property 3 of being compatible. □

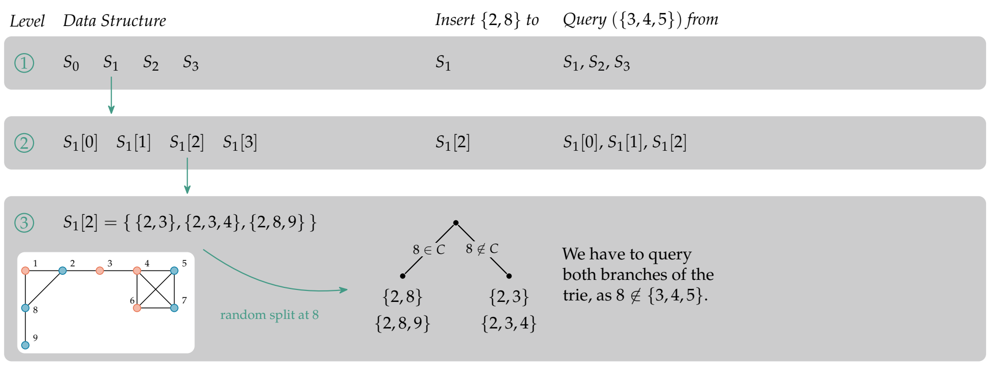

Level
Data Structure
Insert {2,8} to
Query ({3,4,5}) from
① S_0 S_1 S_2 S_3
S_1
S_1, S_2, S_3
② $S_1[0]$ $S_1[1]$ $S_1[2]$ $S_1[3]$
$S_1[2]$
$S_1[0], S_1[1], S_1[2]$
③ $S_1[2] = \{\{2,3\},\{2,3,4\},\{2,8,9\}\}$
random split at 8
$8 \in C$
$8 \notin C$
{2,8}
{2,8,9}
{2,3}
{2,3,4}
We have to query both branches of the trie, as $8 \notin \{3,4,5\}$.

Figure 7. A color coding sieve for the nine vertex graph shown on the bottom left with $k = 4$ and $\gamma = 1$. The three levels are illustrated as gray boxes and show the contained sieves of the next level—green pointers indicate which level-i sieve is shown, i. e., the level-2 sieve is S_1 and the level-3 sieve is $S_1[2]$ (all the other sieves are not shown). In the center, an insert of $\{2, 8\}$ is illustrated: the first sieve inserts it to S_1, the level-2 sieve S_1 inserts it to the level-3 sieve $S_1[2]$ (other branches are not shown). The level-2 sieve uses the random partition shown on the graph, that is, $\{2, 8\}$ has two orange neighbors. The level-3 sieve $S_1[2]$ is an ordinary list before the insert and becomes a trie after the insert, as a random split at vertex 8 occurs. On the right side of the figure, we see all sieves that we have to search through in order to answer the query $(\{3, 4, 5\})$.

5.3. Optimizing Level-2 Sieves

We have not yet specified the colorings used in level 2. In the following, we first consider randomized colorings and then make use of *color coding* for derandomization. Using the guarantees provided by this technique, we obtain a provable running time bound for the level-2 sieve. We are interested in the probability that a block C' with $|N_G(C \cup C')| > k$ is part of the output if we use random colorings.

Lemma 6. *Let $\gamma = 1$ and fix some blocks C and C' with $|N_G(C \cup C')| > k$. If color_1 is chosen randomly, the probability that C' is output by query(C) is at most*

$$1 - \frac{k+1}{2^{k+1}}.$$

Proof of Lemma 6. Let $\hat{k} = |N_G(C \cup C')| \in \{k+1, \ldots, 2k\}$ and let color_1 be a randomly sampled coloring. Note that C' is output by query(C) if, and only if, $\beta(C) + \omega(C') \le k$, where we define $\beta(C) := |\{\, v \in N_G(C) \mid \mathrm{color}(v) = \mathrm{blue} \,\}$ as the number of blue neighbors and $\omega(C)$ equivalently as the number of orange neighbors.

A coloring is called *good* if $\beta(C) + \omega(C') > k$ as it allows to discard C'. A vertex $v \in N_G(C \cup C')$ is *good* if either $\mathrm{color}(v) = \mathrm{blue}$ and $v \in N_G(C)$ or $\mathrm{color}(v) = \mathrm{orange}$ and $v \in N_G(C')$. Hence, a coloring is good if at least $k + 1$ vertices are good. Note that all vertices in $N_G(C) \cap N_G(C')$ are good (as each vertex is colored). In the following, we thus assume that $N_G(C) \cap N_G(C') = \emptyset$, which is the worst case. As each coloring occurs with probability $2^{-\hat{k}}$, the probability to hit a good coloring is then exactly

$$\left(2^{\hat{k}}\right)^{-1} \sum_{i=k+1}^{\hat{k}} \binom{\hat{k}}{i}.$$

Hence, the probability of hitting a non-good coloring (and outputting C') is at most

$$1 - \frac{\sum_{i=k+1}^{\hat{k}} \binom{\hat{k}}{i}}{2^{\hat{k}}} = \frac{\sum_{i=0}^{k} \binom{\hat{k}}{i}}{2^{\hat{k}}}.$$

This term is maximized for $\hat{k} = k+1$, as the denominator grows faster than the nominator. Hence, the probability to output C' is at most

$$\frac{\sum_{i=0}^{k}\binom{\hat{k}}{i}}{2^{\hat{k}}} \leq \frac{\sum_{i=0}^{k}\binom{k+1}{i}}{2^{k+1}} = 1 - \frac{k+1}{2^{k+1}}. \qquad \square$$

If we choose all γ colorings randomly and independent, we obtain:

Corollary 2. *Fix some blocks C and C' with $|N_G(C \cup C')| > k$. If $\mathsf{color}_1, \mathsf{color}_2, \dots, \mathsf{color}_\gamma$ are chosen randomly and independent, the probability that C' is output by query(C) is at most*

$$(1 - \frac{k+1}{2^{k+1}})^\gamma.$$

We can now increase γ until the term $(1 - \frac{k+1}{2^{k+1}})^\gamma$ is below a tolerable threshold. If we add sufficiently many colorings, we may obtain the same result with deterministic colors that are produced by hash functions (that is, the level-2 sieve can be derandomized).

Definition 4. *For two natural numbers n and k, an $(n, k, 2)$-universal coloring family is a set Λ of functions $\lambda\colon \{1, \dots, n\} \to \{\text{orange}, \text{blue}\}$ such that for every subset $S \subseteq \{1, \dots, n\}$ of size $|S| = k$ and for every mapping $\mu\colon S \to \{\text{orange}, \text{blue}\}$, there is at least one function $\lambda \in \Lambda$ with $\mu(s) = \lambda(s)$ for all $s \in S$.*

It is well-known that such families can be constructed via hash functions.

Theorem 3 (Theorem 13.41 in [83]). *For all natural numbers n and k, an $(n, k, 2)$-universal coloring family Λ of size $|\Lambda| \leq 2^{O(k)} \cdot \log^2(n)$ can be found in time $2^{O(k)} \cdot n \cdot \log^2(n)$.*

Let Λ be $(n, k, 2)$-universal coloring family and assume that we operate the level-2 sieve with $\gamma = |\Lambda|$ colorings such that each of the colorings is produced by one of the hash functions in Λ. Now assume we perform query(C) and fixate *any* block C' with $|N_G(C \cup C')| > k$. By the properties of $(n, k, 2)$-universal coloring families, there is at least one coloring in Λ that colors the vertices in $C \cup C'$ in such a way that $N(C)$ contains at least ℓ orange vertices, $N(C')$ contains at least r blue vertices, and $\ell + r > k$. Hence, the level-2 sieve prunes C' and, therefore, prunes *all* blocks that are incompatible to C by property 2 of being compatible. We collect this finding in form of the following theorem:

Theorem 4. *There is a computable function $f\colon \mathbb{N} \to \mathbb{N}$ such that for any graph G on n vertices and any $k \in \mathbb{N}$, the level-2 sieve can be implemented* deterministically *with $\gamma = f(k) \cdot \mathrm{poly}(n)$ colors. When a block C is queried, this sieve prunes* all *stored blocks C' with $|N_G(C \cup C')| > k$.*

5.4. Pruning Queries in Level-3 Sieves

If the level-3 sieve degenerates into a set trie, we can use the structure of the trie to prune queries. Consider a node p of the trie and observe that the path from p to the root defines two sets R_p and F_p of *required* and *forbidden* vertices, respectively. Each block C' stored in the subtrie rooted at p fulfills $R_p \subseteq C'$ and $F_p \cap C' = \emptyset$. Hence, if $\min(R_p) \leq \min(C)$, we do not have to explore p or its children. Define a *pin* to be a vertex $v \in V$ with $v \in F_p$ and $N_G(v) \cap R_p \neq \emptyset$, and let P_p be the set of pins at p. Note that all blocks stored in the subtrie rooted at p are adjacent to v and, thus, we can prune the search if $v \in C$ as all blocks then intersect with C. Pins become neighbors of glued blocks and, hence, we can also prune if $|N_G(C) \cup P_p| > k$.

6. Experimental Evaluation of Color Coding Sieves

In this section, we experimentally evaluate the performance gain of a positive-instance driven algorithm when equipping it with a color coding sieve. We exemplarily do so by adding the color coding sieve (with $\gamma = 1$ and randomly generated coloring) to our treedepth solver PID* that is built on the algorithms developed within this paper [49]. A comparison of PID* with other state-of-the-art treedepth solvers was recently performed in the light of PACE 2020 and can be found in [45].

We perform the experiments on a computer equipped with 64 GB of RAM and an AMD Ryzen Threadripper 3970X with 32 cores of 3.7 Ghz each and with 144 MB of combined cache. The system runs on Ubuntu 18.04.5 LTS, 64bit. We let the solver run on two benchmark sets: the well-known DIMACS graph coloring instances (see [34,84] for an overview of these instances in light of treewidth), and the instances that were used in the exact track of the treedepth challenge PACE 2020 [45]. Both solvers (without and with color coding sieves) were run for 10 min on each instance. The results of the experiment are illustrated as scatter plots in Figure 8 (the left one corresponds with the DIMACS instances, the right one to the PACE 2020 instances).

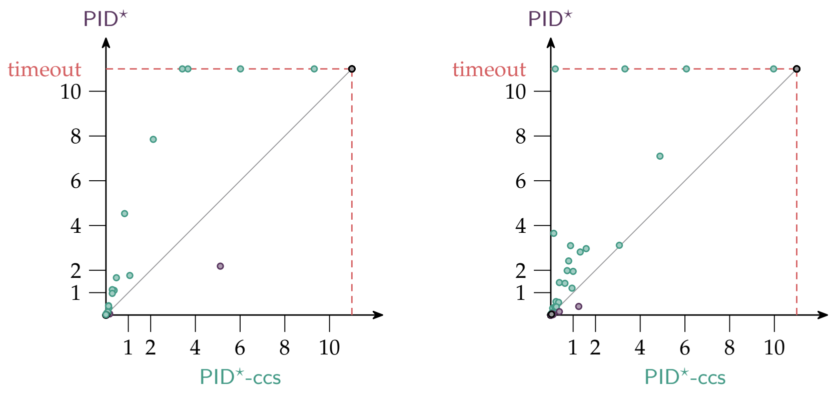


Figure 8. Scatter plots that show the performance of PID* without color coding sieves compared to the performance of the solver with them enabled ($\gamma = 1$ and a randomly generated coloring is used). The left plot contains the DIMACS graph coloring instances, the right plot the PACE 2020 test set. Each point corresponds to an instance; the x-axis is the time needed by the solver using color coding sieves and the y-axis without using this feature. The color of the dot indicates which version was better; it is gray if they are equal. If a solver needed more than 10 min for an instance, the coordinate is set to the red dotted line.

As one can observe, the color coding sieve generally improves the performance of the solver. Just on a few outlier instances, the solver cannot utilize this data structure, which is probably due to many non-compatible blocks that have a small neighborhood and, thus, are not pruned by the level-2 sieve. We can also observe that the obtained speedup is larger on the PACE 2020 instances than on the DIMACS instances. The PACE instances were generally harder to solve for PID* and contained more blocks. Hence, the solver can utilize the color coding sieve better on this benchmark set.

6.1. Color Coding Sieves on Hyperbolic Random Graphs

In this section, we investigate the potential performance boost obtained by using coloring coding sieves on *hyperbolic random graphs.* We use the same experimental setup as in the last section and, again, equip our treedepth solver PID* [49] with the color coding sieve data structure. In contrast to the last section, we will now:

1. Vary the used layers and the number of colorings γ;
2. Run the solver on *random graphs* rather than predefined benchmark sets.

For this experiment, we use *hyperbolic random graphs,* which are known to replicate many structural properties of real-world networks [85,86]. Loosely speaking, these graphs are generated by sampling points randomly in a disk (these are the vertices) and by connecting points by an edge if their *hyperbolic distance* is smaller than some predefined threshold.

We generated a set of 210 hyperbolic random graphs using the *Hyperbolic Graph Generator* [87]. The generator expects six parameters that we set as shown in Table 1. The solver PID* was used with the four configurations explained in Table 2, whereby the level-2 sieves always use uniformly sampled random colorings.

Table 1. Parameters of the used hyperbolic graph generator [87]. We set the number of vertices to a range in which the instances are tractable but challenging for the solver PID*. The expected average degree is k_1 for $n \in \{100, 110, 120\}$, k_2 for $n \in \{130, 140\}$, k_3 for $n \in \{150, 160\}$, and k_4 for $n \in \{170, 180, 190, 200\}$. For each combination of n and the corresponding k_i, we generated six instances with random seed $s \in \{1, \ldots, 6\}$ yielding 210 instances. The remaining parameters of the generator are set to their default value.

number of nodes	$n \in \{100, 110, 120, \ldots, 200\}$
expected average degree	$k_1 \in \{10, 20, 30\}$, $k_2 \in k_1 \cup \{40\}$, $k_3 = k_2 \cup \{50\}$, $k_4 \in \{3, 5\}$
expected power-law exponent	2 (default)
square root of curvature	1 (default)
temperature	0 (default)
seed	$i \in \{1, 2, \ldots, 6\}$

All configuration of the solver were run on all instances for at most 30 min. The results of the experiment are illustrated in a cumulative distribution function plot in Figure 9. Various interesting findings are contained in the plot: First, we can observe that the core solver without any sieve layer (-wo) cannot fully utilize the provided 30 min time window. All instances solved by this configuration are solved within 20 min and, indeed, afterwards the amount of blocks becomes too large to be enumerated naively.

Second, we see the small (and expected) disadvantage of solvers with color coding sieves (-trie and -color-1) compared to the core algorithm (-wo) on "easy" instances (that can be solved in a minute or less). Of course, using involved data structures comes with an overhead and, if there are simply not enough blocks, we may not overcome this disadvantage with the improvements provided by the data structure. However, we can also observe that the performance of the solver with lazily built tries (-trie) and with the full color coding sieve (-color-1) quickly outperforms the core solver on the remaining instances. For the same reason as mentioned above, the lazily built set trie alone is better than the whole sieve on "medium hard" instance (solvable in about five minutes). Then, on even harder instances with more blocks, the additional layers can be utilized and the full color coding sieve provides the overall best performance.

Third, we can observe that, on this test set, choosing $\gamma > 1$ (-color-2) has no positive effect compared to using $\gamma = 1$ (-color-1). Adding more colorings increases the overhead of the data structure and, thus, a negative effect is expected if a layer cannot be fully utilized. Here, we conclude that on instances that are currently tractable for the solver, there are not enough blocks such that a second random partition sufficiently improves the enumeration of compatible blocks. However, we can also observe that the trend is positive, i.e., the overhead of a second coloring is reduced more and more on harder instances. We thus conjecture that, if the solver can be tuned to solve larger instances, the positive effect of $\gamma > 1$ colorings, theoretically provided by Corollary 2, will also have a positive impact in practical implementations.

Table 2. Four configurations of PID* used in the experiment.

Configuration	Meaning
-wo	Color coding sieves are *not* used.
-trie	Only the lazy set tries are used.
-color-1	All three sieves are used and $\gamma = 1$.
-color-2	All three sieves are used and $\gamma = 2$.

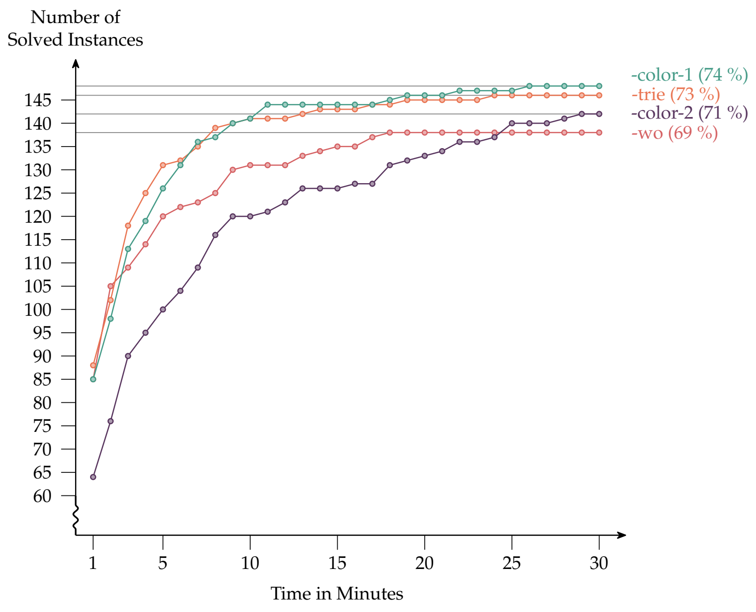


Figure 9. A cumulative distribution function plot that shows the performance of PID* without color coding sieves (-wo), with just the lazily built set trie (-trie), with the full color coding sieve and $\gamma = 1$ (-color-1), and with the full color coding sieve and $\gamma = 2$ (-color-2). The experiments are performed on the set of 210 hyperbolic random graphs described in Table 1.

6.2. Sieve-Quality of the Individual Layers

So far, we measured the performance boost obtained by adding a color coding sieve to PID* with respect to the overall time the solver needs to solve an instance. In this section, we rather focus on the quality of the individual layers of the sieve. That is, we do *not* measure the time used to solve an instance but measure the *amount* of blocks that a sieve filtered.

In more detail, we measure the *total number of blocks* generated by PID* (that is, the total number of blocks inserted into the sieve) and the number of *compatible blocks* that were filtered from the output of the sieve (over the complete run of the solver). We compare this number with the *number of loaded blocks*, which is the number of blocks returned by the sieve. Of course, all compatible blocks are loaded, but (if the sieve works poorly) many other blocks may be loaded, too. We measure the *performance of the sieve* as the fraction of falsely loaded blocks over all non-compatible blocks. See Table 3 for the exact terminology.

Table 3. Variables used to describe the performance of a sieve. The values t, c and ℓ are considered over a complete run of the solver for a given instance, e. g., summed over various values for the target width k.

Variable	Meaning	Comment
t	Total number of blocks.	
c	Number of compatible blocks.	$c \leq t$
ℓ	Number of blocks loaded by the sieve.	$c \leq \ell \leq t$
α	Performance as $\alpha = 1 - \frac{\ell - c}{t - c}$	$\alpha = 0$ for a trivial sieve $\alpha = 1$ for a perfect sieve

Observe that, in the light of Corollary 2 and Theorem 4, the performance of a color coding sieve converges to 1 if γ, the number of colorings, converges to infinity. See Figure 10 for an illustrative example.

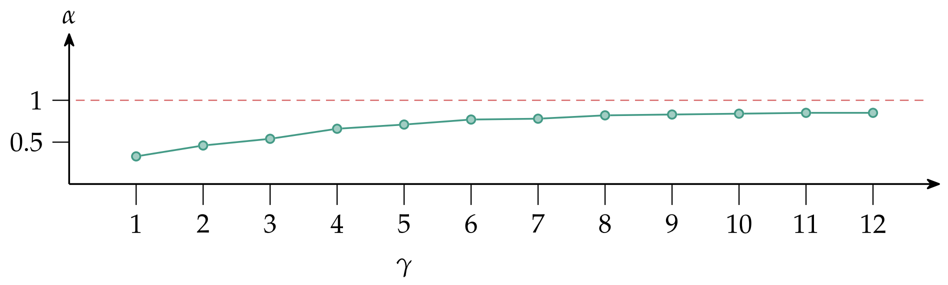


Figure 10. Performance α of a level-2 sieve with γ colorings on instance exact_036 from the PACE 2020 benchmark set [45].

For the following experiments, we only use *a single layer* of the color coding sieve and make all measurements with respect to a complete run of the solver for a given instance (i. e., summed over various values of the target width k). Since we aim to obtain a good estimation of the actual performance of the various sieves via experimental means, we consider rather difficult instances in this section. First, in Table 4, we consider $n \times n$-grid graphs, which are well-known to be difficult for various graph decomposition algorithms. Secondly, we hand-crafted a set of instances that are particularly difficult for the positive-instance driven approach. Let a (n, d, c)-spider be the graph obtained by the following procedure: start with a star with n leaves, then replace every edge by a path with d vertices, and finally replace every vertex by a clique on c vertices that is fully-connected to the vertices of adjacent cliques. (Intuitively, a star is a worst-case instance for the positive-instance driven approach, as initially only the leaves are positive subproblems and, by trying to glue them together, the solver has to explore all 2^n subsets of the leaves. By stretching and thickening the edges, we enforce that this behaviour stays the same and circumvent heuristics internally used by the solver.) The corresponding experiment is summarized in Table 5. Finally, we performed the experiment on a set of instances from the PACE 2020 benchmark set [45] (see Section 6). The results are presented in Table 6.

Table 4. Overview of the performance of PID* if a single layer of the color coding sieve is used. The columns n, m, and td describe the number of vertices and edges, as well as the treedepth of the corresponding graph. The columns labeled "Level-i" indicate that a level-i sieve is used (and none of the other layers). In this case, the level-2 sieve is used with $\gamma = 1$ and the level-3 sieve is used in its plain version (without any improvements). Columns labeled with $\gamma = c$ indicate that a level-2 sieve with c colorings was used (the "random" colorings were generated with a seeded pseudo-random generator, that is, the first coloring of the $\gamma = 3$ sieve is the same as the coloring of the $\gamma = 1$ sieve and the first three colorings of the $\gamma = 6$ sieve are the ones of the $\gamma = 3$ sieve), and the column "imp. Level-3" corresponds to a level-3 sieve with the improvements discussed after Theorem 4. In each row, the sieve with the best performance on this instance is highlighted.

				α of . . .					
Graph	**n**	**m**	**td**	**Level-1**	**Level-2**	**$\gamma = 3$**	**$\gamma = 6$**	**Level-3**	**imp. Level-3**
grid_4×4	16	24	7	0.00	0.22	0.41	**0.63**	0.25	0.34
grid_5×5	25	40	9	0.00	0.20	0.49	**0.66**	0.46	0.65
grid_6×6	36	60	11	0.02	0.26	0.52	0.70	0.62	**0.81**
grid_7×7	49	84	13	0.07	0.29	0.59	0.78	0.69	**0.92**
grid_8×8	649	112	15	0.12	0.31	0.62	0.81	0.72	**0.96**

Table 5. Same as Table 4, but on spider-graphs rather than grids.

				α of . . .					
Graph	**n**	**m**	**td**	**Level-1**	**Level-2**	**$\gamma = 3$**	**$\gamma = 6$**	**Level-3**	**imp. Level-3**
spider_$(10, 5, 5)$	255	1760	20	0.00	0.23	0.38	**0.51**	0.04	0.13
spider_$(10, 10, 5)$	505	3510	25	0.03	0.18	0.35	**0.49**	0.05	0.22
spider_$(10, 10, 6)$	606	5115	30	0.03	0.16	0.34	**0.44**	0.05	0.19
spider_$(20, 10, 3)$	606	2403	15	0.11	0.16	0.38	**0.57**	0.03	0.33
spider_$(30, 10, 5)$	1505	10510	25	0.15	0.14	0.31	**0.42**	0.02	0.34

Table 6. Same as Table 4, but on instances from the PACE 2020 benchmark set [45].

Graph	n	m	td	α of … Level-1	Level-2	$\gamma = 3$	$\gamma = 6$	Level-3	imp. Level-3
exact_057	50	75	13	0.19	0.31	0.65	0.87	0.77	**0.97**
exact_077	66	120	12	0.10	0.27	0.53	0.75	0.51	**0.83**
exact_078	67	152	12	0.04	0.12	0.26	**0.33**	0.13	0.20
exact_079	68	83	9	0.06	0.27	0.59	**0.71**	0.36	0.69
exact_083	70	274	22	0.16	0.33	0.63	0.79	0.67	**0.92**

From the experimental data shown in Tables 4–6, we conclude three observations: First, the level-1 sieve alone does not much. This was expected, as this sieve has semantically the least function. It is intended as fast preprocessing for the other sieves. Secondly, the level-2 and level-3 sieve (a single random partition and a normal set trie) are relatively equivalent. While the former is better on spider graphs, the latter is better on instances from the PACE benchmark set. Our final observation is that both sieves greatly improve their performance if the extensions are added (more colorings for the level-2 sieve or the improvements discussed after Theorem 4 for the trie, respectively). Comparing the level-2 sieve with $\gamma = 6$ to the improved trie leads again to a mixed picture in which both of the sieves are better than the other on some of the instances. We conclude that taking a random partitioning is a valuable alternative to a set trie.

The level-2 sieve has two advantages over the set trie: (i) it is comparatively easy to implement and comes with low constants and (ii) we can easily improve its performance by adding more colorings. On the downside, adding more colorings slows this sieve down. Hence, if we need a large γ, a set trie becomes the better choice.

7. Conclusions and Outlook

Treewidth is one of the most useful graph parameters that is successfully used in many different areas. The positive-instance driven dynamic programming paradigm has led to the first practically relevant algorithm for this parameter, as well as for its close relative treedepth. We formalized such algorithms in the general setting of graph searching, which has allowed us to provide a clean and simple formulation and to extend the algorithm to many natural graph parameters.

With a few modifications of the colosseum, our approach can also be used for the notion of *special treewidth* [88]. We assume that a similar modification may also be possible for other parameters such as *spaghetti treewidth* [89].

We also extended the block sieve data structure to a randomized multi-level sieve that is constructed lazily and that utilizes the well-known color coding technique. By modifying the number of colorings used, we can change the probability that a non-compatible block is pruned by the sieve. Hence, introducing more and more colors makes it unlikely that non-compatible blocks are output by the sieve. On the theoretical side, we may derandomize the data structure and obtain a guarantee that certain non-compatible blocks are filtered.

Experiments have revealed that the color coding sieve in general increases the performance of our treedepth solver PID*. For the instances that are currently tractable by the solver, a single random coloring provides the best overall performance, but we expect that on harder instances the theoretical guarantees of using more colorings will result in a positive practical impact. An interesting next step would be to implement the color coding sieve in state-of-the-art treewidth solvers such as [20,53,63], which currently perform slightly better than the state-of-the-art treedepth solvers [44,45].

Author Contributions: The authors have contributed equally. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The DIMACS graph coloring instances and the PACE 2020 instances are publicly available [45,84]. The hyperbolic random graphs can be generated with the publicly available hyperbolic graph generator [87] and the parameters stated within the main text. The tools Jdrasil and PID* are also publicly available [20,49].

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

Appendix A. Proof of Theorem 2

We dedicate this section to the missing proof of Theorem 2. The theorem states that we can compute various graph parameters in polynomial time width respect to the size of the pit, the formal statement was:

Recall the sketched proof idea from Section 4.5: We wanted to use the fact that all five problems have game theoretic characterizations that can be encoded in the colosseum [61,62,76]; then we argued that, by setting the weights in the pit correctly, we would obtain all parameters by simply computing shortest edge-alternating paths. We further claimed that the required weights are the following:

treewidth: Choose ω_E and ω_A as $(x,y) \mapsto 0$, and set $c_0 = 0$.
pathwidth: Set ω_A to $(x,y) \mapsto \infty$, ω_E to $(x,y) \mapsto 0$, and $c_0 = 0$.
treedepth: Choosing ω_E as $(x,y) \mapsto 1$, ω_A as $(x,y) \mapsto 0$, and $c_0 = 1$.
***q*-branched treewidth:** Set ω_E to $(x,y) \mapsto 0$, ω_A to $(x,y) \mapsto 1$, and $c_0 = 0$.
dependency treewidth As for treewidth, but we have to set the weight of some forbidden edges to infinity.

Let us first observe that, by the definition of the colosseum, k searchers in the search game have a winning strategy if, and only if, the start configuration $V(G)$ is contained in $\mathcal{R}(Q)$. With other words, if there is an edge-alternating path from $V(G)$ to some winning configuration in Q. Note that such a path directly corresponds to the strategy by the searchers in the sense that the used edges directly correspond to possible actions of the searchers.

Since for any graph $G = (V,E)$ and any number $k \in \mathbb{N}$ the edge-alternating graph colosseum(G,k) is universal consistent by Lemma 3, all vertices of an edge-alternating path corresponding to a winning strategy are contained in $\mathcal{R}(Q)$ as well. In fact, *every* edge-alternating path from $V(G)$ to Q (and, thus, *any* winning strategy) is completely contained in $\mathcal{R}(Q)$. Therefore, it will always be sufficient to search such paths within pit(G,k). By Lemma 4, we can find such a path in time $O(|\operatorname{pit}(G,k)|^2)$. In fact, we can even define two weight functions $w_E\colon E \to \mathbb{N}$ and $w_A\colon A \to \mathbb{N}$ and search a *shortest path* from $V(G)$ to Q.

Appendix A.1. Computing Branched Tree Decompositions

To compute the invariants of G as stated in the theorem, we make the following claim:

Claim A1. *Let $G = (V,E)$ be a graph and $k \in \mathbb{N}$. Define w_E as $(x,y) \mapsto 0$ and w_A as $(x,y) \mapsto 1$, and set $c_0 = 0$. Then we have $d(V(G),Q) \le q$ in* pit(G,k) *if, and only if,* $\operatorname{tw_q}(G) \le k-1$.

Proof. We follow the proof of Theorem 1 in [62] and use the following fact that follows from the observation that in a tree decomposition (T,ι), for each three different nodes $i_1, i_2, i_3 \in T$, we have $\iota(i_1) \cap \iota(i_3) \subseteq \iota(i_2)$ if i_2 is on the unique path from i_1 to i_3 in T.

Fact A1. *Let (T,ι) be a tree decomposition of $G = (V,E)$ rooted arbitrarily at some node $r \in T$. Let $i \in T$ be a node and $j \in T$ be a child of i in T. Then, the set $\iota(i) \cap \iota(j)$ is a separator between $C = \big[\bigcup_{d \in \operatorname{desc}(j)} \iota(d)\big] \setminus \big(\iota(i) \cap \iota(j)\big)$ and $(V \setminus C) \setminus \big(\iota(i) \cap \iota(j)\big)$, where* desc$(x)$ *denotes the set of descendants of x including x. Hence, every path from some node $u \in C$ to some node $v \in V \setminus C$ contains a vertex of $\iota(i) \cap \iota(j)$.*

Appendix A.1.1. From Tree Decompositions to Edge-Alternating Paths

Let (T, ι) be a q-branched tree decomposition of $G = (V, E)$ of width k. Without loss of generality, we can assume that G is connected. We will show how to construct an edge-alternating path from the start configuration V of cost at most q in colosseum$(G, k+1)$. As described above, this is also an edge-alternating path with the same costs in pit$(G, k+1)$. The first existential edge from V leads to the configuration $V \setminus \iota(r)$, where r is the root of T. Observe that we clearly have $N(V \setminus \iota(r)) \subseteq \iota(r)$. Suppose a configuration C was reached with $N(C) \subseteq \iota(i) \in V(\text{colosseum}(G, k+1))$ for some node $i \in T$ and we have:

$$C \subseteq [\bigcup_{j \in \text{desc}(i)} \iota(j)] \setminus \iota(i).$$

Clearly, for $i = r$, this assumption holds trivially. If i is a leaf in T, there are no more descendants and thus $C = \emptyset$. Hence, we have reached a winning configuration in colosseum$(G, k+1)$. Now, suppose that i is a non-leaf node. We distinguish two cases:

- If i has exactly one child j, we can find a path P_1 of existential edges leading from C to a configuration C_1 with $N(C_1) \subseteq \iota(i) \cap \iota(j)$. Moreover, we can also find a path P_2 of existential edges from C_1 to a configuration C_2 with $N(C_2) \subseteq \iota(j)$. The path P_1 will be constructed by iteratively removing all vertices $v \in C$ with $N(v) \cap [\iota(i) \setminus \iota(j)] \neq \emptyset$. For the remaining vertices C_1, we have $N(C_1) \subseteq \iota(i) \cap \iota(j)$. If all configurations that we aim to visit on P_1 exist, the corresponding edges also exist by definition. Assume that we are in some configuration C' with $N(C') \cap [\iota(i) \setminus \iota(j)] \neq \emptyset$ and want to remove a vertex $v \in C'$ with $N(v) \cap [\iota(i) \setminus \iota(j)] \neq \emptyset$, but $C' \setminus \{v\} \notin V(\text{colosseum}(G, k+1))$. By definition of colosseum$(G, k+1)$, this means that $|N(C' \setminus \{v\})| \geq k+2$. As we wanted to remove v, we have $N(v) \cap \iota(i) \neq \emptyset$. As $N(C' \setminus \{v\}) \subseteq N(C') \cup \{v\}$ and $|N(C' \setminus \{v\})| \geq k+2$, we know that there is some $u \in C'$ with $v \in N(u)$. Fact A1 implies that $v \in \iota(i) \cap \iota(j)$, a contradiction and, hence, all configurations in P_1 exist. Similarly, we construct P_2 by iteratively removing all vertices in $\iota(j)$ from C_1. It is easy to see that the neighborhood of the visited configurations will always be a subset of $\iota(j)$ and, hence, all configurations on this path exist.
  We have arrived at a configuration C_2 with $N(C_2) \subseteq \iota(j)$ and due to Fact A1:
  $$C_2 \subseteq [\bigcup_{j' \in \text{desc}(j)} \iota(j')] \setminus \iota(j).$$
- If node i has a set of children J with $|J| \geq 2$, we will use universal edges. Let $\mathcal{C}$ be the connected components of $G[\bigcup_{j \in \text{desc}(i)} \iota(j) \setminus \iota(i)]$. We claim that for each component $\Gamma \in \mathcal{C}$ there is a unique index $j(\Gamma) \in J$ such that $\Gamma \cap \iota(j(\Gamma)) \neq \emptyset$. If no such index exists, we have $\iota(j) = \iota(i)$. We can iteratively remove such bags $\iota(j)$ until this cannot happen anymore. If two indices $j_1, j_2 \in J$ exist with $\iota(j_1) \cap \Gamma \neq \emptyset$ and $\iota(j_2) \cap \Gamma \neq \emptyset$, the connectivity property implies that $\iota(i) \cap \Gamma \neq \emptyset$, a contradiction to our assumption. Hence, for each component Γ, we follow the universal edge to Γ and then proceed as above: first, we find a path P_1 of existential edges from Γ to a configuration Γ_1 with $N(\Gamma_1) \subseteq \iota(i) \cap \iota(j(\Gamma))$ and then a path P_2 of existential edges from Γ_1 to a configuration Γ_2 with $N(\Gamma_2) \subseteq \iota(j(\Gamma))$. The same arguments as above imply that all configurations on these paths exist and that we arrive at a configuration Γ_2 with $N(\Gamma_2) \subseteq \iota(j(\Gamma))$ and:
  $$\Gamma_2 \subseteq [\bigcup_{j' \in \text{desc}(j(\Gamma))} \iota(j')] \setminus \iota(j(\Gamma)).$$

This shows that we will eventually reach the leaves of the tree decomposition and, thus, some wining configuration. This is an edge-alternating path in colosseum$(G, k+1)$ and pit$(G, k+1)$. Furthermore, as each path from the root of T to some leaf of T contains at most q nodes with more than one children, this path is q-branched, as we use at most q universal edges from the initial configuration V to any used winning configuration for every induced directed path. Hence, we have found an edge-alternating path in $\mathrm{pit}(G, k+1)$ of cost at most q.

Appendix A.1.2. From Edge-Alternating Paths to Tree Decompositions

Let $P \subseteq V(\mathrm{pit}(G, k+1))$ be an edge-alternating q-branched path from the initial configuration V to a final configuration $\{v^*\}$ in $\mathrm{pit}(G, k+1)$ with $|N(\{v^*\})| \le k$. We argue inductively on q.

- If $q = 0$, the path P does not use any universal edges. Let $\pi = \pi_1, \dots, \pi_s$ be any classical directed path from the initial configuration V to some winning configuration $\{v^*\}$ in $\mathrm{pit}(G, k+1)$ that only uses vertices from P. As the initial configuration is $\pi_1 = V$, the winning configuration is $\pi_s = \{v^*\}$, and there are only existential edges (C, C') with $|C'| = |C| - 1$ in $\mathrm{pit}(G, k+1)$, we know that $|\pi_i| = |V| - i + 1$, and thus $s = |V|$. We say that vertex $v \in V$ is *removed at time i*, if $v \in \bigcap_{j=1}^{i} \pi_j$ and $v \notin \bigcup_{j=i+1}^{|V|} \pi_j$. We also say that v^* was removed at time $|V|$. For $i = 1, \dots, |V|$, let v_i be the vertex removed at time i.
We will now construct a 0-branched tree decomposition (T, ι), i. e., a path decomposition. As T is a path, let $t_1, \dots, t_{|V|}$ be the vertices on the path in their respective ordering with root t_1. We set $\iota(t_i) = N(\pi_i) \cup \{v_i\}$. For $i = 1, \dots, |V| - 1$, there is an existential edge leading from π_i to π_{i+1} and thus $|N(\pi_i)| \le k$. As $\pi_{|V|} = \{v_{|V|}\}$ is a winning configuration, we also have $|N(\pi_{|V|})| \le k$. Hence, the resulting decomposition T has width at most k. As T is a path, it is also 0-branched.
We now need to verify that (T, ι) is indeed a valid tree decomposition. As every vertex v is removed at some time i, we have $v = v_i$ and thus $v \in \iota(t_i)$. Hence, every vertex is in some bag. Let $\{v_i, v_{i'}\}$ be any edge with $i < i'$. As $v_{i'} \in \pi_{i'}$ and $v_i \notin \pi_{i'}$, we have $v_i \in N(\pi_{i'})$ and thus $\{v_i, v_{i'}\} \subseteq N(\pi_{i'}) \cup \{v_{i'}\} = \iota(t_{i'})$. Hence, every edge is in some bag. Finally, let $v_i \in V$. Clearly, as $v_i \in \pi_1, v_i \in \pi_2, \dots, v_i \in \pi_{i-1}$, the first bag where v_i might appear is $\iota(t_i)$. Let $v_{i'} \in N(v_i)$ be the neighbor of v_i that is removed at the latest time. If $i' < i$, we have $N(v_i) \cap \bigcup_{j=i+1}^{|V|} \pi_j = \emptyset$ and v_i thus only appears in $\iota(t_i)$. If $i < i'$, then $v_i \in \bigcap_{j=i+1}^{i'} N(\pi_j)$ and hence $v_i \in \bigcap_{j=i+1}^{i'} \iota(t_j)$.
- Now, assume that $q \ge 1$ and that we can construct for every $q' < q$ a q'-branched tree decomposition of width at most k from any q'-branched edge-alternating path P in $\mathrm{pit}(G, k+1)$. Consider the directed acyclic subgraph H in $\mathrm{pit}(G, k+1)$ induced by P. A configuration $C \in V(H)$ is called a *universal configuration*, if $N_A(C) \subseteq V(H)$ and a *top-level universal configuration* with respect to some directed path π if C is the first universal configuration on π. Note that we can reduce P in such a way that all directed paths π from the initial configuration V to some winning configuration $\{v^*\}$ in H have the same top-level universal configuration, call it C^*. Let $V = \pi_1, \dots, \pi_i = C^*$ be the shared existential path from V to C^* in H and let $N_A(C^*) = \{C_1, \dots, C_\ell\}$ be the universal children of C^*. Note that $\{C_1, \dots, C_\ell\} \subseteq P$ due to the definition of an edge-alternating path. For each child C_j, the edge-alternating path P contains a directed path $\pi^{(j)}$ from C_j to some final configuration in $\mathrm{pit}(G, k+1)$. Furthermore, each $\pi^{(j)}$ contains at most $q' \le q - 1$ universal edges (otherwise, P would not be q-branched). Hence, by induction hypothesis, we can construct a q'-branched tree decomposition $(T^{(j)}, \iota^{(j)})$ for the subgraph induced by the vertices contained in the path $\pi^{(j)}$ with root $r^{(j)}$.
Now, we use the same construction as above to construct a path $(T' = (t'_1, \dots, t'_i), \iota')$ from $\pi_1, \dots, \pi_i$ and for each path $\pi^{(j)}$ we add the root $r^{(j)}$ of the q'-branched tree decomposition $(T^{(j)}, \iota(j))$ as a child of bag t_i to obtain our final tree decomposition (T, ι). As there is a universal edge from C^* to C_j, we know that C_j is a component of

C^*. As all $(T^{(j)}, \iota(j))$ are valid $q-1$-branched tree decompositions of width at most k, we can thus conclude that (T, ι) is a valid q-branched tree decomposition of width k.

This concludes the proof of the claim. □

Combining the above claim with Theorem 1 for computing the pit, we conclude that we can check whether a graph G has q-branched-treewidth k in time $O(|\operatorname{pit}(G, k+1)|^2 \cdot |V|^2)$. We note that the algorithm is fully constructive, as the obtained path (and, hence, the winning strategy of the searchers) directly corresponds to the desired decomposition. Since we have $\operatorname{tw}(G) = \operatorname{tw}_\infty(G)$ and $\operatorname{pw}(G) = \operatorname{tw}_0(G)$, the above results immediately imply the same statement for treewidth and pathwidth by checking $d(V(G), k) < \infty$ or $d(V(G), k) = 0$, respectively.

Appendix A.2. Computing Treedepth Decompositions

In order to show the statement for treedepth, we will require another claim for different weight functions. The proof idea is, however, very similar.

Claim A2. *Let $G = (V, E)$ be a graph and $k \in \mathbb{N}$. Define w_E as $(x, y) \mapsto 1$ and w_A as $(x, y) \mapsto 0$, and $c_0 = 1$. Then we have $d(V(G), Q) \leq k$ in $\operatorname{pit}(G, k)$ if, and only if, $\operatorname{td}(G) \leq k$.*

Proof. To prove the claim, we use an alternative representation of treedepth [58]. Let $G = (V, E)$ be a graph with connected components $C_1, \dots, C_\ell$, then:

$$\operatorname{td}(G) = \begin{cases} 1 & \text{if } |V| = 1; \\ \max_{i=1}^{\ell} \operatorname{td}(G[C_i]) & \text{if } \ell \geq 2; \\ \min_{v \in V} \operatorname{td}(G[V \setminus \{v\}]) + 1 & \text{otherwise.} \end{cases}$$

Let us reformulate this definition a bit. Let $C \subseteq V$ be a subset of the vertices and let $C_1, \dots, C_\ell$ be the connected components of $G[C]$. Define:

$$\operatorname{td}^*(C) = \begin{cases} 1 & \text{if } |C| = 1; \\ \max_{i=1}^{\ell} \operatorname{td}^*(C_i) & \text{if } \ell \geq 2; \\ \min_{v \in C} \operatorname{td}^*(C \setminus \{v\}) + 1 & \text{otherwise.} \end{cases}$$

Obviously, $\operatorname{td}(G) = \operatorname{td}^*(V)$. We proof that for any $C \subseteq V$ we have $d(C, Q) = \operatorname{td}^*(C)$ in $\operatorname{pit}(G, k)$ for every $k \geq \operatorname{td}(G)$ and $d(C, Q) \geq \operatorname{td}^*(C)$ for all $k < \operatorname{td}(G)$.

For the first part, we consider the vertices of $\operatorname{pit}(G, k)$ in inverse topological order and prove the claim by induction. The first vertex C_0 is in Q and thus $d(C_0, Q) = c_0 = 1$. Since the vertices in Q represent sets of cardinality 1, we have $d(C_0, Q) = \operatorname{td}^*(C_0)$. For the inductive step, consider C_i and first assume it is not connected in G. Then

$$\begin{aligned} d(C_i, Q) &= \max_{C_j \in N_\forall(C_i)} \big(d(C_j, Q) + w_A(C_i, C_j)\big) \\ &= \max_{C_j \in N_\forall(C_i)} d(C_j, Q) \\ &= \max_{C_j \text{ is a component in } G[C_i]} \operatorname{td}^*(C_j) \\ &= \operatorname{td}^*(C_i). \end{aligned}$$

Note that there could, of course, also be existential edges leaving C_i. However, since the universal edges are "for free", for every shortest path that uses an existential edge at C_i, there is also one that first uses the universal edges.

For the second case, that is C_i is connected, observe that C_i is not incident to any universal edge. Therefore, we obtain:

$$\begin{aligned} d(C_i, Q) &= \min_{v \in C_i} \left(d(C_i \setminus \{v\}, Q) + w_E(C_i, C_i \setminus \{v\}) \right) \\ &= \min_{v \in C_i} \left(d(C_i \setminus \{v\}, Q) + 1 \right) \\ &= \min_{v \in C_i} \left(\operatorname{td}^*(C_i \setminus \{v\}) + 1 \right) \\ &= \operatorname{td}^*(C_i). \end{aligned}$$

This completes the part of the proof that shows $d(C, Q) = \operatorname{td}^*(C)$ for $k \geq \operatorname{td}(G)$. We are left with the task to argue that $d(C, Q) \geq \operatorname{td}^*(C)$ for all $k < \operatorname{td}(G)$. This follows by the fact that for every $k' < k$ we have that $\operatorname{pit}(G, k')$ is an induced subgraph of $\operatorname{pit}(G, k)$. Therefore, the distance can only increase in the pit for a smaller k—in fact, the distance can even become infinity if $k < \operatorname{td}(G)$. □

Again, combining the claim with Theorem 1 yields the statement of the theorem for treedepth.

Appendix A.3. Computing Dependency Treewidth

This parameter can be characterized by an adaption of the graph searching game [12]: In addition to the graph G and the parameter k, one is also given a partial ordering $\lessdot$ on the vertices of G. For a vertex set V', let $\mu_{\lessdot}(V') = \{ v \in V' \mid \forall w \in V' \setminus \{v\} : (w, v) \notin \lessdot \}$ be the minimal elements of V' with regard to $\lessdot$. If $C \subseteq V(G)$ is the contaminated area, we are only allowed to put a searcher on $\mu_{\lessdot}(C)$, rather than on all of C. The *dependency-treewidth* $\operatorname{dtw}_{\lessdot}(G)$ is the minimal number of searchers required to catch the fugitive in this version of the game. Therefore, we just need a way to permit only existential edges (C, C') with $C \setminus C' \subseteq \mu_{\lessdot}(C)$. We show the following stronger claim:

Claim A3. *Consider a variant of the search game in which at some configurations C_i some fly-moves are forbidden, and in which, furthermore, at some configurations C_j no reveals are allowed. Whether k searcher have a winning strategy in this game can be decided in time $O(|\operatorname{pit}(G, k)|^2 \cdot |V|^2)$.*

Proof. First observe that, if the k searcher has a winning strategy S, this strategy corresponds to a path in $\operatorname{pit}(G, k)$. The reason is that searchers that are allowed to use all fly- and reveal-moves (and for which all winning strategies correspond to paths in $\operatorname{pit}(G, k)$) can, of course, use S as well. We compute the pit with Theorem 1.

Now, to find the restricted winning strategy, we initially set w_E and w_A to $(x, y) \mapsto 0$. Then, for any existential edge (C_i, C_j) that we wish to forbid, we set $w_E(C_i, C_j) = \infty$. Furthermore, for any node C at which we would like to forbid universal edges, we set $w_E(C_i, C_j) = \infty$ for all $C_j \in N_{\forall}(C_i)$. Finally, we search a path from $V(G)$ to Q of weight less than ∞ using Lemma 4. □

This completes the proof of Theorem 2.

References

1. Kreutzer, S. Algorithmic Meta-Theorems. *Electron. Colloq. Comput. Complex. (ECCC)* **2009**, *16*, 147.
2. Cygan, M.; Fomin, F.; Kowalik, L.; Lokshtanov, D.; Marx, D.; Pilipczuk, M.; Pilipczuk, M.; Saurabh, S. *Parameterized Algorithms*; Springer: Berlin/Heidelberg, Germany, 2015.
3. Berg, J.; Järvisalo, M.; Malone, B. Learning optimal bounded treewidth Bayesian networks via maximum satisfiability. In Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics, Reykjavik, Iceland, 22–25 April 2014; pp. 86–95.
4. Darwiche, A. A differential approach to inference in Bayesian networks. *J. ACM (JACM)* **2003**, *50*, 280–305. [CrossRef]
5. Elidan, G.; Gould, S. Learning bounded treewidth Bayesian networks. *J. Mach. Learn. Res.* **2008**, *9*, 2699–2731.
6. Kneis, J.; Langer, A.; Rossmanith, P. Courcelle's theorem—A game-theoretic approach. *Discret. Optim.* **2011**, *8*, 568–594. [CrossRef]
7. Bannach, M.; Berndt, S. Practical Access to Dynamic Programming on Tree Decompositions. *Algorithms* **2019**, *12*, 172. [CrossRef]

8. Bjesse, P.; Kukula, J.H.; Damiano, R.F.; Stanion, T.; Zhu, Y. Guiding SAT Diagnosis with Tree Decompositions. In Proceedings of the International Conference on Theory and Applications of Satisfiability Testing 2003, Santa Margherita Ligure, Italy, 5–8 May 2003; pp. 315–329.
9. Fichte, J.K.; Hecher, M.; Woltran, S.; Zisser, M. Weighted Model Counting on the GPU by Exploiting Small Treewidth. In Proceedings of the 26th Annual European Symposium on Algorithms (ESA 2018), Helsinki, Finland, 20–22 August 2018; pp. 28:1–28:16.
10. Habet, D.; Paris, L.; Terrioux, C. A Tree Decomposition Based Approach to Solve Structured SAT Instances. In Proceedings of the 2009 21st IEEE International Conference on Tools with Artificial Intelligence, Newark, NJ, USA , 2–4 November 2009; pp. 115–122.
11. Charwat, G.; Woltran, S. Dynamic Programming-based QBF Solving. In Proceedings of the QBF 2016, Bordeaux, France, 4 July 2016; pp. 27–40.
12. Eiben, E.; Ganian, R.; Ordyniak, S. Small Resolution Proofs for QBF using Dependency Treewidth. In Proceedings of the 35th Symposium on Theoretical Aspects of Computer Science (STACS 2018), Caen, France, 28 February–3 March 2018; Volume 96, pp. 28:1–28:15.
13. Karakashian, S.; Woodward, R.J.; Choueiry, B.Y. Improving the Performance of Consistency Algorithms by Localizing and Bolstering Propagation in a Tree Decomposition. In Proceedings of the AAAI Conference on Artificial Intelligence, Bellevue, WA, USA, 14–18 July 2013.
14. Koster, A.M.C.A.; van Hoesel, S.P.M.; Kolen, A.W.J. Solving partial constraint satisfaction problems with tree decomposition. *Networks* **2002**, *40*, 170–180. [CrossRef]
15. Eisenbrand, F.; Hunkenschröder, C.; Klein, K. Faster Algorithms for Integer Programs with Block Structure. In Proceedings of the 45th International Colloquium on Automata, Languages, and Programming (ICALP 2018), Prague, Czech Republic, 9–13 July 2018; Volume 107, pp. 49:1–49:13.
16. Ganian, R.; Ordyniak, S.; Ramanujan, M.S. Going Beyond Primal Treewidth for (M)ILP. In Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, San Francisco, CA, USA, 4–9 February 2017; pp. 815–821.
17. Ganian, R.; Ordyniak, S. The complexity landscape of decompositional parameters for ILP. *Artif. Intell.* **2018**, *257*, 61–71. [CrossRef]
18. Koutecký, M.; Levin, A.; Onn, S. A Parameterized Strongly Polynomial Algorithm for Block Structured Integer Programs. In Proceedings of the 45th International Colloquium on Automata, Languages, and Programming (ICALP 2018), Prague, Czech Republic, 9–13 July 2018; Volume 107, pp. 85:1–85:14.
19. Szeider, S. On Fixed-Parameter Tractable Parameterizations of SAT. In Proceedings of the Theory and Applications of Satisfiability Testing 2003, Santa Margherita Ligure, Italy, 5–8 May 2003; pp. 188–202.
20. Bannach, M.; Berndt, S.; Ehlers, T. Jdrasil: A Modular Library for Computing Tree Decompositions. In Proceedings of the 16th International Symposium on Experimental Algorithms (SEA 2017), London, UK, 21–23 June 2017; pp. 28:1–28:21.
21. Bonifati, A.; Martens, W.; Timm, T. An analytical study of large SPARQL query logs. *VLDB J.* **2020**, *29*, 655–679. [CrossRef]
22. Dudek, J.M.; Phan, V.H.N.; Vardi, M.Y. DPMC: Weighted Model Counting by Dynamic Programming on Project-Join Trees. In Proceedings of the Principles and Practice of Constraint Programming—26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, 7–11 September 2020; pp. 211–230. [CrossRef]
23. Dudek, J.M.; Phan, V.H.N.; Vardi, M.Y. ProCount: Weighted Projected Model Counting with Graded Project-Join Trees. In Proceedings of the Theory and Applications of Satisfiability Testing—SAT 2021—24th International Conference, Barcelona, Spain, 5–9 July 2021; pp. 152–170. [CrossRef]
24. Korhonen, T.; Järvisalo, M. Integrating Tree Decompositions into Decision Heuristics of Propositional Model Counters (Short Paper). In Proceedings of the 27th International Conference on Principles and Practice of Constraint Programming, CP 2021, (Virtual Conference), Montpellier, France, 25-29 October 2021; pp. 8:1–8:11. [CrossRef]
25. Maniu, S.; Senellart, P.; Jog, S. An Experimental Study of the Treewidth of Real-World Graph Data. In Proceedings of the 22nd International Conference on Database Theory, ICDT 2019, Lisbon, Portugal, 26-28 March 2019; pp. 12:1–12:18. [CrossRef]
26. 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. [CrossRef]
27. Iwata, Y.; Ogasawara, T.; Ohsaka, N. On the Power of Tree-Depth for Fully Polynomial FPT Algorithms. In Proceedings of the 35th Symposium on Theoretical Aspects of Computer Science, STACS 2018, Caen, France, 28 February–3 March 2018; pp. 41:1–41:14. [CrossRef]
28. Brand, C.; Koutecký, M.; Ordyniak, S. Parameterized Algorithms for MILPs with Small Treedepth. In Proceedings of the Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, 2–9 February 2021; pp. 12249–12257.
29. Bläsius, T.; Fischbeck, P.; Friedrich, T.; Katzmann, M. Solving Vertex Cover in Polynomial Time on Hyperbolic Random Graphs. In Proceedings of the 37th International Symposium on Theoretical Aspects of Computer Science, STACS 2020, Montpellier, France, 10–13 March 2020; pp. 25:1–25:14. [CrossRef]
30. Fomin, F.V.; Giannopoulou, A.C.; Pilipczuk, M. Computing Tree-Depth Faster Than 2^n. *Algorithmica* **2015**, *73*, 202–216. [CrossRef]

31. Reidl, F.; Rossmanith, P.; Villaamil, F.S.; Sikdar, S. A Faster Parameterized Algorithm for Treedepth. In Proceedings of the Automata, Languages, and Programming—41st International Colloquium, ICALP 2014, Copenhagen, Denmark, 8–11 July 2014; pp. 931–942. [CrossRef]
32. Bodlaender, H.L. A Linear-Time Algorithm for Finding Tree-Decompositions of Small Treewidth. *SIAM J. Comput.* **1996**, *25*, 1305–1317. [CrossRef]
33. Röhrig, H. Tree Decomposition: A Feasibility Study. Master's Thesis, Max-Planck-Institut für Informatik in Saarbrücken, Saarbrücken, Germany, 1998.
34. Gogate, V.; Dechter, R. A Complete Anytime Algorithm for Treewidth. In Proceedings of the UAI'04, 20th Conference in Uncertainty in Artificial Intelligence, Banff, AB, Canada, 7–11 July 2004; pp. 201–208.
35. Coudert, D.; Mazauric, D.; Nisse, N. Experimental Evaluation of a Branch-and-Bound Algorithm for Computing Pathwidth and Directed Pathwidth. *ACM J. Exp. Algorithmics* **2016**, *21*, 1.3:1–1.3:23. [CrossRef]
36. Trimble, J. An Algorithm for the Exact Treedepth Problem. In Proceedings of the 18th International Symposium on Experimental Algorithms, SEA 2020, Catania, Italy, 16–18 June 2020; pp. 19:1–19:14. [CrossRef]
37. Hamann, M.; Strasser, B. Graph Bisection with Pareto Optimization. *ACM J. Exp. Algorithmics* **2018**, *23*, 1–34. [CrossRef]
38. Bodlaender, H.L.; Koster, A.M.C.A. Treewidth computations I. Upper bounds. *Inf. Comput.* **2010**, *208*, 259–275. [CrossRef]
39. Tamaki, H. Computing Treewidth via Exact and Heuristic Lists of Minimal Separators. In Proceedings of the Analysis of Experimental Algorithms—Special Event, SEA^2 2019, Kalamata, Greece, 24–29 June 2019; pp. 219–236. [CrossRef]
40. Kobayashi, Y.; Komuro, K.; Tamaki, H. Search Space Reduction through Commitments in Pathwidth Computation: An Experimental Study. In Proceedings of the Experimental Algorithms—13th International Symposium, SEA 2014, Copenhagen, Denmark, 29 June–1 July 2014; pp. 388–399. [CrossRef]
41. Fernando Sánchez Villaamil. About Treedepth and Related Notions. Ph.D. Thesis, RWTH Aachen University, Aachen, Germany, 2017.
42. Kask, K.; Gelfand, A.; Otten, L.; Dechter, R. Pushing the Power of Stochastic Greedy Ordering Schemes for Inference in Graphical Models. In Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, CA, USA, 7–11 August 2011.
43. Dell, H.; Husfeldt, T.; Jansen, B.; Kaski, P.; Komusiewicz, C.; Rosamond, F. The First Parameterized Algorithms and Computational Experiments Challenge. In Proceedings of the 11th International Symposium on Parameterized and Exact Computation (IPEC 2016), Aarhus, Denmark, 24–26 August 2016; pp. 30:1–30:9.
44. Dell, H.; Komusiewicz, C.; Talmon, N.; Weller, M. The PACE 2017 Parameterized Algorithms and Computational Experiments Challenge: The Second Iteration. In Proceedings of the 12th International Symposium on Parameterized and Exact Computation (IPEC 2017), Vienna, Austria, 6–8 September 2017.
45. Kowalik, L.; Mucha, M.; Nadara, W.; Pilipczuk, M.; Sorge, M.; Wygocki, P. The PACE 2020 Parameterized Algorithms and Computational Experiments Challenge: Treedepth. In Proceedings of the 15th International Symposium on Parameterized and Exact Computation, IPEC 2020 (Virtual Conference), Hong Kong, China, 14–18 December 2020; pp. 37:1–37:18. [CrossRef]
46. Tamaki, H. Treewidth-Exact. 2016. Available online: github.com/TCS-Meiji/treewidth-exact (accessed on 2 August 2017).
47. Larisch, L.; Salfelder, F. p17. 2017. Available online: https://github.com/freetdi/p17 (accessed on 2 August 2017).
48. Trimble, J. PACE Solver Description: Bute-Plus: A Bottom-Up Exact Solver for Treedepth. In Proceedings of the 15th International Symposium on Parameterized and Exact Computation, IPEC 2020 (Virtual Conference), Hong Kong, China, 14–18 December 2020; pp. 34:1–34:4. [CrossRef]
49. Bannach, M.; Berndt, S.; Schuster, M.; Wienöbst, M. PACE Solver Description: PID*. In Proceedings of the 15th International Symposium on Parameterized and Exact Computation, IPEC 2020 (Virtual Conference), Hong Kong, China, 14–18 December 2020; pp. 28:1–28:4. [CrossRef]
50. Arnborg, S.; Corneil, D.G.; Proskurowski, A. Complexity of finding embeddings in ak-tree. *SIAM J. Algebr. Discret. Methods* **1987**, *8*, 277–284. [CrossRef]
51. Bouchitté, V.; Todinca, I. Listing all potential maximal cliques of a graph. *Theor. Comput. Sci.* **2002**, *276*, 17–32. [CrossRef]
52. Tamaki, H. Positive-instance driven dynamic programming for treewidth. *J. Comb. Optim.* **2019**, *37*, 1283–1311. [CrossRef]
53. Althaus, E.; Schnurbusch, D.; Wüschner, J.; Ziegler, S. On Tamaki's Algorithm to Compute Treewidths. In Proceedings of the 19th International Symposium on Experimental Algorithms, SEA 2021, Nice, France, 7–9 June 2021; pp. 9:1–9:18. [CrossRef]
54. Fomin, F.V.; Kratsch, D. *Exact Exponential Algorithms*; Texts in Theoretical Computer Science, An EATCS Series; Springer: Berlin/Heidelberg, Germany, 2010.
55. Halin, R. S-functions for graphs. *J. Geom.* **1976**, *8*, 171–186. [CrossRef]
56. Robertson, N.; Seymour, P.D. Graph minors. I. Excluding a forest. *JCT J. Comb. Theory* **1983**, *35*, 39–61. [CrossRef]
57. Robertson, N.; Seymour, P.D. Graph Minors. II. Algorithmic Aspects of Tree-Width. *Algorithms* **1986**, *7*, 309–322. [CrossRef]
58. Nesetril, J.; de Mendez, P.O. *Sparsity*; Springer: Berlin/Heidelberg, Germany, 2012.
59. Seymour, P.; Thomas, R. Graph Searching and a Min-Max Theorem for Tree-Width. *JCT J. Comb. Theory* **1993**, *58*, 22–33. [CrossRef]
60. Kirousis, L.; Papadimitriou, C. Searching and Pebbling. *TCS Theor. Comput. Sci.* **1986**, *47*, 205–218. [CrossRef]
61. Giannopoulou, A.; Hunter, P.; Thilikos, D. LIFO-search: A min-max theorem and a searching game for cycle-rank and tree-depth. *Discret. Appl. Math.* **2012**, *160*, 2089–2097. [CrossRef]

62. Fomin, F.; Fraigniaud, P.; Nisse, N. Nondeterministic Graph Searching: From Pathwidth to Treewidth. *Algorithmica* **2009**, *53*, 358–373. [CrossRef]
63. Tamaki, H. Experimental Analysis of Treewidth. In *Treewidth, Kernels, and Algorithms—Essays Dedicated to Hans L. Bodlaender on the Occasion of His 60th Birthday*; Springer: Berlin/Heidelberg, Germany, 2020, pp. 214–221._15. [CrossRef]
64. Korhonen, T. PACE Solver Description: SMS. In Proceedings of the 15th International Symposium on Parameterized and Exact Computation, IPEC 2020 (Virtual Conference), Hong Kong, China, 14–18 December 2020; pp. 30:1–30:4. [CrossRef]
65. Brokkelkamp, R.; van Venetië, R.; de Vries, M.J.; Westerdiep, J. PACE Solver Description: TdULL. In Proceedings of the 15th International Symposium on Parameterized and Exact Computation, IPEC 2020 (Virtual Conference), Hong Kong, China, 14–18 December 2020; pp. 29:1–29:4. [CrossRef]
66. Lodha, N.; Ordyniak, S.; Szeider, S. A SAT Approach to Branchwidth. *ACM Trans. Comput. Log.* **2019**, *20*, 15:1–15:24. [CrossRef]
67. Ramaswamy, V.P.; Szeider, S. MaxSAT-Based Postprocessing for Treedepth. In Proceedings of the Principles and Practice of Constraint Programming—26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, 7–11 September 2020; pp. 478–495. [CrossRef]
68. Ganian, R.; Lodha, N.; Ordyniak, S.; Szeider, S. SAT-Encodings for Treecut Width and Treedepth. In Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments, ALENEX 2019, San Diego, CA, USA, 7–8 January 2019; pp. 117–129. [CrossRef]
69. Lodha, N.; Ordyniak, S.; Szeider, S. SAT-Encodings for Special Treewidth and Pathwidth. In Proceedings of the Theory and Applications of Satisfiability Testing—SAT 2017—20th International Conference, Melbourne, Australia, 28 August 28–1 September 2017; pp. 429–445. [CrossRef]
70. Samer, M.; Veith, H. Encoding Treewidth into SAT. In Proceedings of the Theory and Applications of Satisfiability Testing—SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, 30 June–3 July 2009; pp. 45–50. [CrossRef]
71. Berg, J.; Järvisalo, M. SAT-Based Approaches to Treewidth Computation: An Evaluation. In Proceedings of the 26th IEEE International Conference on Tools with Artificial Intelligence, Limassol, Cyprus, 10–12 November 2014; pp. 328–335. [CrossRef]
72. Bannach, M.; Berndt, S. Positive-Instance Driven Dynamic Programming for Graph Searching. In Proceedings of the Algorithms and Data Structures—16th International Symposium, WADS 2019, Edmonton, AB, Canada, 5–7 August 2019; pp. 43–56. [CrossRef]
73. Safari, M. D-Width: A More Natural Measure for Directed Tree Width. In Proceedings of the Mathematical Foundations of Computer Science 2005, Gdansk, Poland, 29 August–2 September 2005; pp. 745–756.
74. Bienstock, D. Graph Searching, Path-Width, Tree-Width and Related Problems (A Survey). In Proceedings of the Reliability of Computer and Communication Networks, Proceedings of a DIMACS Workshop, New Brunswick, NJ, USA, 2–4 December 1989; pp. 33–50. [CrossRef]
75. Gruber, H.; Holzer, M. Finite Automata, Digraph Connectivity, and Regular Expression Size. In Proceedings of the International Colloquium on Automata, Languages, and Programming, Reykjavik, Iceland, 7–11 July 2008; pp. 39–50. [CrossRef]
76. Bienstock, D.; Seymour, P.D. Monotonicity in Graph Searching. *J. Algorithms* **1991**, *12*, 239–245. [CrossRef]
77. LaPaugh, A. Recontamination Does Not Help to Search a Graph. *ACM* **1993**, *40*, 224–245. [CrossRef]
78. Mazoit, F.; Nisse, N. Monotonicity of non-deterministic graph searching. *TCS Theor. Comput. Sci.* **2008**, *399*, 169–178. [CrossRef]
79. Immerman, N. *Descriptive Complexity*; Springer: Berlin/Heidelberg, Germany, 1999.
80. Kahn, A.B. Topological sorting of large networks. *Commun. ACM* **1962**, *5*, 558–562. [CrossRef]
81. Evans, W.; Hunter, P.; Safari, M. *D-Width and Cops and Robbers*; Technical Report, 2007; *unpublished*.
82. Savnik, I. Index Data Structure for Fast Subset and Superset Queries. In Proceedings of the 5th International Cross-Domain Conference on Vailability, Reliability, and Security in Information Systems and HCI, Regensburg, Germany, 2–6 September 2013; pp. 134–148. [CrossRef]
83. Flum, J.; Grohe, M. *Parameterized Complexity Theory*; Texts in Theoretical Computer Science, An EATCS Series; Springer: Berlin/Heidelberg, Germany, 2006.
84. Koster, A.M.C.A.; Bodlaender, H.L.; van Hoesel, S.P.M. Treewidth: Computational Experiments. *Electron. Notes Discret. Math.* **2001**, *8*, 54–57. [CrossRef]
85. Gugelmann, L.; Panagiotou, K.; Peter, U. Random Hyperbolic Graphs: Degree Sequence and Clustering—(Extended Abstract). In Proceedings of the Automata, Languages, and Programming—39th International Colloquium, ICALP 2012, Warwick, UK, 9–13 July 2012; pp. 573–585. [CrossRef]
86. Bläsius, T.; Friedrich, T.; Krohmer, A. Hyperbolic Random Graphs: Separators and Treewidth. In Proceedings of the 24th Annual European Symposium on Algorithms, Aarhus, Denmark, 22–24 August 2016; pp. 15:1–15:16. [CrossRef]
87. Aldecoa, R.; Orsini, C.; Krioukov, D.V. Hyperbolic graph generator. *Comput. Phys. Commun.* **2015**, *196*, 492–496. [CrossRef]
88. Courcelle, B. On the model-checking of monadic second-order formulas with edge set quantifications. *Discret. Appl. Math.* **2012**, *160*, 866–887. [CrossRef]
89. Bodlaender, H.L.; Kratsch, S.; Kreuzen, V.J.C.; Kwon, O.; Ok, S. Characterizing width two for variants of treewidth. *Discret. Appl. Math.* **2017**, *216*, 29–46. [CrossRef]