Understanding Model Counting for β-acyclic CNF-formulas

Johann Brault-Baron[*1], **Florent Capelli**[†2], **and Stefan Mengel**[‡3]

1 **LSV UMR 8643, ENS Cachan and Inria, France**
2 **IMJ UMR 7586 - Logique, Université Paris Diderot, France**
3 **LIX UMR 7161, École Polytechnique, France**

Abstract

We show that #SAT on β-acyclic CNF-formulas can be solved in polynomial time. In contrast to previous algorithms for other structurally restricted classes of formulas, our algorithm does not proceed by dynamic programming. Instead, it works along an elimination order, solving a weighted version of constraint satisfaction. We give evidence that this deviation from more standard algorithms is no coincidence by showing that it is outside of the framework recently proposed by Sæther et al. (SAT 2014) which subsumes all other structural tractability results for #SAT known so far.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems, G.2.1 Combinatorics, G.2.2 Graph Theory

Keywords and phrases model counting, hypergraph acyclicity, structural tractability

Digital Object Identifier 10.4230/LIPIcs.STACS.2015.143

1 Introduction

The propositional model counting problem #SAT is, given a CNF-formula F, to count the satisfying assignments of F. #SAT is the canonical #**P**-complete problem and is thus central to the area of counting complexity. Moreover, many important problems in artificial intelligence research reduce to #SAT (see e.g. [19]), so there is also great interest in the problem from a practical point of view.

Unfortunately, #SAT is computationally very hard: even for very restricted CNF-formulas, e.g. monotone 2-CNF-formulas, the problem is #**P**-hard and in fact even #**P**-hard to approximate [19]. Thus the focus of research in finding tractable classes of #SAT-instances has turned to so-called *structural* classes, which one gets by assigning a graph or hypergraph to a CNF-formula and then restricting the class of (hyper)graphs considered. The general idea is that if the (hyper)graph associated with an instance has a treelike decomposition that is "nice" enough, e.g. a tree decomposition of small width, then there is a dynamic programming algorithm that solves #SAT for the instance. In the recent years, many such dynamic programming algorithms for ever more general classes of graphs and hypergraphs have been found, see e.g. [13, 21, 18, 22, 7].

It had been an open question for some time how far this approach could be pushed, until very recently Sæther, Telle and Vatshelle, in a striking contribution [20], introduced a new

[*] This work has received support from the French Agence Nationale dela Recherche, AGGREG project reference ANR-14-CE25-0017-01.
[†] Partially supported by ANR Blanc CompA ANR-13-BS02-0001.
[‡] Partially supported by a grant from Qualcomm.

framework for #SAT. This framework which we call STV-framework is centered around a new width measure called PS-width and aims to formalize the most general class for which efficient dynamic programming for #SAT is possible (see Section 2.5 for details). We consider the STV-framework to be a very convincing formalization, delineating the limits of dynamic programming for #SAT. This belief is supported by the fact that in the full version of this paper we show that the STV-framework gives a uniform explanation for *all* previously known structural tractability results for #SAT.

In this article, we focus on *β-acyclic* CNF-formulas, i.e., formulas whose associated hypergraph is β-acyclic. There are several different reasonable ways of defining acyclicity of hypergraphs that have been proposed [12, 11], and β-acyclicity is the most general acyclicity notion generally considered in the literature for which #SAT could be tractable (see the discussions in [17, 7]). The complexity of #SAT for β-acyclic formulas is left as an open problem in [7] because it is interesting for several reasons: First, up to this paper, it was the only structural class of formulas for which we know that SAT is tractable [17] without this directly generalizing to a tractability result for #SAT. This is because the algorithm of [17] does *not* proceed by dynamic programming but uses resolution, a technique that is known to generally not generalize to counting. Moreover, β-acyclicity can be generalized to a width-measure [15], so there is hope that a good algorithm for β-acyclic formulas might generalize to wider classes for which even the status for SAT is left as an open problem in [17]. Since decomposition techniques based on hypergraph acyclicity tend to be more general than graph-based techniques [14], this might lead to large, new classes of tractable #SAT-instances.

The contribution of this paper is twofold: First, we show that #SAT on β-acyclic formulas is tractable. In fact, we show that a more general counting problem which we call *weighted counting for constraint satisfaction with default values*, for short #CSP$_{\mathrm{def}}$, is tractable on β-acyclic hypergraphs. We remark that there is another line of research on #CSP, the counting problem related to constraint satisfaction, where dichotomy theorems for weighted #CSP depending on fixed constraint languages are proven, see e.g. [5, 6]. We stress that we do *not* assume that the relations of our instances are fixed but we consider them as part of the input. Thus our results and those on fixed constraint languages are completely unrelated. Instead, the structural restrictions we consider are similar to those considered e.g. in [9], but since we allow clauses, resp. relations, of unbounded arity, our results and those of [9] are incomparable as well.

We note that our algorithm is very different in style from the algorithms for structural #SAT in the literature. Instead of doing dynamic programming along a decomposition, we proceed along a vertex elimination order which is more similar to the approach to SAT in [17]. But in contrast to using well-understood resolution techniques, we develop from scratch a procedure to update the weights of our #CSP$_{\mathrm{def}}$ instance along the elimination order. Our algorithm is non-obvious and novel, but it is relatively easy to write down and its correctness is easy to prove. Indeed, most of the work in the full version of this paper is spent on showing the polynomial runtime bound which requires a thorough understanding of how the weights of instances evolve during the algorithm. Unfortunately, these considerations cannot be presented in this version of this paper due to space restrictions.

Our second contribution is showing that our tractability result is not covered by the STV-framework, which, as discussed before, covers all other known structural tractability results for #SAT. In fact, we show that from [20] we cannot even get subexponential runtime bounds for β-acyclic #SAT. This can be seen as an explanation for why the algorithm for β-acyclic #SAT is so substantially different from the algorithms from the literature. We feel that the deviation from the usual dynamic programming techniques is not a coincidence but instead due to the fact that β-acyclic #SAT is the first known tractable class which is not explained by the unifying framework of [20]. Thus, our algorithm indeed introduces a new algorithmic technique for #SAT that allows the solution of instances that could not be solved with techniques known before.

2 Preliminaries and notation

2.1 Weighted counting for constraint satisfaction with default values

Let D and X be two sets. D^X denotes the set of functions from X to D. We think of X as a set of variables and of D as a domain, and thus we call $a \in D^X$ an *assignment* to the variables X. A *partial assignment* to the variables X is a mapping in D^Y where $Y \subseteq X$. If $a \in D^X$ and $Y \subseteq X$, we denote by $a|_Y$ the *restriction* of a onto Y. For $a \in D^X$ and $b \in D^Y$, we write $a \sim b$ if $a|_{X \cap Y} = b|_{X \cap Y}$ and if $a \sim b$, we denote by $a \cup b$ the mapping in $D^{X \cup Y}$ with $(a \cup b)(x) = a(x)$ if $x \in X$ and $(a \cup b)(x) = b(x)$ otherwise. Let $a \in D^X$, $y \notin X$ and $d \in D$. We write $a \oplus_y d := a \cup \{y \mapsto d\}$. We denote by $\mathbb{Q}_+$ the set of nonnegative rationals.

▶ **Definition 1.** A *weighted constraint with default value* $c = (f, \mu)$ on variables X and domain D is a function $f : S \to \mathbb{Q}_+$ with $S \subseteq D^X$ and $\mu \in \mathbb{Q}_+$. $S = \mathsf{supp}(c)$ is called the *support* of c, $\mu(c) = \mu$ is called the default value and we denote by $\mathsf{var}(c) = X$ the variables of c. We define the size $|c|$ of the constraint c to be $|c| := |S| \cdot |\mathsf{var}(c)|$. The constraint c naturally induces a total function on D^X, also denoted by c, defined by $c(a) := f(a)$ if $a \in S$ and $c(a) := \mu$ otherwise.

To ease the notation, when a is an assignment to a set $X \supseteq \mathsf{var}(c)$, we make the convention $c(a) = c(a|_{\mathsf{var}(c)})$. Observe that we do not assume $\mathsf{var}(c)$ to be non-empty. A constraint whose set of variables is empty has only one possible value in its support: the value associated with the empty assignment (the assignment that assigns no variable).

Since we only consider weighted constraints with default value in this paper, we will only say *weighted constraint* where the default value is always implicitly understood. Note that we restrict ourselves to non-negative weights, because non-negativity will be crucial in the proofs. This is not a problem in our context, non-negative numbers are sufficient to encode #SAT as we will see in Section 2.3. We however use rational numbers for convenience and all our results can be extended easily to non-negative algebraic numbers.

▶ **Definition 2.** The problem $\#\mathrm{CSP}_{\mathrm{def}}$ is the problem of computing, given a finite set I of weighted constraints on domain D, the *partition function*

$$w(I) = \sum_{a \in D^{\mathsf{var}(I)}} \prod_{c \in I} c(a),$$

where $\mathsf{var}(I) := \bigcup_{c \in I} \mathsf{var}(c)$.

The *size* $\|I\|$ of a $\#\mathrm{CSP}_{\mathrm{def}}$-instance I is defined to be $\|I\| := \sum_{c \in I} |c|$. Its *structural size* $s(I)$ of I is defined to be $s(I) := \sum_{c \in I} |\mathsf{var}(c)|$.

Note that the size of an instance as defined above roughly corresponds to that of an encoding in which the non-default values, i.e., the values on the support, are given by listing the support and the associated values in one table for each relation. We consider this convention as very natural and indeed it is near to the conventions in database theory and artificial intelligence.

Given an instance I, it will be useful to refer to subinstances of I, that is a set $J \subseteq I$. We will also refer to partition function of subinstances under some partial assignment, that is, the partition function of J where some of its variables are forced to a certain value. To this end, for $a \in D^W$, with $W \subseteq \mathsf{var}(I)$, and $J \subseteq I$ with $V' = \mathsf{var}(J)$ we define

$$w(J, a) := \sum_{\substack{b \in D^{V'} \\ a \sim b}} \prod_{c \in J} c(b).$$

2.2 Graphs and hypergraphs associated to CNF-formulas

We use standard notation for graphs which can e.g. be found in [10]. A *hypergraph* $\mathcal{H} = (V, E)$ consists of a finite set V and a set E of non-empty subsets of V. The elements of V are called *vertices* while the elements of E are called *edges*. As usual for graphs, we sometimes denote the vertex set of a hypergraph $\mathcal{H}$ by $V(\mathcal{H})$ and the edge set of $\mathcal{H}$ by $E(\mathcal{H})$. The size of a hypergraph is defined to be $\|\mathcal{H}\| = \sum_{e \in E(\mathcal{H})} |e|$.

We denote by $\mathcal{H} \setminus v$ the hypergraph we get from $\mathcal{H}$ after deleting v from $V(\mathcal{H})$ and all edges $e \in E(\mathcal{H})$ and then deleting the empty edge if it occurs.

We are interested in structural restrictions of the problem $\#\mathrm{CSP}_{\mathrm{def}}$. i.e., we restrict the way the variables interact in the different constraints. To formalize this, we introduce the hypergraph associated to an instance of $\#\mathrm{CSP}_{\mathrm{def}}$: The hypergraph $\mathcal{H}(I)$ associated to $\#\mathrm{CSP}_{\mathrm{def}}$-instance I is the hypergraph $\mathcal{H}(I) := (\mathsf{var}(I), E_I)$ where $E_I := \{\mathsf{var}(c) \mid c \in I\}$. The hypergraph of a CNF-formula is defined as $\mathcal{H}(F) := (\mathsf{var}(F), E_F)$ where $E_F := \{\mathsf{var}(C) \mid C \in \mathsf{cla}(F)\}$ where $\mathsf{var}(F)$ denotes the set of variables of F and $\mathsf{cla}(F)$ denotes the set of clauses of F.

The incidence graph $I(\mathcal{H})$ of a hypergraph $\mathcal{H} = (V, E)$ is the bipartite graph with the vertex set $V \cup E$ and an edge between $v \in V, e \in E$ if and only if $v \in e$. Similarly, the incidence graph $I(F)$ of a CNF-formula F has the vertex set $\mathsf{var}(F) \cup \mathsf{cla}(F)$ and $x \in \mathsf{var}(F)$ and $C \in \mathsf{cla}(F)$ are connected by an edge if and only if x appears in C.

2.3 Relation to #SAT

We show in this section how we can encode #SAT into $\#\mathrm{CSP}_{\mathrm{def}}$-instances with the same hypergraphs.

Classically, in CSP, all the solutions to a constraint are explicitly listed. For a CNF-formula however, each clause with n variables has $2^n - 1$ solutions, which would lead to a CSP-representation exponentially bigger than the CNF-formula. One way of dealing with this is encoding CNF-formulas into CSP-instances by listing all assignments that are *not* solution of a constraint, see e.g. [8]. In this encoding, each clause has only one counter-example and the corresponding CSP-instance is roughly of the same size as the CNF-formula.

The strength of the CSP with default values is that it can easily embed both representations. This leads to a polynomial reduction from #SAT to $\#\mathrm{CSP}_{\mathrm{def}}$.

▶ **Lemma 3.** *Given a CNF-formula F one can construct in polynomial time a $\#\mathrm{CSP}_{\mathrm{def}}$-instance I on variables $\mathsf{var}(F)$ and domain $\{0,1\}$ such that*

- $\mathcal{H}(F) = \mathcal{H}(I)$,
- *for all $a \in \{0,1\}^{\mathsf{var}(F)}$, a is a solution of F if and only if $\prod_{c \in I} c(a) = 1$, and 0 otherwise, and*
- $s(I) = \|I\| = |F|$.

Proof. For each clause C of F, we define a constraint c with default value 1 whose variables are the variables of C and such that $\mathsf{supp}(c) = \{a\}$ and $c(a) = 0$, where a is the only assignment of $\mathsf{var}(C)$ that is not a solution of C. It is easy to check that this construction has the above properties. ◀

2.4 β-acyclicity of hypergraphs

In this section we introduce the characterizations of β-acyclicity of hypergraphs we will use in this paper. We remark that there are many more characterizations, see e.g. [12, 3, 4].

▶ **Definition 4.** Let $\mathcal{H}$ be a hypergraph. A vertex $x \in V(\mathcal{H})$ is called a *nest point* if $\{e \in E(\mathcal{H}) \mid x \in e\}$ forms a sequence of sets increasing with respect to inclusion, that is $\{e \in E(\mathcal{H}) \mid x \in e\} = \{e_1, \ldots, e_k\}$ with $e_i \subseteq e_{i+1}$ for $i \in \{1, \ldots, k-1\}$.

A *β-elimination order* for $\mathcal{H}$ is defined inductively as follows:

- If $\mathcal{H} = \emptyset$, then only the empty tuple is a β-elimination order for $\mathcal{H}$.
- Otherwise, $(x_1, \ldots, x_n)$ is a β-elimination for $\mathcal{H}$ if x_1 is a nest point of $\mathcal{H}$ and $(x_2, \ldots, x_n)$ is a β-elimination order for $\mathcal{H} \setminus x_1$.

A hypergraph $\mathcal{H}$ called β-acyclic if and only if there exists a β-elimination order for $\mathcal{H}$.

One can easily show [4] that removing a nest point in a hypergraph does not change its β-acyclicity. Since deciding if a vertex is a nest point could be done in polnomial time, a greedy elimination of nest points yields a polynomial time algorithm to test the β-acyclicity of a hypergraph and to compute a β-elimination order if it exists.

We will also make use of another equivalent characterization of β-acyclic hypergraphs. A graph G is defined to be *chordal bipartite* if it is bipartite and every cycle of length at least 6 in G has a chord.

▶ **Theorem 5** ([1]). *A hypergraph is β-acyclic if and only if its incidence graph is chordal bipartite.*

We say that a #CSP$_{\text{def}}$-instance I is β-acyclic if $\mathcal{H}(I)$ is β-acyclic and we use an analogous convention for #SAT. Note that the incidence graph of an instance I and that of its hypergraph in general do not coincide, because I might contain several constraints with the same sets of variables. But with Theorem 5, it is not hard to see that the incidence graph of an instance I is chordal bipartite if and only if the incidence graph of the hypergraph of I is chordal bipartite, so we can interchangeably use both notions of incidence graphs in this paper without changing the class of instances.

Using Lemma 3 gives the following easy corollary.

▶ **Corollary 6.** #SAT *is polynomial time reducible to* #CSP$_{\text{def}}$*. Moreover,* #SAT *restricted to β-acyclic formulas is polynomial time reducible to* #CSP$_{\text{def}}$ *restricted to β-acyclic instances.*

2.5 Width measures of graphs and CNF-Formulas

In this section we introduce several width measures on graphs and CNF-formulas that are used when relating our algorithm for β-acyclic #CSP$_{\text{def}}$ to the framework of Sæther, Telle and Vatshelle [20]. Readers only interested in the algorithmic part of this paper may safely skip to Section 3.

We consider several width notions that are mainly defined by branch decompositions. For an introduction into this area and many more details see [23]. For a tree T we denote by

$L(T)$ the set of the leaves of T and by $V(T)$ the set of vertices of T. A *branch decomposition* (T, δ) of a graph $G = (V, E)$ consists of a subcubic tree T, i.e., a tree in which every vertex has degree at most 3, and a bijection δ between $L(T)$ and V. For convenience we often identify $L(T)$ and V. Moreover, it is often convenient to see a branch decomposition as rooted tree, and as this does not change any of the notions we define (see [23]), we generally follow this convention. For every $x \in V(T)$ we define T_x be the subtree of T rooted in x. From x we get a partition or *cut* of V into two sets defined by $(L(T_x), V \setminus L(T_x))$. For a set $X \subseteq V$ we often write $\bar{X}$ for $V \setminus X$.

Given a symmetric function $f : 2^V \times 2^V \to \mathbb{R}$ we define the f-width of a branch decomposition (T, δ) to be $\max_{x \in V(T)} f(L(T_x), V \setminus L(T_x))$, i.e., the f-width is the maximum value of f over all cuts of the vertices of T. The f-branch width of a graph G is defined as the minimum f-width of all branch decompositions of G.

Given a graph $G = (V, E)$ and a cut $(X, \bar{X})$ of V, we define $G[X, \bar{X}]$ to be the graph with vertex set V and edge set $\{uv \mid u \in X, v \in \bar{X}, uv \in E\}$.

We will use at several places the well-known notion of treewidth of a graph G, denoted by $\mathbf{tw}(G)$. Instead of working with the usual definition of treewidth (see e.g. [2]), it is more convenient for us to work with the strongly related notion of *Maximum-Matching-width* (for short *MM-width*) introduced by Vatshelle [23]. The MM-width of a graph G, denoted by $\mathbf{mmw}(G)$, is defined as the f-branch width of G for the function f that, given a cut $(X, \bar{X})$ of G, computes the size of the maximum matching of $G[X, \bar{X}]$. MM-width and treewidth are linearly related [23, p. 28].

▶ **Lemma 7.** *Let G be a graph, then* $\frac{1}{3}(\mathbf{tw}(G) + 1) \leq \mathbf{mmw}(G) \leq \mathbf{tw}(G) + 1$.

The *Maximum-Induced-Matching-width* (for short *MIM-width*) is another width measure of graphs that we will use extensively: The MIM-width of a graph G, denoted by $\mathbf{mimw}(G)$, is defined as the f-branch width of G for the function f that, given a cut $(X, \bar{X})$ of G, computes the size of the maximum induced matching of $G[X, \bar{X}]$.

Given a CNF-formula F, we say that a set of clauses $\mathcal{C} \subseteq \mathsf{cla}(F)$ is *projection satisfiable* if there is an assignment to F that satisfies all clauses in $\mathcal{C}$ and no clause in $\mathsf{cla}(F) \setminus \mathcal{C}$. The PS-value of F is defined to be the number of projection satisfiable subsets of $\mathsf{cla}(F)$. Let F be a CNF-formula, $X \subseteq \mathsf{var}(F)$ and $\mathcal{C} \subseteq \mathsf{cla}(F)$. Then we denote by $F_{X,\mathcal{C}}$ the formula we get from F by deleting first every clause not in $\mathcal{C}$ and then every variable not in X.

Let $I(F)$ be the incidence graph of F and let $(A, \bar{A})$ be a cut of $I(F)$. Let $X := \mathsf{var}(F) \cap A$, $\bar{X} := \mathsf{var}(F) \cap \bar{A}$, $\mathcal{C} := \mathsf{cla}(F) \cap A$ and $\bar{\mathcal{C}} := \mathsf{cla}(F) \cap \bar{A}$. Let $ps(A, \bar{A})$ be the maximum of the PS-values of $F_{X,\bar{\mathcal{C}}}$ and $F_{\bar{X},\mathcal{C}}$. Then the PS-width of a branch decomposition (T, δ) of $I(F)$ is defined as the ps-branch width of (T, δ). Moreover, the PS-width of F, denoted $\mathbf{psw}(F)$, is defined to be the ps-branch width of $I(F)$.

Let us try to give an intuition why we believe that PS-width is a good notion to model the limits of tractable dynamic programming for #SAT: The dynamic programming algorithms in the literature typically proceed by cutting instances into subinstances and then iteratively solving the instance along these cuts. During this process, some information has to be propagated between the subinstances. Intuitively, a minimum amount of such information is which sets of clauses are already satisfied by certain assignments and which clauses still have to be satisfied later in the process. In doing this, the individual clauses can be "bundled together" if they are satisfied by an assignment simultaneously. The number such bundles is exactly the PS-width of a cut, so we feel that PS-width is a good formalization of the minimum amount of information that has to be propagated during dynamic programming in the style of the algorithms from the literature.

Not only is PS-width in our opinion a good measure for the limits of dynamic programming, but Sæther, Telle and Vatshelle also showed that it allows efficient solving of #SAT.

▶ **Theorem 8** ([20]). *Given a CNF-formula F with n variables and m clauses and of size s, and a branch decomposition (T, δ) of the incidence graph $I(F)$ of F with PS-width k, one can count the number of satisfying assignments of F in time $O(k^3 s(m + n))$.*

We admit that the intuition explained above is rather vague and informal, so the reader might or might not share it, but we stress that it is supported more rigorously by the fact that all known tractability results from the literature that were shown by dynamic programming can be explained by a combination of PS-width and Theorem 8 (see the full version of this paper).

3 The algorithm

In this section we describe an algorithm that, given an instance I of $\#\mathrm{CSP}_{\mathrm{def}}$ on domain D and a nest point x of $\mathcal{H}(I)$, constructs in a polynomial number of arithmetic operations an instance I' such that $\mathcal{H}(I') = \mathcal{H}(I) \setminus x$, $\|I'\| \leq \|I\|$ and $w(I) = |D|w(I')$. We then explain that if I is β-acyclic, we can iterate the procedure to compute $w(I)$ in a polynomial number of arithmetic operations.

To make the presentation of the algorithm more clear, we will first consider a special case before presenting the algorithm for the general case.

3.1 The special case of nested constraints

In this section we consider $\#\mathrm{CSP}_{\mathrm{def}}$-instances whose variable scopes are nested, i.e., instances of the form $I = \{c_1, \ldots, c_p\}$ where $\mathsf{var}(c_1) \subseteq \ldots \subseteq \mathsf{var}(c_p)$. Note that these instances are β-acyclic, but of course there are β-acyclic instances not of this form which will be treated in the next section. Let us first sketch the idea behind the algorithm.

Fix an instance I as above and let $x \in \mathsf{var}(c_1)$. Observe that x chosen this way is a nest point of $\mathcal{H}(I)$. We want to eliminate x from I to get an instance I' such that

$$w(I) = |D|w(I'). \tag{1}$$

For each constraint c_i, the instance I' will have a constraint c'_i in the variables $\mathsf{var}(c_i) \setminus \{x\}$.

The idea behind the computation of the weights of I' is as follows: For every subinstance $I_i = \{c_1, \ldots, c_i\}$ we want for all assignments a to $\mathsf{var}(I')$ that $w(I_i, a) = |D|w(I'_i, a)$ where $I'_i = \{c'_1, \ldots, c'_i\}$. Note that (1) follows directly from this. Since a is an assignment to all the variables of I', we have $w(I'_i, a) = \prod_{j=1}^{i} c'_j(a)$. So let us compute the weights of the c'_j.

For $i = 1$ we have

$$c'_1(a) = w(I'_1, a) = \frac{w(I_1, a)}{|D|} = \frac{\sum_{d \in D} c_1(a \oplus_x d)}{|D|}.$$

For $i > 1$ we have $|D| \prod_{j=1}^{i} c'_j(a) = |D|w(I'_i, a) = w(I_i, a) = \sum_{d \in D} \prod_{j=1}^{i} c_i(a \oplus_x d)$. It follows that

$$c'_i(a) = \frac{\sum_{d \in D} \prod_{j=1}^{i} c_i(a \oplus_x d)}{|D| \prod_{j=1}^{i-1} c'_j(a)} = \frac{\sum_{d \in D} \prod_{j=1}^{i} c_i(a \oplus_x d)}{|D|w(I'_{i-1}, a)}.$$

By construction we have $|D|w(I'_{i-1}, a) = w(I_{i-1}, a) = \sum_{d \in D} \prod_{j=1}^{i-1} c_j(a \oplus_x d)$. The case where the denominator is zero will be dealt with in the proof Theorem 9.

Choosing the weights for the c'_i as described above, yields an instance I' that satisfies (1). Iterating this process then yields an algorithm to solve the instance I. The following theorem formalizes the above discussion, applies some arithmetic simplifications and analyzes the number of arithmetic operations performed in the procedure.

▶ **Theorem 9.** *Let I be a set of weighted constraints on domain D of the form $I = \{c_1, \ldots, c_p\}$ where $\mathsf{var}(c_1) \subseteq \ldots \subseteq \mathsf{var}(c_p)$. Let $x \in \mathsf{var}(c_1)$.*

We define a new instance $I' := \{c'_1, \ldots, c'_p\}$ such that $c'_i := (f'_i, \mu_i)$ is the weighted constraint on variables $\mathsf{var}(c'_i) = \mathsf{var}(c_i) \setminus \{x\}$, with default value $\mu(c_i)$ and $\mathsf{supp}(c'_i) := \{a \in D^{\mathsf{var}(c'_i)} \mid \exists d \in D, (a \oplus_x d) \in \mathsf{supp}(c_i)\}$. Moreover, for all $a \in \mathsf{supp}(c'_i)$, define

$$f'_1(a) := \frac{\sum_{d\in D} c_1(a \oplus_x d)}{|D|}, \text{ and } f'_i(a) := \frac{\sum_{d\in D} \prod_{j=1}^{i} c_j(a \oplus_x d)}{\sum_{d\in D} \prod_{j=1}^{i-1} c_j(a \oplus_x d)}, \text{ for } i > 1$$

if the denominator is non-zero and $f'_i(a) := 0$ otherwise.

Then $\mathcal{H}(I') = \mathcal{H}(I) \setminus x$, $\|I'\| \leq \|I\|$ and $w(I) = |D|w(I')$. Moreover, one can compute I' with $O(p\|I\|)$ arithmetic operations.

Proof. Observe first that for $a \in \mathsf{supp}(c'_i)$ and $j \leq i$ and $d \in D$, the assignment $a \oplus_x d$ assigns values to all variables of c_j, since $\mathsf{var}(c_j) \subseteq \mathsf{var}(c_i)$ and thus all terms defined above are well-defined.

$\mathcal{H}(I') = \mathcal{H}(I) \setminus x$ is obvious because for every i we have $\mathsf{var}(c'_i) = \mathsf{var}(c_i) \setminus \{x\}$.

Moreover, $\|I'\| \leq \|I\|$ because for every i, $|c'_i| \leq |c_i|$ since $|\mathsf{supp}(c'_i)| = |\{a \in D^{\mathsf{var}(c'_i)} \mid \exists d \in D, (a \oplus_x d) \in \mathsf{supp}(c_i)\}| \leq |\mathsf{supp}(c_i)|$.

Inspired by the discussion at the beginning of this section, we now show by induction on i that for all $a \in D^{\mathsf{var}(c'_i)}$,

$$|D| \prod_{j=1}^{i} c'_j(a) = \sum_{d\in D} \prod_{j=1}^{i} c_j(a \oplus_x d).$$

For $i = 1$, let $a \in D^{\mathsf{var}(c'_1)}$. If $a \in \mathsf{supp}(c'_1)$, this statement is clear from the definition.

If $a \notin \mathsf{supp}(c'_1)$, then for all $d \in D$, $(a \oplus_x d) \notin \mathsf{supp}(c_1)$. Thus $c_1(a \oplus_x d) = \mu_1$ for all d and consequently $\sum_{d\in D} c_1(a \oplus_x d) = |D|\mu_1 = |D|c'_1(a)$.

Now suppose that the result holds for i. Let $a \in D^{\mathsf{var}(c'_i)}$. Then we get by induction

$$|D| \prod_{j=1}^{i+1} c'_j(a) = \Big(\sum_{d\in D} \prod_{j=1}^{i} c_j(a \oplus_x d)\Big) c'_{i+1}(a).$$

First, assume that $\sum_{d\in D} \prod_{j=1}^{i} c_j(a \oplus_x d) = 0$. Since this is a sum of non-negative rationals, we have that for all d, $\prod_{j=1}^{i} c_j(a \oplus_x d) = 0$. Thus, $\prod_{j=1}^{i+1} c_j(a \oplus_x d) = 0$ for all d and it follows that $\sum_{d\in D} \prod_{j=1}^{i+1} c_j(a \oplus_x d) = 0$ which proves the claim.

Now assume that $\sum_{d\in D} \prod_{j=1}^{i} c_j(a \oplus_x d) \neq 0$. If $a \in \mathsf{supp}(c'_{i+1})$, then by definition of c'_{i+1}, the claim follows directly.

If $a \notin \mathsf{supp}(c_{i+1})$, then $\prod_{j=1}^{i+1} c_j(a \oplus_x d) = \mu_{i+1} \prod_{j=1}^{i} c_j(a \oplus_x d)$ for all d. Thus

$$\sum_{d\in D} \prod_{j=1}^{i+1} c_j(a \oplus_x d) = \mu_{i+1} \sum_{d\in D} \prod_{j=1}^{i} c_j(a \oplus_x d) = c'_{i+1}(a) \sum_{d\in D} \prod_{j=1}^{i} c_j(a \oplus_x d),$$

which establishes the claim for $i + 1$.

Applying the result for $i = p$, we get

$$|D|w(I', a) = |D| \prod_{j=1}^{p} c'_j(a) = \sum_{d \in D} \prod_{j=1}^{p} c_j(a \oplus_x d) = w(I, a).$$

It follows directly that $w(I) = |D|w(I')$ as desired.

We now analyze the number of arithmetic operations we make in the construction of I'. Clearly, if we have computed the $\sum_{d \in D} \prod_{j=1}^{i} c_j(a \oplus_x d)$ for all $i \leq p$ and $a \in \mathsf{supp}(c'_i)$ then we can compute $c'_i(a)$ with one division. Thus we need to do p divisions. Moreover, if we have computed $\prod_{j=1}^{i} c_j(a \oplus_x d)$, then we only need one more multiplication to compute $\prod_{j=1}^{i+1} c_j(a \oplus_x d)$.

Now, we prove by induction on i that $\prod_{j=1}^{i} c_j(a \oplus_x d)$ can take at most $1 + \sum_{j=1}^{i} |c_j|$ different values when a varies. This is trivial for $i = 0$. Now remark that if $a \oplus_x d \notin \mathsf{supp}(c_i)$, then $\prod_{j=1}^{i} c_j(a \oplus_x d) = \mu_i \prod_{j=1}^{i-1} c_j(a \oplus_x d)$, thus by induction, we have at most $1 + \sum_{j=1}^{i-1} |c_j|$ different values for $\prod_{j=1}^{i} c_j(a \oplus_x d)$. Moreover, there are at most $|\mathsf{supp}(c_i)| \leq |c_i|$ other values for $a \oplus_x d \in \mathsf{supp}(c_i)$, which completes the induction.

It follows that we have to compute at most $O(p\|I\|)$ different values for the $\prod_{j=1}^{i} c_j(a \oplus_x d)$ which can be done with $O(p\|I\|)$ multiplications. Now if i is fixed, for all a, the sum $\sum_{d \in D} \prod_{j=1}^{i} c_j(a \oplus_x d)$ has at most $1 + \sum_{j=1}^{i} |c_j|$ different terms that are already computed. Thus we only need $O(\|I\|)$ operations to compute each of them. As there are p different sums to compute, we can do everything with $O(p\|I\|)$ arithmetic operations. ◀

3.2 The general case

In this section, we extend Theorem 9 to the case where we have a general nest point in a $\#\mathrm{CSP}_{\mathrm{def}}$-instance. This will allow us to solve $\#\mathrm{CSP}_{\mathrm{def}}$ for all β-acyclic instances.

In the following, for $x \in \mathsf{var}(I)$, we denote by $I(x) = \{c \in I \mid x \in \mathsf{var}(c)\}$.

▶ **Theorem 10.** *Let I be a set of weighted constraints on domain D and x a nest point of $\mathcal{H}(I)$. Let $I(x) = \{c_1, \ldots, c_p\}$ with $\mathsf{var}(c_1) \subseteq \ldots \subseteq \mathsf{var}(c_p)$. Let $I' = \{c' \mid c \in I\}$ where*

- *if $c \notin I(x)$ then $c' := c$*
- *if $c = c_i$, then $c'_i := (f'_i, \mu_i)$ is the weighted constraint on variables $\mathsf{var}(c'_i) = \mathsf{var}(c_i) \setminus \{x\}$, with default value $\mu(c_i)$ and $\mathsf{supp}(c'_i) := \{a \in D^{\mathsf{var}(c'_i)} \mid \exists d \in D, (a \oplus_x d) \in \mathsf{supp}(c_i)\}$. Moreover, for all $a \in \mathsf{supp}(c'_i)$, define*

$$f'_1(a) := \frac{\sum_{d \in D} c_1(a \oplus_x d)}{|D|} \quad \text{and} \quad f'_i(a) := \frac{\sum_{d \in D} \prod_{j=1}^{i} c_j(a \oplus_x d)}{\sum_{d \in D} \prod_{j=1}^{i-1} c_j(a \oplus_x d)}, \text{ for } i > 1$$

  if the denominator is non-zero and $f'_i(a) := 0$ otherwise.

Then $\mathcal{H}(I') = \mathcal{H}(I) \setminus x$, $\|I'\| \leq \|I\|$ and $w(I) = |D|w(I')$. Moreover, one can compute I' with a $O(p\|I(x)\|)$ arithmetic operations.

Proof. Note first that the definition of the c'_i is identical to the construction in Theorem 9.

Let us explain why I' is well-defined. As x is a nest point, we can write $I(x) = \{c_1, \ldots, c_p\}$ with $\mathsf{var}(c_1) \subseteq \ldots \subseteq \mathsf{var}(c_p)$. If two constraints have the same variables, we choose an arbitrary order for them. Note that in the full version we will choose a specific order that ensures that the algorithm runs in polynomial time on a RAM, but in this proof any order will do.

$\mathcal{H}(I') = \mathcal{H}(I) \setminus x$ and $\|I'\| \leq \|I\|$ is shown as before. Moreover, the bound on the number of arithmetic operations follows from Theorem 9.

We have $w(I) = \sum_{a \in D^{\mathsf{var}(I)\setminus\{x\}}} \sum_{d \in D} \prod_{i=1}^{p} c_i(a \oplus_x d) \prod_{c \notin I(x)} c(a \oplus_x d)$. For $c \notin I(x)$ we have $c(a \oplus_x d) = c(a) = c'(a)$ by definition. Moreover, we have seen in the proof of Theorem 9 that $\sum_{d \in D} \prod_{i=1}^{p} c_i(a \oplus_x d) = |D| \prod_{i=1}^{p} c'_i(a)$. It follows that

$$w(I) = |D| \sum_{a \in D^{\mathsf{var}(I)\setminus\{x\}}} \prod_{i=1}^{p} c'_i(a) \prod_{c \notin I(x)} c'(a) = |D| w(I').$$

◀

Our algorithm for $\#\mathrm{CSP}_{\mathrm{def}}$ iteratively applies the procedure of Theorem 10 along a β-elimination order. Clearly, this only uses a polynomial number of arithmetic operations, because applying Theorem 10 only needs a polynomial number of arithmetic operations and the size of the instances is decreasing. Note that this does not directly give that the algorithm runs in polynomial time. We iteratively multiply and divide numbers which could lead to an iterative squaring effect and result in numbers of exponential bitsize. However, in the full version of this paper, we show that this actually does not happen: The weights computed in each step are essentially a ratio of weights of two subinstances of the original instance. It follows that the bitsize of the numerator and the denominator of all numbers is polynomially bounded and thus all arithmetic operations can be done in polynomial time. Implementing this yields the following result.

▶ **Theorem 11.** *There exists an algorithm that, given a β-acyclic instance I of $\#\mathrm{CSP}_{\mathrm{def}}$, computes $w(I)$ in polynomial time.*

Combining Theorem 11 and Corollary 6 we get the main tractability result for #SAT.

▶ **Corollary 12.** #SAT *on β-acyclic CNF-formulas can be solved in polynomial time.*

4 Relation to the STV-framework

In this section we compare our algorithmic result for #SAT on β-acyclic hypergraphs to the framework proposed by Sæther, Telle and Vatshelle in [20] which we call for short the STV-framework. In the full version of this paper we show that the STV-framework gives a uniform explanation of all tractability results for #SAT in the literature, extending the results of [20]. We see this as strong evidence that the STV-framework is indeed a good formalization of the intuitive notion of "dynamic programming for #SAT".

To show that our result is not covered by the STV-framework, we now show that it cannot give any subexponential time algorithms for β-acyclic #SAT. To this end, we prove an exponential lower bound on the PS-width of β-acyclic CNF-formulas.

We start off with a simple lemma.

We remind the reader that a CNF-formula F is called *monotone* if all variables appear only positively in F.

▶ **Lemma 13.** *For every bipartite graph G there is a monotone CNF-formula F such that F has the incidence graph G and $\mathbf{psw}(F) \geq 2^{\mathbf{mimw}(G)/2}$.*

Proof. We construct F by choosing arbitrarily one color class of G to represent clauses and the other one to represent variables. This choice then uniquely yields a monotone formula where a clause C contains a variable x if and only if x is connected to C by an edge in G.

Let (T, δ) be a branch decomposition of G and F. Let t be a vertex of T with cut $(A, \bar{A})$. Set $X := \mathsf{var}(F) \cap A$, $\bar{X} := \mathsf{var}(F) \cap \bar{A}$, $\mathcal{C} := \mathsf{cla}(F) \cap A$ and $\bar{\mathcal{C}} := \mathsf{cla}(F) \cap \bar{A}$. Moreover, let M be a maximum induced matching of $G[A, \bar{A}]$ and let V_M be the end vertices of M.

First assume that $|\mathcal{C} \cap V_M| \geq |\bar{\mathcal{C}} \cap V_M|$. Let $C_1, \ldots, C_k$ be the clauses in $\mathcal{C} \cap V_M$ and let $x_1, \ldots, x_k$ be variables in $\bar{X} \cap V_M$. Note that $k \geq |M|/2$. Since M is an induced matching, every clause C_i contains exactly one of the variables x_j, and we assume w.l.o.g. that C_i contains x_i. Let a be an assignment to the x_i and let a' be the extended assignment of $\bar{X}$ that we get by assigning 0 to all other variables. Then a' satisfies in $F_{\bar{X},C}$ exactly the clauses C_i for which $a(x_i) = 1$ since the formula is monotone. Since there are 2^k assignments to the x_i, we have $|PS(F_{\bar{X},C})| \geq 2^k \geq 2^{|M|/2}$.

For $|\mathcal{C} \cap V_M| \leq |\bar{\mathcal{C}} \cap V_M|$ it follows symmetrically that $|PS(F_{X,\bar{C}})| \geq 2^{|M|/2}$. Consequently, we have in either case that the PS-width of F is at least $2^{|M|/2}$ and the claim follows. ◀

We will now define for every graph G a graph G'. The construction will be such that, if G is chosen in the right way, then G' will be chordal bipartite and of high MIM-width. Combining this with Lemma 13 yields a class of β-acyclic CNF-formulas of high PS-width. Since PS-width is the crucial parameter in the STV-framework, this shows that β-acyclic #SAT cannot be solved efficiently by this framework.

Given a graph $G = (V, E)$ we define $G' = (V', E')$ as follows:

- for every $v \in V$ there are two vertices $x_v, y_v \in V'$,
- for every edge $e = uv \in E$ there are four vertices $p_{e,u}, q_{e,u}, p_{e,v}, q_{e,v} \in V'$,
- every $u, v \in V$ we add the edge $x_v y_u$ to E', and
- for every edge $e = uv \in E$ we add the edges $p_{e,u}q_{e,u}, p_{e,v}q_{e,v}, x_u p_{e,u}, y_v q_{e,u}, x_v p_{e,v}, y_u q_{e,v}$.

These are all vertices and edges of G'.

▶ **Lemma 14.** *G' is chordal bipartite.*

▶ **Lemma 15.** *Let G be bipartite. Then* $\mathbf{tw}(G) \leq 6\mathbf{mimw}(G')$.

Proof. Let (T', δ') be a branch decomposition of G'. Let $A, B \subseteq V(G)$ be the two colour classes of G. We construct a branch decomposition (T, δ) of G by deleting the leaves labeled with $p_{e,u}, q_{e,u}, p_{e,v}, q_{e,v}$, and those labeled x_v for $v \in A$ or with y_v for $v \in B$. Then we delete all internal vertices of of T' that have become leaves by these deletions until we get a branch decomposition T with the leaves x_v for $v \in B$ and y_v for $v \in A$. For the leaves of T we define $\delta(t) := v$ where $v \in V$ is such that $\delta'(t) = x_v$ or $\delta'(t) = y_v$. The result (T, δ) is a branch decomposition of G.

Let t be a vertex of T with the corresponding cut $(X, \bar{X})$. Let $M \subseteq E$ be a matching in $G[X, \bar{X}]$. Let $(X', \bar{X}')$ be the cut of t in (T', δ'). Let $e = uv \in M$, then x_u and y_v are on different sides of the cut X' and they are connected by the path $x_u p_{e,u} q_{e,u} y_v$. Consequently, there is at least one edge along this path in $G'[X', \bar{X}']$. Choose one such edge arbitrarily.

Let M' be the set of edges we have chosen for the different edges in M. Let M'_x be the set of edges in M' that do not have an end vertex y_v and let M'_y be the set of edges in M' that do not have an end vertex x_v. Let M'' be the bigger of these two sets. Since $e' \in M'$ can only have an end vertex x_v or y_u but not both, we have $|M'_x| + |M'_y| \geq |M'|$ and thus $|M''| \geq |M'|/2$.

We claim that M'' is an induced matching in G'. Clearly, M' is a matching because M is one. Consequently, $M'' \subseteq M'$ is also a matching. We now show that M'' is also induced. By way of contradiction, assume this were not true. Then there must be two adjacent vertices $u, v \in V'$ that are end vertices of edges in M'' but not in the same edge in M''. If $u = p_{e',w}$ for some $e' \in E$ and $w \in V$, then v must be x_w. But then by construction of M', the vertex

w must be incident to two edges in M which contradicts M being a matching. Similarly, we can rule out that v is $q_{e,w}$. Thus, u must be x_w or y_w and v must be $x_{w'}$ or $y_{w'}$. Since x_w and $x_{w'}$ are in the same colour class of G', they are not adjacent. Similarly y_w and $y_{w'}$ are not adjacent. Consequently, we may assume that $u = x_w$ and $v = y_{w'}$. But then they cannot both be an endpoint of an edge in M'' by construction of M''. Thus M'' is induced.

By Lemma 7 we know that there is a $t \in T$ with cut $(X, \bar{X})$ such that we can find a matching M of size at least $\frac{\mathbf{tw}(G)}{3}$ in $G[X, \bar{X}]$. By the construction above the corresponding cut $(X', \bar{X}')$ yields an induced matching of size $\frac{\mathbf{tw}(G)}{6}$ in $G'[X', \bar{X}']$. This completes the proof. ◀

The connection between expansion and treewidth (see [16]) yields the following lemma.

▶ **Lemma 16.** *There is a family $\mathcal{G}$ of graphs and constants $c > 0$ and $d \in \mathbb{N}$ such that for every $G \in \mathcal{G}$ the graph G has maximum degree d and we have $\mathbf{tw}(G) \geq c|E(G)|$.*

▶ **Corollary 17.** *There is a family $\mathcal{G}'$ of chordal bipartite graphs and a constant c such that for every graph $G \in \mathcal{G}$ we have $\mathbf{mimw}(G) \geq c|V(G)|$.*

Proof. Let $\mathcal{G}$ be the class of Lemma 16. We first transform every graph $G \in \mathcal{G}$ into a bipartite one G_1 by subdividing every edge, i.e., by introducing for each edge $e = uv$ a new vertex w_e and by replacing e by uw_e and w_ev. It is well-known that subdividing edges does not decrease the treewidth of a graph (see e.g. [10]), and thus $\mathbf{tw}(G) \leq \mathbf{tw}(G_1)$. Moreover, $|E(G_1)| = 2|E(G)|$, and thus $\mathbf{tw}(G_1) \geq \frac{1}{2}c|E(G_1)|$. Now let $\mathcal{G}' = \{G_1' \mid G \in \mathcal{G}\}$. Then the graphs in $\mathcal{G}'$ are chordal bipartite by Lemma 14 and the bound on the MIM-width follows by combining Lemma 16 and Lemma 15. ◀

Combining Corollary 17 and Lemma 13 yields the main result of this section.

▶ **Corollary 18.** *There is a family of monotone β-acyclic CNF-formulas of PS-width $2^{\Omega(n)}$ where n is the number of variables in the formulas.*

Since the runtime in Theorem 8 depends linearly on the PS-width, we get that the STV-framework cannot prove subexponential runtime bounds for #SAT on β-acyclic formulas.

5 Conclusion

We have shown that β-acyclic #SAT can be solved in polynomial time, a question left open in [7]. Our algorithm does not follow the dynamic programming approach that was used in all other structural tractability results that were known before, and as we have seen this is no coincidence. Instead, β-acyclic #SAT lies outside the STV-framework of [20] that explains all earlier results in a uniform way.

We close this paper with several open problems that we feel should be explored in the future. First, our algorithm for #SAT is specifically designed for the case of β-acyclic formulas, but we feel that the techniques developed might possibly be extended to other classes of hypergraphs that one can characterize by elimination orders. In this direction, it would be interesting to see if hypergraphs of bounded β-hypertree width, a width measure generalizing β-acyclicity proposed in [15], can be characterized by elimination orders and if such a characterization can be used to solve #SAT on the respective instances. Note that this case lies outside of the STV-framework, therefore dynamic programming without new ingredients is unlikely to work. Also, even the complexity of deciding SAT on instances of bounded β-hypertree width is an open problem [17].

It might also be interesting to generalize our algorithm to solve cases for which we already have polynomial time algorithms. For example, is there any uniform explanation for tractability of bounded cliquewidth #SAT and β-acyclic #SAT, similarly to the way in which the framework of [20] explains tractability for all previously known results?

Finally, we feel that, although we have shown that the STV-framework does not explain all tractability results for #SAT, it is still a framework that should be studied in the future. We believe that there are still many classes to be captured by it and thus we see a better understanding of the framework as an important goal for future research. One question is the complexity of computing branch decompositions of (approximately) minimal MIM-width or PS-width. Alternatively, one could try to find more classes of bipartite graphs for which one can efficiently compute branch decompositions of small MIM-width. This would then directly extend the knowledge on structural classes of CNF-formulas for which dynamic programming can efficiently solve #SAT.

References

1 G. Ausiello, A. D'Atri, and M. Moscarini. Chordality properties on graphs and minimal conceptual connections in semantic data models. *J. Comput. Syst. Sci.*, 33(2):179–202, 1986.

2 H.L. Bodlaender. A tourist guide through treewidth. *Acta Cybern.*, 11(1-2):1–21, 1993.

3 A. Brandstädt, V.B. Le, and J.P. Spinrad. *Graph Classes: A Survey.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.

4 J. Brault-Baron. Hypergraph Acyclicity Revisited. *ArXiv e-prints*, March 2014.

5 A. Bulatov, M. Dyer, L.A. Goldberg, M. Jalsenius, M. Jerrum, and D. Richerby. The complexity of weighted and unweighted #CSP. *Journal of Computer and System Sciences*, 78(2):681–688, March 2012.

6 J.-Y. Cai and X. Chen. Complexity of counting CSP with complex weights. In *Proceedings of the Forty-fourth Annual ACM Symposium on Theory of Computing*, STOC '12, page 909–920, New York, NY, USA, 2012. ACM.

7 F. Capelli, A. Durand, and S. Mengel. Hypergraph Acyclicity and Propositional Model Counting. In *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference*, pages 399–414, 2014.

8 D.A. Cohen, M.J. Green, and C. Houghton. Constraint representations and structural tractability. In *Principles and Practice of Constraint Programming - CP 2009*, pages 289–303, 2009.

9 V. Dalmau and P. Jonsson. The complexity of counting homomorphisms seen from the other side. *Theor. Comput. Sci.*, 329(1-3):315–323, 2004.

10 Reinhard Diestel. *Graph Theory, 4th Edition*, volume 173 of *Graduate texts in mathematics*. Springer, 2012.

11 D. Duris. Some characterizations of γ and β-acyclicity of hypergraphs. *Inf. Process. Lett.*, 112(16):617–620, 2012.

12 R. Fagin. Degrees of acyclicity for hypergraphs and relational database schemes. *Journal of the ACM*, 30(3):514–550, 1983.

13 E. Fischer, J.A. Makowsky, and E.V. Ravve. Counting truth assignments of formulas of bounded tree-width or clique-width. *Discrete Applied Mathematics*, 156(4):511–529, 2008.

14 G. Gottlob, N. Leone, and F. Scarcello. A comparison of structural CSP decomposition methods. *Artif. Intell.*, 124(2):243–282, 2000.

15 G. Gottlob and R. Pichler. Hypergraphs in Model Checking: Acyclicity and Hypertree-Width versus Clique-Width. *SIAM Journal on Computing*, 33(2), 2004.

16 M. Grohe and D. Marx. On tree width, bramble size, and expansion. *J. Comb. Theory, Ser. B*, 99(1):218–228, 2009.

17 S. Ordyniak, D. Paulusma, and S. Szeider. Satisfiability of acyclic and almost acyclic CNF formulas. *Theoretical Computer Science*, 481:85–99, 2013.

18 D. Paulusma, F. Slivovsky, and S. Szeider. Model Counting for CNF Formulas of Bounded Modular Treewidth. In *30th International Symposium on Theoretical Aspects of Computer Science, STACS 2013*, pages 55–66, 2013.

19 D. Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82(1–2):273 – 302, 1996.

20 S. Hortemo Sæther, J.A. Telle, and M. Vatshelle. Solving MaxSAT and #SAT on structured CNF formulas. In *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference*, pages 16–31, 2014.

21 M. Samer and S. Szeider. Algorithms for propositional model counting. *Journal of Discrete Algorithms*, 8(1):50–64, 2010.

22 F. Slivovsky and S. Szeider. Model Counting for Formulas of Bounded Clique-Width. In *Algorithms and Computation - 24th International Symposium, ISAAC 2013*, pages 677–687, 2013.

23 M. Vatshelle. *New Width Parameters of Graphs.* PhD thesis, University of Bergen, 2012.