Understanding model counting for β-acyclic CNF-formulas

Johann Brault-Baron* Florent Capelli† Stefan Mengel‡

May 26, 2014

We extend the knowledge about so-called structural restrictions of #SAT by giving a polynomial time algorithm for β-acyclic #SAT. In contrast to previous algorithms in the area, our algorithm does not proceed by dynamic programming but works along an elimination order, solving a weighted version of constraint satisfaction. Moreover, we give evidence that this deviation from more standard algorithm is not a coincidence, but that there is likely no dynamic programming algorithm of the usual style for β-acyclic #SAT.

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. [Rot96]), 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 [Rot96]. 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 to 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, there has been a push for constructing such dynamic programming algorithms for ever more general classes of graphs and hypergraphs, see e.g. [FMR08, SS10, PSS13, SS13, CDM14].

Very recently, Sæther, Telle and Vatshelle, in a striking contribution [STV14], have introduced a new width measure for CNF-formulas, that they call PS-width. Essentially, it is a measure for how much information has to be propagated from one step to the next in a natural formalization of the known dynamic programming algorithms. In our opinion, PS-width thus gives an upper bound on how far the dynamic programming techniques from the literature can be extended.

*LSV UMR 8643, ENS Cachan and Inria, France
†IMJ UMR 7586 - Logique, Université Paris Diderot, France
‡LIX UMR 7161, Ecole Polytechnique, France

Moreover, Sæther, Telle and Vatshelle have shown that if one is given a formula F and a decomposition of small PS-width, one can efficiently count the number of satisfying assignments of F. Thus they have essentially turned the construction of dynamic programming algorithms into a question of graph theory: If, for a class of formulas, one can efficiently compute decompositions that have small PS-width for all formulas having these graphs, the dynamic programming of [STV14] solves these instances. In fact, PS-width gives a uniform explanation for *all* structural tractability results for #SAT from the literature that we are aware of. On the other hand, since, in our opinion, the framework of [STV14] is a very good formalization of dynamic programming, there is likely no efficient dynamic programming algorithm for a class of CNF-formulas, if it does not have decompositions of small PS-width, or if these decompositions cannot be constructed efficiently.

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 [Fag83, Dur12], and β-acyclicity is the most general acyclicity notion discussed in the literature for which #SAT could be tractable (see the discussions in [OPS13, CDM14]). The complexity of #SAT for β-acyclic formulas 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 [OPS13] without this directly generalizing to a tractability result for #SAT. This is because the algorithm of [OPS13] 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 [GP04], 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 [OPS13]. Since decomposition techniques based on hypergraph acyclicity tend to be more general than graph-based techniques [GLS00], 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 hypergraphs is tractable. In fact, we show that a more general counting problem which we call *weighted counting for constraint satisfaction with default values*, short $\#\mathrm{CSP}_\mathrm{d}$, 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. [BDG$^+$12, CC12]. 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 [DJ04], but since we allow clauses, resp. relations, of unbounded arity, our results and those of [DJ04] are incomparable.

We note that our algorithm is in style very different 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 [OPS13]. But in contrast to using well-understood resolution techniques, we develop from scratch a procedure to update the weights of our $\#\mathrm{CSP}_\mathrm{d}$ 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 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.

Our second contribution is that we show that our tractability result is not covered by the framework of Sæther, Telle and Vatshelle [STV14], short STV-framework, which—as we show—covers all other known structural tractability results for #SAT. We do this by showing that β-acyclic #SAT-instances may have a PS-width so high that from [STV14] we cannot even get subexponential runtime bounds. 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. If one accepts the framework of [STV14] as a good formalization of dynamic programming—which we do—then the deviation from the usual dynamic programming paradigm is not a coincidence but instead due to the fact that there *is* no efficient dynamic programming algorithm in the usual style. 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 $a \oplus_y d := a \cup \{y \mapsto d\}$.

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, $\mu(c) = \mu$ its 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.*

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 to 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 have to 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.

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

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

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

The size *$\|I\|$ of a $\#\mathrm{CSP_d}$-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

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

We use the following straightforward observation throughout this paper

$$w(J, a) = \sum_{b \in D^{V' \setminus W}} \prod_{c \in J} c((a \cup b)|_{\mathsf{var}(c)}).$$

2.2. Graphs and hypergraphs associated to CNF-formulas

We use standard notation for graphs which can e.g. be found in [Die05]. 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 $\|\mathcal{H}\| = \sum_{e \in E(\mathcal{H})} |e|$.

A *subhypergraph* $\mathcal{H}'$ of a hypergraph $\mathcal{H}$ is a hypergraph such that $V(\mathcal{H}') \subseteq V(\mathcal{H})$ and $E(\mathcal{H}') \subseteq \{e \cap V(\mathcal{H}') \mid e \in E(\mathcal{H}), e \cap V(\mathcal{H}') \neq \emptyset\}$. For $S \subseteq V(\mathcal{H})$, the *induced subhypergraph* of $\mathcal{H}$ by S is the hypergraph $\mathcal{H}[S] = (S, \{e \cap S \mid e \in E(\mathcal{H})\} \setminus \{\emptyset\})$. We denote by $\mathcal{H} \setminus S$ the hypergraph $\mathcal{H}[V(\mathcal{H}) \setminus S]$. If S contains only one vertex v, we also write $\mathcal{H} \setminus v$ for $\mathcal{H} \setminus \{v\}$.

We are interested in structural restrictions of the problem $\#\mathrm{CSP_d}$. What we mean by structural restriction is that we restrict the way the variables interact in the different constraints. To formalize this notion, we introduce the hypergraph associated to an instance of $\#\mathrm{CSP_d}$: The hypergraph $\mathcal{H}(I)$ associated to $\#\mathrm{CSP_d}$-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 $\#\mathrm{CSP_d}$-instances with the same hypergraphs.

The problem SAT differs from the classical CSP framework in the way the constraints are represented. 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. [CGH09]. 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 of #SAT to $\#\mathrm{CSP_d}$.

Lemma 3. *Given a CNF-formula F one can construct in polynomial time a $\#\mathrm{CSP_d}$-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* $w(I,a) = 1$, *and otherwise* $w(I,a) = 0$, *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. [BLS99, Bra14].

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

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, $(x_1, \dots, x_n)$ is a β-elimination for $\mathcal{H}$ if x_1 is a nest point of $\mathcal{H}$ and $(x_2, \dots, x_n)$ 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 $\mathcal{H}$.*

It is easy to see that one can test β-acyclicity of a graph in polynomial time and that one can compute a β-elimination order efficiently 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 ([ADM86])**.** *A hypergraph is β-acyclic if and only if its incidence graph is chordal bipartite.*

We say that a $\#\mathrm{CSP}_\mathrm{d}$-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.

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

Proof. Taking the construction of Lemma 3, it is clear that the number of solution of F is equal to $w(I)$. The rest follows from the fact that the hypergraph remains unchanged during the reduction. □

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 $\#\mathrm{CSP}_\mathrm{d}$ to the framework of Sæther, Telle and Vatshelle [STV14]. 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 [Vat12]. For a tree T we denote by $L(T)$ the set of the leaves 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 at most degree 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 [Vat12]), 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. [Bod93]), it is more convenient for us to work with the strongly related notion of *Maximum-Matching-width* (short *MM-width*) introduced by Vatshelle [Vat12]. 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 [Vat12, p. 28].

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

Another width measure of graphs that we will use extensively is *Maximum-Induced-Matching-width* (short *MIM-width*): 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 *precisely 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 precisely 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 G 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 tractable solving of #SAT.

Theorem 8 ([STV14])**.** *Given a CNF-formula F of 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.

Sæther, Telle and Vatshelle showed the following connection between the PS-width of a CNF-formula F and the MIM-width of the incidence graph G of F.

Theorem 9 ([STV14])**.** *For any CNF-formula F over m clauses, any branch decomposition of the incidence graph $I(F)$ of F with MIM-width k has PS-width at most m^k.*

Theorem 9 and Theorem 8 essentially turn finding structural classes of tractable #SAT-instances into a problem of graph theory: it suffices to show that certain classes of formulas have sufficiently small MIM-width or PS-width to show that they are tractable. We will see that all tractability results from the literature can be explained this way. Unfortunately, deciding if a class of formulas has small MIM-width or PS-width seems to be tricky. In fact, even the complexity of deciding if a given graph has MIM-width 1 in polynomial time is left as an open problem in [Vat12].

3. The algorithm and its correctness

In this section we describe an algorithm that, given an instance I of $\#\mathrm{CSP_d}$ 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.

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 $c'_i := (f'_i, \mu)$ is the weighted constraint on variables $\mathsf{var}(c'_i) = \mathsf{var}(c) \setminus \{x\}$, 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 $a \in \mathsf{supp}(c'_i)$, let $P_i(a,d) := \prod_{j=1}^{i} c_j((a \oplus_x d)|_{\mathsf{var}(c_j)})$ and $P_0(a,d) = 1$. We define:*

$$f'_i(a) := \frac{\sum_{d \in D} P_i(a,d)}{\sum_{d \in D} P_{i-1}(a,d)}$$

if $\sum_{d\in D} P_{i-1}(a,d) \neq 0$ *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. First, we explain why I' is well-defined. As x is a nest point, we can write $I(x) = \{c_1, \ldots, c_m\}$ with $\mathsf{var}(c_1) \subseteq \ldots \subseteq \mathsf{var}(c_m)$. If two constraints have the same variables, we choose an arbitrary order for them. Note that in Section 4 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. Finally, remark that $P_i(a,d)$ is well defined since for $a \in \mathsf{supp}(c'_i)$, $d \in D$ and $j \leq i$, $(a \oplus_x d)$ assigns all variables of c_j since $\mathsf{var}(c_j) \subseteq \mathsf{var}(c_i)$. Thus writing $c_j((a \oplus_x d)|_{\mathsf{var}(c_j)})$ is correct. We insist on the fact that it is only because x is a nest point that this definition works.

$\mathcal{H}(I') = \mathcal{H}(I) \setminus x$ is obvious because for a constraint in I with variable set V, there exists a constraint in I' with variable set $V \setminus \{x\}$.

$\|I'\| \leq \|I\|$ because for all $c \in I$, $|c'| \leq |c|$ since $|\{a \in D^{\mathsf{var}(c')} \mid \exists d \in D, (a \oplus_x d) \in \mathsf{supp}(c)\}| \leq |\mathsf{supp}(c)|$.

We now show by induction on i that for all $a \in D^{\mathsf{var}(c'_i)}$,

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

For $i = 1$, let $a \in D^{\mathsf{var}(c'_1)}$. If $a \in \mathsf{supp}(c'_1)$, then by definition:

$$c'_1(a) = \frac{\sum_{d\in D} P_1(a,d)}{\sum_{d\in D} P_0(a,d)}.$$

Since $P_0(a,d) = 1$ for all d, we have the expected result.

If $a \notin \mathsf{supp}(c'_1)$, then for all d, $a \oplus_x d \notin \mathsf{supp}(c_1)$. Thus $P_1(a,d) = \mu_1$ for all d and finally

$$\sum_{d\in D} P_1(a,d) = |D|\mu_1 = |D|c'_1(a).$$

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

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

First, assume that $\sum_{d\in D} P_i(a,d) = 0$. Since this sum is a sum of positive rationals, we have that for all d, $P_i(a,d) = 0$. Thus, $P_{i+1}(a,d) = 0$ for all d, that is $\sum_{d\in D} P_{i+1}(a,d) = 0$ which confirm the induction hypothesis.

Now assume that $\sum_{d\in D} P_i(a,d) \neq 0$. If $a \in \mathsf{supp}(c'_{i+1})$, by definition of c'_{i+1}, the induction hypothesis trivially holds.

If $a \notin \mathsf{supp}(c_{i+1})$, we have $P_{i+1}(a,d) = \mu_{i+1}P_i(a,d)$ for all d. Thus $\sum_{d\in D} P_{i+1}(a,d) = \mu_i \sum_{d\in D} P_i(a,d) = c'_{i+1}(a) \sum_{d\in D} P_i(a,d)$ which establish the induction hypothesis for $i+1$.

Applying the result for $i = p$, we find:

$$|D| \prod_{c\in I(x)} c'(a) = \sum_{d\in D} \prod_{c\in I(x)} c((a \oplus_x d)|_{\mathsf{var}(c)})$$

Now, it is sufficient to remark that for $c \notin I(x)$, for all $d \in D$, $c((a \oplus_x d)|_{\mathsf{var}(c)}) = c(a|_{\mathsf{var}(c)}) = c'(a|_{\mathsf{var}(c)})$ since $x \notin \mathsf{var}(c)$ and $c = c'$. Thus:

$$|D|w(I') = \sum_{a \in D^{\mathsf{var}(I)\setminus\{x\}}} \sum_{d \in D} \prod_{c \in I'} c((a \oplus_x d)|_{\mathsf{var}(c)}) = w(I).$$

We now analyze the number of arithmetic operations we make in the construction of I'. Clearly, if we have computed the $\sum_{d \in D} P_i(a,d)$ 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. Now remark that if we have computed $P_i(a,d)$, then we only need one more multiplication to compute $P_{i+1}(a,d)$.

Now, we prove by induction on i that $P_i(a,d)$ could take at least $1 + \sum_{j=1}^{i} |c_j|$ different values. It is trivial for $i = 0$. Now remark that if $a \oplus_x d \notin \mathsf{supp}(c_i)$, then $P_i(a,d) = \mu_i P_{i-1}(a,d)$, thus by induction, it gives $1 + \sum_{j=1}^{i-1} |c_j|$ different values for P_i. And there is at most $|\mathsf{supp}(c_i)| \leq |c_i|$ other values for $a \oplus_x d \in \mathsf{supp}(c_i)$, which prove the induction.

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

Theorem 11. *If I is a β-acyclic instance of $\#\mathrm{CSP}_\mathrm{d}$, we can compute $w(I)$ with a $O(s(I)^2\|I\|)$ arithmetic operations.*

Proof. We iterate the algorithm of Theorem 10 on a β-elimination order of the variables of I to transform it into an instance I^*. After all variables are eliminated, every constraint of I^* has an empty set of variables, thus $w(I^*) = \prod_{c \in I^*} c(\epsilon)$, where ϵ denotes the empty assignment. Moreover, by Theorem 10, $w(I) = |D|^{|\mathsf{var}(I)|} w(I^*)$. Thus $w(I)$ can be computed with $O(s(I))$ additionnal multiplications.

If we denote by $p_x = |\{c \in I \mid x \in \mathsf{var}(c)\}|$, we have a total complexity of $\sum_{x \in \mathsf{var}(I)} O(p_x \|I(x)\|)$, that is $O((\sum_{x \in \mathsf{var}(I)} p_x)|\mathsf{var}(I)|\|I\|)$. It is easy to see that $\sum_{x \in \mathsf{var}(I)} p_x = s(I)$ and since $|\mathsf{var}(I)| \leq s(I)$, we have a total number of arithmetic operations that is a $O(s(I)^2\|I\|)$. □

4. Runtime analysis of the algorithm

The analysis of Theorem 11 shows that our algorithm uses only a polynomial number of arithmetic operations. Unfortunately, this does not guarantee that the algorithm runs in polynomial time on a RAM. The problem is that, due to the many multiplications and divisions, the bitsize of the new (rational) weights computed by the algorithm at each step could grow exponentially, leading to an overall superpolynomial runtime. In this section we will prove that this is in fact not the case. We will show that at each step of the algorithm, numerous cancellations occur, leading to weights of polynomial bitsize. Combining this with Theorem 11, it will follow that the algorithm runs in polynomial time.

4.1. Some technical lemmas

In this section, we will show some rather technical lemmas we will use later on. Throughout this paper, we follow the convention that for all assignment a, we have $w(\emptyset, a) = 1$. This is motivated by the following lemma.

Lemma 12. *Let I be a set of weighted constraints, $J \subseteq I$ and a a partial assignment of $\mathsf{var}(I)$. If $w(J,a) = 0$ then $w(I,a) = 0$.*

Proof. We have $w(J,a) = 0 = \sum_{b \simeq a} \prod_{c \in J} c(b|_{\mathsf{var}(c)})$. Since every term of the sum is non-negative, we have that for all $b \simeq a$ it holds $\prod_{c \in J} c(b|_{\mathsf{var}(c)}) = 0$. Thus,

$$w(I,a) = \sum_{b \simeq a} \prod_{c \in J} c(b|_{\mathsf{var}(c)}) \prod_{c \in I \setminus J} c(b|_{\mathsf{var}(c)}) = 0.$$

□

One key ingredient in our analysis will be understanding how two subinstances interact under a partial assignment.

Lemma 13. *Let I be a set of weighted constraints on domain D, $J_1 \subseteq I$, $J_2 \subseteq I$ and $a \in D^W$ for $W \subseteq \mathsf{var}(I)$. Let $V_1 = \mathsf{var}(J_1)$, $V_2 = \mathsf{var}(J_2)$. If $V_1 \cap V_2 \subseteq W$ and $J_1 \cap J_2 = \emptyset$, then*

$$w(J_1 \cup J_2, a) = w(J_1, a) w(J_2, a)$$

Proof. Let $V = V_1 \cup V_2$. Since $V \setminus W = (V_1 \setminus W) \cup (V_2 \setminus W)$ and this union is disjoint by definition, there is a natural bijection between $D^{V_1 \setminus W} \times D^{V_2 \setminus W}$ and $D^{V \setminus W}$ that associates to (b_1, b_2) the assignment $b_1 \cup b_2$. Moreover, if $c \in J_1$, then $(b_1 \cup b_2)|_{\mathsf{var}(c)} = b_1|_{\mathsf{var}(c)}$ since $\mathsf{var}(c) \subseteq V_1$. Similarly, for $c \in J_2$, $(b_1 \cup b_2)|_{\mathsf{var}(c)} = b_2|_{\mathsf{var}(c)}$. Consequently,

$$\begin{aligned} w(J_1 \cup J_2, a) &= \sum_{b_1 \in D^{V_1 \setminus W}} \sum_{b_2 \in D^{V_2 \setminus W}} \prod_{c \in J_1} c((a \cup b_1)|_{\mathsf{var}(c)}) \prod_{c \in J_2} c((a \cup b_2)|_{\mathsf{var}(c)}) \\ &= w(J_1, a) w(J_2, a) \end{aligned}$$

□

Corollary 14. *Let I be a set of weighted constraints on domain D, $J_1 \subseteq I$, $J_2 \subseteq I$ and $a \in D^W$ for $W \subseteq \mathsf{var}(I)$. Let $V_1 = \mathsf{var}(J_1 \setminus J_2)$, $V_2 = \mathsf{var}(J_2 \setminus J_1)$ and $V_0 = \mathsf{var}(J_1 \cap J_2)$. If $V_0 \cap V_1 \subseteq W$ and $V_0 \cap V_2 \subseteq W$. If $w(J_2, a) \neq 0$, we have:*

$$\frac{w(J_1, a)}{w(J_2, a)} = \frac{w(J_1 \setminus J_2, a)}{w(J_2 \setminus J_1, a)}$$

Proof. First, remark that $w(J_2 \setminus J_1, a) \neq 0$ by Lemma 12 since $J_2 \setminus J_1 \subseteq J_2$ and $w(J_2, a) \neq 0$.

Apply Lemma 13 on $J_1 \setminus J_2$ and $J_1 \cap J_2$ for the numerator and on $J_2 \setminus J_1$ and $J_2 \cap J_1$ for the denominator and observe that $w(J_1 \cap J_2, a)$ cancels. □

We will use the following corollary heavily in Section 4.

Corollary 15. *Let I be a set of weighted constraints on domain D, $J_1, J_2, J_3, J_4 \subseteq I$ and $a \in D^W$ for $W \subseteq \mathsf{var}(I)$. Assume that $w(J_3, a) \neq 0$ and $w(J_4, a) \neq 0$ and*

(i) $J_1 \cap J_2 \subseteq J_3$ *and* $J_3 \cap J_4 \subseteq J_1$,

(ii) $\mathsf{var}(J_1 \setminus J_3) \cap \mathsf{var}(J_1 \cap J_3) \subseteq W$,

(iii) $\mathsf{var}(J_3 \setminus J_1) \cap \mathsf{var}(J_1 \cap J_3) \subseteq W$,

(iv) $\mathsf{var}(J_1 \setminus J_3) \cap \mathsf{var}(J_2) \subseteq W$, *and*

(v) $\mathsf{var}(J_3 \setminus J_1) \cap \mathsf{var}(J_4) \subseteq W$.

Then

$$\frac{w(J_1,a)w(J_2,a)}{w(J_3,a)w(J_4,a)} = \frac{w((J_1 \setminus J_3) \cup J_2, a)}{w((J_3 \setminus J_1) \cup J_4, a)}.$$

Proof. Apply Corollary 14 on J_1 and J_3 and Lemma 13 on $J_1 \setminus J_3$ and J_2 for the numerator and on $J_3 \setminus J_1$ and J_4 for the denominator. Remark that Condition (i) ensures that $(J_1 \setminus J_3) \cap J_2 = \emptyset$ and $(J_3 \setminus J_1) \cap J_4 = \emptyset$ and that the denominator is not null because $w((J_3 \setminus J_1) \cup J_4, a) = w(J_3 \setminus J_1, a)w(J_4, a)$ and $w(J_4, a) \neq 0$ by assumption and $w(J_3 \setminus J_1, a) \neq 0$ by Lemma 12 and $w(J_3, a) \neq 0$. □

4.2. Defining partial orders

The algorithm of Theorem 10 transforms an instance into a new one with the same number of constraints but with one variable less. In this section we will give an explicit description of the weight of a constraint $c \in I$ after k such elimination steps. In the following, I is a β-acyclic CSP-instance and $\{x_1, \ldots, x_n\} = \mathsf{var}(I)$ is a β-elimination order of $\mathcal{H}(I)$. We assume that we will perform the elimination along this order. Let $X_k = \{x_1, \ldots, x_k\}$ and for $c \in I$, we denote by $c^{(k)}$ the constraint c after the elimination of x_k. By convention, $c^{(0)} = c$. Remark that $\mathsf{var}(c^{(k)}) = \mathsf{var}(c) \setminus X_k$.

In the following, we will introduce for each k, a partial order $\prec_k$ on I. The intuition for this partial order is that for $c, d \in I$, $c \prec_k d$ means that $d^{(k)}$ "depends on" $c^{(0)}$. For example, assume that $x_1 \in \mathsf{var}(c) \subseteq \mathsf{var}(d)$. When we eliminate x_1, we see—in the formula of Theorem 10—that the weight of c appears in the definition of $d^{(1)}$. Hence, we would like to have $c \prec_1 d$.

To simplify the proofs, we make one more assumption on I: If $c, d \in I$ and $c \neq d$, then $\mathsf{var}(c) \neq \mathsf{var}(d)$. We may assume this w.l.o.g. since it is easy to merge two constraints with the same variables without increasing $\|I\|$. Observe that we make this assumption only on the initial instance I. During the elimination process, constraints with the same set of variables might appear, but we can easily handle them.

Definition 16. *For two constraints $c, d \in I$, we write $c \prec d$ if there exists k such that $\mathsf{var}(c) \setminus X_k \subsetneq \mathsf{var}(d) \setminus X_k$. We write $c \preceq d$ if $c \prec d$ or $c = d$.*

Lemma 17. *$\preceq$ is a total order on I.*

Proof. We first show that $\preceq$ is antisymmetric. So let c, d be constraints such that $c \preceq d$ and $d \preceq c$. By way of contradiction, assume that $c \neq d$, so $c \prec d$ and $d \prec c$. By definition there are k, k' such that $\mathsf{var}(c) \setminus X_k \subsetneq \mathsf{var}(d) \setminus X_k$ and $\mathsf{var}(d) \setminus X_{k'} \subsetneq \mathsf{var}(c) \setminus X_{k'}$. W.l.o.g. assume that $k < k'$. Then $\mathsf{var}(c) \setminus X_\ell \subseteq \mathsf{var}(d) \setminus X_\ell$ for all $\ell \geq k$ which is a contradiction to $d \prec c$. It follows that $\preceq$ is antisymmetric.

We now show transitivity of $\preceq$. So let $c, d, e \in I$ with $c \preceq d$ and $d \preceq e$. If we have $c = d$ or $d = e$, then we get immediately $c \preceq d$. Thus we may assume that $c \prec d$ and $d \prec e$. By definition, there exist k, ℓ such that $\mathsf{var}(c) \setminus X_k \subsetneq \mathsf{var}(d) \setminus X_k$ and $\mathsf{var}(d) \setminus X_\ell \subsetneq \mathsf{var}(e) \setminus X_\ell$. For $m := \max(k, \ell)$ we get $\mathsf{var}(c) \setminus X_m \subseteq \mathsf{var}(d) \setminus X_m \subseteq \mathsf{var}(e) \setminus X_m$ and one of these inclusions is strict. Thus $\mathsf{var}(c) \setminus X_m \subsetneq \mathsf{var}(e) \setminus X_m$, that is $c \prec e$ and it follows that $\preceq$ is transitive.

We now show that $\preceq$ is total. So let $c, d \in I$. If $c = d$, then by definition $c \preceq d$. So we assume that $c \neq d$. Let $k = \max\{j \mid x_j (\in \mathsf{var}(c) \setminus \mathsf{var}(d)) \cup (\mathsf{var}(d) \setminus \mathsf{var}(c))\}$. Observe that k is well-defined, since $\mathsf{var}(d) \neq \mathsf{var}(c)$ by assumption on I. Assume first that $x_k \in \mathsf{var}(d) \setminus \mathsf{var}(c)$. Then $\mathsf{var}(c) \setminus \mathsf{var}(d) \subseteq X_{k-1}$ by maximality of k. It follows that $\mathsf{var}(c) \setminus X_{k-1} \subseteq \mathsf{var}(d) \setminus X_{k-1}$ and since $x_k \in \mathsf{var}(d) \setminus X_{k-1}$, we have $\mathsf{var}(c) \setminus X_{k-1} \subsetneq \mathsf{var}(d) \setminus X_{k-1}$. Thus $c \prec d$. Analogously, we get for $x_k \in \mathsf{var}(c) \setminus \mathsf{var}(d)$ that $d \prec c$. Hence $\prec$ is total. □

Definition 18. *For $k \in \{0, \dots, n\}$, we define the relation $\prec_k \subseteq I \times I$ inductively on k as*

- $\prec_0 = \emptyset$
- *for all $c, d \in I$, $c \prec_{k+1} d$ if and only if $c \prec_k d$ or there exists $e \in I$ such that $c \preceq_k e \prec d$ and $x_{k+1} \in \mathsf{var}(d) \cap \mathsf{var}(e)$,*

where we denote by $c \preceq_k d$ if $c = d$ or $c \prec_k d$.

Observe that the definition of $\prec_k$ is compatible with the informal discussion of $c \prec_k d$ at the beginning of this section: If $d^{(k)}$ depends on $c^{(k)}$. For $k = 0$, no constraint depends on another, thus $\prec_0 = \emptyset$. Then, when eliminating x_{k+1}, if $x_{k+1} \notin \mathsf{var}(d)$, then the dependencies of d do not change since d remains the same. But if $x_{k+1} \in \mathsf{var}(d)$, then the weight of each constraint e whose variables are included in $\mathsf{var}(d)$ and $x_{k+1} \in \mathsf{var}(e)$ will appear in $d^{(k+1)}$. And if e depends on c at step k, that is $c \prec_k e$, then d will also depend on c after the elimination of x_{k+1}.

We now show some properties of $\prec_k$ that are crucial for the understanding of how the weights of constraints interact with each other.

Lemma 19. *a)* $(\prec_k) \subseteq (\prec_{k+1})$.

b) For all $c, d \in I$, $c \prec_{k+1} d$ implies $c \prec d$ and $\mathsf{var}(c) \setminus X_k \subseteq \mathsf{var}(d) \setminus X_k$.

c) $(\preceq_k)$ *is a partial order.*

Proof. a) follows directly from the definition of $\prec_{k+1}$.

We prove b) by induction on k. For $k = 0$, let $c, d \in I$ such that $c \prec_1 d$. Since $\prec_0 = \emptyset$, $c \not\prec_0 d$. Thus, by definition, there exists e such that: $c \preceq_0 e \prec d$ and $x_1 \in \mathsf{var}(c) \cap \mathsf{var}(d)$. Again, since $\prec_0 = \emptyset$, we have $c = e$. Thus $c \prec d$ and since x_1 is a nest point, $\mathsf{var}(c) \subseteq \mathsf{var}(d)$, which is the induction hypothesis for $k = 0$ since $X_0 = \emptyset$.

Now assume that $k \geq 0$ and that the statement is true for k. Let $c, d \in I$ such that $c \prec_{k+1} d$. If $c \prec_k d$, then we get from the induction hypothesis that $c \prec d$ and $\mathsf{var}(c) \setminus X_{k-1} \subseteq \mathsf{var}(d) \setminus X_{k-1}$. This directly yields $\mathsf{var}(c) \setminus X_k \subseteq \mathsf{var}(d) \setminus X_k$. Now, if $c \not\prec_k d$, then there exists e such that $c \preceq_k e$, $e \prec d$ and $x_{k+1} \in \mathsf{var}(e) \cap \mathsf{var}(d)$. By induction $c \preceq e$ and thus $c \prec d$ since $\prec$ is transitive by Lemma 17. As x_{k+1} is a nest point after eliminating X_k, we have $\mathsf{var}(e) \setminus X_k \subseteq \mathsf{var}(d) \setminus X_k$. By induction we get $\mathsf{var}(c) \setminus X_k \subseteq \mathsf{var}(e) \setminus X_k$ and thus $\mathsf{var}(c) \setminus X_k \subseteq \mathsf{var}(d) \setminus X_k$ as desired.

For c), observe that $\preceq_k$ reflexive by definition. Furthermore, $\preceq_k$ is antisymmetric since it is a subrelation of the order $\prec$ by b). It remains to show that $\prec_k$ is transitive. We do this by induction on k. The case $k = 0$ is trivial since $(\prec_0) = \emptyset$. Now suppose that $(\prec_k)$ is transitive for $k \geq 0$. Let $c, d, e \in I$ such that $c \prec_{k+1} d$ and $d \prec_{k+1} e$. If $c \prec_k d \prec_k e$, then by induction $c \prec_k e$ and then $c \prec_{k+1} e$ since $(\prec_k) \subseteq (\prec_{k+1})$.

Now assume that $c \not\prec_k d$. Then by definition, there exists c' such that $c \preceq_k c' \prec d$ and $x_{k+1} \in \mathsf{var}(c') \cap \mathsf{var}(d)$. Since $d \prec_{k+1} e$, we also have $c' \prec e$ and $x_{k+1} \in \mathsf{var}(d) \setminus X_k \subseteq \mathsf{var}(e) \setminus X_k$. Thus $x_{k+1} \in \mathsf{var}(c') \cap \mathsf{var}(e)$ and $c \preceq_k c' \prec e$, that is $c \prec_{k+1} e$.

Finally assume that $c \prec_k d$ and $d \not\prec_k e$. Since $d \prec_{k+1} e$, there exists d' such that $d \preceq_k d' \prec e$ and $x_{k+1} \in \mathsf{var}(d') \cap \mathsf{var}(e)$. By induction, $(\prec_k)$ is transitive. Thus $c \prec_k d' \prec e$ and $x_{k+1} \in \mathsf{var}(d') \cap \mathsf{var}(e)$. That is $c \prec_{k+1} e$. □

Again, from our intuitive understanding of $\prec_k$, the transitivity is obvious: if $d^{(k)}$ depends on $c^{(k)}$ and $e^{(k)}$ depends on $d^{(k)}$, then $e^{(k)}$ should depend on $c^{(k)}$. An other informal observation is that if c and d have no common dependencies at step k, then they should not share a variable in X_k since sharing a nest point automatically induces a dependency:

Lemma 20. *For all $c, d \in I$, if $c \prec d$ but $c \not\prec_k d$, then $\mathsf{var}(c) \cap \mathsf{var}(d) \cap X_k = \emptyset$.*

Proof. By way of contradiction. If for $j \leq k$, $x_j \in \mathsf{var}(c) \cap \mathsf{var}(d) \cap X_k$ then $c \preceq_j c \prec d$ and $x_j \in \mathsf{var}(c) \cap \mathsf{var}(d)$. That is $c \prec_j d$ and by Lemma 19 we get $c \prec_k d$. □

We need one final property: if $d \prec e$ both depend on c at step k, then these dependencies were induced by the elimination of at most two nest points. During the elimination of the second nest point, e will get both the dependencies of c but also the dependencies of d. Thus e should depend on d. This is formalized by the following lemma:

Lemma 21. *Let $c, d, e \in I$. If $c \prec_k d$, $c \prec_k e$ and $d \prec e$ then $d \prec_k e$.*

Proof. The proof is by induction on k. The case $k = 0$ is trivial since the precondition cannot hold. Assume the result holds for $k \geq 0$ and let $c, d, e \in I$ be constraints such that $c \prec_{k+1} d$, $c \prec_{k+1} e$ and $d \prec e$. If both $c \prec_k d$ and $c \prec_k e$, then the induction gives $d \prec_k e$ thus $d \prec_{k+1} e$.

Otherwise, assume that $c \not\prec_k d$ and $c \not\prec_k e$. Then by definition $x_{k+1} \in \mathsf{var}(d) \cap \mathsf{var}(e)$. Since $d \prec e$, it gives $d \prec_{k+1} e$.

Now assume $c \not\prec_k d$ but $c \prec_k e$. By definition, there exists c' such that $c \preceq_k c' \prec d$ and $x_{k+1} \in \mathsf{var}(c') \cap \mathsf{var}(d)$. Since $c' \prec d \prec e$, we have $c' \prec e$ and by induction $c \prec_k c'$ and $c \prec_k e$ gives $c' \prec_k e$. Thus $x_{k+1} \in \mathsf{var}(e)$ and $d \prec_{k+1} e$.

Finally assume that $c \prec_k d$ but $c \not\prec_k e$. By definition, there exists c' such that $c \preceq_k c' \prec e$ and $x_{k+1} \in \mathsf{var}(c') \cap \mathsf{var}(e)$. As in the previous case, by induction, we can deduce that $d \preceq_k c'$ or $c' \prec_k d$. Both cases lead to $d \prec_{k+1} e$. □

We now define for every k and every constraint c a subinstance $I_k(c)$ of I that intuitively contains the relations of I that have an influence on the weights of c after the first k variables have been eliminated.

Definition 22. *For every $k \in \{0, \dots, n\}$ and $c \in I$ we define $I_k(c) := \{d \in I \mid d \preceq_k c\}$.*

We will now prove a lemma that helps us understand how $I_k(c)$ is evolving during the algorithm. Again, the behaviour is intuitively very natural: If $x_{k+1} \notin \mathsf{var}(c)$, then c will have no new dependencies, thus $I_{k+1}(c) = I_k(c)$. If $x_{k+1} \in \mathsf{var}(c)$ however, c will take all the dependencies of the constraints d such that $x_{k+1} \in \mathsf{var}(d)$ and $d \prec c$.

Lemma 23. *For $k \leq 0$, if $x_{k+1} \notin \mathsf{var}(c)$ then $I_{k+1}(c) = I_k(c)$. Otherwise, let $I(x_{k+1}) := \{c_1, \dots, c_m\}$ with $c_1 \prec \dots \prec c_m$. Then we have*

$$I_{k+1}(c_1) = I_k(c_1)$$

and for $i < m$

$$I_{k+1}(c_{i+1}) = I_k(c_{i+1}) \cup I_{k+1}(c_i).$$

Proof. First, assume that $x_{k+1} \notin \mathsf{var}(c)$. Since $(\prec_k) \subseteq (\prec_{k+1})$ by Lemma 19, it follows that $I_k(c) \subseteq I_{k+1}(c)$. Now, if $d \in I_{k+1}(c)$ and $d \neq c$, then either $d \prec_k c$ or there exists e such that $d \preceq_k e \prec c$ and $x_{k+1} \in \mathsf{var}(c) \cap \mathsf{var}(e)$. Since $x_{k+1} \notin \mathsf{var}(c)$, we necessarily have $d \prec_k c$, that is $d \in I_k(c)$. This implies $I_k(c) = I_{k+1}(c)$.

For the second equality, $I_k(c_1) \subseteq I_{k+1}(c_1)$ still follows from Lemma 19. For the other direction, consider $d \in I_{k+1}(c_1)$, that is $d \preceq_{k+1} c_1$. By way of contradiction, assume that $d \not\preceq_k c$. By definition of $\prec_{k+1}$, there exists $e \in I$ such that $d \preceq_k e \prec c_1$ and $x_{k+1} \in \mathsf{var}(e)$. However, by definition, c_1 is the minimal constraint with respect to $\preceq$ whose variables contain x_{k+1}. Thus such an $e \in I$ cannot exist. Consequently, $d \in I_k(c_1)$ and it follows $I_{k+1}(c_1) = I_k(c_1)$.

Now fix $i < m$. By definition of c_{i+1}, we have $x_{k+1} \in \mathsf{var}(c_{i+1})$. We first prove that $I_k(c_{i+1}) \cup I_{k+1}(c_i) \subseteq I_{k+1}(c_{i+1})$. By Lemma 19 again, $I_k(c_{i+1}) \subseteq I_{k+1}(c_{i+1})$. Now let $d \in I_{k+1}(c_i)$. We have $c_i \preceq_k c_i \prec c_{i+1}$ and $x_{k+1} \in \mathsf{var}(c_i) \cap \mathsf{var}(c_{i+1})$ and thus, by definition of $\prec_{k+1}$ this implies $c_i \prec_{k+1} c_{i+1}$. This yields $d \preceq_{k+1} c_i \prec_{k+1} c_{i+1}$ and thus $d \in I_{k+1}(c_{i+1})$.

Finally, we prove that $I_{k+1}(c_{i+1}) \subseteq I_k(c_{i+1}) \cup I_{k+1}(c_i)$. So let $d \in I_{k+1}(c_{i+1})$. If $d \preceq_k c_{i+1}$, then, by definition, we have $d \in I_k(c_{i+1})$. So assume now that $d \not\preceq_k c_{i+1}$. By definition of $\prec_{k+1}$, there exists e such that $d \preceq_k e \prec c_{i+1}$ and $x_{k+1} \in \mathsf{var}(e) \cap \mathsf{var}(c_{i+1})$. Since $x_{k+1} \in \mathsf{var}(e)$ and $e \prec c_{i+1}$, it follows that $e = c_j$ for a $j < i+1$. If $j = i$, then $d \preceq_k e = c_i$ and thus $d \preceq_{k+1} c_i$ which implies $d \in I_{k+1}(c_i)$. Otherwise, $j < i$ and we have $d \preceq_k c_j \prec c_i$ and $x_{k+1} \in \mathsf{var}(c_j) \cap \mathsf{var}(c_i)$, which gives $d \prec_{k+1} c_i$. Thus $d \in I_{k+1}(c_i)$ as well. □

4.3. Proof of the runtime bound

In this section, we will prove that for each $c \in I$ and a an assignment of $\mathsf{var}(c^{(k)})$, $c^{(k)}(a)$ is proportional to

$$\frac{w(I_k(c), a)}{w(I_k(c) \setminus \{c\}, a)}.$$

Since $I_k(c)$ is a subinstance of I, the bitsize of the computed rational number is polynomial in the size of the input. Thus, it will follow that the weight of $c^{(k)}$ is a rational number of polynomial bitsize and thus all arithmetic operations of the algorithm can be done in polynomial time.

Remember that by convention $w(\emptyset, a) = 1$ and that for $x \in \mathsf{var}(I)$, $I(x) = \{c \in I \mid x \in \mathsf{var}(c)\}$.

Lemma 24. *Let $k \geq 0$ and $I(x_{k+1}) = \{c_1, \ldots, c_m\}$ with $c_1 \prec \ldots \prec c_m$. For all $j \leq m$ and $a : \mathsf{var}(c_j) \setminus X_k \to D$ we have*

$$\prod_{i=1}^{j} \frac{w(I_k(c_i), a)}{w(I_k(c_i) \setminus \{c_i\}, a)} = \frac{w(I_{k+1}(c_j), a)}{w(I_{k+1}(c_j) \setminus \{c_1, \ldots, c_j\}, a)}.$$

Proof. The proof is by induction on j. For $j = 1$, it is a consequence of Lemma 23 since $I_{k+1}(c_1) = I_k(c_1)$. Assume the result holds for $j \geq 1$. Fix $a : \mathsf{var}(c_{j+1}) \setminus X_k \to D$. Observe first that by Lemma 19 we have $\mathsf{var}(c_i) \setminus X_k \subseteq \mathsf{var}(c_{j+1}) \setminus X_k$ for $i \leq j$ (this could alternatively be seen from the fact that x_{k+1} is a nest point after removing $x_1, \ldots, x_k$). Thus we can use induction for a and get

$$\prod_{i=1}^{j+1} \frac{w(I_k(c_i), a)}{w(I_k(c_i) \setminus \{c_i\}, a)} = \frac{w(I_{k+1}(c_j), a)}{w(I_{k+1}(c_j) \setminus \{c_1, \ldots, c_j\}, a)} \frac{w(I_k(c_{j+1}), a)}{w(I_k(c_{j+1}) \setminus \{c_{j+1}\}, a)}.$$

We will apply Corollary 15 with $J_1 := I_{k+1}(c_j)$, $J_2 := I_k(c_{j+1})$, $J_3 := I_k(c_{j+1}) \setminus \{c_{j+1}\}$ and $J_4 := I_{k+1}(c_j) \setminus \{c_1, \ldots, c_j\}$ and $W := \mathsf{var}(c_{j+1}) \setminus X_k$.

Observe that by Lemma 23 and by the fact that $J_3 \subseteq J_2$, $(J_1 \setminus J_3) \cup J_2 = J_1 \cup J_2 = I_{k+1}(c_{j+1})$. Moreover, $(J_3 \setminus J_1) \cup J_4 = (J_3 \setminus J_1) \cup (J_1 \setminus \{c_1, \ldots, c_j\}) = (J_1 \cup J_3) \setminus \{c_1, \ldots, c_j\} = I_{k+1}(c_{j+1}) \setminus \{c_1, \ldots, c_{j+1}\}$ since $c_{j+1} \notin J_1$. Hence, if the conditions of Corollary 15 are met, the lemma will follow.

We now verify each conditions of Corollary 15:

(i) if $c \in J_1 \cap J_2$, then $c \preceq_{k+1} c_{j+1}$ (it is in J_2) and $c \neq c_{j+1}$ since $c \preceq c_j \prec c_{j+1}$. Thus $c \in J_3$. Moreover $J_4 \subseteq J_1$, thus $J_3 \cap J_4 \subseteq J_1$.

(ii) since $J_3 \subseteq J_2$, this condition is implied by condition (iv).

(iii) Let $c \in J_3 \setminus J_1$ and $d \in J_1$. Since both $c \in J_3$ and $d \in J_1$, we have $c \prec_k c_{j+1}$ and $d \prec_{k+1} c_j \prec_{k+1} c_{j+1}$. By Lemma 19, $\mathsf{var}(c) \setminus X_k \subseteq W$ and $\mathsf{var}(d) \setminus X_k \subseteq W$.

We claim that c and d are incomparable with respect to $\prec_k$.

First, if $c \prec_k d$, then $c \prec_{k+1} d \prec_{k+1} c_j$ that is $c \in J_1$ which is a contradiction. Consequently, $c \not\prec_k d$.

Now, if $d \prec_k c$, then $d \prec_{k+1} c$ and $d \prec_{k+1} c_j$. Thus, since $c \not\preceq_{k+1} c_j$, we have $c_j \prec_{k+1} c \prec_{k+1} c_{j+1}$ thus $c_j \prec c \prec c_{j+1}$. We have that $x_{k+1} \in \mathsf{var}(c_j)$, and by Lemma 19 b) we get $x_{k+1} \in \mathsf{var}(c)$. But this contradicts the definition of c_j as the maximal constraint with respect to $\preceq$ that is less than c_{j+1} and holds x_{k+1}. Hence this is a contradiction and we get $d \not\prec c$.

Thus, c and d are indeed incomparable with respect to $\prec_k$. Since $\prec$ is a total order we have either $d \prec c$ or $c \prec d$ and thus by Lemma 20 we have $\mathsf{var}(c) \cap \mathsf{var}(d) \cap X_k = \emptyset$. Since by Lemma 19 b) we have that $\mathsf{var}(c) \setminus X_k \subseteq \mathsf{var}(c_{j+1})$ and $\mathsf{var}(d) \setminus X_k \subseteq \mathsf{var}(c_{j+1})$, it follows that $\mathsf{var}(c) \cap \mathsf{var}(d) \subseteq W$. Since this is true for all combinations of c and d, it follows that $\mathsf{var}(J_3 \setminus J_1) \cap \mathsf{var}(J_1 \cap J_3) \subseteq W$ as desired.

(iv) let $c \in J_1 \setminus J_3$ and $d \in J_2$. We have $c \prec_{k+1} c_j$ and $d \preceq_k c_{j+1}$. By Lemma 19, $\mathsf{var}(c) \setminus X_k \subseteq \mathsf{var}(c_j) \setminus X_k \subseteq W$ and $\mathsf{var}(d) \setminus X_k \subseteq W$.

We again show that c and d are incomparable with respect to $\prec_k$.

If $c \prec_k d$, we get with $d \preceq_k c_{j+1}$ and transitivity $c \prec_k c_{j+1}$. Thus $c \in J_3$ which is a contradiction. Consequently, $c \not\prec_k d$.

Now assume that $d \prec_k c$. We have $c \prec c_j \prec c_{j+1}$ and $d \preceq_k c_{j+1}$ and thus with Lemma 21 we get $c \prec_k c_{j+1}$. But then $c \in J_3$ which is a contradiction again.

Thus c and d are indeed incomparable with respect to $\prec_k$. Now the claim follows as in (iii).

(v) since $J_4 \subseteq J_1$, this is implied by our proof of condition (iii) (we have not assumed $d \in J_3$ there).

□

We can now state the main theorem of this section. Remember that $c^{(k)}$ is the weighted constraint we get from c after k steps of our elimination procedure.

Theorem 25. *For all $c \in I$ and $k \geq 0$, there exists $\alpha_k(c) \in \mathbb{N} \setminus \{0\}$ such that for all $a : \mathsf{var}(c) \setminus X_k \to D$, either*

$$c^{(k)}(a) = 0$$

or

$$c^{(k)}(a) = \frac{1}{\alpha_k(c)} \cdot \frac{w(I_k(c), a)}{w(I_k(c) \setminus \{c\}, a)}$$

and $\alpha_k(c) \leq |D|^k$.

Proof. The proof is by induction on k. Note that $\prec_0 = \emptyset$ by definition and by convention $w(\emptyset, a) = 1$. So taking $\alpha_0(c) = 1$, proves the result for $k = 0$.

Now assume that the result holds for $k \geq 0$. To lighten the notations, we will denote x_{k+1} by x.

If $x \notin \mathsf{var}(c)$, then $c^{(k)} = c^{(k+1)}$. By Lemma 23, we also know that $I_{k+1}(c) = I_k(c)$. Thus, if by choosing $\alpha_{k+1}(c) = \alpha_k(c)$, the result follows.

So consider now $I(x)$, i.e. the constraints that contain x as a variable. Let $I(x) = \{c_1, \ldots, c_m\}$ with $c_1 \prec \ldots \prec c_m$. We will prove the result for all of the c_i by induction on i. For $i = 1$, we have by definition, for all $a : \mathsf{var}(c_1) \setminus X_{k+1} \to D$, either $c_1^{(k+1)}(a) = 0$ and there is nothing to prove, or

$$c_1^{(k+1)}(a) = \frac{\sum_{d \in D} c_1^{(k)}(a \oplus_x d)}{|D|}.$$

By induction on k, we get

$$c_1^{(k+1)}(a) = \frac{1}{|D|\alpha_k(c_1)} \sum_{d \in D'} \frac{w(I_k(c_1), a \oplus_x d)}{w(I_k(c_1) \setminus \{c_1\}, a \oplus_x d)}$$

where $D' = \{d \in D \mid c_1(a \oplus_x d) \neq 0\}$. As there is no constraint in $I_k(c_1) \setminus \{c_1\}$ having the variable x, the denominator in the sum does not depends on d. Moreover, $I_{k+1}(c_1) = I_k(c_1)$ by Lemma 23. If $d \notin D'$ then $c_1(a \oplus_x d) = 0$ and hence $w(I_k(c_1), a \oplus_x d) = 0$. Thus, if we set $\alpha_{k+1}(c_1) = |D|\alpha_k(c_1)$, we have

$$\begin{aligned} c_1^{(k+1)}(a) &= \frac{\alpha_{k+1}(c_1)^{-1}}{w(I_{k+1}(c_1) \setminus \{c_1\}, a)} \sum_{d \in D} w(I_{k+1}(c_1), a \oplus_x d) \\ &= \frac{1}{\alpha_{k+1}(c_1)} \cdot \frac{w(I_{k+1}(c_1), a)}{w(I_{k+1}(c_1) \setminus \{c_1\}, a)}. \end{aligned}$$

For $i > 1$, for all $a : \mathsf{var}(c_{i+1}) \setminus X_{k+1} \to D$, either $c_{i+1}^{(k+1)}(a) = 0$ and there is nothing to prove, or by definition

$$c_{i+1}^{(k+1)}(a) = \frac{\sum_{d \in D} \prod_{j \leq i+1} c_j^{(k)}((a \oplus_x d)|_{\mathsf{var}(c_j^{(k)})})}{\sum_{d \in D} \prod_{j \leq i} c_j^{(k)}((a \oplus_x d)|_{\mathsf{var}(c_j^{(k)})})}.$$

Applying the induction hypothesis and Lemma 24 on both the numerator and the denominator, by also remarking that $I_k(c_i) \setminus \{c_1, \ldots, c_i\}$ does not contain any constraint with the variable x

$$c_{i+1}^{(k+1)}(a) = \frac{1}{\alpha_k(c_{i+1})} \cdot \frac{w(I_{k+1}(c_{i+1}), a)}{w(I_{k+1}(c_i), a)} \frac{w(I_{k+1}(c_i) \setminus \{c_1, \ldots, c_i\}, a)}{w(I_{k+1}(c_{i+1}) \setminus \{c_1, \ldots, c_{i+1}\}, a)}.$$

We now apply Corollary 15 with $W := \mathsf{var}(c_{i+1}) \setminus X_{k+1}$, $J_1 := I_{k+1}(c_i) \setminus \{c_1, \ldots, c_i\}$, $J_2 := I_{k+1}(c_{i+1})$, $J_3 := I_{k+1}(c_{i+1}) \setminus \{c_1, \ldots, c_{i+1}\}$ and $J_4 := I_{k+1}(c_i)$. Note that this will yields the desired result: We have $(J_1 \setminus J_3) \cup J_2 = J_2 = I_{k+1}(c_{i+1})$ since $J_1 \subseteq J_3$ and $(J_3 \setminus J_1) \cup J_4 = I_{k+1}(c_{i+1}) \setminus \{c_{i+1}\}$, from combining Lemma 23 and the fact that $\{c_1, \ldots, c_i\} \subseteq J_4$ and $c_{i+1} \notin J_4$.

We now check the conditions of Corollary 15.

(i) Since $J_1 \subseteq J_3$, we have $J_1 \cap J_2 \subseteq J_3$. Moreover, $J_3 \cap J_4 \subseteq J_1$ since $J_1 = J_4 \setminus \{c_1, \ldots, c_i\}$ and J_3 does not contain any of the $c_1, \ldots, c_i$.

(ii) This condition holds since $J_1 \setminus J_3 = \emptyset$.

(iii) This condition is a consequence of condition (v) since $J_1 \cap J_3 \subseteq J_4$.

(iv) This condition holds since $J_1 \setminus J_3 = \emptyset$.

(v) Let $c \in J_3 \setminus J_1$ and $d \in J_4$. We have that $c \prec_{k+1} c_{i+1}$. Moreover, $c_i \preceq_k c_i \prec c_{i+1}$ and $x \in \mathsf{var}(c_i) \cap \mathsf{var}(c_{i+1})$ and consequently, by definition of $\prec_{k+1}$, we have $c_i \prec_{k+1} c_{i+1}$. By definition of J_4 we have $d \prec_{k+1} c_i$ thus by transitivity of $\prec_{k+1}$ we get $d \prec_{k+1} c_{i+1}$. Using Lemma 19 b), it follows that $\mathsf{var}(c) \setminus X_{k+1} \subseteq W$ and $\mathsf{var}(d) \setminus X_{k+1} \subseteq W$.

Note that $c \not\preceq_{k+1} c_i$, because $c \notin J_1$ and $c \notin \{c_1, \ldots, c_{i+1}\}$.

We now show that c and d are incomparable with respect to $\prec_{k+1}$.

By way of contradiction, assume first that $c \prec_{k+1} d$. Then as $d \prec_{k+1} c_i$, we get $c \prec_{k+1} c_i$ which is a contradiction.

Now assume that $d \prec_{k+1} c$. With $d \prec_{k+1} c_i$ and the fact that $\prec$ is a total order we get from Lemma 21 that $c \prec_{k+1} c_i$ or $c_i \prec_{k+1} c$. But we know that $c \not\prec_{k+1} c_i$, so it follows that $c_i \prec_{k+1} c \prec_{k+1} c_{i+1}$ and thus $c_i \prec c \prec c_{i+1}$. By definition of c_i, we have $x \in \mathsf{var}(c_i)$ and by Lemma 19 it follows that $x \in \mathsf{var}(c)$. But this contradicts the choice of c_i as the maximal element in $I(x)$ with respect to $\prec$ that is less than c_{i+1}.

Consequently, c and d are in fact incomparable with respect to $\prec_{k+1}$. Now (v) follows as in as in (iii) in the proof of Lemma 24.

Having checked all conditions, we may apply Corollary 15 which concludes the proof. □

Combining the results of Section 3 and Section 4, we now state the main tractability result of this paper.

Theorem 26. *There exists an algorithm that, given a β-acyclic instance I of $\#\mathrm{CSP}_\mathrm{d}$ on domain D, computes $w(I)$ in polynomial time.*

Proof. In a first step, one computes a β-elimination order for $\mathcal{H}(I)$, which can be done naively in polynomial time, iteratively searching by brute force for a nest point. When it is found, we remove the nest point and iterate.

Then we can iterate the elimination procedure of Theorem 10, respecting the order $\prec$ of Section 4 induced by the elimination order. We make $O(s(I)^2\|I\|)$ arithmetic operations to perform all the elimination steps. The other operations needed are the computation of the new supports of the constraints at each step, which can be done in polynomial time.

Finally, Section 4 provides a good upper bound on the size of the rationals on which we need to perform arithmetic operations. They are always of polynomial bitsize (of size $O(|\mathsf{var}(I)| \log |D|)$), thus each operation can be perform in polynomial time. □

Combining Theorem 26 and Corollary 6 we get the main tractability result for #SAT.

Corollary 27. #SAT *on β-acyclic CNF-formulas can be solved in polynomial time.*

5. 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 [STV14] which we call short the STV-framework. We first show that the STV-framework gives a uniform explanation of all tractability results for #SAT in the literature, extending the results of [STV14]. We see this as strong evidence that the STV-framework is indeed a good formalization of the intuitive notion of "dynamic programming for #SAT".

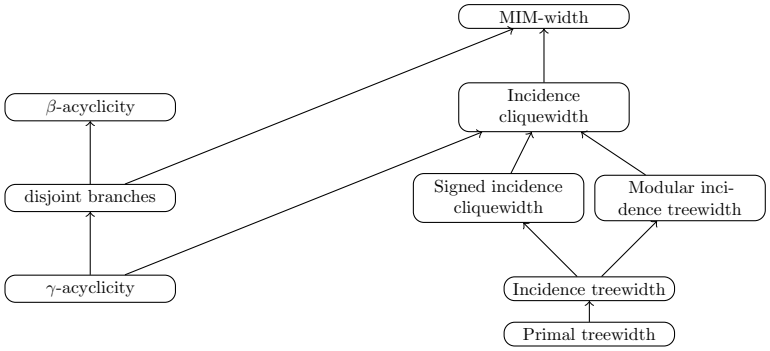

Figure 1: A hierarchy of inclusion of graph and hypergraph classes. Classes not connected by a directed path are incomparable. Note that we leave out PS-width because it is not a graph width measure.

class	lower bound	upper bound
primal treewidth		FPT [SS10]
incidence treewidth		FPT [SS10]
modular incidence treewidth	W1-hard [PSS13]	XP [PSS13]
signed incidence cliquewidth		FPT [FMR08]
incidence cliquewidth	W1-hard [OPS13]	XP [SS13]
MIM-width		XP [STV14]
γ-acyclic		FP [GP04, SS13]
disjoint branches		FP [CDM14]
β-acyclic		FP (this paper)

Table 1: Known complexity results for structural restrictions of #SAT.

Next we show that the STV-framework 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.

5.1. Explaining old results by PS-width

In this section we show that the STV-framework is indeed strong enough to explain all known results on structural #SAT. Figure 1 shows the hierarchy for inclusion formed by the acyclicity notions and classes defined by bounding the width measures from the literature. Most proofs of inclusion can be found in [Fag83, Dur12, GP04, PSS13, CDM14] and the references therein. The relation between disjoint branches and MIM-width and that between β-acyclicity and MIM-width are shown in this paper.

Known complexity results for the restrictions of #SAT can be found in Table 1; for definitions of the appearing complexity classes see e.g. [FG06].

In [Vat12] it is shown that MIM-width is bounded by cliquewidth, so nearly all tractability results of Table 1 follow from [STV14]. To show that the missing results can also be explained in the STV-framework, we only have to recover the tractability results for formulas with disjoint branches decompositions and the fixed-parameter result for formulas of bounded signed incidence cliquewidth. We reprove these results in the following sections by giving upper bounds on the

MIM-width and the PS-width, respectively.

5.1.1. Hypergraphs with disjoint branches

In this section we show how the tractability of #SAT on hypergraphs with a disjoint branches decomposition proved in [CDM14] can be explained by the STV-framework.

A *join tree* (T, λ) of a hypergraph $\mathcal{H} = (V, E)$ consists of a rooted tree T and a mapping $\lambda : V(T) \to E$ such that the following connectivity condition is satisfied: Let $t_1, t_2 \in V(T)$ and $v \in \lambda(t_1) \cap \lambda(t_2)$, then $v \in \lambda(t)$ for every $t \in V(T)$ that lies on the path in T connecting t_1 and t_2. A join tree is a *disjoint branches decomposition* if whenever t_1 and t_2 lie on different branches of T, we have $\lambda(t_1) \cap \lambda(t_2) = \emptyset$. Hypergraphs with disjoint branches decompositions are a strict subclass of β-acyclic hypergraphs [Dur12].

Theorem 28. *[CDM14] There is an algorithm that, given a hypergraph $\mathcal{H}$, in time polynomial in $\|\mathcal{H}\|$ compute a disjoint branches decomposition of $\mathcal{H}$ if one exists and rejects otherwise.*

Lemma 29. *Given a hypergraph $\mathcal{H}$ and a disjoint branches decomposition of $\mathcal{H}$, we can in polynomial time compute a branch decomposition of $I(G)$ of MIM-width at most 2.*

Proof. Let $(\mathcal{T}, \lambda)$ be a disjoint branches decomposition of $\mathcal{H} = (V, E)$. We construct a branch decomposition (T, δ) of $\mathcal{H}$ as follows: The vertices of $\mathcal{T}$ form the internal vertices of T. For every $v \in V$ we introduce a new leaf u labeled by $\delta(u) = v$ connecting it to the vertex of $\mathcal{T}$ that corresponds to the edge containing v that is farthest from the root of $\mathcal{T}$. Observe that this choice is unique because $\mathcal{T}$ has disjoint branches and thus vertices $v \in V$ only appear along a path from the root to a leaf. Furthermore, we add a new leaf u for each $e \in E$ labeled by $\delta(u) = e$, connecting it to the vertex x of $\mathcal{T}$ with $\lambda(x) = e$.

We now make T subcubic: For any internal vertex x, we introduce a binary tree T_x having as leaves the leaf children of x and connect it to x. After that, for every vertex x having more than two children, we introduce again a binary tree T'_x having the children of x as its leaves and connect it to x. The result is a branch decomposition (T, δ) of the incidence graph of $\mathcal{H}$.

We claim that (T, δ) has MIM-width at most 2. So let v be a cut vertex with cut $(X, \bar{X})$. First assume that v lies in one of the T_x. Let $e = \lambda(x)$ be the single $e \in E$ that appears as label of a leaf of T_x. Observe that all $u \in V \cap X$ lie in e. Also, all $u \in V \cap X$ that lie in an edge different from e must lie in a common edge $e' \in E$ that corresponds to the parent of e in $\mathcal{T}$. Since $e' \notin X$ only one vertex in $X \cap V$ can contribute to an independent matching in $I(\mathcal{H})[X, \bar{X}]$. Furthermore, e is the only edge in $E \cap X$, and it follows that the MIM-width of the cut $(X, \bar{X})$ is at most 2.

If v does not lie in any T_x—that is v lies in a T'_y or is a vertex $y \in V(\mathcal{T})$—then the cut $(X, \bar{X})$ corresponds to cutting subtrees $\mathcal{T}_1, \ldots, \mathcal{T}_s$ from a vertex x in $\mathcal{T}$. Every vertex $u \in X \cap V$ lies in an edge $e \in X \cap E$ which is the label $\lambda(x')$ for some vertex x' in a $\mathcal{T}_i$. Now if u is also in an edge $e' \in \bar{X} \cap E$, then $u \in \lambda(x) \in \bar{X} \cap E$. Consequently, only one vertex $u \in X \cap V$ can be an end vertex of an induced matching in $I(\mathcal{H})[X, \bar{X}]$. Furthermore, no vertex u in $\bar{X} \cap V$ is in an edge $e \in X \cap E$, because we connected u to the vertex y farthest from the root in the construction of T and thus cutting outside T_x we cannot be in a situation where $u \notin X$. Consequently, the MIM-width of the cut $(X, \bar{X})$ is at most 1. □

Corollary 30 ([CDM14])**.** #SAT *on hypergraphs with disjoint branches decompositions can be solved in polynomial time.*

Proof. Given a CNF-Formula F, compute a disjoint branches decomposition with Theorem 28. Then apply the construction of Lemma 29 to get a branch decomposition of MIM-width at most 2. Now combining Theorem 9 and Theorem 8 yields the results. □

5.1.2. Signed incidence cliquewidth

In this section we use the STV-framework to reprove a result from [FMR08] stating that #SAT is fixed-parameter tractable parameterized by signed cliquewidth. We first state the relevant definitions from [FMR08].

The *signed incidence graph* $SI(F)$ of a CNF-formula is the incidence graph of F where each edge xC is signed positively or negatively depending on if the variable x appears positively or negatively in the clause C. The set of CNF-formulas of signed cliquewidth at most k is defined as the set of formulas whose signed incidence graph can be obtained by the following operations over graphs whose vertices are coloured by $\{1, \ldots, k\}$, starting from singleton graphs.

1. Disjoint union.
2. Recolouring: For a vertex-coloured signed bipartite graph G, we defined $\rho_{i,j}(G)$ to be the graph that results from recolouring with j all vertices that were previously coloured with i.
3. Positive edge creation: For a vertex-coloured signed bipartite graph G, we define $\eta^{+}_{i,j}(G)$ to be the graph that results from connecting all clause-vertices coloured i to all variable-vertices coloured j, with edges signed positively. We do not add edges between variable vertices coloured i and clause-vertices coloured j, or any other vertices.
4. Negative edge creation: Similarly to above, we define $\eta^{-}_{i,j}(G)$ to be the graph resulting from connecting all clause-vertices coloured with i to all variable-vertices coloured with j, with edges signed negatively.

The *signed cliquewidth* of a CNF-formula is the minimum k such that it has signed cliquewidth at most k.

A *parse tree* for the signed cliquewidth of a formula F is the rooted tree whose leaves hold singleton graphs, whose internal vertices are coloured with the operations of the definitions above (so a vertex corresponding to a disjoint union has two children, and vertices corresponding to other operations have one child), and whose root holds the graph $SI(F)$ (with any vertex colouring).

Given a signed parse tree of a formula F, we construct iteratively a branch decomposition. We assume w.l.o.g. that whenever we make a union, the graphs whose union we take have only disjoint colors in their vertex coloring. This can be easily achieved by at most doubling the number of colors used. Furthermore, we assume that in the end all vertices have the same color.

We construct the branch decomposition along the parse tree iteratively. To this end, we assign a tree T_τ to each sub-parse tree τ. To a singleton v representing a variable of F, we assign a singleton vertex labeled with v. For $\tau = \eta^{+}_{i,j}(\tau')$ and $\tau = \eta^{-}_{i,j}(\tau')$ we set $T_\tau := T_{\tau'}$. For $\tau = \rho_{i,j}(\tau')$ we again let $T_\tau := T_{\tau'}$. Finally, for $\tau = \tau_1 \cup \tau_2$ we introduce a new root and connect it to T_{τ_1} and T_{τ_2}. Observe that T_τ is essentially the tree we get from τ by forgetting internal labels and contracting all paths to edges. Observe that the result (T, δ) is obviously a branch decomposition.

Lemma 31. *(T, δ) has PS-width at most 2^{2k}.*

Proof. Let v be a cut vertex with the cut $(A, \bar{A})$. Let $X := A \cap \mathsf{var}(F)$, $\bar{X} := \bar{A} \cap \mathsf{var}(F)$, $C := A \cap \mathsf{cla}(F)$ and $\bar{C} := \bar{A} \cap \mathsf{cla}(F)$. Let τ be the sub-parse tree which is rooted by the union that led to the introduction of v.

We first show that $|PS(F_{X,\bar{C}})| \leq 2^{2k}$. Observe that when two variables $x, x' \in X$ have the same color in τ, then they must always appear together in every clause in $\bar{C}$ and their sign must be the same. Call X_i the set of variables in X that are colored by i. Then for every assignment of $F_{X,\bar{C}}$ the set of satisfied clauses depends only on if there is a variable in X_i that is set to true if X_i appears positively or if there is a variable in X_i set to false if X_i appears negatively. So to get the same precise satisfiability set, we can delete all but two variables from X_i from $F_{X,\bar{C}}$. It follows that $F_{X,\bar{C}}$ has the same precise satisfiability set as a formula with $2k$ variables. But there are only 2^{2k} assignments to $2k$ variables, so it follows that $|PS(F_{X,\bar{C}})| \leq 2^{2k}$.

We now show that $|PS(F_{\bar{X},C})| \leq 2^{2k}$. To this end observe that if two clauses C, C' in τ have the same color i, then they will contain the same variables in $\bar{X}$ and moreover $C|_{\bar{X}} = C'|_{\bar{X}}$. Thus $F_{\bar{X},C}$ only has k different clauses, so trivially $|PS(F_{\bar{X},C})| \leq 2^{2k}$. □

Corollary 32 ([FMR08])**.** #SAT *on formulas of signed incidence cliquewidth k can be solved in time $2^{O(k)}|F|^2$ assuming that we are provided a parse tree of width k.*

Note that the runtime bound in [FMR08] cannot be easily compared, because the runtime in [FMR08] depends on the size of the parse tree directly and not on the formula. But both results are fixed-parameter results that singly exponentially depend on k, so they are at least very close.

5.2. Lower bounds on MIM-width and PS-width

In this section we will prove the promised lower bound on the PS-width of β-acyclic CNF-formulas. We start off with a simple Lemma that can be seen as a partial reverse of Lemma 9. We remind the reader that a CNF-formula F is called *monotone* if all variables appear only positively in F.

Lemma 33. *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 independent 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 independent 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 follow 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. □

To a graph $G = (V, E)$ we define a graph $G' = (V', E')$ 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 34. *G' is chordal bipartite.*

Proof. We have to show that every cycle C in G' of length at least 6 has a chord. We consider two cases: Assume first that C contains no vertex $p_{e,v}$ and consequently no $q_{e,v}$ either. Then all vertices of C are x_v or y_v and so C is a cycle in the complete bipartite graph induced by the x_v and y_v. Clearly, C has a chord then.

Now assume that C contains a vertex $p_{e,v}$ and consequently also $q_{e,v}$. Let $e = uv$. Then C must also contain x_v and y_u, so $x_v y_u \in E'$ is a chord. □

Lemma 35. *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 independent 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 independent. 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 independent.

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 independent matching of size $\frac{\mathbf{tw}(G)}{6}$ in $G'[X', \bar{X}']$. This completes the proof. □

Using the connection between vertex expansion and treewidth (see [GM09]) the following lemma is easy to show.

Lemma 36. *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 37. *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 36. 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 $w_e v$. It is well-known that subdividing edges does not decrease the treewidth of a graph (see e.g. [Die05]), 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 34 and the bound on the MIM-width follows by combining Lemma 36 and Lemma 35. □

We can now easily prove the main result of this section.

Corollary 38. *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.*

Proof. Let $\mathcal{F}$ be the class of monotone CNF-formulas having the class $\mathcal{G}'$ of Corollary 37 as its incidence graphs. By Theorem 5 the formulas in $\mathcal{F}$ are β-acyclic. Combining the bound on the MIM-width of G' with Lemma 33 then directly yields the result. □

It follows that the STV-framework cannot prove subexponential runtime bounds for #SAT on β-acyclic formulas.

6. Conclusion

We have shown that β-acyclic #SAT can be solved in polynomial time, a question left open in [CDM14]. 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 [STV14] that explains all old 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, in particular those of Section 4, 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 [GP04], 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 [OPS13].

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 [STV14] 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 in the future 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

[ADM86] 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.

[BDG+12] 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.

[BLS99] A. Brandstädt, V.B. Le, and J.P. Spinrad. *Graph Classes: A Survey*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.

[Bod93] H.L. Bodlaender. A tourist guide through treewidth. *Acta Cybern.*, 11(1-2):1–21, 1993.

[Bra14] J. Brault-Baron. Hypergraph Acyclicity Revisited. *ArXiv e-prints*, March 2014.

[CC12] 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.

[CDM14] F. Capelli, A. Durand, and S. Mengel. Hypergraph acyclicity and propositional model counting. *CoRR*, abs/1401.6307, 2014.

[CGH09] 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.

[Die05] R. Diestel. *Graph Theory (Graduate Texts in Mathematics)*. Springer, August 2005.

[DJ04] V. Dalmau and P. Jonsson. The complexity of counting homomorphisms seen from the other side. *Theor. Comput. Sci.*, 329(1-3):315–323, 2004.

[Dur12] D. Duris. Some characterizations of γ and β-acyclicity of hypergraphs. *Inf. Process. Lett.*, 112(16):617–620, 2012.

[Fag83] R. Fagin. Degrees of acyclicity for hypergraphs and relational database schemes. *Journal of the ACM*, 30(3):514–550, 1983.

[FG06] J. Flum and M. Grohe. *Parameterized Complexity Theory*. Springer-Verlag New York Inc, 2006.

[FMR08] 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.

[GLS00] G. Gottlob, N. Leone, and F. Scarcello. A comparison of structural csp decomposition methods. *Artif. Intell.*, 124(2):243–282, 2000.

[GM09] M. Grohe and D. Marx. On tree width, bramble size, and expansion. *J. Comb. Theory, Ser. B*, 99(1):218–228, 2009.

[GP04] G. Gottlob and R. Pichler. Hypergraphs in Model Checking: Acyclicity and Hypertree-Width versus Clique-Width. *SIAM Journal on Computing*, 33(2), 2004.

[OPS13] S. Ordyniak, D. Paulusma, and S. Szeider. Satisfiability of acyclic and almost acyclic CNF formulas. *Theoretical Computer Science*, 481:85–99, 2013.

[PSS13] 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.

[Rot96] D. Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82(1–2):273 – 302, 1996.

[SS10] M. Samer and S. Szeider. Algorithms for propositional model counting. *Journal of Discrete Algorithms*, 8(1):50–64, 2010.

[SS13] 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.

[STV14] S. Hortemo Sæther, J.A. Telle, and M. Vatshelle. Solving MaxSAT and #SAT on structured CNF formulas. *CoRR*, abs/1402.6485, 2014.

[Vat12] M. Vatshelle. *New Width Parameters of Graphs*. PhD thesis, University of Bergen, 2012.

A. Extension to MaxSAT

The algorithm described in this paper can also be turned into an algorithm for MaxCSP_d—the problem of computing, given a set of weighted constraints I, the value $m(I) = \max\{\prod_{c\in I} c(a|_{\mathsf{var}(c)}) \mid a \in D^{\mathsf{var}(I)}\}$. We first show that we can use MaxCSP_d to solve MaxSAT, the problem of computing the maximum number of clauses of a CNF-formula F that can be satisfied simultaneously.

Lemma 39. *Given a CNF-formula F, one can compute in polynomial time a set I of weighted constraints with default values on variables $\mathsf{var}(F)$ and domain $\{0,1\}$ such that*

- $\mathcal{H}(F) = \mathcal{H}(I)$,
- *for all $a \in \{0,1\}^{\mathsf{var}(F)}$, $m(I,a) = 2^s$ where $s = |\{C \in F \mid a \models C\}|$, and*
- $s(I) = \|I\| = |F|$

Proof. For each clause C of F, we define a constraint c with default value 2 whose variables are the variables of C and such that $\mathsf{supp}(c) = \{a\}$ and $c(a) = 1$, where a is the only assignment of $\mathsf{var}(C)$ that is not a satisfying assignment to C. It is easy to check that this construction has the above properties. □

Corollary 40. MaxSAT *is polynomial time reducible to* MaxCSP_d. *Moreover,* MaxSAT *restricted to β-acyclic formulas is polynomial time reducible to* MaxCSP_d *restricted to β-acyclic instances.*

Proof. We transform a CNF-formula F into an instance I of MaxCSP_d using Lemma 39. We have $\text{MaxCSP}_\text{d}(I) = 2^s$ where $s = \text{MaxSAT}(F)$, so it just remains to take the logarithm in base 2. □

We now show how to adapt our algorithm for $\#\text{CSP}_\text{d}$ to MaxCSP_d.

Theorem 41. *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 $c'_i := (f'_i, \mu)$ is the weighted constraint on variables $\mathsf{var}(c) \setminus \{x\}$, with default value $\mu(c_i)$ and $\mathsf{supp}(c'_i) := \{a \in D^{Y \setminus \{x\}} \mid \exists d \in D, (a \oplus_x d) \in \mathsf{supp}(c)\}$. Moreover, for all $a \in \mathsf{supp}(c'_i)$, let $P_i(a,d) := \prod_{j=1}^{i} c_j((a \oplus_x d)|_{\mathsf{var}(c_j)})$ and $P_0(a,d) = 1$. We define:*

$$f'_i(a) := \frac{\max_{d \in D} P_i(a,d)}{\max_{d \in D} P_{i-1}(a,d)}$$

if $\max_{d \in D} P_{i-1}(a,d) \neq 0$ and $f'_i(a) := 0$ otherwise.

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

Proof. The proof is analogous to that of Theorem 10. Remark that the special case where $\max_{d \in D} P_{i-1}(a,d) = 0$ follows similarly to there since $\max_{d \in D} P_{i-1}(a,d) = 0$ implies that for all d we have $P_{i-1}(a,d) = 0$. □

Now remark that max is commutative, associative, that is $\max(a, \max(b,c)) = \max(\max(a,b), c)$ and that distributes with multiplication since all numbers are positive, that is $\max(ab, ac) = a \max(b,c)$. Moreover, we have

$$\max(\frac{a}{b}, \frac{c}{d}) = \frac{\max(ad, cb)}{bd}.$$

Thus, the results of Section 4.1 can be adapted in a straightforward fashion and the results of 4.2 still hold. We can now adapt Theorem 25 (we use adapted notations for $m(J,a)$ for $J \subseteq I$ and a a partial assignment).

Theorem 42. *For all $c \in I$ and $k \geq 0$, for all $a : \mathsf{var}(c) \setminus X_k \to D$, either*

$$c^{(k)}(a) = 0$$

or

$$c^{(k)}(a) = \frac{m(I_k(c), a)}{m(I_k(c) \setminus \{c\}, a)}.$$

Now the tractability results for MaxCSP_d and MaxSAT follow directly.

Theorem 43. *There is an algorithm that, given a β-acyclic instance I of* MaxCSP_d, *computes $m(I)$ in polynomial time.*

Theorem 44. *There is an algorithm that solves* MaxSAT *on β-acyclic CNF-formulas in polynomial time.*