

Hypergraph Acyclicity and Propositional Model Counting

Florent Capelli¹, Arnaud Durand^{1,2}, and Stefan Mengel³ *

¹ IMJ UMR 7586 - Logique, Université Paris Diderot, France

² LSV UMR 8643, ENS Cachan, France

³ Laboratoire d'Informatique, LIX UMR 7161, Ecole Polytechnique, France

Abstract. We show that the propositional model counting problem #SAT for CNF-formulas with hypergraphs that allow a disjoint branches decomposition can be solved in polynomial time. We show that this class of hypergraphs is incomparable to hypergraphs of bounded incidence cliquewidth which were the biggest class of hypergraphs for which #SAT was known to be solvable in polynomial time so far. Furthermore, we present a polynomial time algorithm that computes a disjoint branches decomposition of a given hypergraph if it exists and rejects otherwise. Finally, we show that some slight extensions of the class of hypergraphs with disjoint branches decompositions lead to intractable #SAT, leaving open how to generalize the counting result of this paper.

1 Introduction

Proposition model counting (#SAT) is the problem of counting satisfying assignments (models) to a CNF-formula. It is the canonical #P-hard counting problem and is important due to its applications in Artificial Intelligence. Unfortunately, #SAT is extremely hard to solve: Even on restricted classes of formulas like monotone 2CNF-formulas or Horn 2CNF-formulas it is NP-hard to approximate within a factor of $2^{n^{1-\epsilon}}$ for any $\epsilon > 0$ [13]. Fortunately, this is not the end of the story: While syntactical restrictions on the types of allowed clauses do not lead to tractable counting, there is a growing body of work that successfully applies so-called *structural* restrictions to #SAT, see e.g. [7, 14, 12, 16]. In this line of work one does not restrict the individual clauses of CNF-formulas but instead the interaction between the variables in the formula. This is done by assigning graphs or hypergraphs to formulas and then restricting the class of (hyper)graphs that are allowed for instances (see Section 2 for details). In this paper we present a new class of hypergraphs, such with disjoint branches decompositions [5], for which #SAT is tractable.

Having a disjoint branches decomposition is a so-called acyclicity notion for hypergraphs. Unlike for graphs, there are several reasonable ways of defining acyclicity for hypergraphs [6] which have been very successful in database theory. Mostly three “degrees of acyclicity” have been studied: α -acyclicity, β -acyclicity

* Partially supported by DFG grants BU 1371/2-2 and BU 1371/3-1.

and γ -acyclicity, where the α -acyclic hypergraphs form the most general and the γ -acyclic hypergraphs the least general class. Prior to this paper it was known that $\#SAT$ for CNF-formulas with α -acyclic hypergraphs was $\#P$ -hard [14], while it is tractable for γ -acyclic hypergraphs as the latter have incidence cliquewidth bounded by 3 [9] and thus the results of [16] apply.

To understand the influence of hypergraph acyclicity on the complexity of $\#SAT$, the next natural step is thus analyzing the intermediate case of β -acyclic hypergraphs. For this class it is known that SAT is tractable [11], unlike for α -acyclic hypergraphs. Unfortunately, the algorithm in [11] is based on a resolution-like method and it is not clear whether one can obtain tractability for counting from the method used for decision. In fact, most classical decision results based on tractability of resolution (such as for $2-SAT$) or unit propagation ($Horn-SAT$) do not extend to counting as the respective counting problems are hard (see e.g. [13]).

Unfortunately, $\#SAT$ for CNF-formulas with β -acyclic hypergraphs has turned out to be a stubborn problem whose complexity could so far not be determined despite considerable effort by us and others [15]. A natural approach which we follow in this paper is thus trying to understand slightly more restrictive notions of acyclicity. We focus here on hypergraphs with disjoint branches decompositions, a notion which was introduced by Duris [5] and which lies strictly between β -acyclicity and γ -acyclicity. We show that for CNF-formulas whose hypergraphs have a disjoint branches decompositions we can solve $\#SAT$ in polynomial time. We also show that hypergraphs with disjoint branches decompositions are incomparable to hypergraphs with bounded incidence cliquewidth which so far were the biggest class of hypergraphs for which $\#SAT$ was known to be tractable. Thus our results give a new class of tractable instances for $\#SAT$, pushing back the known tractability frontier for this problem.

Our main contribution is twofold: Most importantly, we present the promised counting algorithm for CNF-formulas whose hypergraphs have a disjoint branches decomposition in Section 3. Secondly, we present in Section 4 a polynomial time algorithm that checks if a hypergraph has a disjoint branches decomposition and if so also constructs it. On the one hand, this gives some confidence that hypergraphs with disjoint branches decompositions form a well-behaved class as it can be decided in polynomial time. On the other hand, the counting algorithm will depend on knowing a decomposition, so its computation is an essential part of the counting procedure. Finally, in Section 5 we then turn to generalizing the results of this paper, unfortunately showing only negative results. We consider some natural looking extensions of hypergraphs with disjoint branches and show that $\#SAT$ is intractable on these classes under standard complexity theoretic assumptions.

Due to lack of space some of the proofs are omitted and can be found in the full version of this paper.

2 Preliminaries and notation

2.1 Hypergraphs and graphs associated to CNF-formulas

The *primal graph* of a CNF-formula F has as vertices the variables of F and two vertices are connected by an edge if they appear in a common clause of F . The *incidence graph* of F is defined as the bipartite graph which has as vertices the variables and the clauses of F and two vertices u and v are connected by an edge if u is a variable and v is a clause such that u appears in v .

A (finite) hypergraph $\mathcal{H}$ is a pair (V, E) where V is a finite set and $E \subseteq \mathcal{P}(V)$. A subhypergraph $\mathcal{H}' = (V', E')$ of $\mathcal{H}$ is a hypergraph with $V' \subseteq V$ and $E' \subseteq \{e \cap V' \mid e \in E, e \cap V' \neq \emptyset\}$. A path between two vertices $u, v \in V$ is a sequence $e_1, \dots, e_k$ such that $u \in e_1$, $v \in e_k$ and for every $i = 1, \dots, k-1$ we have $e_i \cap e_{i+1} \neq \emptyset$. A hypergraph $\mathcal{H}$ is called *connected* if there is a path between every pair of vertices of $\mathcal{H}$. A (connected) *component* of $\mathcal{H}$ is defined to be a maximal connected subhypergraph of $\mathcal{H}$.

To a CNF-formula F we associate a hypergraph $\mathcal{H} = (V, E)$ where V is the variable set of F and the hyperedge set E contains for each clause of F an edge containing the variables of the clause.

Graph Decompositions. We will not recall basic graph decompositions such as tree-width and clique-width (see e.g. [9, 7]). A class of CNF-formulas is defined to be of bounded (signed) incidence clique-width, if their (signed) incidence graphs are of bounded clique-width. A set $X \subseteq V$ of vertices of a graph is called a *module*, if every $v \in V \setminus X$ has the same set of neighbours and non-neighbours in X . If X is a module of a graph G , the graph $G' = (V', E')$ obtained after contraction of X is defined by $V' := (V \setminus X) \cup \{x\}$ where x is a new vertex not in V and $E' := (E \cap (V \setminus X)^2) \cup \{ux : u \notin X \text{ and } \exists v \in X \text{ s.t. } uv \in E\}$. A class of CNF-formulas is of bounded *modular incidence treewidth* if their incidence graphs are of bounded tree-width after contracting all modules.

Acyclicity in hypergraphs. In contrast to the graph setting, there are several non equivalent notions of acyclicity for hypergraphs [6]. Most of these notions have many equivalent definitions, but we will restrict ourselves to definitions using the notion of join trees.

Definition 1. A join tree of a hypergraph $\mathcal{H} = (V, E)$ is a pair $(\mathcal{T}, \lambda)$ where $\mathcal{T} = (N, T)$ is a tree and λ is a bijection between N and E such that:

- for each $e \in E$, there is a $t \in N$ such that $\lambda(t) = e$, and
- for each $v \in V$, the set $\{t \in N \mid v \in \lambda(t)\}$ is a connected subtree of $\mathcal{T}$.

The second condition in Definition 1 is often called the *connectedness condition*. We will follow the usual convention of identifying an edge $e \in E$ with the vertex $\lambda(e)$. We call a join tree $(\mathcal{T}, \lambda)$ a *join path* if the underlying tree $\mathcal{T}$ is a path.

A hypergraph is defined to be α -acyclic if it has a join tree [6]. This is the most general acyclicity notion for hypergraphs commonly considered. However,

class	lower bound	upper bound
primal treewidth		FPT [14]
incidence treewidth		FPT [14]
modular incidence treewidth	W[1] -hard [12]	XP [12]
signed incidence cliquewidth		FPT [7]
incidence cliquewidth	W[1] -hard [11]	XP [16]
γ -acyclic		FP [9, 16]
α -acyclic	#P -hard [14]	#P
disjoint branches		FP (this paper)

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

α -acyclicity is not closed under taking subhypergraphs. To remedy this situation, one considers the restricted notion of β -acyclicity where a hypergraph is defined to be β -acyclic if it is α -acyclic and all of its subhypergraphs are also all α -acyclic.

Definition 2. A disjoint branches decomposition of a hypergraph $\mathcal{H}$ is a join tree $(\mathcal{T}, \lambda)$ such that for every two nodes t and t' appearing on different branches of $\mathcal{T}$ we have $\lambda(t) \cap \lambda(t') = \emptyset$.

Disjoint branches decompositions were introduced in [5]. A hypergraph is called γ -acyclic if and only if it has a disjoint branches decomposition for any choice of hyperedge as a root. Furthermore, hypergraphs with disjoint branches decompositions are β -acyclic.

2.2 Known complexity results and comparisons between classes

Known complexity results for the restrictions of #SAT can be found in Table 2.2; for definitions of the appearing complexity classes see e.g. [8].

The acyclicity notions and classes defined by bounding the introduced width measures form a hierarchy for inclusion (see Figure 1). Most proofs of inclusion can be found in [6, 5, 9, 12] and the references therein.

We now show that the hypergraphs with disjoint branches decompositions have unbounded cliquewidth. In fact we will even show this for hypergraphs with join paths. Since join paths do not branch, these hypergraphs are a subclass of the hypergraphs with disjoint branches decompositions.

We will use the following characterization of hypergraphs with join paths.

Lemma 1. A hypergraph $\mathcal{H} = (V, E)$ has a join path if and only if there exists an order $<_E$ on the edge set E of $\mathcal{H}$ such that for all $e, f, g \in E$ such that $e <_E f <_E g$, if $v \in e \cap g$ then $v \in f$.

Definition 3. Let $G = (X, Y, E)$ be a bipartite graph. A strong ordering $(<_X, <_Y)$ of G is a pair of orderings on X and Y such that for all $x, x' \in X$ and $y, y' \in Y$, such that $x <_X x'$ and $y <_Y y'$, if $(x, y) \in E$ and $(x', y') \in E$, then $(x, y') \in E$ and $(x', y) \in E$. G is called a bipartite permutation graph if it admits a strong ordering.

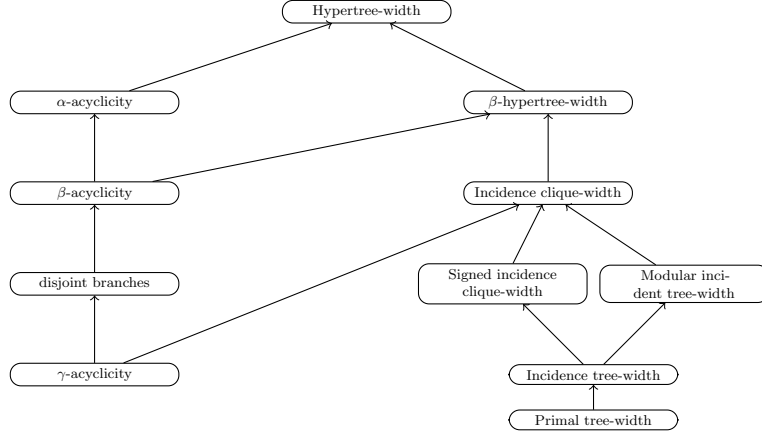


Fig. 1. A hierarchy of inclusion of graph and hypergraph classes. Classes not connected by a directed path are incomparable.

Brandstädt and Lozin showed the following property of bipartite permutation graphs.

Lemma 2 ([2]). *Bipartite permutation graphs have unbounded cliquewidth.*

It turns out that hypergraphs with a bipartite permutation incidence graph admit a join path.

Proposition 1. *Every hypergraph $\mathcal{H}$ without empty hyperedges, whose incidence graph $\mathcal{H}_I$ is a bipartite permutation graph, has a join path.*

Proof. Let $(<_V, <_E)$ be a strong ordering of $\mathcal{H}_I = (V, E, A)$. We claim that for all $e <_E f <_E g$, if $v \in e \cap g$ then $v \in f$. Indeed, as f is not empty, there exists $w \in f$. If $w = v$, there is nothing to prove. Otherwise, suppose that $w <_V v$. Then, by definition of strong orderings, as $(f, w) \in A$ and $(g, v) \in A$, we have $(f, v) \in A$. Thus $v \in f$. The case $v <_V w$ follows symmetrically: $(f, w) \in A$ and $(e, v) \in A$ implies that $(f, v) \in A$ so $v \in f$. Thus the ordering $<_E$ has the property of Lemma 1 and it follows that $\mathcal{H}$ has a join path. $\square$

By combining Lemma 2 and Proposition 1 we get:

Corollary 1. *The class of CNF-formulas with join paths has unbounded incidence cliquewidth. The same is thus true for CNF-formulas with disjoint branches decompositions.*

2.3 Representation of #SAT by constraint satisfaction problems

It will be convenient to describe our counting algorithm in the framework of constraint satisfaction problems in negative representation [4]. We will discuss below how this representation relates to #SAT.

Let D be a finite set called domain. A constraint $C = (R, \bar{x})$ is a pair where $R \subseteq D^r$ is a relation and $\bar{x} = (x_{i_1}, \dots, x_{i_r})$ is a list of variables of length r . An instance Φ of the (uniform) constraint satisfaction problem, is a set of constraints. We denote by $\text{var}(\Phi)$ the set X of variables of Φ . The instance Φ is satisfied by an assignment $a : \text{var}(\Phi) \rightarrow D$ if for all $(R, \bar{x}) \in \Phi$ we have $a(\bar{x}) = (a(x_{i_1}), \dots, a(x_{i_r}))$ is in the relation R . We denote this by $a \models \Phi$.

The associated counting problem, $\#\text{CSP}$, is, given an instance Φ , to compute

$$|\{a \mid a \models \Phi\}|,$$

i.e., the number of satisfying assignments of Φ .

No hypothesis is made above on the arity of relations which is not a priori bounded and may differ for different relations. So, it may be more succinct to represent each relation R by listing the tuples in its complement $R^c := D^r \setminus R$. Consequently, we define the counting constraint satisfaction problem in negative representation, $\#\text{CSP}_{\text{neg}}$, that is to compute, given Φ where each relation R is encoded by listing the tuples in R^c , the number of satisfying assignments of Φ .

The relation to $\#\text{SAT}$: It is natural to represent CNF-formulas by a Boolean CSP_{neg} -instance. Indeed, as we do not bound the length k of clauses, it is more realistic to represent each associated constraint relation not by its set of $2^k - 1$ models (as common in the area of constraint satisfaction) but by its complement containing the unique counter model of the clause.

In the other direction one can easily encode every $\#\text{CSP}_{\text{neg}}$ -instance by a CNF-formula (see also [3]): In a first step encode all domain elements in binary, introducing vertex modules in the hypergraph. Then we encode every tuple in every relation by a clause that disallows the respective tuple. Observe that after the contraction of some modules, the CNF-formula has the same hypergraph as the original $\#\text{CSP}_{\text{neg}}$ -instance. Since the class of hypergraphs with disjoint branches decompositions is stable under introducing or contracting of modules, it follows that $\#\text{SAT}$ and $\#\text{CSP}_{\text{neg}}$ are equivalent for our considerations.

3 Counting solutions of disjoint branches queries

In this section we will show that $\#\text{CSP}_{\text{neg}}$ —and thus also $\#\text{SAT}$ —restricted to hypergraphs with a disjoint branches decomposition can be solved in polynomial time. It will be convenient to work with inputs in a disjunctive form: Let $\#\text{DCSP}$ be the problem of computing, given a finite domain D and a set of constraints $C_1, \dots, C_m$ with variables X , the number of assignments $a : X \rightarrow D$ such that at least one constraint is satisfied. The following is easy to see.

Observation 1 $\#\text{CSP}_{\text{neg}}$ is polynomial time reducible to $\#\text{DCSP}$.

Note that, as discussed in Section 2.3, we may assume that the domain of all relations is $\{0, 1\}$, so we restrict ourselves to this case.

Let us introduce some notation: Let X, Y be two sets of variables and let $a: X \rightarrow \{0, 1\}$ and $b: Y \rightarrow \{0, 1\}$ be two assignments. We call a and b *consistent*, symbol $a \sim b$, if they agree on their common variables $X \cap Y$. If, in addition, $X \subseteq Y$, we write $a \subseteq b$. Finally, if a and b have disjoint domains, i.e., $X \cap Y = \emptyset$, we denote by $a \oplus b$ the assignment on $X \cup Y$ defined in the obvious way.

For an assignment $a: X \rightarrow \{0, 1\}$ and a set of variables $Y \subseteq X$ the restriction of a onto Y is denoted by $a|_Y$.

Let ϕ be an instance and let ψ be a subformula of ϕ . Let X a subset of the variables such that $\text{var}(\psi) \subseteq X \subseteq \text{var}(\phi)$ and let a be a partial assignment of variables in $\text{var}(\phi)$. We denote by $\text{Sol}_X(\psi, a) = \{b: X \rightarrow \{0, 1\}, b \models \psi, a \sim b\}$ and by $S_X(\psi, a) = |\text{Sol}_X(\psi, a)|$. The number of solutions of ϕ is then $S_{\text{var}(\phi)}(\phi, \emptyset)$, where $\emptyset$ denotes the empty partial assignment. We show that by computing a polynomial number of values $S_X(\psi, a)$ in polynomial time, we can compute $S_{\text{var}(\phi)}(\phi, \emptyset)$. To this end we state several lemmas that will allow us a recursive computation.

The first lemma shows how the disjointness naturally appears when we want to count solutions:

Lemma 3. *Let ϕ_1 and ϕ_2 be two instances and $X \supseteq \text{var}(\phi_1) \cup \text{var}(\phi_2)$, then*

$$S_X(\phi_1 \vee \phi_2, a) = S_X(\phi_1, a) + S_X(\phi_2, a) - |\text{Sol}_X(\phi_1, a) \cap \text{Sol}_X(\phi_2, a)|.$$

The next lemma will allow us to efficiently compute $\text{Sol}_X(\phi_1 \vee \phi_2, a)$ recursively.

Lemma 4. *Let $X_1 = \text{var}(\phi_1)$ and $X_2 = \text{var}(\phi_2)$. Assume that $X_1 \cap X_2 = \emptyset$ and let $X = X_1 \cup X_2$. Let a be a partial assignment of the variables of X and $a_1 = a|_{X_1}$, $a_2 = a|_{X_2}$. Then $\text{Sol}_X(\phi_1, a) \cap \text{Sol}_X(\phi_2, a) = \{b_1 \oplus b_2 \mid b_i: X_i \rightarrow \{0, 1\}, b_i \models \phi_i, a_i \subseteq b_i\}$ and $|\text{Sol}_X(\phi_1, a) \cap \text{Sol}_X(\phi_2, a)| = S_{X_1}(\phi_1, a_1)S_{X_2}(\phi_2, a_2)$.*

We now show how we can add variables that do not appear in ϕ .

Lemma 5. *Let $Y \subseteq X$ and $a: X_0 \rightarrow \{0, 1\}$ for $X_0 \subseteq X$ with $\text{var}(\phi) \subseteq X_0 \cup Y$. Then*

$$S_X(\phi, a) = 2^{|X \setminus (Y \cup X_0)|} S_Y(\phi, a|_Y).$$

The next corollary lets us handle the disjunction of more than two terms. The proof is by induction with Lemma 4.

Corollary 2. *Let $\phi_1, \dots, \phi_k$ be formulas with $X_i = \text{var}(\phi_i)$ and $X_i \cap X_j = \emptyset$ for every combination $i, j \in [k], i \neq j$. Let X be a set such that $X_1 \cup \dots \cup X_k \subseteq X$, let $Y := X_0 \cup X_1 \cup \dots \cup X_k$ and let $a: X_0 \rightarrow \{0, 1\}$ for $X_0 \subseteq X$, $a_i = a|_{X_i}$. Then*

$$S_X\left(\bigvee_{j=1}^k \phi_j, a\right) = 2^{|X \setminus Y|} \sum_{i=1}^k S_{X_i}(\phi_i, a_i) \prod_{j=1}^{i-1} (2^{|X_j \setminus X_0|} - S_{X_j}(\phi_j, a_j)) \prod_{j=i+1}^k 2^{|X_j \setminus X_0|}$$

Lemma 6. $|\text{Sol}_X(R, a) \cap \text{Sol}_X(\phi, a)| = \sum_{b \in \text{Sol}_X(R, a)} S_X(\phi, b)$

We now finally show the main result of this section.

Theorem 1. *There is a polynomial time algorithm that, given an instance $\phi = \bigvee_{i=1}^m R_i$ and a disjoint branches decomposition of the hypergraph of ϕ , computes the number of satisfying assignments of ϕ .*

Proof. Let $\phi = R_1 \vee \dots \vee R_m$ be an instance with hypergraph $\mathcal{H}$. Let $r_i := |R_i|$ and let $r := \sum_{i=1}^m r_i$. Let furthermore $(\mathcal{T}, \lambda)$ be a disjoint branches decomposition of $\mathcal{H}$. For a vertex t of $\mathcal{T}$, we denote by $\mathcal{T}_t$ the subtree of $\mathcal{T}$ rooted in t and by ϕ_t the associated subinstance. Furthermore, R_t is defined to be the relation associated to t . Finally, we denote by V_t the set of variables of ϕ_t .

We will give a polynomial time algorithm that computes inductively from the leaves to the root of $\mathcal{T}$ certain values $S_X(\psi, a)$ where ψ is a subinstance of ϕ . Our goal is to compute $S_{V_r}(\phi_r, \emptyset)$ where r is the root of $\mathcal{T}$, since this value is the number of solutions of ϕ . More precisely, for a given t in the tree, we compute $S_{V_t}(\phi_t, \emptyset)$ and for all ancestors u of t and all $b \in R_u$, we compute $S_{V_t}(\phi_t, b|_{\text{var}(R_t)})$. Since there are m vertices in $\mathcal{T}$ and at most $r + 1$ values to compute for each vertex, we will to compute at most $m(r + 1)$ different values. We will show how to compute these values in polynomial time to get a polynomial time algorithm overall.

Let t be a vertex of $\mathcal{T}$, u one of his ancestors and $b \in R_u$. If t is a leaf, then ϕ_t consists only of the relation R_t . Then $S_{V_t}(\phi_t, \emptyset) = R_t$ and $S_{V_t}(\phi_t, b|_{\text{var}(R_t)}) = \{a \in R_t \mid a \sim b\}$ so the computations can be done efficiently.

Now assume that t has children $t_1, \dots, t_k$. We set $V_i := V_{t_i}$, $\phi_i := \phi_{t_i}$ and $R_i := R_{t_i}$. Observe that by Lemma 3 we have

$$S_{V_t}(\phi_t, \emptyset) = S_{V_t}(R_t, \emptyset) + S_{V_t}\left(\bigvee_{j=1}^k \phi_j, \emptyset\right) - |S_{V_t}(R_t, \emptyset) \cap S_{V_t}\left(\bigvee_{j=1}^k \phi_j, \emptyset\right)|.$$

As the variables of the ϕ_i are disjoint, Corollary 2 lets us compute $S_{V_t}(\bigvee_{j=1}^k \phi_j, \emptyset)$ in time $O(k)$ if the values of $S_{V_i}(\phi_i, \emptyset)$ are precomputed, which is the case by induction.

In addition, $S_{V_t}(R_t, \emptyset) = 2^{|V_t \setminus \text{var}(R_t)|} |R_t|$ since a solution of R_t on variables V_t is a solution of R_t on $\text{var}(R_t)$ and any assignment of the other variables.

Finally, we have $|S_{V_t}(R_t, \emptyset) \cap S_{V_t}(\bigvee_{j=1}^k \phi_j, \emptyset)| = \sum_{a \in R_t} S_{V_t}(\bigvee_{j=1}^k \phi_j, a)$ by Lemma 6. By Corollary 2, one can compute for each a the value $S_{V_t}(\bigvee_{j=1}^k \phi_j, a)$ in time $O(k)$ if the values of $S_{V_i}(\phi_i, a|_{V_i})$ are precomputed. But since the domain of a is $\text{var}(R_t)$, we have $a|_{V_i} = a|_{\text{var}(R_t) \cap V_i} = a|_{\text{var}(R_i)}$ by connectedness of the variables in the join tree $\mathcal{T}$. Thus $S_{V_i}(\phi_i, a|_{V_i}) = S_{V_i}(\phi_i, a|_{\text{var}(R_i)})$ which is precomputed by hypothesis.

Let $b' = b|_{\text{var}(R_t)}$. We compute $S_{V_t}(\phi_t, b')$ in the following way, similarly to before. We start with Lemma 3 to get

$$S_{V_t}(\phi_t, b') = S_{V_t}(R_t, b') + S_{V_t}\left(\bigvee_{j=1}^k \phi_j, b'\right) - |S_{V_t}(R_t, b') \cap S_{V_t}\left(\bigvee_{j=1}^k \phi_j, b'\right)|.$$

Again, by Corollary 2, one can compute $S_{V_t}(\bigvee_{j=1}^k \phi_j, b')$ in time $O(k)$ if $S_{V_i}(\phi_i, b'|_{V_i})$ are known. But as the domain of b' is $\text{var}(R_t)$, $b'|_{V_i} = b'|_{\text{var}(R_i)}$ by connectedness of the variables in $\mathcal{T}$. So $S_{V_i}(\phi_i, b'|_{V_i})$ is precomputed since u is also an ancestor of t_i .

Moreover, $S_{V_t}(R_t, b') = \sum_{a \in R_t, b' \subseteq a} 2^{|V_t \setminus \text{var}(R_t)|}$ which can be computed in time $O(|R_t|)$.

Finally, by Lemma 6, we have

$$|Sol_{V_t}(R_t, b') \cap Sol_{V_t}(\bigvee_{j=1}^k \phi_j, b')| = \sum_{a \in R_t, b' \subseteq a} S_{V_t}(\bigvee_{j=1}^k \phi_j, a).$$

And again, by Corollary 2, we can compute $S_{V_t}(\bigvee_{j=1}^k \phi_j, a)$ in time $O(k)$ if $S_{V_i}(\phi_i, a|_{V_i})$ is precomputed. For the same reasons as above, $a|_{V_i} = a|_{\text{var}(R_i)}$, thus these values were already computed by induction.

To conclude, we have seen how to compute the $S_{V_t}(\phi_t, \emptyset)$ and $S_{V_t}(\phi_t, b|_{\text{var}(R_t)})$ for each $b \in R_u$ where u is an ancestor of t with $O(k \cdot r)$ arithmetic operations. Thus we can compute $S_{V_r}(\phi_r, \emptyset)$ in polynomial time. $\square$

4 Computing disjoint branches decompositions

In this section we will show how to compute disjoint branches decompositions of hypergraphs in polynomial time. We will first introduce *PQF*-trees, the data-structure that our algorithm relies on, then consider some structural properties of hypergraphs with disjoint branches decompositions and finally describe the algorithm itself, relying on objects we call *A*-separators.

4.1 PQF-trees

PQ-trees are a data structure introduced by Booth and Lueker [1] originally to check matrices for the so-called consecutive ones property. This problem can be reformulated as follows in our setting: Given a hypergraph $\mathcal{H} = (V, E)$, is there an ordering $\ell = e_1 \dots e_m$ of the edges such that if $v \in e_i \cap e_j$, then for all $i \leq k \leq j$, $v \in e_k$? We encode ordering of edges by lists. We call such a list *consistent* for $\mathcal{H}$. Note that the notion of consistent lists matches exactly our notion of join paths.

A *PQ*-tree is a compact way of representing all the consistent lists for a hypergraph. We introduce a generalization of this data structure which we call *PQF*-trees.

Definition 4. Let $\mathcal{H} = (V, E)$ be a hypergraph. A *PQF*-tree for $\mathcal{H}$ is defined to be an ordered tree with leaf set E such that

- the internal nodes are labeled with *P*, *Q* or *F*,
- the *P*-nodes and *F*-nodes have at least two children, and
- the *Q*-nodes have at least 3 children.

A PQF-tree without F-nodes is called a PQ-tree.

PQF-trees will be used to encode sets of permutations of the edge set of a hypergraph that have certain properties. We write these permutations simply as (ordered) lists. To this end, we define some notation for lists and sets of lists. The concatenation of two ordered lists ℓ_1, ℓ_2 will be denoted by $\ell_1\ell_2$. If L_1, L_2 are two sets of lists, we denote by L_1L_2 the set $\{\ell_1\ell_2 \mid \ell_1 \in L_1, \ell_2 \in L_2\}$.

Definition 5. The frontiers $\mathcal{F}(T)$ of a PQF-tree T for $\mathcal{H} = (V, E)$ are a set of ordered list of the elements of E defined inductively by

- if T is a leaf e , then $\mathcal{F}(T) = \{e\}$,
 - if T is rooted in t , having children $t_1, \dots, t_k$, then
 - if t is an F-node then $\mathcal{F}(T) = \mathcal{F}(T_1) \dots \mathcal{F}(T_k)$,
 - if t is a Q-node then $\mathcal{F}(T) = (\mathcal{F}(T_1) \dots \mathcal{F}(T_k)) \cup (\mathcal{F}(T_k) \dots \mathcal{F}(T_1))$,
 - if t is a P-node then $\mathcal{F}(T) = \bigcup_{\sigma \in \mathcal{S}_k} \mathcal{F}(T_{\sigma(1)}) \dots \mathcal{F}(T_{\sigma(k)})$ where $\mathcal{S}_k$ is the set of permutations of $[k]$,
- where T_i is the subtree of T rooted in t_i .

If for all $\ell \in \mathcal{F}(T)$, ℓ is a consistent list for $\mathcal{H}$, we say that T is consistent for $\mathcal{H}$.

We recall the main theorem of [1], which allows to compute all possible join paths of a hypergraph in polynomial time.

Theorem 2 ([1]). Given a hypergraph $\mathcal{H} = (V, E)$, one can compute in time $O(|E||V|)$ a PQ-tree T such that $\mathcal{F}(T)$ is exactly the set of consistent lists for $\mathcal{H}$.

In order to compute disjoint branches decompositions, we will need to compute join paths with additional restrictions. This is the reason for the introduction of F-nodes. It will be convenient to not have F-nodes that are children of other F-nodes and thus we introduce the following normal form for PQF-trees.

Definition 6. A PQF-tree T is said to be in normal form if there is no F-node in T having an F-node as a child.

Clearly, if an F-node t has a child u in T which is also an F-node, then we can remove u from T and connect its children to t without changing $\mathcal{F}(T)$. Thus we may always assume that all PQF-trees we encounter are in normal form.

We will in the remainder of this section use certain subtrees of PQF-trees which we call PQF-subtrees. These will be trees rooted in a vertex t of a PQF-tree, but they will not necessarily contain all descendants of t . Instead, we allow to “cut off” certain trees that are rooted by children of t . We now give a formal definition of PQF-subtrees. As usual, the subtree rooted in t is defined to be the tree induced by t and all its descendants.

Definition 7. Let T be a PQF-tree, and let t be a vertex of T . A subgraph S of T is said to be a PQF-subtree rooted in t if

- t is a leaf and S consists of the graph containing only t ,
- t is a P -node and S is the subtree rooted in t , or
- t is a Q -node or an F -node with children $t_1, \dots, t_k$ and there exists i, j such that $1 \leq i < j \leq k$ and S is the graph containing t and $T_i, \dots, T_k$, the subtrees rooted in $t_i, \dots, t_j$.

We will now show that PQF -subtrees allow us to filter the frontier of a PQF -tree for certain lists that we will be interested in later. Remember that the *depth* of a node in a tree is its distance from the root.

Lemma 7. *Let T be a consistent PQF -tree for (V, E) in normal form. Let $V' \subseteq V$ and $A = \{e \in E \mid V' \subseteq e\}$. Then there exists a PQF -subtree $T_{V'}$ of T such that the labels of the leaves of $T_{V'}$ are exactly A .*

During the construction of disjoint branches decompositions later, we will put restrictions on the position of some edges in join paths. To do so we will use the algorithm of the following proposition.

Proposition 2. *There is a polynomial time algorithm **Force** that, given a PQF -tree T and a PQF -subtree S of T , computes in polynomial time a PQF -tree $T' = \text{Force}(T, S)$ such that $\mathcal{F}(T') = \{\ell_1 \ell_2 \in \mathcal{F}(T) \mid \ell_2 \in \mathcal{F}(S)\}$. If this set is empty, the algorithm rejects.*

Corollary 3. *Let $\mathcal{H} = (V, E)$ be a hypergraph and T a consistent PQF -tree for $\mathcal{H}$. Let $V' \subseteq V$ and $A = \{e \in E \mid e \cap V' \neq \emptyset\}$. Suppose that for all $e, f \in A$, $e \cap V' \subseteq f \cap V'$ or $f \cap V' \subseteq e \cap V'$. Then we can compute in polynomial time a PQF -tree T' such that $\mathcal{F}(T') = \{e_1 \dots e_m \in \mathcal{F}(T) \mid \forall i < j, e_i, e_j \in A \Rightarrow e_i \cap V' \subseteq e_j \cap V'\}$.*

Proof. We want to compute T' such that the frontiers of T' are the frontiers of T in which the edges of A appear in increasing order with respect to inclusion relative to V' . The set $\{e \cap V' \mid e \in A\}$ is ordered by inclusion, thus it has a smallest element V_1 and a biggest element V_2 . Obviously, $V_1 \subseteq V_2 \subseteq V'$. Furthermore, $A = \{e \in E \mid V_1 \subseteq e\}$ and $A_2 = \{e \in A \mid V_2 \subseteq e\}$ is not empty.

First use Lemma 7 to find a PQF -subtree S of T whose leaves are exactly $\{e \mid V_1 \subseteq e\} = A$. Then use Lemma 7 again to find a PQF -subtree R of S whose leaves are exactly A_2 . Now use the procedure **Force**(S, R) to compute S' as in Proposition 2 and let T' be the tree where we replace S by S' in T . As finding the right subtrees in T can easily be done in polynomial time by finding a least common ancestor and **Force** is a polynomial time, it is clear that one can compute T' as well. We now show that T' has the desired properties.

To this end, let $\ell \in \mathcal{F}(T')$. By definition of T' , we have that ℓ is also in $\mathcal{F}(T)$. In addition, ℓ is of the form $\ell_1 \ell_A \ell_2$ with $\ell_A \in \mathcal{F}(S')$. By definition of **Force**, ℓ_A is of the form $\sigma_1 \sigma_2$ with $\sigma_2 \in \mathcal{F}(R)$, that is, consisting only of the edges in A_2 , which are maximal for the inclusion. Let $g \in A_2$. Let $e, f \in A$ with e appearing before f in ℓ_A . If $e \cap V'$ is not included in $f \cap V'$, then there exists a $v' \in V_2$ such that $v' \in e$, $v' \notin f$ and $v' \in g$ since $V_2 \subseteq g$. That would lead to an inconsistent list which is a contradiction.

Reciprocally, let $\ell \in \{e_1 \dots e_m \in \mathcal{F}(T) \mid \forall i < j, e_i, e_j \in A \Rightarrow e_i \cap V' \subseteq e_j \cap V'\}$. ℓ is of the form $\ell_1 \ell_A \ell_2$ with $\ell_A \in \mathcal{F}(S)$. As it is organized by inclusion relative to V' , the elements of A_2 should all lie at the end of ℓ_A . Thus $\ell_A \in \{\sigma_1 \sigma_2 \mid \sigma_2 \in \mathcal{F}(R)\} = \mathcal{F}(S')$ by Proposition 2. It follows that $\ell \in \mathcal{F}(T')$. $\square$

4.2 Db-rootable hypergraphs and separators

In this section we will prove several structural properties of hypergraphs with disjoint branches decompositions which we will use in the algorithm in the next section.

Definition 8. Let $\mathcal{H} = (V, E)$ be a hypergraph. For $e \in E$, we say that $\mathcal{H}$ is db-rootable in e if there is a disjoint branches decomposition of $\mathcal{H}$ rooted in e .

The algorithm for the construction of disjoint branches decompositions will delete edges of hypergraphs. To this end we introduce the following notation.

Definition 9. For a hypergraph $\mathcal{H} = (V, E)$ and an edge $e \in E$, we denote $\mathcal{H} \setminus e$ the hypergraph $(V_e, E \setminus \{e\})$ where $V_e := \bigcup_{e' \in E \setminus \{e\}} e'$. For a set $A = \{e_1, \dots, e_k\} \subseteq E$, we define $\mathcal{H} \setminus A$ to be the hypergraph $((\mathcal{H} \setminus e_1) \setminus \dots) \setminus e_k$.

We make the following observation which will simplify our arguments later.

Observation 2 Let $\mathcal{H} = (V, E)$ be a hypergraph and $e \in E$. Then $\mathcal{H}$ is db-rootable in e if and only if for every connected component $C = (V_C, E_C)$ of $\mathcal{H} \setminus e$ the hypergraph $C' := (V_C \cup e, E_C \cup \{e\})$ is db-rootable in e .

Lemma 8. Let $\mathcal{H}$ be a hypergraph $\mathcal{H}$ with a disjoint branches decomposition $\mathcal{T}$ that is rooted in e . Let v_1, v_2 be two vertices that appear in different trees $\mathcal{T}_1$ and $\mathcal{T}_2$ of the forest $\mathcal{T} \setminus \{e\}$. Then v_1 and v_2 lie in different connected components of $\mathcal{H} \setminus e$.

Lemma 9. If $\mathcal{H}$ is db-rootable in e and $\mathcal{H} \setminus e = (V', E')$ has one connected component then there exists $e' \in E'$ such that $e \cap V' \subseteq e'$.

By Observation 2 we may deal with the components of $\mathcal{H} \setminus e$ for a hypergraph $\mathcal{H}$ and an edge e independently. Thus we will in this section always assume that $\mathcal{H} \setminus e$ just has a single component. We will consider restricted join paths that we call A -separators. In the following, all join paths will be denoted as ordered lists of edges, which corresponds to the notation in Section 4.1.

Definition 10. Let $\mathcal{H} = (V, E)$ be a hypergraph and let $\mathcal{P} = a_1 \dots a_m$ be a join path of $A \subseteq E$. We call $\mathcal{P}$ an A -separator of $\mathcal{H}$ if for all connected components $C = (V_C, E_C)$ of $\mathcal{H} \setminus A$ we have that if $a_j \cap V_C \neq \emptyset$, then for all $i \leq j$, $a_i \cap V_C \subseteq a_j \cap V_C$.

Theorem 3. There is a polynomial time algorithm `ComputeSeparator`($\mathcal{H}, A$) that, given a hypergraph $\mathcal{H} = (V, E)$ and a set $A \subseteq E$, computes an A -separator of $\mathcal{H}$ if it exists and rejects otherwise.

Proof. We will iterate the algorithm described in Corollary 3. We first compute a PQ -tree T_0 such that $\mathcal{F}(T_0)$ is the set of all join paths for A using Theorem 2. Then, for each component $C = (V_C, E_C)$ of $\mathcal{H} \setminus A$, we iteratively do the following: If there are edges a_i and a_j such that $a_i \cap V_C \not\subseteq a_j \cap V_C$ and $a_j \cap V_C \not\subseteq a_i \cap V_C$, then $\mathcal{H}$ cannot have an A -separator and we reject. Otherwise, we can use the algorithm of Corollary 3 on the PQF -tree we have computed so far to construct a PQF -tree whose frontiers respect the order condition for the edges imposed by C or rejects.

An easy induction shows that if this algorithm does not reject at any point, then the computed PQF -tree T has as frontiers all join paths that satisfy the order conditions of Definition 10. Thus we can choose one of these join paths arbitrarily as the desired A -separator. $\square$

Definition 11. Let $\mathcal{H} = (V, E)$ be a hypergraph and let $A \subseteq E$. We call an A -separator $\mathcal{P} = a_1 \dots a_m$ of $\mathcal{H}$ a strong A -separator if for all connected components $C = (V_C, E_C)$ of $\mathcal{H} \setminus A$, $C' = (V_C, E_C \cup \{a_{l_C}\})$ is db-rootable in a_{l_C} where $l_C = \max\{i \mid a_i \cap V_C \neq \emptyset\}$.

Proposition 3. Let $\mathcal{H} = (V, E)$ be a hypergraph and $A \subseteq E$. If there exists a strong A -separator of $\mathcal{H}$ then all A -separators of $\mathcal{H}$ are strong.

Theorem 4. Let $\mathcal{H} = (V, E)$ be a hypergraph, $e \in E$, and $\mathcal{H}' = (V', E') = \mathcal{H} \setminus e$. Let furthermore $A_e := \{e' \in E' \mid e \cap V' \subseteq e'\}$. Assume that $\mathcal{H}'$ has only one connected component. Then $\mathcal{H}$ is db-rootable in e if and only if there exists a strong A_e -separator of $\mathcal{H}'$.

Proof. Suppose $\mathcal{H}$ is db-rootable in e . Let $\mathcal{T}$ be a disjoint branches decomposition of $\mathcal{H}$ rooted in e . By Lemma 9, A_e is not empty. Let $v \in e \cap V'$. We know that v is contained in all edges $e' \in A_e$. Thus, by disjointness, the edges in A_e are on the same branch of $\mathcal{T}$. Moreover, we claim that A_e is connected in $\mathcal{T}$. To see this, suppose that $b \in E$ is between $a, c \in A_e$ on this branch. Then by connectedness, $e \cap V' \subseteq b$, so $b \in A_e$. Consequently, A_e is connected and thus forms a path. Let $\mathcal{P} = a_1 \dots a_k$ be this path in $\mathcal{T}$ in the direction from the root to the leaves of the tree. We claim that $\mathcal{P}$ is a strong A_e -separator.

To this end, let $C = (V_C, E_C)$ be a connected component of $\mathcal{H}' \setminus A_e$. We consider the forest obtained by removing A_e in $\mathcal{T}$. By Lemma 8, vertices in different trees of this forest are in different connected component of $\mathcal{H}' \setminus A_e$ as well. Thus there is a tree $\mathcal{T}_C$ that contains all the edges in E_C . Let a_{l_C} be the edge of A_e to which the root of $\mathcal{T}_C$ is connected. If $j > l_C$, then we claim that $a_j \cap V_C = \emptyset$. Assume this were not the case, then $\mathcal{T}_C$ and the subtree of a_{l_C} containing a_j were not disjoint which is a contradiction to $\mathcal{T}$ being a disjoint branches decomposition. Thus $a_j \cap V_C = \emptyset$. Now consider $i < j \leq l_C$ and let $v \in a_i \cap V_C$. We have $v \in a_{l_C} \cap V_C$ by the connectedness condition since v appears in $\mathcal{T}_C$ and, again by connectedness the connectedness condition, $v \in a_j \cap V_C$. Thus, $a_i \cap V_C \subseteq a_j \cap V_C$. It follows that $\mathcal{P}$ is an A_e -separator. As $C \cup \{a_{l_C}\}$ is db-rootable in a_{l_C} using the tree $\mathcal{T}_C$, we have that $\mathcal{P}$ is a strong A_e -separator.

Assume now that there is a strong A_e -separator $\mathcal{P} = a_1 \dots a_k$. For a connected component C of $\mathcal{H}' \setminus A_e$, let $l_C := \max\{i \mid a_i \cap V_C \neq \emptyset\}$. By definition, there exists a disjoint branches decomposition $\mathcal{T}_C$ for $C \cup \{a_{l_C}\}$ rooted in a_{l_C} . We construct a disjoint branches decomposition for $\mathcal{H}$ as follows:

- We root the path $ea_1 \dots a_k$ in e .
- For each component C of $\mathcal{H}' \setminus A_e$, we connect the root of $\mathcal{T}_C$ to a_{l_C} .

We claim that the resulting tree $\mathcal{T}$ is a disjoint branches decomposition. We first show that the branches of $\mathcal{T}$ are disjoint. Indeed, if a, b are edges in two different branches then two cases can occur: Either a and b are in two different connected component of $\mathcal{H}' \setminus A_e$. But then they are disjoint, because they lie in different components of $\mathcal{H}$ by construction. Otherwise, let a be in a connected component C of $\mathcal{H}' \setminus A_e$ and let b be in A_e . But as b is on a different branch as a , it follows that b comes after a_{l_C} on $\mathcal{P}$. Thus $b \cap V_C = \emptyset$ and it follows that $a \cap b = \emptyset$ since $a \subseteq V_C$. Thus the branches of $\mathcal{T}$ are disjoint.

Now we show that $\mathcal{T}$ is a join tree, i.e., it satisfies the connectedness condition for all vertices. For a connected component C of $\mathcal{H}' \setminus A_e$, for a vertex $v \in V_C$, its connectedness is ensured along $\mathcal{P}$ since $\mathcal{P}$ is a join path and in $\mathcal{T}_C$ since $\mathcal{T}_C$ is a join tree. Furthermore, by construction $v \in a_{l_C}$ so the edges containing v are connected in $\mathcal{T}$. For a vertex v , which does not appear in any V_C , that is, which only appears in A_e , its connectedness is ensured by the fact that $\mathcal{P}$ is a join path for A_e . $\square$

Corollary 4. *There is a polynomial time algorithm $\text{ComputeDB}(\mathcal{H}, e)$ that, given a hypergraph $\mathcal{H} = (V, E)$ and an edge $e \in E$, returns a disjoint branches decomposition of $\mathcal{H}$ rooted in e if it exists and rejects otherwise.*

A rough analysis of the runtime of the algorithm gives an upper bound of $O(|E|^3|V|)$. We believe firmly this can be improved by using smart data structures.

5 Some negative results on generalizations

We now discuss several approaches to generalizing the counting algorithm for hypergraphs with disjoint branches to more general classes of hypergraphs. Unfortunately, all these results will be negative as we will show hardness results for all extensions we consider. We still feel these results are worthwhile because they might help in guiding future research.

We call a hypergraph $\mathcal{H}$ *undirected path acyclic* if there is a join tree $(\mathcal{T}, \lambda)$ of $\mathcal{H}$ such that for every $v \in V$ the edge set $\{e \in E \mid v \in e\}$ forms an undirected path in $\mathcal{T}$. Undirected path acyclicity is a seemingly natural generalization of disjoint branches acyclicity by allowing undirected paths instead of directed paths. Unfortunately, undirected path acyclicity does not allow tractable SAT and thus it is not a good generalization in our setting.

Theorem 5. *SAT on undirected path acyclic CNF-formulas is NP-complete.*

Algorithm 1 The algorithm `ComputeDB` of Corollary 4.

```

ComputeDB( $\mathcal{H} = (V, E)$ ,  $e$ ) =
  if  $|E| = 1$  then return the tree with the only vertex  $e$ 
  else
    for each connected component  $\mathcal{H}_i = (V_i, E_i)$  of  $\mathcal{H} \setminus e$  do
       $A_e \leftarrow \{e' \in E_i \mid e \cap V_i \subseteq e'\}$ 
      if  $A_e = \emptyset$  then Reject.
       $\mathcal{P} \leftarrow \text{ComputeSeparator}(\mathcal{H}_i, A_e)$ 
       $\mathcal{T}_i \leftarrow \mathcal{P}$ 
      for each connected component  $C = (V_C, E_C)$  of  $\mathcal{H}_i \setminus A_e$  do
         $l_C \leftarrow \max\{j \mid a_j \cap V_C \neq \emptyset\}$  (where  $\mathcal{P} = a_1 \dots a_k$ )
         $C' \leftarrow (V_C \cup a_{l_C}, E_C \cup \{a_{l_C}\})$ 
         $\mathcal{T}_C \leftarrow \text{ComputeDB}(C', a_{l_C})$ 
        connect  $\mathcal{T}_C$  to  $\mathcal{T}_i$  in  $a_{l_C}$ 
    return the tree rooted in  $e$  having  $\mathcal{T}_1, \dots, \mathcal{T}_p$  as children

```

One also can generalize disjoint branches decompositions by allowing limited intersections between branches. Unfortunately, this approach leads to hard counting problems even if we only allow the intersection to contain one variable.

Theorem 6. *#SAT is #P-hard for CNF-formulas that have a join tree in which the branches may have a pairwise intersection containing one variable.*

6 Conclusion

We have presented a new structural class of tractable #SAT instances, those whose hypergraphs admit a disjoint branches decomposition. To this end, we also presented an algorithm that computes the decompositions.

Several questions remain, the most obvious open problem certainly being the complexity of #SAT on β -acyclic hypergraphs. Can one show a #P-completeness result or a polynomial time algorithm for this case?

We would like to turn the disjoint branches property into a hypergraph width measure such that #SAT—or even SAT—for the hypergraphs for which this width measure is bounded is tractable? Can we construct this measure to even allow fixed-parameter tractability? Note that it is known that the parameterization by incidence cliquewidth does not allow fixed-parameter tractability [11].

More generally, we feel that it is very desirable to understand the tractability frontier for SAT and #SAT with respect to structural restrictions better overall. Is there a width measure that generalizes both hypergraphs with disjoint branches and incidence cliquewidth that leads to tractable #SAT? A natural candidate would be β -hypertree width (see Figure 1). Are there other classes of hypergraphs incomparable to those studied so far that give large structural classes of tractable #SAT-instances? Note that there is a similar line in the area of constraint satisfaction (see e.g. [10] for an overview) that has been very successful but unfortunately does not apply directly.

References

1. Booth, K., Lueker, G.: Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *Journal of Computer and System Sciences* 13(3), 335–379 (1976), <http://www.sciencedirect.com/science/article/pii/S0022000076800451>
2. Brandstädt, A., Lozin, V.V.: On the linear structure and clique-width of bipartite permutation graphs. *Ars Combinatoria* 67(1), 273–281 (2003)
3. Brault-Baron, J.: A Negative Conjunctive Query is Easy if and only if it is Beta-Acyclic. In: *Computer Science Logic, 26th International Workshop/21st Annual Conference of the EACSL*. pp. 137–151 (2012)
4. Cohen, D., Green, M., Houghton, C.: Constraint representations and structural tractability. In: *Proceedings of the 15th international conference on Principles and practice of constraint programming*. pp. 289–303. CP’09 (2009), <http://dl.acm.org/citation.cfm?id=1788994.1789021>
5. Duris, D.: Some characterizations of γ and β -acyclicity of hypergraphs. *Inf. Process. Lett.* 112(16), 617–620 (2012)
6. Fagin, R.: Degrees of acyclicity for hypergraphs and relational database schemes. *Journal of the ACM* 30(3), 514–550 (1983)
7. Fischer, E., Makowsky, J., Ravve, E.: Counting truth assignments of formulas of bounded tree-width or clique-width. *Discrete Applied Mathematics* 156(4), 511–529 (2008)
8. Flum, J., Grohe, M.: *Parameterized Complexity Theory*. Springer-Verlag New York Inc (2006)
9. Gottlob, G., Pichler, R.: Hypergraphs in Model Checking: Acyclicity and Hypertree-Width versus Clique-Width. *SIAM Journal on Computing* 33(2) (2004)
10. Miklós, Z.: *Understanding Tractable Decompositions for Constraint Satisfaction*. Ph.D. thesis, University of Oxford (2008)
11. Ordyniak, S., Paulusma, D., Szeider, S.: Satisfiability of acyclic and almost acyclic CNF formulas. *Theoretical Computer Science* 481, 85–99 (2013)
12. Paulusma, D., Slivovsky, F., Szeider, S.: Model Counting for CNF Formulas of Bounded Modular Treewidth. In: *30th International Symposium on Theoretical Aspects of Computer Science, STACS 2013*. pp. 55–66 (2013)
13. Roth, D.: On the hardness of approximate reasoning. *Artificial Intelligence* 82(1–2), 273 – 302 (1996), <http://www.sciencedirect.com/science/article/pii/0004370294000921>
14. Samer, M., Szeider, S.: Algorithms for propositional model counting. *Journal of Discrete Algorithms* 8(1), 50–64 (2010)
15. Slivovsky, F.: personal communication (2014)
16. Slivovsky, F., Szeider, S.: Model Counting for Formulas of Bounded Clique-Width. In: *Algorithms and Computation - 24th International Symposium, ISAAC 2013*. pp. 677–687 (2013)