Université Paris Diderot (Paris 7) Sorbonne Paris Cité
UFR de Mathématiques

Thèse

pour obtenir le titre de

Docteur de l'Université Paris Diderot (Paris 7) Sorbonne Paris Cité

Spécialité Informatique

présentée par

Florent CAPELLI

Structural restrictions of CNF-formulas: applications to model counting and knowledge compilation.

Directeur de thèse : **Arnaud DURAND**

Soutenue publiquement le *27 Juin 2016* devant le jury composé de :

M.	**Arnaud**	**Durand**	Directeur
M.	**Dieter**	**Kratsch**	Examinateur (excusé)
M.	**Pierre**	**Marquis**	Rapporteur
M.	**Luc**	**Segoufin**	Examinateur
M.	**Olivier**	**Serre**	Examinateur
M.	**Stefan**	**Szeider**	Rapporteur (excusé)
M.	**Bruno**	**Zanuttini**	Examinateur

Abstract

It is well-known that clause restrictions of CNF-formulas such as 2-SAT or Horn-SAT are easy instances of the problem SAT. It is however not the case for harder problems such as #SAT, the problem of counting the satisfying assignments of a CNF-formula: #2-SAT is already as hard as the general case. Fortunately, structural restrictions of the input formula, that are restrictions on the way the variables interact with the clauses, have been a successful approach to find large classes of formulas for which #SAT is doable in polynomial time.

In this thesis, we investigate the question of understanding the tractability frontier of the problem #SAT. We prove new tractability results, in particular, the tractability of #SAT on β-acyclic instances, a structural restriction for which only SAT was known to be tractable. Moreover we show that for every tractable class known so far, we can transform the counting algorithm into a compilation algorithm that construct a succinct representation of the input CNF-formula that support many queries in polynomial time such as counting or enumeration.

Finally, we study the theoretical limits of structural restrictions by proving new lower bounds on the size of the such representations of CNFs.

Résumé (en français)

Le problème SAT est connu pour être facile sur des instances particulières où la forme des clauses autorisées est restreinte, comme c'est le cas pour 2-SAT ou Horn-SAT. En revanche, le problème de comptage associé, #SAT, est déjà aussi difficile que le cas général pour ces sous-problèmes. On peut toutefois trouver de nombreux cas où ce problème peut être résolu en temps polynomial en restreignant structurellement la formule d'entrée, c'est-à-dire en restreignant la façon dont les variables interagissent avec les clauses.

Dans cette thèse, nous nous intéressons à la question de comprendre quelles sont les restrictions structurelles pertinentes pour ce problème et essayons de déterminer la frontière entre les cas faciles et les cas aussi difficiles que le cas général. Nous introduisons des algorithmes en temps polynomial pour de nouvelles restrictions structurelles. En particulier, nous prouvons que #SAT sur des formules β-acycliques, une restriction structurelle pour laquelle seule la décision était connue comme étant facile, peut être résolu en temps polynomial. De plus, nous montrons que tous les algorithmes de comptage connus à ce jour pour ces restrictions structurelles peuvent être vus comme des algorithmes de compilation qui transforment une formule CNF en un petit circuit booléen ayant de fortes propriétés qui permettent une résolution rapide de différents problèmes comme le comptage ou l'énumération.

Enfin, nous nous intéressons aux limites de l'approche par restrictions structurelles en prouvant des bornes inférieures non-polynomiales sur la taille de tels circuits représentant des formules CNF.

Remerciements

Je ne serais jamais arrivé à aucun des deux bouts de ce travail de thèse sans la présence, l'aide et les conseils de nombreuses personnes que je souhaite remercier ici.

Recherche. Je tenais tout d'abord à remercier Pierre Marquis et Stefan Szeider pour avoir accepté de rapporter ma thèse. Merci aussi à Dieter Kratsch, Luc Segoufin, Olivier Serre et Bruno Zanuttini qui me font l'honneur d'être membres du jury de soutenance.

J'ai aimé travailler avec de nombreuses personnes lors de ces trois années et c'est grâce à eux que j'ai pris autant de plaisir à faire de la recherche et à enseigner : merci à Yann Strozecki, plus particulièrement pour son aide dans la relecture du manuscrit, Simone Bova et Friedrich Slivovsky, pour nos nombreuses discussions sur la compilation, Amélie Gheerbrant et Cristina Sirangelo, avec qui nous avons beaucoup joué aux jeux d'Ehrenfreucht-Fraïssé, Jean-Marie Lagniez, SATeux magicien, qui m'a souvent rappelé que "de toutes façons, en pratique, ça marchera pas". Merci aussi à Stéphane Boucheron avec qui l'enseignement n'a jamais été aussi simple et agréable.

Bien avant cela, j'étais un étudiant un peu désorienté devant l'étendue de ce qu'est aujourd'hui l'informatique théorique. Je voudrais remercier Hervé Fournier, Guillaume Malod et Sylvain Périfel qui m'ont donné le goût de la recherche lors d'un stage au LIAFA. Merci aussi à tous mes professeurs de l'ENS Lyon qui m'ont fait découvrir de nombreux domaines différents, en particulier Patrick Baillot et Pascal Koiran, grâce à qui j'ai découvert la théorie de la complexité.

Ma vie de chercheur débutant se serait vite compliquée sans l'aide inestimable des gestionnaires de l'équipe Logique. Merci, dans l'ordre chronologique, à Élodie Destrebecq, Aurélie Oumezzaouche, que nous sommes toujours très heureux de voir revenir nous saluer, Claire Lavollay et Mylène Merciris. Merci aussi à Élise Delos qui a toujours été là pour répondre à mes questions hebdomadaires depuis que j'ai rendu ce manuscrit.

Je tiens à remercier plus particulièrement Stefan Mengel qui est la personne avec qui j'ai le plus travaillé durant ma thèse et qui fut un modèle pour moi. Grâce à lui, j'ai beaucoup appris sur les aspects concrets de la recherche comme, par exemple, quelles sont les bonnes habitudes à avoir lors de l'écriture d'un papier ou d'un résumé ou encore comment construire et décrire un programme de

recherche. Il a su souvent me guider lorsque je ne voyais plus comment avancer et nos discussions ont toujours été très enrichissantes. Je le remercie aussi pour son aide dans la relecture des chapitres de ce manuscrit, notamment pour ses corrections utiles de mes erreurs d'anglais récurrentes. J'espère qu'il ne m'en voudra pas d'avoir été un si mauvais élève dans ce cas précis.

Mais surtout je remercie de tout cœur Arnaud Durand pour son travail formidable en tant que directeur de thèse. Je ne suis pas le premier, ni, j'en suis sûr, ne serai le dernier à l'écrire, mais je voudrais rappeler ici à quel point il a su être toujours présent pendant ces trois années pour m'écouter, me guider et me conseiller, tout en me laissant beaucoup d'autonomie et d'indépendance dans mes recherches. J'ai beaucoup appris à ses côtés au travers de nombreuses discussions techniques devant le tableau (à feutres, au diable le romanesque !) ou bavardages informels, tant sur le plan scientifique et humain que sur les aspects administratifs et sociaux du travail de chercheur, moins romantiques certes, mais tout aussi nécessaires. C'est en majeure partie grâce à ses conseils que je pars sans crainte l'an prochain tenter ma chance dans ce vaste monde qu'est la recherche.

Famille. Merci à mes parents sans qui je ne serais jamais arrivé jusqu'ici. Ils m'ont toujours soutenu dans tous les aspects de la vie tout en me laissant libre de mes choix. Merci à eux pour la joie de vivre qu'ils m'ont transmise, leur patience et leur amour. C'est chez eux que j'ai écrit les premières lignes de cette thèse en automne dans le confort matériel nécessaire à la réussite de ce genre de travail. C'est grâce à eux que j'ai le goût de l'enseignement et que je suis assez curieux pour aimer faire de la recherche et suffisamment têtu pour vouloir continuer (c'est aussi un peu parce que j'ai peur de mal calculer la tension d'un câble de télésiège...). Et, plus concrètement, c'est grâce à eux que le pot qui suivra ma soutenance sera aussi réussi et que le présent chapitre sera sans *fôtes d'ortograf*. Merci à ma sœur Estelle, pour ses petits textos bêtes et méchants, ses coups de téléphone nécessaires que je rate une fois sur deux et pour ses petits apéros avec Maxime quand je rentre au bercail. Un merci particulier à Aurélia, qui n'a pas fui lorsque je me suis tranformé en grincheux-irritable-et-étourdi-thésard-qui-écrit, mais m'a compris et soutenu. Je ne regrette pas les heures passées dans le TGV pour la rejoindre.

"Doctorat d'état de *name dropping*" [Buk11]. Un grand merci à mes colocs. Eux non plus n'ont pas fui pendant la rédaction et ont patiemment supporté mes coups de folie passagers. Merci à Xouille pour son amitié et son humour. Merci à Angèle qui rayonne de joie et de force. Ceux qui ne la connaissent pas croiront peut-être à un compliment banal mais il faut avoir passé une heure à ses côtés pour se rendre compte qu'on peut difficilement ne pas la remercier pour cela. Merci à Brice, le petit dernier, rider épicurien, pour son amour universel et son savoir encyclopédique de la punchline.

Merci à tous ces thésards qui font de Sophie Germain un endroit vivant et convivial. Merci à Takfarinas d'être passé du cobureau à l'ami, à vélo entre Nantes

et Bordeaux. Merci au M (Martin) et au A (Antoine), voisins de bureau rêvés et soutiens essentiels de rédaction. Le premier : compagnon infaillible du septième étage, pompeur de café et gourmet gourmand. Le second : *extracornidaire* artiste (auteur du fameux "Une icorne"), toujours à la recherche d'un site internet pour se persuader qu'il fera beau à Fontainebleau. Merci à ceux que je connais depuis encore plus longtemps et qui sont devenus des amis indispensables : Bruno, inlassable compagnon de chanson depuis sept ans déjà, Guillaume, faux-jumeau scientifique et littéraire, Ioana, pour sa gentillesse inégalée, Étienne, compagnon éternel de snack (sauce algérienne bien sûr !). Merci Alexandre, R.E.D.A, Vinicius, Juan-Pablo, Jean-Michel, V.E.R.O (il est tout croche ce paragraphe !), membres éminents du CROUS Crew. Merci aux doctorants du sixième qui travaillent d'arrache-pied pour créer une vraie complicité entre nous tous à l'IMJ : merci Assia, Baptiste, Charles, Charlotte, David, Élie, Kévin, Marco, Victoria. Merci à Laure, Charles et Luc, partis il y a déjà bien longtemps du LIAFA, mais qui furent là pour me faire rire un peu lors de ma toute première conférence à Kiel. Merci à Kuba, Lourdes, Shahin. Merci aux amis du LSV : Konstantinos, Marie et Nadime. Et à ceux rencontrés aux EJCIM : à Alexandra pour ses *cornelamars*, à Vincent, Louis, Félix, Thomas. Merci aux amis de la Cyclofficine d'Ivry, où je suis allé plus d'une fois me changer les idées : Jessica, Una, David, Martin, Nicolas et Paul. Enfin, merci à vous qui êtes venus à la soutenance de ma thèse. Vous avez sans doute décroché en cours de route et êtes donc en ce moment-même en train de finir de lire les remerciements : rassurez-vous, on le fait tous. Si vous vous ennuyez trop, vous pouvez toujours vous amuser à retrouver tous les gens mentionnés ci-dessus dans la grille suivante :

B V F R I E D R I C H R E I V I L O S P
O L B A P O L M K G I E T E V D G L T Z
N E R D N A X E L A X I T H L J P V E E
I W P A W A R E G E K I E S O O Y L F I
T G A D V S O D R N H C L M I M D O A R
R U S E I P A I N I A C I E I T A I N A
A I C N C D A L Z A A M I R F X P S E M
M L A N T I P U O J X L A M T F A A D N
F L L E O V L A L C L E C R N A A M B A
S A N I R A F K A T I L L O I A P N C E
V U J T I D A N T O I N E A V E E R U J
I M R E A G C H A R L E S L Q A I J A C
N E Y F L A I L E R U A J Q C S Y R H O
C B K O N S T A N T I N O S T L U A R A
E V I N I C I U S B U X F I O A R X E C
N H Z R E T E I D E R H N U L L L H X I
T Y E E N A H P E T S A R A O A B U K S
W U I E L L I U O X W D N T Q F O S C S
L O L P A U R E L I E N T J L O U I S E
P B E E R B V F X S A E E N E L Y M R J

Contents

1 Preliminaries **1**
- 1.1 Complexity . . . 1
  - 1.1.1 Classical complexity . . . 1
  - 1.1.2 The problem SAT . . . 5
  - 1.1.3 Counting complexity . . . 8
  - 1.1.4 Parametrized complexity . . . 9
- 1.2 Graphs, hypergraphs and decompositions . . . 12
  - 1.2.1 Generalities on graphs and hypergraphs . . . 12
  - 1.2.2 Graph measures and decompositions . . . 15
  - 1.2.3 Hypergraphs: acyclicity and decompositions . . . 19
- 1.3 Knowledge compilation . . . 25
  - 1.3.1 Generalities . . . 25
  - 1.3.2 Binary decision diagrams . . . 27
  - 1.3.3 DNNF and its restrictions . . . 29

2 Structural restrictions of #SAT **39**
- 2.1 Structure of a CNF-formula . . . 40
  - 2.1.1 Primal and dual graphs . . . 40
  - 2.1.2 Incidence graph and hypergraph . . . 41
  - 2.1.3 Structural restriction of CNF-formulas . . . 42
- 2.2 A first tractable class: disjoint branches . . . 44
  - 2.2.1 Disjoint branches hypergraphs . . . 45
  - 2.2.2 Model counting of disjoint branches formula . . . 46
  - 2.2.3 Finding a disjoint branches decomposition . . . 49
- 2.3 Tractability frontier . . . 57
  - 2.3.1 Parametrized polynomial time algorithms . . . 58
  - 2.3.2 Hardness results . . . 67
  - 2.3.3 Unknown complexity hardness . . . 68

3 Parametrized compilation of CNF-formulas **73**
- 3.1 Compilation of bounded PS-width formulas . . . 74
  - 3.1.1 Shapes . . . 74
  - 3.1.2 Constructing a Structured d-DNNF . . . 76

3.2 Consequences of the compilation algorithm . . . 80
3.2.1 Compilation for other graph measures . . . 80
3.2.2 Solving MaxSAT . . . 82

4 Compilation of β-acyclic formulas 85
4.1 Incomparability with other measures . . . 86
4.2 Structure of β-acyclic hypergraphs . . . 89
4.2.1 Orders . . . 89
4.2.2 Applications . . . 90
4.3 The compilation algorithm . . . 93
4.3.1 Compilation to dec-DNNF . . . 93
4.3.2 Corollaries . . . 97
4.4 Conclusion . . . 98

5 Weighted DP-resolution 99
5.1 DP-Resolution . . . 100
5.1.1 A well-known algorithm for SAT . . . 100
5.1.2 Resolution on 2-CNF . . . 103
5.1.3 Resolution on bounded primal tree width . . . 103
5.1.4 Resolution on β-acyclic formulas . . . 105
5.2 Weighted DP-resolution for β-acyclic instances . . . 105
5.2.1 Encoding SAT using CSP with default values . . . 106
5.2.2 Computing the weight of a chain . . . 109
5.2.3 Computing the weight of β-acyclic instances . . . 112
5.2.4 Runtime of Algorithm 8 . . . 117
5.3 Weighted DP-resolution on general instances . . . 123
5.3.1 Description of the algorithm . . . 123
5.3.2 Cover-width . . . 133
5.4 Conclusion . . . 137

6 Unconditional separations 139
6.1 Preliminaries . . . 141
6.1.1 Certificates . . . 141
6.1.2 Rectangles and covers . . . 144
6.2 Separating CNF-formulas from DNNF . . . 149
6.2.1 A weakly exponential lower bound . . . 150
6.2.2 Lifting known lower bound from communication complexity 151
6.2.3 A family of CNF having no small DNNF . . . 152
6.2.4 Corollaries . . . 157
6.3 Separating structured DNNF from FBDD . . . 158
6.3.1 Rectangle covers of structured DNNF . . . 158
6.3.2 Rectangle covers of graph CNF . . . 160
6.3.3 Separations . . . 161
6.4 Conclusion . . . 163

Introduction

In the Thirties, the notion of computation went from an intuitive definition – everybody *knows* that adding two numbers is a computation – to the following formal definition: a computation *is* what a computer does. Of course, computers did not exist back then, but researchers such as Turing, Church or Kleene had defined formal models of computation, that allowed one to describe a computation – an *algorithm* – that a human, with infinite patience, could perform with a pen and paper. The construction in the Forties of electronic devices that could automatically perform such computations reinforced the idea that this notion was indeed natural and effective. As logic before it, computation became more than a tool used by mathematicians, it became a whole field that can be studied in itself: computer science.

It quickly became clear that not all computable functions are equally hard to compute: some – such as adding two numbers – can be done in a few steps of computation, others – such as finding a divisor greater than one of a given number – apparently need much more time or memory to be computed. In other words, even if a function is computable, it may not be computable in a reasonable amount of time. This led to the introduction of the notion of complexity: the complexity of a function is the minimal amount of resources, such as time or memory, needed to compute it. It was proven early that some functions indeed need a very large amount of resources to be computed. However, such functions were artificial or trivially hard to compute. Even today, very little is known on most functions of interest. Proving that there is no algorithm for a given function using a fixed amount of resources turned out to be a hard mathematical problem. It is however easier to compare the relative complexity of two functions. For example, if we have an algorithm to find the smallest divisor greater than one of an integer, we can easily decide if an integer is prime. Thus, finding divisors is at least as hard as deciding if an integer is prime. This approach has led to a rich and useful theory for classifying problems according to their relative hardness.

The most successful theory in this line of research is arguably the theory of NP-completeness that was introduced by Cook in the early seventies. Intuitively, a problem is in the class NP if its solutions can be checked efficiently. Say, for example, that a traveler wants to visit every city in a country. He knows the distance between the cities and he wants to know if he can do it by driving less than 1000km. If someone shows him a possible tour, he can check by himself if the

proposed itinerary is actually shorter than 1000km. However, finding such a tour is much harder and one of the most famous open question of complexity theory – known as P vs NP – is to understand if this problem can be solved quickly. The problem – known as the traveling salesman problem – has a surprising property: it is NP-complete, meaning that it is at least as hard as any problem in NP. That is, if we know how to solve the traveling salesman problem efficiently, then we can solve every problem of NP efficiently. We know thousands problems of this kind and none of them seems to be solvable efficiently. The common assumption is thus that these problems are hard and, given a problem that we do not know how to solve efficiently, we generally cannot prove that it is hard in itself but we usually can prove that it is at least as hard as the traveling salesman problem, which is used as an evidence that it is indeed hard.

Unfortunately, many problems arising in practice are as hard as the traveling salesman problem. In theory, it is believed that they are not solvable efficiently by a computer in a reasonable amount of time. Surprisingly, efficient software for solving such problems in practice are developed. Of course, such tools do not solve NP-hard problems quickly on all instances, otherwise we would have P = NP. But for many real-life scenarios, they perform well. This difference between theory and practice can be partially explained by the fact that in theory, every input is possible but in practice, the input is not completely arbitrary. For example, a real input of the traveling salesman problem will be an existing country with roads that were constructed by humans following an underlying plan for improving transportation. Such structure may be implicitly used by software to solve hard problems on real life instances. Understanding which kind of structure can or cannot be exploited to solve a particular problem efficiently is the core problem of this thesis.

In the rest of this introduction, we present the questions that are addressed in this thesis together with our contributions. We conclude by giving a detailed overview of each chapter.

CNF-formulas and the problem SAT. In this thesis, we are interested in problems that are NP-complete or harder. All these problems are problems concerning *formulas in conjunctive normal form*, CNF-formulas for short. Let X be a set of variables. A CNF-formula on X is a conjunction of clauses, where a clause is the disjunction of literals, that is, variables or negation of variables of X. In other words, a CNF-formula is a formula of the following form:

$$\bigwedge_{i=1}^{m} C_i \text{ where } C_i = \bigvee_{j=1}^{k_i} \ell_{i,j}$$

where each $\ell_{i,j}$ is either a variable x or the negation of a variable $\neg x$. For instance,

$$F_0 = (x \vee y) \wedge (\neg x \vee \neg y)$$

is a CNF-formula on variables $\{x, y\}$. A CNF-formula F on variables X naturally defines a subset of $\{0,1\}^X$ of *satisfying assignments*: an assignment $\tau \in \{0,1\}^X$ *satisfies* F if when we replace each variable $x \in X$ by its value $\tau(x)$ in F, the resulting boolean formula evaluates to 1. For example, in the previous example, the assignment $\{x \mapsto 0, y \mapsto 1\}$ is a satisfying assignment of F_0 whereas $\{x \mapsto 1, y \mapsto 1\}$ is not.

The problem SAT is the problem of deciding whether a given CNF-formula F has a satisfying assignment. It is the first problem to have been shown to be NP-complete by Cook [Coo71] and Levin [Lev73]. Even if it is unlikely that there exists an efficient algorithm for SAT, this problem is omnipresent in computer science. Despite its simple formulation, it has been used very early [Kar72] to efficiently encode many other natural problems. For example, it has been shown that given a graph G and an integer k, one can write a small CNF-formula F_G such that F_G is satisfiable if and only if G has a k-clique. Over the last thirty years, programs called SAT-solvers such as MiniSAT [ES03], Glucose [AS12, SA09], Chaff or ZChaff [MMZ$^+$01] are constantly improved and optimized to solve SAT in practice. Such programs are often used as black boxes to solve other hard problems that have an efficient encoding into SAT. Almost all of them use clever refinements of a very simple algorithm due to Davis, Putnam, Logeman and Loveland [DLL62, DP60] called DPLL. This algorithm iteratively chooses a variable and assigns it a value in $\{0,1\}$ until it reaches either a satisfying assignment or a contradiction. If a contradiction is found, the algorithm backtracks to a previous choice and changes the chosen value at this point. Modern solvers use smart heuristics on the way they choose variables and learn new clauses derived from the contradictions they have encountered to speed up the computation. Over time, such methods have been improved and the implementations have been made extremely efficient. SAT-solvers today are able to solve industrial instances having several millions of variables and hundred thousands clauses, an impressive performance considering that the problem is NP-hard. The exact reasons why such solvers perform so well in practice are still not well understood but it is generally believed that they implicitly exploit hidden structure in the input [AGCL12]. For instances that do come from the industry, for example, instances from cryptography, SAT-solver indeed fails to find satisfying assignments in a reasonable time, which can be seen as the fact that the underlying structure of such instances is different of the one they were optimized for.

Counting satisfying assignments. One central problem studied in this thesis is the counting version of SAT, the problem #SAT which is the problem of counting the number of satisfying assignments of a given CNF-formula. #SAT is at least as hard as SAT and theoretical evidences suggest that #SAT is actually much harder. Toda's theorem states that with one call to an oracle able to solve #SAT, the entire polynomial hierarchy – a class of problems that contains NP but is believed to be much bigger – can be decided in polynomial time.

In practice too, programs for solving #SAT such as Cachet [SBB+04] or sharpSAT [Thu06] do not perform as well as SAT solvers. Almost all of them are based on a generalization of DPLL for counting. The main difference here to SAT-solvers is that the #SAT-solver cannot stop when it encounters a satisfiable assignment since it has to count them all and the algorithm backtracks much more even if the same heuristics and optimizations as for SAT-solvers are used. Consequently, the size of the instances that can be solved in practice is much smaller than the size of instances solved by SAT-solvers.

One can argue that, in practice, we sometimes do not need the exact number of satisfying assignments and that an approximation would be enough for some applications. Although this is a relevant observation, it has been shown that for all $\epsilon > 0$, computing a $2^{n^{1-\epsilon}}$-approximation of the number of satisfying assignment of a CNF-formula with n variables is as hard as computing this number exactly [Rot96].

Another natural idea is to restrict the type of clauses of the formula in order to find easy cases for solving #SAT, a type of restriction that has been extensively studied for SAT [Sch78]. For example, it is known that SAT is easy if every clause has at most 2 variables [Sch78, APT79]. Similarly, if the formula is monotone, that is it does not use negations, then it is satisfiable since assigning every variable to 1 yields a satisfying assignment. But these restrictions do not give easy classes for #SAT. Roth [Rot96] has shown that approximating the number of satisfying assignments of monotone formulas where each clause is of size at most two is as hard as #SAT. Creignou and Hermann [CH96] have shown that the only restriction on the type of clauses that is easy for exact counting is a case that boils down to counting the number of solutions of a system of $\mathbb{Z}/2\mathbb{Z}$. This suggests that the complexity of #SAT should be studied from another point of view where the input is restricted differently.

Understanding which kind of restrictions yield tractable classes for #SAT and using them to solve other problems on CNF-formulas is the main topic of this thesis. In the rest of this section, we explain in details the questions that are addressed in this thesis and review our contributions.

Structural restrictions of CNF-formulas. In order to understand the complexity of #SAT, a successful line of research, inspired by previous work in database theory [Fag83, BFMY83, GLS01a, GLS99], focuses on so-called *structural restrictions* of CNF-formulas. Here, it is not the type of clauses that is restricted but the way they interact with each other. In order to model such interactions, a graph or a hypergraph is associated to the formula and the tractability of #SAT is addressed with respect to a given class of graphs, that is, we study the tractability of #SAT when it is assumed that the graph associated to the formula is in a given class of graphs. For example, it can be proven that #SAT can be solved in polynomial time if the graph associated to the formula is a tree.

Several notions from graph theory, called *width*, aim to quantify the complex-

ity of the structure of a graph. The most famous is the *tree width* of a graph that intuitively measures how far it is from a tree. Tree width and other width are usually defined with respect to a decomposition of the graph into subgraphs. The complexity of the decomposition is then measured with an integer. These notions have been used successfully to find tractable instances for #SAT. For example, Samer and Szeider have shown that #SAT is tractable when the tree width of the graph of the input formula is bounded by a constant k [SS10]. Many generalizations of this result have followed for graph widths that are more general than tree width [PSS13, SS13]. With Arnaud Durand and Stefan Mengel, we contributed to this knowledge by showing that #SAT is tractable on a class of hypergraphs, called disjoint branches hypergraphs [CDM14]. This result is presented in Chapter 2 together with the first polynomial time recognition algorithm for this class of hypergraphs.

All these structure-based algorithms for #SAT follow a similar approach: the graph of the formula is first decomposed according to the structural restriction and a dynamic programming algorithm is performed along the decomposition, where the number of satisfying assignments of sub-formulas is propagated along the decomposition. The decomposition usually has tree-shape and the dynamic programming usually solve sub-problems associated to each vertex in the tree.

In a recent contribution, Sæther, Telle and Vatshelle [STV14] have given a dynamic programming algorithm for #SAT that works on a very general decomposition. They proved that their algorithm may be used to rediscover known tractable classes for #SAT. With Stefan Mengel and Johann Brault-Baron, we prove [BCM15] that actually every known tractable class for #SAT could be explained in their framework. Indeed, for every graph decomposition for which #SAT is known to be tractable, we explain how to construct a good decomposition for the framework of [STV14]. We present these contributions in Chapter 2.

The singular case of β-acyclic hypergraphs. β-acyclicity is a generalization of acyclicity on graphs to hypergraphs. The complexity of #SAT on β-acyclic formulas – that is, formulas whose associated hypergraph is β-acyclic – was for long a singularity among the structural restrictions of CNF-formula. SAT was known to be tractable on such instances [OPS13] but every attempt for solving #SAT on β-acyclic formulas by using the classical dynamic programming approach failed. The main reason for this failure is that no characterization of β-acyclicity in terms of tree-like decomposition is known. Besides, the proof of tractability of SAT on such formulas is not helpful since it is based on Davis-Putnam resolution [DP60], an algorithm that does not generalize to counting.

With Johann Brault-Baron and Stefan Mengel, we proved the tractability of #SAT on β-acyclic formulas and provided evidences that the classical dynamic programming approach was unlikely to work on such formulas [BCM15]. We explain this deviation from standard techniques by showing that the algorithm from [STV14], which gives a uniform explanation of this approach, in general

takes exponential time on β-acyclic formulas. This result is presented in the first section of Chapter 4 where we construct an explicit family of β-acyclic formulas that cannot be efficiently decomposed to be used with the algorithm from [STV14].

To solve #SAT on β-acyclic instances, we introduced a generalization of Davis-Putnam resolution to counting called the *weighted DP-resolution*. It works on a weighted version of CNF formulas where we have added weights on clauses. In the beginning, the weights are chosen such that the total weight of the formula is equal to its number of satisfying assignments. We then give a procedure that iteratively removes variables from the formula and updates the weights on the clauses to preserve the total weight of the resulting formula. We present a more general result in which our algorithm works on a generalization of CNF-formulas on domains different from $\{0, 1\}$. The algorithm is presented in Chapter 5 together with a new parameter for hypergraphs called the *cover-width* that generalizes β-acyclicity and such that #SAT is still tractable on such hypergraphs by using weighted DP-resolution. We also show that this new width is a natural generalization of tree width to hypergraphs and that both widths coincide when restricted to graphs.

Knowledge compilation. In the framework of *knowledge compilation*, we assume that knowledge on a system is stored into a *knowledge base*. The goal is to query the knowledge base in order to infer new knowledge or to perform reasoning on it. For example, a common query that is often performed for reasoning is the problem of *clause entailment* which is to decide if a knowledge base implies a given clause. The complexity of answering such query strongly depends on the way the knowledge base is encoded. If the knowledge base is encoded as a CNF-formula, then it is NP-complete to answer clause entailment queries, thus, we cannot do it in polynomial time.

The idea of *knowledge compilation* is to preprocess the knowledge base in an *offline phase*, in order to encode it with a representation language that can be used to answer queries efficiently during an *online phase*. Such approaches have been used successfully in practice [ACF10, Par03]. In order to choose the right language for a given application, a systematic study of the properties of different representation languages and the relations between them has been initiated by Darwiche and Marquis [DM02]. This study focuses on three key aspects of representation languages: their succinctness, the queries they support in polynomial time such as deciding satisfiability or counting satisfying assignments and the operations that can be done on them with only a polynomial increase in size such as negation or conjunction.

In this thesis we focus on the particular representation language of DNNF and some of its restrictions. DNNF is a representation language based on boolean circuits with additional properties which ensure that deciding satisfiability is supported in polynomial time. DNNF is a particularly interesting language: it is more succinct than almost every other representation language used in practice and still supports interesting queries in polynomial time.

Our first contribution in knowledge compilation is to show that structural restrictions of CNF-formulas may be successfully used for compilation. More precisely, we show that every known structural restriction for which #SAT is tractable can also be used to compile CNF-formulas into succinct deterministic DNNF, a restriction of DNNF that supports model counting. This result explains why the structure-based algorithms for #SAT can be so easily adapted to solve various problems such as weighted model counting or efficient enumeration by showing that these algorithms may be decomposed into a compilation phase that takes advantage of the structure followed by a query on the compiled representation of the formula. This result is shown in Chapter 3 by proving that the general framework of [STV14] may also be used for compilation. Since this framework encompasses every known tractable class for #SAT, we can show that all these classes can be efficiently compiled into deterministic DNNF. This result was elaborated with Simone Bova, Stefan Mengel and Friedrich Slivovsky and published in [BCMS15]. We also show how to use this result to show the tractability of several other interesting problems on CNF-formulas for these classes such as MaxSAT. Since the case of β-acyclic formulas is not covered by the framework of [STV14], we present in Chapter 4 an algorithm to compile β-acyclic formulas into an even more restrictive version of DNNF called decision DNNF. We rely on new results concerning the structure of β-acyclic hypergraphs that we prove in the second section of Chapter 4.

Our second contribution in knowledge compilation is proving lower bounds on the succinctness of representation languages. In [DM02], the succinctness of CNF and DNNF was shown to be incomparable only under assumptions from complexity theory. We show that this result holds unconditionally. We prove this by making a new connection between knowledge compilation and communication complexity. We show that the communication complexity (in a very general model) of a DNNF is no more than its size. By using known lower bounds on the communication complexity of some functions, we are able to prove the first strong exponential lower bound on the size of every DNNF computing a family of CNF. We also give a family of monotone 2-CNF and prove an exponential lower bound on their communication complexity by using tools from graph theory, making the proof of the lower bound self-contained. Finally, we push the connection with communication complexity further to get new separations left open in [DM02] and reprove known lower bounds from [PD10b] in a new unified framework. These results are presented in Chapter 6 and a different presentation of this results, which does not rely on the connection with communication complexity, may be found in an arXiv preprint [BCMS14].

Overview of the thesis. We now give an overview of the content of each chapter of this thesis. More details are given in the introduction of each chapter.

Chapter 1 contains the basic definitions and properties of the objects that are used in this thesis. We give a brief overview of each area connected to our results

and provide some references.

In Chapter 2, we introduce the notion of structural restrictions of CNF-formulas and give an overview of the known results concerning the complexity of #SAT on such restrictions. We start by defining the existing ways of characterizing the structure of a formula and give examples of such notions. We then present a contribution with Arnaud Durand and Stefan Mengel [CDM14] concerning the complexity of #SAT an so-called disjoint branches hypergraphs. We show that #SAT can be solved efficiently if the underlying hypergraph of the formula is disjoint branches and show that deciding if a hypergraph has this property can be done in polynomial time. We use this results as an illustration of the main techniques that are commonly used for proving such results. We finish the chapter by presenting the known classes for which #SAT can be solved in polynomial time. We review in particular the result work of Sæther, Telle and Vatshelle [STV14] and show how it can be used to rediscover all other known results concerning the tractability of #SAT. We finally review the known hardness results concerning #SAT and conclude the chapter by giving open questions.

In Chapter 3, we show how to leverage structure-based algorithms for #SAT to compilation algorithm into deterministic DNNF. The results presented in this chapter were elaborated with Simone Bova, Stefan Mengel and Friedrich Slivovsky and are published in [BCMS15]. Our compilation algorithm is based on the general algorithm of [STV14] in Chapter 2. This allows us to prove that every known structure-based algorithm for #SAT known so far can actually be lifted to a compilation algorithm.

In Chapter 4, we show that #SAT is tractable on β-acyclic instances by giving an algorithm that compiles β-acyclic formulas into decision DNNF, a restriction of DNNF that supports model counting. The first section of this chapter is dedicated to the proof that the complexity of #SAT on β-acyclic instances cannot be proven by using the algorithm of [STV14] presented in Chapter 3. The second section studies the structure of β-acyclic hypergraphs and shows several useful properties of such hypergraphs. The third and last section presents the compilation algorithm and corollaries on the tractability of several problems related to β-acyclic formulas.

In Chapter 5, we give a simpler algorithm for efficiently solving #SAT on β-acyclic instances that was done in collaboration with Johan Brault-Baron and Stefan Mengel and published in [BCM15]. This algorithm is based on a generalization of Davis-Putnam resolution [DP60]. We recall this algorithm and several result concerning its complexity in the first section. In a second section, we present a generalization of DP-resolution working on β-acyclic formulas only. We prove that our algorithm is sound and runs in polynomial time. Bounding the runtime of the algorithm is done in two steps: we first show that the algorithm does a polynomial number of arithmetic operations. We then show that the size of the numbers involved in these arithmetic operations is polynomially bounded. The second part of the proof is surprisingly much more challenging than the first one and relies on results shown in Chapter 5 concerning the structure of β-acyclic hypergraphs. In the last section, we present a generalized version of the previous

algorithm, the weighted DP-resolution which runs on any CNF-formulas and returns its number of satisfying assignments. We then present a generalization of β-acyclicity, the cover-width, for which weighted DP-resolution runs with a polynomial number of arithmetic operations. We also give additional results on how cover-width compares to other hypergraph widths.

Finally, in Chapter 6, we show, in the first section, a connection between the size of a DNNF and the communication complexity of the function it computes. We use this connection in the second section to construct CNF-formulas that cannot be represented succinctly by DNNF. Our proof settles several open questions concerning knowledge representation [DM02]. Such CNF-formulas can be seen as hard instances for the known algorithm for #SAT. Indeed, Darwiche and Huang [HD05] have observed that every practical tool for solving #SAT was implicitly constructing a succinct DNNF equivalent to the input formula. Moreover, we show in the other chapters that all structured-based algorithms for #SAT can also be turned into compilation algorithm into succinct DNNF. Thus, CNF-formulas having no small DNNF are hard for both practical tools and structure-based algorithms for #SAT. Finally, we use our connection to rediscover known results on a representation language called structured DNNF [PD10b] and reprove, by reusing results in communication complexity and graph theory, separations between different representation languages. We conclude this chapter by giving open questions and perspectives on such techniques for knowledge compilation.

Chapter 1

Preliminaries

This chapter provides the main definitions of the objects that will later appear in this thesis. It also aims at introducing background on the different concepts that lie at the core of our results. Each section presents a different topic and will serve as a reference for the next chapters.

1.1 Complexity

Most of the results presented in this thesis originate from complexity theory. The aim of complexity theory is to understand how much resources – such as time or memory – one needs to solve a problem on a given computational device. This question for a given problem is however never answered with a precise value and it is usually not desired since even the necessary mathematical formalization of the device is already an approximation of what is happening in practice. The Turing machine model for instance fails to capture the complex interactions between the architecture of a processor, the allocation of resources by the OS and the optimizations done by the compiler. However this approximation is often sufficiently accurate to grasp the difficulty of a problem. That is why complexity theorists are mainly interested in asymptotic behavior of the resources needed.

In this thesis, we need concepts from classical complexity but also from more advanced subjects such as counting complexity, that is, the complexity of computing a function expressed as the cardinal of some set or parametrized complexity, that is, the complexity of computing a function when some parameter of the input is assumed to be bounded by a constant. In the following, we only present the main tools we need for stating and proving our results but we pay attention, for each topic, to give to the interested reader good textbook references.

1.1.1 Classical complexity

In this section, we introduce the main concept of computational complexity theory that are widely used in the rest. The interested reader can find details in one of

the most complete book on the subject by Arora and Barak [AB09] or the very-well written book by Papadimitriou [Pap94]. The french reader can also use the marvelous book by Perifel [Per14].

In computational complexity, we are often interested in decision problems that are formalized as subset of $\{0,1\}^* = \bigcup_{n\in\mathbb{N}}\{0,1\}^n$. A set $A \subseteq \{0,1\}^*$ defines the following decision problem: given $x \in \{0,1\}^*$, does $x \in A$? The aim of complexity is to understand how the time or the space needed to solve this problem grows with $|x|$ on a given model of computation.

Model of computation. In this thesis, we use the RAM model of computation, which is a good compromise between the modern architectures and a formalization simple enough to be used in theory. This architecture will proved particularly useful in Chapter 5 where most of our results are stated in terms of number of arithmetic operations, which are easier to deal with in this model than in other models such as Turing machines. Moreover, this model is particularly adapted to enumeration problems that we will briefly mention. We follow the definition of [Str10] which itself follows [Gra96].

Definition 1.1. *A* RAM machine *is an infinite sequence of read-only input register $(I(i))_{i\in\mathbb{N}}$, an infinite sequence of computation register $(R(i))_{i\in\mathbb{N}}$, two special registers A and B and a finite sequence of indexed instructions $P = \{p_1, \dots, p_n\}$, called the program, among the following:*

1. $A \leftarrow I(A)$
2. $A \leftarrow R(A)$
3. $B \leftarrow A$
4. $R(A) \leftarrow B$
5. $A \leftarrow A + B$
6. $A \leftarrow A - B$
7. If$A > 0$ Goto(i), *for* $i \in [n]$
8. Stop

The semantic of a RAM machine is as follows: it executes the instructions one after the other until it reaches the instruction Stop. The $\leftarrow$ denotes the affectation, thus executing the instruction $A \leftarrow B$ for instance will affect the value of register B to register A. If the program reaches the instruction If $A > 0$ Goto(i) and if the value of register A is positive, then it goes to the instruction p_i and resumes its execution from this point.

The input of a machine is a sequence $x = x_1 \dots x_n$ with $x_i \in \{0,1\}$. Each x_i is stored in the input register $I(i)$ and we assume that $I(0) = n$. We assume that

each instruction of a RAM machine costs 1 and we define the runtime $T(M, x)$ of RAM machine M on input x to be the number (possibly infinite) of instructions it executes before reaching the instruction STOP on input x. Given a function $f : \mathbb{N} \to \mathbb{N}$, we say that a machine M runs in time $f(n)$ if the maximal runtime on input of size n is $O(f)$, that is, if the function $n \mapsto \max_{x \in \{0,1\}^n} T(M, x)$ is $O(f)$. The memory used by a RAM machine M on x is the maximal indices of the registers it accesses on input x. For a function $f : \mathbb{N} \mapsto \mathbb{N}$, we say that a machine M runs with memory $f(n)$ is the maximal memory used by M on input of size n is $O(f)$.

A RAM machine M computes a function of $\{0,1\}^* \to \{0,1\}^*$ that associates to x the sequence of register $R(1) \dots R(m)$ when M stops after having been executed on input x. We classify functions depending on their *complexity*, that is, the minimum amount of resources needed to compute them on a RAM machine. This is usually formalized in complexity theory by defining *complexity classes*.

Reductions and complexity classes. A *complexity class* is a class of computable functions that share a common property regarding the resources they need. One of the most natural class is the class of problems that can be solved in polynomial time. We say that a function is computable if there exists a RAM machine M that computes this function. In this thesis, we will mostly be interested in the resources needed to compute a given function. We say that a function g can be computed in polynomial time if there exists a RAM machine M that computes g in time $p(n)$, for p a polynomial.

Definition 1.2. *The class* P *is the class of decision problems* $L \subseteq \{0,1\}^*$ *that can be decided by a RAM machine in time* $p(n)$ *for* p *a polynomial.*

The class P is often considered as the class of easy problems. This is arguably not true since a problem solvable in time $n^{10^{30}}$ or in time n with a hidden constant of 2^{10000} is in P but certainly not that easy to solve. In this thesis however, one of our goal will be to understand for which inputs a problem that is unlikely to be in P can still be solved in polynomial time and for which inputs the problem is unlikely to be solved in polynomial time. We aim to understand where the frontier between the inputs that can be solved in polynomial time and those which cannot lies. In this setting, it makes sense to consider P as the class of easy problems.

Another well-known class is the class NP of problems. Historically, NP problems were introduced as problems that could be solved by non-deterministic machines in polynomial time. In this thesis, we will not need to introduce non-determinism and we use the following characterization of NP:

Definition 1.3. *A problem* L *is in* NP *if and only if there exists a polynomial* p*, a machine* M *in time* p *such that for every* $x \in \{0,1\}^*$*,*

$$x \in L \Leftrightarrow \exists y \in \{0,1\}^{p(|x|)}, M(x, y) = 1.$$

Intuitively, a problem L is in NP if for every member $x \in L$, there exists a proof y of the fact that $x \in L$ that can be checked in polynomial time. Clearly, from the definition, if a problem is in P, then it is in NP as well, that is, $\mathsf{P} \subseteq \mathsf{NP}$.

Many practical problems are in the class NP. For example, consider the problem of finding a planning for a university satisfying a set of constraints concerning the available rooms, the disponibility of teachers etc. It is easy to check if a given planning meets these constraints by simply checking that each one of them is satisfied. Such problem are typical instances of NP problems. The problem of finding such planning however seems much more difficult. One of the main open question in computer science is to know if every NP problems can be solved in polynomial time. In other words, is $P \neq \mathsf{NP}$? Even if this is widely believed that $\mathsf{P} \neq \mathsf{NP}$, no one has yet been able to prove this separation.

In order to classify problems depending on their hardness, it is crucial to have tools to compare such hardness. We introduce the notion of polynomial-time (Karp) reduction from one a problem to another.

Definition 1.4. *A decision problem $L_1 \subseteq \{0,1\}^*$ is* polynomial time reducible *to a decision problem $L_2 \subseteq \{0,1\}^*$, denoted by $L_1 \leq_p L_2$, if there exists a function $f : \{0,1\} \to \{0,1\}$, computable in polynomial time, such that for every $x \in \{0,1\}^*$, $x \in L_1$ if and only if $f(x) \in L_2$.*

Keeping in mind that the polynomial time computable functions are the easy to compute functions, polynomial time reductions may be seen as a way of comparing the hardness of different decision problems. Indeed, if $L_1 \leq_p L_2$, then L_2 is harder to solve than L_1 in the sense that if there exists a polynomial time algorithm for L_2 (that is if L_2 is "easy"), then there exists a polynomial time algorithm for L_1 as well. It is easy to see that polynomial time reductions are transitive since the composition of two polynomial time computable functions is still computable in polynomial time. Moreover, classes such as P and NP are closed by polynomial time reductions, that is:

Theorem 1.5. *Let $L_1, L_2 \subseteq \{0,1\}^*$ be decision problems. It holds that:*

- *if there exists $L \subseteq \{0,1\}^*$ such that $L_1 \leq_p L$ and $L \leq_p L_2$ then $L_1 \leq_p L_2$,*
- *if $L_1 \leq_p L_2$ and $L_2 \in \mathsf{P}$ then $L_1 \in \mathsf{P}$ and,*
- *if $L_1 \leq_p L_2$ and $L_2 \in \mathsf{NP}$ then $L_1 \in \mathsf{NP}$.*

Using this, we can compare a decision problem with a class of problems. Indeed, if C is a class of problems, we say that a decision problem L is C-hard if it is harder than every problem in C. In other words, if for every $L' \in C$, $L' \leq_p L$. It means that if we are able to solve L quickly, then every problem of C are also easy to solve. This led to the notion of NP-completeness, introduced by Cook [Coo71]:

Definition 1.6. *A decision problem L if NP-hard if for every $L' \in \mathsf{NP}$, it holds that $L' \leq_p L$. It is NP-complete if L is NP-hard and $L \in \mathsf{NP}$.*

Intuitively, NP-complete problems are the hardest problems of NP. If one can solve quickly an NP-complete problem, then we could solve every other problems of NP in quickly and we would have P = NP. What makes the notion so important is that there actually exist NP-complete problems and these problems arise naturally in various scientific fields. Of course, since P $\neq$ NP is still an open question, no NP-complete problem is known to be computable in polynomial time, nor it is known not to be. The fact that numerous problems, formalized in a hundred of different ways, are NP-complete and no one has found any efficient algorithms for one of them is one of the reasons it is believed that P $\neq$ NP. The fact that a problem is NP-complete is usually used as strong evidence it does not have a polynomial time algorithm.

1.1.2 The problem SAT

The first natural problem to have been shown to be NP-complete is the problem SAT. It was shown independently by Cook [Coo71] and Levin [Lev73]. In the rest of this section, we define the problem SAT and states the Cook-Levin Theorem. Since SAT (and its counting version) is the main problem we study in this thesis, we also introduce several notations on CNF-formulas in the end of this section.

Let X be a finite set of variables. A *literal* is either a variable $x \in X$ or its negation $\neg x$. A *clause* on variables X is a finite set of literals on variables X. A formula F in *conjunctive normal form* , CNF-formula for short, on variables X is a finite set of clauses. A mapping $\tau : X \to \{0, 1\}$ is a *satisfying assignment* of F if for every clause $C \in F$, there exists a literal $\ell \in C$ such that $\tau(\ell) = 1$ where $\tau(\ell)$ is defined to be $\tau(\ell) = \tau(x)$ if ℓ is the variable x and $\tau(\ell) = 1 - \tau(x)$ if $\ell = \neg x$. A formula F is said to be *satisfiable* if there exists a satisfying assignment of F. It is said *unsatisfiable* otherwise.

A clause can be seen as the disjunction of its literals since an assignment satisfies a clause if that at least one literal is satisfied in each clause. A CNF-formula can be seen as the conjunction of its clauses since an assignment satisfies a formula if all its clauses are satisfied. We denote conjunction as $\wedge$ and disjunction as $\vee$. Following this observation, we will often denote CNF-formula as:

$$\bigwedge_{C \in F} \bigvee_{\ell \in C} \ell.$$

For example,

$$F = (x \vee \neg y \vee t) \wedge (\neg x \vee z \vee w) \wedge (\neg t \vee \neg w)$$

is a CNF-formula whose clauses are $\{x, \neg y, t\}$, $\{\neg x, z, w\}$ and $\{\neg t, \neg w\}$. The mapping $\{x \mapsto 1, y \mapsto 1, w \mapsto 1, t \mapsto 0, z \mapsto 1\}$ is a satisfying assignment of F whereas $\{x \mapsto 0, y \mapsto 1, w \mapsto 1, t \mapsto 0, z \mapsto 1\}$ is not.

The problem SAT can be formulated as follows: given a CNF-formula F on variables X, is F satisfiable? Until now, we have only mentioned decision problems to be subset of $\{0, 1\}^*$ which is not how we just defined SAT. To fall in our

theoretical framework, we should observe that a formula F can be encoded by an element of $\{0,1\}^*$ of size linear in $(\sum_{C\in F}|C|)\log(n)$ where n is the size of the variable set X. See [AB09] for more details on how to encode problems.

It is easy to see why SAT is in NP. Indeed, given a formula, a short proof of the fact that the formula is satisfiable could simply be a satisfying assignment of F. A trivial algorithm to solve SAT is to bruteforce every possible assignment $\tau : X \to \{0,1\}$ and to check whether it satisfies F or not. If F is not satisfiable, such an algorithm would require at least $2^{|X|}$ steps, at least one per assignment τ which is not a polynomial time algorithm. Actually, finding a polynomial time algorithm for SAT is very unlikely. The Cook-Levin Theorem states that SAT is NP-complete:

Theorem 1.7 ([Coo71, Lev73])**.** SAT *is* NP*-complete.*

Notations and definitions. We introduce here several notations concerning CNF-formulas and assignments that will be used later in this thesis. We denote by $\perp$ the *empty clause* , that is, the clause that has no literal. Observe that by definition of a satisfying assignment, the empty clause cannot be satisfied, thus if $\perp \in F$ for F a CNF-formula, then F is not satisfiable. It is different from the empty formula $\emptyset$, that is the formula that has no clause, which is always satisfiable since for every assignment, every clause of F are satisfied.

Given a clause C, we denote by $\mathsf{var}(C)$ the set of variables of a clause, that is, $\mathsf{var}(C) = \{x \mid x \in C \text{ or } \neg x \in C\}$. We denote by $\mathsf{var}(F) = \bigcup_{C\in F}\mathsf{var}(C)$ the set of variables that appears in F. Observe that if F is a CNF-formula on variable X, we only have $\mathsf{var}(F) \subseteq X$ since F may not mention some variables of X.

Given a CNF-formula F on variables X, we call τ a partial assignment if τ is a mapping $Y \to \{0,1\}$ for $Y \subseteq X$. We call Y the support of τ and we denote it by $\mathsf{supp}(\tau)$. Let F be a CNF-formula and τ a partial assignment. For a clause C, we say that τ satisfy C, denoted by $\tau \models C$, if there exists $\ell \in C$ such that $\tau(\ell) = 1$. We denote by $\tau \not\models C$ otherwise. A clause is said to be *tautological* if there exists a variable x such that $x \in C$ and $\neg x \in C$. A tautological clause is satisfied by every assignment so we usually assume that they are automatically removed from F. Observe that if C is not tautological, then there exists exactly one assignment $\tau_C : \mathsf{var}(C) \to \{0,1\}$ such that $\tau_C \not\models C$. We will often refer to such assignment as the *counter-example* of C. For example, if $C = x \vee y \vee \neg z$ then:

- $\mathsf{var}(C) = \{x, y, z\}$,
- $\tau = \{x \mapsto 0, y \mapsto 1\} \models C$,
- $\tau' = \{x \mapsto 0, z \mapsto 1\} \not\models C$ and,
- $\tau_C = \{x \mapsto 0, y \mapsto 0, z \mapsto 1\} \not\models C$ and is the only assignment of $\mathsf{var}(C)$ that does not satisfy C.

Similarly, we denote by $\tau \models F$ if for every $C \in F$, $\tau \models C$. Observe that τ may be extended to a satisfying assignment of F by choosing the values of the remaining variables arbitrarily. Moreover, every satisfying assignment τ for F verifies $\tau \models F$.

We denote by $F[\tau]$ the CNF-formula on variables $X \setminus Y$ obtained by removing the clauses of F that are satisfied by τ and removing the literals whose value is fixed by τ from clauses. For example, if $F = (x \vee \neg y \vee t) \wedge (\neg x \vee z \vee w) \wedge (\neg t \vee \neg w)$ and $\tau = \{x \mapsto 1\}$ then $F[\tau] = (z \vee w) \wedge (\neg t \vee \neg w)$. We denote by $\mathsf{sat}(F)$ the set of its satisfying assignments and by $\#F = |\mathsf{sat}(F)|$. If F' is a CNF-formula on variables X, we denote by $F \equiv F'$ if $\mathsf{sat}(F) = \mathsf{sat}(F')$. A CNF-formula is said to be *monotone* if it does not have negation. Observe that the boolean function defined by a monotone CNF-formula is actually increasing, thus monotone.

Let $\tau_1 : Y_1 \to \{0,1\}$ and $\tau_2 : Y_2 \to \{0,1\}$ be two assignments. We denote say that τ_1 and τ_2 are compatible, denoted by $\tau_1 \simeq \tau_2$, if for every $y \in Y_1 \cap Y_2$, $\tau_1(y) = \tau_2(y)$. If $\tau_1 \simeq \tau_2$, we denote by $\tau_1 \cup \tau_2$ the mapping from $Y_1 \cup Y_2$ to $\{0,1\}$ defined by

$$(\tau_1 \cup \tau_2)(y) = \begin{cases} \tau_1(y) \text{ if } y \in Y_1 \\ \tau_2(y) \text{ if } y \in Y_2 \end{cases}$$

Given $\tau : X \to \{0,1\}$ and $Y \subseteq X$, we denote by $\tau|_Y$ the restriction of τ on Y, that is, the mapping $\tau' : Y \to \{0,1\}$ such that $\tau' \simeq \tau$.

We denote by

$$\mathsf{size}(F) = (\log |X|) \sum_{C \in F} |C|.$$

Observe that $\mathsf{size}(F)$ is proportional to the size of a reasonable encoding of CNF-formulas on a RAM machine. We often use $|F|$ too which is the number of clauses in F since F is a set of clauses, it is different from $\mathsf{size}(F)$. If every clause of F are of size at most k for $k \in \mathbb{N}$, then we say that F is a k-CNF. The problem SAT where the input are restricted to k-CNF is called k-SAT .

DNF-formulas. We briefly mention a normal form of formulas that is closely related to CNF-formulas called DNF-formulas. A DNF-formula D is a disjunction of conjunction of literals, that is:

$$D = \bigvee_i \bigwedge_j \ell_{i,j}.$$

Observe that the negation of a CNF-formula may naturally be rewritten as a DNF by using the De Morgan's laws $\neg(a \vee b) = \neg a \wedge \neg b$ and $\neg(a \wedge b) = \neg a \vee \neg b$. However, it is easy to see that there exists CNF-formula F such that every equivalent DNF-formula is of size exponential in the size of F. For example, it is easy to see that every DNF-formula equivalent to

$$F = \bigwedge_{i=1}^{n} (x_i \vee y_i)$$

must be of size at least 2^n.

1.1.3 Counting complexity

One of our main focus in this thesis are not decision problems as we have defined them in the last section but *counting problems*, that are, problems where our goal is to count the number of solutions. For example, our input could be a linear system over a finite field, and the problem would be to count the number of solutions of the system. Such problems do not fall in the scope of the previously defined classes since we do not want to decide if an element is a solution to a problem but we want to actually compute a function $f : \{0,1\}^* \to \{0,1\}^*$. The class of functions computable in polynomial time is called FP, for Functional P.

Definition 1.8. *A function $f : \{0,1\}^* \to \{0,1\}^*$ is in FP if there exists a RAM machine that computes f in polynomial time.*

As for decision problems, we want to be able to compare the hardness of problems and to find a way of stating that some functions are unlikely to have polynomial time algorithm. In the following, we define an analog of NP for counting problems, the class $\#\mathsf{P}$, and define analog notions of completeness and reductions.

Definition 1.9. *The class $\#\mathsf{P}$ is the class of function $f : \{0,1\}^* \to \{0,1\}^*$ such that there exists a polynomial time machine M such that for every $x \in \{0,1\}^*$, $f(x) = |\{y \mid M(x,y) = 1\}|$.*

A function of $\#\mathsf{P}$ may be understood as a counting function f such that $f(x)$ is the number of witnesses that x is in a language of NP. Most of the problems of NP have a natural counting problem associated to them. For example, one problem that is central in this thesis is the problem $\#\mathsf{SAT}$ that can be formulated as follows: given a CNF-formula on variables X, return the number of satisfying assignments $\tau : X \to \{0,1\}$ of F. $\#\mathsf{SAT}$ is in $\#\mathsf{P}$: we can construct a polynomial time machine M such that $M(F,\tau) = 1$ if and only if τ is a satisfying assignment of F. Then $\#\mathsf{SAT}(F) = |\{y \mid M(x,y) = 1\}|$. Clearly, if we can count the number of satisfying assignments of a CNF-formula, then we can also decide if there exists a satisfying assignment of the formula. Thus, it is even less likely that there exists a polynomial time algorithm for $\#\mathsf{SAT}$ than it is for SAT.

The class $\#\mathsf{P}$ was first introduced by Valiant [Val79b] to study the complexity of counting problems. Of course, counting problems built from NP-complete problems as $\#\mathsf{SAT}$ are hard instances of this class. However, he shows in [Val79b] that there are functions that can be defined as counting the number of solution of problems in P that lead to problems as hard as $\#\mathsf{SAT}$. Such results are proven through a notion of reduction for the class $\#\mathsf{P}$ which gives then a natural notion of $\#\mathsf{P}$-completeness. The notion of reduction we use for $\#\mathsf{P}$ relies on the notion of oracle:

Definition 1.10. *Let $g : \{0,1\}^* \to \{0,1\}^*$. A RAM machine M with oracle g is a RAM machine with the extra-instruction $R(A) \leftarrow g(B)$.*

The runtime of a RAM machine with oracle is still the number of instruction it executes before reaching the STOP instruction. In other word, we assume that the extra-instruction is executed in unit time.

Definition 1.11. *A function $f : \{0,1\}^* \to \{0,1\}^*$ is polynomial time Turing reducible to a function $g : \{0,1\}^* \to \{0,1\}^*$, denote by $f \leq_T g$ if there exists a RAM machine M with oracle g that computes f in polynomial time.*

Turing reductions are transitives and the class #P is closed by Turing reductions:

Theorem 1.12. *Let $f, g : \{0,1\}^* \to \{0,1\}^*$. It holds that:*

- *if there exists $h : \{0,1\}^* \to \{0,1\}^*$ such that $f \leq_T h$ and $h \leq_T g$ then $f \leq_T g$ and,*
- *if $f \leq_T g$ and $g \in \#\mathsf{P}$ then $f \in \#\mathsf{P}$.*

Definition 1.13. *A function g is $\#\mathsf{P}$-hard if for every function $f \in \#\mathsf{P}$, it holds $f \leq_T g$. A function g is $\#\mathsf{P}$-complete if it is $\#\mathsf{P}$-hard and if $g \in \#\mathsf{P}$.*

Unsurprisingly, the Cook-Levin Theorem generalizes to the counting case.

Theorem 1.14 ([Val79b])**.** #SAT *is $\#\mathsf{P}$-complete.*

What is more surprising is that there exists counting problems constructed from decision problems in P that are as hard as #SAT. This was proven in [Val79b] by Valiant who shows that computing the permanent $\mathsf{PER}(M) = \sum_\sigma \prod_{i=1}^n M[i, \sigma(i)]$ of a $n \times n$ matrix M is #P-complete whether deciding if there exists a permutation σ of $[n]$ such that $\prod_{i=1}^n M[i, \sigma(i)] = 1$ can be done in polynomial time.

1.1.4 Parametrized complexity

The results we have recalled on complexity theory until then were focused on classifying problems depending on their complexity. We have shown that some problems were harder than others. In the framework we have presented until then, a hard problem is a problem that we cannot solve efficiently on *every* input. But even a hard problem may be easy on some particular instance. For example, it is well-known that the problem 2-SAT can be solved with a linear time algorithm (see [APT79] or Chapter 5). Thus SAT on such instances is easy to solve.

The aim of parametrized complexity is to understand the complexity of a problem depending on some parameter of the input. It can be argued that this idea is quite natural and that algorithms complexity has been expressed as function of parameters without relying on complexity theory. The role of parametrized complexity is somewhat similar to the one of complexity theory. Designing better algorithms and analyzing their runtime was done long before complexity classes were defined. Complexity theory is a framework to assert the hardness of problems.

In this section, we present a framework due to Downey and Fellows [DF12] to analyze the complexity of problems together with a parameter, that is, we give the necessary tools to compare the quality of different parameters for a same problem and to provide evidences of the hardness of a problem for a given parameter. We only present the basic tools we need in this thesis but the domain of parametrized complexity is much richer than this. The interesting reader may refer to the book of Flum and Grohe [FG06] that proposed a good overview of the subject.

Definition 1.15. *A* parametrization *is a mapping* $\kappa : \{0,1\}^* \to \mathbb{N}$ *that can be computed in polynomial time. A* parametrized problem *is a pair* (L, κ) *where* $L \subseteq \{0,1\}^*$ *and* κ *is a parametrization.*

For example, a parametrization of CNF-formula can be the function $\kappa = \mathsf{var}$ that associates to an encoding of a CNF-formula F the number of variables of F and the parametrized problem associated is $(\mathsf{SAT}, \mathsf{var})$. Observe that by brute forcing over every assignment, we can decide if F is satisfiable in time $O(2^{\mathsf{var}(F)} \cdot |F|)$.

Given a parametrized problem, we will be mostly interested in finding polynomial time algorithms for inputs where the parameter is bounded. The class of such problem is known as XP:

Definition 1.16. *The class* XP *is the class of parametrized problems* (L, κ) *such that there exists a computable function* f *and a RAM machine* M *deciding* L *in time* $n^{f(k)}$.

We can already use the tools from classical complexity theory to show that some problems are unlikely to be in XP. For example, it can be shown that 3-SAT is NP-complete. Thus:

Theorem 1.17. *If* $\mathsf{P} \neq \mathsf{NP}$*, then the parametrized problem* $(\mathsf{SAT}, \mathsf{cla})$ *is not in* XP*, where* cla *is the function which associates to a formula the size of its biggest clause.*

The class XP is very large and the runtime of M may depend differently on the parameter. For example, $(\mathsf{SAT}, \mathsf{var})$ is in XP since we have an algorithm running in time $O(2^{\mathsf{var}(F)} \cdot |F|)$. The dependence on the size of F here is linear whatever is the size of the parameter. But a parametrized problem (L, κ) that can be solved in time $|x|^{\kappa(x)}$ is in XP. We see here that the dependence on the parameter is much worse than before and that such algorithm may be stuck already for small values of n. One of the main success of parametrized complexity is to provide a fine hierarchy of problems depending on how the value of the parameter acts on the complexity of the algorithm. Usually, the dependence on the parameter that is desirable is the following:

Definition 1.18. *The class of* fixed-parameter tractable *problems, denoted by* FPT *is the class of parametrized problem* (L, κ) *such that there exists a machine* M*, a polynomial* p *and a computable function* $f : \mathbb{N} \to \mathbb{N}$ *that decides* L *in time* $f(\kappa(n)) \cdot p(n)$.

From the previous observation, (SAT, var) is fixed-parameter tractable. Again, the notion of easiness is relative here and if the computable function f of the definition is $O(2^{2^{2^k}})$ then the parameter is certainly not relevant for practical purposes. While it is better than having an algorithm with $O(n^\kappa)$ complexity such algorithm may take already too much time for small inputs.

In the rest of this section, we will present a tool analog to NP-completeness or #P-completeness that can be used to provide strong evidences that a parametrized problem is unlikely to have an FPT algorithm. We start by introducing a notion of FPT-reduction:

Definition 1.19. *Let (L, κ) and (L', κ') be parametrized problems. The problem (L, κ) is FPT-reducible to (L', κ'), denoted by $(L, \kappa) \leq_{\mathsf{FPT}} (L', \kappa')$, if there exists a mapping $f : \{0,1\}^* \to \{0,1\}^*$ such that:*

- *for every $x \in \{0,1\}^*$, $x \in L$ if and only if $f(x) \in L'$,*
- *f is computable by an FPT-algorithm with parameter κ and,*
- *there exists a computable function $g : \mathbb{N} \to \mathbb{N}$ such that $\kappa'(f(x)) \leq g(\kappa(x))$.*

The class FPT is closed by FPT-reduction:

Theorem 1.20. *Let (L, κ), (L, κ') be parametrized problems such that $(L, \kappa) \leq_{\mathsf{FPT}} (L', \kappa')$. If $(L', \kappa') \in \mathsf{FPT}$, then $(L, \kappa) \in \mathsf{FPT}$.*

Finally, we introduce the class W[1]. There are several characterizations of W[1] and we let the interested reader refer to Chapter 5 of [FG06] for an exhaustive presentation of W[1]. We present here only the definition we will need in this thesis. The problem k-CLIQUE is the parametrized problem of deciding, given a graph $G = (V, E)$ and an integer k, if there exists a clique of size k in G, that is a set $v_1, \ldots, v_k$ of V such that $\{v_i, v_j\} \in E$ for every $i \neq j$ where k is the parameter (see Section 1.2 for more details on graphs). More formally the parameter is $\kappa((G, k)) = k$. Observe that by brute forcing over every subset of size k of V gives a $O(k^2\binom{n}{k})$ algorithm which is not FPT.

Definition 1.21. *W[1] is the class of problems (L, κ) that are FPT-reducible to k-CLIQUE. A problem (L, κ) is W[1]-hard if k-CLIQUE $\leq$ FPT(L, κ).*

As for $\mathsf{P} \neq \mathsf{NP}$, it is believed that $\mathsf{W}[1] \neq \mathsf{FPT}$ but these hypothesis are not known to be equivalent. Having a W[1]-hard problem is an evidence that it is unlikely to have FPT-algorithm. This hypothesis is not equivalent to $\mathsf{P} \neq \mathsf{NP}$ and no implication are know between the two hypothesis. However, we can show that if $\mathsf{W}[1] = \mathsf{FPT}$, then it would give an algorithm that runs in $2^{o(n)}$ to solve 3-SAT, which would be a breakthrough in our understanding of the class NP.

Until now we have defined parametrized classes only for decision problems. In this thesis, we will essentially present parametrized algorithms for counting problems. A theory for parametrized counting complexity has been proposed

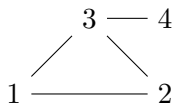

Figure 1.1: A graph

by Flum and Grohe [FG04] and by McCartin [McC02] to prove the hardness of parametrized counting problem. In this thesis however, every hard parametrized counting problem we are interested in are already W[1]-hard for decision, thus we do not need the whole theory. The interested reader may refer to Chapter 14 in [FG06] for more details on the subject. We only define parametrized counting problems:

Definition 1.22. *A* parametrized counting problem *is a pair* (c, κ) *where* $c : \{0,1\}^* \to \mathbb{N}$ *and* κ *is a parametrization. A parametrized counting problem* (c, κ) *is computable in* FPT *time if there exists a computable function* f*, a polynomial* p *and a machine* M *that computes* c *in time* $f(\kappa(n)) \cdot p(n)$*.*

1.2 Graphs, hypergraphs and decompositions

Graphs and hypergraphs are common objects to represent the relations between the elements of a finite set. In this section, we define these objects and introduce the notations we will use toward this thesis. We also introduce the notion of graph and hypergraph decompositions, a powerful tool to study the structure of graphs. For more details on graphs, the interested reader may refer to [Die12].

1.2.1 Generalities on graphs and hypergraphs

Graphs. A graph $G = (V, E)$ is given by a finite set V and a set $E \subseteq \{\{u, v\} \mid u \in V, v \in V, u \neq v\}$. We call the *vertices* of G the elements of V and the *edges* of G the elements of E. An edge $e = \{u, v\}$ is uniquely defined by its *endpoints* u and v. A graph is usually represented graphically as in Figure 1.1.

The neighborhood of a vertex $u \in V$, denoted by $\mathcal{N}(u)$, is defined to be the vertices which are connected to u by an edge. Formally, $\mathcal{N}(u) = \{v \in V \mid \{u, v\} \in E\}$. The (open) neighborhood of a set $W \subseteq V$, denoted by $\mathcal{N}(W)$, is defined to be the neighbors of elements of W that are outside W. Formally, $\mathcal{N}(W) = \bigcup_{w \in W} \mathcal{N}(w) \setminus W$. The degree of a vertex u, denoted by $d(u)$, is the size of its neighborhood $d(u) = |\mathcal{N}(u)|$. The degree of a graph is the maximal degree of its vertices.

We now define the main notions on graphs we need in this thesis. A *path* of length k from $u \in V$ to $v \in V$ is a sequence of distinct vertices $v_1, \ldots, v_{k+1}$ such that $v_1 = u$, $v_{k+1} = v$ and such that for all $i \leq k$, $\{v_i, v_{i+1}\} \in E$. A *cycle* is a path of length $k > 1$ from a vertex $u \in V$ to u itself. In Figure 1.1, $(1, 3, 4)$ is a path of length to from 1 to 4 and $(1, 2, 3, 1)$ is a cycle. A graph is *connected* if for

every $u \in V$ and $v \in V$, there exists a path from u to v. A *connected component* is a set $W \subseteq V$ of vertices such that for every $u \in W$ and $v \in V$, there exists a path from u to v if and only if $v \in W$.

A *vertex cover* is a subset $C \subseteq V$ of vertices such that for every $e \in E$, $C \cap e \neq \emptyset$, that is, every edge of G has at least one endpoint in C. In particular, V is a vertex cover of G. In Figure 1.1, $\{1, 3\}$ is a vertex cover.

A *matching* is a subset $M \subseteq E$ of edges such that for every $e, f \in M$, if $e \neq f$, then $e \cap f = \emptyset$, that is, e and f do not share any endpoints. In Figure 1.1, $\{\{1, 2\}, \{3, 4\}\}$ is a matching.

We call a graph $G' = (V', E')$ a *subgraph* of G if $V' \subseteq V$ and $E' \subseteq E'$. G' is said to be *induced* if $E' = \{\{u, v\} \in E \mid u \in V', v \in V'\}$. Given $V' \subseteq V$ a subset of vertices, we denote by $G[V']$ the subgraph induced by V', that is, the graph $G' = (V', E')$ with $E' = \{\{u, v\} \in E \mid u \in V', v \in V'\}$.

A *clique* is a graph $G = (V, E)$ such that for every $u \neq v \in V$, $\{u, v\} \in E$. For $n \in \mathbb{N}$, we denote by K_n the n-clique (also called the n-complete graph) defined as $K_n = ([n], \{\{i, j\} \mid i < j \leq n\})$. For example, the graph induced by $\{1, 2, 3\}$ in Figure 1.1 is a 3-clique.

A graph $G = (V, E)$ is *bipartite* if there exists a partition X, Y of V such that for all $e = \{u, v\} \in E$, $u \in X$ and $v \in Y$. X and Y are called the *colors* of G. Given X, Y two disjoint sets and $E \subseteq X \times Y$, we denote by (X, Y, E) the corresponding bipartite graph. Given a graph $G = (V, E)$ and two subsets $X, Y \subseteq V$ such that $X \cap Y = \emptyset$, we denote by $G[X, Y]$ the bipartite graph induced by X and Y defined as $(X, Y, \{e \in E \mid e \cap X \neq \emptyset, e \cap Y \neq \emptyset\})$. Intuitively, $G[X, Y]$ is the bipartite subgraph of G where we have kept only the edges that have one endpoint in X and one endpoint in Y.

Trees. Trees are an important class of graphs and are omnipresent in this thesis since we intensively use them for graph and hypergraph decompositions. A *tree* is defined to be a connected acyclic graph, that is, a graph that has no cycle. A *forest* is defined to be an acyclic graph. By definition, the connected component of a forest are trees. A *leaf* is a vertex of degree 1. Every tree having more than one node has at least two leaves. The set of leaves of a tree T is denoted by $L(T)$.

A *rooted tree* is defined to be a tree T and a distinguished vertex r of T called the *root*. Rooted trees are very handy to deal with since they naturally induce a preorder on the vertices. Indeed, since T is acyclic and connected, for every u, v there exists exactly one path from u to v (remember that a path is a sequence of *distinct* vertices). Thus, for every $u \in V$, there exists a unique path from the root r to u.

For $u \neq r$, we call the *father* of u the vertex that appears before u in the path from r to u. The *children* of u are defined to be the neighbors of u that are not its father. We say that a vertex u is an *ancestor* of v if the path from r to v goes through u. We say that a vertex u is a *descendant* of v if v is an ancestor of u. If every vertex of T has at most 2 children, we say that T is *binary*.

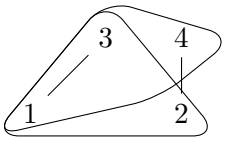

Figure 1.2: A hypergraph

For a vertex v of T, the *subtree of T rooted in v*, denoted by T_v, is defined to be the rooted subtree induced by v and its descendants, with v as the root. Observe that if $v_1, \ldots, v_k$ are the children of v, T_v is the union of v and $T_{v_1}, \ldots, T_{v_k}$ and if v is a leaf different from the root, then T_v is the graph having one vertex $\{v\}$ and no edge.

Hypergraphs. A graph can be seen as a set of sets of size 2, the edges. A natural generalization of graphs is to allow the edges to hold more than two elements. A hypergraph $\mathcal{H}$ is a finite set of sets. An set $e \in \mathcal{H}$ is called an *edge* or a *hyperedge*. The vertices of $\mathcal{H}$, denoted by $V(\mathcal{H})$, is defined to be the union of the sets in $\mathcal{H}$, that is, $V(\mathcal{H}) = \bigcup_{e \in \mathcal{H}} e$. A hypergraph is usually represented graphically as in Figure 1.2.

Many notions of graphs naturally translate in the framework of hypergraphs. A *path* of length k from $u \in V(\mathcal{H})$ to $v \in V(\mathcal{H})$ is a sequence of distinct vertices and edges of $\mathcal{H}$ $v_1, e_1, \ldots, v_k, e_k, v_{k+1}$ such that $u = v_1$, $v = v_{k+1}$ and for all $i \leq k$, $e_i \in \mathcal{H}$, $v_i \in e_i$ and $v_{i+1} \in e_i$. A hypergraph is *connected* if for every $u, v \in V(\mathcal{H})$, there exists a path from u to v. A *connected component* of $\mathcal{H}$ is a set $W \subseteq V(\mathcal{H})$ such that for every $u \in W$ and $v \in V(\mathcal{H})$, there exists a path from u to v if and only if $v \in W$.

The notion of cycle is more delicate to generalize to hypergraphs. We could define a cycle to be a path of length $k > 1$ from u to u but this definition is rarely relevant for applications. For example, $(1, \{1, 3\}, 3, \{1, 3, 4\}, 1)$ is a cycle of the hypergraph represented in Figure 1.2 but it is "simpler" than the cycle $(1, \{1, 2, 3, 4\}, 4, \{2, 4\}, 2, \{1, 2, 3\}, 1)$. In Section 1.2.3, we will formally define notions of acyclicity for hypergraphs that formalize this notion of cycles.

One can naturally associate graphs to hypergraphs that describe their structure. The *primal graph* of a hypergraph $\mathcal{H}$, denoted by $\mathcal{G}_{\mathsf{prim}}(\mathcal{H})$, is defined to be the graph having vertices $V(\mathcal{H})$ where each edge $e \in \mathcal{H}$ is replaced by the clique on the vertices in e. Formally, $\mathcal{G}_{\mathsf{prim}}(\mathcal{H}) = (V(\mathcal{H}), \{(u, v) \mid \exists e \in \mathcal{H}, \{u, v\} \subseteq e\}$. Much structure is lost in the primal graph. For example, if $V(\mathcal{H}) \in \mathcal{H}$, then $\mathcal{G}_{\mathsf{prim}}(\mathcal{H})$ is a clique of size $|V(\mathcal{H})|$ independently from the rest of $\mathcal{H}$.

The *incidence graph* of a hypergraph $\mathcal{H}$, denoted by $\mathcal{G}_{\mathsf{inc}}(\mathcal{H})$, is the bipartite graph having the vertices of $\mathcal{H}$ on one side and the hyperedges of $\mathcal{H}$ on the other side and such that there is an edge between a vertex v and a hyperedge e if and only if $v \in e$. Formally, $\mathcal{G}_{\mathsf{inc}}(\mathcal{H}) = (\mathcal{H}, V(\mathcal{H}), \{\{v, e\} \mid e \in \mathcal{H}, v \in e\})$. The incidence graph of a hypergraph has as much structure as the whole hypergraph

since one can entirely reconstruct $\mathcal{H}$ from $\mathcal{G}_{\mathsf{inc}}(\mathcal{H})$.

A hypergraph $\mathcal{H}'$ is a *subhypergraph* of $\mathcal{H}$ if $\mathcal{H}' \subseteq \mathcal{H}$. Given a subset $W \subseteq V(\mathcal{H})$, the *hypergraph induced by* W, denoted by $\mathcal{H}$ is defined to be $\mathcal{H}[W] = \{e \cap W \mid e \in \mathcal{H}\} \setminus \{\emptyset\}$. For $x \in V(\mathcal{H})$, we often use the notation $\mathcal{H} \setminus \{x\})$ for $\mathcal{H}[V(\mathcal{H}) \setminus \{x\}]$.

1.2.2 Graph measures and decompositions

Graph decomposition is a very general and handy technique that aims to understand the structure of a complex graph. This has many applications, particularly in parametrized complexity [FG06] where algorithms are designed to take advantage of the input structure. The complexity of such decompositions is usually measured by an integer, which define a notion of graph measure, also called width. Such graph width may be seen as a method to quantify the complexity of the structure of a graph. In this thesis, we use graph decompositions to design better algorithms for #SAT on CNF-formulas but the algorithmic applications go far beyond this restricted framework.

Tree width. Tree width [Bod93b] is one of the most notorious graph measure. Intuitively, it aims to measure the distance of a given graph to a tree. For example, a cycle is "almost" a tree since it is sufficient to disconnect one edge to have a tree. We will see that a cycle has tree width 2. Cliques on the other hand are highly connected and intuitively they are very far from being a tree. We will see that a k-clique has tree width k.

Tree width is based on a special kind of decomposition of graphs called *tree decomposition*.

Definition 1.23. *A tree decomposition of a graph $G = (V, E)$ is a tree T where each vertex t of T is labeled with a subset $\lambda(t)$ of V, called a* bag. *Moreover, the bags respect the following conditions:*

- Connectedness: *for every $v \in V$, the vertices of t such that $v \in \lambda(t)$ form a connected subtree of T. In symbols, for every $v \in V$, it holds that $\{t \in V(T) \mid v \in \lambda(t)\}$ is a connected subtree of T.*
- Completeness: *for every $e \in E$, there exists a bag covering e. Formally, there exists $t \in V(T)$ such that $e \subseteq \lambda(t)$.*

It is easy to see that every graph has a least one tree decomposition: the tree having one vertex t labeled by V is a tree decomposition of G.

Definition 1.24. *Let G be a graph and (T, λ) be a tree decomposition of G. The tree width of T, denoted by $\mathbf{tw}(T)$, is the size of the biggest bag in G minus one, that is, $\max_{t \in V(T)} |\lambda(t)| - 1$. The tree width of a graph G, denoted by $\mathbf{tw}(G)$, is the minimal tree width over every tree decompositions of G.*

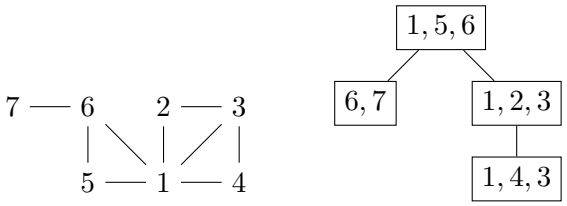

Figure 1.3: A graph and a tree decomposition of width 2

Graph	Tree Width
Trees	1
Cycles	2
k-cliques	$k-1$
$(m \times n)$ grids	$\min(m,n)$

Table 1.1: The tree width of some graphs

Table 1.1 gives the tree width of some graphs and Figure 1.3 gives an example of a tree decomposition of width 2.

In order to ease heavy notations, we will often use tree decompositions without explicitly mentioning the labeling function λ. By writing "let T be a tree decomposition of G", we implicitly assume that T is a labeled tree and we will often speak of the label of a vertex t of T without explicitly mentioning $\lambda(t)$.

Computing the tree width and the best tree decomposition of a graph is an NP-hard problem. Fortunately, tree width can be computed efficiently in FPT time, that is, we can decide in linear time if a graph is of tree width at most k, for k a constant:

Theorem 1.25 ([Bod93a])**.** *The problem of deciding whether the tree width of a graph G is smaller than an integer k is* NP*-complete. There exists an algorithm that given a graph $G = (V, E)$ and an integer k outputs a tree decomposition of G of width k if it exists and fails otherwise in time $2^{O(k)}O(n+m)$ where $n = |V|$ and $m = |E|$.*

Bodlaender [Bod06] has proved a useful characterization of tree width in terms of elimination orders which helps to find new algorithmic techniques that can be used on bounded tree width graphs. For a graph $G = (V, E)$ and a vertex $x \in V$, we denote by G/x the graph obtained by removing x from V and replacing its neighborhood by a clique. More formally $G/x = (V', E')$ is the graph with vertices $V' = V \backslash \{x\}$ and edges $E' = \{(u, v) \in E \mid u \neq x, v \neq x\} \cup \{(u, v) \in \mathcal{N}(x)^2 \mid u \neq v\}$.

Definition 1.26. *Let $G = (V, E)$ be a graph. An elimination order of width k for G is an ordering $x_1, \ldots, x_n$ of the vertices of G such that the degree of x_i in G_i is at most k where $G_1 = G$ and $G_{i+1} = (G_i/x_i)$ for all $i \leq n-1$.*

Theorem 1.27 ([Bod06]). *A graph G is of tree width k if and only if it has an elimination order of width k. Moreover, given a graph G, one can find an elimination of width $\mathbf{tw}(G)$ in time $2^{O(k)}O(n+m)$ where n is the number of vertices of G and m the number of edges.*

Proof (sketch). Let G be a graph of tree width k. We find a vertex x of G of degree at most k such that G/x is of tree width k. Let T be a tree decomposition of G of tree width k. Let t be a leaf of T. We can assume that the bag B_t of t is not included in its father, otherwise, we could remove this bag from the tree decomposition. Thus, there is a vertex x in this bag that is nowhere else in the graph, that is, the entire neighborhood of x is included in B_t. Thus x is of degree at most $|B_t| - 1 \leq k$. Now observe that T where we remove x from B_t is a tree decomposition of G/x of tree width at most k. Indeed, it is clearly a tree decomposition of tree width at most k since we only have removed a vertex that appeared only in one bag, so we have not breaked the connectedness nor have made any bag bigger. Moreover, every edge of G/x that are also in G are still covered by a bag in T. Now, if we take an new edge of G/x, then it is between two neighbors of x by construction. Thus this edge is covered by the bag where x was.

For the other way around, assume we have a graph G, x of degree k in G and a tree decomposition T of G/x of width at most k. We show how to construct a tree decomposition of T of width k. We rely on a well-known property of tree decomposition: if K is a clique of a graph G', then for any tree decomposition T' of G', there is a bag of T' that contains K. Applying this observation to G/x, we know that there is a bag B_t of T that contains the clique with the neighbors $\mathcal{N}(x) = \{y_1, \ldots, y_k\}$ of x in G since they form a clique of G/x. Now, we connect a bag $B = \{x, y_1, \ldots, y_k\}$ to t in T. This is a tree decomposition of G and it is of tree width at most k. □

On trees, Theorem 1.27 only states that one can get the empty graph by iteratively removing leaves in a tree. In a cycle, observe that removing a vertex from an n-cycle with $n > 2$ results in a $(n-1)$-cycle. Since every vertex of a cycle is of degree 2, this gives a proof that cycles are of tree width 2.

Clique width. We have seen that cliques have maximal tree width. However, cliques are graphs having a very simple structure and are thus easy instances in numerous problems: in such cases, tree width is thus not the right measure of the structure of the graph. Clique width was introduced in [CER91] by Courcelle, Engelfriet and Rozenberg to understand the structure of graphs even if they are dense. It is not based on tree decomposition as tree width but on a set of algebraic operations on colored graphs.

Let V be a finite set. A *parse tree* T of width k for V is a rooted tree whose leaves are labeled by an element of $v \in V$ and an integer $i \leq k$, called the color of v. Each v labels at most one leaf of T. The internal vertices of T are labeled with

the operations $\cup$, $\rho_{i,j}$ and $\eta_{i,j}$ with $i, j \leq k$. The colored graph G_t computed by a vertex t of T is defined inductively by:

- if t is a leaf labeled by (v, i), then G_t is the singleton graph having one vertex v colored with i,
- if t is labeled with $\cup$ then t has two children t_1, t_2 and $G_t = G_{t_1} \cup G_{t_2}$,
- if t is labeled with $\rho_{i,j}$ where $i \neq j$ then t has one child u and G_t is obtained by recoloring with j every vertex of G_u colored with i,
- if t is labeled with $\eta_{i,j}$ where $i \neq j$ then t has one child u and G_t is obtained by connecting with edges all vertices of G_u colored with i to all vertices colored j.

A graph $G = (V, E)$ is of *clique width* at most k if there exists a parse tree T for V of width k such that the root of T computes G with any coloring. The clique width of a graph G, denoted by $\mathbf{cw}(G)$, is the minimal k such that G is of clique width at most k.

It is easy to see that for every $G = (V, E)$, $\mathbf{cw}(G) \leq |V|$. Indeed, it is sufficient to construct the union of each vertex $v \in V$ with a distinct color c_v and then apply η_{c_u, c_v} for each edge $\{u, v\} \in E$ to add the edges.

Cliques are of clique width 2. Indeed, if T_n is a parse tree of width 2 for K_n, then $T_{n+1} = \eta_{1,2}(\rho_{2,1}(T) \cup (n+1, 2))$ is a parse tree of width 2 for K_{n+1}. This shows that we can have an arbitrary large gap between tree width and clique width since K_n is of tree width 2. In general, classes of graph where the tree width is bounded also have bounded clique width by the following bound:

Theorem 1.28 ([CO00])**.** *For every graph G, it holds that* $\mathbf{cw}(G) \leq 2^{\mathbf{tw}(G)+1} + 1$.

In contrast to tree width, the computation of clique width is not known to be FPT nor W[1]-hard. However, given a graph G and an integer k, it is NP-complete to decide whether G is of clique width at most k [FRRS09]. We will see in Chapter 2 how one can approximate clique-width by relating it to another graph width, the rank-width.

Branch decompositions. Let $G = (V, E)$ be a graph. A *branch decomposition of G* is a rooted binary tree T and with a one-to-one correspondence between the leaves $L(T)$ of T and V. As usual, given a vertex v of T, we denote by T_v the subtree of T rooted in v. Naturally $L(T_v)$ denotes the set of vertices labeling the leaves of T_v and $\overline{L(T_v)}$ denotes $L(T) \setminus L(T_v) = V \setminus L(T_v)$. A vertex v induces the natural partition of variables $(L(T_v), \overline{L(T_v)})$. In this paragraph, we denote by $X_v = L(T_v)$ and $\overline{X_v} = \overline{L(T_v)}$.

Branch decompositions is a handy tool for defining graph measures. A function $f : 2^V \to \mathbb{N}$ is said to be *symmetric* if for every $X \subseteq V$, $f(X) = f(\overline{X})$. Given a symmetric function f and a branch decomposition T of G, we define the f-width

of T to be $\max f(X_v) = \max f(\overline{X_v})$ where v runs over the vertices of T. The f-width of a graph G is the minimum of the f-widths of the branch decompositions of G.

For example, let $\mathsf{MM} : 2^V \to \mathbb{N}$ be the function that associates to $X \subseteq V$ the size of the biggest matching of $G[X, \overline{X}]$. This is clearly a symmetric function since $G[X, \overline{X}] = G[\overline{X}, X]$. The MM-width [Vat12] of a branch decomposition T of G is the biggest matching one can find in the graphs $G[X_v, \overline{X_v}]$ for every vertex v in T. We denote by $\mathbf{mmw}(G)$ the MM-width of G. MM-width is actually roughly the tree width of G.

Lemma 1.29 ([Vat12])**.** *Let G be a graph, then $\frac{1}{3}(\mathbf{tw}(G) + 1) \leq \mathbf{mmw}(G) \leq \mathbf{tw}(G) + 1$.*

Another graph width defined on branch decompositions is the MIM-width. MIM-width is defined to be the width associated to the symmetric function $\mathsf{MIM} : 2^V \to \mathbb{N}$ that associates to $X \subseteq V$ the size of the biggest *induced* matching of $G[X, \overline{X}]$. An induced matching of a graph G is a matching M such that $G[V(M)]$ is a matching. In other word, if $\{u, u'\}, \{v, v'\}$ are two distinct edges of M, then $\{u, v\}$, $\{u', v'\}$, $\{u, v'\}$ and $\{u', v\}$ are not edges of G. Classes of graphs with bounded clique width have also bounded MIM-width by:

Lemma 1.30 ([Vat12])**.** *Let G be a graph, then $\mathbf{mimw}(G) \leq \mathbf{cw}(G)$.*

The complexity of computing a branch decomposition of optimal MIM-width is still unknown.

1.2.3 Hypergraphs: acyclicity and decompositions

In Section 1.2.2, we have presented several graph widths that were helpful to characterize the structure of graphs. Similar decompositions have been proposed for hypergraphs to generalize tree width. In this section, we start by presenting how acyclicity generalizes to hypergraph. This question is already insightful as we shall see that many definitions of hypergraph acyclicity are possible, all leading to interesting classes. We then present the main decomposition notion for hypergraphs: the (generalized) hypertree width.

Acyclicity notions. Generalizing the notion of acyclicity to hypergraphs is tricky as even the notion of cycle is not easy to generalize. Figure 1.4 pictures four hypergraphs where there is a path from 1 to 3 that may be interpreted as a cycle for at least one acyclicity notion we will present in this thesis. Most of these notions were introduced by Fagin [Fag83] in order to find tractable database queries. An overview of known results on hypergraph acyclicity with simplified proofs can be found in a survey of Johann Brault-Baron [BB14] and a good introduction with several results on the subject can be found in the thesis of Duris [Dur09].

A minimal requirement for any notion of hypergraph acyclicity is that when restricted to graphs, the notion should coincide with the usual definition of acyclicity for graphs.

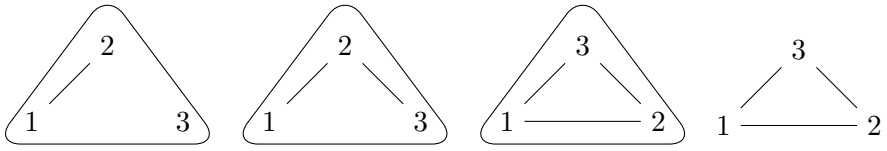

Figure 1.4: Which one is a cycle?

Berge acyclicity One of the most simple definition of hypergraph acyclicity was given by Berge in [Ber85]. A hypergraph $\mathcal{H}$ is Berge-acyclic if and only if $\mathcal{G}_{\mathsf{inc}}(\mathcal{H})$ is acyclic. This notion is very restrictive and it is sufficient that it exists $e, f \in \mathcal{H}$ such that $e \subseteq f$ to induce a cycle in $\mathcal{G}_{\mathsf{inc}}(\mathcal{H})$. For example, the first hypergraph of Figure 1.4 is not Berge-acyclic. When restricted to graph, Berge acyclicity collapses with graph acyclicity since the incidence graph of a graph is the same graph when each edge $e = \{u, v\}$ is split in two and a new vertex e is introduced in between. However, we will not effectively use this acyclicity in this thesis since it is too restrictive.

α-acyclicity A much more general notion of acyclicity is the α-acyclicity introduced by Beeri, Fagin, Maier and Yannakakis in [BFMY83]. This notion is inspired by notions of tree decompositions. Intuitively, a hypergraph is α-acyclic if its edges may be organized in a tree. Formally, a *join tree* (T, λ) for a hypergraph $\mathcal{H}$ consists of a tree T and a one-to-one mapping λ from the vertices of T and $\mathcal{H}$ having the connectedness property: for every $x \in V(\mathcal{H})$, $\{t \in V(T) \mid x \in \lambda(t)\}$ is a connected subtree of T. A hypergraph is said to be *α-acyclic* if it has a join tree.

When restricted to graph, the notion of join tree collapses with the notion of tree decomposition of width 1. Thus the notion of α-acyclicity collapses to the notion of graph acyclicity too.

Other characterizations of α-acyclicity have been given and some of them led to efficient algorithmic techniques to test if a hypergraph is α-acyclic. An *α-elimination order* for $\mathcal{H}$ is an ordering $x_1, \ldots, x_n$ of $V(\mathcal{H})$ such that the neighborhood of x_{i+1} in $\mathcal{H}[x_{i+1}, \ldots, x_n]$ is covered by an edge of $\mathcal{H}$. We have the following [TY84, BFMY83]:

Proposition 1.31. *A hypergraph $\mathcal{H}$ is α-acyclic if and only if one of the following holds:*

1. *Every clique of $\mathcal{G}_{\mathsf{prim}}(\mathcal{H})$ is included in an edge of $\mathcal{H}$ and $\mathcal{G}_{\mathsf{prim}}(\mathcal{H})$ is chordal.*
2. *There exists an α-elimination order of $V(\mathcal{H})$.*

The first characterization has been used by Yannakakis and Tarjan in [TY84] to construct a non-trivial linear time algorithm for testing α-acyclicity:

Theorem 1.32 ([TY84])**.** *There exists an algorithm that given $\mathcal{H}$ outputs an α-elimination order for $\mathcal{H}$ if it exists and fails otherwise in time $O(|\mathcal{H}| + |V(\mathcal{H})|)$.*

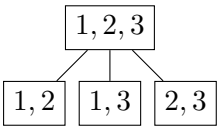

Figure 1.5: A join tree for the third hypergraph of Figure 1.4

A very counter intuitive aspect of α-acyclicity is that this notion is not hereditary. Indeed, it may be that a hypergraph $\mathcal{H}$ is α-acyclic when there exists $\mathcal{H}' \subseteq \mathcal{H}$ that is not α-acyclic. As an example, consider the third hypergraph of Figure 1.4. This hypergraph is acyclic and a join tree for it is given on Figure 1.5. However, the subhypergraph obtained by removing the hyperedge $\{1, 2, 3\}$ is a graph cycle of length 3, thus it is not α-acyclic. This phenomenon actually appears a lot in hypergraph, as stated in the following observation:

Observation 1. *If $V(\mathcal{H}) \in \mathcal{H}$ then $\mathcal{H}$ is α-acyclic. Indeed, the tree consisting of a node labeled with $V(\mathcal{H})$ connected to $|\mathcal{H}| - 1$ other nodes, each labeled with an edge of $\mathcal{H}$ is a join tree of $\mathcal{H}$.*

β-acyclicity. The notion of β-acyclicity is a notion of acyclicity that aims to fix the counter-intuitive fact that subhypergraph of α-acyclic hypergraphs may not be α-acyclic. The most natural way of doing so is simply to ask for every subhypergraph to be α-acyclic as well:

Definition 1.33. *A hypergraph $\mathcal{H}$ is* β-acyclic *if and only if for every $\mathcal{H}' \subseteq \mathcal{H}$, $\mathcal{H}'$ is α-acyclic.*

As α-acyclicity, when restricted to graphs, the notion of β-acyclicity collapses with the notion of graph acyclicity since every subgraph of an acyclic graph are also acyclic.

It can be checked that the first two hypergraphs of Figure 1.4 are β-acyclic. The third one however is not β-acyclic since we have seen earlier that the graph cycle of length 3 was one of its subhypergraph.

Definition 1.33 is not interesting since it is hard to use it algorithmically. Fortunately, many natural equivalent characterizations of β-acyclic hypergraphs have been given. One of the most useful is defined in terms of elimination order. Let $\mathcal{H}$ be a hypergraph. A β-leaf in $\mathcal{H}$ is a vertex $x \in V(\mathcal{H})$ such that $\mathcal{H}_x = \{e \in \mathcal{H} \mid x \in e\}$ is ordered by inclusion, that is, $\mathcal{H}_x = \{e_1, \ldots, e_m\}$ with $e_1 \subseteq \cdots \subseteq e_m$.

A β-elimination order for $\mathcal{H}$ is an ordering $x_1, \ldots, x_n$ of $V(\mathcal{H})$ such that for every $i \leq n$, x_i is a β-leaf of $\mathcal{H}[\{x_i, \ldots, x_n\}]$. For example, $1, 3, 2$ is a β-elimination order of the second hypergraph of Figure 1.4. Such orders characterize β-acyclicity:

Theorem 1.34 ([BFMY83])**.** *A hypergraph $\mathcal{H}$ is β-acyclic if and only if there exists a β-elimination order for $\mathcal{H}$.*

Theorem 1.34 provides an algorithm for testing if a hypergraph is β-acyclic. Indeed, it is sufficient to look for β-leaves and to greedily eliminate them. One can test if a vertex x is a β-leaf of $\mathcal{H}$ by just testing set inclusions, thus, it can be done in polynomial time, thus, finding a β-elimination order can be done in polynomial time.

This algorithm is not optimal and may be refined by using the right data structures. The best known algorithms for testing β-acyclicity are those solving the problem of *double lexical ordering*. A double lexical ordering of a $\{0,1\}$-matrix M is a permutation of its rows and it columns such both its rows and columns are sorted in lexicographical order. It is observed in [Lub87] that if we are provided a double lexical ordering of the adjacency matrix of a hypergraph $\mathcal{H}$ then we can decide in linear time if it is β-acyclic. The best known algorithms for double lexical ordering are due to Praigen and Tarjan [PT87] and Spinrad [Spi93]. This gives the following:

Theorem 1.35 ([PT87])**.** *Given the adjacency matrix of a hypergraph $\mathcal{H}$, one can decide if $\mathcal{H}$ is β-acyclic and, if so, compute a β-elimination order for $\mathcal{H}$ in time $O(\min(mn, s\log(n+m)))$ where $m = |\mathcal{H}|$, $n = |V(\mathcal{H})|$ and $s = \|\mathcal{H}\|$.*

Given a cycle $C = x_1, \ldots, x_n$ in a graph G, we call a *chord* of C an edge (x_i, x_j) where $j \neq i+1 \mod n$. A graph G is said to be *chordal* if every cycle of G of length more than 3 has a chord. If G is bipartite, then cycle of length 4 or 5 cannot have a chord. Thus, a graph G is said *chordal bipartite* if it is bipartite and every cycle of length at least 6 has a chord. We can characterize β-acyclicity in terms of its incidence graph which will prove useful when we want to compare β-acyclicity, which is a hypergraph property, to other graph measures:

Theorem 1.36 ([ADM86])**.** *A hypergraph is β-acyclic if and only if its incidence graph is chordal bipartite.*

γ-acyclicity. This notion of acyclicity is less general than β-acyclicity but still more general than Berge-acyclicity. Several definitions exist in term of join tree or in term of elimination order. We do not recall such definitions here since we will not really need it in the rest of the thesis. The interested reader may refer to the original paper of Fagin [BFMY83] or to this paper of Duris [Dur12]. We only recall a result concerning how γ-acyclicity compares with other widths:

Theorem 1.37 ([GP04])**.** *Let $\mathcal{H}$ be a γ-acyclic hypergraph then $\mathbf{cw}(\mathcal{G}_{\mathsf{inc}}(calH)) \leq 3$.*

We quickly recall how the different acyclicity notions compare:

$$\text{Berge-acyclicity} \subseteq \gamma\text{-acyclicity} \subseteq \beta\text{-acyclicity} \subseteq \alpha\text{-acyclicity}.$$

Hypertree width. Now that we have defined notions of acyclicity for hypergraph, it is natural to ask whether such notions can be turned into width measures such as tree width. In this section, we define the notions of hypertree width and of generalized hypertree width that were introduced by Gottlob, Leone and Scarcello [GLS99] to understand the complexity of answering database queries. The interested reader may find more details in a survey by the same authors [GLS01b]. These notions are based on generalizations of the notion of tree decomposition to the setting of hypergraph:

Definition 1.38. *Let $\mathcal{H}$ be a hypergraph. A* generalized hypertree decomposition *$(\mathcal{T}, \lambda)$ of $\mathcal{H}$ is a tree $\mathcal{T}$ together with a labeling function λ which associates to each vertex of t a subset of $V(\mathcal{H})$. Moreover, $(\mathcal{T}, \lambda)$ respects the following conditions:*

- *for every $e \in \mathcal{H}$, there exists a vertex t of $\mathcal{T}$ such that $e \subseteq \lambda(t)$,*
- *for every vertex x of $\mathcal{H}$, the set $\{u \mid x \in \lambda(u)\}$ is a connected subtree of $\mathcal{T}$.*

A generalized hypertree decomposition of a hypergraph $\mathcal{H}$ can actually be seen as tree decomposition of $\mathcal{G}_{\mathsf{prim}}(\mathcal{H})$. We however use another width measure on such decomposition:

Definition 1.39. *Let $\mathcal{H}$ be a hypergraph and $(\mathcal{T}, \lambda)$ a generalized hypertree decomposition of $\mathcal{H}$. The decomposition $\mathcal{T}$ is said to be of* generalized hypertree width *at most k if for every vertex t of $\mathcal{T}$, there exists $S \subseteq \mathcal{H}$ such that $|S| \leq k$ and $\lambda(t) \subseteq \bigcup_{e \in S} e$. We denote by $\mathbf{ghtw}(\mathcal{T})$ the smallest k such that $\mathcal{T}$ is of generalized hypertree width k. The* generalized hypertree width *of $\mathcal{H}$, denoted by $\mathbf{ghtw}(\mathcal{H})$, is the minimal generalized hypertree width of generalized hypertree decompositions of $\mathcal{H}$.*

It is easy to see that a join tree is a generalized hypertree decomposition of width 1. The converse is not true since a bag could cover more than one edge but this can be actually transformed into a join tree. Thus, generalized hypertree width 1 coincides with α-acyclicity:

Proposition 1.40. *A hypergraph $\mathcal{H}$ is α-acyclic if and only if $\mathbf{ghtw}(\mathcal{H}) = 1$.*

Unfortunately, computing the generalized hypertree width of a hypergraph is NP-hard, even if we are only interested in deciding whether it is at most 3:

Theorem 1.41 ([GMS09])**.** *Deciding whether a hypergraph has generalized hypertree width at most 3 is NP-complete.*

In order to overcome this difficulty, variants have been introduced. They are also based on hypertree decompositions but the edges that can be used to cover each bag are now more constrained which makes the width less general but tractable.

Definition 1.42. *Let $\mathcal{H}$ be a hypergraph. A* hypertree decomposition *$(\mathcal{T}, \lambda, \chi)$ of $\mathcal{H}$ is a tree $\mathcal{T}$ together with a labeling function λ which associates to each vertex of t a subset of $V(\mathcal{H})$ and a labeling function χ which associates to each vertex of t a subset of $\mathcal{H}$. Moreover, $(\mathcal{T}, \lambda, \chi)$ respects the following conditions:*

- *for every $e \in \mathcal{H}$, there exists a vertex t of $\mathcal{T}$ such that $e \subseteq \lambda(t)$,*
- *for every vertex x of $\mathcal{H}$, the set $\{u \mid x \in \lambda(u)\}$ is a connected subtree of $\mathcal{T}$,*
- *for every vertex t of $\mathcal{T}$, $\lambda(t) \subseteq \bigcup_{e \in \chi(t)} e$ and,*
- *for every vertex t of $\mathcal{T}$, for every vertex $x \in \bigcup_{e \in \chi(t)} e$, if there exists a vertex u of $\mathcal{T}_t$ such that $x \in \lambda(u)$ then $x \in \lambda(t)$.*

The hypertree width *of $\mathcal{T}$, denoted by $\mathbf{htw}(\mathcal{T})$, is defined to be $\max_t |\chi(t)|$ and the* hypertree width *of a hypergraph $\mathcal{H}$ is the minimal hypertree width of hypertree decompositions of $\mathcal{H}$.*

Hypertree width is much easier to deal with than generalized hypertree width and can be computed in polynomial time.

Theorem 1.43 ([GLS99])**.** *Let $k \in \mathbb{N}$. There is a polynomial time procedure that given $\mathcal{H}$ outputs a hypertree decomposition of $\mathcal{H}$ of width k if it exists and rejects otherwise.*

Fortunately, hypertree width is a good approximation of generalized hypertree width.

Theorem 1.44 ([AGG07])**.** *For every hypergraph $\mathcal{H}$, it holds that*

$$\mathbf{htw}(\mathcal{H}) \leq 3 \cdot \mathbf{ghtw}(\mathcal{H}) + 1.$$

Hypertree width however has the same counter-intuitive aspect as α-acyclicity: a subhypergraph may have a hypertree width greater than the hypertree width of the original one. Indeed, adding the edge $V(\mathcal{H})$ to a hypergraph $\mathcal{H}$ makes it α-acyclic but the hypertree width of $\mathcal{H}$ may be arbitrary large. We can define a width that behaves more naturally with respect to inclusion by choosing the width of the subhypergraph of maximal hypertree width.

Definition 1.45. *The β-hypertree width of a hypergraph $\mathcal{H}$ denoted by β-$\mathbf{htw}(\mathcal{H})$ is defined to be*

$$\beta\text{-}\mathbf{htw}(\mathcal{H}) = \max_{\mathcal{H}' \subseteq \mathcal{H}} \mathbf{htw}(\mathcal{H}').$$

From this definition, it is clear that a hypergraph is β-acyclic if and only if its β-hypertree width is 1. Unlike β-acyclicity however, this is the only characterization of β-hypertree width that is known so far. Thus we have a very poor understanding of this measure and such definition is not convenient to be used algorithmically. Some directions are given in Chapter 5 that may lead to other characterizations of β-hypertree width in terms of elimination orders.

1.3 Knowledge compilation

There are many ways of representing a boolean function $f : \{0,1\}^n \to \{0,1\}$. For example, we have seen in the previous section that a boolean function may be represented by CNF-formula, that is, a conjunction of clauses. We have also seen that deciding whether a CNF-formula has a satisfying assignments is NP-complete. It is thus very unlikely that a polynomial time algorithm exists for deciding whether a CNF-formula is satisfiable. If a boolean function is given as a DNF however, it is easy to find a satisfying assignments. This suggests that the complexity of deciding whether a boolean function is satisfiable strongly depends on its representation and that going from one representation to another may be hard or lead to a blow-up in the size of the representation. Indeed, if we could transform quickly any CNF into an equivalent small DNF, then we could also solve SAT quickly. Besides, it is possible to show that there are CNF formulas F such that any equivalent DNF formula F' is of exponential size in the size of F.

One aim of *knowledge representation* is to understand how different representations of boolean functions compare with one another. For example, given two representation languages L_1 and L_2, we say that L_1 is more succinct that L_2 if for every boolean function f, the size needed to represent f with L_1 is polynomially bounded by the size needed to represent it in L_2. For example, if we represent a boolean function by a boolean circuit, then it is more succinct than to represent it with a CNF or a DNF since both representations may be seen as a special kind of boolean circuits.

The question of going from one representation of a function to another is the main focus of *knowledge compilation*. The typical scenario where knowledge compilation is useful is the following: suppose we have a *fixed* boolean function f that has to be queried many times, with queries such as finding a satisfying assignment or counting the satisfying assignment of $f[x \mapsto 1]$ for some variable x. If f is given as a CNF, then each query is a hard problem to solve and may require an exponential time to be answered. If we can find however a succinct representation of f where such queries can be answered in polynomial time, then the only costly operation that remains is to *compile* f into this better representation. The other queries will be then answered quickly. We thus have replaced several costly operations by a phase of precomputation, which can be costly too, followed by a sequence of easy operations.

In this section, we present different representations of boolean functions and study their succinctness and the queries they support in polynomial time. A wider overview of the different possible representations and their properties can be found in [DM02].

1.3.1 Generalities

In this section, we will mention representation languages in a very informal way since in the rest of this thesis, we will only deal with concrete representation lan-

guages. The interested reader may find formal foundations for studying the theory of compilability of problems in the work of Cadoli , Liberatore and Schaerf [CDLS02] and Chen [Che05].

A *representation language* L is a way of representing boolean functions f, that is, it is a set together with an interpretation function $[\![.]\!]$ from L to the boolean functions. For example, L can be thought as boolean circuits or CNF. A representation language L comes with a notion of size $\mathsf{size}_L : L \to \mathbb{N}$ to measure the complexity of each element of L. For example, in the case of boolean circuits, the interpretation function could be the function computed by the circuit and the size could be the number of gates in the circuits. For a boolean function f, we denote by $L(f)$ the size of the smallest element C of L representing f that is $[\![C]\!] = f$. Given two representation languages L_1 and L_2, we say that L_1 is more *succinct* than L_2 and we write $L_1 \leq L_2$ if there exists a polynomial P such that for every boolean function f, $L_1(f) \leq P(L_2(f))$. For example, circuits are more succinct than CNF. We say that L_1 is not comparable with L_2 if $L_1 \not\leq L_2$. In other words, there exists a family of boolean functions $(f_n)_{n\in\mathbb{N}}$ with $\lim_{n\to\infty} L_2(f_n) = \infty$ such that for every polynomial P and every $n \in \mathbb{N}$, there exists N_0 such that $L_1(f_{N_0}) > P(L_2(f_{N_0}))$. L_2 is strictly more succinct than L_1, denoted by $L_1 < L_2$, if $L_1 \leq L_2$ and $L_2 \not\leq L_1$.

Given a problem, usually called *a query* , Q on boolean functions, we say that L supports Q in polynomial time if given an element C of L, $Q([\![C]\!])$ can be solved in polynomial time in $\mathsf{size}_L(C)$. For example, the query of deciding whether a boolean function has a satisfying assignment is supported in polynomial time by DNF. The main queries we will be interested in are:

- Find a satisfying assignment of f.
- Count the number of satisfying assignments of f.
- Enumerate the satisfying assignments f.

A *transformation* O is a function that transform one or several boolean function into another. For example, the negation $O : f \mapsto 1 - f$ or the conjunction $(f_1, f_2) \mapsto f_1 \wedge f_2$ are transformations. We say that a representation language polynomially supports an operation if there exists a polynomial P such that for every f, $L(O(f)) \leq P(L(f))$ (or $L(O(f_1, \ldots, f_k)) \leq P(L(f_1), \ldots, L(f_k))$ when there are more than one function). The main operations we will be interested in are:

- Negation, conjunction and disjunction of boolean functions.
- The conditioning of a boolean function by a partial assignment τ, that is the operation that maps a boolean function f and a partial assignment τ of its variables to $f(\tau)$.
- The universal/existential projection that is the operation that maps a boolean function f and a variable x to $\exists x.f$.

Usually, there is a trade-off between the succinctness of a representation language and the queries and transformations it supports in polynomial time. This will be illustrated in Section 1.3.2 and Section 1.3.3 where we give concrete examples of representation languages and their supported queries and transformations. Generally, the more succinct a language is, the more transformations it supports with a polynomial increase in their size and the less queries it supports in polynomial time.

1.3.2 Binary decision diagrams

In this section, we introduce representations based on binary decision diagrams. Numerous results concerning such representations may be find in the book of Wegener [Weg00].

Definition 1.46. *A functional binary decision diagram on variable X,* FBDD *for short, is a labeled directed acyclic graph $G = (V, E)$ such that:*

- *there exists exactly one vertex s, the source, of in-going degree* 0*,*
- *there exists two vertices t_0, t_1 of out-degree* 0 *labeled with* 0 *and* 1 *respectively,*
- *each vertex but t_0, t_1 are labeled with a variable $x \in X$ and has exactly two outgoing edges, one labeled with* 0 *and the other with* 1 *and,*
- *for every path $\mathcal{P}$ from s to t_i and $x \in X$, there is at most one node of $\mathcal{P}$ labeled with x.*

The size of G denoted by $\mathsf{size}(G)$ *is $|V|$. The set of variables labeling the vertices of G is denoted by* $\mathsf{var}(G)$*.*

Let G be an FBDD on variables X. Given $\tau : X \to \{0, 1\}$ and a node v of G, the successor of v labeled with $x \in X$ for τ is the node w such that (v, w) is the edge going out of G and labeled with $\tau(x)$. The truth assignment τ defines a path $\mathcal{P}_\tau$ starting from s and following the successor of the current node for τ until either t_0 or t_1 is reached. An truth assignment τ satisfies G if $\mathcal{P}_\tau$ ends in t_1. The function computed by G is the boolean function on variables X that consists of every satisfying assignments of G. See Figure 1.6 for an example of FBDD.

We will often identify the boolean function computed by G with G itself. For example, we will denote $\neg G$ the boolean function that is the negation of the boolean function computed by G.

Since satisfying assignments of an FBDD are characterized by paths in a directed graph, FBDD support many queries in polynomial time:

Theorem 1.47. *The following queries a supported in polynomial time for* FBDD*:*

- *satisfiability in linear time,*
- *counting in linear time,*

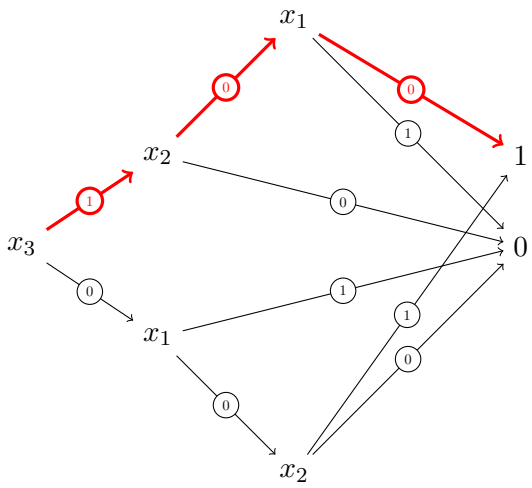

Figure 1.6: An FBDD F. In thick red, the path corresponding to the satisfying assignment $\{x_3 \mapsto 1, x_2 \mapsto 0, x_1 \mapsto 0\}$. F computes the boolean function $(x_3 \wedge \neg x_2 \wedge \neg x_1) \vee (\neg x_3 \wedge x_2 \wedge \neg x_1)$.

- *enumeration with a* $O(|\mathsf{var}(F)|)$ *delay.*

Similarly, it is easy to negate an FBDD by switching the labels of the leaves labeled with 0 and 1. An FBDD can be conditioned by a partial assignment without blow-up in size. Indeed, to force the value of a variable x to 1 in an FBDD, it is sufficient to remove every outgoing edge labeled by 0 from nodes labeled with x. It follows:

Theorem 1.48. *Let F be an* FBDD *on variables X then*

- *there exists an* FBDD *F' such that* $\mathsf{size}(F') = \mathsf{size}(F)$ *and $F' \equiv \neg F$ and,*
- *for every partial assignment τ of X, there exists F' such that* $\mathsf{size}(F') \leq \mathsf{size}(F)$ *and $F' \equiv F[\tau]$.*

However, transformations such as conjunctions, disjunctions or existential projection may lead to exponential blow-up of the size for FBDD (see Theorem 6.3.2 in [Weg00] or Chapter 6 for generalizations). A common way of making such transformations possible is to add more structure to the representation language. For FBDD, this is actually done by forcing the order of the variable along a path to be always the same. Such representation are called OBDD:

Definition 1.49. *An* OBDD *G on variables X is an* FBDD *on variables X such that there exists an ordering $x_1, \ldots, x_n$ of X such that every path from the source s to a node v of G testing x_j only tests variables in $\{x_1, \ldots, x_{j-1}\}$.*

For example the FBDD given in Figure 1.6 is not an FBDD since there is a path that tests x_2 before x_1 and a path that tests x_1 before x_2. An OBDD equivalent to this FBDD is depicted in Figure 1.7.

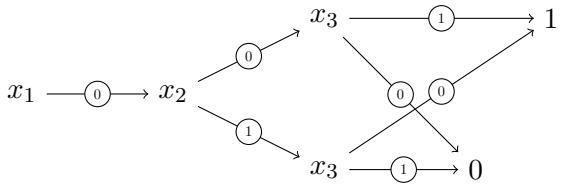

Figure 1.7: An OBDD equivalent to the FBDD in Figure 1.6 with order x_1, x_2, x_3. The missing edges are assumed to go to the 0-node.

Since OBDD are FBDD, they still support in polynomial time the queries given in Theorem 1.47. However, the transformation given in Theorem 1.48 may not be supported anymore since it is not clear from the statement of Theorem 1.48 that the order of variables are preserved by the transformation. As a matter of fact, it is. Moreover, since OBDD are more structured, we can actually show that they support even more transformation with only a polynomial increase:

Theorem 1.50. *Let F be an* OBDD *on variables X with order $x_1, \ldots, x_n$ then*

- *there exists an* OBDD *F' such that $\mathsf{size}(F') = \mathsf{size}(F)$ and $F' \equiv \neg F$ and,*
- *for every partial assignment τ of X, there exists F' such that $\mathsf{size}(F') \leq \mathsf{size}(F)$ and $F' \equiv F[\tau]$*
- *for every binary boolean function $\otimes : \{0,1\}^2 \to \{0,1\}$, for every* OBDD *$F'$ on variables X with order $x_1, \ldots, x_n$, there exists an* OBDD *F'' of size less than $\mathsf{size}(F) \cdot \mathsf{size}(F')$ computing $F \otimes F'$.*

Observe that Theorem 1.50 states that the conjunction of two OBDD can be represented efficiently if the two OBDD use the same underlying order. This is actually crucial and it is even possible to construct functions that have small OBDD using two different orders and such that their conjunction has no polynomial size FBDD (see Theorem 6.2.13 in [Weg00] and Chapter 6 for generalizations).

1.3.3 DNNF and its restrictions

In this section, we introduce representations based on boolean circuits having nice properties ensuring numerous queries in polynomial time.

Negation Normal Form. A boolean circuit C is in negation normal form, NNFs in short, if it has conjunction and disjunction gates, labeled by $\wedge$ and $\vee$ respectively, a distinguished gate called the output of C denoted by $\mathsf{output}(C)$ and if its inputs are labeled by literals or constant 1 or 0. The size of an NNF C, denoted by $\mathsf{size}(C)$, is the number of gates in C, its fan-in is the maximal in-degree of its gate and its depth is the longest path from the root to the leaf.

We denote by $\mathsf{var}(C)$ the set of variables labeling the inputs of C. Given a gate α, we denote by C_α the NNF whose circuit is the subcircuit of C rooted in

α and whose output is α. The gates β such that there is a wire from β to α are called the children or the inputs of α.

The boolean function on variables $\mathsf{var}(C)$ computed by a gate α of C is defined inductively as:

- if α is an input then the boolean function computed by α is the boolean function corresponding to its literal
- if α is an $\wedge$-gate then the boolean function computed by α is the conjunction of the boolean functions computed by its children
- if α is an $\vee$-gate then the boolean function computed by α is the disjunction of the boolean functions computed by its children.

The function computed by an NNF C is the function computed by the gate $\mathsf{output}(C)$. A CNF-formula F may be seen as an NNF of size $1 + |F| + \mathsf{size}(F)$ having one $\wedge$-gate, the output, connected to $|F|$ $\vee$-gates, one per clause, each of them connected to inputs labeled by its literals. NNF are thus already too general to support interesting queries such as finding a satisfying assignments. In the following, we thus consider restrictions of NNF.

Observe that if α is the output of C, it does not necessarily means that $C_\alpha = C$. Indeed, it may be that some gates are not reachable from α. This gates could be easily dropped as they do not really change the function computed by C. However, we choose to allow them in the circuit anyway to avoid normalization when we perform some operation such as disconnecting gates in C.

We extend most notations on CNF-formulas defined in Section 1.1.2 to NNF . For example, given a NNF C, we denote by $\mathsf{sat}(C)$ the set of satisfying assignments of C. Given an partial assignment $\tau : X \subseteq \mathsf{var}(C) \to \{0,1\}$, we denote by $C[\tau]$ the boolean function on variables $\mathsf{var}(C) \setminus X$ that is true on an assignment $\tau' : \mathsf{var}(C) \setminus X \to \{0,1\}$ such that $\tau \cup \tau'$ is a satisfying assignment of C. We denote by $\tau \models C$ if $C[\tau]$ is a tautology.

Decomposable NNF. An $\wedge$-gate α of an NNF C is said to be *decomposable* if it holds that for every β, γ children of α, $\mathsf{var}(C_\beta) \cap \mathsf{var}(C_\gamma) = \emptyset$. An NNF D is said to be a decomposable NNF, in short, DNNF if all the $\wedge$-gates of D are decomposable. Decomposability is a natural notion that arises independently in the area of circuit complexity under the name of *multilinearity*.

DNNF are a generalization of DNF since a DNF-formula F may be seen as a DNNF equivalent to F and of size $1 + |F| + \mathsf{size}(F)$ having one $\vee$-gate, the output, connected to $|F|$ decomposable $\wedge$-gates, one per clause, each of them connected to inputs labeled by its literals. As we may assume the variables of each literal of a clause to be disjoint (or it would yield an unsatisfiable clause), the conjunction are decomposable. Figure 1.8 represents DNNF for a DNF-formula.

DNNF were introduced and studied by Darwiche in [Dar01a]. They are restricted enough to support conditioning, decision, existential projection and enumeration in polynomial time. However, since counting the number of satisfying

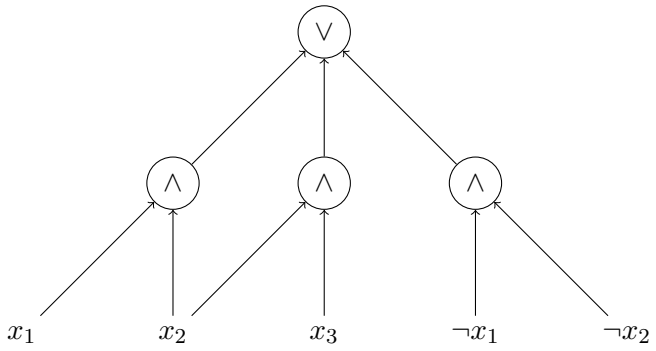

Figure 1.8: A DNNF for $(x_1 \wedge x_2) \vee (x_2 \wedge x_3) \vee (\neg x_1 \wedge \neg x_2)$

assignments of a DNF is already #P-complete, it is also #P-complete to count the number of satisfying assignments of a DNNF. Before studying DNNF in more details, we introduce the different restrictions of DNNF we will use in this thesis.

Determinism. DNNFs already support a large number of useful queries in polynomial time but lack one of the most interesting one for our purpose of understanding the complexity of #SAT: model counting. The main difficulty of counting the number of satisfying assignments of a DNNF is to take into account the fact that $\vee$-gates may have common satisfying assignments. Therefore, one has to be careful not to count the same satisfying assignment twice. To overcome this difficulty, the notion of determinism has been introduced in [Dar01b]. Let D be a DNNF and let α be an $\vee$-gate of D with children $\alpha_1, \ldots, \alpha_k$. The gate α is said *deterministic* if for every $i \neq j$, $D_{\alpha_i} \wedge D_{\alpha_j}$ is not satisfiable, that is, D_{α_i} and D_{α_j} have disjoint models. A DNNF D is said *deterministic*, in short d-DNNF, if for every $\vee$-gate α of D, α is deterministic.

For example, the DNNF given in Figure 1.8 is not deterministic since $\tau = \{x_1 \mapsto 1, x_2 \mapsto 1, x_3 \mapsto 1\}$ satisfies two children of the $\vee$-gate.

Decision. Contrary to decomposability, determinism is not a syntactic property of the DNNF but a semantic property. Given an NNF, it is easy to check that it is decomposable. This is not true for determinism. In Chapter 3, we will see an algorithm constructing d-DNNF. The guarantee that they are deterministic is given by the algorithm constructing it but we usually have no efficient way of checking whether a given DNNF is deterministic or not.

A way of syntactically enforcing the determinism of an $\vee$-gate is to use only decision nodes, a notion originating from binary decision diagrams [Bry92]. A *decision node* in a DNNF is an $\vee$-gate having the form $(x \wedge \alpha) \vee (\neg x \wedge \beta)$ where x is a variable and α, β are gates in the DNNF. A decision node is clearly deterministic since $\alpha \wedge (x \wedge \neg x) \wedge \beta$ is unsatisfiable. A *decision* DNNF, dec-DNNF for short, is a DNNF such that every $\vee$-gate is a decision node. An example of a dec-DNNF is

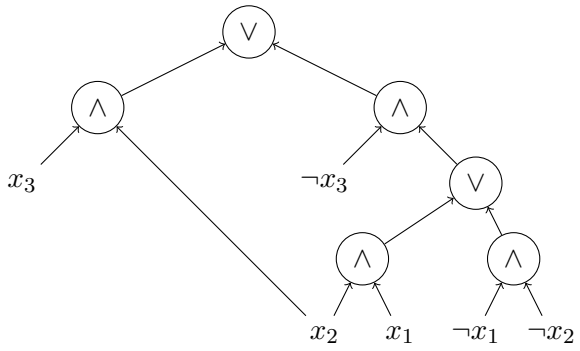

Figure 1.9: A dec-DNNF for $(x_1 \wedge x_2) \vee (x_2 \wedge x_3) \vee (\neg x_1 \wedge \neg x_2)$

given in Figure 1.9.

A natural question is to ask if such a syntactic restriction is as expressive as the semantic restriction of determinism. Unfortunately, it results in a loss of succinctness. Deterministic DNNFs are strictly more succinct than dec-DNNF. This separation can be found in [BLRS13] and is based on a lower bound on FBDD from [Weg00] and on an efficient simulation of dec-DNNF by FBDD:

Proposition 1.51. *There exists a family* $(D_n)_{n\in\mathbb{N}}$ *of* d-DNNF *such that* $\mathsf{size}(D_n) \leq n^2$ *and any decision* DNNF *computing* D_n *is of size at least* $2^{\Omega(\sqrt{n})}$.

dec-DNNF are however already more expressive than FBDD. Indeed, each node of an FBDD may be replaced by a decision node:

Proposition 1.52. *Let* F *be an* FBDD. *There exists a* dec-DNNF D *equivalent to* F *of size at most* $5 \cdot \mathsf{size}(F)$.

Structuredness. We have seen that some operation not supported in polynomial time by FBDD such as conjunction can be enabled by forcing an order on the variable. This idea can be generalized to DNNF by restricting the way the $\wedge$-gate can partition the variables. Such restricted DNNF are called *structured* DNNF [PD08]. A *variable tree* T for a set X, in short *vtree*, is a binary rooted tree whose leaves are in one-to-one correspondence with X. From a graph point of view, it can be seen as a branch decomposition of X. Given a vertex t of T, we denote by $\mathsf{var}(T_t) \subseteq X$ the set of variables labeling the leaves of the subtree of T rooted in t.

Given a fan-in 2 DNNF D on variables X and an $\wedge$-gate α of D with children β, γ, we say that α respects a vertex t of T if $\mathsf{var}(D_\beta) \subseteq \mathsf{var}(T_{t_1})$ and $\mathsf{var}(D_\gamma) \subseteq \mathsf{var}(T_{t_2})$ where t_1, t_2 are the children of t in T. We say that a fan-in 2 DNNF D *respects* a vtree T if for every $\wedge$-gate α of D, there exists a vertex t of T such that α respects t. We say that a DNNF D is *structured* if there exists a vtree T such that D respects t. For example, the DNNF of Figure 1.8 respects the vtree of Figure 1.10.

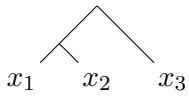

Figure 1.10: A vtree for the DNNF of Figure 1.8

Supported queries and transformations. We now recall some known result on DNNF and its restrictions. Most of these proofs can be found in [Dar01a]. An overview of these results can be found in Table 1.2.

Since DNNF are boolean circuits, one can easily perform conditioning. Indeed, to condition a DNNF by a partial truth assignment τ, it is sufficient to replace the input literals by the right constant. This transformation preserves decision-nodes, determinism and structuredness.

Proposition 1.53. *Let D be a (structured, deterministic, decision)* DNNF *and $\tau : X \subseteq \mathsf{var}(D) \to \{0,1\}$. There exists a (structured, deterministic, decision)* DNNF *D' equivalent to $D[\tau]$ of size at most $\mathsf{size}(D)$.*

Proof (sketch). We relabel every input labeled by a literal ℓ such that $\mathsf{var}(\ell) \in X$ by the constant $\tau(\ell)$.

It is easy to check by induction that for every gate α of D', D'_α computes $D_\alpha[\tau]$ where we identify the gate α of D' with the corresponding gate in D. $\wedge$-gate are still decomposable in D'.

If a $\vee$-gate is a decision gate on variable $x \in X$, then one side of the decision gate is equivalent to 0, thus we can remove the decision gate. That is, if D is a decision DNNF, so is D'.

We have to show that $\vee$-gates are still deterministic. Let α be an $\vee$-gate of D' and let β, γ be two children of α. We have $D'_\beta \wedge D'_\gamma \equiv D_\beta[\tau] \wedge D_\gamma[\tau] \equiv (D_\beta \wedge D_\gamma)[\tau]$ which is not satisfiable since $(D_\beta \wedge D_\gamma)$ is not satisfiable by definition of deterministic gates. Thus α is deterministic and if D is deterministic so is D'.

Finally, observe that for every gate α, $\mathsf{var}(D'_\alpha) \subseteq \mathsf{var}(D_\alpha)$ thus, if D respects a vtree T then D' also respects this tree. □

Using the same idea, one can existentially project a variable. However, determinism is not preserved by such existential projection.

Proposition 1.54. *Let D be a* DNNF *and let $x \in \mathsf{var}(D)$. There exists a* DNNF *D' equivalent to $\exists x.D$ of size at most $\mathsf{size}(D)$.*

Proof (sketch). Replace each input labeled with x or $\neg x$ by the constant 1. □

As DNF, DNNF are restricted enough to support linear time decision [Dar01a]:

Proposition 1.55. *Given a* DNNF *D, one can find a satisfying assignment of D in time $O(\mathsf{size}(D))$ and space $O(\mathsf{size}(D)|\mathsf{var}(D)|)$.*

Proposition 1.55 and Proposition 1.53 yield an efficient enumeration algorithm, Algorithm 1, by adapting an algorithm for enumerating monomial in a multilinear circuit that can be found in [Str10]. An algorithm for enumerating satisfying assignments of a DNNF was already given in [Dar01a]. We give here a more accurate result taking into account the delay between two solutions.

Proposition 1.56. *Given a* DNNF D*, one can enumerate the satisfying assignments of* D *with delay* $O(\mathsf{size}(D)|\mathsf{var}(D)|)$ *and using* $O(\mathsf{size}(D)|\mathsf{var}(D)|)$ *space.*

Proof. We use a well-known technique in enumeration called backtrack search. The key observation is that given a partial truth assignment τ of $\mathsf{var}(D)$, one can decide in time $O(\mathsf{size}(D))$ if $D[\tau]$ is satisfiable. It is indeed sufficient to find a satisfying assignment on the conditioned DNNF D, both operations being possible in linear time by Proposition 1.55 and Proposition 1.53.

The invariant of Algorithm 1 is that at each step, τ contains a truth assignment of $\{x_1, \ldots, x_{i-1}\}$ such that:

1. there exists $\tau' \simeq \tau$ such that $\tau' \in \mathsf{sat}(D)$,
2. for every $\tau' \in \mathsf{sat}(D)$ such that τ' comes before τ in the lexicographical order induced by $\{x_1, \ldots, x_n\}$, τ' has already been enumerated.

This invariant holds at the beginning of the algorithm since we ensure that $\mathsf{sat}(D) \neq \emptyset$. Thus there exists $\tau' \in \mathsf{sat}(D)$ such that $\tau' \simeq \emptyset$ and since the empty assignment is the smallest one for the lexicographical order, we have enumerated every satisfying assignment of D smaller than $\emptyset$, that is, none.

Now assume $i < |\mathsf{var}(D)|$. If $D[\tau \cup \{x_i \mapsto 0\}]$ is satisfiable, then we set $\tau \leftarrow \tau \cup \{x_i \mapsto 0\}$. We have to check that $\tau \cup \{x_i \mapsto 0\}$ respects items 2 and 1. This trivially respects item 1. Moreover, if $\tau' \in \mathsf{sat}(D)$ comes before $\tau \cup \{x_i \mapsto 0\}$ in the lexicographical order, then it also comes before τ, and by invariant, has already been enumerated, thus $\tau \cup \{x_i \mapsto 0\}$ still respects item 2.

If $D[\tau \cup \{x_i \mapsto 0\}]$ is not satisfiable, then we set $\tau \leftarrow \tau \cup \{x_i \mapsto 1\}$. Since τ respects item 1 and $D[\tau \cup \{x_i \mapsto 0\}]$ is not satisfiable, it means that $D[\tau \cup \{x_i \mapsto 1\}]$ has to be satisfiable, thus $\tau \cup \{x_i \mapsto 1\}$ respects item 1. Moreover, if $\tau' \in \mathsf{sat}(D)$ comes before $\tau \cup \{x_i \mapsto 1\}$ in the lexicographical order, either it also comes before τ, and by invariant, has already been enumerated or it comes between $\tau \cup \{x_i \mapsto 0\}$ and $\tau \cup \{x_i \mapsto 1\}$. That is $\tau' \simeq \tau \cup \{x_i \mapsto 0\}$. Since $D[\tau \cup \{x_i \mapsto 0\}]$ is not satisfiable, we know that such a τ' does not exist and then $\tau \cup \{x_i \mapsto 1\}$ respects 2.

Finally, assume that $i = |\mathsf{var}(D)|$. Since, by induction, τ respects item 1, we know that $\tau \in \mathsf{sat}(D)$ and we can output it. Now assume that

$$\{j \mid \tau(x_j) = 0 \text{ and } \mathsf{satDNNF}(D[\tau|_{x_1,\ldots,x_{j-1}} \cup \{x_j \mapsto 1\}]) \neq \text{UNSAT}\} = \emptyset.$$

In this case, we stop the enumeration. We claim that we have enumerated every satisfying assignment of D. Indeed, if there is a satisfying assignment that has not been enumerated, then it comes after τ in the lexicographical order. But then, it

has to come after $\tau|_{x_1,\ldots,x_{i-1}} \cup \{x_i \mapsto 1\}]$ for some i such that $\tau(x_i) = 0$. Thus, such assignment does not exist and we have enumerated all satisfying assignments of D.

Now, if this set is not empty, let

$$i = \max\{j \mid \tau(x_j) = 0 \text{ and } \mathsf{satDNNF}(D[\tau|_{x_1,\ldots,x_{j-1}} \cup \{x_j \mapsto 1\}]) \neq \text{UNSAT}\}.$$

We update τ to be $\tau' = \tau \cup \{x_i \mapsto 1\}$. By definition, $D[\tau']$ is satisfiable and then τ' respects item 1. Now, let τ'' be a satisfying assignment coming before τ' in the lexicographical order. Either it comes before τ and has already been enumerated. Or it comes before τ' and after τ. That is, there exists $j > i$ such that $\tau(x_j) = 0$, $\tau''(x_j) = 1$. But then, since i is maximal, it means that $D[\tau'']$ is not satisfiable. Thus τ' respect item 2.

It remains to show that the delay is $O(|\mathsf{var}(D)|\mathsf{size}(D))$. It is easy to observe that we run procedure satDNNF at most $2|\mathsf{var}(D)|$ times between two outputs: at most $|\mathsf{var}(D)|$ times to backtrack to the last partial assignment that can still be completed (the second while loop) and at most $|\mathsf{var}(D)|$ times to construct the next solution. Since satDNNF can be executed in time $O(\mathsf{size}(D))$, we have a delay of $O(|\mathsf{var}(D)|\mathsf{size}(D))$. Finally, since we store only one extra partial assignment τ and one integer $i \leq |\mathsf{var}(D)|$, we only need a space $O(|\mathsf{var}(D)|\mathsf{size}(D))$ to enumerate, which corresponds to the space needed to execute satDNNF. □

Since DNNFs are more general than DNFs and it is #P-complete to count the number of satisfying assignments of a DNNF, it is very unlikely that DNNFs support model counting in polynomial time. Determinism can however be used to count the number of model of a d-DNNF in linear time:

Proposition 1.57. *Given a* d-DNNF D*, one can count the satisfying assignments of* D *in time* $O(\mathsf{size}(D))$*.*

Proof. See Algorithm 2. It is based on the observation that if f and f' are boolean functions on disjoint variables then $|\mathsf{sat}(f \wedge f')| = |\mathsf{sat}(f)| \cdot |\mathsf{sat}(f')|$ and if f and f' are boolean functions having disjoint models, then $|\mathsf{sat}(f \vee f')| = 2^{|\mathsf{var}(f')\setminus\mathsf{var}(f)|}|\mathsf{sat}(f)| + 2^{|\mathsf{var}(f)\setminus\mathsf{var}(f')|}|\mathsf{sat}(f')|$. □

As for FBDD, the conjunctions of two DNNF may not be possible with only a polynomial increase (see Chapter 6, Section 6.2.4). However, as for OBDD, adding structure makes these transformations possible [PD08]:

Theorem 1.58. *Let* D_1, D_2 *be two* DNNF *that respect the same vtree* T*. One can construct in time* $O(\mathsf{size}(D_1) \cdot \mathsf{size}(D_2))$ *a* DNNF D *that respects* T *such that* $D \equiv D_1 \wedge D_2$ *and* $\mathsf{size}(D) \leq \mathsf{size}(D_1) \cdot \mathsf{size}(D_2)$*.*

We summarize the previous result in Table 1.2. In this thesis, we focus on a small subset of queries. A complete overview of the queries supported in polynomial time for the languages we have presented may be found in [DM02] and [PD08]. In Chapter 6, we use unconditional separations of languages to prove that some of the conditional results of Table 1.2 actually hold unconditionally.

Algorithm 1: An algorithm enumDNNF to enumerate the satisfying assignments of a DNNF

```
Data: A DNNF D
begin
    Give an arbitrary order {x_1, ..., x_n} to var(D) ;
    if satDNNF(D) = UNSAT then
        return STOP ;
    τ ← ∅ ;
    i ← 1 ;
    while 1 = 1 do
        if i < |var(D)| then
            if D[τ ∪ {x_i ↦ 0}] ≠ UNSAT then
                τ ← τ ∪ {x_i ↦ 0} ;
            else
                τ ← τ ∪ {x_i ↦ 1} ;
            i ← i + 1 ;
        else
            OUTPUT(τ) ;
            while i > 0 do
                if τ(x_i) = 0 and satDNNF(D[τ|_{x_1,...,x_{i-1}} ∪ {x_i ↦ 1}]) ≠
                UNSAT then
                    τ ← τ|_{x_1,...,x_{i-1}} ∪ {x_i ↦ 1} ;
                    i ← i + 1 ;
                    break ;
                i ← i − 1 ;
            return STOP ;
```

Algorithm 2: An algorithm #d-DNNF to count the satisfying assignments of a d-DNNF

Data: A d-DNNF D
begin
  $\alpha \leftarrow \mathsf{output}(D)$;
  if α *is labeled with a literal or constant* 1 **then**
    return 1
  if α *is labeled with* 0 **then**
    return 0
  if α *is labeled with* $\wedge$ **then**
    $n \leftarrow 1$;
    for β *child of* α **do**
      $n \leftarrow n \times \#\mathsf{d\text{-}DNNF}(D_\beta)$;
    return n ;
  if α *is labeled with* $\vee$ **then**
    $n \leftarrow 0$;
    for β *child of* α **do**
      $n \leftarrow n + 2^{|\mathsf{var}(D_\alpha) \setminus \mathsf{var}(D_\beta)|} \cdot \#\mathsf{d\text{-}DNNF}(D_\beta)$;
    return n ;

	DNNF	d-DNNF	dec-DNNF
Decision	Linear	Linear	Linear
Counting	•	Linear	Linear
Enumeration	delay $O(ns)$	delay $O(n)$	delay $O(n)$
Conditioning	Yes	Yes	Yes
$\exists$-projection	Yes	•	•

Table 1.2: The queries supported in polynomial time by DNNF. • means that the query is not possible unless P = NP. n stands for the number of variables and s for the size of the DNNF.

Normal form. We finish this section by presenting a useful normal form of DNNF. We say that a DNNF D is in normal form if it is of fan-in 2, if $\mathsf{output}(D)$ is the only gate of D of fan-out 0 and if no input is labeled by a constant. Fortunately, every DNNF computing a non-trivial function can be transformed into a DNNF in normal form with only a polynomial increase in its size. The transformation preserves determinism and structuredness.

Proposition 1.59. *Let D be a (structured, decision, deterministic)* DNNF *of fan-in k. There exists a (decision, deterministic)* DNNF *D' equivalent to D, having fanin 2, and such that $\mathsf{size}(D') \leq (k-1)\mathsf{size}(D) \leq \mathsf{size}(D)^2$.*

Proof (sketch). Replace each gate α of fan-in $k > 2$ in D by a binary tree having k leaves, each node of the tree being labeled as α. This preserves decomposability, determinism and structuredness (for structured DNNF, observe that, by definition, $\wedge$-gates are already of fan-in 2 so only $\vee$-gates are modified). The resulting DNNF has size at most $(k-1)\mathsf{size}(D) \leq \mathsf{size}(D)^2$ since $k \leq \mathsf{size}(D)$. □

We say that a DNNF D is constant free if no input of D is labeled by a constant.

Proposition 1.60. *Let D be a (decision, deterministic, structured)* DNNF *computing a non-trivial function. There exists a* DNNF *D' such that $\mathsf{size}(D') \leq \mathsf{size}(D)$ and D' is constant free. Moreover, the fanin of D' is smaller than the fanin of D.*

Proof. We iteratively propagate the constants by using the following rules $0 \vee f \equiv f$, $1 \vee f \equiv 1$, $0 \wedge f \equiv 0$ and $1 \wedge f \equiv f$ until there is no constant in the circuit. □

We can then prove the normal form theorem of DNNF:

Theorem 1.61. *Let D be a* DNNF*. There exists a* DNNF *D' in normal form, equivalent to D such that $\mathsf{size}(D') \leq \mathsf{size}(D)^2$.*

Proof. Start by transforming D into a fan-in 2 DNNF using Proposition 1.59 and then remove constants using Proposition 1.60 and gates of fan-out 0. □

The following feature of DNNF in normal form will be useful:

Proposition 1.62. *Let D be a* DNNF *in normal form and let α be an $\wedge$-gate of D with children α_1, α_2. The subcircuits D_{α_1} and D_{α_2} are disjoint.*

Proof. Assume the contrary and let β be a common gate. By definition, D_β is contained in both D_{α_1} and D_{α_2}. However, since $\mathsf{var}(D_{\alpha_1}) \cap \mathsf{var}(D_{\alpha_2}) = \emptyset$, we have $\mathsf{var}(D_\beta) = \emptyset$. Since D is constant free, it is a contradiction. □

Chapter 2

Structural restrictions of #SAT

The problem #SAT, which is given a CNF-formula, counting its satisfying assignments, is, unsurprisingly, a #P-complete problem [Val79a]. It is well-known that clause restrictions such as 2-SAT or Horn-SAT yield tractable classes for SAT and the complexity of such restrictions is well-understood, since Schaeffer [Sch78] exhibited a dichotomy: such restriction either yields a tractable class of instances or an NP-complete one. The picture however differs significantly for #SAT: computing and even approximating the number of solutions of a monotone 2-CNF is already as hard as the general case [Rot96]. A similar dichotomy is known for counting [CH96]: it states that the only tractable class such restrictions yield is the class of problem having affine clauses, which boils down to counting the number of solutions of a linear system over $\mathbb{Z}/2\mathbb{Z}$, which is a very restricted class of instances.

Hence, in order to discover tractable families of formulas for #SAT, it is not relevant to restrict the clauses individually. A successful line of research focuses on so-called structural restrictions of #SAT: contrary to 2-SAT, it is not the clauses that are independently restricted but the way they interact with one another. In order to quantify this interaction, a graph or a hypergraph representing the structure of a given formula is constructed and parameters from graph theory, such as tree width, are used to measure it [PSS13, SS13, SS10, STV14].

In this chapter, we review the main results concerning the complexity of structural restrictions of #SAT. Section 2.1 is focused on giving the main tools and definitions. In Section 2.2, we illustrate how one can take advantage of the underlying structure of a formula to discover new tractable classes by presenting results from [CDM14] on disjoint branches acyclicity. This serves as an example of a typical scenario of the discovery of new tractable classes of #SAT based on structural restrictions. Finally, we review in Section 2.3 all tractability results known prior to this thesis concerning structural restrictions of #SAT, including hardness results.

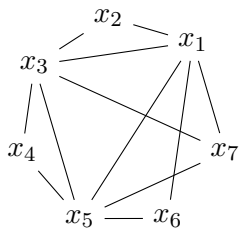

Figure 2.1: The primal graph $\mathcal{G}_{\mathsf{prim}}(F_{\mathsf{ex}})$ of F_{ex}

2.1 Structure of a CNF-formula

We explain in this section what we intend by "structure" of a formula. We present and compare different graphs and hypergraphs associated to formulas and explain how tools from graph theory presented in Section 1.2 can be used in this framework. We illustrate each definition on the same example

$$F_{\mathsf{ex}} = (x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee x_2 \vee x_3) \wedge (x_3 \vee x_4 \vee \neg x_5) \wedge (x_1 \vee x_5 \vee x_6) \wedge (x_1 \vee \neg x_3 \vee x_5 \vee x_7).$$

We denote the clauses of F_{ex} by $C_0 = (x_1 \vee x_2 \vee \neg x_3), C_1 = (x_1 \vee x_2 \vee x_3), C_2 = (x_3 \vee x_4 \vee \neg x_5), C_3 = (x_1 \vee x_5 \vee x_6)$ and $C_4 = (x_1 \vee \neg x_3 \vee x_5 \vee x_7)$.

2.1.1 Primal and dual graphs

One of the most basic and first studied graph associated to a formula is the so-called primal graph. Given a CNF-formula F, *the primal graph* of F denoted by $\mathcal{G}_{\mathsf{prim}}(F)$ is defined as the graph whose vertices are the variables of F and for which there is an edge between a pair of variables if and only they both appear in a clause of F. The negations on variables from the CNF-formula do not appear in the primal graph. Formally, $\mathcal{G}_{\mathsf{prim}} = (\mathsf{var}(F), E)$ where $E = \{\{x, y\} \mid \exists C \in F, \{x, y\} \subseteq \mathsf{var}(C)\}$. Figure 2.1 pictures the primal graph of the formula F_{ex}.

A closely related graph is the *dual graph* of the formula. The dual graph of a formula F, denoted by $\mathcal{G}_{\mathsf{dual}}(F)$, is the graph whose vertices are the clauses of F and there is an edge between two clauses if and only if they share a variable. Formally, $\mathcal{G}_{\mathsf{dual}} = (F, E)$ where $E = \{\{C_1, C_2\} \in F \mid \mathsf{var}(C_1) \cap \mathsf{var}(C_2) \neq \emptyset\}$. Figure 2.2 pictures the dual graph of the formula F_{ex}.

One of the main disadvantage of primal and dual graphs is that they forgot many information on the structure of the formula. A well-known observation is that if F is a CNF-formula, then adding only a clause $C = \{\mathsf{var}(F)\}$ containing every variables of F transforms the primal graph into an n-clique where $n = |\mathsf{var}(F)|$. In this case, the internal structure of F is completely lost in $\mathcal{G}_{\mathsf{prim}}(F)$, therefore primal graph is not the best tool to understand the structure of a formula. The same observation goes for the dual graph: adding a fresh variable x in every clauses of a CNF-formula F transform the dual graph into a clique. In the next

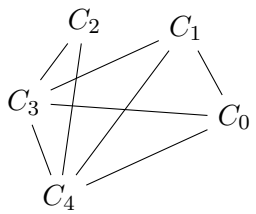

Figure 2.2: The dual graph $\mathcal{G}_{\mathsf{dual}}(F_{\mathsf{ex}})$ of F_{ex}

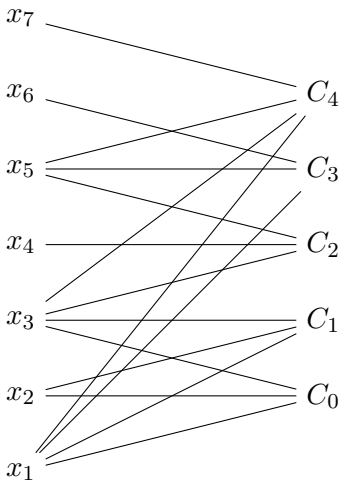

Figure 2.3: The incidence graph $\mathcal{G}_{\mathsf{inc}}(F_{\mathsf{ex}})$ of F_{ex}

section, we introduce another graph, the incidence graph, which characterizes the structure of the formula more accurately.

2.1.2 Incidence graph and hypergraph

The *incidence graph* of a formula is a bipartite graph whose vertices are both the variables and the clauses of the formula. There is an edge between a variable x and a clause C if and only if the variable x is in $\mathsf{var}(C)$. Again, the negations on the variables do not appear in the incidence graph anymore. We denote this graph $\mathcal{G}_{\mathsf{inc}}(F)$. Formally, $\mathcal{G}_{\mathsf{inc}}(F) = (\mathsf{var}(F), F, E)$ where $E = \{\{x, C\} \mid x \in \mathsf{var}(F), C \in F, x \in \mathsf{var}(C)\}$. Figure 2.3 pictures the incidence graph of the formula F_{ex}.

The *hypergraph* of a formula, denoted by $\mathcal{H}(F)$, is the hypergraph whose vertices are the variables of F and the hyperedges are the variables of each clauses. Formally, $\mathcal{H} = \{\mathsf{var}(C) \mid C \in F\}$. Figure 2.4 pictures the hypergraph of formula F_{ex}.

Observe that the incidence graph of $\mathcal{H}(F)$ is very close to $\mathcal{G}_{\mathsf{inc}}(F)$ but if F holds clauses C_1, C_2 having the same variables, that is $\mathsf{var}(C_1) = \mathsf{var}(C_2)$, it generates only one hyperedge in $\mathcal{H}(F)$ but two different vertices having the same neigh-

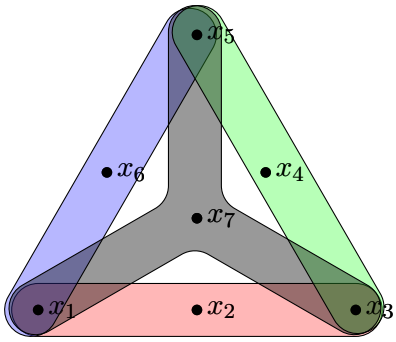

Figure 2.4: The hypergraph $\mathcal{H}(F_{\mathsf{ex}})$ of F_{ex}

borhood for C_1, C_2. In graph theory, two vertices having the same neighborhood are called *modules*. Our observation is equivalent to say that the incidence graph of the hypergraph $\mathcal{H}(F)$ is obtained by contracting every module of $\mathcal{G}_{\mathsf{inc}}(F)$ that corresponds to clauses of F.

This is illustrated in F_{ex}: clauses C_0 and C_1 have the same variables $\{x_1, x_2, x_3\}$ but $\mathcal{H}(F_{\mathsf{ex}})$ has only one hyperedge $\{x_1, x_2, x_3\}$ when $\mathcal{G}_{\mathsf{inc}}(F_{\mathsf{ex}})$ has two modules C_0, C_1.

Most of the graph decompositions we will be interested in for formulas are stable by contracting modules so it is usually equivalent to use the hypergraph or the incidence graph of a formula but one has to be careful since they do not quite define the same object.

A closely related notion is the *signed incidence graph* of a formula. It is an oriented version of the incidence graph, where an edge between a variable and a clause is oriented depending on the sign of the variable in this clause. Formally, the signed incidence graph $\mathcal{G}^+_{\mathsf{inc}}(F)$ of a CNF-formula F is the oriented bipartite graph $\mathcal{G}^+_{\mathsf{inc}}(F) = (\mathsf{var}(F), F, E)$ where $E = \{(x, C) \mid x \in \mathsf{lit}(C)\} \cup \{(C, x) \mid \neg x \in \mathsf{lit}(C)\}$. Figure 2.5 illustrates the signed incidence graph of F_{ex}.

Observe that no information on the formula is lost in the signed incidence graph. Indeed, one can easily reconstruct the formula given only $\mathcal{G}^+_{\mathsf{inc}}(F)$ and which subset of variables corresponds to the clauses of F. However, this graph is less useful for studying structural restrictions of #SAT since most well-studied graph measures concern non-oriented graphs. Signed clique-width of the signed incidence graph has however been used to design an FPT-algorithm for #SAT [FMR08] and it is not excluded that progress in graph measures for directed graphs may yield new tractable classes for #SAT.

2.1.3 Structural restriction of CNF-formulas

We now have the tools needed to explain more precisely what we intend by structural restriction of a CNF-formula. These restrictions will be defined by structural conditions on one of their associated graphs (primal, dual, incidence, hypergraph).

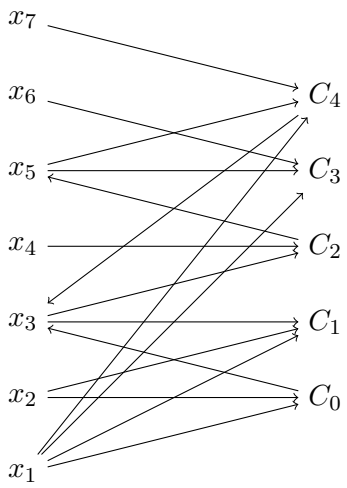

Figure 2.5: The signed incidence graph $\mathcal{G}^+_{\mathsf{inc}}(F_{\mathsf{ex}})$ of F_{ex}

Given a graph width p, we will refer to the primal (resp. dual, incidence) p-width of a CNF-formula as the p-width of its primal (resp. dual, incidence) graph. Given a hypergraph width p, the p-width of the formula is the width p of its hypergraph. Given a hypergraph property $\mathcal{P}$, we refer to a formula F having the property $\mathcal{P}$ if $\mathcal{P}$ is true on $\mathcal{H}(F)$.

For instance, the primal tree width of a CNF-formula F is defined to be the tree width of the primal graph of F, in symbols $\mathbf{tw}(\mathcal{G}_{\mathsf{prim}}(F))$ and the incidence tree width of F is the tree width of its incidence graph, in symbols $\mathbf{tw}(\mathcal{G}_{\mathsf{inc}}(F))$. Similarly, a formula F is said β-acyclic if $\mathcal{H}(F)$ is β-acyclic.

The primal tree width of F_{ex} is 3 since $\mathcal{G}_{\mathsf{prim}}(F)$ contains a 4-clique and the decomposition having bag $\{x_1, x_3, x_5, x_7\}$ connected to the bags $\{x_3, x_4, x_5\}$, $\{x_1, x_5, x_6\}$ and $\{x_1, x_2, x_3\}$ is a tree decomposition of $\mathcal{G}_{\mathsf{prim}}(F)$ of width 3. It can be shown similarly that the dual tree width and the incidence tree width of F_{ex} are also 3. The incidence tree width of a formula is however usually smaller than its primal and dual tree width. The most striking example is the formula having only one clause of n variables. Its primal graph is an n-clique, therefore, the primal tree width of the formula is n. However its incidence graph is a tree and thus the incidence tree width of this formula is 1. More generally, the following holds:

Theorem 2.1. *For all* CNF*-formula F, the incidence tree width of F is smaller than its primal tree width and its dual tree width.*

Proof. Let F be a CNF-formula of primal tree width k and let $(x_1, \ldots, x_n)$ be an elimination order of $\mathsf{var}(F)$ of width k, given by Theorem 1.27. Let $(C_1, \ldots, C_m)$ be an arbitrary order on F. We claim that $(C_1, \ldots, C_m, x_1, \ldots, x_n)$ is an elimination order of $\mathcal{G}_{\mathsf{inc}}(F)$ of width a most k. Indeed, by removing a clause C in $\mathcal{G}_{\mathsf{inc}}(F)$, the only edges that are added are $\{(x, y) \mid x, y \in \mathsf{var}(C)\}$. Thus after removing every clauses in $\mathcal{G}_{\mathsf{inc}}(F)$, we end up with the primal graph of F.

Moreover, when we remove a vertex corresponding to a clause C, its degree in the graph is $|C|$ since we never add new edges incident to clauses (only edges between variables). Thus, the width of $(C_1, \ldots, C_m, x_1, \ldots, x_n)$ is $\max(\max\{|C| \mid C \in F\}, k)$. Since each clause $C \in F$ induces a $|C|$-clique in $\mathcal{G}_{\mathsf{prim}}(F)$, we have $k \geq \max\{|C| \mid C \in F\}$, that is, $(C_1, \ldots, C_m, x_1, \ldots, x_n)$ is of width k and $\mathbf{tw}(\mathcal{G}_{\mathsf{inc}}(F)) \leq k = \mathbf{tw}(\mathcal{G}_{\mathsf{prim}}(F))$.

The proof for dual tree width is symmetric. It is sufficient to remove the variables of F first in $\mathcal{G}_{\mathsf{inc}}(F)$ and then use an elimination order of $\mathcal{G}_{\mathsf{dual}}(F)$ of width k on the clauses. □

#SAT is said to be tractable on a class $\mathcal{C}$ of formulas if there exists a polynomial time algorithm that given a formula F rejects if $F \notin \mathcal{C}$ and outputs $\#F$ otherwise. In a context of structurally defined classes of formulas, proving that a class $\mathcal{C}$ is tractable is often done in two steps: first, it is shown that if $F \in \mathcal{C}$, then there exists a decomposition of the formula that can be used to count its number of satisfying assignments in polynomial time. Then it is shown that there exists a polynomial time algorithm that given a formula decides if $F \in \mathcal{C}$ and if so, outputs the needed decomposition for the algorithm to work. This scenario is illustrated in more details in Section 2.2, where we give a complete study of the tractability of #SAT on so-called disjoint branches formulas as an example. In Section 2.3.1, we give a very broad class of formulas for which one can show that, if provided a good decomposition of the formula, one can count its satisfying assignments in polynomial time. However, for this class, the problem of deciding if one can construct such a decomposition is still open.

2.2 A first tractable class: disjoint branches

In this section, we introduce a hypergraph acyclicity (see [Dur12]), called disjoint branches acyclicity, which fits between γ-acyclicity and β-acyclicity. We first present its definition and its relations to other hypergraph acyclicity notions and give a dynamic programming algorithm to perform model counting on disjoint branches formulas. Finally, we give a polynomial time algorithm which accepts and outputs a disjoint branches decomposition of the input hypergraph if it exists, and rejects otherwise.

The model counting algorithm presented here is superseded by results from Section 2.3.1 or Chapter 4. We choose however to explain it here as an easy illustration of how one can use such decompositions to solve #SAT in polynomial time. Indeed, most of the known algorithms for #SAT, presented in Section 2.3 perform this kind of dynamic programming on the formula and this example may help the reader to grasp the main techniques used in this area. The reader only interested in state of the art results may safely skip to Section 2.3. Section 2.2.3 presents however a result of independent interest since it gives an algorithm to compute a new hypergraph decomposition that could be used for different problems. All results of this section are adapted from [CDM14].

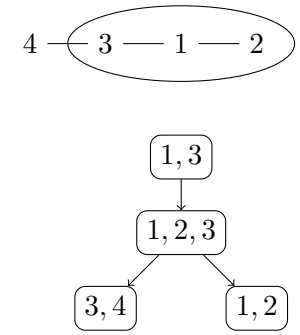

Figure 2.6: A disjoint branches acyclic hypergraph

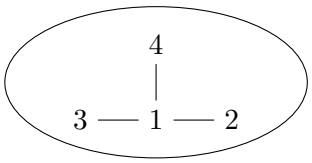

Figure 2.7: A β-acyclic hypergraph that is not disjoint branches acyclic.

2.2.1 Disjoint branches hypergraphs

Disjoint branches decompositions were introduced in [Dur12] as follows:

Definition 2.2. *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 $\lambda(t) \cap \lambda(t') = \emptyset$.*

A hypergraph with a disjoint branches decomposition is called disjoint branches acyclic. Figure 2.6 pictures a disjoint branches acyclic hypergraph and a disjoint branches decomposition of this hypergraph.

Disjoint branches acyclic hypergraphs are obviously α-acyclic since a disjoint branches decomposition is a join tree. Duris showed in [Dur12] that such a hypergraph is even β-acyclic. It can be easily seen as follows: in a given branch decomposition, if one removes a bag and connects all its children to its father, then one gets a new disjoint branches decomposition. If one removes the root and connects its first child to all its other children, then again, one gets a new disjoint branches decomposition. Thus, the property of having a disjoint branches decomposition is hereditary. In particular, every subhypergraph of a disjoint branches acyclic hypergraph are α-acyclic, that is, every disjoint branches acyclic hypergraph is β-acyclic. The converse is not true however. The hypergraph $\mathcal{H}$ depicts in Figure 2.7 is β-acyclic since $(2, 3, 4, 1)$ is a β-elimination order of $\mathcal{H}$. Moreover, assume that there exists a disjoint branches decomposition of $\mathcal{H}$. Since 1 is in all edges, this decomposition has to be a path of length 4. On this path, at least one edge $\{1, i\}$ for $i = 2, 3, 4$ is not a neighbor of the edge $\{1, 2, 3, 4\}$. Assume this is $\{1, 2\}$ (the other cases are symmetric). Since the only edges holding 2 are

$\{1,2\}$ and $\{1,2,3,4\}$ and since they are not neighbors, 2 is not connected in the decomposition. Therefore $\mathcal{H}$ is not disjoint branches acyclic.

Duris [Dur12] also showed an intriguing connection between γ-acyclicity and disjoint branches decompositions: a hypergraph $\mathcal{H}$ is γ-acyclic if and only if for all edge $e \in \mathcal{H}$, there exists a disjoint branches decomposition rooted in this edge. Thus γ-acyclic hypergraphs are also disjoint branches acyclic. The converse is not true however and it can be proved that the hypergraph depicted in Figure 2.6 is not γ-acyclic. This shows how disjoint branches decompositions are sensible to the choice of root, a problem which is particularly challenging when it comes to constructing a disjoint branches decomposition of a given hypergraph, as explained in Section 2.2.3.

2.2.2 Model counting of disjoint branches formula

In this section we will show that #SAT restricted to hypergraphs with a disjoint branches decomposition can be solved in polynomial time. In the following, we let F be a disjoint branches acyclic formulas and $(\mathcal{T}, \lambda)$ a disjoint branches decomposition of $\mathcal{H}(F)$. The principle of the algorithm is as follows: we associate a subformula F_t of F to each vertex t of the decomposition and then describe a dynamic programming on $\mathcal{T}$ to infer the number of solution of F_t from the number of solutions of $F_{t_1}, \dots, F_{t_k}$, where $t_1, \dots, t_k$ are the children of t in $\mathcal{T}$.

We now define the notations that we will use to describe the algorithm. We recall that given a hyperedge $e \in \mathcal{H}(F)$, there may be more than one clause C of F such that $\mathsf{var}(C) = e$. This makes the notations a bit heavier. For a vertex t of $\mathcal{T}$, we denote $C_t = \bigwedge_{C \in F, \mathsf{var}(C) = \lambda(t)} C$ the set of clauses corresponding to the bag $\lambda(t)$. We denote $F_t = \bigwedge_{u \in V(\mathcal{T}_t)} C_u$, the subformula of F that has all clauses corresponding to bags in the subtree of $\mathcal{T}$ rooted in t. Observe that if t has children $t_1, \dots, t_k$ in $\mathcal{T}$, then $F_t = C_t \wedge \bigwedge_{i=1}^{k} F_{t_i}$ and by disjointness, for all $i < j \leq k$, we have $\mathsf{var}(F_{t_i}) \cap \mathsf{var}(F_{t_j}) = \emptyset$. We denote by $V_t = \mathsf{var}(F_t) = \bigcup_{u \in \mathcal{T}_t} \lambda(u)$. For a clause $C \in F$, we denote by $a_C : \mathsf{var}(C) \to \{0,1\}$ the only assignment of $\mathsf{var}(C)$ that do not satisfy C. That is $a_C(x) = 0$ if $x \in C$ and $a_C(x) = 1$ if $\neg x \in C$.

We recall that for truth assignments a, b, we denote by $a \simeq b$ if $a(x) = b(x)$ for every x where they are both defined. We also denote by $a \subseteq b$ if $a \simeq b$ and if every variable that is assigned by a is also assigned by b. Given a formula G and $X \supseteq \mathsf{var}(G)$ and an assignment b of $Y \subseteq X$, we denote by $\mathsf{sat}_X(G, b) = \{a : X \to \{0,1\} \mid a \models G \text{ and } a \simeq b\}$ the set of satisfying assignments of G on variables X that are compatible with b. If b is the empty assignment, we only write $\mathsf{sat}(G)$. We omit the subscript X when $X = \mathsf{var}(G)$. We have the following:

Lemma 2.3. *Let $F_1, \dots, F_k$ be* CNF*-formulas such that for all $i \neq j$, $\mathsf{var}(F_i) \cap \mathsf{var}(F_j) = \emptyset$. Let $V = \bigcup_{i=1}^{k} \mathsf{var}(F_i)$, $X \supseteq V$, $Y \subseteq X$ and $a : Y \to \{0,1\}$. Then*

$$|\mathsf{sat}_X\big((\bigwedge_{i=1}^{k} F_i, a)\big)| = 2^{|X \setminus (V \cup Y)|} \prod_{i=1}^{k} |\mathsf{sat}(F_i, a)|.$$

Proof. Observe that if two CNF-formulas A and B do not share variables then $\mathsf{sat}(A \wedge B) = \{c \cup d \mid c \in \mathsf{sat}(A), d \in \mathsf{sat}(B)\}$. Hence, since $\mathsf{var}(F_i[a]) \cap \mathsf{var}(F_j[a]) = \emptyset$ for $i \neq j$, we have $|\mathsf{sat}(\bigwedge_{i=1}^{k} F_i, a)| = \prod_{i=1}^{k} |\mathsf{sat}(F_i, a)|$. Now the variables that do not appear in Y nor in V are free and their values do not influence the satisfiability of F_i. Thus

$$|\mathsf{sat}_X(\bigwedge_{i=1}^{k} F_i, a)| = 2^{|X \setminus (V \cup Y)|} \prod_{i=1}^{k} |\mathsf{sat}(F_i, a)|.$$

□

Lemma 2.4. *Let t be a vertex of $\mathcal{T}$ with children $t_1, \ldots, t_k$. Let $Y \subseteq \lambda(t)$ and let $b : Y \to \{0,1\}$. Let $W = V_t \setminus (Y \cup \bigcup_{i=1}^{k} V_{t_i})$. We have*

$$|\mathsf{sat}(F_t, b)| = 2^{|W|} \prod_{i=1}^{k} |\mathsf{sat}(F_{t_i}, b)| - \sum_{\substack{C \in C_t \\ b \subseteq a_C}} \prod_{i=1}^{k} |\mathsf{sat}(F_{t_i}, a_C)|.$$

Proof. To ease notation, denote $G = \bigwedge_{i=1}^{k} F_{t_i}$. We have $F_t = C_t \wedge G$. We start by counting the number of satisfying assignments of G compatible with b and remove from them the satisfying assignments of G that do not satisfy C_t.

For all $i \neq j$, we have $\mathsf{var}(F_{t_i}) \cap \mathsf{var}(F_{t_j}) = \emptyset$ since the decomposition is disjoint branches. Thus by Lemma 2.3:

$$|\mathsf{sat}_{V_t}(G, b)| = 2^{|W|} \prod_{i=1}^{k} |\mathsf{sat}(F_{t_i}, b)|.$$

Now it is sufficient to count the assignments $a \in \mathsf{sat}_{V_t}(G, b)$ such that $a \notin \mathsf{sat}(F_t, b)$. Let a be in $\mathsf{sat}_{V_t}(G, b) \setminus \mathsf{sat}(F_t, b)$. In particular $b \subseteq a$. Moreover since a already satisfies G, it means $a \not\models C_t$, that is, there exists $C \in C_t$ such that $a_C \subseteq a$. Again by disjointness, the number of assignments of $\mathsf{sat}_{V_t}(G)$ that verifies $a_C \subseteq a$ is, by Lemma 2.3:

$$\prod_{i=1}^{k} |\mathsf{sat}(F_{t_i}, a_C)|.$$

Finally, for all $C, C' \in C_t$, if $C \neq C'$ then $a_C \neq a_{C'}$ since $\mathsf{var}(C) = \mathsf{var}(C')$. Thus if $a_C \subseteq a$ then $a_{C'} \not\subseteq a$. That is, the number of assignments a such that $a \models G$, $b \subseteq a$ and $a \not\models C_t$ is:

$$\sum_{\substack{C \in C_t \\ b \subseteq a_C}} \prod_{i=1}^{k} |\mathsf{sat}(F_{t_i}, a_C)|.$$

□

Lemma 2.4 can be used to compute the number of satisfying assignments of a disjoint branches formula by dynamic programming if a disjoint branches decomposition is given as shown by the following theorem:

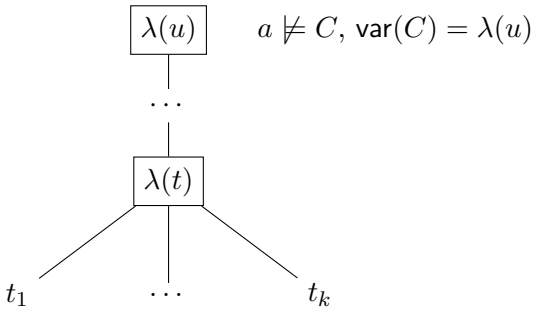

Figure 2.8: Notations for the proof of Theorem 2.5

Theorem 2.5. *Given F a disjoint branches formula and $(\mathcal{T}, \lambda)$ a disjoint branches decomposition of $\mathcal{H}(F)$, one can compute $|\mathsf{sat}(F)|$ in polynomial time.*

Proof. We compute the number of satisfying assignments of F_t for all vertices t of $\mathcal{T}$ on well-chosen partial truth assignments by dynamic programming. More precisely, for every vertex t of $\mathcal{T}$, for every ancestor u of t in $\mathcal{T}$ and for every $C \in F$ such that $\mathsf{var}(C) = \lambda(u)$, we compute the value $\left|\mathsf{sat}_{V_t}(F_t, a_C|_{\lambda(t)})\right|$ and $|\mathsf{sat}(F_t)|$. Figure 2.8 represents the notations used in this proof.

If we manage to compute these values for all t, then choosing t to be the root of the tree leads to $F = F_t$ and then $|\mathsf{sat}(F_t)| = |\mathsf{sat}(F)|$ is computed. Moreover, for each t, we have to compute at most $O(|F|)$ values and there are at most $|F|$ such t. Thus, we have at most $O(|F|^2)$ values to compute. It is thus enough to show that each value can be computed in polynomial time to prove the theorem.

We do this by induction. If t is a leaf then F_t is the conjunction of clauses having the same variables $\lambda(t)$. For each clause C of F_t, there exists exactly one assignment of $\lambda(t)$ that do not satisfy C. Thus, the number of assignments of $\lambda(t)$ that do not satisfy F_t is exactly $|F_t|$. That is $|\mathsf{sat}(F_t)| = 2^{|\lambda(t)|} - |F_t|$. Similarly, if u is an ancestor of t and $C \in F$ such that $\mathsf{var}(C) = \lambda(u)$, then let $a = a_C|_{\lambda(t)}$. We just have to go over all clauses of F_t to detect the assignments of $\lambda(t)$ which do not satisfy F_t and deduce $|\mathsf{sat}(F_t, a)|$ from it. This initialization can be done in polynomial time.

Now let t be a vertex of $\mathcal{T}$ with children $t_1, \ldots, t_k$ and assume that all values have been precomputed for the other vertices of $\mathcal{T}_t$. See Figure 2.8. We denote by $W = V_t \setminus (\bigcup_{i=1}^{k} V_{t_i})$. By Lemma 2.4,

$$|\mathsf{sat}(F_t)| = 2^{|W|} \prod_{i=1}^{k} |\mathsf{sat}(F_{t_i})| - \sum_{C \in C_t} \prod_{i=1}^{k} |\mathsf{sat}(F_{t_i}, a_C)|. \tag{2.1}$$

Let $i \leq k$. Observe that $V_{t_i} \cap \lambda(t) \subseteq \lambda(t_i)$ by connectedness of variables in $\mathcal{T}$. Since for all $C \in C_t$, a_C assigns variables in $\lambda(t)$, we have $\mathsf{sat}(F_{t_i}, a_C) = \mathsf{sat}(F_{t_i}, a_C|_{V_t \cap V_{t_i}}) = \mathsf{sat}(F_{t_i}, a_C|_{\lambda(t_i)})$. Since t is an ancestor of t_i, each term in Equation 2.1 has been precomputed and one can thus compute $|\mathsf{sat}(F_t)|$ with $O(k|C_t|)$ arithmetic operations.

Now let u be an ancestor of t and $D \in C_u$. We denote by $Y = \lambda(t) \cap \lambda(u)$ and $W = V_t \setminus (Y \cup \bigcup_{i=1}^{k} V_{t_i})$ the vertices of $\lambda(t)$ that appear neither in the labels of its children nor in those of u. Let $b = a_D|_{\lambda(t)}$. The domain of b is $Y \subseteq \lambda(t)$, thus by Lemma 2.4 again,

$$|\mathsf{sat}(F_t, b)| = 2^{|W|} \prod_{i=1}^{k} |\mathsf{sat}(F_{t_i}, b)| - \sum_{\substack{C \in C_t \\ a_C \subseteq b}} \prod_{i=1}^{k} |\mathsf{sat}(F_{t_i}, a_C)|. \tag{2.2}$$

Let $i \leq k$. As before $\mathsf{sat}(F_{t_i}, a_C) = \mathsf{sat}(F_{t_i}, a_C|_{\lambda(t_i)})$, and thus, the cardinal of this set has already been computed. Moreover, $V_{t_i} \cap \lambda(u) \subseteq \lambda(t) \cap \lambda(t_i)$ by connectedness of variables in $\mathcal{T}$. Thus $a_D|_{V_{t_i}} = a_D|_{\lambda(t) \cap \lambda(t_i)} = b|_{\lambda(t_i)}$. Thus $\mathsf{sat}(F_{t_i}, b) = \mathsf{sat}(F_{t_i}, b|_{\lambda(t_i)})$ is a precomputed value. Thus each term in Equation 2.2 has been precomputed and one can thus compute $|\mathsf{sat}(F_t, b)|$ with $O(k|C_t|)$ arithmetic operations.

In the end, one can compute the number of satisfying assignments of F in polynomial time by dynamic programming. □

We do not exhibit the exact runtime of the algorithm given in Theorem 2.5 nor do we try to improve it since this result is generalized in Chapter 4 with a better algorithm for a larger class of formula. Our purpose here is only to give an illustration of the type of algorithms used for the tractability results reviewed in Section 2.3, hence we rather like presenting the main ideas of the dynamic programming than optimizing the runtime of the algorithm.

2.2.3 Finding a disjoint branches decomposition

Theorem 2.5 gives an algorithm that computes the number of satisfying assignments of a disjoint branches CNF-formula if a disjoint branches decomposition is given in the input. Theorem 2.5 is however not enough to ensure the tractability of #SAT on disjoint branches instances because a disjoint branches decomposition may still be NP-hard to compute. This section is dedicated to the proof of Theorem 2.13 that states that one can construct a disjoint branches decomposition in polynomial time. The tractability of #SAT on disjoint branches instances thus follows from combining both Theorem 2.5 and Theorem 2.13.

Our decision algorithm relies on a precise understanding of the structure of disjoint branches decomposition (described in Theorem 2.11) and on PQ-trees, a powerful data structure, introduced by Booth and Lueker [BL76], to decide the existence of join paths having nice properties (see Theorem 2.10).

To the best of our knowledge it is the first polynomial time algorithm for constructing disjoint branches decompositions of hypergraphs. Gavril [Gav75] has proposed an algorithm for recognizing intersection graphs of directed paths in directed trees, a related notion that has been used successfully for quickly answering queries in probabilistic databases [KGS13]. The intersection graph associated to a directed tree T and V a set of paths in T is the graph on vertices V such that v, w

are connected by an edge if and only if the paths v and w intersect. This is related to disjoint branches: given a disjoint branches decomposition T of $\mathcal{H}$, $V(\mathcal{H})$ naturally defines a set of paths in T that are for each $x \in \mathcal{H}$, the path of vertices of T holding x. The intersection graph of such a family is the primal graph of $\mathcal{H}$. However, this is not an equivalence since every clique is an intersection graph of such family. However, the primal graph of the hypergraph of Figure 2.7 is a clique and the hypergraph is not disjoint branches acyclic. Thus, the work of Gavril is not appropriate in our settings.

Join paths. Our algorithm relies on a notion close to join tree: join paths. A *join path* for a hypergraph $\mathcal{H}$ is a join tree whose underlying tree is a path. More formally, a join path for $\mathcal{H}$ is a path $\mathcal{P}$ together with a labeling function λ from the vertices of $\mathcal{P}$ to $\mathcal{H}$ such that:

- for every $e \in \mathcal{H}$, there exists a unique vertex p of $\mathcal{P}$ such that $\lambda(p) = e$ and,
- for every $x \in V(\mathcal{H})$, the set of vertices p of $\mathcal{P}$ such that $x \in \lambda(p)$ is a connected subpath of $\mathcal{P}$.

Observe that if we have a disjoint branches decomposition $\mathcal{T}$ of $\mathcal{H}$, then for every $x \in V(\mathcal{H})$, the edges that contain x are on the same path of $\mathcal{T}$, hence, the edges of $\mathcal{H}$ that contains x have a join path.

Outline of the algorithm. Our recognition algorithm works roughly as follows: we start by choosing an edge e of the input hypergraph $\mathcal{H}$ and try to construct a disjoint branches decomposition of $\mathcal{H}$ rooted in e. For this, we characterize the disjoint branches decomposition of $\mathcal{H}$ rooted in e: we pick $x \in e$ and observe that in such a decomposition, the edges of $\mathcal{H}_x = \{f \in \mathcal{H} \mid x \in f\}$ lie on a join path starting from e and increasing for a certain preorder. On this path, disjoint branches decompositions of the different connected components $\mathcal{H}_1, \ldots, \mathcal{H}_k$ of $\mathcal{H} \setminus \mathcal{H}_x$ are grafted. More importantly, we can find unique edges $f_1, \ldots, f_k$ in $\mathcal{H}_x$ that have to be used as roots for these decompositions, allowing us to recursively call the algorithm on each $\mathcal{H}_i$. Figure 2.9 pictures the structure of such decompositions.

There is one remaining difficulty: we have to find join paths increasing for a given preorder. This is achieved by using PQ-trees, a powerful data structure which can be efficiently computed [BL76] and used to encode every join path of a hypergraph. We then extract from this representation a join path respecting a given preorder.

PQ-trees. PQ-trees are a data structure [BL76] originally defined to check matrices for the so-called consecutive ones property. This problem can be reformulated as follows in our setting: given a hypergraph $\mathcal{H}$, is there an ordering $(e_1, \ldots, e_m)$ of the edges such that if $v \in e_i \cap e_j$, then for all $i \leq k \leq j$, $v \in e_k$? In other words, is there a join path for $\mathcal{H}$ that is, a join tree whose underlying tree is a path?

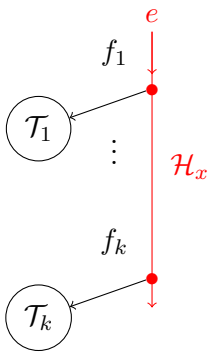

Figure 2.9: An illustration of the decomposition given by Theorem 2.11

We now give the definition of PQ-trees and the properties we will be interested in.

Definition 2.6. *Let $\mathcal{H}$ be a hypergraph. A PQ-*tree *for $\mathcal{H}$ is an ordered tree with leaf set $\mathcal{H}$ such that*

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

PQ-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 2.7. *The* frontiers *$\mathcal{F}(T)$ of a PQ-tree T for $\mathcal{H}$ are a set of ordered list of elements of edges defined inductively by*

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

  where T_i is the subtree of T rooted in t_i.

For example, the frontier of the PQ-tree depicted in Figure 2.10 is:

$$\mathcal{F}(T) = \{(e_1, e_2, e_3, e_4), (e_2, e_1, e_3, e_4), (e_4, e_3, e_2, e_1), (e_4, e_3, e_1, e_2)\}.$$

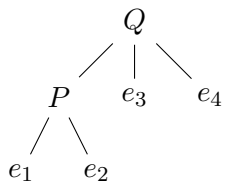

Figure 2.10: A PQ-tree

Observe that a PQ-tree is a handy way of representing a set of size possibly exponential with a structure that is polynomial in the number of edges in the hypergraph.

We recall the main theorem of [BL76], which allows to represent all possible join paths of a hypergraph in polynomial time.

Theorem 2.8 ([BL76])**.** *Given a hypergraph $\mathcal{H}$, one can compute a PQ-tree T in time $O(|\mathcal{H}| \cdot |V(\mathcal{H})|)$ such that $\mathcal{F}(T)$ is exactly the set of join paths for $\mathcal{H}$.*

We say that a PQ-tree T *represents* the join paths of $\mathcal{H}$ if $\mathcal{F}(T)$ is exactly the set of join paths of $\mathcal{H}$. Theorem 2.8 states that one can easily find, given a hypergraph, a PQ-tree representing $\mathcal{H}$.

Our main concern for computing disjoint branches decompositions of hypergraph is to find join paths having the additional property of respecting a given preorder. More precisely, let $\mathcal{H}$ be an hypergraph and $\prec$ be a preorder on $\mathcal{H}$. We say that a join path $(e_1, \ldots, e_m)$ for $\mathcal{H}$ is compatible with $\prec$ if for all $i < j$, $(e_j \not\prec e_i)$. Fortunately, extracting compatible join paths from a PQ-tree can be done in polynomial time.

We need some notations. Given a PQ-tree T representing a hypergraph $\mathcal{H}$ and a vertex t of T, we denote by $\mathcal{H}(t) \subseteq \mathcal{H}$ the labels of the leaves of T_t. Moreover, given a preorder $\prec$ on $\mathcal{H}$, $A \subseteq \mathcal{H}$ and $B \subseteq \mathcal{H}$, we write $A \preceq B$ if for all $a \in A$ and $b \in B$, $\neg(a \prec b)$.

Lemma 2.9. *Let $\mathcal{H}$ be a hypergraph, $\prec$ a preorder on $\mathcal{H}$ such that there exists a join path $\mathcal{P}$ of $\mathcal{H}$ compatible with $\prec$ and T a PQ-tree representing $\mathcal{H}$. Then for any non-leaf vertex t of T with children $t_1, \ldots, t_k$, the following holds:*

- *if t is a P-node, then there exists a permutation σ of $[k]$ such that for all $i < j$, $\mathcal{H}(t_{\sigma(i)}) \preceq \mathcal{H}(t_{\sigma(j)})$*
- *if t is a Q-node, then either for all $i < j$, $\mathcal{H}(t_i) \preceq \mathcal{H}(t_j)$ or for all $i < j$, $\mathcal{H}(t_j) \preceq \mathcal{H}(t_i)$.*

Proof. Since T represents all join paths of $\mathcal{H}$, $\mathcal{P} \in \mathcal{F}(T)$. For each node t of T, we choose a permutation σ_t of its children (among the identity or the reverse order) such that the resulting path is $\mathcal{P}$. We let $\mathcal{P}(t)$ be the join path for $\mathcal{H}(t)$ it defines.

Let t be a P-node and let σ be the chosen permutation of its children. We have $\mathcal{P}(t) = \mathcal{P}(t_{\sigma(1)}) \ldots \mathcal{P}(t_{\sigma(k)})$. We claim that $\mathcal{P}(t)$ is a compatible join path

of $\mathcal{H}(t)$. Indeed if there exists e, f in $\mathcal{P}(t)$ with e appearing before f in $\mathcal{P}(t)$ and $f \prec e$, then e appears before f in $\mathcal{P}$ and $f \prec e$, contradicting the fact that $\mathcal{P}$ is a compatible join path. Thus, for all $i < j$, $e \in \mathcal{H}(t_{\sigma(i)})$ and $f \in \mathcal{H}(t_{\sigma(j)})$, since e appears before f in $\mathcal{P}(t)$, we have $\neg(e \prec f)$. That is $\mathcal{H}(t_{\sigma(i)}) \preceq \mathcal{H}(t_{\sigma(j)})$.

The case where t is a Q-node is similar, since we also have a permutation of its children, only here, the permutation is either the identity or the reverse order. □

Theorem 2.10. *Let $\mathcal{H}$ be a hypergraph and $\prec$ a preorder on $\mathcal{H}$. One can decide if a join path of $\mathcal{H}$ compatible with $\prec$ exists and, if so, construct it in polynomial time.*

Proof. We start by constructing in polynomial time a PQ-tree T such that $\mathcal{F}(T)$ is exactly the set of join paths of $\mathcal{H}$, which is possible by Theorem 2.8.

We then check for all vertices t of T that it respects the properties listed in Lemma 2.9. This can be done in polynomial time. Indeed, if t has children $t_1, \dots, t_k$, we start by deciding if $\mathcal{H}(t_i) \preceq \mathcal{H}(t_j)$ for all $i < j$. For Q-nodes, we check that for every $i < k$, $\mathcal{H}(t_i) \preceq \mathcal{H}(t_{i+1})$ or that for every $i < k$, $\mathcal{H}(t_{i+1}) \preceq \mathcal{H}(t_i)$. For P-nodes, we try to do a topological sort of $\mathcal{H}_i$ according to $\preceq$.

If we find a vertex t in T which does not meet these conditions, then by Lemma 2.9, there exists no join path for $\mathcal{H}$ compatible with $\prec$. Hence we return an error and fail.

Otherwise, we inductively construct bottom-up a join path compatible for $\prec$. More precisely, for all node t of T, we construct a join path for $\mathcal{H}(t)$ compatible with $\prec$.

If t is a leaf labeled with $e \in \mathcal{H}$, then we return (e). It is obviously a join path of $\mathcal{H}_t = \{e\}$ compatible with $\prec$. If t has children $t_1, \dots, t_k$, we first construct by induction compatible join paths $\mathcal{P}_1, \dots, \mathcal{P}_k$ for $\mathcal{H}(t_1), \dots, \mathcal{H}(t_k)$.

Now, if t is a P-node, we know that there exists a permutation σ of $[k]$ such that for all $i < j$, $\mathcal{H}(t_{\sigma(i)}) \preceq \mathcal{H}(t_{\sigma(j)})$. We then return the join path $\mathcal{P}_t = (\mathcal{P}_{\sigma(1)} \dots \mathcal{P}_{\sigma(k)})$. It is a compatible path for $\mathcal{H}(t)$. Indeed, let e, f be two edges of $\mathcal{P}_t$ and assume that e appears before f in $\mathcal{P}_t$. Let $i \leq j$ be the such that e is in $\mathcal{P}_{\sigma(i)}$ and f is in $\mathcal{P}_{\sigma(j)}$. If $i = j$, then e appears before f in $\mathcal{P}_{\sigma(i)}$ and since $\mathcal{P}_{\sigma(i)}$ is compatible with $\prec$, we know that $\neg(e \prec f)$. Otherwise, if $i < j$, then by construction of σ, $\mathcal{H}_{\sigma(i)} \preceq \mathcal{H}_{\sigma(j)}$, that is, by definition, $\neg(e \prec f)$. Thus $\mathcal{P}_t$ is compatible. Since σ has been precomputed, we can do this step in polynomial time.

The case where t is a Q-node is similar. It is sufficient to observe that it is the same case where the permutation is either the identity or the reverse order of $[k]$. □

Rooted decompositions. We now turn to the main algorithm of this section, which computes disjoint branches decompositions of a hypergraph. To do so, we first explain how we can reduce this question to the computations of join paths. For a hypergraph $\mathcal{H}$ and a vertex $x \in V(\mathcal{H})$, we denote by $\mathcal{H}_x = \{e \in \mathcal{H} \mid x \in e\}$

the set of edges holding x. Given $e \in \mathcal{H}$, we say that $\mathcal{H}$ is *db-rootable in* e if there exists a disjoint branches decomposition of $\mathcal{H}$ such that the label of the root is e. Moreover, given $\mathcal{H}' \subseteq \mathcal{H}$, we define a relation $\prec_{\mathcal{H}'}$ on $\mathcal{H}'$ as for all $e, f \in \mathcal{H}'$, $e \prec_{\mathcal{H}'} f$ if and only if there exists a connected component $\mathcal{C}$ of $\mathcal{H} \setminus \mathcal{H}'$ such that $\emptyset \neq e \cap V(\mathcal{C}) \subsetneq f \cap V(\mathcal{C})$.

Theorem 2.11. *Let $\mathcal{H}$ be a connected hypergraph, $e \in \mathcal{H}$ and $x \in V(\mathcal{H})$. $\mathcal{H}$ is db-rootable in e if and only if the following holds:*

1. *for every connected component $\mathcal{C}$ of $\mathcal{H} \setminus \mathcal{H}_x$, $\mathcal{C} \cup \{V(\mathcal{H}_x) \cap V(\mathcal{C})\}$ is db-rootable in $\{V(\mathcal{H}_x) \cap V(\mathcal{C})\}$,*
2. *for all $f, g \in \mathcal{H}_x$, if there exists a connected component $\mathcal{C}$ of $\mathcal{H} \setminus \mathcal{H}_x$ such that $f \cap g \cap V(\mathcal{C}) \neq \emptyset$ then either $f \cap V(\mathcal{C}) \subseteq g$ or $g \cap V(\mathcal{C}) \subseteq f$,*
3. *the transitive closure $(\prec^*_{\mathcal{H}_x})$ of $(\prec_{\mathcal{H}_x})$ is a preorder,*
4. *there exists a join path for $\mathcal{H}_x$ compatible with $(\prec^*_{\mathcal{H}_x})$ starting with e.*

Proof. We start by proving that if $\mathcal{H}$ is db-rootable in e then items (1)-(4) hold. Let $(\mathcal{T}, \lambda)$ be a disjoint branches decomposition of $\mathcal{H}$ rooted in e. Since $\mathcal{T}$ is disjoint branches, exactly one branch of $\mathcal{T}$ contains the edges holding x. Since $x \in e$ and e is at the root of $\mathcal{T}$, we have a join path for $\mathcal{H}_x$ going down from e in $\mathcal{T}$.

Let $\{\mathcal{T}_1, \ldots, \mathcal{T}_k\}$ be the connected component of the forest resulting from removing from $\mathcal{T}$ the vertices labeled by an edge of $\mathcal{H}_x$, that is, we remove from $\mathcal{T}$ the set $\{u \in V(\mathcal{T}) \mid x \in \lambda(u)\}$. Let $\mathcal{C}$ be a connected component of $\mathcal{H} \setminus \mathcal{H}_x$. We show that there exists i such that $\mathcal{T}_i$ is a disjoint branches decomposition of $\mathcal{C}$. We denote by E_i the set of edges labeling the vertices of $\mathcal{T}_i$. Observe that $\mathcal{H} \setminus \mathcal{H}_x = \biguplus_{i=1}^{k} E_i$. Moreover, for $i \neq j$, $\mathcal{T}_i$ and $\mathcal{T}_j$ are hanged on two different branches of $\mathcal{T}$. Thus, $V(E_i) \cap V(E_j) = \emptyset$. Thus each E_i contains disjoint connected components of $\mathcal{H} \setminus \mathcal{H}_x$. That is there exists i such that $\mathcal{C} \subseteq E_i$.

Now we show that we actually have $\mathcal{C} = E_i$. Indeed, assume there exists $f \in E_i \setminus \mathcal{C}$. Let $\mathcal{C}'$ be the connected component of $\mathcal{H} \setminus \mathcal{H}_x$ such that $f \in \mathcal{C}'$. From what precedes, $\mathcal{C}' \subseteq E_i$ too. Moreover, $V(\mathcal{C}') \cap V(\mathcal{C}) = \emptyset$ since they are different connected components. Let e_i be the label of the root of $\mathcal{T}_i$. Assume $e_i \in \mathcal{C}$. By connectedness in $\mathcal{T}$, it means that $V(\mathcal{C}') \cap V(\mathcal{H}_x) = \emptyset$ since $V(\mathcal{C}') \cap e_i = \emptyset$. Thus $\mathcal{C}'$ is a connected component of $\mathcal{H}$. Since $\mathcal{H}$ is connected, we have $\mathcal{C}' = \mathcal{H}$, contradiction. Thus, $E_i = \mathcal{C}$.

We have shown that $\mathcal{T}$ can be decomposed as in Figure 2.9: a join path for $\mathcal{H}_x$ rooted in e on which disjoint branches decompositions for each connected component of $\mathcal{H} \setminus \mathcal{H}_x$ are grafted. For a given connected component $\mathcal{C}$ of $\mathcal{H} \setminus \mathcal{H}_x$, we denote by $\mathcal{T}_C$ the subtree of $\mathcal{T}$ corresponding to a disjoint branches decomposition of $\mathcal{C}$ and we let $f_C \in \mathcal{H}_x$ be the father of the root of $\mathcal{T}_i$, that is, the edge of $\mathcal{H}_x$ from which $\mathcal{T}_C$ hangs. By connectedness, for every connected component $\mathcal{C}$ of $\mathcal{H} \setminus \mathcal{H}_x$, we have $f_C \cap V(\mathcal{H}_x) = V(\mathcal{C}) \cap V(\mathcal{H}_x)$. Thus, $\mathcal{C} \cup \{V(\mathcal{C}) \cap V(\mathcal{H}_x)\}$ is db-rootable

in $V(\mathcal{C}) \cap V(\mathcal{H}_x)$ since replacing f_C by $V(\mathcal{C}) \cap V(\mathcal{H}_x)$ in the tree whose root is f_C attached to $\mathcal{T}_C$ yields the desired decomposition. This proves item 1.

Let $\mathcal{C}$ be a connected component of $\mathcal{H} \setminus \mathcal{H}_x$ and $g \in \mathcal{H}_x$ such that $g \cap V(\mathcal{C}) \neq \emptyset$. We prove that g is an ancestor of f_C. Indeed, if g is a descendant of f_C then it lies on a branch of $\mathcal{T}$ different from the branch of $\mathcal{T}_C$. Thus we have two disjoint branches in $\mathcal{T}$ having non-disjoint variable sets, contradiction.

Now let $f, g \in \mathcal{H}_x$ such that there exists a connected component $\mathcal{C}$ of $\mathcal{H}_x$ such that $f \cap g \cap V(\mathcal{C}) \neq \emptyset$. Assume f is an ancestor of g. Then $f \cap V(\mathcal{C}) \subseteq g$. Indeed, they are both ancestor of f_C. Thus, if $y \in f \cap V(\mathcal{C})$, $y \in g$ by connectedness of y since $y \in f_C$. This proves item 2.

Finally, observe that if $f \prec_{\mathcal{H}_x} g$ then f is an ancestor of g in $\mathcal{T}$. Indeed, by definition, there exists a connected component $\mathcal{C}$ of $\mathcal{H} \setminus \mathcal{H}_x$ such that $\emptyset \neq f \cap V(\mathcal{C}) \subsetneq g \cap V(\mathcal{C})$. From what precedes, if f is a descendant of g in $\mathcal{T}$ then $g \cap V(\mathcal{C}) \subseteq f \cap V(\mathcal{C})$, contradiction. Thus, f is an ancestor of g. From this, we get that $(\prec^*_{\mathcal{H}_x})$ is a preorder. Indeed, it is transitive by definition and if $e \prec^*_{\mathcal{H}_x} f$ and $f \prec^*_{\mathcal{H}_x} e$ then e is an ancestor of f and f is an ancestor of e, that is $e = f$. This proves item 3. Now let $\mathcal{P}$ be the join path of $\mathcal{H}_x$ induced by the branch of $\mathcal{T}$ containing $\mathcal{H}_x$. From what precedes, if f is after e in $\mathcal{P}$, then $\neg(e \prec^*_{\mathcal{H}_x} f)$, that is $\mathcal{P}$ is compatible with $(\prec^*_{\mathcal{H}_x})$, which concludes the first part of the proof.

We now prove that if items (1)-(4) hold then $\mathcal{H}$ has a disjoint branches decomposition rooted in e. Let $\mathcal{P}$ be a join path of $\mathcal{H}_x$ rooted in e compatible with $(\prec^*_{\mathcal{H}_x})$. We root $\mathcal{P}$ in e. Let $\mathcal{C}$ be a connected component of $\mathcal{H} \setminus \mathcal{H}_x$. We define f_C to be the last edge of $\mathcal{H}_x$ on $\mathcal{P}$ to have a non-empty intersection with $\mathcal{C}$. Observe that by item 2, $f_{\mathcal{C}} \cap V(\mathcal{C}) = V(\mathcal{H}_x) \cap V(\mathcal{C})$. Let $\mathcal{T}_C$ be the disjoint branches decomposition of $\mathcal{C} \cup \{V(\mathcal{H}_x) \cap V(\mathcal{C})\}$ rooted in $V(\mathcal{H}_x) \cap V(\mathcal{C})$. We replace the root of $\mathcal{T}_{\mathcal{C}}$ by $f_{\mathcal{C}}$ to have a disjoint branches decomposition of $\mathcal{C} \cup f_{\mathcal{C}}$ that we graft on $\mathcal{P}$ on $f_{\mathcal{C}}$. It is readily verified that the decomposition of $\mathcal{H}$ constructed is disjoint branches and rooted in e. □

We are now ready to give a polynomial time algorithm that decides, given a hypergraph $\mathcal{H}$ and $e \in \mathcal{H}$, if there exists a disjoint branches decomposition of $\mathcal{H}$ rooted in e.

Proposition 2.12. *Algorithm 3 is correct and runs in polynomial time.*

Proof. First, observe that since $\mathcal{H}$ is connected, if it is not reduced to e, then we can find $x \in e$ such that $|\mathcal{H}_x| > 1$. Now observe that we always call the procedure on strictly smaller connected hypergraphs since $|\mathcal{H}_x| > 1$.

Moreover, we call the procedure at most $|\mathcal{H}|$ times since we always call it on disjoint smaller hypergraphs and each step of the procedure can be done in polynomial time: it is well-known that we can compute connected components of a hypergraph in polynomial time. Moreover, computing a join path for $\mathcal{H}_x$ compatible with a preorder can be done in polynomial time by Theorem 2.10. Thus, Algorithm 3 runs in polynomial time.

The correctness of Algorithm 3 follows from Theorem 2.11 since we construct the decomposition as in the proof of this theorem. □

Algorithm 3: An algorithm to construct disjoint branches decompositions

Data: A connected hypergraph $\mathcal{H}$ and $e \in \mathcal{H}$
begin
  if $\mathcal{H} = \{e\}$ **then**
    return e
  else
    Choose $x \in e$ such that $|\mathcal{H}_x| > 1$;
    Compute the connected components $\mathcal{C}_1, \ldots, \mathcal{C}_k$ of $\mathcal{H} \setminus \mathcal{H}_x$;
    Check that for all $e, f \in \mathcal{H}_x$ and $i \leq k$:
      if $e \cap f \cap V(\mathcal{C}_i) \neq \emptyset$, then $e \cap V(\mathcal{C}_i) \subseteq f$ or $f \cap V(\mathcal{C}_i) \subseteq e$;
    if *there exists* e, f *that do not respect these conditions* **then**
      Rejects;
    Compute $\prec^*_{\mathcal{H}_x}$;
    if $\prec^*_{\mathcal{H}_x}$ *is not antisymetric* **then**
      Rejects;
    if *there exists a join path* $\mathcal{P}$ *for* $\mathcal{H}_x$ *compatible with* $\prec^*$ **then**
      $\mathcal{T} \leftarrow \mathcal{P}$, as a tree rooted in e ;
      for $i = 1, ..., k$ **do**
        Compute recursively $\mathcal{T}_i$ a disjoint branches decomposition of $\mathcal{C}_i \cup \{V(\mathcal{C}_i) \cap V(\mathcal{H}_x)\}$ rooted in $V(\mathcal{C}_i) \cap V(\mathcal{H}_x)$;
        $g_i \leftarrow$ the deepest edge f of $\mathcal{T}$ such that $f \cap V_i = \emptyset$;
        Hang $\mathcal{T}_i$ to g_i ;
      return $\mathcal{T}$;
    else
      Rejects

Theorem 2.13 ([CDM14])**.** *Given $\mathcal{H}$, one can decide in polynomial time if it has a disjoint branches decomposition.*

Proof. If $\mathcal{H}$ is connected, then we can try to compute a disjoint branches decomposition of $\mathcal{H}$ rooted in e for every $e \in \mathcal{H}$ using Algorithm 3. This runs in polynomial time.

If $\mathcal{H}$ is not connected, then we start by computing the connected components $\{\mathcal{C}_1, \ldots, \mathcal{C}_m\}$ of $\mathcal{H}$ and use what precedes to find for all $i \leq m$ a disjoint branches decomposition $\mathcal{T}_i$ of $\mathcal{C}_i$. We plug to the root of $\mathcal{T}_1$ the decompositions $\mathcal{T}_i$ for $i \geq 2$. Since each connected component have disjoint vertices, this results in a disjoint branches decomposition of $\mathcal{H}$. □

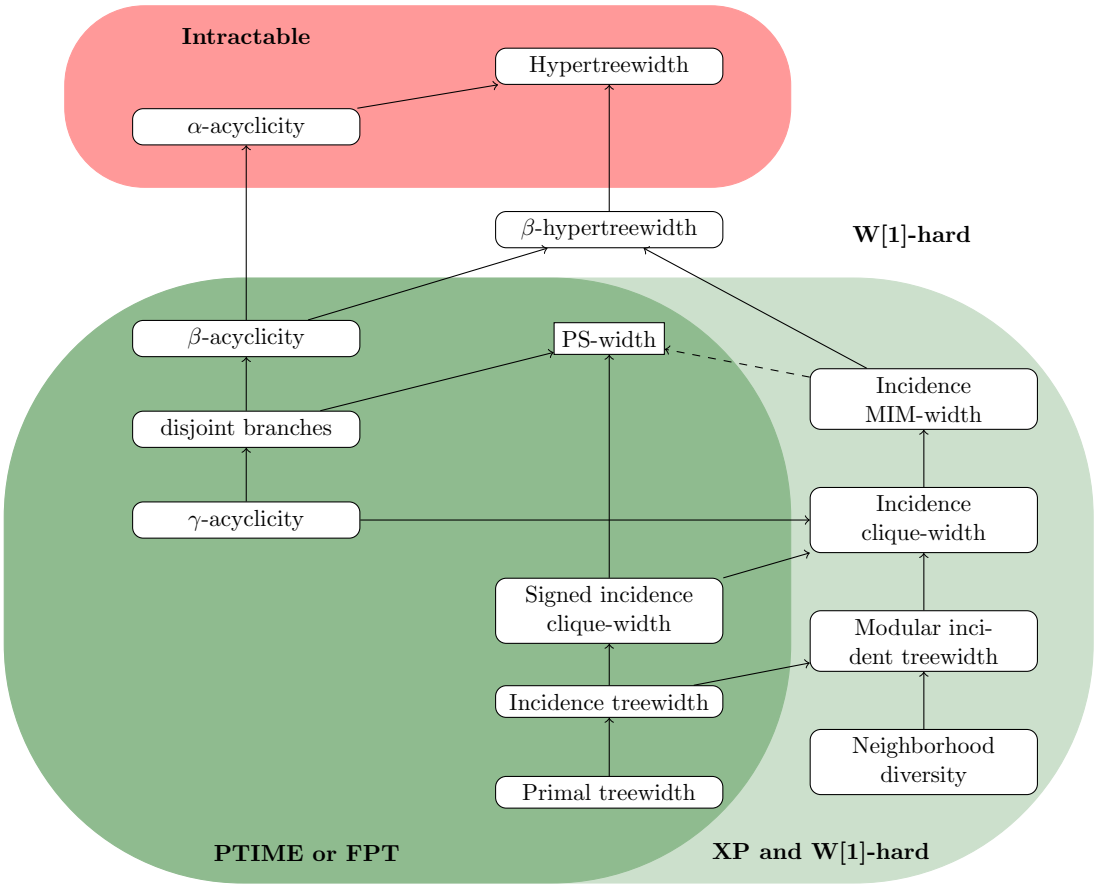

Figure 2.11: Structural restriction of #SAT

2.3 Tractability frontier

In this section, we review results concerning the complexity of structural restrictions of #SAT that are depicted in Figure 2.11. An arrow between two restrictions means that if a family of formula satisfies the first restriction, then it also satisfies the second. For example, a family of γ-acyclic formulas has also bounded incidence clique-width, since a γ-acyclic formulas has clique-width at most 3 [GP04].

A dashed arrow between two parameters means that if the first parameter is bounded for a CNF-formula F, then the second is bounded by a polynomial (whose degree may depend on k) in $\mathsf{size}(F)$.

All known efficient algorithms for structural restrictions of #SAT are performing a dynamic programming algorithm along a decomposition of the formula, as it is done for disjoint branches formulas in Theorem 2.5. It turns out that they can be summarized as one unique dynamic programming algorithm proposed by Sæther, Telle and Vatshelle in [STV14] on a very general kind of decomposition

of the formula. In the first section, we explain this result and show why it encompasses every tractable restriction of #SAT known so far, save the tractability of β-acyclic formulas which are intensively studied in Chapters 4 and 5. In a second section, we give hardness results, by showing both NP-completeness or W[1]-hardness of SAT for some parameters of CNF-formulas. Both sections aim to understand where the frontier of tractability for structural restriction of #SAT lie.

2.3.1 Parametrized polynomial time algorithms

In this section, we review the result of [STV14] concerning the tractability of #SAT on the so-called bounded PS-width instances and its consequences on the complexity of structural restrictions of this problem. We start by giving the definition of PS-width. We then show that the tractability of #SAT given a bounded PS-width decomposition of the formula explains almost every known tractability results on #SAT known so far.

The main concept of PS-width , introduced in [STV14], is that of *projections*. Given a CNF F and an assignment τ of $X \subseteq \mathsf{var}(F)$, we define the projection of F on τ, denoted by F/τ, as the set of clauses that are satisfied by τ, that is,

$$F/\tau = \{C \in F \mid \tau \models C\}.$$

Given $X \subseteq \mathsf{var}(F)$, we denote by $\mathsf{proj}(F, X) = \{F/\tau \mid \tau : X \to \{0,1\}\}$ the set of projections of F. Given a branch decomposition T of $\mathcal{G}_{\mathsf{inc}}(F)$ and a vertex v of T, we denote by F_v the set clauses of F such that the corresponding vertex of $\mathcal{G}_{\mathsf{inc}}(F)$ appears in the leaves of T_v and by X_v the set of variables of F that similarly appear in the leaves of T_v. We denote by $\overline{F_v} = F \setminus F_v$ and $\overline{X_v} = \mathsf{var}(F) \setminus X_v$. The PS-width of T is defined to be

$$\mathbf{psw}(T) = \max_{v \in V(T)} \max(|\mathsf{proj}(F_v, \overline{X_v})|, |\mathsf{proj}(\overline{F_v}, X_v)|).$$

The PS-width of a formula is the minimum PS-width of a branch decomposition of $\mathsf{var}(F) \cup F$. That is:

$$\mathbf{psw}(F) = \min\{\mathbf{psw}(T) \mid T \text{ is a branch decomposition of } F \cup \mathsf{var}(F)\}.$$

Example 2.14. We illustrate the concept of PS-width on an example. Let

$$\begin{aligned} C_0 &= x_1 \vee x_2 \vee x_3, \\ C_1 &= x_1 \vee x_2 \vee \neg x_3, \\ C_2 &= \neg x_1 \vee x_3 \vee x_4, \\ F &= C_0 \wedge C_1 \wedge C_2. \end{aligned}$$

Figure 2.12 gives a branch decomposition of $F \cup \mathsf{var}(F)$. We show how to compute the PS-width of v. We have $X_v = \{x_1, x_2\}$ and $F_v = \{C_2\}$. Thus we also have

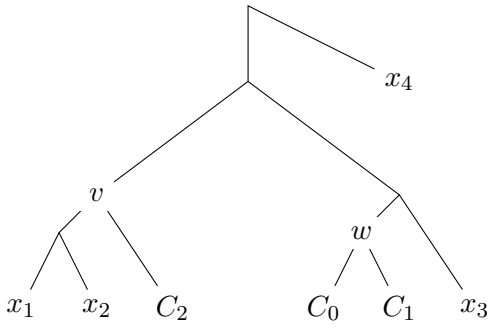

Figure 2.12: A branch decomposition for $F = C_0 \wedge C_1 \wedge C_2$ with $C_0 = x_1 \vee x_2 \vee x_3$, $C_1 = x_1 \vee x_2 \vee \neg x_3$ and $C_2 = \neg x_1 \vee x_3 \vee x_4$.

$\overline{X_v} = \{x_3, x_4\}$ and $\overline{F_v} = \{C_0, C_1\}$. We have $\mathsf{proj}(\overline{F_v}, X_v) = \mathsf{proj}(\{C_0, C_1\}, \{x_1, x_2\})$. If $\tau = \{x_1 \mapsto 0, x_2 \mapsto 0\}$, then $\overline{F_v}/\tau = \emptyset$ since neither C_0 nor C_1 are satisfied by τ. For any other $\tau : \{x_1, x_2\} \to \{0, 1\}$, we have $\overline{F_v}/\tau = \{C_0, C_1\}$. Thus $\mathsf{proj}(\overline{F_v}, X_v) = \{\emptyset, \{C_0, C_1\}\}$. Similarly, we have $\mathsf{proj}(F_v, \overline{X_v}) = \{\emptyset, \{C_2\}\}$. Thus, the width of v is 2.

We now show how to compute the PS-width of w. We have $X_w = \emptyset$ and $F_w = \{C_0, C_1\}$. Thus we also have $\overline{X_w} = \{x_1, x_2, x_3, x_4\}$ and $\overline{F_w} = \{C_2\}$. Since $X_w = \emptyset$, it holds that $\mathsf{proj}(\overline{F_w}, X_w) = \{\emptyset\}$. Now let $\tau : \overline{X_w} = \{x_1, x_2, x_3, x_4\} \to \{0, 1\}$. If $\tau(x_1) = 1$ or $\tau(x_2) = 1$ then $F_w/\tau = \{C_0, C_1\}$. Now, if $\tau(x_1) = \tau(x_2) = 0$, then either $\tau(x_3) = 1$ and in this case $F_w/\tau = \{C_0\}$ or $\tau(x_3) = 0$ and in this case $F_w/\tau = \{C_1\}$. Thus, we have $\mathsf{proj}(F_w, \overline{X_w}) = \{\{C_0\}, \{C_1\}, \{C_0, C_1\}\}$. Hence the width of w is 3.

It can be verified all nodes of the branch decomposition in Figure 2.12 have width at most 3. Thus, the PS-width of this branch decomposition is 3.

We now recall the main theorem of [STV14] concerning the complexity of #SAT parametrized by PS-width.

Theorem 2.15. *Given a formula F having n variables, m clauses of size at most s and a branch decomposition T of $F \cup \mathsf{var}(F)$ of PS-width k, one can find the number of satisfying assignments of F in time $k^3 \cdot O(s(n+m))$.*

We do not provide a proof of Theorem 2.15. We however prove later a more general result on the compilation of bounded PS-width CNF-formulas in d-DNNF which directly implies Theorem 2.15 (see Theorem 3.9 in Chapter 3).

Observe that the dependence in PS-width in the complexity of the algorithm of Theorem 2.15 is polynomial. This is a bit unusual in the setting of parametrized complexity where we are more used to dealing with exponential dependence in parameter that are polynomial in the size of the input. In this case, we have a polynomial dependence in the parameter but the parameter can be exponential in the size of the input.

The strength of Theorem 2.15 is that we can show that almost all known tractability results for structural restrictions of #SAT boils down to constructing a branch decomposition of small PS-width and use the algorithm of Theorem 2.15.

For instance, we will show that every disjoint branches formula with m clauses has PS-width at most m^2. Thus, if we are able to construct an optimal branch decomposition of such formulas, the algorithm of Theorem 2.15 will solve #SAT on disjoint branches formulas in time $O((m^2)^3 s(n+m)) = O(m^6 s(n+m))$ on a formula having n variables and m clauses of size at most s. Therefore, the tractability of #SAT on disjoint branches formulas boils down to the construction of a branch decomposition of small PS-width. Fortunately, the proof we give of the fact that disjoint branches formulas have small PS-width is constructive, that is, given a disjoint branches decomposition, we are able to transform it into a branch decomposition of PS-width at most m^2 in polynomial time (see Lemma 2.26). Since we know how to construct disjoint branches decompositions in polynomial time by Theorem 2.13, the tractability of #SAT on such formulas follows.

This scenario works for all structural restrictions in the green zone of Figure 2.11 but β-acyclicity. This singularity is discussed in details in Chapter 4. We now show the relations between PS-width and various graph parameters and explain how Theorem 2.15 implies tractability results.

Incidence tree width. As an example, we show how to use Theorem 2.15 to prove the result of [SS10, FMR08] on the tractability of bounded incidence tree width formulas.

Proposition 2.16. *A formula of incidence tree width k has PS-width at most 2^{k+1}.*

Proof. Let F be a formula and let T be a tree decomposition of its incidence graph of width k. Without lost of generality, we can assume T to be binary. This can be achieved by duplicating bags. We construct a branch decomposition T' of $F \cup \mathsf{var}(F)$ as illustrated on Figure 2.13. We start by adding a vertex r as the father of the root of T. Then for every element $x \in \mathsf{var}(F) \cup F$, let t be the closest vertex to the root of T such that x appears in the label of t. We hang a leaf labeled with x on the edge between t and its father. We then forget the labels of T and remove the leaves that have no label. The resulting tree T' is binary and for every element of $F \cup \mathsf{var}(F)$, there exists exactly one leaf labeled with this element. This is thus a branch decomposition of $F \cup \mathsf{var}(F)$. We claim that the PS-width of T' is at most 2^{k+1}.

Indeed, let t' be a vertex of T' then t' was introduced between a vertex t of T and its father u in T (u may be the new root r). By construction, the leaves of $T'_{t'}$ are labeled with variables and vertices that appears in the labels of T_t only. Moreover, we have introduced each element x of $F \cup \mathsf{var}(F)$ as leaf such that x does not appear in the labels of $T \setminus T_t$. Thus, the leaves of $T' \setminus T'_{t'}$ are labeled with variables and vertices that appears in the labels of $T \setminus T_t$ or in the label of t.

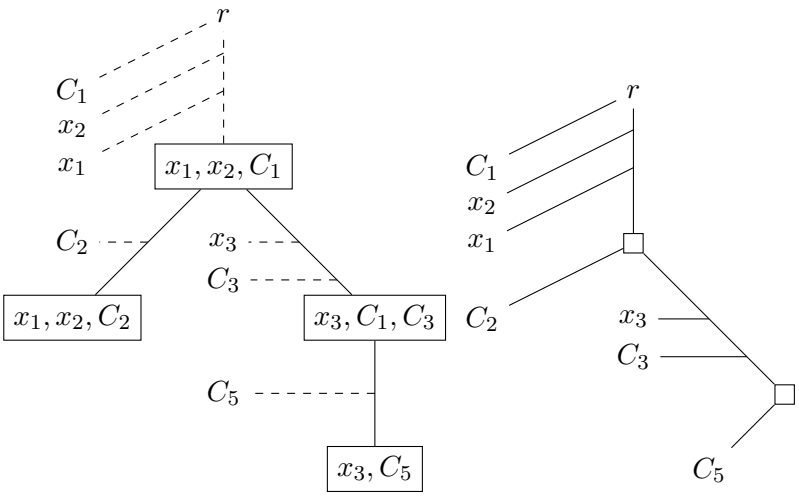

Figure 2.13: Transforming a tree decomposition into a small PS-width branch decomposition. We hang the leaves (dashed edges) and forget the bag labels and non-labeled leaves.

Now assume there is a variable x in $L(T'_{t'})$ that is in a clause C of $L(T') \setminus L(T'_{t'})$. We claim that C appears on the label of t in T. Assume the contrary. By the previous observation, C only appears in $T \setminus T_t$ and x only appears in the label of T_t. In particular, x and C never appear in the same bag. But since $\{x, C\}$ is an edge of $\mathcal{G}_{\mathsf{inc}}(F)$, such a bag has to exist. Contradiction. Similarly, we show that if a clause C in $L(T'_{t'})$ contains a variable x of $L(T') \setminus L(T'_{t'})$ then x has to appear in the label of t.

Thus, let F' be the clauses in $L(T'_{t'})$ and $\overline{X'}$ be the variables of $L(T') \setminus L(T'_{t'})$. From what precedes, the variables of $\overline{X'}$ that are in clauses of F' must be in the bag labeling t. Thus, there is at most $k+1$ such variables. That is, $\mathsf{proj}(F', \overline{\{X'\}})$ has at most 2^{k+1} elements. Similarly, $\mathsf{proj}(F \setminus F', \mathsf{var}(F) \setminus \overline{\{X'\}})$ has at most 2^{k+1} elements, thus the PS-width of T' is at most 2^{k+1}. □

Combining Proposition 2.16 and Theorem 2.15, we obtain the following result.

Corollary 2.17. #SAT *can be solved in time* $2^{O(k)}O(s(m+n))$ *on a formula* F *of incidence tree width* k *having* n *variables and* m *clauses of size at most* s.

Proof. Compute a tree decomposition of tree width k of $\mathcal{G}_{\mathsf{inc}}(F)$ using Theorem 1.25, then transform it into a branch decomposition of PS-width 2^{k+1} with Proposition 2.16 and use the algorithm of Theorem 2.15. □

MIM-width. We now show that PS-width and MIM-width are nicely related. This relation is already known from [STV14] but we give an alternative proof by proving a min-max theorem on the size of maximum induced matchings of bipartite

graph that is interesting in itself and that we will use later in Section 2.3.3. This relation is mainly used to connect PS-width with other graph measures.

Given a graph $G = (V, E)$, we say that G is k-covered if for every $W \subseteq V$, there exists $W' \subseteq W$ such that $|W'| \leq k$ and $\mathcal{N}(W) = \mathcal{N}(W')$. Observe in particular that G is $|V|$-covered. We define the *cover-value* of G, denoted $cv(G)$, to be the minimum k such that G is k-covered. The cover-value of a graph is equal to the size of its biggest induced matching:

Theorem 2.18. *Let $G = (X, Y, E)$ be a bipartite graph and let k be the size of its biggest induced matching. It holds:*

$$cv(G) = k.$$

Proof. We start by showing $cv(G) \leq k$. Let $W \subseteq X$ and let $W' \subseteq W$ such that $\mathcal{N}(W') = \mathcal{N}(W)$ and W' is of minimal size. Since W' if minimal, for all $x \in W'$, $\mathcal{N}(W' \setminus \{x\}) \subsetneq \mathcal{N}(W)$. Thus, there exists $y_x \in \mathcal{N}(W)$ such that for all $x' \in W'$, if $x' \neq x$, then y_x is not a neighbor of x'. We claim that $M = \{(x, y_x) \mid x \in W\}$ is an induced matching of G. Indeed, for all $x, x' \in W'$, $(x, x') \notin E$ and $(y_x, y_{x'}) \notin E$ since G is bipartite. Moreover, by construction, $(x', y_x) \notin E$ and $(x, y_{x'}) \notin E$. Thus M is an induced matching of G, that is, $|W'| = |M| \leq k$. That is $cv(G) \leq k$.

We now show that $k \leq cv(G)$. Let M be an induced matching of G of size k and let $W = V(M) \cap X$. We claim that for all $W' \subsetneq W$, $\mathcal{N}(W') \neq \mathcal{N}(W)$. Indeed, let $W' \subsetneq W$ and let $x \in W \setminus W'$. Let $y \in V(M)$ be the only neighbor of x in M, that is, $(x, y) \in M$. We have $y \in \mathcal{N}(W)$. Since M is an induced matching, we have for all $z \in W \setminus \{x\}$, $(z, y) \notin M$. That is, $y \notin \mathcal{N}(W')$. Thus, the only set $W' \subseteq W$ such that $\mathcal{N}(W') = \mathcal{N}(W)$ is W itself. That is $cv(G) \geq |W| = |M| = k$. □

The following theorem follows easily from Theorem 2.18:

Theorem 2.19. *Let F be a* CNF*-formula, $G = \mathcal{G}_{\mathsf{inc}}(F)$ and let T be a branch decomposition of G of* MIM*-width k. Then* PS*-width$(T) \leq |F|^k$. In particular,*

$$\mathbf{psw}(F) \leq |F|^{\mathbf{mimw}(G)}.$$

Proof. Let v be a vertex of T. We want to show that $|\mathsf{proj}(F_v, \overline{X_v})| \leq |F|^k$ and $|\mathsf{proj}(\overline{F_v}, X_v)| \leq |F|^k$. We show the first inequality, the other case being completely symmetric to the first one.

By definition of MIM-width, the biggest inducted matching of $G[\overline{X_v}, F_v]$ is at most k and by Theorem 2.19, $cv(G[\overline{X_v}, F_v]) \leq k$. That is for every $W \subseteq F_v$, there exists $W' \subseteq W$ such that $|W'| \leq k$ and $\mathsf{var}(W') = \mathsf{var}(W)$.

Let $\mathcal{C}, \mathcal{C}' \in \mathsf{proj}(F_v, \overline{X_v})$ with $\mathcal{C} \neq \mathcal{C}'$. Let τ, τ' be such that $\mathcal{C} = F_v/\tau$ and $\mathcal{C}' = F_v/\tau'$. Let $\mathcal{D} = F_v \setminus \mathcal{C}$ and $\mathcal{D}' = F_v \setminus \mathcal{C}'$. Observe that $\mathcal{D} \neq \mathcal{D}'$, and assume without lost of generality that $\mathcal{D} \not\subseteq \mathcal{D}'$. By definition, $\mathcal{D}$ is the set of clauses of F_v that are not satisfied by τ. Let $W \subseteq \mathcal{D}$ such that $|W| \leq k$ and $\mathsf{var}(W) = \mathsf{var}(\mathcal{D})$. Let $W' \subseteq \mathcal{D}$ such that $|W'| \leq k$ and $\mathsf{var}(W') = \mathsf{var}(\mathcal{D})$. $\tau|_{\mathsf{var}(\mathcal{D})}$ is the unique assignment that do not satisfy any clause of $\mathcal{D}$ and hence of W. Since $\mathcal{D} \not\subseteq \mathcal{D}'$,

there exists a clause of $\mathcal{D}$ that is satisfied by τ', that is, $\tau'_{\mathsf{var}(\mathcal{D})} \neq \tau_{\mathsf{var}(\mathcal{D})}$. Thus there also exists a clause of W that is satisfied by τ', that is $W \neq W'$.

We can then map an assignment τ to a subset $W(\tau)$ of F_v of size at least k such that if $F/\tau \neq F/\tau'$, then $W(\tau) \neq W(\tau')$. Since there is at most $|F_v|^k \leq |F|^k$ such subset, we have $|\mathsf{proj}(F_v, \overline{X_v})| \leq |F|^k$ which is what we wanted. □

This settles the relation between MIM-width and PS-width. Unfortunately, to the best of our knowledge, computing decomposition of small MIM-width is not known to be doable in FPT nor XP time. This relation is thus not sufficient to explain the tractability of #SAT on bounded clique-width.

Incidence clique-width. We explain how one can use the algorithm of Theorem 2.15 to actually solve #SAT on bounded incidence clique-width. The main remaining problem is to compute a branch decomposition of small incidence clique-width in parametrized time. This can be achieved by computing a decomposition of small *rank-width* . Rank-width is defined on branch decomposition from the symmetric function rank which associate to a subset X of vertices, the rank of the incidence matrix of $G[X, \overline{X}]$. Rank-width decomposition can be computed in FPT time and relates nicely to clique-width:

Theorem 2.20 ([HO08])**.** *Let $k \in \mathbb{N}$. For a graph $G = (V, E)$, we can output a branch decomposition of rank-width at most k or confirm that the rank-width of G is larger than k in time $f(k) \cdot O(n^3)$ for a computable function f.*

Theorem 2.21 ([OS06])**.** *For every graph $G = (V, E)$ it holds that*

$$\mathbf{rw}(G) \leq \mathbf{cw}(G) \leq 2^{\mathbf{rw}(G)+1} - 1.$$

Moreover, it can be observed that the rank-width of a branch decomposition is an upper bound of its MIM-width:

Proposition 2.22. *Given a graph $G = (V, E)$ and a branch decomposition T of G it holds that* $\mathbf{mimw}(T) \leq \mathbf{rw}(T)$.

Proof. Let v be a vertex of T and let $M = \{(x_1, y_1), \ldots, (x_k, y_k)\}$ be an induced matching of $G[V(T_v), V \setminus V(T_v)]$. We show that $k \leq \mathbf{rw}(T)$. Indeed, since M is an induced matching, for all $i \neq j$, x_i is not a neighbor of y_j. Thus, in the adjacency matrix, it yields k independent column vectors. That is the rank of the matrix is at least k, hence $k \leq \mathbf{rw}(T)$. □

Combining these results yields the following:

Corollary 2.23. #SAT *on a formula F of incidence clique-width k having n variables and m clauses of size at most s can be solved in time $f(k) \cdot O((n + m)^3) + m^{3k} \cdot O(s(m + n))$ for a computable function f.*

Proof. First compute a branch decomposition of $\mathcal{G}_{\mathsf{inc}}(F)$ of rank-width at least k using Theorem 2.20 and the fact that $\mathbf{rw}(G) \leq \mathbf{cw}(G) \leq k$ by Theorem 2.21. This can be computed in time $f(k) \cdot O((n+m)^3)$ for a computable function f by Theorem 2.21 again. By Proposition 2.22, this decomposition has MIM-width at most k thus PS-width at most m^k by Theorem 2.19. Using the algorithm of Theorem 2.15 solves #SAT for F on decomposition T in time $m^{3k}O(s(m+n))$. □

Signed incidence clique width. We explain now the result of Fischer, Makowski and Ravve [FMR08] concerning the parametrized complexity of #SAT on bounded signed clique-width. Observe that Figure 2.11 exhibits a relation between signed incidence clique width and incidence clique width. However, this is not sufficient to use this relation to explain the FPT algorithm for bounded signed incidence clique width of [FMR08] since we only have an XP algorithm for bounded incidence clique width formulas. In this paragraph, we show direct connection between PS-width and signed incidence clique width.

The set of CNF-formulas of signed clique-width at most k is defined as the set of formulas whose signed incidence graph $\mathcal{G}^{+}_{\mathsf{inc}}(F)$ can be obtained by the following operations over graphs whose vertices are colored by $\{1, \ldots, k\}$, starting from singleton graphs.

1. Disjoint union.

2. Recoloring: For a vertex colored signed bipartite graph G, we defined $\rho_{i,j}(G)$ to be the graph that results from recoloring with j all vertices that were previously colored with i.

3. Vertices to clauses edge creation: For a vertex-colored signed bipartite graph G, we define $\eta^{+}_{i,j}(G)$ to be the graph that results from connecting all clause-vertices colored i to all variable-vertices colored j, with edges going from the variables side to the clauses side.

4. Clauses to vertices edge creation: Similarly to above, we define $\eta^{-}_{i,j}(G)$ to be the graph resulting from connecting all clause-vertices colored with i to all variable-vertices colored with j, with edges going from clauses to variables.

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

A *parse tree* for the signed clique-width of a formula F is the rooted tree whose leaves hold singleton graphs, whose internal vertices are colored 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 $\mathcal{G}^{+}_{\mathsf{inc}}(F)$ (with any vertex coloring).

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

Proof. Let v be a vertex of T. Let τ be the sub-parse tree which is rooted by the union that led to the introduction of v.

We first show that $|\mathsf{proj}(\overline{F_v}, X_v)| \leq 2^{2k}$. Observe that when two variables $x, x' \in X_v$ have the same color in τ, then they must always appear together in every clause in $\overline{F_v}$ and their sign must be the same. Call X_i the set of variables in X_v that are colored by i. Then for every assignment of X_v 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 projection set, we can delete all but two variables from X_i from $\overline{F_v}$. It follows that $\overline{F_v}$ has the same projection set as a formula with $2k$ variables. But there are only 2^{2k} assignments to $2k$ variables, so it follows that $|\mathsf{proj}(\overline{F_v}, X_v)| \leq 2^{2k}$.

We now show that $|PS(F_v, \overline{X_v})| \leq 2^{2k}$. 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 $\overline{X_v}$ and moreover $C|_{\overline{X_v}} = C'|_{\overline{X_v}}$. Thus F_v only has k different clauses containing variables of $\overline{X_v}$, so trivially $|\mathsf{proj}(F_v, \overline{X_v})| \leq 2^{2k}$. □

Corollary 2.25 ([FMR08])**.** #SAT *on a formula of signed incidence clique-width k having n variables and m clauses of size at most s can be solved in time $2^{O(k)}O(s(n+m))$ 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.

Disjoint branches. We finally explain how results from Section 2.2.2 may be explained by Theorem 2.15. To do this, we turn a disjoint branches decomposition of the hypergraph of a formula into a branch decomposition of MIM-width at most 2.

Lemma 2.26. *Given a hypergraph $\mathcal{H}$ and a disjoint branches decomposition of $\mathcal{H}$, we can in polynomial time compute a branch decomposition of $\mathcal{G}_{\mathsf{inc}}(\mathcal{H})$ of* MIM*-width at most* 2*.*

Proof. Let $(\mathcal{T}, \lambda)$ be a disjoint branches decomposition of $\mathcal{H}$. 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 binary: 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 $(X, \overline{X})$. 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 $\mathcal{G}_{\mathsf{inc}}(\mathcal{H})[X, \overline{X}]$. Furthermore, e is the only edge in $E \cap X$, and it follows that the MIM-width of the cut $(X, \overline{X})$ 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 $(X, \overline{X})$ 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 $e' \in \overline{X} \cap E$, then $u \in \lambda(x) \in \overline{X} \cap E$. Consequently, only one vertex $u \in X \cap V$ can be an end-vertex of an induced matching in $\mathcal{G}_{\mathsf{inc}}(\mathcal{H})[X, \overline{X}]$. Furthermore, no vertex u in $\overline{X} \cap V$ 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 $(X, \overline{X})$ is at most 1. □

Corollary 2.27 ([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 2.13. Then apply the construction of Lemma 2.26 to get a branch decomposition of MIM-width at most 2. Now combining Theorem 2.19 and Theorem 2.15 yields the results. □

2.3.2 Hardness results

In this section, we review hardness result from the literature, explaining the red and light green parts of Figure 2.11.

Intractability results. We first investigate hardness results, which show that some restriction of #SAT are already as hard as the general case. We show that SAT on α-acyclic hypergraphs is NP-complete. Since α-acyclic hypergraphs are the hypergraphs of hypertree width 1, it shows that #SAT is intractable for bounded hypertree width instances, even when this hypertree width is 1.

Theorem 2.28. *It is* NP*-hard to decide if an* α*-acyclic* CNF*-formula is satisfiable.*

Proof. We reduce the problem to SAT. Let F be a CNF-formula. Let y be a fresh variable, that is $y \notin \mathsf{var}(F)$. We let $F' = F \cup \{\mathsf{var}(F) \cup \{y\}\}$. We claim that F' is α-acyclic and F is satisfiable if and only if F' is satisfiable.

The fact that F' is α-acyclic directly follows from Observation 1 since there is an edge in $\mathcal{H}(F')$ that covers every variables of F'. Now let τ be a satisfying assignment of F'. We clearly have $\tau|_{\mathsf{var}(F)} \models F$ since F is a subformula of F'. If τ is a satisfying assignment of F then $\tau' := \tau \cup \{y \mapsto 1\}$ is a satisfying assignment of F' since $\tau' \models \{\mathsf{var}(F) \cup \{y\}\}$ and $\tau' \models F$ by definition. □

This yields the following corollary:

Corollary 2.29. *Let* $k \in \mathbb{N}$*. It is* NP*-hard to decide if a* CNF*-formula* F *such that* $\mathcal{H}(F)$ *is of hypertree width at most* k *is satisfiable.*

W[1]-hardness results. We now investigate hardness results, which show that some restrictions of #SAT for which XP algorithms exist are very unlikely to have FPT algorithms. This concerns parameters in the light green part of Figure 2.11. Observe that if we are given two parameters p_1 and p_2 of CNF-formulas such that for all F, if $p_1(F)$ is bounded then $p_2(F)$ is bounded too, then it is enough to show the $W[1]$-hardness of SAT for parameter p_1. Thus, showing W[1]-hardness of SAT for neighborhood diversity is enough to show W[1]-hardness of modular incident tree width, clique-width and MIM-width.

Neighborhood diversity was introduced in [Lam10] and the W[1]-hardness of SAT parametrized by neighborhood diversity was shown in [DKL$^+$15] where its relation with modular incident tree width was observed too. A graph $G = (V, E)$ has *neighborhood diversity* k if there exists a k-partition $V_1, \ldots, V_k$ of V such that for every $v \in V$ and $i \in [k]$, either v is adjacent to every vertex of V_i or v is adjacent to no vertex of V_i. Observe that every V_i is either a clique or an independent set of G. The *neighborhood diversity* of a graph G, denoted by $\mathbf{nd}(G)$, is the smallest k such that G has neighborhood diversity k.

The neighborhood diversity of a CNF-formula is the neighborhood diversity $\mathbf{nd}(\mathcal{G}_{\mathsf{inc}}(F))$ of the incidence graph of F. A formula F of neighborhood diversity k

can be seen as a formula for which there exist at most k different kind of variables and at most k kind of clauses where two clauses of the same kind have the same variables and variables of the same kind are in the same clauses. The relation between modular incident tree width and neighborhood diversity becomes clearer:

Theorem 2.30. *For every* CNF*-formula F, the modular incident tree width of F is at most its neighborhood diversity.*

Proof. Let F be a CNF-formula such that $k = \mathbf{nd}(F)$. There are at most k kinds of variables. Variables of the same kind are in the same clauses, that is, they are modules in the incidence graph. Similarly there are at most k kinds of clauses having the same variables, that is, they are modules. Thus, after contracting the modules of $\mathcal{G}_{\mathsf{inc}}(F)$, we have a bipartite graph having at most k vertices in each side, that is, having tree width at most k. Thus the modular incident tree width of F is at most k. □

Even if neighborhood diversity is a very restrictive parameter of CNF-formulas, SAT is already W[1]-hard for this measure:

Theorem 2.31 ([DKL$^+$15])**.** SAT *parametrized by* **nd** *is* W[1]*-hard.*

The W[1]-hardness of numerous parameters follows from Theorem 2.31:

Corollary 2.32. SAT *parametrized by neighborhood diversity, modular incident tree width, clique-width,* MIM*-width or β-hypertree width is* W[1]*-hard.*

Proof. All of these results follows from the fact that each of these measures are smaller that neighborhood diversity. For modular tree width, it follows from Theorem 2.30. For clique-width, it follows from the fact that clique-width is stable by contracting modules and from the relation between clique-width and tree width (see [PSS13]). The relation between MIM-width and clique-width follows from Theorem 2.22 and Theorem 2.21. Finally, the proof that β-hypertree width is smaller than clique-width can be found in [GP04]. We actually prove a more general result in the next section (Theorem 2.33) by directly proving a relation between MIM-width and β-hypertree width. □

2.3.3 Unknown complexity hardness

We conclude this chapter with a few words on the status of β-hypertree width. Few is known on the complexity of #SAT, and even SAT, on classes of bounded β-hypertree width. As mentioned in Corollary 2.32, #SAT parametrized by β-hypertree width is W[1]-hard so we cannot hope for an FPT algorithm (it follows from the fact that β-hypertree width is more general than clique width [GP04]). However, it is an intriguing open question to know if this problem is at least in XP.

We show in Chapter 4 and Chapter 5 that for the case of β-acyclicity, that is, β-hypertree width 1, #SAT is tractable. The algorithmic techniques that we use

here differ widely from the classical dynamic programming approach on branch decompositions and we have evidences that such approaches do not work in general on β-acyclic instances. In Section 5.3.2, we give some directions that may lead to an XP algorithm for #SAT on bounded β-hypertree width instances by generalizing the algorithmic techniques that were used for β-acyclic instances. However, our lack of understanding of the structure of hypergraphs of β-hypertree width k makes the use of such techniques difficult.

In this section, we prove a new result showing that β-hypertree width is more general than MIM-width. This is interesting since it shows that if we can prove that #SAT parametrized by β-hypertree width is in XP, then it would give an unified explanation of the tractability of #SAT on β-acyclic instances and on bounded incidence MIM-width instances:

Theorem 2.33. *For every hypergraph $\mathcal{H}$, it holds:*

$$\beta\text{-}\mathbf{htw}(\mathcal{H}) \leq 6 \cdot \mathbf{mimw}(\mathcal{G}_{\mathsf{inc}}(\mathcal{H})) + 1.$$

Theorem 2.33 follows from Lemma 2.34 and Lemma 2.35. The former states that the MIM-width of a hypergraph is greater than the MIM-width of its subhypergraphs. The later states that MIM-width is greater than generalized hypertree width. Applying Theorem 1.44 that states that $\mathbf{htw}(\mathcal{H}) \leq 3\mathbf{ghtw}(\mathcal{H}) + 1$ yields the theorem.

Lemma 2.34. *Let $G = (X, Y, E)$ be a bipartite graph of* MIM*-width k and let $Y' \subseteq Y$. The* MIM*-width of $G' = G[X, Y']$ is at most k.*

Proof. Let T be a branch decomposition of G of MIM-width k. We transform T into a branch decomposition T' of G' by removing leaves labeled with vertices in $Y \setminus Y'$. We claim that the MIM-width of T' is at most k. For a vertex t of T, we denote by $X_t = X \cap L(T_t)$, $\overline{X_t} = (L(T) \setminus L(T_t)) \cap X$ and $Y_t = Y \cap L(T_t)$, $\overline{Y_t} = (L(T) \setminus L(T_t)) \cap Y$.

Let t' be a vertex of T' and let t be its corresponding vertex in T. It is clear that $X_t = X_{t'}$ and $\overline{X_t} = \overline{X_{t'}}$. Moreover, $Y_{t'} = Y_t \cap Y'$ and $\overline{Y_{t'}} = \overline{Y_t} \cap Y'$. Now let M be an induced matching of $G[X_{t'}, \overline{Y_{t'}}]$. M is also an induced matching of $G[X_t, Y_t]$ since $G[X_{t'}, \overline{Y_{t'}}]$ is a graph induced from $G[X_t, Y_t]$. Thus M is of size at most k. The case where M is a matching of $G[\overline{X_{t'}}, Y_{t'}]$ is symmetric. It follows that the MIM-width of T' is at most k, so the MIM-width of G' is also at most k. □

Lemma 2.34 states in particular that if $\mathcal{H}$ is a hypergraph and $\mathcal{H}' \subseteq \mathcal{H}$, then the MIM-width of $\mathcal{G}_{\mathsf{inc}}(\mathcal{H}')$ is smaller than the MIM-width of $\mathcal{G}_{\mathsf{inc}}(\mathcal{H})$. Thus, it remains to prove that generalized hypertree width is bounded by MIM-width:

Lemma 2.35. *For every hypergraph $\mathcal{H}$, it holds:*

$$\mathbf{ghtw}(\mathcal{H}) \leq 2\mathbf{mimw}(\mathcal{G}_{\mathsf{inc}}(\mathcal{H})).$$

Proof. Let $G = \mathcal{G}_{\mathsf{inc}}(\mathcal{H})$, $k = \mathbf{mimw}(G)$ and let (T, δ) be a branch decomposition of G of MIM-width k. For a vertex t of T, we denote by $V_t = V(\mathcal{H}) \cap L(T_t)$, $\overline{V_t} = V(\mathcal{H}) \setminus V_t$, $E_t = \mathcal{H} \cap L(T_t)$, $\overline{E_t} = \mathcal{H} \setminus E_t$. We construct a hypertree decomposition $(\mathcal{T}, \lambda(t))$ of $\mathcal{H}$ from T as follows: the underlying tree of $\mathcal{T}$ is T. Moreover for each vertex t, let

$$W_t = (\mathsf{var}(E_t) \cap \overline{V_t}) \cup (\mathsf{var}(\overline{E_t}) \cap V_t).$$

For every vertex t of T, we define $\lambda(t)$ as follows:

- if t is a leaf then $\lambda(t) = W_t$
- otherwise let t_1, t_2 be the children of t. We define $\lambda(t) = W_t \cup (W_{t_1} \cap W_{t_2})$.

We claim that $\mathcal{T}$ is a branch decomposition of $\mathcal{H}$ and that it is of generalized hypertree width at most $2k$. First, we have to show that for every $e \in \mathcal{H}$, there exists t such that $e \subseteq \lambda(t)$. Let $e \in \mathcal{H}$ and let t_e be the leaf of T such that $\delta(t_e) = e$. Then $\lambda(t_e) \supseteq e$ since $E_{t_e} = \{e\}$ and $\overline{V_{t_e}} = V(\mathcal{H})$ and thus $\mathsf{var}(E_{t_e}) \cap \overline{V_{t_e}} = e$.

Moreover, let $v \in V(\mathcal{H})$. We claim that the set of vertices t of $\mathcal{T}$ such that $v \in \lambda(t)$ is connected in $\mathcal{T}$. Let t_1 and t_2 be two vertices of $\mathcal{T}$ such that $t \in \lambda(t_1) \cap \lambda(t_2)$.

First assume that there exists t_0, t_1, t_2 such that $x \in \lambda(t_0)$, t_1 is a child of t_0 and $x \notin \lambda(t_1)$ and t_2 is a descendant of t_1 with $x \in \lambda(t_2)$. Since $x \notin W_{t_1}$ and $x \in \lambda(t_0)$, we have $x \in W_{t_0}$. Now assume that x is in $L(T_{t_1})$. Then x is also in $L(T_{t_0})$ and since $x \in W_{t_0}$, there exists $e \in \overline{E_{t_0}}$ such that $x \in e$. But then $e \in \overline{E_{t_1}}$ and then $x \in W_{t_1}$, which is a contradiction. Now if x is not in $L(T_{t_1})$, then it is not in $L(T_{t_2})$ too. We can assume w.l.o.g that no child u of t_2 verifies $x \in \lambda(u)$ by selecting t_2 to be the deepest vertex of T_{t_0} having this property. Now since $x \in \lambda(t_2)$, it means that $x \in W_{t_2}$. That is, there exists an edge $e \in L(T_{t_2})$ such that $x \in e$. But $e \in L(T_{t_1})$ too and since $x \in \overline{V_{t_1}}$, it would imply $x \in W_{t_1}$, contradiction.

We thus have shown that if t_0 and t_2 are on the same branch of T and if $v \in \lambda(t_0) \cap \lambda(t_2)$, then for every vertex t_1 between t_0 and t_2, it holds $v \in \lambda(t_1)$. Now assume that t_0 and t_2 are not on the same branch of T and that $v \in \lambda(t_0) \cap \lambda(t_2)$. Let t_1 be the father of t_0. We show that $v \in \lambda(t_1)$.

Assume first that $v \notin V_{t_1}$. In this case, since $v \in \lambda(t_0)$ we have that either $v \in W_{t_0}$ or $v \in W_u$ for u a child of t_0. In any case, it means that there exists an edge $e \in L(T_{t_0})$ such that $v \in e$. Then $v \in L(T_{t_1})$ too and then since $v \in \overline{V_{t_1}}$, we have $v \in W_{t_1}$, that is, $v \in \lambda(t_1)$.

Now assume that $v \in V_{t_1}$. There are two cases. First assume that t_2 is not a descendant of t_1. In this case, there exists $e \in L(T_{t_2})$ such that $v \in e$. But then, if t_2 is not a descendant of t_1, we also have $e \in \overline{E_{t_1}}$ and then $v \in \lambda(t_1)$. Now assume t_2 is a descendant of t_1. It is not a descendant of t_0 since we have assumed t_0 and t_2 to be on two different branches. Let u be the other child of t_1. We claim that $x \in W_{t_0} \cap W_u$. Indeed, since $v \in V_{t_1}$, we either have $x \in V_{t_0}$ or $x \in V_u$. If $x \in V_u$ then there exists $e \in E_{t_0}$ such that $x \in e$ since $x \in \lambda(t_0)$. And then $x \in W_{t_0} \cap W_u$.

Similarly, if $x \in V_{t_0}$ then $e \in E_{t_2} \subseteq E_u$ such that $x \in e$ since $x \in \lambda(t_2)$ and then again $x \in W_{t_0} \cap W_u$.

We thus have shown that if $v \in \lambda(t_0) \cap \lambda(t_2)$ and if t_0 and t_2 are not on the same branch of T, then $v \in \lambda(t_1)$ where t_1 is the father of t_0. But then we can apply this to t_1 and t_2. By induction, we get that $v \in \lambda(t)$ for t the least common ancestor of t_0 and t_2. Thus we have shown that $\mathcal{T}$ is a tree decomposition.

It remains to show that $\mathcal{T}$ is of hypertree width at most $2k$ that is, for every vertex t, there exists $S \subseteq \mathcal{H}$ such that $\lambda(t) \subseteq \bigcup_{f \in S} f$ and $|S| \leq 2k$. We show that for every t, there exists $S \subseteq \mathcal{H}$ such that $W_t \subseteq \bigcup_{f \in S} f$ and $|S| \leq k$. The desired result follows since $\lambda(t) \subseteq W_t \cup W_u$ where u is a child of t.

Let t be a vertex of T. By definition of MIM-width, we know that any induced matching in $G' = G[L(T_t), \overline{L(T_t)}]$ is of size at most k. Let E be the edges of $\mathcal{H}$ that are neighbors of W_t in G'. Every vertex of W_t has at most one neighbor in G' by definition of W_t. Thus the neighborhood in G' of E contains W_t. By Theorem 2.18, the cover-value of G' is also k, that is, there exists $E' \subseteq E$ with $|E'| \leq k$ and such that the neighborhood of E' in G' is the same as the neighborhood of E in G', that is, E' covers W_t and is of size at most k. □

Chapter 3

Parametrized compilation of CNF-formulas

Most of the practical tools existing for #SAT are based on a common algorithm called exhaustive DPLL. In [HD05], Darwiche and Huang observed that such tools were implicitly constructing a decision DNNF equivalent to the input formula, whose size was roughly the runtime of the algorithm. This remark of Huang and Darwiche does however not apply to the algorithms presented in Chapter 2 for structural restrictions of CNF formulas. Indeed, we have seen in Chapter 2 that all these algorithms perform a dynamic programming algorithm on a well-chosen decomposition of the CNF-formula, an approach that is quite different from exhaustive DPLL.

In this chapter, we show that the observation of Darwiche and Huange can be extended to the structure-based algorithm for #SAT. More precisely, we show that the trace of the algorithm of [STV14] for #SAT can be seen as a compilation algorithm from a CNF-formula into a structured deterministic DNNF. As in Chapter 2, we can show that this general compilation algorithm yields compilation algorithms for most of the tractable structural restrictions of #SAT presented in Figure 2.11. This result allows us to lift the results on structural restrictions of CNF-formulas for #SAT to other important problems on CNF-formulas such as enumeration or weighted model counting. We also show how we can use this compilation algorithm to solve MaxSAT, a result that was already presented in [STV14] for bounded PS-width as a separated algorithm.

Structural restrictions of CNF have already been used to compile formulas efficiently. Darwiche[Dar01b] have shown that formulas of bounded primal tree width can be compiled efficiently into d-DNNF, a result later generalized with Pipatsrisawat to the compilation into structured DNNF [PD08] and to the compilation into structured d-DNNF for bounded dual tree width instances [PD10a]. In [RP13], Razgon and Petke have shown that boolean circuits whose underlying graph is of bounded clique-width can be efficiently transformed into DNNF. This may be used as a compilation algorithm for CNF formulas of bounded signed clique-width

into DNNF. All these results are explained and generalized in this chapter since we have shown in Chapter 2 that PS-width generalizes all these parameters. Other parameters have been introduced by Umut Oztok and Adnan Darwiche for CNF-formulas such as CV-width [OD14a] and Decision-width [OD14b] for which they can have efficient compilation algorithms into d-DNNF and dec-DNNF respectively. We do not know how PS-width relates with such widths however.

This chapter is organized as follows. We start by presenting the compilation algorithm that constructs a small d-DNNF from a formula F and a branch decomposition of F of small PS-width. We then show how to use this algorithm for known structural classes of formulas and how we use it to solve optimization problems such as MaxSAT.

3.1 Compilation of bounded PS-width formulas

We start by presenting the compilation algorithm for bounded PS-width CNF-formulas. The algorithm is a strengthening form of the counting algorithm of [STV14]. The results of this section were published in [BCMS15].

3.1.1 Shapes

To compile bounded PS-width formulas into d-DNNF, we perform a dynamic programming algorithm along a branch decomposition of $F \cup \mathsf{var}(F)$ of small width. In this section, we present the records that are dynamically propagated by the algorithm. We rely on the notion of *shape* that was introduced in [SS13].

Let F be a formula, let T be a branch decomposition of $F \cup \mathsf{var}(F)$, and let v be a node of T. A *shape* (for v, with respect to T) is a pair $\vec{S} = (S, S')$ of subsets of F such that $S \in \mathsf{proj}(\overline{F_v}, X_v)$ and $S' \in \mathsf{proj}(F_v, \overline{X_v})$. We say that an assignment $\tau : X_v \to \{0,1\}$ has shape $\vec{S}$ if

1. $\overline{F_v}/\tau = S$, that is, S is the set of clauses of $\overline{F_v}$ that are satisfied by τ,
2. $(F_v/\tau) \cup S' = F_v$ that is S' contains every clauses of F_v that are *not* satisfied by τ.

The intuition behind the notion of shape is that if $\tau : X_v \to \{0,1\}$ has shape (S, S') then it can be extended to a satisfying assignment of F_v by combining it with $\tau' : \overline{X_v} \to \{0,1\}$ such that F_v/τ' – the clauses of F_v that are satisfied by τ' – is S'.

We write $N_v^T(\vec{S})$ for the set of assignments of shape $\vec{S}$ or simply $N_v(\vec{S})$ if T is clear from the context. Observe that if T is of PS-width k, then there is at most k^2 different shapes for v. However, N_v^T does not partition the set of assignments. Indeed, an assignment τ can have more than one shape. If τ has shape $\vec{S_1} = (S_1, S_1')$ and shape $\vec{S_2} = (S_2, S_2')$, then it only implies $S_1 = S_2 = \overline{F_v}/\tau$.

We now explain how shapes for an inner node can be related to shapes for its child nodes. Let $\vec{S} = (S, S')$ be a shape for an inner node v of T, and let

$\vec{S}_1 = (S_1, S_1')$, $\vec{S}_2 = (S_2, S_2')$ be shapes for its children v_1 and v_2, respectively. We say that $\vec{S}_1$ and $\vec{S}_2$ *generate* $\vec{S}$ if

(a) $S = (S_1 \cup S_2) \cap \overline{F_v}$,

(b) $S_1' = (S' \cup S_2) \cap F_{v_1}$, and

(c) $S_2' = (S' \cup S_1) \cap F_{v_2}$.

The following result relates the shapes for an inner node to the generating shapes for its children.

Lemma 3.1. *Let F be a formula, let T be a branch decomposition of $F \cup \mathsf{var}(F)$, and let v be an inner node of T with children v_1 and v_2. Let $\vec{S}$ be a shape for v, and let G denote the set of pairs of shapes $\vec{S}_1$ for v_1 and $\vec{S}_2$ for v_2 such that $\vec{S}_1$ and $\vec{S}_2$ generate $\vec{S}$. Then*

$$N_v(\vec{S}) = \biguplus_{(\vec{S}_1,\vec{S}_2)\in G} \{\tau_1 \cup \tau_2 \mid \tau_1 \in N_{v_1}(\vec{S}_1), \tau_2 \in N_{v_2}(\vec{S}_2)\}.$$

Lemma 3.1 is an easy consequence of the following two lemmas that can be found in [SS13].

Lemma 3.2. *Let v be a node of T with children v_1 and v_2. Let $\vec{S}_1 = (S_1, S_1')$ be a shape for v_1, let $\vec{S}_2 = (S_2, S_2')$ be a shape for v_2, and let $\vec{S} = (S, S')$ be a shape for v generated by $\vec{S}_1$ and $\vec{S}_2$. If $\tau_1 \in N_{v_1}(\vec{S}_1)$ and $\tau_2 \in N_{v_2}(\vec{S}_2)$ then $\tau_1 \cup \tau_2 \in N_v(\vec{S})$.*

Proof. Let $\tau_1 \in N_{v_1}(\vec{S}_1)$ and $\tau_2 \in N_{v_2}(\vec{S}_2)$. As $\overline{F_v} \subseteq \overline{F_{v_1}} \cup \overline{F_{v_2}}$, $\overline{F_{v_1}}/\tau_1 = S_1$ and $\overline{F_{v_2}}/\tau_2 = S_2$, we get $\overline{F_v}/(\tau_1 \cup \tau_2) = (S_1 \cup S_2) \cap \overline{F_v} = S$. This shows that Condition 1 is satisfied.

Consider a clause $C \in F_v$ and assume without loss of generality that $C \in F_{v_1}$. Suppose $C \notin F_v/(\tau_1 \cup \tau_2)$. In particular, τ_1 does not satisfy C that is, $C \notin F_{v_1}/\tau_1$. By condition 2, $(F_{v_1}/\tau_1 \cup S_1') = F_{v_1}$, thus $C \in S_1'$.

Moreover τ_2 does not satisfy C either, so $C \notin S_2$. The shapes $\vec{S}_1$ and $\vec{S}_2$ generate $\vec{S}$, so $S_1' \subseteq S' \cup S_2$ and thus $C \in S'$ by Condition (b). This proves that $F_v/(\tau_1 \cup \tau_2) \cup S' = F_v$, that is Condition 2 is satisfied. We conclude that $\tau_1 \cup \tau_2$ has shape $\vec{S}$ as claimed. □

Lemma 3.3. *Let v be a node of T with children v_1 and v_2, let $\vec{S} = (S, S')$ be a shape for v, and let $\tau \in N_v(\vec{S})$. Let τ_1 and τ_2 denote the restrictions of τ to X_{v_1} and X_{v_2}, respectively. There is a unique pair of shapes $\vec{S}_1$ for v_1 and $\vec{S}_2$ for v_2 generating $\vec{S}$ such that $\tau_1 \in N_{v_1}(\vec{S}_1)$ and $\tau_2 \in N_{v_2}(\vec{S}_2)$.*

Proof. Let

- $S_1 = \overline{F_{v_1}}/\tau_1$,
- $S_2 = \overline{F_{v_2}}/\tau_2$,

- $S'_1 = (S' \cup S_2) \cap F_{v_1}$ and,
- $S'_2 = (S' \cup S_1) \cap F_{v_2}$.

Observe that $S'_1 \in \mathsf{proj}(F_{v_1}, \overline{X_{v_1}})$. Indeed, let $\tau' : \overline{X_v} \to \{0,1\}$ be such that $F_v/\tau' = S'$, it holds that $F_{v_1}/(\tau' \cup \tau_2) = (S_2 \cup S') \cap F_{v_1} = S'_1$. Similarly, $F_{v_2}/(\tau' \cup \tau_1) = (S_1 \cup S') \cap F_{v_2}$. It follows that $\vec{S}_1 = (S_1, S'_1)$ is a shape for v_1 and $\vec{S}_2 = (S_2, S'_2)$ is a shape for v_2. By definition, $\vec{S}1$ and $\vec{S}_2$ generate $\vec{S}$.

Observe that $F_{v_1} = (F_{v_1}/\tau_1) \cup S'_1$. It is clear that $(F_{v_1}/\tau_1) \cup S'_1 \subseteq F_{v_1}$ since $S'_1 \subseteq F_{v_1}$ and $F_{v_1}/\tau \subseteq F_{v_1}$ by definition. Moreover, if $C \in S'_1 \setminus (F_{v_1}/\tau)$ then by definition is not satisfied by τ_1. We have two cases:

- either C is satisfied by τ_2 and in this case, $C \in S_2$ that is, $C \in S'_1$,
- or C is not satisfied by τ_2 and in this case, it is not satisfied by $\tau = \tau_1 \cup \tau_2$, and since τ has shape (S, S') we have $C \in S'$, that is, $C \in S'_1$.

Since $S_1 = \overline{F_{v_1}}/\tau_1$ by definition, τ_1 has shape $\vec{S}_1$. Similarly, we have $F_{v_2} = (F_{v_2}/\tau_2) \cup S'_2$ and since $S_2 = \overline{F_{v_2}}/\tau_2$ by definition, τ_2 has shape $\vec{S}_2$.

Finally, we prove the uniqueness of $\vec{S}_1$ and $\vec{S}_2$. Let $\vec{R}_1 = (R_1, R'_1)$ and $\vec{R}_2 = (R_2, R'_2)$ be shapes for v_1 and v_2 such that $\vec{R}_1$ and $\vec{R}_2$ generate $\vec{S}$ and such that $\tau_1 \in N_{v_1}(\vec{R}_1)$ and $\tau_2 \in N_{v_2}(\vec{R}_2)$. We have $R_1 = \overline{F_{v_1}}(\tau_1) = S_1$ and $R_2 = \overline{F_{v_2}}(\tau_2) = S_2$ by Condition 1. As $\vec{R}_1$ and $\vec{R}_2$ generate $\vec{S}$, we further have $R'_1 = (S' \cup R_2) \cap F_{v_1}$ and $R'_2 = (S' \cup R_1) \cap F_{v_2}$. That is, $R'_1 = S'_1$ and $R'_2 = S'_2$, so $\vec{R}_1 = \vec{S}_1$ and $\vec{R}_2 = \vec{S}_2$. □

3.1.2 Constructing a Structured d-DNNF

Lemma 3.1 can be turned into a recurrence for determining the model count of F by dynamic programming [SS13, STV14]. It can also be used to construct a structured d-DNNF for F.

To simplify matters, for the remainder of this section let F be an arbitrary, but fixed, formula, and let T be an arbitrary, but fixed, branch decomposition of $F \cup \mathsf{var}(F)$. Starting at the leaves of T, we are going to construct a d-DNNF $\varphi_v(\vec{S})$ for each node v and each shape $\vec{S}$ for v. The intending meaning of $\varphi_v(\vec{S})$ is that the satisfying assignment of $\varphi_v(\vec{S})$ are exactly the assignments of $N_v(\vec{S})$.

For a leaf node v of T, we have to consider two cases:

1. Suppose $\delta(v) = x$ for a variable x of F. For $\ell \in \{x, \neg x\}$, let τ_ℓ denote the assignment $\tau_\ell : \{x\} \to \{0,1\}$ such that $\tau(\ell) = 1$. The pairs $\vec{S}_x = (F/\tau_x, \emptyset)$ and $\vec{S}_{\neg x} = (F/\tau_{\neg x}, \emptyset)$ are the only shapes for v, and $N_v(\vec{S}_x) = \{\tau_x\}$ as well as $N_v(\vec{S}_{\neg x}) = \{\tau_{\neg x}\}$. Accordingly, we let $\varphi_v(\vec{S}_x) \equiv x$ and $\varphi_v(\vec{S}_{\neg x}) \equiv \neg x$.

2. Let $\delta(v) = C$ for a clause $C \in F$. The pairs $\vec{S}_\bot = (\emptyset, \emptyset)$ and $\vec{S}_\top = (\emptyset, \{C\})$ are the only shapes for v. Since $X_v = \emptyset$ it suffices to determine whether the empty assignment $\varepsilon : \emptyset \to \{0,1\}$ has one of these shapes. Because the empty assignment does not satisfy any clause we get $N_v(\vec{S}_\top) = \{\varepsilon\}$ and $N_v(\vec{S}_\bot) = \emptyset$, so we define $\varphi_v(\vec{S}_\bot) \equiv 0$ and $\varphi_v(\vec{S}_\top) \equiv 1$.

Let v be an inner node of T with children v_1 and v_2, and assume we have constructed $\varphi_{v_1}(\vec{S}_1)$ for each shape $\vec{S}_1$ for v_1 and $\varphi_{v_2}(\vec{S}_2)$ for each shape $\vec{S}_2$ for v_2. Let $\vec{S}$ be a shape for v and let G denote the set of pairs of shapes $\vec{S}_1$ for v_1 and $\vec{S}_2$ for v_2 that generate $\vec{S}$. We construct $\varphi_v(\vec{S})$ as

$$\varphi_v(\vec{S}) \equiv \bigvee_{(\vec{S}_1,\vec{S}_2)\in G} \varphi_{v_1}(\vec{S}_1) \wedge \varphi_{v_2}(\vec{S}_2). \tag{3.1}$$

That is, we create an AND node conjoining every pair $\varphi_{v_1}(\vec{S}_1)$ and $\varphi_{v_2}(\vec{S}_2)$ such that $\vec{S}_1$ and $\vec{S}_2$ generate $\vec{S}$, and then add an OR node that has an incoming edge from each AND node thus created. We assume that the resulting DNNF has been simplified by propagating constants.

Lemma 3.4. *For each node v of T and shape $\vec{S}$ for v, $\varphi_v(\vec{S})$ is a* d-DNNF *such that* $\mathsf{var}(\varphi_v(\vec{S})) \subseteq X_v$ *and such that an assignment $\tau : X_v \to \{0,1\}$ satisfies $\varphi_v(\vec{S})$ if, and only if, $\tau \in N_v(\vec{S})$.*

Proof. It is easy to check that the statement holds for each leaf node v of T. Let v be an inner node and suppose the statement holds for its children v_1 and v_2. Let $\vec{S}$ be a shape for v. By assumption, $\mathsf{var}(\varphi_{v_1}(\vec{S}_1)) \subseteq X_{v_1}$ and $\mathsf{var}(\varphi_{v_2}(\vec{S}_2)) \subseteq X_{v_2}$ for every shape $\vec{S}_1$ for v_1 and every shape $\vec{S}_2$ for v_2. We have $X_v = X_{v_1} \cup X_{v_2}$ and since X_{v_1} and X_{v_2} are disjoint it follows that $\varphi_v(\vec{S})$ is a DNNF satisfying $\mathsf{var}(\varphi_v(\vec{S})) \subseteq X_v$. Let $\tau : X_v \to \{0,1\}$ be a satisfying assignment of $\varphi_v(\vec{S})$, and let τ_1 and τ_2 denote the restrictions of τ to X_{v_1} and X_{v_2}, respectively. By definition of $\varphi_v(\vec{S})$, there is a pair of shapes $\vec{S}_1$ and $\vec{S}_2$ generating $\vec{S}$ such that τ satisfies the disjunct $\varphi_{v_1}(\vec{S}_1) \wedge \varphi_{v_2}(\vec{S}_2)$. By assumption, the lemma holds for v_1 and v_2. In particular, $\mathsf{var}(\varphi_{v_1}(\vec{S}_1)) \subseteq X_{v_1}$ and $\mathsf{var}(\varphi_{v_2}(\vec{S}_2)) \subseteq X_{v_2}$, so τ_1 satisfies $\varphi_{v_1}(\vec{S}_1)$ and τ_2 satisfies $\varphi_{v_2}(\vec{S}_2)$, which in turn implies that $\tau_1 \in N_{v_1}(\vec{S}_1)$ and $\tau_2 \in N_{v_2}(\vec{S}_2)$. It now follows from Lemma 3.1 that τ has shape $\vec{S}$.

In addition, Lemma 3.1 tells us that $(\vec{S}_1, \vec{S}_2)$ is the unique pair of shapes generating $\vec{S}$ such that τ_1 has shape $\vec{S}_1$ and τ_2 has shape $\vec{S}_2$. Thus $\varphi_{v_1}(\vec{S}_1) \wedge \varphi_{v_2}(\vec{S}_2)$ is the unique disjunct satisfied by τ. By assumption, $\varphi_{v_1}(\vec{S}'_1)$ and $\varphi_{v_2}(\vec{S}'_2)$ are deterministic DNNFs for each shape $\vec{S}'_1$ for v_1 and $\vec{S}'_2$ for v_2, so $\varphi_v(\vec{S})$ is deterministic as well.

Now let $\tau : X_v \to \{0,1\}$ be an assignment of shape $\vec{S}$, and let τ_1 and τ_2 denote its restrictions to X_{v_1} and X_{v_2}, respectively. By Lemma 3.1, there has to be a pair $(\vec{S}_1, \vec{S}_2)$ of shapes $\vec{S}_1$ for v_1 and $\vec{S}_2$ for v_2 generating $\vec{S}$ such that $\tau_1 \in N_{v_1}(\vec{S}_1)$ and $\tau_2 \in N_{v_2}(\vec{S}_2)$. It follows from our assumption that the lemma holds for v_1 and v_2 that τ_1 satisfies $\varphi_{v_1}(\vec{S}_1)$ and that τ_2 satisfies $\varphi_{v_2}(\vec{S}_2)$. Thus τ satisfies $\varphi_{v_1}(\vec{S}_1) \wedge \varphi_{v_2}(\vec{S}_2)$, hence τ satisfies $\varphi_v(\vec{S})$. □

To show that $\varphi_v(\vec{S})$ is a *structured* DNNF, we have to provide a vtree respected by $\varphi_v(\vec{S})$. For a node v of T, let $\mathit{vtree}(T, v) = T'$, where T' is the tree obtained from the subtree T_v by deleting all leaves w labeled by a clause of F, followed—if necessary—by a sequence of operations to make the resulting tree binary. Verify

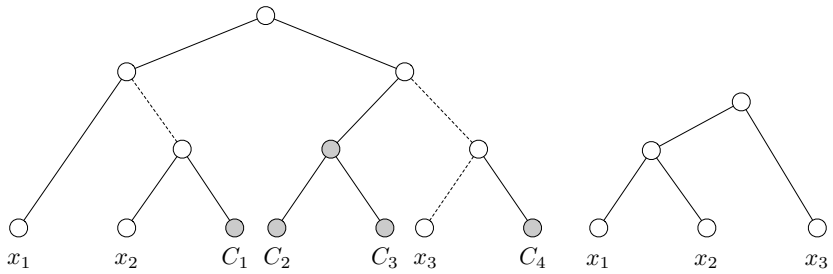

Figure 3.1: The tree on the left is a branch decomposition of a formula $F = \{C_1, C_2, C_3, C_4\}$ with $\mathsf{var}(F) = \{x_1, x_2, x_3\}$. To obtain the vtree on the right, we first delete each leaf node associated with a clause, as well inner nodes turned into leaf nodes by these deletions (the corresponding vertices are shown in gray). The resulting tree is turned into a binary tree by contracting edges incident to nodes of degree two (these edges are represented by dashed lines).

that $\mathit{vtree}(T, v)$ is a branch decomposition of X_v and hence a vtree. We illustrate this construction in Figure 3.1.

Lemma 3.5. *For each node v of T and shape $\vec{S}$ for v, the* DNNF $\varphi_v(\vec{S})$ *respects* $\mathit{vtree}(\vec{T}, v)$.

Proof. The lemma trivially holds for each leaf node v of T and shape $\vec{S}$ for v, as $\varphi_v(\vec{S})$ does not contain any AND nodes. Let v be an inner node of T with children v_1 and v_2, and assume the lemma holds for v_1 and v_2 and their respective shapes. Let $\vec{S}$ be a shape for v. By construction, each AND node introduced in $\varphi_v(\vec{S})$ computes a conjunction $\varphi_{v_1}(\vec{S}_1) \wedge \varphi_{v_2}(\vec{S}_2)$, where $\vec{S}_1$ and $\vec{S}_2$ are shapes for v_1 and v_2, respectively, that generate $\vec{S}$. Since we assume $\varphi_v(\vec{S})$ to be simplified, both X_{v_1} and X_{v_2} have to be nonempty: otherwise, one of the conjuncts $\varphi_{v_i}(\vec{S}_i)$ for $i \in \{1, 2\}$ would satisfy $\mathsf{var}(\varphi_{v_i}(\vec{S}_i)) = \emptyset$ by Lemma 3.4 and would have been simplified to a constant, which in turn would have been propagated through the AND node. Let $\mathit{vtree}(T, v) = T'$, let $\mathit{vtree}(T, v_1) = T_1$, and let $\mathit{vtree}(T, v_2) = T_2$. As both X_{v_1} and X_{v_2} are nonempty, T' is a binary tree whose principal subtrees are T_1 and T_2. By Lemma 3.4, the conjuncts satisfy $\mathsf{var}(\varphi_{v_1}(\vec{S}_1)) \subseteq X_{v_1}$ and $\mathsf{var}(\varphi_{v_2}(\vec{S}_2)) \subseteq X_{v_2}$. In combination with the assumption that the DNNF $\varphi_{v_i}(\vec{S}'_i)$ respects $\mathit{vtree}(T, v_i)$ for each $i \in \{1, 2\}$ and shape $\vec{S}'_i$ for v_i, this implies that $\varphi_v(\vec{S})$ respects $\mathit{vtree}(T, v)$. □

Let r denote the root of T and let $\vec{\emptyset} = (\emptyset, \emptyset)$. We now prove that our construction yields a structured d-DNNF representation of F.

Lemma 3.6. *The pair $\vec{\emptyset}$ is the only shape for r and $\varphi_r(\vec{\emptyset})$ is a structured* d-DNNF *computing F.*

Proof. The first part follows from the fact that $X_r = \mathsf{var}(F)$ and $F_r = F$, so that $\overline{X_r} = \emptyset$ and $\overline{F_r} = \emptyset$. By Lemma 3.4 and Lemma 3.5, $\varphi_r(\vec{\emptyset})$ is a structured

d-DNNF such that an assignment $\tau : \mathsf{var}(F) \to \{0,1\}$ satisfies $\varphi_r(\vec{\emptyset})$ if, and only if, $\tau \in N_r(\vec{\emptyset})$. Condition 2 states that if an assignment $\tau : \mathsf{var}(F) \to \{0,1\}$ has shape $\vec{\emptyset}$ then $(F/\tau) \cup \emptyset = F$ that is τ satisfies every clauses of F. In other words, $N_r(\vec{\emptyset})$ is the set of satisfying assignments of F. □

Let n be the number of variables of F, let m be the number of clauses in F, and let k denote the PS-width of T. The size of the structured d-DNNF constructed for F can be bounded as follows.

Lemma 3.7. *The* DNNF $\varphi_r(\vec{\emptyset})$ *has size at most* $(5k^3+2)(n+m)$.

Proof. We can assume without loss of generality that T contains at least one inner node. Let v be an inner node of T with children v_1 and v_2. Consider the DNNFs $\varphi_{v_1}(\vec{S}_1)$ for shapes $\vec{S}_1$ for v_1 and $\varphi_{v_2}(\vec{S}_2)$ for shapes $\vec{S}_2$ for v_2. We claim that all DNNFs $\varphi_v(\vec{S})$ for shapes $\vec{S}$ of v can be constructed from these DNNFs by introducing at most $5k^3$ new nodes and edges. If $\vec{S}$ is a shape for v and $\vec{S}_1$ and $\vec{S}_2$ are shapes for v_1 and v_2 that generate $\vec{S}$, we have to introduce an AND node and two edges to construct the DNNF computing $\varphi_{v_1}(\vec{S}_1) \wedge \varphi_{v_2}(\vec{S}_2)$, as well an edge from this AND node to the OR node that will eventually compute $\varphi_v(\vec{S})$. In the worst case, we have to create this OR node first. In total, we have to introduce at most 5 nodes and edges for each triple $(\vec{S}, \vec{S}_1, \vec{S}_2)$ of shapes such that $\vec{S}_1$ and $\vec{S}_2$ generate $\vec{S}$. How many such triples are there? For any three projections $S_1 \in \mathsf{proj}(\overline{F_{v_1}}, X_{v_1})$, $S_2 \in \mathsf{proj}(\overline{F_{v_2}}, X_{v_2})$, and $S' \in \mathsf{proj}(F_v, \overline{X_v})$, the projections $S'_1 \in proj(F_{v_1}, \overline{X_{v_1}})$, $S'_2 \in proj(F_{v_2}, \overline{X_{v_2}})$, and $S \in \mathsf{proj}(\overline{F_v}, X_v)$ such that $\vec{S}_1 = (S_1, S'_1)$ and $\vec{S}_2 = (S_2, S'_2)$ generate $\vec{S} = (S, S')$ is uniquely determined. As there are at most k^3 such projections, we have to introduce at most $5k^3$ nodes and edges. The tree T has exactly $n+m-1$ inner nodes, so we need at most $5k^3(n+m)$ nodes and edges to construct the DNNF $\varphi_r(\vec{\emptyset})$ from the DNNFs constructed for leaves of T. For each leaf node there at most two DNNFs consisting of a single node and there are $n+m$ leaves, so we require at most $(5k^3+2)(n+m)$ nodes and edges in total. □

Since we did not make any assumptions about the formula F and the branch decomposition T, Lemma 3.6 and Lemma 3.7 yield the following result.

Theorem 3.8. *A CNF formula with n variables, m clauses, and PS-width k can be compiled into a structured* d-DNNF *of size $O(k^3(n+m))$.*

The above construction leads to an algorithm which, given a formula F and a branch decomposition T of $F \cup \mathsf{var}(F)$, computes a structured d-DNNF representation of F. The pseudo code listed as Algorithm 4 provides the outlines of this procedure.[1] We can show that Algorithm 4 runs in time $O(k^3 m(n+m))$ making Theorem 3.8 effective: not only a small d-DNNF exists but it can be computed efficiently:

[1] To enhance readability, we suppress double brackets around shapes, writing, for instance, $\varphi_v(S, S')$ instead of $\varphi_v((S, S'))$.

Theorem 3.9. *A CNF formula with n variables, m clauses, and PS-width k can be compiled into a structured* d-DNNF *of size $O(k^3(n+m))$ in time $O(k^3m(n+m))$.*

Proof. We have to show that Algorithm 4 runs in time $O(k^3m(n+m))$. If for every vertex v of T we have precomputed every shape $\vec{S}$ for v and the shapes for its children v_1, v_2 that generates it, then the construction of the d-DNNF can easily be done in time $O(k^3(n+m))$.

Thus the only remaining step if to compute the shapes, that is, the sets $\mathsf{proj}(F_v, \overline{X_v})$ and $\mathsf{proj}(\overline{F_v}, X_v)$ for every vertex v of T. It is shown in [STV14] that it can be done in time $O(k^3m(n+m))$ which completes the proof. □

Theorem 2.15 from [STV14] stating that #SAT is tractable on bounded PS-width instances can be seen as a corollary of Theorem 3.9: compile a d-DNNF D for F in time $O(k^3m(n+m))$ and then use Proposition 1.57 to count its number of satisfying assignments in time $O(\mathsf{size}(D)) = O(k^3(n+m))$.

Observe that we need roughly the same time for constructing the deterministic DNNF than what is needed to count the number of satisfying assignments in [STV14] but the size of the resulting DNNF is smaller by a factor of m. This may be of interest if the formula is queried many times for counting after conditioning.

The question of the optimality of such a compilation algorithm is still open. It would be interesting to understand if deterministic d-DNNF is the best representation language we can hope for. In particular:

Open question 1. *Can we compile formulas of small* PS*-width into succinct* dec-DNNF*?*

In our compilation algorithm, determinism is ensured by the fact that our assignments have different projections on some subformula which seems stronger than the way dec-DNNF ensured determinism. To be able to compile formulas of small PS-width into succinct dec-DNNF would require a better understanding of the structure of the formula.

3.2 Consequences of the compilation algorithm

3.2.1 Compilation for other graph measures

In Section 2.3.1, we used Theorem 2.15 to prove that #SAT was tractable for many graph parameters. The same can be done for compilation into structured d-DNNF using Theorem 3.8 and the relations between PS-width and the other graph parameters we have shown in Section 2.3.1.

Corollary 3.10. *A formula with n variables, m clauses, and incidence treewidth k can be compiled into a structured* d-DNNF *of size $O(8^k(n+m))$.*

Corollary 3.11. *A formula with n variables, m clauses, and incidence cliquewidth k can be compiled into a structured* d-DNNF *of size $O(m^{3k}(n+m))$.*

Algorithm 4: Compiling CNFs into structured d-DNNFs.

Input: a CNF F and a branch decomposition (T, δ) of $F \cup \mathsf{var}(F)$
Output: a structured d-DNNF computing F
// initialization, precomputing shapes
for v in T
  compute $\mathsf{proj}(\overline{F_v}, X_v)$ and $\mathsf{proj}(F_v, \overline{X_v})$
// compilation, leaf nodes
for v in $L(T)$
  if $\delta(v)$ in $\mathsf{var}(F)$
    $x = \delta(v)$
    $S_x = \{C \in F \mid x \in C\}$
    $S_{\neg x} = \{C \in F \mid \neg x \in C\}$
    $\varphi_v(S_x, \emptyset) = x$
    $\varphi_v(S_{\neg x}, \emptyset) = \neg x$
  else
    $C = \delta(v)$
    $\varphi_v(\emptyset, \{C\}) = 1$
    $\varphi_v(\emptyset, \emptyset) = 0$
  mark v as processed
// compilation, inner nodes
while T contains an unprocessed node
  let v be an unprocessed node whose children v_1 and v_2 have been processed
  for (S_1, S_2, S') in $\mathsf{proj}(\overline{F_{v_1}}, X_{v_1}) \times \mathsf{proj}(\overline{F_{v_2}}, X_{v_2}) \times \mathsf{proj}(F_v, \overline{X_v})$
    $S = S_1 \cup S_2$
    $S'_1 = S' \cup S_2$
    $S'_2 = S' \cup S_1$
    if $\varphi_v(S, S')$ has not been created
      // initialize $\varphi_v(S, S')$
      $\varphi_v(S, S') = 0$
    $\varphi_v(S, S') = \varphi_v(S, S') \vee (\varphi_{v_1}(S_1, S'_1) \wedge \varphi_{v_2}(S_2, S'_2))$
    propagate constants in $\varphi_v(S, S')$
  mark v as processed
return $\varphi_r(\emptyset, \emptyset)$

Corollary 3.12. *A formula with n variables, m clauses, and directed incidence clique-width k can be compiled into a structured* d-DNNF *of size $O(64^k(n+m))$.*

Consequently, every query supported in polynomial time by d-DNNF can be shown to be tractable for these structural restrictions. For example, we can directly use Corollary 3.10 together with Proposition 1.56 to show that the satisfying assignments of CNF-formula F of tree width k can be enumerated with a preprocessing of $O(8^k(n+m)m)$ and a delay of $O(8^k(n+m)n)$ where n is the number of variables of F and m the number of clauses of F.

3.2.2 Solving MaxSAT

It is shown in [STV14] that MaxSAT is tractable on instances of bounded PS-width. This result is proven along the same lines of Theorem 2.15, using a modified version of the dynamic programming algorithm. Since the structure of the CNF-formula is lost during the compilation, Theorem 3.8 does not imply this result directly. However, we can still use Theorem 3.8 to compute MaxSAT and other optimization problems on bounded PS-width CNF-formula. The idea is to add a new variable for each clause that will preserve the structure of F during compilation without increasing its PS-width.

Let F be a CNF formula and let $Y = \{y_C \mid C \in F\}$ be a set of variables disjoint from $\mathsf{var}(F)$ of size $|F|$. We denote by $\hat{F} = \{C \cup \{y_C\} \mid C \in F\}$. Transforming F into $\hat{F}$ does not increase the PS-width of F:

Lemma 3.13. *Let F be a* CNF*-formula. Then* PS*-width*$(\hat{F}) \leq$ PS*-width*(F).

Proof. Let T be a branch decomposition of $F \cup \mathsf{var}(F)$ of PS-width k. We construct a branch decomposition T' of $\hat{F} \cup \mathsf{var}(\hat{F})$ by replacing each leaf of T labeled by a clause $C \in F$ with two leaves labeled by y_C and $C \cup \{y_C\} \in \hat{F}$. To ease notations, we denote by $F' = \hat{F}$. We claim that the PS-width of T' is k.

As usual, for a vertex t of T, we denote by $F_t = L(T_t) \cap F$, $X_t = L(T_t) \cap \mathsf{var}(F)$, $\overline{F_t} = F \setminus F_t$, $\overline{X_t} = \mathsf{var}(F) \setminus X_t$. Similarly, for a vertex t of T', we denote by $F'_t = L(T'_t) \cap F'$, $X'_t = L(T'_t) \cap \mathsf{var}(F')$, $\overline{F'_t} = F' \setminus F'_t$, $\overline{X'_t} = \mathsf{var}(F') \setminus X'_t$.

By identifying every vertex t of T that is not a leaf with it corresponding vertex t in T', we have $F'_t = F_t$ and $\overline{F'_t} = \overline{F_t}$, if we identify the clauses of F with those of F'. Moreover, $X'_t = X_t \cup \{y_C \mid C \in F_t\}$ and $\overline{X'_t} = \overline{X_t} \cup \{y_C \mid C \in \overline{F'_t}\}$. In particular, for every clause C of F, every variables y_C is on the same side of T'_t as its corresponding clause in F'.

If τ is a truth assignment of $\overline{X'_t}$, then the projection F'_t/τ does not depend on the value of τ on variables y_C for $C \in \overline{F'_t}$ since these variables are not in F'_t. Thus, $F'_t/\tau = F_t/(\tau|_{\overline{X_t}})$. Similarly, if τ is a truth assignment of X'_t, $\overline{F'_t}/\tau = \overline{F_t}/(\tau|_{X_t})$. Thus PS-width$(T')$ = PS-width(T) that is PS-width$(F') \leq$ PS-width(F). □

Lemma 3.13 can be combined with Theorem 3.8 to efficiently compile $\hat{F}$ for a formula F of PS-width k into a deterministic DNNF. Such a DNNF still contains information on the structure of F that can be exploited to solve new problems on

F. For example, if $F' \subseteq F$, then we can transform a DNNF D for $\hat{F}$ into a DNNF for $\hat{F'}$ by simply conditioning D on $\{y_C \mapsto 1 \mid C \notin F'\}$.

The satisfying assignments of $\hat{F}$ relate nicely with the maximum number of clauses that can be satisfied in F. Given a boolean function f, a satisfying assignment $\tau \in \mathsf{sat}(f)$ is said minimal w.r.t $X \subseteq \mathsf{var}(f)$ if for every $\tau' \in \mathsf{sat}(f)$, $|\tau^{-1}(0) \cap X| \geq |\tau'^{-1}(0) \cap X|$.

Lemma 3.14. *Let F be a* CNF *formula. Let $\tau \in \mathsf{sat}(\hat{F})$ be a minimal satisfying assignment of $\hat{F}$ w.r.t $Y = \{y_C \mid C \in F\}$. It holds that*

$$\mathsf{MaxSAT}(F) = |\tau^{-1}(0) \cap Y|.$$

Proof. Let μ be a truth assignment of $\mathsf{var}(F)$ that satisfies $m = \mathsf{MaxSAT}(F)$ clauses. Let μ' be the truth assignment of $\mathsf{var}(\hat{F})$ that agrees with μ on $\mathsf{var}(F)$ and such that $\mu'(y_C) = 1$ if $\mu \not\models C$ and $\mu'(y_C) = 0$ otherwise. Then μ' is clearly a satisfying assignment of $\hat{F}$ and thus $m = |\mu'^{-1}(0) \cap Y| \leq |\tau^{-1}(0) \cap Y|$ since τ is minimal w.r.t Y. Moreover observe that for every $C \in F$ such that $\tau(y_C) = 0$ we have $\tau|_{\mathsf{var}(F)} \models C$. In other words, $|\tau^{-1}(0) \cap Y| \leq m$. That is $m = |\tau^{-1}(0) \cap Y|$. □

As observed in [Dar01a], minimal satisfying assignments of a DNNF can easily be found.

Proposition 3.15. *Let D be a* DNNF *and let $X \subseteq \mathsf{var}(D)$. A minimal satisfying assignment of D w.r.t X can be found in time $O(\mathsf{size}(D))$.*

Proof (Sketch). The key observation is that if $D = D_1 \wedge D_2$ with $\mathsf{var}(D_1) \cap \mathsf{var}(D_2) = \emptyset$, and if τ_i is a minimal satisfying assignment of D_i w.r.t. $\mathsf{var}(D_i) \cap X$ for $i \in \{1, 2\}$, then $\tau_1 \cup \tau_2$ is minimal for D w.r.t. X.

Moreover, if $D = D_1 \vee D_2$, we let τ_i be a minimal satisfying assignment of D_i w.r.t $\mathsf{var}(D_i) \cap X$ for $i \in \{1, 2\}$. Assume w.l.o.g that $|\tau_1^{-1}(1) \cap X| \leq |\tau_2^{-1}(1) \cap X|$. Then $\tau_1 \cup \{x \mapsto 0 \mid x \in \mathsf{var}(D) \setminus \mathsf{var}(D_1)\}$ is a minimal satisfying assignment of D w.r.t x.

Dynamically propagating this satisfying assignment from the input of D to its output can be done in time $O(\mathsf{size}(D))$. □

Proposition 3.15 combined with Lemma 3.14 yields the following:

Theorem 3.16. MaxSAT *can be solved in time $O(k^3 m(n+m))$ on CNF formulas with n variables and m clauses.*

Chapter 4

Compilation of β-acyclic formulas

CNF-formulas whose underlying hypergraph is β-acyclic remained a singularity among the tractable families for SAT for a while. For the tractable families presented in Chapter 2, the algorithm for decision [Sze04] could easily be generalized to counting [SS10] or even directly designed for counting [SS13, PSS13, CDM14, STV14] and later, as we have seen in Chapter 3, generalized to compilation [BCMS15] which uniformly explains why almost the same algorithm works for other queries as well. All such structure-based algorithms perform a dynamic programming algorithm along a branch decomposition of small width. This similarity is explained in Chapter 2 since the structural restrictions they are using have bounded PS-width. In contrast, only decision of β-acyclic formulas was known to be tractable [OPS13]. Every attempt at solving #SAT using the classical dynamic programming approach failed. The main reason is that no characterization of β-acyclicity in terms of branch decomposition were known. Moreover, the algorithm proposed by Ordyniak, Paulusma and Szeider is based on resolution, a well-known algorithm for SAT that does not generalize to counting.

In this chapter, we settle the complexity of #SAT on β-acyclic formulas by giving an efficient compilation algorithm of such formulas into dec-DNNF. In addition, we show that for every n there exist β-acyclic formulas of size $\Omega(n)$ whose PS-width is $2^{\Omega(n)}$. This deviation from the framework of [STV14] is an evidence that the classical dynamic programming approach is unlikely to work on such instances.

This chapter is organized as follows. In a first section, we show the exponential lower bound on the PS-width of β-acyclic formulas which motivates the quest for a compilation algorithm different from the one of Chapter 3 by using new techniques. The second section proves new results on the structure of β-acyclic hypergraph that will be used for our algorithm but are of independent interest. Finally we present the compilation algorithm of β-acyclic formulas into succinct dec-DNNF and give some immediate corollaries of the existence of such algorithm.

4.1 Incomparability with other measures

In this section, we construct β-acyclic formulas which have a PS-width exponential in their size. This result can be found in [BCM15]. PS-width is not a usual structural parameter since it does not only depend on the incidence graph of the formula thus, to compare β-acyclicity with PS-width, we need to relate PS-width with a graph parameter. The following connects PS-width (of monotone formulas) and MIM-width:

Lemma 4.1. *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 $(A, \overline{A})$. Set $X := \mathsf{var}(F) \cap A$, $\overline{X} := \mathsf{var}(F) \cap \overline{A}$, $\mathcal{C} := F \cap A$ and $\overline{\mathcal{C}} := F \cap \overline{A}$. Moreover, let M be a maximum induced matching of $G[A, \overline{A}]$ and let V_M be the end vertices of M.

First assume that $|\mathcal{C} \cap V_M| \geq |\overline{\mathcal{C}} \cap V_M|$. 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 $\overline{X} \cap V_M$. Note that $k \geq |M|/2$. Since M is an induced matching, every clause C_i contains exactly one of the variables x_j, and we assume w.l.o.g. that C_i contains x_i. Let a be an assignment to the x_i and let a' be the extended assignment of $\overline{X}$ that we get by assigning 0 to all other variables. Then a' satisfies in $F_{\overline{X},C}$ 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 $|PS(F_{\overline{X},C})| \geq 2^k \geq 2^{|M|/2}$.

For $|\mathcal{C} \cap V_M| \leq |\overline{\mathcal{C}} \cap V_M|$ it follows symmetrically that $|PS(F_{X,\overline{C}})| \geq 2^{|M|/2}$.

Consequently, we have in either case that the PS-width of F is at least $2^{|M|/2}$ and the claim follows. □

We now give a way of transforming a graph G into a chordal bipartite graph whose MIM-width is linearly related to the tree width of G. We then choose the right graph G to obtain the desired lower bound.

Given 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 $p_{e,u}, q_{e,u}, p_{e,v}, q_{e,v} \in V'$,
- for 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'. Figure 4.1 illustrates the transformation.

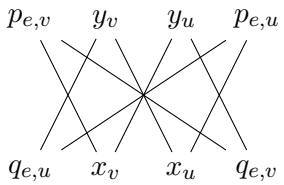

Figure 4.1: The construction of G': transforming the edge uv of G

Lemma 4.2. *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 4.3. *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 color 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 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 $(X, \overline{X})$. Let $M \subseteq E$ be a matching in $G[X, \overline{X}]$. Let $(X', \overline{X}')$ 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 $G'[X', \overline{X}']$. Choose one such edge arbitrarily.

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

We claim that M'' is an induced matching in G'. Clearly, M' is a matching because M is one. Consequently, $M'' \subseteq M'$ is also a matching. We now show that M'' is also 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 col-our 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 1.29 we know that there is a $t \in T$ with cut $(X, \overline{X})$ such that we can find a matching M of size at least $\frac{\mathbf{tw}(G)}{3}$ in $G[X, \overline{X}]$. By the construction above the corresponding cut $(X', \overline{X}')$ yields an induced matching of size $\frac{\mathbf{tw}(G)}{6}$ in $G'[X', \overline{X}']$. This completes the proof. □

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

Lemma 4.4. *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 4.5. *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 4.4. We first transform every graph $G \in \mathcal{G}$ into a bipartite one G_1 by subdividing every edge, i.e., by introducing for each edge $e = uv$ a new vertex w_e and by replacing e by uw_e and w_ev. It is well-known that subdividing edges does not decrease the tree width of a graph (see e.g. [Die12]), 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 4.2 and the bound on the MIM-width follows by combining Lemma 4.4 and Lemma 4.3. □

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

Corollary 4.6. *There is a family of monotone β-acyclic CNF-formulas of* MIM*-width $\Omega(n)$ and* PS*-width $2^{\Omega(n)}$ where n is the size of the formula.*

Proof. Let $\mathcal{F}$ be the class of monotone CNF-formulas having the class $\mathcal{G}'$ of Corollary 4.5 as its incidence graphs. By Theorem 1.36 the formulas in $\mathcal{F}$ are β-acyclic. Combining the bound on the MIM-width of G' with Lemma 4.1 then directly yields the result. □

Since MIM-width is smaller than cliquewidth, the incomparability of β-acyclicity with cliquewidth or incidence tree width follows from Corollary 4.6. Observe that the incomparability of β-acyclicity and cliquewidth was already known from [GP04].

We conclude this section by observing that the family of formulas we construct to get the lower bound has large clauses. We show that it is indeed necessary.

Indeed, β-acyclic k-CNF have small primal tree width. This is even true for α-acyclic k-CNF formulas:

Theorem 4.7. *Let F be an α-acyclic k-CNF. Then its primal tree width is at most $k-1$.*

Proof. Let $(\mathcal{T}, \lambda)$ be a join tree of $\mathcal{H}(F)$. We see $\mathcal{T}$ as a tree decomposition where for every vertex t of $\mathcal{T}$, the bag in t is $\lambda(t)$. Each bag is thus of size at most k since each bag contains the variables of a k-clause of F. Since $\mathcal{T}$ is a join tree, for every $x \in \mathsf{var}(F)$, the set of vertices of $\mathcal{T}$ whose bag contains x is a connected subtree of $\mathcal{T}$. Moreover, let $e = \{x, y\}$ be an edge of $\mathcal{G}_{\mathsf{prim}}(F)$, then by definition, there exists a clause $C \in F$ such that $\{x, y\} \subseteq \mathsf{var}(C)$. Let t be the vertex of $\mathcal{T}$ which is labeled by $\mathsf{var}(C)$. It holds that the edge $\{x, y\}$ is covered by the bag labeling t. Thus $\mathcal{T}$ is a tree decomposition of $\mathcal{G}_{\mathsf{prim}}(F)$ and it is of tree width $k-1$. Thus, $\mathbf{tw}(\mathcal{G}_{\mathsf{prim}}(F)) \leq k-1$. □

Observe that if F is an α-acyclic k-CNF with a clause of size exactly k, then $\mathcal{G}_{\mathsf{prim}}(F)$ contains a k-clique, thus the primal tree width of F is exactly k. Theorem 4.7 shows that the most interesting α-acyclic instances for us will be of unbounded arity.

4.2 Structure of β-acyclic hypergraphs

Our compilation algorithm relies on a better understanding of the structure of β-acyclic hypergraphs. We describe the structure of the hypergraph induced by the removed vertices following a β-elimination order. We use these orders to define a family of subhypergraphs that will be helpful to describe a dynamic programming compilation algorithm.

4.2.1 Orders

Let $\mathcal{H}$ be a hypergraph and $<$ be an order on $V(\mathcal{H})$. We define an order on $\mathcal{H}$ induced by $<$, denoted by $<_{\mathcal{H}}$, as $e <_{\mathcal{H}} f$ if and only if $e \neq f$ and $\max_<(e \Delta f) \in f$ where $e \Delta f$ denotes the symmetric difference $(e \setminus f) \cup (f \setminus e)$ of e and f. Another way of seeing $<_{\mathcal{H}}$ is to see it as a lexicographical order. Indeed, assume that $V(\mathcal{H}) = \{x_1, \ldots, x_n\}$ with $x_i < x_j$ if and only if $i < j$. An edge e can be seen as a vector of $\vec{e} \in \{0,1\}^n$ such that $\vec{e}_i = 1$ if $x_i \in e$ and $\vec{e}_i = 0$ otherwise. Now $e <_{\mathcal{H}} f$ if and only if $\vec{e}$ is smaller than $\vec{f}$ for the lexicographical order. It follows:

Lemma 4.8. *If $<$ is an order on V then $<_{\mathcal{H}}$ is an order on $\mathcal{H}$.*

From the orders $<$ and $<_{\mathcal{H}}$, one can construct a family of subhypergraphs of $\mathcal{H}$ which will be interesting for us later. Let $x \in V$ and $e \in \mathcal{H}$. We denote by $V_{\leq x} = \{y \in V \mid y \leq x\}$, by $V_{<x} = \{y \in V \mid y < x\}$, by $V_{\geq x} = \{y \in V \mid y \geq x\}$ and $V_{>x} = \{y \in V \mid y > x\}$. We denote by $\mathcal{H}_e^x$ the subhypergraph of $\mathcal{H}$ that contains the edges $f \in \mathcal{H}$ such that there is a path from f to e that goes only through edges

smaller than e and vertices smaller than x. In particular, by definition, $\mathcal{H}_e^x$ is a connected subhypergraph of $\mathcal{H}$, with $e \in \mathcal{H}_e^x$ and for all $f \in \mathcal{H}_e^x$, $f \leq_{\mathcal{H}} e$. Observe also that even if there is a path from $f \in \mathcal{H}_e^x$ to e that goes only through vertices smaller than x, f may hold vertices that are bigger than x. We insist on the fact that the whole edge f is in $\mathcal{H}_e^x$ and not only its restriction to $V_{\leq x}$. Lemma 4.12 gives a precise characterization of the vertices of $\mathcal{H}_e^x$. The hypergraphs $\mathcal{H}_e^x$ are ordered as follows:

Lemma 4.9. *Let $x, y \in V(\mathcal{H})$ such that $x \leq y$ and $e, f \in \mathcal{H}$ such that $e \leq_{\mathcal{H}} f$ and $V(\mathcal{H}_e^x) \cap V(\mathcal{H}_f^y) \cap V_{\leq x} \neq \emptyset$. Then $\mathcal{H}_e^x \subseteq \mathcal{H}_f^y$. In particular, for all y, if $e \in \mathcal{H}_f^y$ then $\mathcal{H}_e^y \subseteq \mathcal{H}_f^y$.*

Proof. Let $z \in V(\mathcal{H}_e^x) \cap V(\mathcal{H}_f^y) \cap V_{\leq x}$, that is, there exist $g_1 \in \mathcal{H}_e^x$ and $g_2 \in \mathcal{H}_f^y$ such that $z \in g_1$ and $z \in g_2$. There exists a path $\mathcal{P}_1$ from f to g_2 going through vertices smaller than y and edges smaller than f and a path $\mathcal{P}_2$ from g_1 to e going through vertices smaller than x and edges smaller than e. Since $z \leq x \leq y$ and $e \leq_{\mathcal{H}} f$, $\mathcal{P} = (\mathcal{P}_1, z, \mathcal{P}_2)$ is a walk from f to e going through edges smaller than f and vertices smaller than y, that is $e \in \mathcal{H}_f^y$. Now let $h \in \mathcal{H}_e^x$ and let $\mathcal{P}_3$ be a path from e to h going through vertices smaller than x and edges smaller than e. Then $(\mathcal{P}, \mathcal{P}_3)$ is a walk from f to h going through vertices smaller than y and edges smaller than f. That is $h \in \mathcal{H}_f^y$, so $\mathcal{H}_e^x \subseteq \mathcal{H}_f^y$. □

4.2.2 Applications

If $\mathcal{H}$ is β-acyclic and $<$ is a β-elimination order then the hypergraphs $\mathcal{H}_e^x$ have interesting properties that we will use to describe the compilation algorithm in terms of dynamic programming and to analyze the runtime of weighted resolution on β-acyclic instances. *For the rest of this section, we assume that $\mathcal{H}$ is a β-acyclic hypergraph and that $<$ is a β-elimination order for $V(\mathcal{H})$.*

We start by giving an example that may help the reader to have an intuitive representation of what follows. We are interested in the β-acyclic hypergraph $\mathcal{H} = \{\{1,2\}, \{3,4\}, \{2,5\}, \{4,5\}, \{2,4,5\}\}$ depicted on Figure 4.2. One can easily check that $1 < 2 < 3 < 4 < 5$ is a β-elimination order and that the order $<_{\mathcal{H}}$ on $\mathcal{H}$ given by Lemma 4.8 is the following $e_1 = \{1,2\} <_{\mathcal{H}} e_2 = \{3,4\} <_{\mathcal{H}} e_3 = \{2,5\} <_{\mathcal{H}} e_4 = \{4,5\} <_{\mathcal{H}} e_5 = \{2,4,5\}$. $\mathcal{H}_{e_5}^4$ is the whole hypergraph since one can reach any edge from e_5 by going through vertices smaller than 4. $\mathcal{H}_{e_5}^3$ however is lacking the edge $e_2 = \{3,4\}$ since the only way of reaching e_2 from e_5 is to go through the vertex 4 which is not allowed.

We now prove useful lemmas to characterize the variables of $\mathcal{H}_e^x$ in terms of e and $V_{\geq x}$, the vertices left before eliminating x.

Lemma 4.10. *Let $e, f \in \mathcal{H}$ such that $e \cap f \neq \emptyset$ and let $x \in e \cap f$. If $e <_{\mathcal{H}} f$ then $e \cap V_{\geq x} \subseteq f$.*

Proof. Let $x_0 = \min(e \cap f)$. We show $e \cap V_{\geq x_0} \subseteq f$. Since $<$ is an elimination order, we already know that either $e \cap V_{\geq x_0} \subseteq f$ or $f \cap V_{\geq x_0} \subseteq e$. Let $y = \max(e \Delta f)$.

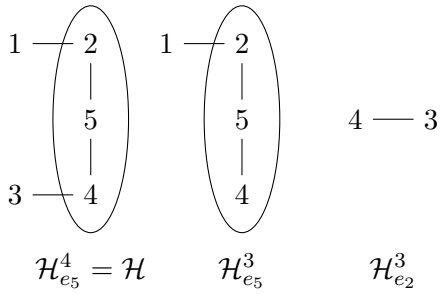

Figure 4.2: An example of $\mathcal{H}^x_e$

We have $y \in f \setminus e$ since $e <_{\mathcal{H}} f$. If $x_0 \leq y$, then $y \in f \cap V_{\geq x_0}$ and $y \notin e \cap V_{\geq x_0}$, thus we must have $e \cap V_{\geq x_0} \subseteq f$.

Now, if $y < x$, then $e \Delta f \subseteq V_{\leq y} \subseteq V_{<x_0}$. Thus $e \cap V_{\geq x_0} = (e \cap f) \cap V_{\geq x_0} = f \cap V_{\geq x_0}$. In particular, $e \cap V_{\geq x_0} \subseteq f$.

We have shown $e \cap V_{\geq x_0} \subseteq f$. Now observe that since $x_0 \leq x$, $e \cap V_{\geq x} \subseteq e \cap V_{\geq x_0}$. Thus $e \cap V_{\geq x} \subseteq f$. □

The next lemma is crucial since it introduces a normal form of the paths we consider. A path $\mathcal{P} = (e_0, x_0, \ldots, x_n, e_n)$ from e_0 to e_n is defined to be decreasing if for all $i < n$, $x_i > x_{i+1}$ and $e_i >_{\mathcal{H}} e_{i+1}$.

Lemma 4.11. *For every $x \in V$, $e \in \mathcal{H}$ and $f \in \mathcal{H}^x_e$, there exists a decreasing path from e to f going through vertices smaller than x.*

Proof. By definition of $\mathcal{H}^x_e$, there exists a path $\mathcal{P} = (e_0, x_0, \ldots, x_{n-1}, e_n)$ with $e_0 = e$ and $e_n = f$ and such that for all $i \leq n$, $e_i \leq_{\mathcal{H}} e$ and $x_i \leq x$. We show that if $\mathcal{P}$ is a shortest path among those going through vertices smaller than x, then it is also decreasing. Assume that $\mathcal{P}$ is a non-decreasing such shorter path. Remember that by definition of paths, the edges (e_i) are pairwise distinct. The same is true for the vertices (x_i). Moreover, observe that since $\mathcal{P}$ is a shortest path, then it holds that:

$$\forall k < n, j \notin \{k, k+1\} \Rightarrow x_k \notin e_j. \qquad (\star)$$

Indeed, if there exists k and $j \notin \{k, k+1\}$ such that $x_k \in e_j$, $\mathcal{P}$ could be shorten by going directly from e_i to e_j if $j > i+1$ or from e_j to e_{i+1} if $j < i$.

Let $i = \min\{j \mid x_{j+1} > x_j \text{ or } e_{j+1} >_{\mathcal{H}} e_j\}$ be the first indices where $\mathcal{P}$ does not respect the decreasing condition, which exists if $\mathcal{P}$ is not decreasing.

First assume $i = 0$. By definition of $\mathcal{P}$, $e_0 = e >_{\mathcal{H}} e_1$. Thus it holds that $x_0 < x_1$. By definition, $x_0 \in e_0 \cap e_1$ and by Lemma 4.10, $e_1 \cap V_{\geq x_0} \subseteq e_0$. Since $x_1 > x_0$, $x_1 \in e_1 \cap V_{\geq x_0}$, thus $x_1 \in e_0$ which contradicts $(\star)$.

Now assume $i > 0$. First, assume that $e_{i+1} >_{\mathcal{H}} e_i$. By definition of $\mathcal{P}$, it holds that $x_i \in e_i \cap e_{i+1}$ and then by Lemma 4.10, $e_i \cap V_{\geq x_i} \subseteq e_{i+1}$. Now observe that

by minimality of i, $x_{i-1} > x_i$. Since $x_{i-1} \in e_i$, $x_{i-1} \in e_i \cap V_{\geq x_i} \subseteq e_{i+1}$, which contradicts $(\star)$.

Otherwise, $e_i >_{\mathcal{H}} e_{i+1}$ and $x_{i+1} > x_i$. But then, by Lemma 4.10 again, $e_{i+1} \cap V_{\geq x_i} \subseteq e_i$. Since $x_{i+1} \in e_{i+1}$, it implies that $x_{i+1} \in e_{i+1} \cap V_{\geq x_i} \subseteq e_i$, which contradicts $(\star)$.

It follows that such i does not exist, that is, $\mathcal{P}$ is decreasing. □

The next lemma is characterizes the variables of $\mathcal{H}_e^x$. It will be crucial for the compilation algorithm since it is what makes the dynamic programming works.

Lemma 4.12. *For every $x \in V$ and $e \in \mathcal{H}$, $V(\mathcal{H}_e^x) \cap V_{\geq x} \subseteq e$.*

Proof. We show by induction on n that for any decreasing path $\mathcal{P} = (e_0, x_0, \ldots, e_n)$ from e_0 to e_n that $e_0 \supseteq e_n \cap V_{\geq x_0}$. If $n = 0$, then $e_n = e_0$ and the inclusion is obvious. Now, let $\mathcal{P} = (e_0, x_0, \ldots, e_n, x_n, e_{n+1})$. By induction, $e_0 \supseteq e_n \cap V_{\geq x_0}$ since $(e_0, x_0, \ldots, e_n)$ is a decreasing path from e_0 to e_n. Now by Lemma 4.10, since $x_n \in e_{n+1} \cap e_n$ and $e_{n+1} <_{\mathcal{H}} e_n$, we have $e_{n+1} \cap V_{\geq x_n} \subseteq e_n$. Since $x_0 > x_n$, $e_{n+1} \cap V_{\geq x_0} \subseteq e_{n+1} \cap V_{\geq x_n} \subseteq e_n$. Thus $e_{n+1} \cap V_{\geq x_0} \subseteq e_n \cap V_{\geq x_0} \subseteq e_0$ which concludes the induction.

Now let $e \in \mathcal{H}$, $x \in V(\mathcal{H})$ and $f \in \mathcal{H}_e^x$. By Lemma 4.11, there exists a decreasing path from e to f going through vertices smaller than x. From what precedes, $f \cap V_{\geq x} \subseteq e$. That is $V(\mathcal{H}_e^x) \cap V_{\geq x} \subseteq e$. □

The following lemma states that $\mathcal{H}_e^x$ is the same as $\mathcal{H}_e^{x'}$ if $x' \notin e$ where x' is the successor of x in the β-elimination order.

Lemma 4.13. *Let $e \in \mathcal{H}$, $x, x' \in V(\mathcal{H})$ such that x' is the successor of x in the β-elimination order of $\mathcal{H}$. If $x' \notin e$, then $\mathcal{H}_e^x = \mathcal{H}_e^{x'}$.*

Proof. Let $f \in \mathcal{H}_e^{x'}$. By Lemma 4.11, there exists a decreasing path from e to f. Since $x' \notin e$, the first vertex of this path is less than x' and different from x', that is lesser than x. Thus $f \in \mathcal{H}_e^x$. The other inclusion follows from Lemma 4.9. □

Finally, we show how $\mathcal{H}_e^x$ can be decomposed into smaller hypergraph:

Lemma 4.14. *Let $e \in \mathcal{H}$, $x \in V(\mathcal{H})$. There exists $U \subseteq \mathcal{H}_e^x \setminus \{e\}$ such that:*

- $\mathcal{H}_e^x \setminus \{e\} = \biguplus_{f \in U} \mathcal{H}_f^x$ *and,*
- *for all $f, f' \in U$, if $f \neq f'$, $V(\mathcal{H}_f^x) \cap V(\mathcal{H}_{f'}^x) \subseteq V_{>x}$.*

Proof. Let C be a connected component of $\mathcal{H}_e^x[V_{\leq x}]$. We identify each edge of C with the corresponding edge of $\mathcal{H}_e^x$. Let f be the biggest edge of C for $<_{\mathcal{H}}$. We claim that $C = \mathcal{H}_f^x$. Indeed, if $g \in C$ then there exists a path from f to g that goes through vertices smaller than x and through edges in C since they are in the same connected component of $\mathcal{H}_e^x[V_{\leq x}]$. Since f is maximal, this path goes through edges smaller than f too, that is, $g \in \mathcal{H}_f^x$. Now by Lemma 4.9, since $f \leq_{\mathcal{H}} e$, we have $\mathcal{H}_f^x \subseteq \mathcal{H}_e^x$. Thus, $\mathcal{H}_f^x \subseteq C$ since $\mathcal{H}_f^x$ is connected in $\mathcal{H}[V_{\leq x}]$.

Now it is enough to choose U to be the set of the maximal edges of the connected components of $\mathcal{H}_e^x[V_{\geq x}]$ to get the desired results. □

4.3 The compilation algorithm

In this section, we use results from Section 4.2 to describe the dynamic programming algorithm used to compile β-acyclic formulas.

4.3.1 Compilation to dec-DNNF

Given a CNF-formula F with hypergraph $\mathcal{H}$, we can naturally define a family of subformulas F_e^x from $\mathcal{H}_e^x$ as the conjunction of clauses corresponding to the edges in $\mathcal{H}_e^x$, that is $F_e^x = \{C \in F \mid \mathsf{var}(C) \leq_{\mathcal{H}} e\}$ for $e \in \mathcal{H}(F)$. Lemma 4.12 implies in particular that $\mathsf{var}(F_e^x) \subseteq (e \cup V_{<x})$. Thus, if τ is an assignment of variables $(e \cap V_{>x})$, then $F_e^x[\tau]$ has all its variables in $V_{\leq x}$. We will be particularly interested in such assignments: for a clause $C \in F$, denote by τ_C the only assignment of $\mathsf{var}(C)$ such that $\tau_C \not\models C$ and by $\tau_C^x = \tau_C|_{V_{>x}}$. We compute a dec-DNNF D by dynamic programming such that for each clause C with $\mathsf{var}(C) = e$ and variable $x \in V$, there exists a gate in D computing $F_e^x[\tau_C^x]$, which is a formula with variables in $V_{\leq x}$. Lemma 4.15 and Corollary 4.16 describe everything needed for the dynamic programming algorithm by expressing F_e^x as a decomposable conjunction of precomputed values.

Lemma 4.15. *Let $x \in \mathsf{var}(F)$ such that $x \neq \min(\mathsf{var}(F))$ and let $y \in \mathsf{var}(F)$ be the predecessor of x for order $<$. Let $e \in \mathcal{H}(F)$ and $\tau : (e \cap V_{\geq x}) \to \{0,1\}$.*

Then either $F_e^x[\tau] \equiv 1$ or there exists $U \subseteq \mathcal{H}_e^x$ and for all $g \in U$ a clause $C(g) \in F_e^x$ with $\mathsf{var}(C(g)) = g$ such that

$$F_e^x[\tau] \equiv \bigwedge_{g \in U} F_g^y[\tau_{C(g)}^y].$$

Moreover, this conjunction is decomposable.

Proof. Assume first that for all $C \in F_e^x$, $\tau \models C$. Thus $F_e^x[\tau] \equiv 1$ since every clause of F_e^x is satisfied.

Now assume that there exists $C \in F_e^x$ is such that $\tau \not\models C$. This means that $\tau \simeq \tau_C$.

We let $A = \{\mathsf{var}(C) \mid C \in F_e^x \text{ and } \tau \not\models C\} \neq \emptyset$ by assumption. Observe that

$$F_e^x[\tau] \equiv \bigwedge_{\substack{C \in F_e^x \\ \mathsf{var}(C) \in A}} C[\tau]$$

since for every $C \in F_e^x$, if $\mathsf{var}(C) \notin A$, $\tau \models C$ by construction of A.

Let $U = \{g \in A \mid \forall f \in A \setminus \{g\}, g \notin \mathcal{H}_f^y\}$. For each $g \in U$, we choose an arbitrary clause $C(g)$ such that $\mathsf{var}(C(g)) = g$ and $\tau \not\models C(g)$. Such a clause exists since $U \subseteq A$. We claim that U meets the conditions given in the statement of the lemma.

First, let $f \in A$. We show that there exists $g \in U$ such that $f \in \mathcal{H}_g^y$. If $f \in U$, then we are done since $f \in \mathcal{H}_f^y$. Now assume that $f \notin U$. By definition of U,

$B = \{g \in A \setminus \{g\} \mid f \in \mathcal{H}^y_g\} \neq \emptyset$. We choose g to be the maximum of B for $\leq_{\mathcal{H}}$. We claim that $g \in U$. Indeed, assume there exists $g' \in A$ such that $g \in \mathcal{H}^y_{g'}$ and $g < g'$. By Lemma 4.9, $\mathcal{H}^y_g \subseteq \mathcal{H}^y_{g'}$ and since $f \in \mathcal{H}^y_g$, we also have $f \in \mathcal{H}^y_{g'}$, that is, $g' \in B$. Yet, $g = \max(B)$ and $g \leq g'$, that is, $g = g'$. Thus $g \in U$.

We proved that for all $f \in A$, there exists $g \in U$ such that $f \in \mathcal{H}^y_g$. Thus if C is a clause of F^x_e, either $\mathsf{var}(C) \notin A$ and then $\tau \models C$ by definition of A, or $\mathsf{var}(C) \in A$, then there exists $g \in U$ such that $\mathsf{var}(C) \in \mathcal{H}^y_g$, that is, $C \in F^y_g$. Thus

$$F^x_e[\tau] \equiv \bigwedge_{g \in U} F^y_g[\tau].$$

Let $g \in U$. We show that $\tau|_{\mathsf{var}(F^y_g)} = \tau^y_{C(g)}$. Observe that by Lemma 4.12, $\mathsf{var}(F^y_g) \cap V_{\geq x} = V(\mathcal{H}^y_g) \cap V_{\geq x} \subseteq g \cap V_{\geq x}$. Since τ assigns variables from $e \cap V_{\geq x}$:

$$\begin{aligned}\tau|_{\mathsf{var}(F^y_g)} &= \tau|_{\mathsf{var}(F^y_g) \cap V_{\geq x} \cap e} \\ &= \tau|_{g \cap V_{\geq x} \cap e}\end{aligned}$$

Moreover, since $g \in \mathcal{H}^x_e$, by Lemma 4.12 again, $g \cap V_{\geq x} \subseteq e \cap V_{\geq x}$. Thus $g \cap V_{\geq x} \cap e = g \cap V_{\geq x}$. In other words,

$$\tau|_{\mathsf{var}(F^y_g)} = \tau|_{g \cap V_{\geq x}}.$$

Since τ assigns every variables of $e \cap V_{\geq x}$ by assumption, $\tau|_{g \cap V_{\geq x}}$ assigns every variables of $g \cap V_{\geq x}$. Finally, since $\tau \not\models C(g)$, we have $\tau \simeq \tau^y_{C(g)}$. Since by definition $\mathsf{var}(C(g)) = g$, it follows that:

$$\tau|_{\mathsf{var}(F^y_g)} = \tau^y_{C(g)}.$$

So far, we have thus proven

$$F^x_e[\tau] \equiv \bigwedge_{g \in U} F^y_g[\tau^y_{C(g)}].$$

It remains to show that this conjunction is decomposable, that is, for all $g_1, g_2 \in U$, $\mathsf{var}(F^y_{g_1}[\tau^y_{C(g_1)}]) \cap \mathsf{var}(F^y_{g_2}[\tau^y_{C(g_2)}]) = \emptyset$. Let $g_1, g_2 \in U$ with $g_1 <_{\mathcal{H}} g_2$ and assume there exists $z \in \mathsf{var}(F^y_{g_1}[\tau^y_{C(g_1)}]) \cap \mathsf{var}(F^y_{g_2}[\tau^y_{C(g_2)}])$, that is, $z \in \mathsf{var}(F^y_{g_1}) \cap \mathsf{var}(F^y_{g_2}) \cap V_{\leq y}$ since from what precedes, τ assigns every variable of $F^y_{g_1}$ greater than x. By Lemma 4.9, we have $F^y_{g_1} \subseteq F^y_{g_2}$, which contradicts the fact that $g_1 \in U$. □

Corollary 4.16. *Let $x \in \mathsf{var}(F)$ such that $x \neq \min(\mathsf{var}(F))$ and let $y \in \mathsf{var}(F)$ be the predecessor of x for order $<$. For every $C \in \mathcal{H}(F)$, there exist $U_0, U_1 \subseteq \mathcal{H}^x_{\mathsf{var}(C)}$ and for all $g \in U_0 \cup U_1$ a clause $C(g) \in F^x_{\mathsf{var}(C)}$ with $\mathsf{var}(C(g)) = g$ such that*

$$F^x_{\mathsf{var}(C)}[\tau^x_C] \equiv (x \wedge \bigwedge_{g \in U_1} F^y_g[\tau^y_{C(g)}]) \vee (\neg x \wedge \bigwedge_{g \in U_0} F^y_g[\tau^y_{C(g)}]).$$

Moreover, all conjunctions are decomposable.

Proof. Let $\tau_1 = \tau_C^x \cup \{x \mapsto 1\}$ and $\tau_0 = \tau_C^x \cup \{x \mapsto 0\}$. We observe that

$$F^x_{\mathsf{var}(C)}[\tau_C^x] = (x \wedge F^x_{\mathsf{var}(C)}[\tau_1]) \vee (\neg x \wedge F^x_{\mathsf{var}(C)}[\tau_0]).$$

Clearly, $x \notin \mathsf{var}(F^x_{\mathsf{var}(C)}[\tau_1])$ and $x \notin \mathsf{var}(F^x_{\mathsf{var}(C)}[\tau_0])$, thus, both conjunctions are decomposable. Now, applying Lemma 4.15 on $F^x_{\mathsf{var}(C)}[\tau_0]$ and on $F^x_{\mathsf{var}(C)}[\tau_1]$ yields the desired decomposition. □

Theorem 4.17. *Let F be a β-acyclic* CNF*-formula. There exists a* dec-DNNF *D of size $O(\mathsf{size}(F))$ and fanin at most $|\mathcal{H}|$ computing F.*

Proof. Let $\mathcal{H}$ be the hypergraph of F and $<$ a β-elimination order. Let $\mathsf{var}(F) = \{x_1, \ldots, x_n\}$ where $x_i < x_j$ if and only if $i < j$. We construct by induction on i a dec-DNNF D_i of fanin $|\mathcal{H}|$ at most such that for each $e \in \mathcal{H}$, $C \in F$ such that $\mathsf{var}(C) = e$ and $j \leq i$, there exists a gate in D_i computing $F_e^{x_j}[\tau_C^{x_j}]$ and $|D_i| \leq 7 \cdot (\sum_{j=1}^{i} c(x_i))$ where $c(x_i)$ is the number of clauses in F holding x_i.

We start by explaining how D_1 is constructed. Let $e \in \mathcal{H}$. If $x_1 \notin e$, then $F_e^{x_1}$ contains only the clauses C such that $e = \mathsf{var}(C)$. For such a C, $\tau_C^{x_1} = \tau_C$, thus $F_e^{x_1}[\tau_C] = 0$. Now, if $x_1 \in e$, $F_e^{x_1}$ contains only clauses D such that $x_1 \in \mathsf{var}(D) \subseteq e$ since x_1 is nested in $\mathcal{H}$. Let C be a clause such that $\mathsf{var}(C) = e$. For every $D \in F_e^{x_1}$, $\mathsf{var}(D) \subseteq \mathsf{var}(C)$, thus $F_e^{x_1}[\tau_C^{x_1}]$ has only one variable: x_1. Thus $F_e^{x_1}[\tau_C^{x_1}]$ is equivalent to either x_1, $\neg x_1$ or 0. We thus define D_1 to be the dec-DNNF with at most three input gates $x_1, \neg x_1$ and 0. We have $|D_1| \leq 7 \cdot c(x_1)$.

Now let assume D_i is constructed. To ease notations, let $x = x_{i+1}$. Let $e \in \mathcal{H}$ and C be a clause such that $\mathsf{var}(C) = e$. We want to add a gate in D_i that will compute $F_e^x[\tau_C^x]$. If $x \notin e$, then $\mathcal{H}_e^x = \mathcal{H}_e^{x_i}$ since by Lemma 4.12, $\mathsf{var}(\mathcal{H}_e^{x_i}) \subseteq (e \cup V_{<x_i})$. Thus $F_e^x = F_e^{x_i}$ and $\tau_C^x = \tau_C^{x_i}$. Therefore, there is already a gate computing $F_e^x[\tau_C^x]$ in D_i.

Assume now that $x \in e$. By Corollary 4.16, we can compute $F^x_{\mathsf{var}(C)}[\tau_C^x]$ for every C with $\mathsf{var}(C) = e$ by adding at most one decision-gate and a fanin $|\mathcal{H}|$ decomposable and-gate to D_i since for every values appearing in statement of Corollary 4.16 there exists a gate in D_i computing it. That is, we add to D_i at most 7 gates to compute $F^x_{\mathsf{var}(C)}[\tau_C^x]$. We have to do this for each $C \in F$ such that $x \in \mathsf{var}(C)$. We thus add at most $7c(x)$ gates in D_i. Thus $|D_{i+1}| \leq 7 \cdot \sum_{j \leq i+1} c(x_j)$.

To conclude, assume that $\mathcal{H}$ is connected and let $e = \max(\mathcal{H})$. We have $\mathcal{H}_e^{x_n} = \mathcal{H}$ since there is a path from e to every other edge in $\mathcal{H}$. Thus $F_e^{x_n} = F$. Let C be a clause with $\mathsf{var}(C) = e$. $\tau_C^{x_n}$ is the empty assignment, thus $F_e^{x_n}[\tau_C^{x_n}] \equiv F$. Hence, there is a gate in D_n that computes F and D_n is of size at most $7 \cdot \mathsf{size}(F)$ and it is of fanin $|\mathcal{H}|$ at most.

If $\mathcal{H}$ is not connected, then each connected component of $\mathcal{H}$ is β-acyclic, thus we can compile them independently and take the decomposable conjunction of these dec-DNNF. □

Proposition 4.18. *Algorithm 5 runs in polynomial time and is correct.*

Algorithm 5: An algorithm to compile β-acyclic formulas into dec-DNNF

Data: A β-acyclic CNF-formula F
begin
  Compute $x_1 < \cdots < x_n$ a β-elimination order for $\mathcal{H}(F)$;
  Compute $<_{\mathcal{H}}$ from $<$;
  Compute F_e^x for every $x \in \mathsf{var}(F)$, $e \in \mathcal{H}(F)$;
  Construct D the DNNF with three input gates $x_1, \neg x_1, 0$;
  for $C \in F$ **do**
    $M[C][x_1] \leftarrow$ the gate of D corresponding to $F^{x_1}_{\mathsf{var}(C)}[\tau_C^{x_1}]$;
  for $i = 2 \dots n$ **do**
    for $E \in F$ *such that* $x_i \in E$ **do**
      $e \leftarrow \mathsf{var}(E)$;
      $\tau_0 \leftarrow \tau_C^{x_i} \cup \{x_i \mapsto 0\}$;
      Let $A_0 \leftarrow \{(\mathsf{var}(C), C) \mid C \in F_e^{x_i}, \tau_0 \not\models C\}$;
      Let $U_0 \leftarrow \{(g, C) \in A_0 \mid \forall f \in A \setminus \{g\}, g \notin \mathcal{H}_f^{x_{i-1}}\}$;
      Add an $\wedge$-gate α_0 in D ;
      Connect $\alpha_0$to a new input labeled with x_i ;
      For every $(g, C) \in U_0$, connect α_0 with the gate $M[C][x_{i-1}]$;
      $\tau_1 \leftarrow \tau_C^{x_i} \cup \{x_i \mapsto 1\}$;
      Let $A_1 \leftarrow \{(\mathsf{var}(C), C) \mid C \in F_e^{x_i}, \tau_1 \not\models C\}$;
      Let $U_1 \leftarrow \{(g, C) \in A_1 \mid \forall f \in A \setminus \{g\}, g \notin \mathcal{H}_f^{x_{i-1}}\}$;
      Add an $\wedge$-gate α_1 in D ;
      Connect α_1 to a new input labeled with $\neg x_i$;
      For every $(g, C) \in U_1$, connect α_1 with the gate $M[C][x_{i-1}]$;
      Add an $\vee$-gate α in D ;
      Connect α to α_0 and α_1 ;
      $M[E][x_i] \leftarrow \alpha$;
  $C_m \leftarrow$ a clause of $\mathcal{H}(F)$ such that $\mathsf{var}(C_m)$ is maximal for $<_{\mathcal{H}}$;
  $\mathsf{output}(D) \leftarrow M[C_m][x_n]$;
  return D ;

Proof. The precomputation of the orders can be done in polynomial time. Moreover, one can precompute F_e^x for every x, e as well since it boils down to compute the connected component of e in the hypergraph $\mathcal{H}(F)$ where only edges smaller than e and vertices smaller than x are kept.

Finally, the body of the loops is executed at most $\|\mathcal{H}\|$ times and one can compute the sets A_0, U_0, A_1, U_1 in polynomial time by simply trying every $C \in F_e^{x_i}$.

For correction, observe that $M[C][x]$ is always a gate of D that computed $F^x_{\mathsf{var}(C)}(\tau^x_C)$ since the sets A_i, U_i are constructed as in Lemma 4.15 and we add the decision gates given by Corollary 4.16 at each step. □

Proposition 4.18 give an effective version of Theorem 4.17:

Theorem 4.19. *Let F be a β-acyclic* CNF*-formula. One can compute in polynomial time a* dec-DNNF *D of size $O(\mathsf{size}(F))$ computing F.*

4.3.2 Corollaries

We use Theorem 4.19 to settle the complexity of #SAT and other related problems on β-acyclic CNF-formulas.

Theorem 4.20. #SAT *can be solved in polynomial time on β-acyclic formulas.*

Proof. Given a β-acyclic formula F, compile it into a polynomial size dec-DNNF D in polynomial time using Theorem 4.19 and count the number of satisfying assignments of D in polynomial time using Proposition 1.57. □

More generally, every tractable queries for dec-DNNF may be done in polynomial time for β-acyclic formulas. In particular, one can enumerate its satisfying assignments with a small delay:

Theorem 4.21. *Given a β-acyclic formula F with n variables, one can enumerate $\mathsf{sat}(F)$ with delay $O(n \cdot \mathsf{size}(F))$.*

In Section 3.2.2, we have shown how to use the compilation algorithm for bounded PS-width formulas to solve MaxSAT. The same technique can be used for β-acyclic formulas. Indeed, recall (see Section 3.2.2) that for a CNF-formula F, we define $\hat{F} = \{C \cup \{y_C\} \mid C \in F\}$ where the variables y_C are fresh. The following holds:

Lemma 4.22. *If F is a β-acyclic formula then $\hat{F}$ is β-acyclic.*

Proof. Observe that for every $C \in \hat{F}$, the only clause having the variable y_C is C itself. Thus y_C is β-leaf of $\mathcal{H}(\hat{F})$. If we remove every y_C from $\mathcal{H}(\hat{F})$, we end up with $\mathcal{H}(F)$ which is β-acyclic. Thus $\mathcal{H}(\hat{F})$ is also β-acyclic. □

Since $\mathsf{size}(\hat{F}) = m + \mathsf{size}(F)$ where m is the number of clauses of F, we have the following:

Corollary 4.23. *If F is a β-acyclic formula with m clauses then one can compute in polynomial time a* dec-DNNF *of size $O(m + \mathsf{size}(F))$ computing $\hat{F}$.*

Remember that we have shown in Lemma 3.14 how minimal satisfying assignments of $\hat{F}$ can be used to solve MaxSAT on F and the minimal satisfying assignments could be found quickly for DNNF by Proposition 3.15. Thus Corollary 4.23 together with Proposition 3.15 and Lemma 3.14 gives:

Theorem 4.24. MaxSAT *can be solved in polynomial time on β-acyclic formulas.*

4.4 Conclusion

We have given a compilation algorithm that can transform any β-acyclic formulas into a succinct dec-DNNF in polynomial time. The fact that this construction can be done in polynomial time allows to settle the complexity of many problems concerning β-acyclic CNF-formulas such as model counting or enumeration. By using the same technique of Chapter 3, we are able to transform the formula in order to use the compilation algorithm to solve optimization problems such as MaxSAT.

Observe however that contrary to the compilation algorithm of Theorem 3.9 concerning bounded PS-width formulas, the compilation algorithm for β-acyclic formulas does not necessarily construct a *structured* DNNF. In Theorem 3.9, the very nature of the characterization of PS-width leads to this structuredness since branch decompositions induce vtrees. However, all attempts so far at characterizing β-acyclicity in terms of branch decomposition have failed. Showing that β-acyclic formulas does not have small structured DNNF may be a way to understand why such characterizations are hard to find.

Open question 2. *Can β-acyclic formulas be compiled into polynomial size* **structured** *(deterministic)* DNNF*?*

Moreover, the compilation of β-acyclic formulas relies on a dynamic programming algorithm. The way we have defined the needed subformulas may possibly be improved in order to construct small FBDD.

Open question 3. *Can β-acyclic formulas be compiled into polynomial size* FBDD*?*

In the next chapter, we present another algorithm for solving #SAT and related problems on β-acyclic formulas and that can also solve counting problems on instances more general than CNF-formulas.

Chapter 5

Weighted DP-resolution

In this chapter, we introduce a natural generalization of DP-resolution that operates on weighted constraints. Resolution is a well-studied proof system that can be used to describe a proof of unsatisfiability. It was introduced in a paper by Robinson [Rob65] but Davis and Putnam [DP60] already introduced an algorithmic version of it which is now usually called Davis-Putnam resolution, DP-resolution for short. In this chapter, we will essentially be interested in the DP-resolution algorithm. DP-resolution is a syntactic rule that allows one to eliminate a variable in a formula by merging clauses without changing its satisfiability. By iteratively applying this process, one either gets a proof of unsatisfiability or a satisfying assignment. However, even if the satisfiability is preserved, the set of satisfying assignments of the formula changes in a uncontrollable way which makes this technique – as it is – unsuited for model counting. By lifting it to weighted constraints, we are able to preserve more information after the elimination of a variable which enables model counting.

Like resolution, our algorithm in general leads to a blow-up of the formula size since we possibly introduce an exponential number of clauses at each step. However, a careful analysis allows us to define a parameter of hypergraph, the *cover-width*, such that if the hypergraph of a formula is of bounded cover-width, then the size of the formula remains bounded during the execution of the algorithm. In particular, cover-width generalizes β-acyclicity and we can show that weighted resolution runs in polynomial time on such formulas. The tractability of SAT of such formulas was already known to be tractable [OPS13] since Ordyniak, Paulusma and Szeider shows that DP-resolution may be down in polynomial time on such instances. Our algorithm generalizes the one of [OPS13] and provides another proof that model counting on β-acyclic instances is tractable, that is more efficient and simpler to implement than the compilation approach of Chapter 4. Moreover, we actually describe our algorithm in a framework that is more general than CNF-formulas. We are thus able to show that not only #SAT is tractable on β-acyclic instances, but also more general problems on arbitrary domains.

The first part of this chapter is dedicated to the DP-resolution algorithm. We

first present the algorithm and show how structural restrictions of CNF-formulas may lead to polynomial time execution of DP-resolution. In a second section, we start by introducing the generalization of CNF formulas we will use in the algorithm and show how we can encode #SAT in this model. Our model is a generalization of different encoding used in the theory of CSPs. We then present a first version of DP-resolution specific to the case of β-acyclic instances where it can be presented in an lighter way than the general case. A quick analysis shows that this algorithm runs with a polynomial number of arithmetic operations and that the formula do not grow during the elimination process. However, it is not sufficient to guarantee a polynomial runtime since the size of the numbers we compute during the execution of the algorithm. This analysis is more challenging and we have to use structural results from Chapter 4 to show that the algorithm is actually in polynomial time. The third part is dedicated to a presentation of weighted DP-resolution. We start by describing the algorithm. We then introduce a new width, the *cover-width*. We show that weighted DP-resolution is tractable on bounded cover-width instances and we study how cover-width relates with other measures. We show that cover-width generalizes β-acyclicity and that is greater than β-hypertree width. Finally, we argue that cover-width is a natural generalization of tree width to hypergraph since both notions collapses when restricted to graphs.

5.1 DP-Resolution

In this section, we recall the resolution rule and how one can use it to solve SAT. We then study some families of formulas where we can exploit structural properties to ensure resolution to run in polynomial time.

5.1.1 A well-known algorithm for SAT

Let x be a variable, $C = \{x\} \cup C'$ and $D = \{\neg x\} \cup D'$ be two clauses. The clause $C' \cup D'$ is called the *resolvent* of C and D on variable x. Resolution comes from the simple observation that the satisfiability of $C \wedge D$ is equivalent to the satisfiability of their resolvent on x. An intuitive way of seeing this is to interpret $C \wedge D$ as an if-then-else structure: if x is true, then one must satisfy D', else one must satisfy C'. Thus $C' \cup D'$ has to be satisfied. Reciprocally, if $C' \cup D'$ is satisfied, then one of them has to be satisfied and one can choose the appropriate value of x to satisfy $C \wedge D$. This transformation can be done on a CNF-formula by resolving the clauses pairwise. For a CNF-formula F and a variable $x \in \mathsf{var}(F)$, let F_x be the set of clauses of F that contain the literal x, let $F_{\neg x}$ be the set of clauses of F that contain the literal $\neg x$ and let $G = F \setminus (F_x \cup F_{\neg x})$, the set of clauses that do not contain the variable x. We denote by $\mathsf{res}(F, x) = G \cup \{(C \setminus \{x\}) \cup (D \setminus \{\neg x\}) \mid C \in F_x, D \in F_{\neg x}\}$. The satisfiability of F is equivalent to the satisfiability of $\mathsf{res}(F, x)$ as stated in the following lemma:

Lemma 5.1. *Let F be a* CNF*-formula and $x \in \mathsf{var}(F)$, then F is satisfiable if and only if* $\mathsf{res}(F, x)$ *is satisfiable.*

Proof. We keep the notations of the previous paragraph, that is, F_x is the set of clauses of F that contain the literal x, $F_{\neg x}$ is the set of clauses of F that contain the literal $\neg x$ and let $G = F \setminus (F_x \cup F_{\neg x})$ is the set of clauses that do not contain the variable x.

Assume F is satisfiable. Let τ be a satisfying assignment of F. In particular, τ satisfies G. If $\tau(x) = 1$, then for every $D \in F_{\neg x}$, τ satisfies $D \setminus \{\neg x\}$. Hence for every $C \in F_x$, τ satisfies $(C \setminus \{x\}) \cup (D \setminus \{\neg x\})$. Thus τ satisfies $\mathsf{res}(F, x)$. If $\tau(x) = 0$, then for every $C \in F_x$, τ satisfies $C \setminus \{x\}$ and by a symmetrical reasoning, τ satisfies $\mathsf{res}(F, x)$. Thus, $\mathsf{res}(F, x)$ is also satisfiable.

Assume now that $\mathsf{res}(F, x)$ is satisfiable and let τ be a satisfying assignment of $\mathsf{res}(F, x)$. Observe that since $x \notin \mathsf{var}(\mathsf{res}(F, x))$, τ do not assign any value to x. Assume that there exists $D_0 \in F_{\neg x}$ such that τ does not satisfies D_0. For every $C \in F_x$, since τ satisfies $(C \setminus \{x\}) \cup (D_0 \setminus \{\neg x\})$, τ has to satisfy $C \setminus \{x\}$. Thus, $\tau' = \tau \cup \{x \mapsto 0\}$ satisfies every $C \in F_x$ and τ' also satisfies every $D \in F_{\neg x}$ since $\neg x \in D$. Since τ also satisfies G, τ' satisfies F, that is, F is satisfiable. Now, if there is no such D_0, it means that τ satisfies every $D \in F_{\neg x}$. A similar reasoning shows that $\tau' = \tau \cup \{x \mapsto 1\}$ satisfies F, that is, F is satisfiable. □

Since at each step we remove a variable from the formula F, this directly transforms to an algorithm for SAT, introduced first by Davis and Putnam, known as DP-resolution [DP60] : iteratively pick a variable, resolve on it and remove tautological clauses. If the empty clause appears, it is because we are resolving on a variable x such that clauses $\{x\}$ and $\{\neg x\}$ appear in the formula. Thus the formula is unsatisfiable. Otherwise, we get the empty formula in the end, then the formula is satisfiable. Algorithm 6 present this algorithm and Figure 5.1 gives an example of a formula where the resolution algorithm is used to derive the empty clause. If a formula is found to be unsatisfiable by only applying the resolution rule, then it also provides a proof of this fact that can be checked by an external prover.

Resolution is however not an efficient algorithm. Each step may square the size of F leading to an exponential blow-up when repeated. The order in which the variables are eliminated has a great impact on the runtime of the algorithm. One can still ask whether it is always possible to find (not necessarily in polynomial time) an order for which resolution works in polynomial time. It is unlikely however since it would have a surprising consequence in complexity theory, since it would imply NP = coNP. Even unconditionally, one can find formulas for which every proof of unsatisfiability using resolution is of exponential length, that is, whatever the order we choose to eliminate the variables, we will have an exponential number of clauses at some point. One of the most famous example is the "Pigeon Hole Principle" $\mathsf{PHP}_{n,m}$. The formula $\mathsf{PHP}_{n,m}$ states that the fact that we have n pigeons, m holes and each pigeon is in one hole, alone. Clearly, if $m < n$,

Algorithm 6: DP-Resolution solves SAT using resolution

Data: A CNF-formula F and an order $x_1, \ldots, x_n$ on $\mathsf{var}(F)$
begin
  if *F contains the empty clause* **then**
    return False
  else
    if $n > 0$ **then**
      $F' \leftarrow \mathsf{res}(F, x_1)$;
      Remove tautological clauses from F' ;
      return DP-Resolution$(F', \{x_2, \ldots, x_n\})$
    else
      return True

$$
\begin{aligned}
F &= (x \vee y) \wedge (\neg x \vee z \vee y) \wedge (\neg x \vee \neg z) \wedge (\neg y \vee z) \\
F_1 &= \mathsf{res}(F, x) \\
&= (z \vee y) \wedge (y \vee \neg z) \wedge (\neg y \vee z) \\
F_2 &= \mathsf{res}(F_1, z) \\
&= y \wedge \neg y \\
F_3 &= \mathsf{res}(F_2, y) \\
&= \{\emptyset\}
\end{aligned}
$$

Figure 5.1: An example of resolution

then the formula is unsatisfiable. Formally, $\mathsf{PHP}_{n,m}$ has nm variables $p_{i,j}$ for $i \leq n$ and $j \leq m$. The intended meaning of variable $p_{i,j}$ is that it is true if pigeon i stands in hole j. $\mathsf{PHP}_{n,m}$ contains clauses C_i for each $i \leq n$ that states that each pigeon is in at least one hole: $C_i = \bigvee_{j=1}^{m} p_{i,j}$. It also contains clauses $D_{i,j,j'}$ for each $i \leq n$, $j < j' \leq m$ that states that if pigeon i is in hole j then it is not in hole j': $D_{i,j,j'} = p_{i,j} \rightarrow \neg p_{i,j'} = \neg p_{i,j} \vee \neg p_{i,j'}$. Haken [Hak85] proved that whatever order we choose on variables to refute $\mathsf{PHP}_{n+1,n}$, (since $n+1$ pigeons cannot stand in n boxes) we will at some point have a formula with $2^{\Omega(n)}$ clauses, that is $2^{\Omega(\sqrt{N})}$ where $N = n(n+1)$ is the number of variables of $\mathsf{PHP}_{n,n+1}$. Urquhart [Urq87] presented a strong exponential lower bound, that is a CNF formula F with a $2^{\Omega(\mathsf{size}(F))}$ lower bound on the number of clauses needed for the refutation of a family of formulas built on expander graphs. The proof was later simplified by Schöning [Sch97] and it was shown by Sasson and Wigderson [BSW99] that the number of clauses needed for refuting a formula is strongly related to the size of the clauses needed in the proof.

In some cases however, we can ensure that resolution works in polynomial time if the right order for the variables is picked. We present examples where resolution does not blow-up the size of the formula that will be used afterwards to compare with the case of weighted DP-resolution: 2-SAT, bounded primal tree width and β-acyclic formulas.

5.1.2 Resolution on 2-CNF

The case for 2-SAT is easy to deal with and is known since the sixties [Kro67]. It is sufficient to observe that the resolution of two clauses of size 2 is also a clause of size 2. Thus if F is a 2-CNF and $x \in \mathsf{var}(F)$, $\mathsf{res}(F, x)$ is also a 2-CNF. Since the number of clauses of size at most 2 on n variables is bounded by $4n^2$, we know that during the algorithm, there will be no exponential blow-up. Since the number of operations needed for a resolution step is roughly the number of clauses of the formula, we have the following:

Theorem 5.2. *Let F be a 2-CNF. Whatever order of variables we choose, Algorithm 6 runs in time $O(|\mathsf{var}(F)|^3)$.*

This algorithm is presented here only as an example on how we can find tractable classes for resolution: its runtime is indeed not as good as the well-known linear time algorithm of Aspvall, Plass and Tarjan [APT79] based on the search for strong connected component in the implication graph of the formula.

5.1.3 Resolution on bounded primal tree width

The fact that we can find an elimination order of the variables such that resolution runs in polynomial time on formulas of bounded primal tree width has already been shown by Alekhnovich and Razborov [AR11]. Their result is stated in terms of branch width but it is known that it is linearly related to primal tree width [Sze04].

In this section, we reprove this result by showing how one can use resolution on a well-chosen elimination order to prove this tractability result. We will use the elimination order characterizing tree width, recalled in Theorem 1.27. Intuitively, we can show that we can always resolve on a variable such that there is at most k variables that appear in a clause with x. Thus we can bound the number of clauses holding x and then the number of clauses that we introduce at each step of the algorithm. The main observation is the following:

Lemma 5.3. *Let F be a* CNF*-formula, G its primal graph and $x \in \mathsf{var}(F)$. The primal graph of $\mathsf{res}(F, x)$ is a subgraph of G/x.*

Proof. First observe that the variables $\mathsf{res}(F, x)$ are $\mathsf{var}(F) \setminus \{x\}$, thus they can be identified with the vertices of G/x.

Let $\{y, z\}$ be an edge of the primal graph of $\mathsf{res}(F, x)$. In particular, we have $y \neq x$ and $z \neq x$. We show that it is also an edge of G/x. If $\{y, z\}$ was already an edge of G then it is still an edge of G/x. Otherwise, it means it has been introduced by a new clause of $\mathsf{res}(F, x)$. That is, there exist clauses $C, C' \in F$ such that $x, y \in \mathsf{var}(C)$ and $x, z \in \mathsf{var}(C')$. Thus $\{x, y\}$ and $\{x, z\}$ are edges of G, that is, $\{y, z\}$ is an edge of G/x. □

Moreover, it is easy to see that if G' is a subgraph of G, then G'/x is also a subgraph of G'/x:

Lemma 5.4. *Let $G = (V, E)$ be a graph and $G' = (V, E')$ a subgraph of G (that is $E' \subseteq E$). Then for all $x \in V$, G'/x is a subgraph of G/x.*

Proof. It is sufficient to show that each edge of G'/x is also in G/x. Every edges of G'/x are either edges of G' that does not contain x and then they are also in G/x or edges $\{y, z\}$ such that $\{y, x\}$ and $\{z, x\}$ are edges of G'. But then $\{y, x\}$ and $\{z, x\}$ are also edges of G and then $\{y, z\}$ is an edge of G/x. □

Lemma 5.5. *Let F be a* CNF*-formula of primal tree width k. One can find in* FPT *times in the primal tree width of F an order $x_1, \ldots, x_n$ on $\mathsf{var}(F)$ such that for all $i \leq n$, $|F_i| \leq |F| + 3^k i$ where $F_0 = F$ and $F_{i+1} = \mathsf{res}(F, x_i)$.*

Proof. Let $x_1, \ldots, x_n$ be an elimination order of width k of the primal graph of F given by Theorem 1.27. Let G be the primal graph of F. We denote by $G_0 = G$ and $G_{i+1} = G_i/x_i$. Let G'_i be the primal graph of F_i. By Lemma 5.3 and Lemma 5.5, G'_i is a subgraph of G_i/x_i. Let N_i be the neighborhood of x_i in G'_i. Since the order is of width k, we know that the degree of x_i in G_i is at most k, and since the degree of x_i in G'_i is smaller than in G_i we have $|N_i| \leq k$. By definition of the primal graph of F_i, we know that each clause containing x_i in F_i has its variables included in $\{x_i\} \cup N_i$. Let $C \in F_{i+1}$ be a clause. If $C \notin F_i$, that is because C is the union of two clauses of F_i containing x. Thus $\mathsf{var}(C) \subseteq N_i$. Since there is at most $3^{|N_i|} \leq 3^k$ clauses on variables N_i, we have $|F_{i+1}| \leq |F_i| + 3^k$. By induction, for all i, $|F_i| \leq |F| + 3^k i$. □

Theorem 5.6. *Let F be a formula of primal tree width k. One can find in* FPT *time in the primal tree width of F an order $x_1, \dots, x_n$ on $\mathsf{var}(F)$ such that Algorithm 6 runs on time $O(3^{2k}|\mathsf{var}(F)|)$ on this order.*

5.1.4 Resolution on β-acyclic formulas

In this section, we recall the result of Ordyniak, Paulusma and Szeider [OPS13] concerning the tractability of deciding β-acyclic formulas. Their algorithm to decide β-acyclic formula is a resolution following a β-elimination order. The key observation is that if x is a β-leaf in the hypergraph of the formula and if C, D are two clauses with $x \in \mathsf{var}(C) \cap \mathsf{var}(D)$, then $(C \setminus \{x, \neg x\}) \cup (D \setminus \{x, \neg x\})$ is either tautological, or equal to $C \setminus \{x, \neg x\}$ or equal to $D \setminus \{x, \neg x\}$. Thus resolving a formula on a β-leaf does not introduce new clauses. This is formalized by the following lemma:

Lemma 5.7. *Let F be a* CNF*-formula and $x \in \mathsf{var}(F)$ a β-leaf of $\mathcal{H}(F)$. Let F' be the formula we get by removing tautological clauses from $\mathsf{res}(F, x)$. We have $|F'| \leq |F|$ and $\mathcal{H}(F') \subseteq \mathcal{H}(F) \setminus \{x\}$.*

Proof. Since x is a β-leaf in $\mathcal{H}(F)$, we have that $\{\mathsf{var}(C) \mid C \in F, x \in \mathsf{var}(C)\}$ is ordered by inclusion. Let $C_1 \in F$ with $x \in C_1$ and $C_2 \in F$ with $\neg x \in C_2$. Since x is a β-leaf, we have $\mathsf{var}(C_1) \subseteq \mathsf{var}(C_2)$ or $\mathsf{var}(C_2) \subseteq \mathsf{var}(C_1)$. Assume w.l.o.g that $\mathsf{var}(C_1) \subseteq \mathsf{var}(C_2)$. If $C_1 \setminus \{x\} \not\subseteq C_2 \setminus \{\neg x\}$, then there exists a literal $\ell \in C_1$ such that $\neg\ell \in C_2$. Thus $(C_1 \setminus \{x\}) \cup (C_2 \setminus \{\neg x\})$ is tautological and does not appear in F'. Otherwise $C_1 \setminus \{x\} \subseteq C_2 \setminus \{\neg x\}$ and $(C_1 \setminus \{x\}) \cup (C_2 \setminus \{\neg x\}) = C_2 \setminus \{\neg x\}$.

From what precedes, let C be a clause of F'. Either $C \in F$ and $x \notin \mathsf{var}(C)$ or $C = D \setminus \{x, \neg x\}$ for some $D \in F$ thus $|F'| \leq |F|$ and $\mathcal{H}(F') \subseteq \mathcal{H}(F) \setminus \{x\}$. □

Theorem 5.8. *Let F be a β-acyclic* CNF*-formula and $x_1, \dots, x_n$ a β-elimination order for $\mathcal{H}(F)$. Algorithm 6 on input F and $x_1, \dots, x_n$ runs in polynomial time.*

Proof. This is a straightforward consequence of Lemma 5.7. Let $F_0 = F$ and F_{i+1} is $\mathsf{res}(F, x_{i+1})$ where we have removed tautological clauses. We show by induction that $\mathcal{H}(F_i) \subseteq \mathcal{H}(F) \setminus \{x_1, \dots, x_i\}$, $|F_i| \leq |F|$ and that x_{i+1} is a β-leaf of $\mathcal{H}(F_i)$. By definition, x_1 is a β-leaf of $\mathcal{H}(F) = \mathcal{H}(F_0)$ and $\mathcal{H}(F_0) \subseteq \mathcal{H}(F)$ and $|F_0| \leq |F|$ since they are equal. Now let $i > 0$. By definition, x_{i+1} is a β-leaf of $\mathcal{H}(F) \setminus \{x_1, \dots, x_i\}$, thus x_{i+1} is a β-leaf of $\mathcal{H}(F_i) \subseteq \mathcal{H}(F) \setminus \{x_1, \dots, x_i\}$. Now by Lemma 5.7, $\mathcal{H}(F_{i+1}) \subseteq \mathcal{H}(F_i) \setminus \{x_{i+1}\} \subseteq \mathcal{H}(F) \setminus \{x_1, \dots, x_{i+1}\}$ and $|F_{i+1}| \leq |F_i| \leq |F|$.

Thus each recursive call to DP-Resolution is done on a formula of size at most $|F|$ and Algorithm 6 runs in polynomial time. □

5.2 Weighted DP-resolution for β-acyclic instances

It is a natural question to ask whether there is a relation between the number of satisfying assignments of a CNF-formula F and those of $\mathsf{res}(F, x)$. If one can

find such a relation, one may derive from it an algorithm for #SAT based on resolution and use it on the families presented in the previous section to get new tractability results for #SAT. This is however unlikely to happen without changing the algorithm as #SAT is known to be #P-complete on 2-CNF instances and Theorem 5.2 states that resolution works in polynomial time on 2-CNF. One can also realize that resolution is not adapted as it is for counting by studying the proof of Lemma 5.1. In this proof, we reconstruct a satisfying assignment of F from a satisfying assignment of $\mathsf{res}(F, x)$. We have seen that every satisfying assignment of $\mathsf{res}(F, x)$ either satisfies every clause containing the literal x or every clause containing the literal $\neg x$ in F. We thus construct a satisfying assignment of F by choosing the value of x to satisfy the remaining clauses of F. However, there may be some satisfying assignments of $\mathsf{res}(F, x)$ that satisfy both every clause containing x *and* every clause containing $\neg x$ and thus, the value of x could be chosen arbitrarily to construct a satisfying assignment of F. Those assignments account for two distinct satisfying assignments of F. Thus, to recover the number of models of F from the number of models of $\mathsf{res}(F, x)$, one must know the number of models of the conjunction of clauses containing the variable x which seems hard *a priori*.

The previous observation tells us that some satisfying assignments of $\mathsf{res}(F, x)$ should be counted twice if one wants to recover the number of satisfying assignments of F from those of $\mathsf{res}(F, x)$. This suggests the idea of working with weights on assignments and to discover a relation between the weight of F and the weight of some formula where we have removed x and updated the weights. Of course, since the resolvent of a 2-CNF formula is still a 2-CNF formula and since #2-SAT is #P-complete, we will have to modify the resolvent too. Some extensions of resolution already exist for MaxSAT such as [BLM07], presented as a resolution system. We do not think that such system could be extended for model counting however.

In this section, we give an algorithm inspired by the algorithm of Ordyniak, Paulusma and Szeider [OPS13] based on this idea of weighted clauses that solves #SAT on β-acyclic instances and precisely analyze its complexity. We show the result for a more general model than CNF-formula: CSP with default value. We start by introducing this notion and then present the algorithm and prove its correctness. We finish by analyzing its complexity, a task involving a better understanding of the arithmetic operations involved during the computation.

5.2.1 Encoding SAT using CSP with default values

To generalize resolution to model counting, we need to assign weights to clauses in order to keep more information on the original formula during the elimination process. This transformation can in fact be done for a much general problem than #SAT, namely, the problem of evaluating weighted *constraint satisfaction problems*, CSP for short . In this section, we define the problem without appealing to specific knowledge on CSP. Nevertheless, we briefly recall the basic ideas of

X_1	X_2	X_3	Weight
0	1	0	0
1	1	1	0
otherwise			1

A weighted constraint that is equivalent to
$(X_1 \vee \neg X_2 \vee X_3) \wedge (\neg X_1 \vee \neg X_2 \vee \neg X_3)$.

X_1	X_2	X_3	Weight
1	1	0	1
1	0	1	1
otherwise			0

A weighted constraint that is equivalent to $(X_1 \wedge X_2 \wedge \neg X_3) \vee (X_1 \wedge \neg X_2 \wedge X_3)$.

Figure 5.2: Weighted constraints with default value

this area in the next paragraph. The reader already familiar with CSP or only interested in how we generalize the problem may safely skip to Definition 5.9.

Constraint satisfaction problems is a natural generalization of SAT to non-boolean domains. Informally, a CSP on variables X and (finite) domain D is a set of constraints. A constraint C can be seen as a set $S_C \subseteq D^k$ together with a tuple $(x_1, \ldots, x_k)$ of variables X. An assignment $\tau : X \to D$ satisfies a constraint C if $(\tau(x_1), \ldots, \tau(x_k))$ is an element of S_C. An assignment τ satisfies the CSP if it satisfies every constraint in it. The interested reader may find more details on CSP in a survey by Chen [Che06],

SAT can be seen as a specific instance of CSP where $D = \{0, 1\}$ and where each clause C with k variables is a constraint on $\mathsf{var}(C)$ and such that the subset of $\{0, 1\}^k$ is the set of satisfying assignment of the clause C. However, since a clause with k variables has $2^k - 1$ satisfying assignments, such encoding of SAT into CSP may lead to a blow in size. Such encoding of CSP is known as positive encoding. We could also assume that instead of giving every satisfying assignment of a constraint, we give only the non-satisfying assignment. For the encoding of CSP, known as negative encoding, SAT could be encoded efficiently. The influence of the encoding of constraints on the complexity of the problem has been studied in [CGH09, CG10]. In this section, we introduce a new way of encoding CSPs with weights that encompasses both the positive encoding and the negative encoding.

In the problem we consider, the constraints will be encoded with a default value. Informally, a weighted constraint C with default value on variables X is given as a list of tuples associated with a weight in $\mathbb{Q}_+$ and a value μ, called the default value. The value of C on an assignment τ of X is either the weight associated to τ if τ is in the list, and μ otherwise. Formally:

Definition 5.9. *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 *is called*

the support *of C, and is denoted by $\mathsf{supp}(C)$. μ is called the default value of C, denoted by $\mathsf{def}(C)$ 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:*

$$C(a) = \begin{cases} f(a) \text{ if } a \in S \\ \mu \text{ otherwise.} \end{cases}$$

Figure 5.2 depicts two examples of how one can encode conjunctive and disjunctive clauses using weighted constraints with default value. In the rest of this section, we will identify a weighted constraint with default value $C = (f, \mu)$ with the total function it defines on $D^{\mathsf{var}(C)}$. Thus, when defining a constraint C, we will usually first define its support $\mathsf{supp}(C)$, its default value $\mathsf{def}(C) \in \mathbb{Q}_+$ and finally define the value of C on a for every $a \in \mathsf{supp}(C)$, without explicitly naming the underlying function f.

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

Observe that we are restricting weights to non-negative values. We will see that it is crucial for the correctness of our algorithm. The assumption that weights are rational is however only a commodity to only work with constants that are easy to encode. Our algorithm also works for positive algebraic numbers but we prefer to use rationals in order to highlight the main ideas of the algorithm. If one is only interested in the correctness of the algorithm, one can use weights in $\mathbb{R}_+$ but all results concerning the bit size of the weights appearing during the computation are not usable in this case.

We are interested in the problem $\#\mathrm{CSP}_{\mathrm{def}}$ of computing the weight of a given set I of weighted constraints with default value on a finite domain D where the weight of I is defined as:

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

and $\mathsf{var}(I) := \bigcup_{C \in I} \mathsf{var}(C)$.

The *size* $\|I\|$ of a $\#\mathrm{CSP}_{\mathrm{def}}$-instance I is defined to be $\|I\| := \sum_{C \in I} |C|$. Its *structural size* $s(I)$ is defined to be $s(I) := \sum_{C \in I} |\mathsf{var}(C)|$. A hypergraph $\mathcal{H}(I)$, an incidence graph $\mathcal{G}_{\mathsf{inc}}(I)$ and a primal graph $\mathcal{G}_{\mathsf{prim}}(I)$ are associated to I in the same way as for CSP.

Observe 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.

Given an instance I, it will be useful to condition it under some partial assignment, that is, the weight of I where some of its variables are forced to a certain value. To this end, for $a \in D^{|\mathsf{var}(I)|}$, we define:

$$w(I, a) := \sum_{\substack{b \in D^{|\mathsf{var}(I)|} \\ a \sim b}} \prod_{C \in I} C(b).$$

As usual, we associate a hypergraph $\mathcal{H}(I)$ to a $\#\mathrm{CSP}_{\mathrm{def}}$ instance on vertices $\mathsf{var}(I)$ defined as $\mathcal{H}(I) = \{\mathsf{var}(C) \mid C \in I\}$.

Encoding #SAT. Weighted CSPs with default values are enough to encode the #SAT problem without losing the structure of the formula as stated formally by the following lemma:

Lemma 5.10. *Given a CNF-formula F one can construct in polynomial time a $\#\mathrm{CSP}_{\mathrm{def}}$-instance I on variables $\mathsf{var}(F)$ and domain $\{0,1\}$ such that*

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

5.2.2 Computing the weight of a chain

In this section, we present a method that, given some instance I on domain D and a variable x of $\mathsf{var}(I)$, produces an instance I' such that $\mathcal{H}(I') = \mathcal{H}(I) \setminus \{x\}$ and $w(I') = w(I)$. We present the method on a very simple case, the case where the variable of the constraints of I are ordered by inclusion, that is, $I = \{C_1, \ldots, C_m\}$ with $\mathsf{var}(C_1) \subseteq \cdots \subseteq \mathsf{var}(C_m)$. This will be later generalized for β-acyclic instances in Section 5.2.3. The main purpose of this section is to explain how we come up with the procedure described in the forthcoming Proposition 5.11. The reader only interested in the main algorithm may safely skip this section.

An example. Figure 5.3 shows how the weights are updated when we remove the variable X_1 from the chain C_1, C_2, C_3 where the domain is $\{0,1\}$. It can be verified that the weight of the first instance is the same as the weight of the updated instance.

Indeed, we denote by $I = \{C_1, C_2, C_3\}$ and by $I' = \{C_1', C_2', C_3'\}$. We can verify that for every $a : \{X_2, X_3, X_4\} \to \{0,1\}$, $w(I', a) = w(I, a)$. As an example,

C_1		
X_1	X_2	w
0	0	2
1	0	3
otherwise		1

C_2			
X_1	X_2	X_3	w
0	0	0	4
1	1	1	5
otherwise			0

C_3				
X_1	X_2	X_3	X_4	w
0	0	0	0	6
otherwise				1

C'_1	
X_2	w
0	$(2+3)=5$
otherwise	2

C'_2		
X_2	X_3	w
0	0	$(2\cdot 4+3\cdot 0)/(2+3)=8/5$
1	1	$(1\cdot 0+1\cdot 5)/(1+1)=5/2$
otherwise		0

C'_3			
X_2	X_3	X_4	w
0	0	0	$(2\cdot 4\cdot 6+3\cdot 0\cdot 1)/(2\cdot 4+3\cdot 0)=6$
otherwise			1

Figure 5.3: Eliminating X_1 from the chain C_1, C_2, C_3 on domain $\{0,1\}$.

let $a = \{X_2 \mapsto 0, X_3 \mapsto 0, X_4 \mapsto 0\}$. By definition, $C'_1(a) = 5$, $C'_2(a) = 8/5$ and $C'_3(a) = 6$. Thus $w(I', a) = 5 \cdot (8/5) \cdot 6 = 48$. Now, we denote by $a_0 = a \cup \{X_1 \mapsto 0\}$ and by $a_1 = a \cup \{X_1 \mapsto 1\}$. We have by definition:

$$\begin{aligned} w(I, a) &= C_1(a_0)C_2(a_0)C_3(a_0) + C_1(a_1)C_2(a_1)C_3(a_1) \\ &= 2 \cdot 4 \cdot 6 + 3 \cdot 0 \cdot 1 \\ &= 48. \end{aligned}$$

A sufficient condition. Let x be a variable in $\mathsf{var}(C_1)$. We want to build an instance $I' = \{C'_1, \ldots, C'_m\}$ with $\mathsf{var}(C'_i) = \mathsf{var}(C_i) \setminus \{x\}$ for all i and $w(I') = w(I)$. To see how it can be done, we reinforce the hypothesis and we want that the new instance I' verifies that for all $i \leq m$ and for all assignments a of $\mathsf{var}(I')$, the weight of the subinstance $I_i = \{C_1, \ldots, C_i\}$ conditioned by a is preserved. In other word, we want that for all a, $w(I'_i, a) = w(I_i, a)$ where $I_i = \{C_1, \ldots, C_i\}$ and $I'_i = \{C'_1, \ldots, C'_i\}$.

Observe that if a is an assignment of $\mathsf{var}(I')$, then for all $i \leq m$, $w(I'_i, a) = \prod_{j=1}^{i} C'_j(a)$ and since $\mathsf{var}(I') = \mathsf{var}(I) \setminus \{x\}$,

$$\begin{aligned} w(I_i, a) &= \sum_{d \in D} w(I_i, a \cup \{x \mapsto d\}) \\ &= \sum_{d \in D} \prod_{j=1}^{i} C_j(a \cup \{x \mapsto d\}). \end{aligned}$$

We construct I'_i by induction on i. Let $i = 1$. We construct $I'_1 = \{C'_1\}$ such that for all $a : \mathsf{var}(C_1) \setminus \{x\} \to D$:

$$C'_1(a) = w(I'_1, a) = w(I_1, a) = \sum_{d \in D} C_1(a \cup \{x \mapsto d\}). \tag{5.1}$$

Now let $\mathsf{supp}(C'_1) = \{b|_{\mathsf{var}(C_1)\setminus\{x\}} \mid b \in \mathsf{supp}(C_1)\}$. Observe that $\mathsf{supp}(C'_1)$ is smaller than $\mathsf{supp}(C_1)$ by definition. Moreover, if $a \notin \mathsf{supp}(C'_1)$, then for $d \in D$, $(a \cup \{x \mapsto d\}) \notin \mathsf{supp}(C_1)$. In other words,

$$w(I_1, a) = \sum_{d \in D} C_1(a \cup \{x \mapsto d\}) = |D|\mathsf{def}(C_1).$$

Moreover, $C'_1(a) = \mathsf{def}(C'_1)$ since $a \notin \mathsf{supp}(C'_1)$. Thus, if we choose $\mathsf{def}(C'_1) = |D|\mathsf{def}(C_1)$, we get $w(I'_1, a) = w(I_1, a)$.

Now, if $a \in \mathsf{supp}(C'_1)$, the Equation (5.1) implies:

$$C'_1(a) = w(I'_1, a) = w(I_1, a) = \sum_{d \in D} C_1(a \cup \{x \mapsto d\}).$$

Thus, if we choose this value as the new weight for $C'_1(a)$, we have $w(I'_1, a) = w(I_1, a)$.

We just have constructed a new constraint C'_1, smaller than C_1 since its support is a projection of the support of C_1 and such that $w(I'_1, a) = w(I_1, a)$ for every $a : \mathsf{var}(C_1) \setminus \{x\} \to D$ which is our induction hypothesis at rank 1.

Now, let $i > 1$ and assume that we have constructed $C'_1, \ldots, C'_{i-1}$ such that for all $a : \mathsf{var}(C_{i-1}) \setminus \{x\} \to D$, it holds:

$$w(I'_{i-1}, a) = \prod_{j=1}^{i-1} C'_j(a) = w(I_i, a) = \sum_{d \in D} \prod_{j=1}^{i} C_j(a \cup \{x \mapsto d\}).$$

We want to construct C'_i such that for all $a : \mathsf{var}(C_i) \setminus \{x\} \to D$:

$$\prod_{j=1}^{i} C'_j(a) = w(I'_i, a) = w(I_i, a) = \sum_{d \in D} \prod_{j=1}^{i} C_j(a \cup \{x \mapsto d\}).$$

That is, if $\prod_{j=1}^{i-1} C'_j(a) \neq 0$:

$$C'_i(a) = \frac{\sum_{d \in D} \prod_{j=1}^{i} C_j(a \cup \{x \mapsto d\})}{\prod_{j=1}^{i-1} C'_j(a)} = \frac{\sum_{d \in D} \prod_{j=1}^{i} C_j(a \cup \{x \to d\})}{w(I'_{i-1}, a)}.$$

By construction we have $w(I'_{i-1}, a) = w(I_{i-1}, a) = \sum_{d \in D} \prod_{j=1}^{i-1} C_j(a \cup \{x \mapsto d\})$. Thus, we want for all $a : \mathsf{var}(C'_i) \to D$:

$$C'_i(a) = \frac{\sum_{d \in D} \prod_{j=1}^{i} C_j(a \cup \{x \mapsto d\})}{\sum_{d \in D} \prod_{j=1}^{i-1} C_j(a \cup \{x \mapsto d\})} \tag{5.2}$$

Again, we choose $\mathsf{supp}(C'_i) = \{b|_{\mathsf{var}(C_i)\setminus\{x\}} \mid b \in \mathsf{supp}(C_i)\}$. If $a \notin \mathsf{supp}(C'_i)$, then for all $d \in D$, $(a \cup \{x \mapsto d\}) \notin \mathsf{supp}(C_i)$. It follows that

$$\sum_{d\in D}\prod_{j=1}^{i} C_j(a \cup \{x \mapsto d\}) = \mathsf{def}(C_i)\sum_{d\in D}\prod_{j=1}^{i-1} C_j(a \cup \{x \mapsto d\}).$$

Thus, since $C'_i(a) = \mathsf{def}(C'_i)$, it we choose $\mathsf{def}(C'_i) = \mathsf{def}(C_i)$, we have for $a \notin \mathsf{supp}(C'_i)$, $w(I'_i, a) = w(I_i, a)$. Now if $a \in \mathsf{supp}(C_i)$, we choose $C'_i(a)$ as in Equation 5.2 and we have by induction $w(I'_i, a) = w(I_i, a)$.

However, choosing $\mathsf{supp}(C'_i)$ as in Equation 5.2 is possible only if the denominator is non zero. Otherwise, assume that

$$\sum_{d\in D}\prod_{j=1}^{i-1} C_j(a \cup \{x \mapsto d\}) = 0.$$

This is where the hypothesis that the constraint have positive values is useful. It follows that the previous sum is a sum of positive terms evaluating to 0. Thus for all $d \in D$, $\prod_{j=1}^{i-1} C_j(a\cup\{x \mapsto d\}) = 0$ and then $\sum_{d\in D}\prod_{j=1}^{i} C_j(a\cup\{x \mapsto d\}) = 0$ too. That is, whatever value we choose for $C'_i(a)$, we have

$$w(I_i, a) = \sum_{d\in D}\prod_{j=1}^{i} C_j(a \cup \{x \mapsto d\}) = 0 = \prod_{j=1}^{i} C'_j(a) = w(I'_i, a).$$

In particular, we could choose $C'_i(a) = \mathsf{def}(C_i)$. In other words, we can remove a from the support of C'_i and still preserve the weight of the instance.

5.2.3 Computing the weight of β-acyclic instances

We just described a procedure that removes a β-leaf in a specific $\#\mathrm{CSP}_{\mathrm{def}}$ instance without changing its weight. Using a β-elimination order, we are able to generalize this method to any β-acyclic instance. Given a set A of assignment from a set of variables X to D and $x \in X$, we denote by $\pi_x(A)$ the set of assignments of A restricted to $V \setminus \{x\}$, that is:

$$\begin{aligned}\pi_x(A) &= \{a|_{V\setminus\{x\}} \mid a \in A\}\\ &= \{b : V \setminus \{x\} \to \{0,1\} \mid \exists d \in D, b \cup \{x \mapsto d\} \in A\}.\end{aligned}$$

Let I be a set of weighted constraints on domain D and x a β-leaf of $\mathcal{H}(I)$. Let $I(x) = \{C_1, \ldots, C_p\}$ be the constraints C of I such that $x \in \mathsf{var}(C)$ with $\mathsf{var}(C_1) \subseteq \ldots \subseteq \mathsf{var}(C_p)$. We denote by $\mathsf{wres}(I, x) = (I \setminus I(x)) \cup \{C'_i \mid i \leq p\}$ where for every $i \leq p$, C'_i is defined as follows:

- $\mathsf{var}(C'_i) = \mathsf{var}(C_i) \setminus \{x\}$,
- $\mathsf{def}(C'_1) = |D| \cdot \mathsf{def}(C_1)$ and $\mathsf{def}(C'_i) = \mathsf{def}(C_i)$ if $i > 1$,

- $\mathsf{supp}(C'_1) = \pi_x(\mathsf{supp}(C_1))$ and for $i > 1$, $\mathsf{supp}(C'_i)$ is the set of assignments a of $\pi_x(\mathsf{supp}(C_i))$ such that

$$\sum_{d \in D} \prod_{j=1}^{i-1} C_j(a \cup \{x \mapsto d\}) \neq 0$$

and,

- for all $a \in \mathsf{supp}(C'_i)$,

$$C'_1(a) := \sum_{d \in D} C_1(a \cup \{x \mapsto d\})$$

and for $i > 1$,

$$C'_i(a) := \frac{\sum_{d \in D} \prod_{j=1}^{i} C_j(a \cup \{x \mapsto d\})}{\sum_{d \in D} \prod_{j=1}^{i-1} C_j(a \cup \{x \mapsto d\})}.$$

Observe that the denominator is non-zero by definition of $\mathsf{supp}(C'_i)$. We have the following:

Proposition 5.11. *Let I be a set of weighted constraints on domain D and x a β-leaf of $\mathcal{H}(I)$. Then*

- $\mathcal{H}(\mathsf{wres}(I, x)) = \mathcal{H}(I) \setminus x$,
- $\|\mathsf{wres}(I, x)\| \leq \|I\|$ *and*

$$w(I) = w(\mathsf{wres}(I, x)).$$

Moreover, one can compute $\mathsf{wres}(I, x)$ with $O(p \cdot |D| \cdot \|I(x)\|)$ arithmetic operations where $I(x)$ is the set of constraints C such that $x \in \mathsf{var}(C)$ and p is the number of such constraints.

Proof. In this proof, we denote by $I' = \mathsf{wres}(I, x)$ and $I(x) = \{C_1, \ldots, C_p\}$ with $\mathsf{var}(C_1) \subseteq \ldots \subseteq \mathsf{var}(C_p)$.

First observe that $\mathcal{H}(I') = \mathcal{H}(I) \setminus x$ since for every constraint C' of I', there exists a constraint C of I with $\mathsf{var}(C) \setminus \{x\} = \mathsf{var}(C')$. Thus the edges of $\mathcal{H}(I')$ are exactly the edges of $\mathcal{H}(I)$ where x has been removed.

Moreover $\|I'\| \leq \|I\|$ since for all $C \in I$, $|\mathsf{supp}(C')| \leq |\mathsf{supp}(C)|$ because every assignment of $\mathsf{supp}(C')$ is the projection of an assignment of $\mathsf{supp}(C)$ on $\mathsf{var}(C) \setminus \{x\}$.

We now prove that $w(I) = w(I')$. To this end, we prove by induction on i that for all $a : \mathsf{var}(C'_i) \to D$:

$$\prod_{j=1}^{i} C'_j(a) = \sum_{d \in D} \prod_{j=1}^{i} C_j(a \cup \{x \mapsto d\})$$

We start with $i = 1$. We want to prove that for all assignments a of $\mathsf{var}(C'_1)$, $C'_1(a) = \sum_{d\in D} C_1(a \cup \{x \mapsto d\})$. If $a \in \mathsf{supp}(C'_1)$, this follows directly from the definition.

Otherwise, if $a \notin \mathsf{supp}(C'_1)$, for all $d \in D$, $(a \cup \{x \mapsto d\}) \notin \mathsf{supp}(C_1)$ by definition of $\mathsf{supp}(C'_1)$. Thus $\sum_{d\in D} C_1(a \cup \{x \mapsto d\}) = |D|\mathsf{def}(C_1)$. Since for $a \notin \mathsf{supp}(C'_1)$, we have

$$\begin{aligned} C'_1(a) &= \mathsf{def}(C'_1) \\ &= |D|\mathsf{def}(C_1) \text{ by definition of } C'_1 \\ &= \sum_{d\in D} C_1(a \cup \{x \mapsto d\}). \end{aligned}$$

Now assume the result is true for all $k \leq i$. We want to prove that for all assignment a of $\mathsf{var}(C'_{i+1})$:

$$|D| \prod_{j=1}^{i+1} C'_j(a) = \sum_{d\in D} \prod_{j=1}^{i+1} C_j(a \cup \{x \mapsto d\}).$$

By induction,

$$|D| \prod_{j=1}^{i} C'_j(a) = \sum_{d\in D} \prod_{j=1}^{i} C_j(a \cup \{x \mapsto d\}).$$

Assume first that $a \notin \mathsf{supp}(C'_{i+1})$. By definition, we either have that for all $d \in D$, $C_{i+1}(a \cup \{x \mapsto d\}) = \mathsf{def}(C_{i+1}) = \mathsf{def}(C'_{i+1})$ and then

$$\begin{aligned} \sum_{d\in D} \prod_{j=1}^{i+1} C_j(a \cup \{x \mapsto d\}) &= \sum_{d\in D} \mathsf{def}(C_{i+1}) \prod_{j=1}^{i} C_j(a \cup \{x \mapsto d\}) \\ &= \mathsf{def}(C'_{i+1}) \sum_{d\in D} \prod_{j=1}^{i} C_j(a \cup \{x \mapsto d\}) \\ &= \prod_{j=1}^{i+1} C'_j(a). \end{aligned}$$

Or we have $\sum_{d\in D} \prod_{j=1}^{i} C_j(a \cup \{x \mapsto d\}) = 0$. Since the weights are positive, each term of the previous sum is null, that is, $\prod_{j=1}^{i} C_j(a \cup \{x \mapsto d\}) = 0$ for all $d \in D$.

Thus $\prod_{j=1}^{i+1} C_j(a \cup \{x \mapsto d\}) = 0$ too. It follows:

$$\begin{aligned}\sum_{d\in D}\prod_{j=1}^{i+1} C_j(a\cup\{x\mapsto d\}) &= 0\\ &= \sum_{d\in D}\prod_{j=1}^{i} C_j(a\cup\{x\mapsto d\})\\ &= \prod_{j=1}^{i} C'_j(a) \text{ by induction.}\\ &= \prod_{j=1}^{i+1} C'_j(a)(=0).\end{aligned}$$

Finally, assume that $a \in \mathsf{supp}(C'_{i+1})$. By definition,

$$C'_{i+1}(a) = \frac{\sum_{d\in D}\prod_{j=1}^{i+1} C_j(a\cup\{x\mapsto d\})}{\sum_{d\in D}\prod_{j=1}^{i} C_j(a\cup\{x\mapsto d\})}$$

By induction:

$$C'_{i+1}(a) = \frac{\sum_{d\in D}\prod_{j=1}^{i+1} C_j(a\cup\{x\mapsto d\})}{\prod_{j=1}^{i} C'_j(a)}$$

That is:

$$\prod_{j=1}^{i+1} C'_j(a) = \sum_{d\in D}\prod_{j=1}^{i+1} C_j(a\cup\{x\mapsto d\})$$

which concludes the induction. Applying it to $i = m$ yields:

$$\prod_{C\in I(x)} C'(a) = \sum_{d\in D}\prod_{C\in I(x)} C(a\cup\{x\mapsto d\}).$$

Thus

$$\begin{aligned}w(I) &= \sum_{a:\mathsf{var}(I)\to D}\prod_{C\in I(x)} C(a)\prod_{C\notin I(x)} C(a)\\ &= \sum_{a:\mathsf{var}(I)\setminus\{x\}\to D}\sum_{d\in D}\prod_{C\in I(x)} C(a\cup\{x\mapsto d\})\prod_{C\notin I(x)} C(a)\\ &= \sum_{a:\mathsf{var}(I)\setminus\{x\}\to D}\prod_{C\notin I(x)} C(a)\Big(\sum_{d\in D}\prod_{C\in I(x)} C(a\cup\{x\mapsto d\})\Big)\\ &= \sum_{a:\mathsf{var}(I)\setminus\{x\}\to D}\prod_{C\notin I(x)} C(a)\Big(\prod_{C\in I(x)} C'(a)\Big)\\ &= w(I').\end{aligned}$$

Finally we can compute I' with $O(p \cdot |D| \cdot \|I(x)\|)$ arithmetic operations since for each $i \leq p$ we have to compute at most $|C_i|$ sums over D terms that are products of at most p values. □

Theorem 5.12. *Given a β-acyclic $\#\mathrm{CSP}_{\mathrm{def}}$-instance I and a β-elimination order of $\mathcal{H}(I)$, we can compute $w(I)$ with $O(s(I) \cdot |D| \cdot \|I\|)$ arithmetic operations.*

Proof. We eliminate variables from I using Proposition 5.11 following the β-elimination order until we get an instance J with no variables and such that $w(I) = w(J) = \prod_{C \in J} C(a_\emptyset)$ where $a_\emptyset$ denotes here the empty assignment. Eliminating x takes $O(|D| \cdot |I(x)| \cdot \|I(x)\|)$ arithmetic operations where $I(x)$ is the set of constraints holding x. Thus, we need a total number of

$$O\Big(\sum_{x \in \mathsf{var}(I)} |D| \cdot |I(x)| \cdot \|I(x)\|\Big) = O\big(|D| \cdot \|I\| \cdot \sum_{x \in \mathsf{var}(I)} |I(x)|\big)$$
$$= O\big(|D| \cdot s(I) \cdot \|I\|\big)$$

arithmetic operations. □

Algorithm 7: An algorithm to compute $\mathsf{wres}(I, x)$ given an instance I and a β-leaf x

Data: A $\#\mathrm{CSP}_{\mathrm{def}}$ instance I, a β-leaf x, $<$ an order on I
begin
  $J \leftarrow I(x)$ sorted according to $<$;
  $I \leftarrow I \setminus J$;
  $V \leftarrow \mathsf{var}(J[1]) \setminus \{x\}$;
  $C \leftarrow$ empty constraints on variables V and default value $|D|\mathsf{def}(J[1])$;
  for $a \in \mathit{supp}(J[1])$ **do**
    Add $a|_V$ to $\mathsf{supp}(C)$ with weight $\sum_{d \in D} J[1](a \cup \{x \mapsto d\})$;
  $I \leftarrow I \cup \{C\}$;
  for $j = 2 \ldots |J|$ **do**
    $V \leftarrow \mathsf{var}(J[j]) \setminus \{x_i\}$;
    $C \leftarrow$ empty constraints on variables V and default value $\mathsf{def}(J[j])$;
    for $a \in \mathit{supp}(J[j])$ **do**
      $A \leftarrow \sum_{d \in D} \prod_{k=1}^{j-1} J[k](a \cup \{x_i \mapsto d\})$;
      if $A \neq 0$ **then**
        $B \leftarrow \sum_{d \in D} \prod_{k=1}^{j} J[k](a \cup \{x_i \mapsto d\})$;
        Add $a|_V$ to $\mathsf{supp}(C)$ with weight B/A;
    $I \leftarrow I \cup \{C\}$;
  return I

In the case of a CNF-formula, $s(I)$ is roughly the same as $\|I\|$ since each clause corresponds to a constraint of size 1 and thus, the algorithm of Theorem 5.12 is

Algorithm 8: An algorithm to compute $w(I)$ for I a β-acyclic $\#\mathrm{CSP}_{\mathrm{def}}$-instance

Data: A β-acyclic $\#\mathrm{CSP}_{\mathrm{def}}$ instance I, a β-elimination order $x_1, \ldots, x_n$ of $\mathsf{var}(I)$

begin

Compute the order $<_{\mathcal{H}(I)}$ (see Lemma 4.8) ;

Sort I according to $<_{\mathcal{H}(I)}$;

for $i = 1 \ldots n$ **do**

$I \leftarrow \mathsf{wres}(I, x_i, <_{\mathcal{H}_i})$

return $\prod_{C \in I} C(a_\emptyset)$

quadratic in the size of the formula. However, in the case of general queries, $s(I)$ is usually much smaller than $\|I\|$ thus it is then interesting to have a good complexity in $\|I\|$. Brault-Baron proved in [BB12] under some reasonable conditions that the problem of evaluating a negatively encoded conjunctive query I with a fixed hypergraph can be done in quasi-linear time in $\|I\|$ if and only if $\mathcal{H}(I)$ is β-acyclic. Theorem 5.12 is not strong enough to prove that the result still holds for counting since we have a $|D| \cdot \|I\|$ factor in the complexity of the query which is not linear in the size of the domain. Moreover, we do not know exactly what is the size of the weights during the computation.

The next section gives a more precise analysis of the runtime of Algorithm 8. We prove that Algorithm 8 runs in polynomial time.

5.2.4 Runtime of Algorithm 8

Theorem 5.12 only states that one needs a polynomial number of *arithmetic operations* to compute $w(I)$ for a β-acyclic instance I. Yet it is possible that these arithmetic operations are done on rational numbers which are not of polynomial size in $\|I\|$. Take for instance the sequence defined by $u_0 = 2$ and $u_{n+1} = u_n^2$. It is easy to see that u_n can be evaluated with n multiplications but since $u_n = 2^{2^n}$, the result is of bit size 2^n thus u_n cannot truly be evaluated in polynomial time in n. It follows that in order to prove that our algorithm works in polynomial time, we need to bound the size of the rational numbers we are dealing with, that is, the value of each constraint after the deletion of a vertex. This is delicate since naively bounding the numerator and the denominator of each constraint independently does not work: we are multiplying too many terms at each step. The trick is that most of the terms in the multiplication are canceling out and in fact, the algorithm runs in polynomial time. To prove this, we prove that, at each step, the weight of a constraint is the ratio of the weight of two sub-instances of the original instance.

The key part of what follows is to remark that not only $w(I)$ is preserved during the elimination procedure of Proposition 5.11 but also the weight of other sub-

instances of I. We actually already proved in the proof of Proposition 5.11 that there are many sub-instances of I whose weight is preserved by our elimination procedure. The same sort of preservation is true for β-acyclic instances.

Notations and assumptions. We rely on results of Chapter 4 concerning the structure of a β-acyclic hypergraph. In the following, we fix a β-acyclic instance I of hypergraph $\mathcal{H}$. We assume that for all $C_1, C_2 \in I$, $\mathsf{var}(C_1) \neq \mathsf{var}(C_2)$. Any instance I can be transformed in order to verify this property without increasing its size nor changing its weight. If there exist $C_1, C_2 \in I$ with $\mathsf{var}(C_1) = \mathsf{var}(C_2)$, we replace them with a constraint C with $\mathsf{supp}(C) = \mathsf{supp}(C_1) \cup \mathsf{supp}(C_2)$, $\mathsf{def}(C) = \mathsf{def}(C_1) \cdot \mathsf{def}(C_2)$ and for all $a \in \mathsf{supp}(C)$, $C(a) = C_1(a)C_2(a)$. We have $|C| \leq |C_1| + |C_2|$ thus we have not increased the size of the instance and we have chosen the default value of C such that for all assignments a, $C(a) = C_1(a)C_2(a)$. Thus we have not changed the size of the instance. We iterate this transformation until we have removed all such constraints. This assumption on I allows us to identify the edges of $\mathcal{H}$ with the constraints of I.

We choose $x_1, \ldots, x_n$, a β-elimination order of $\mathcal{H} = \mathcal{H}(I)$. We denote by $(\leq_{\mathcal{H}})$ the order on $\mathcal{H}$ associated to this elimination order given by Lemma 4.8 and use notations $\mathcal{H}_e^{x_i}$ of Section 4.2.1, page 89. This yields an order on I as well since the constraints of I have been identified with the edges of $\mathcal{H}$. Given $i \in [n]$ and $C \in I$, we denote by I_C^i the sub-instance corresponding to $\mathcal{H}^{x_i}_{\mathsf{var}(C)}$.

Each time we eliminate a variable from I, we get a new instance where we have transformed each constraint of I but we have not added nor deleted any constraint from I, thus, each constraint of the new instance can be associated to a constraint of the original instance. We denote by $I(k)$ the instance we have after the deletion of x_k. By convention, $I = I(0)$. Given $C \in I$, we denote by $C^{(k)} \in I(k)$ the constraint corresponding to C in the new instance $I(k)$. By convention, $C = C^{(0)}$. These notations extend to sub-instances $J \subseteq I$, that is, we denote by $J(k) = \{C^{(k)} \mid C \in J\}$. In particular, we are interested in $I_C^i(k)$ sub-instances with $i \geq k$. Observe that the variables of $J(k)$ are included $V_{>k}$ and $\mathsf{var}(C^{(k)}) = \mathsf{var}(C) \cap V_{>k}$. Observe also that by Lemma 4.12, $\mathsf{var}(I_C^k) \subseteq \mathsf{var}(C) \cup V_{>k}$ and thus $\mathsf{var}(I_C^k(j)) \subseteq \mathsf{var}(C) \cap V_{>j}$ for $j > k$ since we have removed every variable of $V_{>k}$ in this instance.

The intuitive reason why we consider such sub-instances is the following: when we remove a β-leaf from I, the weight of each constraint containing x is updated. The weight of the bigger constraints will now depend on the weight of the constraints that are included in them. Now if there is a path from a constraint C_1 to C_2 with $C_1 < C_2$ going through vertices smaller than x_k, then the weight of the constraint C_2 after having eliminated $x_1, \ldots, x_k$ will depend on the original weight of C_1 since the original weight of C_1 will "transfer" along this path to C_2. This dependency is precisely stated in the forthcoming Lemma 5.13.

Bounding the weights. In the definition of $\mathsf{wres}(I, x)$, we order the constraints containing x by inclusion, but it is not deterministic as some constraints may become equal once we have removed sufficiently enough variables. We assume that we always follow the order $(\leq_{\mathcal{H}})$ on $\mathcal{H}$ during the elimination process as it is done in Algorithm 8. We show the following:

Lemma 5.13. *For all $C \in I$, $k \leq n$ and $a \in \mathsf{supp}(C^{(k)})$, it holds that:*

$$C^{(k)}(a) \cdot w(I_C^k \setminus \{C\}, a) = w(I_C^k, a).$$

We delay the proof of Lemma 5.13 a bit to explain how it ensures that Algorithm 8 runs in polynomial time on a RAM machine. We can use Lemma 5.13 to show that the size of $C^{(k)}(a)$ is polynomially bounded for every $C \in I$ and $k \leq n$, thus, every arithmetic operations that are done by Algorithm 8 can be done in polynomial time. The bound is the following:

Proposition 5.14. *For every $k \leq n$, for every $C \in I$ and $a \in \mathsf{supp}(C^{(k)})$, $C^{(k)}$ is a positive rational number of size bounded by $O(n \cdot \log |D| \cdot s \cdot \|I\|)$ where s is the maximal size of the initial weights.*

Proof. If $J \subseteq I$ and b is a partial assignment of $\mathsf{var}(I)$, it holds that $w(J, b)$ is a rational number of size bounded by $O(n \cdot \log |D| \cdot s \cdot \|I\|)$. Indeed, by definition,

$$w(J, b) = \sum_{a:\mathsf{var}(I)\to D, a\simeq b} \prod_{C\in I} C(a)$$

is a sum of at most $|D|^n$ terms (where $n = |\mathsf{var}(I)|$). Each term of this sum is a product of $|I|$ rational numbers. We first choose a common denominator for these terms. Observe that in I, there is at most $\|I\|$ distinct weights, thus at most $\|I\|$ denominators. For every term in the sum $w(J, b)$, we can choose the product of all denominators of the initial weights of I. Each initial denominator being of size at most s, this common denominator is of size $O(\|I\| \cdot s)$. Now, with this common denominator, each numerator is of size $O(\|I\| \cdot s)$ also. Thus, the numerator of $w(J, b)$ is of size bounded by $O(n \cdot \log |D| \cdot s \cdot \|I\|)$ and the size of $w(J, b)$ is thus bounded by $O(n \cdot \log |D| \cdot s \cdot \|I\|)$ too.

Now by Lemma 5.13, it holds

$$C^{(k)}(a) \cdot w(I_C^k \setminus \{C\}, a) = w(I_C^k, a).$$

If $w(I_C^k \setminus \{C\}, a) \neq 0$ then $C^{(k)}(a)$ is the ratio of two rational numbers of size bounded by $O(n \cdot \log |D| \cdot s \cdot \|I\|)$, thus the size of $C^{(k)}(a)$ is bounded by $O(n \cdot \log |D| \cdot s \cdot \|I\|)$.

Now assume that $w(I_C^k \setminus \{C\}, a) = 0$. We construct a new instance $I(X)$ from I by replacing each weight equal to 0 by a constant $X > 0$. In this instance, there is no null weight, thus the weight of sub-instances is not null. The weight of each sub-instance may be seen as a polynomial in X. We denote by $Q(X) = w(I_C^k(X) \setminus$

$\{C\}, a)$ and $P(X) = w(I^k_C(X), a)$ (where C in $I(X)$ is the constraint corresponding to C). Observe that $Q(0) = w(I^k_C \setminus \{C\}, a)$ and $P(0) = w(I^k_C, a)$. Moreover, the coefficients of P and Q are rational numbers of size at most $O(n \cdot \log |D| \cdot s \cdot \|I\|)$ by the same argument as before.

We also have that $C^{(k)}(a)$ in $I(X)$ is an algebraic fraction $F(X)$ since every update of the weights only performs additions, multiplications or divisions. Moreover, observe that $C^{(k)}(a) = F(0)$ is finite.

During the execution of the algorithm, the only way the support of a constraint can diminish is when a sub-instance has a null weight. Since it does not happen in $I(X)$, we know that $a \in \mathsf{supp}(C^{(k)})$ even in $I(X)$ (again we identify C in I and $I(X)$). Lemma 5.13 still holds for $I(X)$ and thus we have:

$$F(X) \cdot Q(X) = P(X).$$

Thus $F(X) = (P/Q)(X)$. Moreover, $\lim_{X \to 0}(P/Q)(X) = C^{(k)}(a)$ is finite. Since the limit exists, it means that there exists $l \geq l'$ such that $P(X) = X^{l'} p_{l'} + o_{X \to 0}(X^{l'})$ and $Q(X) = X^l i q_l + o_{X \to 0}(X^l)$. In other words, if $l' > l$ then $C^{(k)}(a) = 0$ and $C^{(k)}(a) = p_l / q_l$ otherwise. Since both p_l and q_l are coefficients of P and Q they are of size bounded by $O(n \cdot \log |D| \cdot s \cdot \|I\|)$, so the size of $C^{(k)}(a)$ is at most $O(n \cdot \log |D| \cdot s \cdot \|I\|)$. □

Before proving Lemma 5.13, we show that the weight of some sub-instances of I is also preserved during the elimination process. In Lemma 5.15, we show that the weight of $I^j_C(k)$ for $j \geq k$ is the same as the weight of I^j_C. In Lemma 5.16, we show the same weight preservation for the sub-instance $(I^j_C \setminus \{C\})(k)$. The proof of Lemma 5.13 then follows since we can easily express $C^{(k)}$ from the weights of $I^j_C(k)$ and $(I^j_C \setminus \{C\})(k)$.

Lemma 5.15. *For all $C \in I$, $k, j \in [n]$, $j \geq k$ and $a : V_{>k} \cap \mathsf{var}(I^j_C) \to D$,*

$$w(I^j_C(k), a) = w(I^j_C, a).$$

Proof. The proof is by induction on k, the equality being trivial if $k = 0$. Now let $k \geq 0$, $C \in I$, $j \geq k+1$ and $a : V_{>k+1} \cap \mathsf{var}(I^j_C) \to D$. Observe that $\mathsf{var}(I(k+1)) = V_{>k+1}$ since $I(k+1)$ is the instance obtained after the deletion of $\{x_1, \ldots, x_{k+1}\}$. Thus, a assigns every variable of $I^j_C(k+1)$. It follows:

$$\begin{aligned} w(I^j_C(k+1), a) &= \prod_{K \in I^j_C(k+1)} K(a) \\ &= \prod_{K \in I^j_C} K^{(k+1)}(a) \text{ by definition of } I^j_C(k+1) \end{aligned}$$

Only the constraints K such that $x_{k+1} \in \mathsf{var}(K)$ are updated when x_{k+1} is removed. Thus, if $x_{k+1} \notin \mathsf{var}(K)$, it holds $K^{(k+1)} = K^{(k)}$. We denote by

$J = \{K \in I_C^j \mid x_{k+1} \in \mathsf{var}(K)\}$. It follows:

$$w(I_C^j(k+1), a) = \prod_{K \in I_C^j \setminus J} K^{(k)}(a) \cdot \prod_{K \in J} K^{(k+1)}(a). \tag{5.3}$$

If $J = \emptyset$, that is $x_{k+1} \notin \mathsf{var}(I_C^j)$, we have:

$$\begin{aligned} w(I_C^j(k+1), a) &= \prod_{K \in I_C^j} K^{(k)}(a) \\ &= w(I_C^j(k), a) \quad \text{since } a \text{ assigns all variables of } I_C^j(k) \\ &= w(I_C^j, a) \quad \text{by induction.} \end{aligned}$$

Now assume $J \neq \emptyset$. We follow the notations of Proposition 5.11 and denote $I(x_{k+1}) = \{C_1, \ldots, C_p\}$ with $C_1 \leq_{\mathcal{H}} \ldots \leq_{\mathcal{H}} C_p$. Let r be such that $C_r = \max_{\leq_{\mathcal{H}}} J$. For all $i \leq r$, $C_i \in J$ by Lemma 4.9. That is $J = \{C_1, \ldots, C_r\}$. Moreover, it is shown in the proof of Proposition 5.11 that:

$$\prod_{i=1}^{r} C_i^{(k+1)}(a) = \sum_{d \in D} \prod_{i=1}^{r} C_i^{(k)}(a \cup \{x_{k+1} \mapsto d\}).$$

Since $J = \{C_1, \ldots, C_r\}$, we have:

$$\begin{aligned} \prod_{K \in J} K^{(k+1)}(a) &= \prod_{i=1}^{r} C_i^{(k+1)}(a) \\ &= \sum_{d \in D} \prod_{K \in J} K^{(k)}(a \cup \{x_{k+1} \mapsto d\}). \end{aligned}$$

The equality 5.3 can thus be rewritten:

$$\begin{aligned} w(I_C^j(k+1), a) &= \prod_{K \in I_C^j \setminus J} K^{(k)}(a) \cdot \sum_{d \in D} \prod_{K \in J} K^{(k)}(a \cup \{x_{k+1} \mapsto d\}) \\ &= \sum_{d \in D} \prod_{K \in I_C^j} K^{(k)}(a \cup \{x_{k+1} \mapsto d\}) \\ &= \sum_{d \in D} w(I_C^j(k), a \cup \{x_{k+1} \mapsto d\}) \end{aligned}$$

Since $j \geq k+1$, we also have $j \geq k$, and $a \cup \{x_{k+1} \mapsto d\}$ assigns variables in $V_{>k} \cap \mathsf{var}(I_C^j(k))$. Thus, by induction, for all $d \in D$,

$$w(I_C^j(k), a \cup \{x_{k+1} \mapsto d\}) = w(I_C^j, a \cup \{x_{k+1} \mapsto d\}),$$

that is:

$$\begin{aligned} w(I_C^j(k+1), a) &= \sum_{d \in D} w(I_C^j, a \cup \{x_{k+1} \mapsto d\}) \\ &= w(I_C^j, a) \end{aligned}$$

which concludes the induction and the proof. □

Before proving Lemma 5.13, we need a last lemma showing that the weights of sub-instances are preserved during the elimination process:

Lemma 5.16. *Let $k \leq n$ and $a : V_{>k} \cap \mathsf{var}(I_C^k) \to D$. We denote by $J = I_C^k \setminus \{C\}$. It holds that*

$$w(J(k), a) = w(J, a).$$

Proof. The main argument of the proof is to decompose J into disjoint instances of type I_K^k for some constraints K and use Lemma 5.15 on each sub-instance. By Lemma 4.14, there exists $U \subseteq J$ such that:

$$J = \biguplus_{K \in U} I_K^k$$

and for all $K, K' \in U$, if $K \neq K'$ then $\mathsf{var}(I_K^k) \cap \mathsf{var}(I_{K'}^k) \subseteq V_{>k}$.

Thus J and $J(k)$ are both the disjoint union of sub-instances whose common variables are assigned by a, that is:

$$w(J(k), a) = \prod_{K \in U} w(I_K^k(k), a)$$

and

$$w(J, a) = \prod_{K \in U} w(I_K^k, a).$$

It follows that:

$$\begin{aligned} w(J(k), a) &= \prod_{K \in U} w(I_K^k(k), a) \\ &= \prod_{K \in U} w(I_K^k, a) \text{ by Lemma 5.15} \\ &= w(J, a). \end{aligned}$$

□

Proof (of Lemma 5.13). Let $C \in I$, $k \geq 1$ and $a \in \mathsf{supp}(C^{(k)})$. To ease notation, we denote by $J = I_C^k \setminus \{C\}$. By Lemma 4.12, $\mathsf{var}(I_C^k) \subseteq \mathsf{var}(C) \cup V_{\geq k}$. In other words, $\mathsf{var}(C^{(k)}) = \mathsf{var}(C) \cap V_{>k} = V_{>k} \cap \mathsf{var}(I_C^k)$.

Thus,

$$\begin{aligned} w(I_C^k(k), a) &= \prod_{K \in I_C^k(k)} K(a) \\ &= C^{(k)}(a) \cdot \prod_{K \in J(k)} K(a) \\ &= C^{(k)}(a) \cdot w(J(k), a). \end{aligned}$$

Now apply Lemma 5.15 on the left term and Lemma 5.16 on the right term yields:

$$w(I_C^k) = C^{(k)}(a) \cdot w(I_C^k \setminus \{C\}, a),$$

which is what we wanted to show. □

Lemma 5.13 guarantees that every arithmetic operations that are done during the elimination process are done on poly size rational numbers. Thus, we have the following:

Theorem 5.17. *Given a β-acyclic $\#\mathrm{CSP}_{\mathrm{def}}$-instance I with n variables on domain D and an β-elimination order of $\mathcal{H}(I)$, we can compute $w(I)$ with $O(|D| \cdot s(I) \cdot \|I\|)$ arithmetic operations on rational numbers of size $O(n \log(D) \cdot s \cdot \|I\|)$ where s is the size needed to encode the initial weights in I.*

In the particular case of counting the number of satisfying assignments, the weight of the sub-instances is bounded by the number of possible assignments, that is, by D^n where n is the number of variables. This gives:

Theorem 5.18. *Given a β-acyclic CSP on domain D, we can count its number of satisfying assignments with a $O(|D| \cdot s(I) \cdot \|I\|)$ arithmetic operations on rational numbers of size $O(n \log(D))$.*

5.3 Weighted DP-resolution on general instances

In this section, we generalize the previous algorithm for β-acyclic instances to make it work on any $\#\mathrm{CSP}_{\mathrm{def}}$ instances and analyze it. As before, each step of the algorithm consists of removing a variable x and updating the weights of constraints. The novelty here is that we have to introduce new constraints with well-chosen weights in order to preserve the weight of the instance. This was implicitly done on β-acyclic instances but as for resolution, we merge clauses that are included in one another which prevent new clauses to be created.

We start by describing the algorithm. We then show how it could be applied to reprove tractability results for $\#\mathsf{SAT}$. Finally, we introduce and study a new hypergraph measure, the cover-width, and study its relations with other measures. We finally show that weighted DP-resolution runs in polynomial time on instances of bounded cover-width if the arithmetic operations are assumed to be done in polynomial time.

5.3.1 Description of the algorithm

Let I be an instance of $\#\mathrm{CSP}_{\mathrm{def}}$ on domain D and let $x \in \mathsf{var}(I)$. We denote by $I_x = \{C \in I \mid x \in \mathsf{var}(C)\}$ the set of constraint holding variable x. Our goal is to find a procedure to transform I into a new instance I' with $\mathsf{var}(I') = \mathsf{var}(I) \setminus \{x\}$ and such that $w(I)$ can be easily deduced from $w(I')$.

We denote by $L(I_x) = \{\mathsf{var}(J) \mid J \subseteq I_x\}$ the lattice induced by the union of $\mathsf{var}(C)$ for $C \in I_x$. For $W \in L(I_x)$, we call a set $\mathcal{C} \subseteq I_x$ of constraints a *generator* of W if $\mathsf{var}(\mathcal{C}) = W$. We denote by $\mathsf{gen}(W) = \{\mathcal{C} \subseteq I_x \mid \mathsf{var}(\mathcal{C}) = W\}$ the set of generators of W. In what follows, to ease subscripts notations, we assume that C ranges over I_x only. For each $W \in L(I_x)$, we define a weighted constraint with default value W^* which is intuitively the fusion of every constraint having their variables in W:

- $\mathsf{var}(W^*) = W \setminus \{x\}$
- $\mathsf{def}(W^*) = |D| \cdot \prod_{\substack{C \in I_x \\ \mathsf{var}(C)=W}} \mathsf{def}(C)$
- $\mathsf{supp}(W^*) = \bigcup_{\mathcal{C} \in \mathsf{gen}(W)} \{a : W \setminus \{x\} \to D \mid \forall C \in \mathcal{C}, \exists b \in \mathsf{supp}(C), a \simeq b\}$, that is, $\mathsf{supp}(W^*)$ is the union of the join of every subset of constraints $\mathcal{C}$ such that $\mathsf{var}(\mathcal{C}) = W$,
- for all $a \in \mathsf{supp}(W^*)$,
  - if $\prod_{\substack{Z \in L(I_x) \\ Z \subsetneq W}} Z^*(a) \neq 0$ then
$$W^*(a) = \frac{\sum_{d \in D} \prod_{C \in I_x, \mathsf{var}(C) \subseteq W} C(a \cup \{x \mapsto d\})}{\prod_{Z \in L(I_x), Z \subsetneq W} Z^*(a)}$$
  - otherwise $W^*(a) = 0$.

The weighted resolvent of I by a variable $x \in \mathsf{var}(I)$ is the instance on domain D and variables $\mathsf{var}(I) \setminus \{x\}$ defined as

$$\mathsf{wres}(I, x) = (I \setminus I_x) \cup \bigcup_{W \in L(I_x)} W^*.$$

The main motivation of this definition is the following, which can be seen as a generalization of Lemma 5.1:

Theorem 5.19. *Let I be an instance of $\#\mathrm{CSP}_{\mathrm{def}}$ and $x \in \mathsf{var}(I)$. Then*

$$w(I) = w(\mathsf{wres}(I, x)).$$

Before proving Theorem 5.19, we make some handful remarks and present examples. In the following, we fix an instance I of $\#\mathrm{CSP}_{\mathrm{def}}$ and $x \in \mathsf{var}(I)$. We start by giving a better explanation of $\mathsf{supp}(W^*)$ for $W \in L(I_x)$.

Lemma 5.20. *Let $W \in L(I_x)$ and $a : W \setminus \{x\} \to D$. The assignment $a \in \mathsf{supp}(W^*)$ if and only if there exists $\mathcal{C}_0 \in \mathsf{gen}(W)$ such that for every $C \in \mathcal{C}_0$, there exists $d \in D$ such that $a|_{\mathsf{var}(C)} \cup \{x \mapsto d\} \in \mathsf{supp}(C)$.*

Proof. It is only a rewriting of the definition of $\mathsf{supp}(W^*)$. □

Lemma 5.20 yields a useful observation:

Corollary 5.21. *Let $C \in I_x$, $W = \mathsf{var}(C)$ and $a : W \setminus \{x\} \to D$. If $a \notin \mathsf{supp}(W^*)$ then for every $d \in D$, $C(a \cup \{x \mapsto d\}) = \mathsf{def}(C)$.*

Proof. It is a direct consequence of Lemma 5.20 since $\{C\}$ is a generator of W. □

C_1			
X	X_1	X_2	w
0	0	0	1
1	0	0	2
1	1	1	3
otherwise			4

C_2			
X	X_2	X_3	w
0	0	0	4
1	0	1	5
otherwise			0

C_3			
X	X_3	X_4	w
0	0	0	0
1	1	1	3
otherwise			1

C_4			
X	X_1	X_4	w
0	0	0	0
1	1	1	0
otherwise			1

Figure 5.4: An instance I of $\#\mathrm{CSP}_{\mathrm{def}}$ on domain $D = \{0, 1\}$.

An Example. We now give an example of the weighted resolvent of the instance I given in Figure 5.4. We want to compute $\mathsf{wres}(I, X)$. First observe that X appears in the four constraints $\{C_1, C_2, C_3, C_4\}$. However, even if there is $2^4 = 16$ subsets of $\{C_1, \ldots, C_4\}$, we see that $L(I_X)$ is much smaller since for example $\mathsf{var}(C_1) \cup \mathsf{var}(C_3) = \mathsf{var}(C_2) \cup \mathsf{var}(C_4) = \{X, \ldots, X_4\}$. Indeed, the lattice $L(I_X)$, depicted in Figure 5.5 has 9 points. Thus, in $\mathsf{wres}(I, X)$, we will have 9 constraints.

We give in Figure 5.6 the new tables for 3 of these constraints: $\{X_1, X_2\}^*$, $\{X_1, X_2, X_3\}^*$ and $\{X_1, X_2, X_3, X_4\}^*$. The only generator of $\{X, X_1, X_2\}$ is $\{C_1\}$. Thus the support of $\{X_1, X_2\}^*$ is actually the support of C_1 projected on $\{X_1, X_2\}$. Moreover, the default value of $\{X_1, X_2\}^*$ is $|D| = 2$ times the default value of C_1 since only $\mathsf{var}(C_1) = \{X, X_1, X_2\}$.

For $\{X, X_1, X_2, X_3\}$, we still have only one generator $\{C_1, C_2\}$. The only assignments of $\{X_1, X_2, X_3\}$ that can be extended to an element of $\mathsf{supp}(C_1)$ and to an element of $\mathsf{supp}(C_2)$ are $\{X_1 \mapsto 0, X_2 \mapsto 0, X_3 \mapsto 1\}$ and $\{X_1 \mapsto 0, X_2 \mapsto 0, X_3 \mapsto 0\}$ which gives the support of $\{X_1, X_2, X_3\}^*$. Moreover, since no constraint of I has variables $\{X, X_1, X_2, X_3\}$, the default value of $\{X_1, X_2, X_3\}^*$ is 1.

The case of $\{X_1, X_2, X_3, X_4\}^*$ is more delicate since it has many generators. Indeed, $\{C_1, C_3\}$, $\{C_2, C_4\}$, $\{C_1, C_2, C_3\}$, $\{C_1, C_2, C_4\}$, $\{C_2, C_3, C_4\}$ and $\{\mathsf{C}_1, C_3, C_4\}$ and $\{C_1, C_2, C_3, C_4\}$ are all generators of $\{X, X_1, X_2, X_3, X_4\}$. To construct the support of $\{X_1, \ldots, X_4\}^*$ however, it is sufficient to look at the generators $\{C_1, C_3\}$ and $\{C_2, C_4\}$ since for every other generator, one of them is included in it. It is interesting to observe that all weights in the support of $\{X_1, \ldots, X_4\}^*$ are null. Indeed, it can be observed that if a is an assignment of $\{X_1, \ldots, X_4\}$ and if $a(X_3) = a(X_4) = 0$ then $C_3(a) = C_4(a) = 0$, leading to a 0 weight in $\{X_1, \ldots, X_4\}^*$. Similarly if $a(X_2) = a(X_3) = 1$ then $C_2(a)$ takes its default value, 0. Finally, if $a(X_1) = a(X_4) = 1$, then $C_4(a) = 0$.

We now prove Theorem 5.19 by induction:

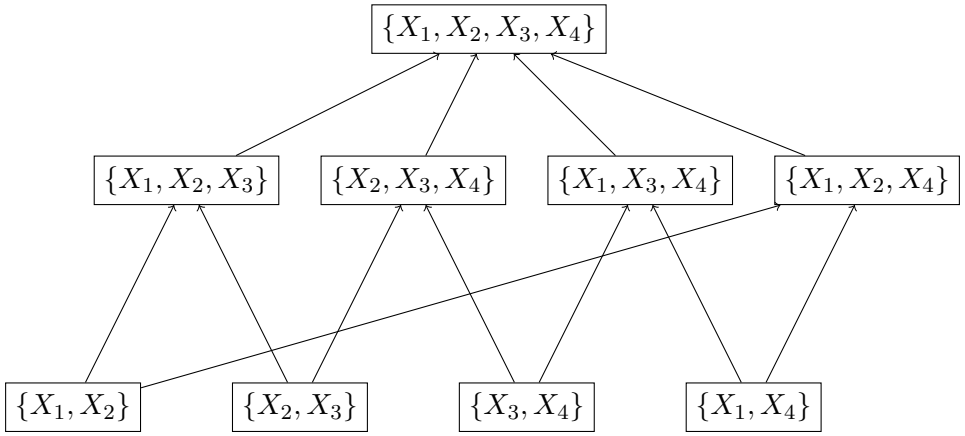

Figure 5.5: $L(I_X)$ where I is the instance of Figure 5.4.

$\{X_1, X_2\}^*$		
X_1	X_2	w
0	0	1+2=3
1	1	3+4=7
otherwise		8

$\{X_1, X_2, X_3\}^*$			
X_1	X_2	X_3	w
0	0	0	$(1 \cdot 4 + 2 \cdot 0)/(3 \cdot 4) = 1/3$
0	0	1	$(1 \cdot 0 + 2 \cdot 5)/(3 \cdot 5) = 2/3$
otherwise			1

$\{X_1, X_2, X_3, X_4\}^*$				
X_1	X_2	X_3	X_4	w
0	0	0	0	0
0	0	1	1	0
1	0	1	1	0
1	1	1	1	0
otherwise				1

Figure 5.6: Some constraints of $\mathsf{wres}(I, X)$ for the instance I of Figure 5.4.

Proof (of Theorem 5.19). Let $W \in L(I_x)$. We prove by induction on $|\{Z \in L(I_x) \mid Z \subseteq W\}|$ that for all $a : W \setminus \{x\} \to D$,

$$\prod_{\substack{Z \subseteq W \\ Z \in L(I_x)}} Z^*(a) = \sum_{d \in D} \prod_{\substack{C \in I_x \\ \mathsf{var}(C) \subseteq W}} C(a \cup \{x \to d\}).$$

Base case. Let $W \in L(I_x)$ such that $|\{Z \in L(I_x) \mid Z \subseteq W\}| = 1$ that is $\{Z \in L(I_x) \mid Z \subseteq W\} = \{W\}$. Thus for all $a : W \setminus \{x\} \to D$,

$$\prod_{Z \subseteq W, Z \in L(I_x)} Z^*(a) = W^*(a).$$

If $a \in \mathsf{supp}(W^*)$, then by definition

$$W^*(a) = \sum_{d \in D} \prod_{\mathsf{var}(C) \subseteq W} C(a \cup \{x \to d\}).$$

Now, observe that since $\{Z \in L(I_x) \mid Z \subseteq W\} = \{W\}$, it means in particular that there is no constraint $C \in I_x$ such that $\mathsf{var}(C) \subsetneq W$. Thus, the only generators of W are $\{C \in I_x \mid \mathsf{var}(C) = W\}$. If $a \notin \mathsf{supp}(W^*)$ then for every $C \in I_x$ such that $\mathsf{var}(C) = W$ and for every $d \in D$, $C(a \cup \{x \mapsto d\}) = \mathsf{def}(C)$ by Corollary 5.21. In other words,

$$\begin{aligned}\sum_{d \in D} \prod_{\mathsf{var}(C) \subseteq W} C(a \cup \{x \mapsto d\}) &= |D| \prod_{\mathsf{var}(C) = W} \mathsf{def}(C) \\ &= W^*(a) \text{ by definition of } \mathsf{def}(W).\end{aligned}$$

Induction step. Let $W \in L(I_x)$ such that $|\{Z \in L(I_x) \mid Z \subseteq W\}| = k + 1$ and assume the induction hypothesis holds for every $l \leq k$. Let $a : W \setminus \{x\} \to D$ such that $a \in \mathsf{supp}(W)$. Assume first that $\prod_{Z \in L(I_x), Z \subsetneq W} Z^*(a) \neq 0$. Then by definition of W^*, it holds that:

$$\prod_{\substack{Z \in L(I_x) \\ Z \subseteq W}} Z^*(a) = W^*(a) \prod_{\substack{Z \in L(I_x) \\ Z \subsetneq W}} Z^*(a) = \sum_{d \in D} \prod_{\substack{C \in I_x \\ \mathsf{var}(C) \subseteq W}} C(a \cup \{x \to d\}).$$

Now, assume that $\prod_{Z \in L(I_x), Z \subsetneq W} Z^*(a) = 0$. In particular, there exists $Z_0 \subsetneq W$ such that $Z_0^*(a) = 0$. By induction,

$$\prod_{\substack{Z \in L(I_x) \\ Z \subseteq Z_0}} Z^*(a) = \sum_{d \in D} \prod_{\substack{C \in I_x \\ \mathsf{var}(C) \subseteq Z_0}} C(a \cup \{x \mapsto d\}) = 0.$$

This sum is a sum of positive rational numbers evaluating to 0. Thus each term of the sum is 0. It implies that

$$\sum_{d \in D} \prod_{\mathsf{var}(C) \subseteq W} C(a \cup \{x \mapsto d\}) = 0.$$

That is

$$\prod_{Z\in L(I_x)Z\subseteq W} Z^*(a) = 0 = \sum_{d\in D} \prod_{\substack{C\in I_x \\ \mathsf{var}(C)\subseteq W}} C(a\cup\{x\to d\}).$$

Finally assume that $a \notin \mathsf{supp}(W^*)$. Let

$$\mathcal{C}_0 = \{C \in I_x \mid \mathsf{var}(C) \subseteq W \text{ and } \exists d \in D, a|_{\mathsf{var}(C)} \cup \{x \mapsto d\} \in \mathsf{supp}(C)\}.$$

Let $Z_0 = \mathsf{var}(\mathcal{C}_0)$. By Lemma 5.20, $\mathcal{C}_0$ is not a generator of W since $a \notin \mathsf{supp}(W^*)$, that is $Z_0 \subsetneq W$. Thus, by induction, we have:

$$\prod_{Z\subseteq Z_0} Z^*(a) = \sum_{d\in D} \prod_{\mathsf{var}(C)\subseteq Z_0} C(a\cup\{x\mapsto d\}).$$

Now let $Z \in L(I_x)$ such that $Z \subseteq W$ but $Z \not\subseteq Z_0$. If $a \in \mathsf{supp}(Z)$ then we would have $Z \subseteq Z_0$ by Lemma 5.20. Thus $a \notin \mathsf{supp}(Z^*)$, that is, $Z^*(a) = \mathsf{def}(Z^*) = \prod_{\mathsf{var}(C)=Z} \mathsf{def}(C)$. By Corollary 5.21, it even holds that for every $d \in D$,

$$\prod_{\mathsf{var}(C)=Z} \mathsf{def}(C) = \prod_{\mathsf{var}(C)=Z} C(a\cup\{x\mapsto d\}).$$

It follows that:

$$\begin{aligned}
\prod_{Z\subseteq W} Z^*(a) &= \prod_{Z\not\subseteq Z_0} \mathsf{def}(Z^*) \sum_{d\in D} \prod_{\mathsf{var}(C)\subseteq Z_0} C(a\cup\{x\mapsto d\}) \\
&= \sum_{d\in D} \prod_{Z\not\subseteq Z_0} \mathsf{def}(Z^*) \prod_{\mathsf{var}(C)\subseteq Z_0} C(a\cup\{x\mapsto d\}) \\
&= \sum_{d\in D} \prod_{\mathsf{var}(C)\not\subseteq Z_0} C(a\cup\{x\mapsto d\}) \prod_{\mathsf{var}(C)\subseteq Z_0} C(a\cup\{x\mapsto d\}) \\
&= \sum_{d\in D} \prod_{\mathsf{var}(C)\subseteq W} C(a\cup\{x\mapsto d\})
\end{aligned}$$

which finishes the induction.

The theorem follows from:

$$\begin{aligned}
w(I) &= \sum_{a:\mathsf{var}(I)\to D} \prod_{C\in I_x} C(a) \prod_{C\in I\setminus I_x} C(a) \\
&= \sum_{a:\mathsf{var}(I)\setminus\{x\}\to D} (\sum_{x\in D} \prod_{C\in I_x} C(a\cup\{x\mapsto d\})) \prod_{C\in I\setminus I_x} C(a) \\
&= \sum_{a:\mathsf{var}(I)\setminus\{x\}\to D} \prod_{Z\in L(I_x)} Z^*(a) \prod_{C\in I\setminus I_x} C(a) \\
&= w(\mathsf{wres}(I,x)).
\end{aligned}$$

□

Theorem 5.19 suggests an algorithm for $\#\mathrm{CSP}_{\mathrm{def}}$ similar to Algorithm 6: choose a variable $x \in \mathsf{var}(I)$ and compute $\mathsf{wres}(I, x)$. Repeat this operation until there is no more variable left. The resulting instance J has only constraints with no variable which can be seen as scalars, thus $w(J) = \prod_{C \in J} C(\emptyset) = w(I)$. This is illustrated in Algorithm 9.

Algorithm 9: An algorithm CSPwres for $\#\mathrm{CSP}_{\mathrm{def}}$ using weighted DP-resolution

Data: A set I of weighted constraints with default value and an order $\{x_1, \ldots, x_n\}$ on $\mathsf{var}(I)$

begin

  if $\mathsf{var}(I)$ *is empty* **then**

    return $\prod_{c \in I} c(\emptyset)$

  else

    $I' \leftarrow \mathsf{wres}(I, x_1)$;

    return $\mathsf{CSPwres}(I', \{x_2, \ldots, x_n\})$

Contrary to the classical resolution, $\mathsf{wres}(I, x)$ can already be of size exponential in $\|I\|$. For instance, if I is an instance with n constraints $C_1, \ldots, C_i$ with $\mathsf{var}(C_i) = \{x_0, x_i\}$, then for all $A \subseteq [n]$, $\{x_i \mid i \in A \cup \{0\}\} \in L(I_{x_0})$. Thus $\mathsf{wres}(I, x_0)$ has 2^n constraints. However, if I has n constraints $C_1, \ldots, C_n$ with $\mathsf{var}(C_i) = \{x_0, \ldots, x_i\}$, then $\mathsf{var}(C_1) \subseteq \ldots \subseteq \mathsf{var}(C_i)$. Thus, $L(I_{x_0}) = \{\{x_0, \ldots, x_i\} \mid i \leq n\}$ is of size n and then $\mathsf{wres}(I, x_0)$ has the same number of constraints as I.

The main challenge with weighted DP-resolution is thus to find classes of instances for which there exists an elimination order of the variables for which Algorithm 9 does not lead to a blow up in the size of the instance. For example, using weighted DP-resolution following a β-elimination order of the hypergraph of the instance does not lead to a blow-up: the number of constraints in the resolvent does not increase, nor the size of the constraints. This observation is explained in details in Section 5.3.2.

Computation of the weighted resolvent. In this paragraph, we make observations on how one can compute the weighted resolvent. In the following, I is a $\#\mathrm{CSP}_{\mathrm{def}}$ instance on domain D, $x \in \mathsf{var}(I)$ and we want to compute $\mathsf{wres}(I, x)$. From the definition alone, it is not clear how one can compute $\mathsf{wres}(I, x)$ without brute forcing on every subset $\mathcal{C}$ of I_x since for $W \in L(I_x)$, $\mathsf{supp}(W^*)$ is defined using the generators of W. Thus, even if we have a guarantee that $\mathsf{wres}(I, x)$ is small, the naive algorithm would still have a runtime exponential in $|I_x|$. We describe a procedure to compute $\mathsf{wres}(I, x)$ in polynomial time in $\|\mathsf{wres}(I, x)\| + \|I\|$ assuming that the arithmetic operations can be done in polynomial time. We get this result by explaining how we can efficiently compute $\mathsf{supp}(W^*)$ from $\mathsf{supp}(Z^*)$ for $Z, W \in L(I_x)$ and $Z \subsetneq W$. We rely on the notion of *primal generators*.

Let $W \in L(I_x)$. A *primal generator* $\mathcal{C}$ for W is a generator of W — that is, $\mathcal{C} \subseteq I_x$ and $\mathsf{var}(\mathcal{C}) = W$ — such that for every $C \in \mathcal{C}$, $\mathsf{var}(\mathcal{C} \setminus \{C\}) \neq W$. The set of primal generators of W is denoted by $\mathsf{pgen}(W)$. For example, in the instance of Figure 5.4, we had seen that $\{X_1, \ldots, X_4\}$ had a lot of generators but only $\{C_1, C_3\}$ and $\{C_2, C_4\}$ are primal generators. In this example, the number of prime generators is smaller than the total number of generator. We show that $\mathsf{supp}(W^*)$ can be defined by looking only at primal generators:

Lemma 5.22. *Let $W \in L(I_x)$ and let $\mathcal{C} \subseteq I_x$ be a generator of W. There exists a primal generator $\mathcal{C}_0$ of W such that $\mathcal{C}_0 \subseteq \mathcal{C}$.*

Proof. If $\mathcal{C}$ is not primal, then there exists $C \in \mathcal{C}$ such that $(\mathcal{C} \setminus \{C\})$ is still a generator of W. We remove such constraints until we have a primal generator of W. □

Proposition 5.23. *Let I be an instance of $\#\mathrm{CSP}_{\mathrm{def}}$ and $x \in \mathsf{var}(I)$. For every $W \in L(I_x)$, it holds that*

$$\mathit{supp}(W^*) = \bigcup_{\mathcal{C} \in \mathsf{pgen}(W)} \{a : W \setminus \{x\} \to D \mid \forall C \in \mathcal{C}, \exists b \in \mathit{supp}(C), a \simeq b\}.$$

Proof. The right-to-left inclusion is clear since $\mathsf{pgen}(W) \subseteq \mathsf{gen}(W)$. Now let $a \in \mathsf{supp}(W^*)$. By definition, there exists $\mathcal{C} \in \mathsf{gen}(W)$ such that $\forall C \in \mathcal{C}, \exists b \in \mathsf{supp}(C), a \simeq b$. By Lemma 5.22, there exists a primal generator $\mathcal{C}_0$ of W such that $\mathcal{C}_0 \subseteq \mathcal{C}$ and it clearly holds that $\forall C \in \mathcal{C}_0, \exists b \in \mathsf{supp}(C), a \simeq b$ which proves the left-to-right inclusion. □

Given A and B two subsets of assignments of variables, we denote by $A \bowtie B$ the join of A and B, that is, $A \bowtie B = \{a \cup b \mid a \in A, b \in B, a \simeq b\}$. It is easy to see that we can compute $A \bowtie B$ in polynomial time. The key result to allow efficient computation of $\mathsf{wres}(I, x)$ is the following:

Proposition 5.24. *Let I be an instance of $\#\mathrm{CSP}_{\mathrm{def}}$ and $x \in \mathsf{var}(I)$. For every $W \in L(I_x)$, it holds that*

$$\mathit{supp}(W^*) = \bigcup_{\substack{Z \in L(I_x) \\ Z \subsetneq W}} \bigcup_{\substack{C \in I_x \\ W = Z \cup \mathsf{var}(C)}} \mathit{supp}(Z^*) \bowtie \mathit{supp}(C \setminus \{x\}).$$

Proof. The right-to-left inclusion is easy to prove: let C be such that $Z \cup \mathsf{var}(C) = W$. Let $a \in \mathsf{supp}(Z^*) \bowtie \mathsf{supp}(C \setminus \{x\})$. Let $b \in \mathsf{supp}(Z^*)$ for $Z \subsetneq W$ such that $a \simeq b$ and $c \in \mathsf{supp}(C \setminus \{x\})$ such that $a \simeq c$. Let $\mathcal{C}$ be a generator of Z such that for every $C' \in \mathcal{C}$, there exists $b' \in \mathsf{supp}(C')$ such that $b' \simeq b$. Observe that $\mathcal{C} \cup \{C\}$ is a generator of W and that for every $C' \in \mathcal{C} \cup \{C\}$ there exists $b' \in \mathsf{supp}(C')$ such that $a \simeq b'$. Indeed, either C' is in $\mathcal{C}$ and b' is given by definition of $\mathcal{C}$. Or $C' = C$, and we choose $a = c$.

Now let $a \in \mathsf{supp}(W^*)$ and let $\mathcal{C}$ be a primal generator of W such that $\forall C \in \mathcal{C}, \exists b \in \mathsf{supp}(C), a \simeq b$ which exists by Proposition 5.23. Let C be any constraint in $\mathcal{C}$ and let $\mathcal{C}' = \mathcal{C} \setminus \{C\}$ and $Z = \mathsf{var}(\mathcal{C}')$. Since $\mathcal{C}$ is primal, $Z \subsetneq W$. It is easy to see that $a|_Z \in \mathsf{supp}(Z^*)$ and $a|_{\mathsf{var}(C)} \in \mathsf{supp}(C \setminus \{x\})$, that is, $a \in \mathsf{supp}(Z^*) \bowtie \mathsf{supp}(C \setminus \{x\})$. □

Proposition 5.24 suggests that we can compute $\mathsf{wres}(I, x)$ in time polynomial in $\|\mathsf{wres}(I, x)\|$ and $\|I\|$. We start by computing W^* for $W \in L(I_x)$ having no $Z \subsetneq W$. Then, we find an element W in $L(I_x)$ such that we have already computed Z^* for every Z that is strictly included in W. We can find such W by adding the variables of a constraint of I_x and choosing one that is minimal for inclusion. We now compute W^* using Proposition 5.24. This can be done in time polynomial in $\|\mathsf{wres}(I, x)\|$ and $\|I\|$ since for every $Z \subsetneq W$, $|\mathsf{supp}(Z^*)|$ is bounded by $\|\mathsf{wres}(I, x)\|$. This gives the following result.

Theorem 5.25. *Let I be a $\#\mathrm{CSP}_{\mathrm{def}}$ instance and $x \in \mathsf{var}(I)$. We can compute $\mathsf{wres}(I, x)$ in polynomial time in $\|\mathsf{wres}(I, x)\|$ and $\|I\|$ if the arithmetic operations are done in polynomial time.*

We believe that Theorem 5.25 could be improved a lot by using the right data structures to efficiently test inclusions and to quickly compute joins but it would require a better description of the exact encoding of $\#\mathrm{CSP}_{\mathrm{def}}$ instances. For now, we are only interested in using weighted DP-resolution to discover or rediscover tractable classes for #SAT so we are not interested in efficiency. Testing weighted DP-resolution in practice or as a precomputation phase to simplify the structure of a formula is however a perspective that we believe should be explored more precisely in practice. We let this work as an open question:

Open question 4. *How can we improve the computation time needed to compute the weighted resolvent in the general case?*

The case of bounded primal tree width. In this paragraph, we present a simple example where we can show that weighted DP-resolution runs in a polynomial number of arithmetic operations. For an instance I of $\#\mathrm{CSP}_{\mathrm{def}}$ on domain D, we define the primal graph of I, denoted by $\mathcal{G}_{\mathsf{prim}}(I)$, similarly as the primal graph of CNF-formulas: it is the graph whose vertices are $\mathsf{var}(I)$ and there is an edge between x and y in $\mathcal{G}_{\mathsf{prim}}(I)$ if there is a constraint $C \in I$ such that $\{x, y\} \subseteq \mathsf{var}(C)$. We show that if $\mathcal{G}_{\mathsf{prim}}(I)$ is of tree width k, then following an elimination order of $\mathcal{G}_{\mathsf{prim}}(I)$ of width k, weighted DP-resolution on I can be done with a polynomial number of arithmetic operations, which was already known for boolean domains [SS10] but using a more classical dynamic programming approach. Even if the result of [SS10] yields a much better runtime than ours, we use primal tree width as an example to illustrate how weighted DP-resolution can be used efficiently on some instances. This result is a generalization of Theorem 5.6.

Lemma 5.26. *Let I be an instance of $\#\mathrm{CSP}_{\mathrm{def}}$, $x \in \mathsf{var}(I)$ and let $G = \mathcal{G}_{\mathsf{prim}}(I)$. The primal graph of $\mathsf{wres}(I, x)$ is G/x.*

Proof. Let $\{y, z\}$ be an edge of the primal graph of $\mathsf{wres}(I, x)$. Either this edge is generated by a constraint $C \in I \setminus I_x$ and thus $\{y, z\}$ is also in G/x. Or it is generated by W^* for some $W \subseteq I_x$. In this case, it means that $\{x, y\}$ and $\{x, z\}$ were both edges in G, and thus, $\{y, z\}$ is in G/x.

Now let $\{y, z\}$ be an edge of G/x. If this edge is also in G, then it means that there exists a constraint $C \in I$ such that $\{y, z\} \subseteq \mathsf{var}(C)$. If $x \notin \mathsf{var}(C)$, then $C \in \mathsf{wres}(I, x)$. If $x \in \mathsf{var}(C)$, then $\{y, z\} \subseteq \mathsf{var}(C)^*$. In both case, $\{y, z\}$ is an edge of the primal graph of $\mathsf{wres}(I, x)$. If $\{y, z\}$ is not an edge of G, then it means that $\{x, y\}$ and $\{x, z\}$ were edges in G, that is, there exists $C_1, C_2 \in I_x$ such that $\{x, y\} \subseteq \mathsf{var}(C_1)$ and $\{x, z\} \subseteq \mathsf{var}(C_2)$. Thus $\{y, z\} \subseteq W^* \in \mathsf{wres}(I, x)$ where $W = \mathsf{var}(C_1) \cup \mathsf{var}(C_2)$. That is, $\{y, z\}$ is an edge of the primal graph of $\mathsf{wres}(I, x)$. □

Lemma 5.27. *Let I be an instance of $\#\mathrm{CSP}_{\mathrm{def}}$ on domain D and $x \in \mathsf{var}(I)$ such that the degree of x in $\mathcal{G}_{\mathsf{prim}}(I)$ is k. Let $I' = \mathsf{wres}(I, x)$. It holds that $\|I'\| \leq (|D| + 1)^k + \|I\|$.*

Proof. Let $Y = \{y_1, \ldots, y_k\}$ be the neighbors of x in $G = \mathcal{G}_{\mathsf{prim}}(I)$. By definition, for every $C \in I_x$, it holds that $\mathsf{var}(C) \subseteq \{x\} \cup Y$. Thus, for every $\mathcal{C} \subseteq I_x$, $\mathsf{var}(\mathcal{C}) \subseteq \{x\} \cup Y$ and then there are at most 2^k new constraints in $\mathsf{wres}(I)$. Now, let W^* be a new constraint in I'. It holds that $W \subseteq Y \cup \{x\}$. Moreover W^* has arity $|W| - 1 = i \leq k$ and thus $|\mathsf{supp}(W^*)| \leq |D|^i$. Thus, the size of the new clauses is at most:

$$\sum_{W \subseteq Y} |D|^{|W|} = \sum_{i=0}^{k} \binom{n}{i} |D|^i = (|D| + 1)^k.$$

In other words, $\|I'\| \leq \|I\| + (|D| + 1)^k$. □

Following an elimination order of width k and using Lemma 5.26 and Lemma 5.27 together with Theorem 5.25 yields the following theorem, which can be seen as a generalization of Theorem 5.6:

Theorem 5.28. *Given an instance I of $\#\mathrm{CSP}_{\mathrm{def}}$ on domain D with n variables and primal tree width k, one can compute $w(I)$ in time polynomial in $(|D| + 1)^k$ and $\|I\|$ if the arithmetic operations are done in polynomial time.*

Proof. By following an elimination order of width k, it follows from Lemma 5.27 and Lemma 5.26 that during the elimination process, the size of the instance is always smaller than $n(|D| + 1)^k + \|I\|$. We use Theorem 5.25 to conclude. □

Theorem 5.28 does not however state that weighted DP-resolution runs in polynomial time on instances of bounded primal tree width. Indeed, as in Theorem 5.12, Theorem 5.28 does not ensure that the weights computed during the

elimination procedure can be encoded in polynomial bit size. To prove tractability results using weighted DP-resolution would require a finer analysis of the algorithm as it was conducted in Section 5.2.4. Our understanding of weighted resolution is however not mature enough to generalize the already technical proof of Section 5.2.4. We left this study as an open question.

Open question 5. *Is the bit size of the weights computed during weighted DP-elimination polynomially bounded?*

5.3.2 Cover-width

In this section, we generalize the elimination order of width k characterizing the tree width to the hypergraph setting. We introduce the *cover-width* of a hypergraph $\mathcal{H}$ to be the minimal k such that $\mathcal{H}$ has such an elimination order of width k. We show that cover-width one corresponds to β-acyclicity and that the cover-width of a hypergraph is bigger than its β-hyper tree width in general. Finally, we show that weighted DP-resolution can be done with a polynomial time number of arithmetic operations following an elimination order of constant width k. We finally conclude this chapter by presenting open questions and perspective concerning weighted DP-resolution.

We start by defining the notion of covering we need. This corresponds to the cover-value defined in Section 2.3.1 in the setting of hypergraphs.

Definition 5.29. *A hypergraph $\mathcal{H}$ is k covered if and only if for every $\mathcal{H}' \subseteq \mathcal{H}$, there exists $\mathcal{H}'' \subseteq \mathcal{H}'$ such that $|\mathcal{H}''| \leq k$ and $V(\mathcal{H}'') = V(\mathcal{H}')$. The cover-value of a hypergraph $\mathcal{H}$, denoted by $cv(\mathcal{H})$, is the minimal k such that $\mathcal{H}$ is k-covered.*

The cover-width is defined using an elimination order where we bound the cover-value of subhypergraphs at each step of the elimination procedure.

Definition 5.30. *Let $\mathcal{H}$ be a hypergraph and $x_1, \ldots, x_n$ be an ordering of its vertices. The ordering $x_1, \ldots, x_n$ is an* elimination order of cover-width at most k *for $\mathcal{H}$ if for every $i \leq n$, $\mathcal{H}^i[x_i, \ldots, x_n]$ is k-covered, where $\mathcal{H}^i$ contains exactly the edges of $\mathcal{H}$ that can be reach from x_i by going through $x_1, \ldots, x_i$. The* cover-width *of $\mathcal{H}$, denoted by $\mathbf{covw}(\mathcal{H})$, is the minimal k such that there exists an elimination order of $\mathcal{H}$ of cover-width at most k.*

Observe that the cover-width of a hypergraph is bounded by it number of vertices.

Proposition 5.31. *For every hypergraph $\mathcal{H}$, $\mathbf{covw}(\mathcal{H}) \leq |V(\mathcal{H})|$.*

Proof. We show that, actually, the cover-value of a hypergraph is bounded by its number of vertices. Indeed, let $\mathcal{H}$ be a hypergraph and let $\mathcal{H}' \subseteq \mathcal{H}$. Let $\mathcal{H}'' \subseteq \mathcal{H}'$ is such that $V(\mathcal{H}'') = V(\mathcal{H}')$ and $\mathcal{H}''$ is of minimal size. By definition, for every $e \in \mathcal{H}''$, $V(\mathcal{H}'' \setminus \{e\}) \neq V(\mathcal{H}')$, otherwise, we would have a smaller subhypergraph of $\mathcal{H}'$ satisfying the property. Thus for every $e \in \mathcal{H}''$, there exists $x_e \in e$ such that for $e' \in \mathcal{H}'' \setminus \{e\}$, $x_e \notin e'$. It follows that $|\mathcal{H}''| \leq |V(\mathcal{H})|$ and then the cover-value of $\mathcal{H}$ is smaller than $|V(\mathcal{H})|$. □

Cover-width and other measures. In this paragraph, we study how cover-width is related to other measures. We start by showing that the cover-width of a graph is exactly its tree width and that elimination order of width k for graph and hypergraph actually defines the same notion on graphs.

Proposition 5.32. *Let $G = (V, E)$ be a graph and $x_1, \dots, x_n$ an ordering of V. The ordering $x_1, \dots, x_n$ is of width k if and only if this ordering is of cover-width k. In particular,* $\mathbf{covw}(G) = \mathbf{tw}(G)$.

Proof. As usual, we define $G_1 = G$ and $G_{i+1} = G_i/x_i$ for all $i < n$. We show that for $\{y, z\}$ is an edge of G_i if and only if there exists a path from y to z in the only goes through vertices in $\{y, z, x_1, \dots, x_{i-1}\}$ in G (for $y, z \notin \{x_1, \dots, x_{i-1}\}$).

The proof is by induction. If $i = 1$, then the result is trivial since there is an edge between y and z if and only there is a path in G that goes through vertices in $\{y, z\}$.

Now let $\{y, z\}$ be an edge in G_{i+1}. If $\{y, z\}$ is an edge of G_i also then by induction, there exists a path in G from y to z going through vertices in $\{y, z, x_1, \dots, x_{i-1}\} \subseteq \{y, z, x_1, \dots, x_i\}$. If $\{y, z\}$ is not an edge of G_i, then it means that $\{y, x_i\}$ and $\{z, x_i\}$ are edges of G_i by definition of G_i/x_i. By induction, there exists a path from y to x_i and a path from x_i to z in G that both go through vertices in $\{y, z, x_1, \dots, x_{i-1}\}$. The concatenation of these paths is a path from y to z going through vertices in $\{y, z, x_1, \dots, x_i\}$.

Now assume there exists a path from y to z going through vertices $\{y, z, x_1, \dots, x_i\}$. Either this path does not go through x_i and by induction, $\{y, z\}$ is an edge of G_i and thus of G_{i+1}. Or this path goes by x_i. But then we can cut this path into a path from y to x_i going through vertices in $\{y, x_i, x_1, \dots, x_{i-1}\}$ and a path from x_i to z going going through vertices in $\{z, x_i, x_1, \dots, x_{i-1}\}$. In particular, by induction, the existence of the first path implies that $\{y, x_i\}$ is an edge of G_i, and similarly, $\{z, x_i\}$ is an edge of G_i. This, $\{y, z\}$ is an edge of $G_{i+1} = G_i/x_i$.

To conclude, let $\mathcal{H}^i$ be the edges of G than can be reached from x_i by going through vertices in $\{x_1, \dots, x_i\}$. These are edges of the form $\{x_j, y\}$ for $j \leq i$ and then, we know from what precedes that there is an edge between x_i and y in G_i. That is, the neighborhood of x_i in G_i is exactly $V(\mathcal{H}^i) \setminus \{x_i\}$ and $\mathcal{H}^i[x_i, \dots, x_n]$ contains edges of the form $\{x_i, y\}$ or $\{y\}$ if y is not a neighbor of x_i in G. The cover-value of $\mathcal{H}^i$ is exactly the degree of x_i in G_i. □

An immediate corollary of Proposition 5.32 is that it is as hard to compute cover-width than tree width on graphs. Since computing the tree width of a graph is NP-hard by Theorem 1.25 [Bod93a], we have:

Theorem 5.33. *Given a hypergraph $\mathcal{H}$ and $k \in \mathbb{N}$, it is* NP*-hard to decide whether* $\mathbf{covw}(\mathcal{H}) \leq k$.

We now explore the relation between β-hypertree width and cover-width.

Proposition 5.34. *Let $\mathcal{H}$ be a hypergraph and $x_1, \ldots, x_n$ an ordering of its variables. The elimination order $x_1, \ldots, x_n$ is of cover-width 1 if and only if it is a β-elimination order. In particular, a hypergraph $\mathcal{H}$ is β-acyclic if and only if* $\mathbf{covw}(\mathcal{H}) = 1$.

Proof. Assume first that $x_1, \ldots, x_n$ is of cover-width 1. It is clear that every edge containing x_i can be reach from x_i by going only through x_i. Let e, f be two edges containing x_i. Since the cover-width of $x_1, \ldots, x_n$ is 1, then $e \cap \{x_i, \ldots, x_n\}$ and $f \cap \{x_i, \ldots, x_n\}$ are covered by one edge among them. In other word, $e \cap \{x_i, \ldots, x_n\} \subseteq f \cap \{x_i, \ldots, x_n\}$ or $f \cap \{x_i, \ldots, x_n\} \subseteq e \cap \{x_i, \ldots, x_n\}$, that is x_i is a β-leaf in $\mathcal{H}[x_i, \ldots, x_n]$.

Now assume that $x_1, \ldots, x_n$ is an β-elimination order and let e, f be an edge that can be reached from x_i by going through vertices in $\{x_1, \ldots, x_i\}$. We appeal to results and definitions of Chapter 4. Assume w.l.o.g that $e <_{\mathcal{H}} f$. Then by definition $e \in \mathcal{H}_f^{x_i}$ and then $e \cap \{x_i, \ldots, x_n\} \subseteq f$ by Lemma 4.12. □

Proposition 5.35. *For every hypergraph $\mathcal{H}$, it holds that* β-$\mathbf{htw}(\mathcal{H}) \leq 3 \cdot \mathbf{covw}(\mathcal{H}) + 1$.

Proposition 5.35 follows directly from the following two lemmas:

Lemma 5.36. *For every hypergraph $\mathcal{H}$, it holds that* $\mathbf{ghtw}(\mathcal{H}) \leq \mathbf{covw}(\mathcal{H})$.

Proof. Let $x_1, \ldots, x_n$ be an elimination order of width k of $\mathcal{H}$. We claim that $\mathcal{H}' = \mathcal{H} \cup \bigcup_{i=1}^{n} (V(\mathcal{H}_{x_i}) \cap \{x_j \mid j \geq i\})$ is α-acyclic. Indeed, $x_1, \ldots, x_n$ is an α-elimination order of $\mathcal{H}'$ since when we remove x_i, its neighborhood is covered by $V(\mathcal{H}_{x_i}) \cap \{x_j \mid j \geq i\}$. This is actually a general fact that do not depend on the width of the elimination order.

Now by Proposition 1.31, it follows that $\mathcal{H}'$ is α-acyclic. Thus there exists a join tree $\mathcal{T}$ for $\mathcal{H}'$. We claim that $\mathcal{T}$ is a hypertree decomposition of $\mathcal{H}$ of generalized hypertree width k. It is sufficient to show that each bag of $\mathcal{T}$ can be covered with at most k edges of $\mathcal{H}$. Since $\mathcal{T}$ is a join tree of $\mathcal{H}'$, its bags are labeled with edges of $\mathcal{H}'$ and thus, it is sufficient to show that each edge of $\mathcal{H}'$ can be covered with at most k edges of $\mathcal{H}$.

Let $e \in \mathcal{H}'$. If $e \in \mathcal{H}$, then e is covered by itself and then it is covered by one edge of $\mathcal{H}$. Otherwise, there exists i such that $e = V(\mathcal{H}_{x_i}) \cap \{x_j \mid j \geq i\}$. But every edges of $\mathcal{H}_{x_i}$ can be reached from x_i by going only through x_i. Thus $V(\mathcal{H}_{x_i}) \cap \{x_j \mid j \geq i\}$ is covered by k edges of $\mathcal{H}_{x_i}$ by definition of cover-width. □

Lemma 5.37. *For every hypergraph $\mathcal{H}$ and $\mathcal{H}_0 \subseteq \mathcal{H}$, it holds that* $\mathbf{covw}(\mathcal{H}_0) \leq \mathbf{covw}(\mathcal{H})$.

Proof. Let $x_1, \ldots, x_n$ be an elimination order of $\mathcal{H}$ of width k. We show that this order, restricted to the variables of $\mathcal{H}_0$, is also an elimination order of $\mathcal{H}_0$. We denote $\mathcal{H}^i$ the set of edges of $\mathcal{H}$ that can be reach from x_i by going through vertices in $\{x_1, \ldots, x_i\}$. We denote by ${\mathcal{H}_0}^i$ the set of edges of $\mathcal{H}_0$ that can be reach

from x_i by going through vertices in $\{x_1, \ldots, x_i\}$. It is easy to see that $\mathcal{H}_0^i \subseteq \mathcal{H}^i$. But then the cover-value of ${\mathcal{H}_0}^i$ is smaller than the cover-value of $\mathcal{H}^i$. Thus the elimination order $x_1, \ldots, x_n$ is also an elimination order for $\mathcal{H}_0$ of cover-width at most k. □

Proposition 5.35 shows that β-hypertree width is more general than cover-width, that is, if a class of hypergraph has bounded cover-width, then it has also bounded β-hypertree width. Unfortunately, we do not know if the cover width of a hypergraph can be polynomially bounded by its β-hypertree width.

Open question 6. *Find a polynomial P (or prove that such polynomial does not exist) such that:*

$$\mathbf{covw}(\mathcal{H}) \leq P(\beta\text{-}\mathbf{htw}(\mathcal{H})).$$

A positive answer to the previous question may provide new perspectives to prove the tractability of some problems, like SAT or #SAT, parametrized by β-hypertree width. It may also provide new tools to study β-hypertree width. Such characterizations may be a way of finding parametrized algorithm to compute the β-hypertree width of a given hypergraph. Unfortunately, the parametrized complexity of computing a good elimination order for a hypergraph is still open.

Open question 7. *Given a hypergraph $\mathcal{H}$ and an integer $k \in \mathbb{N}$, what is the parametrized complexity of computing an elimination order of $\mathcal{H}$ of width at most k when it exists and rejects otherwise?*

Weighted DP-resolution and cover-width. In this paragraph, we show that weighted resolution can be done on bounded cover-width instances without an exponential blow-up in the size of the instance if one follows an elimination order of width k. The main argument is that every constraint that has been introduced by weighted resolution after having eliminating x_i is actually the fusion of constraints that are reachable from one another by going through vertices in $x_1, \ldots, x_i$. By the cover-width definition, it follows that such constraints are covered by k original constraints and then we can use it to bound both the size of their support and the number of constraints that are introduced during weighted resolution.

In the following, we fix I an instance of $\#\mathrm{CSP}_{\mathrm{def}}$. We let $\mathcal{H} = \mathcal{H}(I)$ and $x_1, \ldots, x_n$ be an elimination order of $\mathcal{H}$ of cover-width k. For every $i \leq n$, we denote by $\mathcal{H}^i$ the set of edges of $\mathcal{H}$ that can be reached from x_i by going through vertices in $\{x_1, \ldots, x_i\}$. We denote by $I^1 = I$ and by $I^{i+1} = \mathsf{wres}(I^i, x_i)$.

Lemma 5.38. *Let $i \leq n$, $C \in I^i$ and $a \in \mathsf{supp}(C)$. There exists $C_1, \ldots, C_p \in I$ and $a_1 \in \mathsf{supp}(C_1), \ldots, a_p \in \mathsf{supp}(C_p)$ with $p \leq k$ such that:*

1. *$a \simeq a_j$ for all $j \leq p$ and,*
2. *$\mathsf{var}(C) \subseteq \bigcup_{j=1}^{p} \mathsf{var}(C_j)$.*

Proof. Let C be a constraint of I^i. We say that C was introduced at step j if C is a constraint of I^{j+1} and C is not a constraint of I^j. Constraints of $I = I^1$ have been introduced at step 0. We prove by induction on i that if a constraint C have been introduced at step i then for every $a \in \mathsf{supp}(C)$:

- there exist $C_1, \ldots, C_p \in I$ such that $\mathsf{var}(C) = \bigcup_{j=1}^{p} \mathsf{var}(C_j) \setminus \{x_1, \ldots, x_i\}$,
- for $j, j' \leq p$, there is a path from $\mathsf{var}(C_j)$ to $\mathsf{var}(C_{j'})$ in $\mathcal{H}$ that goes through $\{x_1, \ldots, x_i\}$ only,
- there exists $a_j \in \mathsf{supp}(C_j)$ such that $a \simeq a_j$ for every $j \leq p$.

The case where $i = 0$ is trivial since every constraint of I is covered by itself. Now let C be a constraint introduced at step i and $a \in \mathsf{supp}(C)$. By definition of weighted DP-resolution, there exists constraints $C'_1, \ldots, C'_r$ of I^i, all containing x_i, such that $\mathsf{var}(C) = (\mathsf{var}(C'_1) \cup \cdots \cup \mathsf{var}(C'_r)) \setminus \{x_i\}$ and for every j, there exists $a_j \in \mathsf{supp}(C'_j)$ such that $a_j \simeq a$. The constraint C'_j has been introduced before step i, thus by induction there exists $C_{j,1}, \ldots, C_{j,p_j}$ that respects the desired properties for C'_j. We choose $C_1, \ldots, C_p$ to be all these clauses. They satisfy the desired properties.

The statement of the lemma follows from the fact that since the elimination order is of cover-width k, given $a \in \mathsf{supp}(C)$ and the corresponding $C_1, \ldots, C_p$, we can extract k constraints from $C_1, \ldots, C_p$ that still covers $\mathsf{var}(C)$. □

Corollary 5.39. *Let I be a $\#\mathrm{CSP}_{\mathrm{def}}$ instance with m constraints and $x_1, \ldots, x_n$ an elimination order of $\mathcal{H}(I)$ of cover-width k. Let $I^1 = I$ and $I^{i+1} = \mathsf{wres}(I^i, x_i)$. For every $i \leq n$, $\|I^i\| \leq m^{k+1}\|I\|^k$.*

Proof. By Lemma 5.38, the variables of every constraint of I^i is covered with the variables of at most k constraints of I. Thus, there is are most $\sum_{j=1}^{k} \binom{m}{j} \leq m^{k+1}$ constraints in I^i and the support of each of them is obtained by joining tuples of at most k original constraints, that is, their support is of size at most $\|I\|^k$. □

Combining Corollary 5.39 with Theorem 5.25, it follows that weighted DP-resolution may be done in polynomial time if one follows an elimination order of cover-width k and if arithmetic operations can be done in polynomial time:

Theorem 5.40. *Given I a $\#\mathrm{CSP}_{\mathrm{def}}$ instance with m constraints and $x_1, \ldots, x_n$ an elimination order of $\mathcal{H}(I)$ of cover-width k, one can compute $w(I)$ in polynomial time in $m^k\|I\|^k$ assuming that arithmetic operations can always be done in polynomial time.*

5.4 Conclusion

There is still much to understand on weighted resolution. The main challenge is to bound the size of the weights that appear during the elimination procedure. A

reasonable goal would be to prove they are bounded when the cover-width of the instance is also bounded, as we did it in Section 5.2.3 for β-acyclic instance which corresponds to cover-width 1.

Our main purpose with weighted DP-resolution is to offer new perspectives for understanding the complexity of #SAT on bounded β-hypertree width formulas, the difficult task of characterizing the β-hypertree width better is still an interesting question and characterizations based on elimination order such as cover-width may be an interesting direction.

We have also seen in the previous chapters that the trace of every algorithm for #SAT can actually be seen as a compilation algorithm from CNF into d-DNNF. It is not clear however if this applies to weighted DP-resolution as well. Indeed, the algorithm for PS-width was not hard to generalize to a compilation algorithm since the algorithm count the number of satisfying assignments by multiplying the number of satisfying assignments of disjoint variables formulas or by adding the number of satisfying assignments of formulas having disjoint satisfying assignments. This could be almost directly translated into a compilation algorithm into d-DNNF by replacing multiplications by decomposable $\wedge$-nodes and additions by deterministic $\vee$-gate. Weighted resolution however relies on weights that are not even necessarely integers. It is not straightforward to see how divisions of weights could be translated into boolean circuits.

However, since weighted DP-resolution is working on an elimination order, it may suggests that it implicitly constructs a dec-DNNF, and that the weights are only an implicit way of encoding the shared part of an underlying boolean circuits. We thus let the question of turning weighted DP-resolution into a compilation algorithm as an open question:

Open question 8. *Can weighted resolution be turned into a compilation algorithm of* CNF*-formulas into* dec-DNNF*?*

Chapter 6

Unconditional separations

The previous chapters were mostly focused on proving positive algorithmic results. We have indeed designed specific polynomial time algorithms in order to discover tractable classes for knowledge compilation. This naturally raises the question of finding CNF-formulas for which such algorithmic techniques will always fail. Such question can be reformulated as lower bounds in knowledge compilation. Indeed, if a class of CNF-formulas cannot be represented succinctly by d-DNNF, then the structure-based algorithms for #SAT – and for compiling – presented in this thesis will fail. This was already observed by Huang and Darwiche [HD05] who argued that most of the practical tools for #SAT were implicitly compiling the formula into a dec-DNNF. Thus lower bounds on the size of dec-DNNF representing a family of formulas directly translate into lower bounds on the runtime of state-of-the-art tools for #SAT.

For a function $g : \mathbb{N} \to \mathbb{N}$, we say that we have an $g(n)$ lower bound on the compilation of a language L_1 into an other language L_2 if there exists a boolean function $f : \{0,1\}^* \to \{0,1\}$ such that the size of f represented in L_1 – denoted by $L_1(f)$ – is at least $O(g(L_2(f)))$. We say that L_1 and L_2 separated if we have a super polynomial lower bound on the compilation of L_1 into L_2. We often refer to strongly exponential lower bound as $2^{\Omega(n)}$ lower bound in contrast to weakly exponential lower bounds that are $2^{\Omega n^\alpha}$ lower bounds for $\alpha < 1$.

Few unconditional lower bounds are known for representation languages based on DNNF. Most separation of representation langages are known only under common hypothesis in complexity theory such as NP $\not\subseteq$ P/poly [KS96]. Most of these separations have been left open in the survey of Darwiche and Marquis [DM02] on the study of representation languages for knowledge compilation. Unconditional separation are not entirely satisfactory since they give no hint on the reasons why a function is hard to represent in a given language.

More is known however on simple languages. Lower bounds for FBDD and OBDD have been proven very early [Žák84, Weg88] (see also [Weg00] for an overview of these results) but the first lower bounds for DNNF based languages have been proven only very recently. Beame and al. have proven an unconditional

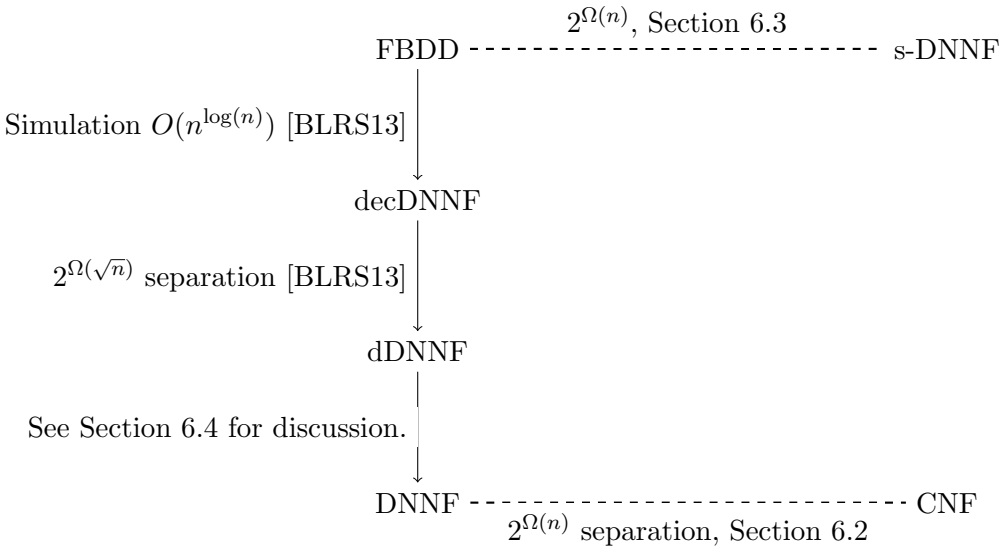

Figure 6.1: Overview of some known and new separations concerning DNNF and its restriction. An arrow represents the inclusion, a dash line the fact that the languages are incomparable. Here s-DNNF stands for structured DNNF.

separation of dec-DNNF and d-DNNF in [BLRS13] and an unconditional separation of dec-DNNF and DNF in [BLRS14] but even there, the separations are usually based on known lower bounds for FBDD that are then leveraged to restrictions of DNNF. Based on the result of [PD10b], Pipatsrisawat gives in his thesis [Pip10] lower bounds on the size of structured DNNF.

In this chapter, we establish a connection between the size of DNNF-based languages and the number of rectangles needed to cover their computed function. Rectangles is a notion that is well-studied in communication complexity and our technique can be used to leverage lower bounds from communication complexity into knowledge representation. Using this connection, we are able to provide the first strongly exponential separation of DNNF and CNF. This can be achieved by using known lower bound from communication complexity [JS02], but we also provide a self-contained proof of this fact that is shorter and more elementary than the proof of [JS02]. We also show that our connection with communication complexity may be used to characterized other restrictions of DNNF such as deterministic DNNF or structure DNNF. We use the last connection to rediscover results of Darwiche and Pipatsrisawat [PD10b] and prove a strongly exponential separation of structured DNNF and FBDD, which can be seen as a separation of every representation language with their structured equivalent. This fact was already proven in [Pip10] but our method is more generic and relies on basic tools of graph decompositions.

In the first section, we introduce essential notions to study DNNF and prove the connection with the size of a DNNF and the number of balanced rectangles needed

to cover it. In the second section, we use this connection to leverage known lower bounds in communication complexity and give a strongly exponential separation of 3-CNF and DNNF. We also present a self-contained proof of such lower bound that allows us to separated montone 2-CNF from DNNF. Finally, we show how the known lower bounds on structured DNNF from [PD10b] can be explained by our connection to communication complexity and we use these techniques to reprove lower bounds on the size of structured DNNF. We conclude this chapter by giving an outlined proof of a strongly exponential separation of DNNF and deterministic DNNF.

Figure 6.1 summarizes the contributions of this chapter and provides references for the already known separations concerning DNNF and its restrictions.

6.1 Preliminaries

In this section, we introduce the main tools needed to prove lower bounds. We start by defining the notion of certificates in DNNF that allows us to nicely operate on the set of satisfying assignments of a DNNF. We then define normal forms of DNNF and give algorithm to transform DNNF into such normal form with only a polynomial increase in size. Finally, we introduce the notion of boolean rectangle, a well-studied notion in communication complexity, used intensively for proving lower bounds in this area.

6.1.1 Certificates

Most of the notions introduced here are classical notions in arithmetic circuit complexity [Bü00, MP06] adapted to the framework of DNNF. We start by introducing *certificates*, that is the analogous notion of *parse trees* in arithmetic circuits. Certificates can be seen as a generalization to DNNF of the notion of paths in FBDD. A path in an FBDD is a witness of the fact that an assignment is a satisfying assignment of the FBDD. We use the certificate similarly in our proofs.

Definition 6.1. *Let D be a* DNNF *and let $s = \mathsf{output}(D)$. A* certificate *of D is a connected* DNNF *T, included in D such that:*

- $s \in T$
- *if $u \in T$ is an $\wedge$-gate then every child of u is in T*
- *if $u \in T$ is an $\vee$-gate then exactly one child of u is in T*

The set of certificates of D is denoted by $\mathsf{cert}(D)$. The set of certificates that contain a gate v is denoted by $\mathsf{cert}(D, v)$

Certificates are very useful since they have a very simple form and characterize entirely the function computed by a DNNF. One can observe the following:

Proposition 6.2. *Let D be a* DNNF *and $T \in \mathsf{cert}(D)$. Let v be a gate of T. Then $T_v \in \mathsf{cert}(D_v)$.*

Proof. By definition, $\mathsf{output}(D_v) = v$. Thus T_v is a set of gate of D_v containing v such that if u is an $\wedge$-gate of T_v, then every child of u is in T_v and if u is an $\vee$-gate of T_v, then exactly one child of u is in T_v. That is T_v is a certificate of D_v. □

Proposition 6.3. *Let D be a* DNNF *and $T \in \mathsf{cert}(D)$. The inputs of T are labeled by literals with pairwise distinct variables and the underlying graph of T is a tree rooted in* $\mathsf{output}(D)$*. Moreover*

$$T \equiv \bigwedge_{\ell \in \mathsf{lit}(T)} \ell$$

Proof. The proof is by induction on the depth of D. If D is of depth 0, then its output has no children, that is, the output of D is an input of D. Thus the only certificate T of D is $\{\mathsf{output}(D)\}$. The underlying graph of T is an isolated vertex, that is a tree rooted in s where $\mathsf{lit}(T)$ is the label of s thus $T \equiv \bigwedge_{\ell \in \mathsf{lit}(T)} \ell$.

Now assume the result holds for DNNF of depth smaller than d and let D be a DNNF of size $d+1$. Let $s = \mathsf{output}(D)$ and let $s_1, \ldots, s_k$ be the inputs of s. Let T be a certificate of D. By definition, T contains s. Now, if s is an $\vee$-gate, then T also contains exactly one gate s_i. Thus $T = \{s\} \cup T_{s_i}$. By Proposition 6.2, T_{s_i} is a certificate of D_{s_i}, thus, by induction, its underlying graph is a tree rooted in s_i and its input gates are labeled with literals having distinct variables. Thus the underlying graph of T is the tree rooted in s connected to a tree rooted in s_i. Moreover, the input gate of T are the same as those of T_{s_i}, thus are labeled with pairwise distinct variables by induction and $T \equiv T_{s_i} \equiv \bigwedge_{\ell \in \mathsf{lit}(T)} \ell$.

If s is an $\wedge$-gate, then $T = \{s\} \cup \bigcup_{i=1}^{k} T_{s_k}$. By Proposition 6.2, for all $i \leq k$, $T_{s_i} \in \mathsf{cert}(D_{s_i})$ that are trees rooted in s_k by induction. Since s is decomposable, $D_{s_1}, \ldots, D_{s_k}$ are disjoint circuits having disjoint variables. Thus T is the rooted tree in s connected to trees $T_{s_1}, \ldots, T_{s_k}$. Moreover, the input gate of T are the union of input gates of T_{s_i}, which are all labeled by disjoint variables since s is decomposable. That is, all input gates of T are labeled by disjoint variables. Finally,

$$T \equiv \bigwedge_{i=1}^{k} T_{s_i} \equiv \bigwedge_{i=1}^{k} \bigwedge_{\ell \in \mathsf{lit}(T_{s_i})} \ell \equiv \bigwedge_{\ell \in \mathsf{lit}(T)} \ell.$$

□

Proposition 6.4. *Let D be a* DNNF*. Then*

$$\mathsf{sat}(D) = \bigcup_{T \in \mathsf{cert}(D)} \mathsf{sat}(T).$$

Moreover, if D is deterministic, the union is disjoint.

Proof. Let $\tau : X \to \{0,1\}$ be a satisfying assignment of D. A set T of gate of D is said satisfied by τ if for all $\alpha \in T$, $\tau \models D_\alpha$. Let $T_0 = \{\mathsf{output}(D)\}$. Since τ satisfies D, T_0 is satisfied by τ.

Assume we have constructed T_i satisfied by τ. If there exists an $\wedge$-gate $\alpha \in T_i$ and a child α_0 of α such that $\alpha_0 \notin T_i$ then we set $T_{i+1} = T_i \cup \{\alpha_0\}$. Observe that since $\tau \models D_\alpha$, we also have $\tau \models D_{\alpha_0}$, that is, T_{i+1} is satisfied by τ. If there exists an $\vee$-gate α in T_i such that no child of α is in T_i, then choose a child α_0 of α such that $\tau \models D_{\alpha_0}$. It exists since $\tau \models D_\alpha$ and set $T_{i+1} = T_i \cup \{\alpha_0\}$. If no such gate exists, then let $T = T_i$.

By construction, T is a certificate of D since it is connected, contains $\mathsf{output}(D)$, for every $\vee$-gate, there is exactly one of its children in T and for every $\wedge$-gate, every child of this gate are in T. Moreover, for every $\ell \in \mathsf{lit}(T)$, $\tau \models \ell$ since τ satisfies T by construction. Thus by Proposition 6.3, $\tau \models T$, that is $\tau \in \mathsf{sat}(T)$.

Now let $T \in \mathsf{cert}(D)$ and let $\tau \in \mathsf{sat}(T)$. One can easily see by induction that for all gate v in T, $\tau \models D_v$. Since $s = \mathsf{output}(D)$ is in T, we have $\tau \models D_s = D$.

Assume that D is deterministic and let $T, T' \in \mathsf{cert}(D)$ such that $T \neq T'$. Assume there exists $\tau \in \mathsf{sat}(T) \cap \mathsf{sat}(T')$. Let w be a gate of T that is also in T' and such that there exists a child of w in T that is not in T'. Such a gate exists since T and T' have at least $\mathsf{output}(D)$ as a common gate since $T \neq T'$. Observe that w cannot be an $\wedge$-gate since if it were, every children of w would be in T and in T'. Thus w is an $\vee$-gate and there exists a child w_1 of w that is in T and another child w_2 of w that is in T'. But since $\tau \in \mathsf{sat}(T) \cap \mathsf{sat}(T')$, it holds that τ satisfies both D_{w_1} and D_{w_2} which contradicts the fact that D is deterministic. Thus $\mathsf{sat}(T) \cap \mathsf{sat}(T') = \emptyset$. □

The following lemma shows how one can recombine certificates:

Lemma 6.5. *Let D be a* DNNF *and v be a gate of D. Let $T, T' \in \mathsf{cert}(D, v)$, then $T'' = (T \setminus T_v) \cup T'_v \in \mathsf{cert}(D, v)$ and $\mathsf{var}(T_v) \subseteq \mathsf{var}(D_v)$ and $\mathsf{var}(T \setminus T_v) \subseteq \mathsf{var}(D) \setminus \mathsf{var}(D_v)$.*

Proof. Since $v \in T'_v$, we also have $v \in T''$. Moreover, the output of D is in T'' and for every $\wedge$-gate w of T'', every child of w is also in T'' since if w is in T'_v then its children are also all in T'_v since T' is a certificate. If w is in $T \setminus T_v$ then either no children of w is v, thus every children of w are also in $T \setminus T_v$ since T is a certificate of D. If v is one of the children of T then it is also in T'' since $v \in T'_v$. We can similarly show that for every $\vee$-gate w of T'', exactly one child of w is in T''.

Now observe that by Lemma 6.2, T_v is a certificate of D_v. Thus $\mathsf{var}(T_v) \subseteq \mathsf{var}(D_v)$. Moreover, assume that there exists a variable x such that $x \in \mathsf{var}(T \setminus T_v) \cap \mathsf{var}(D_v)$. Since $x \in \mathsf{var}(D_v)$, there exists a certificate T''' of D_v such that $x \in \mathsf{var}(T''')$. But then, from what precedes, $(T \setminus T_v) \cup T'''$ is a certificate of D having two leaves labeled by x, contradicting Lemma 6.3. □

6.1.2 Rectangles and covers

In this section, we make a new connection between knowledge compilation and communication complexity. Our presentation of the result may be understood without specific knowledge in communication complexity since we only rely on the notion of rectangles that is presented later. Nevertheless, before presenting the main result, we quickly explain the framework of communication complexity and review previous work concerning the connection of knowledge compilation and communication complexity. The interested reader may find additional background concerning communication complexity in the book of Kushilevitz and Nisan [KN97].

The usual model in communication complexity is the following: we are given a computable boolean function $f : Z \to \{0,1\}$ and a partition of $Z = X \uplus Y$. We want to evaluate f on input $z = (x,y) \in \{0,1\}^n$. We assume that Alice knows x and Bob knows y. In this model, the complexity of f with respect to the partition (X,Y) is measured as the maximal number of bits Alice and Bob have to exchange to compute $f(x,y)$. The time needed to compute f is not relevant here. Of course, the complexity of f is smaller than $\min(|X|,|Y|)$ since Bob (or Alice) could send its input y to Alice and let her compute the function. Thus, the goal is to find non trivial ways of computing such function. Several variations can be made here. We could assume that (X,Y) is not chosen in advance and then define the complexity of f to be the best complexity over every partition (X,Y). Of course, we would like to avoid the trivial cases where $|X| = 1$ which is not relevant. Hence a common assumption is that the partition is balanced, that is, $|X| = |Y| \pm 1$. Such model is called the best partition model.

A connection between the size of the smallest OBDD computing a boolean function f and the complexity of f in the best partition model have been established by Kushilevitz and Nisan [KN97, Kus97] and is reviewed in [Weg00]. The idea is that an OBDD F can be seen as a communication protocol for f with partition $X = \{x_1, \ldots, x_{\lfloor n/2 \rfloor}\}$ and $Y = \{x_{\lfloor n/2 \rfloor + 1}, \ldots, x_n\}$ where $x_1, \ldots, x_n$ is the order of the variables in the OBDD. Indeed, Alice can communicate to Bob the node she reaches in the OBDD F after having followed the partial path corresponding to its partial input. Bob only can then resume the computation in the OBDD from the node Alice has transmitted. This can be done by communicating $O(\log(\mathsf{size}(F)))$ bits. Thus if one has a lower bound on the complexity of f in the best partition model, it can be leveraged to a lower bound on the OBDD size of f.

For richer representation languages such as DNNF, a lower bound on the complexity of f in the best partition model is generally not enough to give a lower bound in this language. However, several variations in the model of computation can be made to match the representation language. Such variations in the model and corresponding lower bounds have been used recently by Beame and Liew [BL15] to prove lower bound on the size of richer representation language such as SDD and DNNF. Their approach however only yields a weakly exponential separation of DNNF and CNF and does not take full advantage of the structure

of DNNF. Moreover, it can hardly be adaptated to restrictions of DNNF such as deterministic DNNF.

The model of computation that is related to our result is the following: the partition of the variable is chosen non-deterministically from the input. It can be thought as if an oracle could see the entire input and choose which bits it gives to Alice, and which bits it gives to Bob. Such model is known as multipartition communication protocol. A presentation of this model and lower bounds can be found in [DHJ+04, JS02]. Our connection to the size of DNNF and the number of rectangles needed to cover it can be understood as a connection between the DNNF size of a boolean function and its complexity in the multipartition model. However, we present our result independently from this framework and the only tool from communication complexity needed to understand the proof is the notion of rectangle.

Rectangles and balanced rectangle cover. Let X be a finite set and $r : \{0,1\}^X \to \{0,1\}$. The boolean function r is said to be a rectangle over X if there exists a partition (X_1, X_2) of X, $r_1 : \{0,1\}^{X_1} \to \{0,1\}$ and $r_2 : \{0,1\}^{X_2} \to \{0,1\}$ such that for all $\tau : X \to \{0,1\}$, $r(\tau) = 1$ if and only if $r_1(\tau|_{X_1}) = r_2(\tau|_{X_2}) = 1$. Intuitively, it means that $r \equiv r_1 \wedge r_2$ and $\mathsf{var}(r_1) \cap \mathsf{var}(r_2) = \emptyset$. A rectangle is said to be *balanced* if $|X|/3 \leq |X_1| \leq 2|X|/3$.

Let $f : \{0,1\}^X \to \{0,1\}$ be a function. A *rectangle cover* of f is a finite set R of rectangles over X such that $\mathsf{sat}(f) = \bigcup_{r \in R} \mathsf{sat}(r)$. A rectangle cover R is said *balanced* if for every $r \in R$, the underlying partition of r is balanced. It is said *disjoint* if for every $r_1, r_2 \in R$, if $r_1 \neq r_2$ then $\mathsf{sat}(r_1) \cap \mathsf{sat}(r_2) = \emptyset$. The *size* of R is simply $|R|$, the number of rectangles in R.

By extension, we say that a set $F \subseteq \{0,1\}^X$ is a rectangle if the function $f_F : \{0,1\}^X \to \{0,1\}$ defined as $f_F(\tau) = 1$ if and only if $\tau \in F$ is a rectangle. A rectangle cover of F is a rectangle cover of f_F.

Observation 2. *Given $F \subseteq \{0,1\}^X$ and a partition (X_1, X_2) of X, we have that F is a rectangle over X with underlying partition (X_1, X_2) if and only if for every $\tau_1, \tau_2 \in F$, $\tau_1|_{X_1} \cup \tau_2|_{X_2} \in F$.*

Rectangle covers of DNNF. In this section, we show a strong connection between the size of a balanced (disjoint) rectangle cover of a function $f : \{0,1\}^X \to \{0,1\}$ and the size of any (deterministic) DNNF computing f. This allows to lift known lower bounds from communication complexity on the size of balanced rectangle cover needed for some functions to the framework of knowledge compilation.

For a DNNF D and a gate v of D, we denote by $\mathsf{sat}(D, v) = \bigcup_{T \in \mathsf{cert}(D,v)} \mathsf{sat}(T)$. Intuitively, $\mathsf{sat}(D, v)$ is the set of satisfying assignments of D that also satisfy D_v. We start by showing that for all v, $\mathsf{sat}(D, v)$ is a rectangle:

Lemma 6.6. *Let D be a DNNF on variable X and v a gate of D. $\mathsf{sat}(D, v)$ is a rectangle over X whose underlying partition is $(\mathsf{var}(D_v), X \setminus \mathsf{var}(D_v))$.*

Proof. To ease notations, we denote by $X_v = \mathsf{var}(D_v)$ and by $\overline{X_v} = X \setminus X_v$. By Observation 2, it is sufficient to show that for every $\tau, \tau' \in \mathsf{sat}(D, v)$, $\tau'' := \tau|_{\overline{X_v}} \cup \tau'|_{X_v} \in \mathsf{sat}(D, v)$. Let $T \in \mathsf{cert}(D, v)$ such that $\tau \models T$ and $T' \in \mathsf{cert}(D, v)$ such that $\tau' \models T'$.

By Lemma 6.5, $T'' = (T \setminus T_v) \cup T'_v$ is a certificate of D going through v, $\mathsf{var}(T'_v) \subseteq X_v$ and $\mathsf{var}((T \setminus T_v)) \subseteq \overline{X_v}$. Thus, $\tau_{\overline{X_v}}$ satisfies every literal labeling the inputs of $T \setminus T_v$ since it satisfies every literals of T and $\mathsf{var}(T \setminus T_v) \subseteq \overline{X_v}$. Similarly, $\tau'|_{X_v}$ satisfies every literals of T'_v. Therefore, τ'' satisfies every literal of T'' that is $\tau'' \models T''$ and since $v \in T''$, $\tau'' \in \mathsf{sat}(D, v)$. □

Lemma 6.6 gives a natural way of covering the satisfying assignments of a DNNF D with rectangles by simply taking

$$\bigcup_{v \in D} \mathsf{sat}(D, v).$$

Balanced rectangle covers of DNNF. This covering is however of little interest for proving lower bounds since it is not balanced. It is easy to see that one can cover any boolean function f on variables X with two (unbalanced) rectangles on partition $(\{x\}, X \setminus \{x\})$ for $x \in X$ by remarking that

$$\mathsf{sat}(f) = \mathsf{sat}(x \wedge f(\{x \mapsto 1\})) \cup \mathsf{sat}(\neg x \wedge f(\{x \mapsto 0\}).$$

The rest of this section is dedicated to the extraction of a set of gate A of D such that $\bigcup_{v \in A} \mathsf{sat}(D, v)$ is a balanced rectangle cover of D.

Our strategy is as follows: we iteratively construct a set R which will be in the end a balanced rectangle cover of $\mathsf{sat}(D)$. We start by looking for a gate v such that the underlying partition of the rectangle $\mathsf{sat}(D, v)$ given by Lemma 6.6 is balanced. We add $\mathsf{sat}(D, v)$ to R and then drop the gate v by replacing it by the constant 0. We iterate this process us until D is unsatisfiable. In this end, we have a guarantee that R is a balanced rectangle cover of D. More precisely, we prove the following theorem:

Theorem 6.7. *Let D be a* DNNF*. There exists a balanced rectangle cover R of $\mathsf{sat}(D)$ of size at most $\mathsf{size}(D)^2$. Moreover, if D is deterministic, we can assume R to be disjoint.*

Our strategy is essentially based on the procedure of eliminating a gate in a DNNF. Given a DNNF D and a gate v in D, we denote by $D \setminus v$ the DNNF where we have removed every input wire of v and relabeled v with the constant 0. If G is a set of gates of D, we denote by $D \setminus G$ the DNNF obtained by successively removing the gates in G. Observe that for every gate v, w, it holds that $(D \setminus v) \setminus w = (D \setminus w) \setminus v$. Therefore the order we eliminate the gate of G in D is not important and $D \setminus G$ is well defined. There is a nice connection between the satisfying assignments of D and those of $D \setminus v$:

Lemma 6.8. *Let D be a* DNNF *and v a gate. We have*

$$\mathsf{sat}(D) \setminus \mathsf{sat}(D, v) \subseteq \mathsf{sat}(D \setminus v) \subseteq \mathsf{sat}(D).$$

Moreover, if D is deterministic then $D \setminus v$ is deterministic and $\mathsf{sat}(D) \setminus \mathsf{sat}(D, v) = \mathsf{sat}(D \setminus v)$.

Proof. In this proof, we identify the gates of D different from v to the gates of $D \setminus v$. This is possible since the only gate that has been relabeled in $D \setminus v$ is v.

We first show that $\mathsf{sat}(D \setminus v) \subseteq \mathsf{sat}(D)$. Let $\tau \in \mathsf{sat}(D \setminus v)$. By Proposition 6.4, there exists a certificate T of $D \setminus v$ such that $\tau \in \mathsf{sat}(T)$. Moreover, since T is equivalent to the conjunction of its inputs, we know that the constant 0 does not label any of the inputs of T. Thus T only contains gates of D, that is, T is also a certificate of T. Thus by Proposition 6.4 again, $\tau \in \mathsf{sat}(D)$. That is $\mathsf{sat}(D \setminus v) \subseteq \mathsf{sat}(D)$.

We now prove $\mathsf{sat}(D) \setminus \mathsf{sat}(D, v) \subseteq \mathsf{sat}(D \setminus v)$. Let $\tau \in \mathsf{sat}(D) \setminus \mathsf{sat}(D, v)$. By Proposition 6.4, there exists $T \in \mathsf{cert}(D)$ such that $\tau \in \mathsf{sat}(T)$. Moreover, by definition of $\mathsf{sat}(D, v)$, we can assume that T does not contain v, otherwise, $\mathsf{sat}(T) \subseteq \mathsf{sat}(D, v)$. Since T does not contain v, it only contains gates that are in $D \setminus v$. Thus, $T \in \mathsf{cert}(D \setminus v)$, that is, $\tau \in \mathsf{sat}(D \setminus v)$ by Proposition 6.4 again, that is, $\mathsf{sat}(D) \setminus \mathsf{sat}(D, v) \subseteq \mathsf{sat}(D \setminus v)$.

We now prove that if D is deterministic, then we actually have $\mathsf{sat}(D) \setminus \mathsf{sat}(D, v) = \mathsf{sat}(D \setminus v)$. It only remains to prove the inclusion $\mathsf{sat}(D \setminus v) \subseteq \mathsf{sat}(D) \setminus \mathsf{sat}(D, v)$. Assume D is deterministic and let $\tau \in \mathsf{sat}(D \setminus v)$. From what precedes, we have in particular $\tau \in \mathsf{sat}(D)$ and there even exists a certificate T of D that does not contain v and such that $\tau \in \mathsf{sat}(T)$. Assume toward a contradiction that $\tau \in \mathsf{sat}(D, v)$. This means that there exists $T' \in \mathsf{cert}(D, v)$ such that $\tau \in \mathsf{sat}(T')$, But since v is not in T, we have $T \neq T'$ and $\tau \in \mathsf{sat}(T) \cap \mathsf{sat}(T')$. Since D is deterministic, it contradicts Proposition 6.4. □

A immediate consequence of Lemma 6.8 is that $\mathsf{sat}(D) = \mathsf{sat}(D, v) \cup \mathsf{sat}(D \setminus v)$ and even $\mathsf{sat}(D) = \mathsf{sat}(D, v) \uplus \mathsf{sat}(D \setminus v)$ if D is deterministic. More generally, the following holds:

Corollary 6.9. *Let D be a* DNNF*, $v_1, \ldots, v_k$ a sequence of gates of D. Then it holds that*

$$\mathsf{sat}(D) = \mathsf{sat}(D^k) \cup \bigcup_{j=0}^{k-1} \mathsf{sat}(D^j, v_{j+1}).$$

where $D^0 = D$ and $D^i = D \setminus \{v_1, \ldots, v_i\}$ for all $i \leq k$. Moreover, if D is deterministic, then D^k is deterministic and

$$\mathsf{sat}(D) = \mathsf{sat}(D^k) \uplus \biguplus_{j=0}^{k-1} \mathsf{sat}(D^j, v_{j+1}).$$

Proof. The proof is by induction on k. The result holds for $k = 0$ since $D^0 = D$. Assume that the result holds at rank k, that is

$$\mathsf{sat}(D) = \mathsf{sat}(D^k) \cup \bigcup_{j=0}^{k-1} \mathsf{sat}(D^j, v_{j+1}) \tag{6.1}$$

and let v_{k+1} be a gate of D different from $v_1, \ldots, v_k$. By Lemma 6.8, it holds that $\mathsf{sat}(D^k) \setminus \mathsf{sat}(D^k, v_{k+1}) \subseteq \mathsf{sat}(D^k \setminus v_{k+1}) \subseteq \mathsf{sat}(D^k)$. In other words, $\mathsf{sat}(D^k) = \mathsf{sat}(D^k \setminus v_{k+1}) \cup \mathsf{sat}(D^k, v_{k+1})$, that is $\mathsf{sat}(D^k) = \mathsf{sat}(D^{k+1}) \cup \mathsf{sat}(D^k, v_{k+1})$. Injecting this equality in Equation 6.1 yields the result at rank $k + 1$.

If D is deterministic, then by induction, D^k is also deterministic and

$$\mathsf{sat}(D) = \mathsf{sat}(D^k) \uplus \biguplus_{j=0}^{k-1} \mathsf{sat}(D^j, v_{j+1}). \tag{6.2}$$

By Lemma 6.8, we have $\mathsf{sat}(D^k) \setminus \mathsf{sat}(D^k, v_{k+1}) = \mathsf{sat}(D^k \setminus v_{k+1}) = \mathsf{sat}(D^{k+1})$ and $D^k \setminus v_{k+1} = D^{k+1}$ is deterministic.

In other words, $\mathsf{sat}(D^k) = \mathsf{sat}(D^{k+1}) \uplus \mathsf{sat}(D^k, v_{k+1})$. Injecting this in Equation 6.2 yields the result at rank $k + 1$.

□

We say that a gate v in a DNNF D is *n-balanced* if $n/3 \leq |\mathsf{var}(D_v)| \leq 2n/3$. If D is in normal form and has sufficiently many variables, then we can find an n-balanced gate. We have a small technicality in the lemma that follows. Our elimination process disconnects gates of the DNNF and thus, some input may become inaccessible from the output of the DNNF. The set of variables of the DNNF do not change since we have define $\mathsf{var}(D)$ to be the variables of the circuit. But if we denote $s = \mathsf{output}(D)$, then $\mathsf{var}(D_s)$ may be different from $\mathsf{var}(D)$ since D_s contains only the subcircuit rooted in s. In particular, it contains only the accessible inputs.

Lemma 6.10. *Let D be a* DNNF*, $s = \mathsf{output}(D)$ and $n \in \mathbb{N}$. If $|\mathsf{var}(D_s)| \geq n/3$ then there exists an n-balanced gate in D.*

Proof. The proof is by induction on $\mathsf{size}(D)$. Let D be a DNNF such that $|\mathsf{var}(D_s)| \geq n/3$ where $s = \mathsf{output}(D)$. If $|\mathsf{var}(D_s)| \leq 2n/3$ then s is n-balanced and we are done. Otherwise, we have $|\mathsf{var}(D_s)| > 2n/3$. Let s_1, s_2 be the children of s. Since $\mathsf{var}(D_s) = \mathsf{var}(D_{s_1}) \cup \mathsf{var}(D_{s_2})$, it holds that $|\mathsf{var}(D_s)| \leq |\mathsf{var}(D_{s_1})| + |\mathsf{var}(D_{s_2})|$. Thus $2n/3 \leq |\mathsf{var}(D_{s_1})| + |\mathsf{var}(D_{s_2})|$. That is, either $n/3 \leq |\mathsf{var}(D_{s_1})|$ or $n/3 \leq |\mathsf{var}(D_{s_2})|$. Assume without lost of generality that $n/3 \leq |\mathsf{var}(D_{s_1})|$. By induction, there exists an n-balanced gate in D_{s_1} that is also an n-balanced gate of D. □

Theorem 6.11. *Let D be a* DNNF *in normal form. There exists a balanced rectangle cover R of $\mathsf{sat}(D)$ of size at most $\mathsf{size}(D)$. Moreover, if D is deterministic, we can assume R to be disjoint.*

Proof. Let $n = |\mathsf{var}(D)|$. By iteratively applying Lemma 6.10, we construct a sequence $v_1, \ldots, v_k$ of gates of D such that for every $i \leq k$, v_i is n-balanced in $D^{i-1} = D \setminus \{v_1, \ldots, v_{i-1}\}$ and such that $|\mathsf{var}(D^k_s)| < n/3$ where $s = \mathsf{output}(D^k)$.

By Corollary 6.9, it holds that:

$$\mathsf{sat}(D) = \mathsf{sat}(D^k) \cup \bigcup_{j=0}^{k-1} \mathsf{sat}(D^j, v_{j+1}).$$

By Lemma 6.6, for every $j < k$, $\mathsf{sat}(D^j, v_{j+1})$ is a rectangle with underlying partition $(V_j, \mathsf{var}(D) \setminus V_j)$ where $V_j = \mathsf{var}(D^j_{v_{j+1}})$ and since v_{j+1} is n-balanced in D^j, we have $\mathsf{sat}(D^j, v_{j+1})$ is a balanced rectangle. Moreover, $\mathsf{sat}(D^k)$ depends only on variables $V = \mathsf{var}(D^k_s)$ where $s = \mathsf{output}(D^k)$. By definition, we have $|V| < n/3$. Let $W \subseteq |\mathsf{var}(D)|$ such that $n/3 \leq |V \cup W| \leq 2n/3$. We have that $\mathsf{sat}(D^k)$ is a rectangle with underlying partition $(V \cup W, \mathsf{var}(D) \setminus (W \cup V))$ since $D^k \equiv (D^k \wedge \top_W) \wedge \top_{\mathsf{var}(D)\setminus(V \cup W)}$ where $\top_X$ is the 1-constant boolean function on variables X.

If D is deterministic, then by Corollary 6.9, we have:

$$\mathsf{sat}(D) = \mathsf{sat}(D^k) \uplus \biguplus_{j=0}^{k-1} \mathsf{sat}(D^j, v_{j+1}).$$

Following the same argument as in the non-deterministic case, we conclude that this is a rectangle cover of D. This rectangle cover is disjoint. □

The proof of Theorem 6.7 follows from Theorem 6.11 and the fact that for every DNNF D, there exists a DNNF D' in normal form of size at most $\mathsf{size}(D)^2$ by Theorem 1.61.

6.2 Separating CNF-formulas from DNNF

Theorem 6.7 provides a nice tool for proving lower bounds in knowledge compilation. Indeed, if one needs at least N balanced rectangles to cover a boolean function f, then Theorem 6.7 says that there exists no DNNF D equivalent to f of size smaller than $\sqrt{N}$. Many functions are known from communication complexity for being hard to cover with balanced rectangles, that is, any rectangle cover of such functions must be of size exponential in the number of its variables.

In this section, we separate unconditionally CNF-formulas from DNNF, that is, we exhibit families of CNF-formulas for which every equivalent DNNF is of size exponential in the size of the formula. We start by separating CNF-formulas from DNNF by using only known lower bounds on monotone circuits [AB87]. This yields a separation of CNF-formulas and DNNF but this gives no strong lower bounds. We then show how we can lift known lower bounds from communication complexity [Juk12] to separate CNF from DNNF. This technique provides a strong

exponential lower bound. In the last subsection, we provide a self-contained proof of the separation of CNF-formulas and DNNF by showing a family of 2-CNF that cannot be covered by a small number of balanced rectangles.

6.2.1 A weakly exponential lower bound

This bound relies on lower bounds on the size of monotone circuits for some boolean functions. The key observation is that DNNF computing monotone functions can be assumed to be negation free, that is, they could be seen as monotone circuits.

Proposition 6.12. *Let D be a* DNNF *computing a monotone boolean function. There exists a* DNNF D' *equivalent to D having only positive literals as input such that* $\mathsf{size}(D') \leq \mathsf{size}(D)$.

Proof. We show that if D computes a monotone boolean function then replacing an input of D labeled with a negative literal by the constant 1 does not change the function computed by D. It is then sufficient to replace every input labeled by a negative literal by 1 to get the desired negation-free D'.

Let u be an input gate labeled with the negative literal $\neg x$ for $x \in \mathsf{var}(D)$ and let D_1 be the DNNF where u is replaced by constant 1. We claim that $\mathsf{sat}(D) = \mathsf{sat}(D_1)$. Let $\tau \in \mathsf{sat}(D)$. If $\tau(x) = 0$, then $\tau(\neg x) = 1$ and we clearly have $\tau \in \mathsf{sat}(D_1)$. If $\tau(x) = 1$, then by Proposition 6.4, there exists $T \in \mathsf{cert}(D)$ such that $\tau \in \mathsf{sat}(T)$. Since $\tau(x) = 1$ and T is equivalent to the conjunction of its inputs, $\neg x$ does not appear in the inputs of T, that is, u is not a gate of T. In other words, T is also a certificate of D_1, that is, $\tau \in \mathsf{sat}(D_1)$.

Now let $\tau \in \mathsf{sat}(D_1)$. Again, if $\tau(x) = 0$ then we clearly have $\tau \in \mathsf{cert}(D)$. Otherwise assume $\tau(x) = 1$ and let $T \in \mathsf{cert}(D_1)$ such that $\tau \in \mathsf{sat}(T)$. If u is not in T then T is also a certificate of D and we have $\tau \in \mathsf{cert}(D)$. Otherwise, let T' be the certificate of D having the same gates as T. The only difference between T and T' is that u is labeled by 1 in T and by $\neg x$ in T'. Let τ' be the truth assignment such that τ' is equal to τ on every variable but x and such that $\tau'(x) = 0$. It is readily verified that τ' satisfies T', that is, $\tau' \in \mathsf{sat}(D)$. Since D is monotone, we also have $\tau \in \mathsf{sat}(D)$. □

We now show how Proposition 6.12 can be combined with known lower bounds on monotone circuits (boolean circuits without negation gates) to prove the desired separation.

For $1 \leq k \leq n$, let CLIQUE(n, k) denote the boolean function of $\binom{n}{2}$ variables representing the edges of an undirected n-vertex graph G such that CLIQUE$(n, k) = 1$ if, and only if, G contains a clique on k vertices. Owing to a result by Alon and Boppana [AB87], every *monotone* circuit computing CLIQUE(n, k) for $k \leq n^{1/4}$ has size $2^{\Omega(\sqrt{n})}$. This lower bound also applies to DNNFs computing CLIQUE(n, k) since a minimum-size DNNF computing a monotone function is monotone by

Proposition 6.12, and CLIQUE(n,k) is a monotone function. This leads to a weakly exponential separation of DNNFs from CNFs as follows.

The problem CLIQUE is in NP so there exists a non deterministic polynomial-time decision algorithm. By the Cook-Levin theorem, given n,k, there exists a CNF F_n of size polynomial in n that encodes the run of this algorithm on a graph of size n. Let D_n be a DNNF computing F_n. They both have $m = \binom{n}{2}$ variables encoding the graph G as for CLIQUE plus auxiliary variables encoding the run of the algorithm. We can existentially project these auxiliary variables in D_n without increasing its size by Proposition 1.54, resulting in a DNNF D'_n computing CLIQUE, thus, of size at least $2^{\Omega(\sqrt{n})}$. Since $\mathsf{size}(D'_n) \leq \mathsf{size}(D_n)$, we also have $\mathsf{size}(D_n) \geq 2^{\Omega(\sqrt{n})}$ resulting in a (weak) separation of CNF and DNNF since F_n is of size polynomial in n.

6.2.2 Lifting known lower bound from communication complexity

In [JS02], Jukna and Schnitger exhibit for all n, a boolean function JS_n having n^2 variables and such that every balanced rectangle cover of JS_n is of size at least $2^{\Omega(n^2)}$. Using Theorem 6.7 on JS_n, we immediately get an exponential lower bound on the size of any DNNF representing JS_n. Moreover, we can show that JS_n can be represented by a CNF of size $\Omega(n^2)$, thus it implies a strong exponential separation of CNF-formulas and DNNF. In the following, we recall precisely the result of [JS02] and show how to represent JS_n as a small CNF in order to prove the desired strong exponential separation.

For $n \geq 2$, let K_n be the set of all 2-element subsets (edges) of $\{1,\ldots,n\}$. We view every subset of K_n as the edge set of a graph G whose vertex set is $\{1,\ldots,n\}$. We identify edges in K_n with boolean variables, so that the graph $G \subseteq K_n$ is encoded by the $\{0,1\}$-assignment of K_n mapping a variable (edge) to 1 if and only if it is in the edge set of G.

A triangle $T = \{i,j,k\}$ on n vertices is a set of three distinct integers smaller than n. We say that a boolean function $f : \{0,1\}^{K_n} \to \{0,1\}$ avoids a triangle T if every graph with n vertices satisfying f does not contain the three edges $\{i,j\},\{i,k\},\{j,k\}$. In other words, if $\tau \in \mathsf{sat}(f)$, then $\tau(\{i,j\}) = 0$ or $\tau(\{i,k\}) = 0$ or $\tau(\{j,k\}) = 0$.

For a set A of triangles, let

$$JS_n^A \colon \{0,1\}^{K_n} \to \{0,1\} \tag{6.3}$$

denote the function accepting exactly those graphs over $\{1,\ldots,n\}$ that avoid all triangles in A.

Jukna and Schnitger show an exponential lower bound on the size of balanced rectangle covers for functions as in (6.3).

Theorem 6.13 (Jukna and Schnitger)**.** *For every n, there exists a set A_n of triangles of size $O(n^2)$ such that any balanced rectangle cover of $JS_n^{A_n}$ in (6.3) has size $2^{\Omega(n^2)}$.*

The *Jukna-Schnitger function* $JS_n\colon \{0,1\}^{K_n} \to \{0,1\}$ is defined by

$$JS_n = JS_n^{A_n} \tag{6.4}$$

where A_n is chosen by Theorem 6.13 ($n \geq 2$).

Lemma 6.14. *Let A be a set of triangles. The function JS_n^A is equivalent to the* CNF*-formula:*

$$F_n^A = \bigwedge_{(i,j,k)\in A} \neg\{i,j\} \vee \neg\{i,k\} \vee \neg\{j,k\}.$$

Proof. The equivalence follows easily from the fact that a satisfying assignment τ avoids a triangle (i,j,k) if and only if τ satisfies the clause $\neg\{i,j\}\vee\neg\{i,k\}\vee\neg\{j,k\}$. Indeed, if τ avoids triangle (i,j,k), then it should set either $\{i,j\}$ or $\{j,k\}$ or $\{i,k\}$ to 0 and then it satisfies the clause. Reciprocally, if it satisfies the clause, it set either $\{i,j\}$ or $\{j,k\}$ or $\{i,k\}$ to 0, that is, it avoids the triangle (i,j,k). □

Applying Lemma 6.14 with the family A_n of Theorem 6.13, it follows:

Corollary 6.15. *For every $n \geq 2$, there exists a monotone* 3-CNF *of size $3|A_n| = \Omega(n^2)$ equivalent to JS_n.*

The exponential separation of CNF-formula and DNNF follows easily from Theorem 6.13, Theorem 6.7 and Corollary 6.15. The consequences of such a separation are discussed later in Section 6.2.4. We first present another family of CNF-formulas having no small DNNF with a complete proof of the lower bound.

6.2.3 A family of CNF having no small DNNF

In this section we exhibit a simple family of monotone 2-CNF requiring an exponential number of balanced rectangle to be covered. This provides an alternative simpler proof of the result of Jukna and Schnitger [JS02] presented above. Our formulas are constructed from graphs by adding a constraint between two variables if and only if there is an edge between these variables in the graph. We rely on an simple exponential lower bound on the ratio of vertex covers of a bounded degree graph containing a fixed set S of vertices (Theorem 6.17). We use this lower bound to prove that, for a well-chosen family of graphs, any balanced rectangle can cover only an exponentially small ratio of the total number of satisfying assignments of the formula, giving the lower bound.

CNF and vertex covers A graph $G = (V,E)$ naturally defines a monotone 2-CNF F_G on variables V as follows:

$$F_G = \bigwedge_{(x,y)\in E} (x \vee y)$$

This CNF inherits the properties of the graph and it will be useful to prove lower bounds. The solutions of F_G exactly correspond to vertex covers of G: $W \subseteq V$

is a vertex cover of G if and only if $\mathbb{1}_W$ is a satisfying assignment of F_G. In this section, we prove a combinatorial result on the number of vertex covers of graphs of bounded degree that will be crucial in the next section. Given a graph G, we denote by $\mathsf{VC}(G)$ the set of vertex covers of G and given $W \subseteq V$, we denote by $\mathsf{VC}(G, W) = \{\mathcal{C} \in \mathsf{VC}(G) \mid W \subseteq \mathcal{C}\}$, the set of vertex covers containing W. The following result states that the number of vertex covers of G containing W decreases exponentially in $|W|$. This result was first proven by Razgon in [Raz14] but we present here a much shorter and elementary proof:

Lemma 6.16. *Let $G = (V, E)$ be a graph, $W \subseteq V$ a non-empty subset of vertices and $v \in W$ a vertex. Let d be the degree of v in G. We have:*

$$(1 + 2^{-d})|\mathsf{VC}(G, W)| \leq |\mathsf{VC}(G, W \setminus \{v\})|$$

Proof. We denote by K the set of vertex covers that contains $W \setminus \{v\}$ but *not* v. Clearly,

$$\mathsf{VC}(G, W \setminus \{v\}) = \mathsf{VC}(G, W) \uplus K \tag{6.5}$$

Let f be the mapping from $\mathsf{VC}(G, W)$ to $\mathcal{P}(\mathcal{N}(v)) \times K$ defined for all $\mathcal{C} \in \mathsf{VC}(G, W)$ by

$$f(\mathcal{C}) = (\mathcal{C} \cap \mathcal{N}(v), (\mathcal{C} \setminus \{v\}) \cup \mathcal{N}(v)).$$

For $\mathcal{C} \in \mathsf{VC}(G, W)$, we have $\mathcal{C} \cap \mathcal{N}(v) \subseteq \mathcal{N}(v)$, that is $\mathcal{C} \cap \mathcal{N}(v) \in \mathcal{P}(\mathcal{N}(v))$. Moreover $(\mathcal{C} \setminus \{v\}) \cup \mathcal{N}(v)$ is in K: it contains $W \setminus \{v\}$ since $W \subseteq \mathcal{C}$, and it does not contain v by construction. Moreover it is a vertex cover of G. Indeed, let e be an edge of G. If e does not contain v, then it is covered by a vertex $u \neq v$ in $\mathcal{C}$. In particular, $u \in (\mathcal{C} \setminus \{v\}) \cup \mathcal{N}(v)$. Now if e contains v, then $e = (u, v)$ with $u \in \mathcal{N}(v)$. Thus e is also covered by $(\mathcal{C} \setminus \{v\}) \cup \mathcal{N}(v)$.

Now we prove that f is an injection. Let $\mathcal{C}, \mathcal{D} \in \mathsf{VC}(G, W)$ be such that $f(\mathcal{C}) = f(\mathcal{D})$ that is $\mathcal{C} \cap \mathcal{N}(v) = \mathcal{D} \cap \mathcal{N}(v)$ and $(\mathcal{C} \setminus \{v\}) \cup \mathcal{N}(v) = (\mathcal{D} \setminus \{v\}) \cup \mathcal{N}(v)$. Since $v \in \mathcal{C}$ and $v \in \mathcal{D}$, we deduce from the second equality that $\mathcal{C} \setminus \mathcal{N}(v) = \mathcal{D} \setminus \mathcal{N}(v)$. Thus $\mathcal{C} = (\mathcal{C} \cap \mathcal{N}(v)) \cup (\mathcal{C} \setminus \mathcal{N}(v)) = (\mathcal{D} \cap \mathcal{N}(v)) \cup (\mathcal{D} \setminus \mathcal{N}(v)) = \mathcal{D}$.

Since f is an injection, we have

$$|\mathsf{VC}(G, W)| \leq |\mathcal{P}(\mathcal{N}(v))| \cdot |K| = 2^d |K|.$$

Plugging this inequality into Equation 6.5, we have:

$$|\mathsf{VC}(G, W \setminus \{v\})| \geq (1 + 2^{-d})|\mathsf{VC}(G, W)|$$

□

By iteratively applying this lemma to a bounded degree graph, we get, as a corollary, the following theorem:

Theorem 6.17. *Let $G = (V, E)$ be a graph of degree d and $W \subseteq V$ a subset of vertices. We have*

$$|\mathsf{VC}(G, W)| \leq \left(\frac{2^d}{2^d + 1}\right)^{|W|} |\mathsf{VC}(G)|.$$

Proof. The proof is by induction on $|W|$. If $W = \emptyset$, the inequality is clear. Now assume that $W \neq \emptyset$ and let $v \in W$. By induction,

$$|\mathsf{VC}(G, W \setminus \{v\})| \leq \Big(\frac{2^d}{2^d+1}\Big)^{|W|-1} |\mathsf{VC}(G)|.$$

Since the degree of v is smaller than d, by Lemma 6.16, we have

$$|\mathsf{VC}(G, W)| \leq \Big(\frac{2^d}{2^d+1}\Big) |\mathsf{VC}(G, W \setminus \{v\})|$$

thus

$$|\mathsf{VC}(G, W)| \leq \Big(\frac{2^d}{2^d+1}\Big)^{|W|} |\mathsf{VC}(G)|.$$

□

Observe that $\left(\frac{2^d}{2^d+1}\right) < 1$. Theorem 6.17 thus establishes an exponential lower bound on the proportion of vertex cover of a bounded degree graph G containing a given subset of variables.

Rectangle covers of graph CNF Given a boolean function $f : \{0,1\}^X \to \{0,1\}$ and a rectangle r over X, we say that r is compatible with f if $\mathsf{sat}(r) \subseteq \mathsf{sat}(f)$. Observe that if R is a rectangle cover of f, then for all $r \in R$, r is compatible with f. We give some property of rectangles compatible with graph CNF.

Lemma 6.18. *Let $G = (V, E)$ be a graph, $(v_1, v_2) \in E$ and (V_1, V_2) a partition of V such that $v_1 \in V_1$ and $v_2 \in V_2$. Let r be rectangle over V, compatible with F_G with underlying partition (V_1, V_2). There exists $v \in \{v_1, v_2\}$ such that for every $\tau \in \mathsf{sat}(r)$, $\tau(v) = 1$.*

Proof. Assume there exists $\tau, \tau' \in \mathsf{sat}(r)$ such that $\tau(v_1) = 0$ and $\tau'(v_2) = 0$. Since r is a rectangle, $\tau'' := \tau|_{V_1} \cup \tau'|_{V_2} \in \mathsf{sat}(r)$ and $\tau''(v_1) = \tau''(v_2) = 0$. It follows that τ'' does not satisfy F_G since it does not satisfy the clause $v_1 \vee v_2$, which contradicts the fact that r is compatible with F_G. □

Lemma 6.18 can be extended to the case where we have a large matching between the two parts of the partition:

Lemma 6.19. *Let $G = (V, E)$ be a graph, (V_1, V_2) a partition of V and M a matching between V_1 and V_2. Let r be a rectangle over V, compatible with F_G with underlying partition (V_1, V_2). There exists $I_M \subseteq V$ such that $|I_M| = |M|$ and for every $\tau \in \mathsf{sat}(r)$, for every $v \in I_M$, $\tau(v) = 1$.*

Proof. Let $e = (v_1, v_2) \in M$. Since M is between V_1 and V_2, we have $v_1 \in V_1$ and $v_2 \in V_2$. By Lemma 6.18, there exists $v_e \in \{v_1, v_2\}$ such that for all $\tau \in \mathsf{sat}(r)$, $\tau(v_e) = 1$. Therefore, $I_M = \{v_e \mid e \in M\}$ satisfies the fact that for all $\tau \in \mathsf{sat}(r)$ and $v \in I_M$, $\tau(v) = 1$. Moreover, since M is a matching, $|I_M| = |M|$. □

This gives the following useful corollary:

Corollary 6.20. *Let $G = (V, E)$ be a graph of degree d, (V_1, V_2) a partition of V and M a matching between V_1 and V_2. Let r be rectangle over V, compatible with F_G with underlying partition (V_1, V_2). Then*

$$|\mathsf{sat}(r)| \leq \Big(\frac{2^d}{1+2^d}\Big)^{|M|} |\mathsf{sat}(F_G)|.$$

Proof. By Lemma 6.19, there exists $I_M \subseteq V$ such that $|I_M| = |M|$ and for every $\tau \in \mathsf{sat}(r)$, for all $x \in I_M$, $\tau(x) = 1$. Thus, the vertex cover of G corresponding to τ contains I_M. That is

$$|\mathsf{sat}(r)| \leq |\mathsf{VC}(G, I_M)|.$$

The desired bound follows by Theorem 6.17 since $|\mathsf{VC}(G)| = |\mathsf{sat}(F_G)|$. □

Balanced rectangle cover of expander graphs Corollary 6.20 states that we have a rectangle r compatible with a bounded degree graph CNF and if there exists a large matching between the two parts of the underlying partition of r, then r covers only a small ratio of the total number of satisfying assignments of the formula. This ratio is exponentially smaller in $|M|$.

Expander graphs are bounded degree graphs that are well connected. We show that if we choose a balance partition in an expander graph, we are sure to find a large matching across this partition. We then use it to derive an exponential lower bound on the size of any balanced rectangle cover of such graph CNF.

We start by defining expander graphs. There are numerous different definitions of expander graphs but we will only focus on the so-called boundary expansion. We recall that $\mathcal{N}(S)$ is the open neighborhood of S that is, the set of vertices that are neighbors of an element of S but not in S.

Definition 6.21. *A graph $G = (V, E)$ is a (c, d)-expander if it is of degree d and for every $S \subseteq V$, if $|S| \leq |V|/2$ then $\mathcal{N}(S) \geq c|S|$.*

The property of expander graphs given in the following definition are the only one that we need to prove the lower bound. Fortunately, such graphs exist:

Theorem 6.22 (Section 9.2 in [AS00])**.** *For all $d \geq 3$, there exists $c > 0$ and a sequence of graphs $\{G_i \mid i \in \mathbb{N}\}$ such that $G_i = (V_i, E_i)$ is a (c, d)-expander and $|V_i| \to \infty$ as $i \to \infty$ ($i \in \mathbb{N}$).*

Lemma 6.23. *Let $G = (V, E)$ be a (c, d)-expander with $c \leq 1$ and let (V_1, V_2) be a balanced partition of V. There exists a matching M between V_1 and V_2 with $|M| \geq c|V|/(3d)$.*

Proof. Since (V_1, V_2) is a partition of V, we have $\mathcal{N}(V_1) \subseteq V_2$ and we can assume w.l.o.g that $|V_1| \leq |V|/2$. Since G is a (c, d)-expander, $|\mathcal{N}(V_1)| \geq c|V_1|$. And since (V_1, V_2) is balanced, we have $|V_1| \geq |V|/3$ that is $|\mathcal{N}(V_1)| \geq c|V|/3$. We construct a matching M between V_1 and $\mathcal{N}(V_1) \subseteq V_2$ of size at least $c|V|/(3d)$.

We construct M iteratively. We start with $M = \emptyset$, $A = V_1$ and $B = \mathcal{N}(V_1)$. Pick an edge (v_1, v_2) between A and B and add it to M. Remove v_1 from A, v_2 from B and every vertices of B that are not connected to an element of A anymore. Repeat until A or B is empty.

Since at each step we remove isolated vertices from B, if B is non-empty, then we always have an edge between A and B to select. Thus we can repeat this operations until A or B is empty. Moreover, since we delete the endpoint of the selected edge at each step, we never add in M an edge that share vertices with another edge of M. Thus M is a matching between V_1 and $\mathcal{N}(V_1)$.

Finally, observe that at each step, we only remove neighbors of v_1 from B and v_1 from A. Thus, we remove one vertex from A and at most d from B since G is of degree d. Thus, we iterate at least $|\mathcal{N}(V_1)|/d \geq |c|V|/(3d)$ times, that is, $|M| \geq c|V|/(3d)$. □

Theorem 6.24. *Let $G = (V, E)$ be a (c, d)-expander with $c \leq 1$ and let R be a balanced rectangle cover of F_G. There exists a constant $\alpha > 0$ such that*

$$|R| \geq 2^{\alpha|V|}.$$

Proof. Let $\alpha = \log\left(\frac{1+2^d}{2^d}\right)^{c/(3d)} = (c \cdot \log(1 + 2^{-d}))/(3d) > 0$. Let $r \in R$ with underlying balanced partition (V_1, V_2). By Lemma 6.23, there exists a matching M between V_1 and V_2 of size at least $c|V|/(3d)$. By Corollary 6.20, we have

$$|\mathsf{sat}(r)| \leq 2^{-\alpha|V|}|\mathsf{sat}(F_G)|.$$

Since R is a rectangle cover of F_G, we have $\bigcup_{r\in R} \mathsf{sat}(r) = \mathsf{sat}(F_G)$ thus $\sum_{r\in R} |\mathsf{sat}(r)| \geq |\mathsf{sat}(F_G)$. By the previous bound on $\mathsf{sat}(r)$, we have:

$$2^{-\alpha|V|}|R||\mathsf{sat}(F_G)| \leq |\mathsf{sat}(F_G)|.$$

It follows that

$$|R| \geq 2^{\alpha|V|}.$$

□

Using the connections between the number of balanced rectangle needed to cover a DNNF of Theorem 6.7 and Theorem 6.24, we can answer several open questions of [DM02] concerning the expressivity of DNNF:

Theorem 6.25. *There exists a family $(F_n)_{n\in\mathbb{N}}$ of monotone* 2-CNF *and a constant $\alpha > 0$ such that for every n, $\mathsf{DNNF}(F_n) > 2^{\alpha|F_n|}$.*

6.2.4 Corollaries

In this section we summarize the consequences of Theorem 6.25 on the knowledge compilation map of Darwiche and Marquis [DM02]. We start by showing that some queries cannot be supported efficiently by DNNF.

We start by showing that negating a DNNF may lead to an exponential blow-up:

Corollary 6.26. *There exists a family of* DNNF $(D_n)_{n\in\mathbb{N}}$ *such that for every* n, $\mathsf{DNNF}(\neg D_n) > 2^{\Omega(\mathsf{size}(D_n))}$.

Proof. We simply take the negation of the CNF family of Theorem 6.25. The negation of a CNF is a DNF, thus it can be written as a DNNF of the same size as the CNF. Negating this family again gives the lower bound. □

We now show that DNNF does not support conjunction with only a polynomial increase in size:

Corollary 6.27. *There exists families* $(D_{1,n})_{n\in N}$, $(D_{2,n})_{n\in N}$, $(D_{3,n})_{n\in N}$ *and* $(D_{4,n})_{n\in N}$ *of* DNNF *such that for every* n,

$$\mathsf{DNNF}(D_{1,n} \wedge D_{2,n} \wedge D_{3,n} \wedge D_{4,n}) > 2^{\mathsf{size}(D_{1,n})+\mathsf{size}(D_{2,n})+\mathsf{size}(D_{3,n})+\mathsf{size}(D_{4,n})}.$$

Proof. Vizing's theorem [Die12] states that for every graph $G = (V, E)$ of degree d, the there exists a coloring of its edges with at most $d + 1$ colors, that is, a function $\lambda : E \to [d + 1]$ such that if $e \cap f \neq \emptyset$, $\lambda(e) \neq \lambda(f)$. Take a family $(G_n = (V_n, E_n))_{n\in\mathbb{N}}$ of expander graphs of degree 3 and for all n, let λ_n be a 4-coloring the edges of G_n. We define for $i \leq 4$, $F_{i,n} = \bigwedge_{\lambda_n(\{x,y\})=i}(x \vee y)$ where $\{x, y\}$ ranger over E_n. Observe that $F_{i,n}$ has a DNNF $D_{i,n}$ of size $|E_n|$ since it is a conjunction of disjoint variable disjunctions. But $F_n = F_{1,n} \wedge F_{2,n} \wedge F_{3,n} \wedge F_{4,n}$ has no small DNNF by Theorem 6.25. □

Similarly, we can show that universal projections of DNNF may lead to an exponential blow-up:

Corollary 6.28. *There exists a family of* DNNF $(D_n)_{n\in\mathbb{N}}$ *such that for every* n, *there exist* $x_1, \ldots, x_p \in \mathsf{var}(D_n)$ *such that* $\mathsf{DNNF}(\forall x_1, \ldots, x_p, D_n) > 2^{\Omega(\mathsf{size}(D_n))}$ *and* $p = O(\log|\mathsf{size}(D_n)|)$.

Proof. Given a CNF $F = \bigwedge_{i=0}^{m-1} C_i$ and $Y = \{y_1, \ldots, y_k\}$ with $k = \lceil\log(m)\rceil$, we define the DNNF $D = \bigvee_{i=0}^{m}(\overline{y} = i \wedge C_i)$ where $\overline{y} = i$ encodes the fact that $y_1 \ldots y_k$ is the binary representation of i. It is easy to check that $\forall y_1, \ldots, y_k.D$ is equivalent to F. The corollary follows by applying this transformation to the family of CNF of Theorem 6.25. □

We can also use Theorem 6.25 to derive new separation results that were left open in [DM02]. A CNF formula F is said to be in *prime implicate form*, PI-form for short, if:

- No clause C of F entails another. In other words, for every $C, C' \in F$, if $C \neq C'$ then C does not imply C'.
- For every clause C entailed by F, there exists $C' \in F$ such that C' entails C.

It is readily verified that monotone 2-CNF are in PI-form. Thus Theorem 6.25 also separates DNNF from PI-form CNF:

Corollary 6.29. *There exists a family $(F_n)_{n\in\mathbb{N}}$ of* 2-CNF *in PI-form and a constant $\alpha > 0$ such that for every n,* $\mathsf{DNNF}(F_n) > 2^{\alpha|F_n|}$.

6.3 Separating structured DNNF from FBDD

For almost every compilation language, we have defined a structured version of it. When we deal with linear language, the structure is usually given as an order on the variables. This is the case with OBDD, which are FBDD plus a structural condition that enforces the variables to appear in a fixed order. For treelike language such as DNNF, the structural condition is a tree whose leaves are variables and which forces some uniformity in the way the $\wedge$-gates are partitioning the variables. It is known that there exists boolean functions which are efficiently computable by an FBDD but not by OBDD [Weg00]. The trick to separate such classes is to take the disjunction of two functions that can be both easily computed by an OBDD but each function uses two very different orders. The disjunction is easily computable by an FBDD but not by an OBDD. This suggests that adding structures usually lead to an exponential blow-up, even if the resulting function is just a disjunction of two easily computable functions.

Proving lower bounds on structure DNNF relies on understanding precisely the notion of vtree, which are harder to work with than order on variables. In [PD10b], Darwiche and Pipatsrisawat proposed a framework to prove lower bounds on the size of structured DNNF. Their framework can be understood as relating the size of a structured DNNF and the size of rectangle cover of its satisfying assignments where every rectangle in the cover share the same underlying partition.

In this section, we start by restating the result of Pipatsrisawat and Darwiche in the framework of rectangle covers. We then use a connection between the size of a matching across a partition of the variables of a graph CNF as in Section 6.2.3 and the number of rectangles needed to cover such functions to exhibit graph CNF that are hard to compute for structured DNNF. We use such hard functions to separate structured DNNF from FBDD. Such separation was already shown in the PhD thesis of Pipatsrisawat [Pip10] which is not publicly available. We reprove this result here using classical results in graph theory.

6.3.1 Rectangle covers of structured DNNF

In this section, we prove again the main theorem of [PD10b] connecting the size of a structured DNNF and the number of rectangles needed to cover its satisfying

assignments using a unique partition of the variables. More precisely, we prove the following theorem:

Theorem 6.30. *Let T be a vtree on variables X, t a vertex of T and D be a structured* DNNF *form respecting T. There exists a rectangle cover R of D such that $|R| \leq \mathsf{size}(D)$ and such that for every $r \in R$, the underlying partition of r is $(X_t, X \setminus X_t)$. Moreover, if D is deterministic, R may be assumed to be disjoint.*

We follow the same proof structure as for Theorem 6.7. We start by showing that each $\wedge$-gate of a structured DNNF defines a rectangle:

Lemma 6.31. *Let T be a vtree on variables X, t a vertex of T and D a structured* DNNF *respecting T. Let v be a gate of D such that $\mathsf{var}(D_v) \subseteq X_t$ and such that for every ancestor w of v, we have $\mathsf{var}(D_w) \not\subseteq X_t$. Then $\mathsf{sat}(D, v)$ is a rectangle whose underlying partition is $(X_t, X \setminus X_t)$.*

Proof. The proof is similar as the proof of Lemma 6.6. We only need to prove that for every $T \in \mathsf{cert}(D, v)$, we have $\mathsf{var}(T_v) \subseteq X_t$ and $\mathsf{var}(T) \setminus \mathsf{var}(T_v) \subseteq X \setminus X_t$. The fact that $\mathsf{var}(T_v) \subseteq X_t$ directly follows from the fact that $\mathsf{var}(T_v) \subseteq \mathsf{var}(D_v) \subseteq X_t$. Now let $x \in \mathsf{var}(T) \setminus \mathsf{var}(T_v)$. By definition, there exists an input u of T labeled with variable x that is not in D_v. Let w be the least common ancestor of u and v in T. It is necessarily a vertex of degree 2 in T, that is, w is an $\wedge$-gate that is an ancestor of v in D. By hypothesis, D respects T and $\mathsf{var}(D_w) \not\subseteq X_t$. Thus w has to respect a vertex t' of T that is an ancestor of t. Moreover, if t_1, t_2 are the children of t' in T, we have $X_{t_1} \cap X_{t_2} = \emptyset$, $x \in X_{t_1}$ and $X_t \subseteq X_{t_2}$. That is $x \notin X_t$. Thus $\mathsf{var}(T) \cap X_t = \emptyset$. In other words, $\mathsf{var}(T) \subseteq X \setminus X_t$.

The rest of the proof is similar to the proof of Lemma 6.6. □

We now show that gates such as in Lemma 6.31 almost always exists in structured DNNF:

Lemma 6.32. *Let T be a vtree on variables X, t a vertex of T and D a structured* DNNF *respecting T. If $X_t \cap \mathsf{var}(D) \neq \emptyset$, then there exists a gate v in D such that $\mathsf{var}(D_v) \subseteq X_t$ and such that for every ancestor w of v, we have $\mathsf{var}(D_w) \not\subseteq X_t$.*

Proof. Since $\mathsf{var}(D) \cap X_t \neq \emptyset$, there exists an input u of D labeled with a literal ℓ such that $\mathsf{var}(\ell) \in X_t$. We have $\mathsf{var}(D_u) \subseteq X_t$. Thus the set $S = \{u \mid \mathsf{var}(D_u) \subseteq X_t\}$ is not empty. We choose v to be an element of S that has no ancestor in S. □

We can now prove Theorem 6.30 in a similar fashion as Theorem 6.7.

Proof (of Theorem 6.30). By iteratively applying Lemma 6.32, we get a sequence $v_1, \ldots, v_k$ of gates such that $\mathsf{var}(D^i_{v_{i+1}}) \subseteq X_t$, where $D^i = D \setminus \{v_1, \ldots, v_i\}$ for every $i < k$ and such that $\mathsf{var}(D^k) \cap X_t = \emptyset$.

By Corollary 6.9, it holds that

$$\mathsf{sat}(D) = \mathsf{sat}(D^k) \cup \bigcup_{i=1}^{k-1} \mathsf{sat}(D^i, v_{i+1}).$$

By Lemma 6.31, for every $i < k$, $\mathsf{sat}(D^i, v_{i+1})$ is a rectangle whose underlying partition is $(X_t, X \setminus X_t)$. Moreover, D^k does not depends on variables in X_t. Thus, D^k can be seen as a rectangle whose underlying partition is $(X_t, X \setminus X_t)$. We thus have a rectangle cover of size at most $\mathsf{size}(D)$ of D for which each rectangle has the same underlying partition $(X_t, X \setminus X_t)$.

If D is deterministic, then by Corollary 6.9 again,

$$\mathsf{sat}(D) = \mathsf{sat}(D^k) \uplus \biguplus_{i=1}^{k-1} \mathsf{sat}(D^i, v_{i+1}).$$

This gives a disjoint rectangle cover of D for which each rectangle has the same underlying partition $(X_t, X \setminus X_t)$. □

6.3.2 Rectangle covers of graph CNF

In this section, we study the complexity of graph CNF for structured DNNF. The key observation is that a vtree for a graph CNF can be seen as a branch decomposition of the underlying graph. This connection allows us to construct graphs having no small structured DNNF. From this, we derive a separation of FBDD and structured DNNF.

We start by showing a relation between the existence of a large matching across a partition of the vertices of a graph CNF and the number of rectangles needed to cover the formula if they all have the same underlying partition:

Lemma 6.33. *Let $G = (V, E)$ be a graph, (X, Y) a partition of V and M an induced matching of G across X and Y. The number of rectangles with underlying partition (X, Y) needed to cover F_G is at least $2^{|M|}$.*

Proof. Let $M = \{(x_1, y_1), \ldots, (x_k, y_k)\}$ with $x_i \in X$ and $y_i \in Y$ for every $i \leq k$. For $K \subseteq [k]$, we define τ_K^X to be the truth assignment of X such that for all $x \in X$, $\tau_K^X(x) = 0$ if and only if $x = x_i$ for some $i \in K$. Otherwise, we let $\tau_K^X(x) = 1$. We define τ_K^Y to be the truth assignment of Y such that for all $y \in Y$, $\tau_K^Y(y) = 0$ if and only if $y = y_i$ for some $i \in K$. Otherwise, we let $\tau_K^Y(y) = 1$. We let $\tau_K = \tau_K^X \cup \tau_{\overline{K}}^Y$.

For every K, it holds that $\tau_K \models F_G$. Indeed, for every $e \in E$, either $e \in M$, that is $e = (x_i, y_i)$ for some i and the clause $x_i \vee y_i$ is satisfied by τ since if $i \in K$ then $\tau_K(y_i) = 1$ and if $i \notin K$, $\tau_K(x_i) = 1$. If $e \notin M$, then at least one of its endpoint z is not in $V(M)$ since M is an induced matching. It holds that $\tau_K(z) = 1$, thus the corresponding clause in F_G is satisfied by τ_K.

We now show that if r is a rectangle with underlying partition (X, Y) such that $\mathsf{sat}(r) \subseteq \mathsf{sat}(F_G)$ and such that $\tau_K \in \mathsf{sat}(r)$, then for every $K' \subseteq [k]$, $\tau_{K'} \in \mathsf{sat}(r)$

if and only if $K' = K$. Indeed, if there exists $K' \neq K$ such that $\tau_{K'} \in \mathsf{sat}(r)$, we would have $\tau_K^X \cup \tau_{\overline{K'}}^Y \in \mathsf{sat}(r)$ and $\tau_{K'}^X \cup \tau_{\overline{K}}^Y \in \mathsf{sat}(r)$ since r is a rectangle with underlying partition (X, Y). Since $K' \neq K$, there exists $i \in K \Delta K'$. Assume without lost of generality that $i \in K \setminus K'$. Thus $\tau_K^X \cup \tau_{\overline{K'}}^Y$ maps both x_i and y_i to 0. Thus it is not a satisfying assignment of F_G which contradicts the fact that $\mathsf{sat}(r) \subseteq \mathsf{sat}(F_G)$.

Thus we need one rectangle with underlying partition (X, Y) for each $K \subseteq [k]$ to cover F_G, that is, we need at least 2^k rectangles with underlying partition (X, Y) to cover F_G. □

If G has a high MIM-width, then for every branch decomposition of G, we may find a large induced matching across the leaves of a subtree of the branch decomposition and the other vertices of G. Combining this with Lemma 6.33 and Theorem 6.30 yields the following:

Theorem 6.34. *Let $G = (V, E)$ be a graph and T a branch decomposition of G of* MIM*-width k. Any structured* DNNF *respecting T and computing F_G is of size at least 2^k.*

Proof. Let D be a DNNF computing F_G and respecting T. We show that D is of size at least 2^k. The tree T is a branch decomposition of G. Since $\mathsf{MIM\text{-}width}(T) = k$, there exists a vertex t of T and an induced matching M of size k across $X_t = L(T_t)$ and $Y_t = V \setminus X_t$. By Theorem 6.30, there exists a cover R of F_G with rectangles having the underlying partition (X_t, Y_t) such that $|R| \leq \mathsf{size}(R)$. Besides, by Lemma 6.33, any rectangle cover of F with underlying partition (X_t, Y_t) is of size $2^{|M|} = 2^k$. It means that $|R| \geq 2^k$ and therefore $\mathsf{size}(D) \geq 2^k$. □

An immediate corollary of Theorem 6.34 is that high MIM-width graphs CNF have no small DNNF:

Corollary 6.35. *Let $G = (V, E)$ be a graph of* MIM*-width k. Any structured* DNNF *computing F_G is of size at least 2^k.*

6.3.3 Separations

Corollary 6.35 gives numerous boolean functions that are hard for structured DNNF but it is not sufficient separate FBDD from structured DNNF since these functions are likely to be hard for FBDD as well. To construct a boolean function that is easy for FBDD but hard for DNNF, we use a disjunction on simple functions that all have a small structured DNNF (and even a small OBDD) but they all needs a very different structure, making their disjunction hard for structured DNNF but easy for FBDD. To do so, we start from a graph having a high MIM-width and split it into disjoint matchings.

We start by illustrating this process on grids. Let $V = \{x_{i,j} \mid 1 \leq i, j \leq 2n\}$ and $G = (V, E)$ be the $2n \times 2n$. We split the edges of G into four graphs:

- $G_0 = (V, E_0)$ with $E_0 = \{(x_{i,2j+1}, x_{i,2j+2}) \mid 1 \leq i \leq 2n, 0 \leq j \leq n-1\}$,
- $G_1 = (V, E_1)$ with $E_1 = \{(x_{i,2j}, x_{i,2j+1}) \mid 1 \leq i \leq 2n, 1 \leq j \leq n-1\}$,
- $G_2 = (V, E_2)$ with $E_2 = \{(x_{2i+1,j}, x_{2i+2,j}) \mid 0 \leq i \leq n-1, 1 \leq j \leq 2n\}$
- $G_3 = (V, E_3)$ with $E_3 = \{(x_{2i,j}, x_{2i+1,j}) \mid 1 \leq i \leq n-1, 1 \leq j \leq 2n\}$.

Observe that G_i is a matching of G for each $i \leq 3$. We define E_n as:

$$E_n \equiv (\neg l_0 \wedge \neg l_1 \wedge F_{G_0}) \vee (l_0 \wedge \neg l_1 \wedge F_{G_1}) \vee (\neg l_0 \wedge l_1 \wedge F_{G_2}) \vee (l_0 \wedge l_1 \wedge F_{G_3})$$

Observe that the clauses of F_{G_0} have disjoint variables. Thus it can be computed by an OBDD of size $O(n^2)$ by testing that each clause is satisfied consecutively. The same is true for F_{G_1}, F_{G_2} and F_{G_3}. This gives an FBDD for E_n of size $O(n^2)$: test the value of l_0 and l_1. If $l_0 = 0$ and $l_1 = 0$, then plug the OBDD for F_{G_0}. More generally, if $l_0 = i$ and $l_1 = j$, plug the OBDD for $F_{G_{i+2j}}$. The resulting FBDD is of size $O(n^2)$ and computes E_n. We now show that there is no small structured DNNF computing E_n:

Proposition 6.36. *Let $n \in \mathbb{N}$. Every structured* DNNF *computing E_n is of size at least $2^{n/6}$.*

Proof. Let T be a vtree for V. We show that every DNNF computing E_n and respecting T is of size $2^{n/6}$ at least. We see T as a branch decomposition of G. Since G is of tree width $2n$, there exists by Lemma 1.29 a vertex t of T and a matching M across $L(T_t)$ and $V \setminus L(T_t)$ of size $2n/3$.

Let $M_i = M \cap E_i$. Since $M = \bigcup_{i \leq 3} M_i$, there exists $i_0 \leq 3$ such that $|M_{i_0}| \geq |M|/4 \geq n/6$. Moreover, M_{i_0} is an induced matching of G_{i_0} since G_{i_0} is itself a matching, across $L(T_t)$ and $V \setminus L(T_t)$. Thus by Theorem 6.34, any DNNF computing $F_{G_{i_0}}$ and respecting T is of size at least $2^{n/6}$.

Let D be a DNNF computing E_n and respecting T. By conditioning D with $\{l_0 \mapsto i_0 \mod 2, l_1 \mapsto \lfloor i_0/2 \rfloor\}$ (see Proposition 1.53), we get a DNNF D', respecting T, computing $F_{G_{i_0}}$ and such that $\mathsf{size}(D') \leq \mathsf{size}(D)$. From what precedes, $2^{n/6} \leq \mathsf{size}(D')$ that is $2^{n/6} \leq \mathsf{size}(D)$. □

This leads to the following separation:

Theorem 6.37. *There exists an infinite family of* FBDD $(F_n)_{n \in N}$ *such that for all $n \in N$, any structured* DNNF *equivalent to F_n is of size at least $2^{\Omega(\sqrt{\mathsf{size}(F_n)})}$.*

The construction we did on grids can be generalized to any graph: given a graph $G = (V, E)$ and a k-coloring $w : E \to [k]$ of the edges, one can consider the graphs $G_i = (V, E_i)$ where $E_i = w^{-1}(i)$ and the formula $F_{G,w} = \bigvee_{i=1}^{k} (Eq(L, i) \wedge F_{G_i})$ where $L = \{\ell_0, \ldots, \ell_p\}$ is a fresh set of selector variables with $p = \lfloor \log k \rfloor$ and $Eq(L, j) = \bigwedge_{i \in I} \ell_i \wedge \bigwedge_{i \notin I} \neg \ell_i$ where $j = \sum_{i \in I} 2^i$. That is $Eq(L, j) = 1$ if and only if $j = \sum_{i=0}^{p} \ell_i 2^i$.

Since G_i is a matching (w is a coloring), F_{G_i} can be computed by a linear size OBDD. Just test each clauses one after the other. Thus, $F_{G,w}$ can be computed by a small FBDD: test each ℓ_i, and depending on the value of $j = \sum_{i=0}^{p} \ell_i 2^i$, plug the OBDD for F_{G_j}.

Moreover, we have the following:

Lemma 6.38. *If $G = (V, E)$ is a graph of tree width p and $w : E \to [k]$ is a k-coloring of the edges of G, then any structured* DNNF *computing $F_{G,w}$ is of size at least $2^{p/(3k)}$.*

Proof. Let T be a vtree for V. We show that every DNNF respecting T and computing $F_{G,w}$ is of size at least $2^{p/(3k)}$. We see T as a branch decomposition of G. Since G is of tree width p, by Lemma 1.29, there exists a vertex t of $\mathcal{T}$ and a matching M of G across $L(T_t)$ and $V \setminus L(T_t)$ of size at least $p/3$. Let $M_i = E_i \cap M$. Since $M = \bigcup_{i \leq k} M_i$, there exists $j \leq k$ such that M_j is of size at least $|M|/(3k) \geq p/(3k)$. M_j is an induced matching of G_j of size $p/(3k)$ since G_j is a matching. Thus by Theorem 6.34, any DNNF computing F_{G_i} and respecting T is of size at least $2^{p/(3k)}$. Since F_{G_i} is a projection of $F_{G,w}$, any DNNF computing $F_{G,w}$ and respecting T is of size at least $2^{p/(3k)}$. □

This lemma can be used to lift the bound of Theorem 6.37 to a strongly exponential one: instead of using grids, one can use (c, d)-expanders, whose tree width is linear in the number of edges [GM09]. Moreover, since they are of bounded degree d, one can find a $(d+1)$-coloring of the edges by Vizing's theorem [Die12]. Applying Lemma 6.38 to expander graphs with such coloring yields a strong separation of FBDD and structured DNNF.

Theorem 6.39. *There exists an infinite family of* FBDD *$(F_n)_{n \in N}$ such that for all $n \in N$, any structured* DNNF *equivalent to F_n is of size at least $2^{\Omega(\mathsf{size}(F_n))}$.*

Theorem 6.39 separates one of the most general structures language from the less general unstructured language we have introduce in Chapter 1. Thus, the following separations directly follows from Theorem 6.39: (decision, deterministic) DNNF are exponentially more succinct than their structured restrictions.

6.4 Conclusion

We conclude this chapter by giving perspectives and open questions. There is still a few separations that are not known to hold unconditionally. One of the most promising direction is to separate d-DNNF from DNNF. Indeed, Theorem 6.7 states that d-DNNF can be covered with few *disjoint* rectangles. Thus, constructing a family of DNNF that is hard to cover with disjoint rectangles would yield the desired upper bounds. Sauerhoff gives in [Sau03] such lower bound. We explain how his result can be used to separate d-DNNF from DNNF and what is left to

do. The *Sauerhoff function* $S_n\colon \{0,1\}^{n^2} \to \{0,1\}$ is defined on the $n \times n$ matrix $X = (x_{ij})_{1\leq i,j\leq n}$ of variables by

$$S_n(X) = R_n(X) \vee C_n(X) \tag{6.6}$$

where $R_n, C_n\colon \{0,1\}^{n^2} \to \{0,1\}$ are defined by

$$R_n(X) = \bigoplus_{i=1}^{n} \mathsf{mod}_3^n(x_{i1}, x_{i2}, \ldots, x_{in})$$

and $C_n(X) = R_n(X^\top)$, where $X^\top$ denotes the transpose of X, $\oplus$ denotes addition modulo 2 and $\mathsf{mod}_3^n\colon \{0,1\}^n \to \{0,1\}$ is the function evaluating to 1 if and only if the sum of its inputs is divisible by 3.

The Sauerhoff function has polynomial DNNF size:

Proposition 6.40. *S_n has DNNF size $O(n^2)$.*

Proof (Sketch). The functions R_n and C_n have OBDD of size $O(n^2)$, ordering the variables by rows and columns, respectively; their disjunction has size $O(n^2)$. □

A partition (X_1, X_2) of a set X is said to be *strictly balanced* if $|X_1| = |X_2| \pm 1$. A rectangle is strictly balanced if its underlying partition is strictly balanced. Sauerhoff has proven in [Sau03] the following lower bound:

Theorem 6.41 (Sauerhoff)**.** *Any* strictly balanced *disjoint rectangle cover of the Sauerhoff function S_n has size $2^{\Omega(n)}$.*

Unfortunately, we cannot use Theorem 6.41 as it is to prove a separation of DNNF and d-DNNF since Theorem 6.7 only gives upper bounds on the size of balanced rectangle cover of d-DNNF. The proof of Theorem 6.41 in [Sau03] seems to still work for the less restrictive notion of balanced rectangles. However, we still have not check every details nor rewritten the proof, we thus let it as an open question:

Open question 9. *Separate* d-DNNF *and* DNNF *using the bound of Sauerhoff and Theorem 6.7.*

Another intriguing open question would be to separate d-DNNF from DNF. The separation of DNF from dec-DNNF is already known [BLRS13] but the proof uses a simulation of dec-DNNF by FBDD to leverage lower bounds on FBDD, which make the proof hard to generalize since such a simulation cannot holds for d-DNNF since dec-DNNF and d-DNNF are exponentially separated. It is not clear how far rectangle covers techniques can be pushed in this case.

Open question 10. *Separate* DNF *from* d-DNNF *or prove a that* d-DNNF *can express* DNF *with a quasi-polynomial increase.*

Index

certificate, 141
CNF-formula, 5, 40
  clause, 5
  k-CNF, 7
  dual graph, 40
  empty clause ($\bot$), 6
  hypergraph, 41
  incidence graph, 41
  literal, 5
  monotone, 7
  primal graph, 40
  satisfiable, 5
  satisfying assignment, 5
  signed incidence graph, 42
  size, 7
communication complexity, 144
Complexity classes, 3
  FP, 8
  FPT, 10
  NP, 3
  P, 3
  #P, 8
  parametrization, 10
  reduction, 4, 11
  W[1], 11
  XP, 10
constraint satisfaction problems, 106
counting complexity, 8
CSP, 106

DNF
  DNF, 7

Graphs, 12
  bipartite, 13
  chord, 22
  chordal bipartite, 22
  clique, 13
  connected component, 13
  cycle, 12
  decomposition, 15
    branch decomposition, 18
    clique width (**cw**), 17
    elimination order for tree width, 16
    MIM-width (**mimw**), 19
    MM-width (**mmw**), 19
    PS-width (**psw**), 58
    rank-width (**rw**), 63
    signed clique width (**scw**), 64
    tree decomposition, 15
    tree width (**tw**), 15
  expander, 155
  matching, 13
  neighborhood diversity (**nd**), 67
  path, 12
  subgraph, 13
    induced, 13
  tree, 13
    rooted, 13
    rooted subtree, 14
  vertex cover, 13

Hypergraphs, 14
  acyclicity, 19
    α-acyclicity, 20
    α-elimination order, 20
    Berge acyclicity, 20
    β-acyclicity, 21
    β-elimination order, 21
    db-rootable, 54
    disjoint branches, 45

γ-acyclicity, 22
join path, 50
join tree, 20
connected component, 14
decomposition, 23
cover width (**covw**), 133
elimination order for **covw**, 133
generalized hypertree decomposition, 23
generalized hypertree width (**ghtw**), 23
hypertree decomposition, 24
hypertree width (**htw**), 24
β-hypertree width (β-**htw**), 24
incidence graph, 14
path, 14
primal graph, 14
subhypergraph, 15
induced, 15

Knowledge compilation, 25
decision node, 31
decomposable, 30
determinism, 31
DNNF, 29, 30
decision (dec-DNNF), 31
deterministic (d-DNNF), 31
structured, 32
FBDD, 27
negation normal form (NNF), 29
OBDD, 28
query, 26
representation language, 26
structuredness, 32
succinctness, 26
transformation, 26
variable tree (vtree), 32

parametrized complexity, 9
PQ-trees, 51

RAM machine, 2
rectangle, 145
balanced, 145
cover, 145
disjoint, 145
resolution, 100
DP-resolution, 101
pigeon hole principle, 101
resolvent, 100
$\mathsf{res}(F, x)$, 100

SAT, 5
k-SAT, 7
#SAT, 8
shape, 74
generate, 75

weighted constraint, 107
$\#\mathrm{CSP}_{\mathrm{def}}$, 108
generator, 123
size, 108
structural size, 108
support, 108

Bibliography

[AB87] N. Alon and R. B. Boppana. The monotone circuit complexity of boolean functions. *Combinatorica*, 7(1):1–22, 1987. 149, 150

[AB09] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, New York, NY, USA, 1st edition, 2009. 2, 6

[ACF10] J. M. Astesana, L. Cosserat, and H. Fargier. Constraint-based vehicle configuration: A case study. In *Tools with Artificial Intelligence (ICTAI)*, volume 1, pages 68–75. IEEE, 2010. xiv

[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. 22

[AGCL12] C. Ansótegui, J. Giráldez-Cru, and J. Levy. The community structure of sat formulas. In *Theory and Applications of Satisfiability Testing*, pages 410–423. Springer, 2012. xi

[AGG07] I. Adler, G. Gottlob, and M. Grohe. Hypertree width and related hypergraph invariants. *European Journal of Combinatorics*, 28(8):2167 – 2181, 2007. 24

[APT79] B. Aspvall, M. F. Plass, and R. E. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3):121–123, March 1979. xii, 9, 103

[AR11] M. Alekhnovich and A. Razborov. Satisfiability, Branch-Width and Tseitin tautologies. *Computational Complexity*, 20(4):649–678, November 2011. 103

[AS00] N. Alon and J. H. Spencer. *The Probabilistic Method*. Wiley, 2000. 155

[AS12] G. Audemard and L. Simon. Glucose 2.1: Aggressive-but reactive-clause database management, dynamic restarts. In *International Workshop of Pragmatics of SAT (Affiliated to SAT)*, 2012. xi

[BB12] J. Brault-Baron. A Negative Conjunctive Query is Easy if and only if it is Beta-Acyclic. In *Computer Science Logic - 21st Annual Conference of the EACSL*, volume 16 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 137–151. Schloss Dagstuhl, 2012. 117

[BB14] J. Brault-Baron. Hypergraph acyclicity revisited. *ArXiv e-prints*, March 2014. 19

[BCM15] J. Brault-Baron, F. Capelli, and S. Mengel. Understanding model counting for beta-acyclic CNF-formulas. In *32nd International Symposium on Theoretical Aspects of Computer Science*, volume 30 of *LIPIcs*, pages 143–156. Schloss Dagstuhl, 2015. xiii, xvi, 86

[BCMS14] S. Bova, F. Capelli, S. Mengel, and F. Slivovsky. Expander cnfs have exponential DNNF size. *CoRR*, abs/1411.1995, 2014. xv

[BCMS15] S. Bova, F. Capelli, S. Mengel, and F. Slivovsky. On Compiling CNFs into Structured Deterministic DNNFs. In *Theory and Applications of Satisfiability Testing*, Lecture Notes in Computer Science, pages 199–214. Springer International Publishing, September 2015. xv, xvi, 74, 85

[Ber85] C. Berge. *Graphs and Hypergraphs*. Elsevier Science Ltd., 1985. 20

[BFMY83] C. Beeri, R. Fagin, D. Maier, and M. Yannakakis. On the desirability of acyclic database schemes. *J. ACM*, 30(3):479–513, July 1983. xii, 20, 21, 22

[BL76] K. Booth and G. Lueker. 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. 49, 50, 52

[BL15] P. Beame and V. Liew. New limits for knowledge compilation and applications to exact model counting. In *Proceedings of the Thirty-First Conference on Uncertainty in Artificial Intelligence*, pages 131–140, 2015. 144

[BLM07] M. L. Bonet, J. Levy, and F. Manyà. Resolution for Max-SAT. *Artificial Intelligence*, 171(8–9):606–618, June 2007. 106

[BLRS13] P. Beame, J. Li, S. Roy, and D. Suciu. Lower bounds for exact model counting and applications in probabilistic databases. In *Proceedings of the Twenty-Ninth Conference on Uncertainty in Artificial Intelligence*, 2013. 32, 140, 164

[BLRS14] P. Beame, J. Li, S. Roy, and D. Suciu. Counting of query expressions: Limitations of propositional methods. In *Proc. 17th International Conference on Database Theory (ICDT)*, pages 177–188, 2014. 140

[Bod93a] H. L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, STOC '93, pages 226–234. ACM, 1993. 16, 134

[Bod93b] H. Bodlaender. A tourist guide through treewidth. *Acta Cybern.*, 11(1-2):1–21, 1993. 15

[Bod06] H. L. Bodlaender. *Treewidth: Characterizations, Applications, and Computations*, page 1–14. Lecture Notes in Computer Science. Springer Berlin Heidelberg, Jun 2006. 16, 17

[Bry92] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24:293–318, 1992. 31

[BSW99] E. Ben-Sasson and A. Wigderson. Short proofs are narrow—resolution made simple. In *Proceedings of the Thirty-first Annual ACM Symposium on Theory of Computing*, STOC '99, pages 517–526. ACM, 1999. 103

[Buk11] L. Bukowski. 58, 2011. featuring Kacem Wapalek Anton Serra NAdir, chez Oster Lapwass. iv

[Bü00] P. Bürgisser. *Completeness and Reduction in Algebraic Complexity Theory*, volume 7 of *Algorithms and Computation in Mathematics*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000. 141

[CDLS02] M. Cadoli, F. M. Donini, P. Liberatore, and M. Schaerf. Preprocessing of intractable problems. *Information and Computation*, 176(2):89 – 120, 2002. 26

[CDM14] F. Capelli, A. Durand, and S. Mengel. Hypergraph Acyclicity and Propositional Model Counting. In *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference*, pages 399–414, 2014. xiii, xvi, 39, 44, 56, 66, 85

[CER91] B. Courcelle, J. Engelfriet, and G. Rozenberg. *Graph Grammars and Their Application to Computer Science*, chapter Context-free handle-rewriting hypergraph grammars, pages 253–268. Springer Berlin Heidelberg, 1991. 17

[CG10] H. Chen and M. Grohe. Constraint satisfaction with succinctly specified relations. *Journal of Computer and System Sciences*, 76(8):847 – 860, 2010. 107

[CGH09] D. Cohen, M. Green, and C. Houghton. Constraint representations and structural tractability. In *Principles and Practice of Constraint Programming*, pages 289–303, 2009. 107

[CH96] N. Creignou and M. Hermann. Complexity of Generalized Satisfiability Counting Problems. *Information and Computation*, 125:1–12, February 1996. xii, 39

[Che05] H. Chen. Parameterized compilability. In *International Joint Conference on Artificial Intelligence*, volume 19, page 412, 2005. 26

[Che06] H. Chen. Logic column 17: A rendezvous of logic, complexity, and algebra. *CoRR*, abs/cs/0611018, 2006. 107

[CO00] B. Courcelle and S. Olariu. Upper bounds to the clique width of graphs. *Discrete Applied Mathematics*, 101(1–3):77 – 114, 2000. 18

[Coo71] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971. xi, 4, 5, 6

[Dar01a] A. Darwiche. Decomposable negation normal form. *J. ACM*, 48(4):608–647, 2001. 30, 33, 34, 83

[Dar01b] A. Darwiche. On the tractable counting of theory models and its application to truth maintenance and belief revision. *Journal of Applied Non-Classical Logics*, 11(1-2):11–34, 2001. 31, 73

[DF12] R. G. Downey and M. R. Fellows. *Parameterized complexity*. Springer Science & Business Media, 2012. 10

[DHJ+04] P. Duriš, J. Hromkovič, S. Jukna, M. Sauerhoff, and G. Schnitger. On multi-partition communication complexity. *Information and Computation*, 194(1):49–75, October 2004. 145

[Die12] R. Diestel. *Graph Theory, 4th Edition*, volume 173 of *Graduate texts in mathematics*. Springer, 2012. 12, 88, 157, 163

[DKL+15] H. Dell, E. J. Kim, M. Lampis, V. Mitsou, and T. Mömke. Complexity and Approximability of Parameterized MAX-CSPs. In *10th International Symposium on Parameterized and Exact Computation*, volume 43 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 294–306. Schloss Dagstuhl, 2015. 67, 68

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962. xi

[DM02] A. Darwiche and P. Marquis. A Knowledge Compilation Map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002. xiv, xv, xvii, 25, 35, 139, 156, 157

[DP60] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *J. ACM*, 7(3):201–215, July 1960. xi, xiii, xvi, 99, 101

[Dur09] D. Duris. *Acyclicité des hypergraphes et liens avec la logique sur les structures relationnelles finies.* PhD thesis, Université Paris Diderot - Paris 7, 2009. 19

[Dur12] D. Duris. Some characterizations of γ and β-acyclicity of hypergraphs. *Inf. Process. Lett.*, 112(16):617–620, 2012. 22, 44, 45, 46

[ES03] N. Eén and N. Sörensson. An extensible sat-solver. In *Theory and applications of satisfiability testing*, pages 502–518. Springer, 2003. xi

[Fag83] R. Fagin. Degrees of acyclicity for hypergraphs and relational database schemes. *Journal of the ACM*, 30(3):514–550, 1983. xii, 19

[FG04] J. Flum and M. Grohe. The parameterized complexity of counting problems. *SIAM Journal on Computing*, 33(4):892–922, 2004. 12

[FG06] J. Flum and M. Grohe. *Parameterized Complexity Theory.* Springer-Verlag New York Inc, 2006. 10, 11, 12, 15

[FMR08] E. Fischer, J. Makowsky, and E. Ravve. Counting truth assignments of formulas of bounded tree-width or clique-width. *Discrete Applied Mathematics*, 156(4):511–529, 2008. 42, 60, 64, 65

[FRRS09] M. R. Fellows, F. A. Rosamond, U. Rotics, and S. Szeider. Clique-width is np-complete. *SIAM J. Discret. Math.*, 23(2):909–939, May 2009. 18

[Gav75] F. Gavril. A recognition algorithm for the intersection graphs of directed paths in directed trees. *Discrete Mathematics*, 13(3):237–249, 1975. 49

[GLS99] G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions and tractable queries. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 21–32. ACM, 1999. xii, 23, 24

[GLS01a] G. Gottlob, N. Leone, and F. Scarcello. The Complexity of Acyclic Conjunctive Queries. *J. ACM*, 48(3):431–498, May 2001. xii

[GLS01b] G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions: A survey. In *Mathematical Foundations of Computer Science 2001*, pages 37–57. Springer, 2001. 23

[GM09] M. Grohe and D. Marx. On tree width, bramble size, and expansion. *J. Comb. Theory, Ser. B*, 99(1):218–228, 2009. 88, 163

[GMS09] G. Gottlob, Z. Miklós, and T. Schwentick. Generalized hypertree decompositions: Np-hardness and tractable variants. *J. ACM*, 56(6), September 2009. 23

[GP04] G. Gottlob and R. Pichler. Hypergraphs in Model Checking: Acyclicity and Hypertree-Width versus Clique-Width. *SIAM Journal on Computing*, 33(2), 2004. 22, 57, 68, 88

[Gra96] E. Grandjean. Sorting, linear time and the satisfiability problem. *Annals of Mathematics and Artificial Intelligence*, 16(1):183–236, 1996. 2

[Hak85] A. Haken. The intractability of resolution. *Theoretical Computer Science*, 39:297–308, 1985. 103

[HD05] J. Huang and A. Darwiche. DPLL with a trace: From SAT to knowledge compilation. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, pages 156–162, 2005. xvii, 73, 139

[HO08] P. Hliněný and S. Oum. Finding Branch-Decompositions and Rank-Decompositions. *SIAM Journal on Computing*, 38(3):1012–1032, January 2008. 63

[JS02] S. Jukna and G. Schnitger. Triangle-freeness is hard to detect. *Combinatorics, Probability & Computing*, 11(6):549–569, 2002. 140, 145, 151, 152

[Juk12] S. Jukna. *Boolean Function Complexity - Advances and Frontiers*, volume 27 of *Algorithms and combinatorics*. Springer, 2012. 149

[Kar72] R. M. Karp. *Reducibility among combinatorial problems*. Springer, 1972. xi

[KGS13] B. Kenig, A. Gal, and O. Strichman. A new class of lineage expressions over probabilistic databases computable in p-time. In *Scalable Uncertainty Management*, pages 219–232. Springer, 2013. 49

[KN97] E. Kushilevitz and N. Nisan. *Communication Complexity*. Cambridge University Press, 1997. 144

[Kro67] M. R. Krom. The Decision Problem for a Class of First-Order Formulas in Which all Disjunctions are Binary. *Mathematical Logic Quarterly*, 13:15–20, January 1967. 103

[KS96] H. Kautz and B. Selman. Knowledge compilation and theory approximation. *Journal of the ACM*, 43:193–224, 1996. 139

[Kus97] E. Kushilevitz. Communication complexity. *Advances in Computers*, 44:331–360, 1997. 144

[Lam10] M. Lampis. *18th Annual European Symposium*, chapter Algorithmic Meta-theorems for Restrictions of Treewidth, pages 549–560. 2010. 67

[Lev73] L. A. Levin. Universal sequential search problems. *Problemy Peredachi Informatsii*, 9(3):115–116, 1973. xi, 5, 6

[Lub87] A. Lubiw. Doubly lexical orderings of matrices. *SIAM Journal on Computing*, 16(5):854–879, 1987. 22

[McC02] C. McCartin. Parameterized counting problems. In *Mathematical Foundations of Computer Science*, pages 556–567. Springer, 2002. 12

[MMZ+01] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001. xi

[MP06] G. Malod and N. Portier. Characterizing Valiant's Algebraic Complexity Classes. In R. Královič and P. Urzyczyn, editors, *Mathematical Foundations of Computer Science 2006*, number 4162 in Lecture Notes in Computer Science, pages 704–716. Springer Berlin Heidelberg, January 2006. 141

[OD14a] U. Oztok and A. Darwiche. CV-width: A New Complexity Parameter for CNFs. In *21st European Conference on Artificial Intelligence*, pages 675–680, 2014. 74

[OD14b] U. Oztok and A. Darwiche. On Compiling CNF into Decision-DNNF. In *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014*, pages 42–57, 2014. 74

[OPS13] S. Ordyniak, D. Paulusma, and S. Szeider. Satisfiability of acyclic and almost acyclic CNF formulas. *Theoretical Computer Science*, 481:85–99, 2013. xiii, 85, 99, 105, 106

[OS06] S.-i. Oum and P. Seymour. Approximating clique-width and branch-width. *Journal of Combinatorial Theory, Series B*, 96(4):514–528, July 2006. 63

[Pap94] C. Papadimitriou. *Computational Complexity*. Theoretical computer science. Addison-Wesley, 1994. 2

[Par03] B. Pargamin. Extending cluster tree compilation with non-boolean variables in product configuration: A tractable approach to preference-based configuration. In *Proceedings of the IJCAI*, volume 3. Citeseer, 2003. xiv

[PD08] K. Pipatsrisawat and A. Darwiche. New compilation languages based on structured decomposability. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI*, pages 517–522, 2008. 32, 35, 73

[PD10a] K. Pipatsrisawat and A. Darwiche. Top-down algorithms for constructing structured dnnf: Theoretical and practical implications. In *ECAI*, pages 3–8, 2010. 73

[PD10b] T. Pipatsrisawat and A. Darwiche. A Lower Bound on the Size of Decomposable Negation Normal Form. In *Twenty-Fourth AAAI Conference on Artificial Intelligence*, July 2010. xv, xvii, 140, 141, 158

[Per14] S. Perifel. *Complexité algorithmique.* Références sciences. Ellipses, 2014. 2

[Pip10] T. Pipatsrisawat. *Reasoning with Propositional Knowledge: Frameworks for Boolean Satisfiability and Knowledge Compilation.* PhD thesis, University of California Los Angeles, 2010. 140, 158

[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*, pages 55–66, 2013. xiii, 39, 68, 85

[PT87] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987. 22

[Raz14] I. Razgon. No small nondeterministic read-once branching programs for cnfs of bounded treewidth. In *Parameterized and Exact Computation - 9th International Symposium, IPEC*, pages 319–331, 2014. 153

[Rob65] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM*, 12(1):23–41, January 1965. 99

[Rot96] D. Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82(1–2):273 – 302, 1996. xii, 39

[RP13] I. Razgon and J. Petke. Cliquewidth and knowledge compilation. In *Theory and Applications of Satisfiability Testing - 16th International Conference*, pages 335–350, 2013. 73

[SA09] L. Simon and G. Audemard. Predicting learnt clauses quality in modern sat solver. In *Twenty-first International Joint Conference on Artificial Intelligence*, 2009. xi

[Sau03] M. Sauerhoff. Approximation of boolean functions by combinatorial rectangles. *Theor. Comput. Sci.*, 1-3(301):45–78, 2003. 163, 164

[SBB+04] T. Sang, F. Bacchus, P. Beame, H. A. Kautz, and T. Pitassi. Combining component caching and clause learning for effective model counting. *Theory and Applications of Satisfiability Testing*, 4:7th, 2004. xii

[Sch78] T. J. Schaefer. The Complexity of Satisfiability Problems. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, STOC '78, pages 216–226, New York, NY, USA, 1978. ACM. xii, 39

[Sch97] U. Schöning. Resolution proofs, exponential bounds, and Kolmogorov complexity. In I. Prívara and P. Ružička, editors, *Mathematical Foundations of Computer Science 1997*, number 1295 in Lecture Notes in Computer Science, pages 110–116. Springer Berlin Heidelberg, August 1997. 103

[Spi93] J. P. Spinrad. Doubly lexical ordering of dense 0–1 matrices. *Information Processing Letters*, 45(5):229–235, 1993. 22

[SS10] M. Samer and S. Szeider. Algorithms for propositional model counting. *Journal of Discrete Algorithms*, 8(1):50–64, 2010. xiii, 39, 60, 85, 131

[SS13] F. Slivovsky and S. Szeider. Model Counting for Formulas of Bounded Clique-Width. In *Algorithms and Computation - 24th International Symposium, ISAAC*, pages 677–687, 2013. xiii, 39, 74, 75, 76, 85

[Str10] Y. Strozecki. *Enumeration complexity and matroid decomposition.* PhD thesis, Université Paris Diderot - Paris 7, 2010. 2, 34

[STV14] S. H. Sæther, J. Telle, and M. Vatshelle. Solving MaxSAT and #SAT on structured CNF formulas. In *Theory and Applications of Satisfiability Testing*, pages 16–31, 2014. xiii, xiv, xv, xvi, 39, 57, 58, 59, 61, 73, 74, 76, 80, 82, 85

[Sze04] S. Szeider. On fixed-parameter tractable parameterizations of SAT. In E. Giunchiglia and A. Tacchella, editors, *Theory and Applications of Satisfiability, 6th International Conference*, volume 2919 of *LNCS*, pages 188–202. Springer, 2004. 85, 103

[Thu06] M. Thurley. sharpsat–counting models with advanced component caching and implicit bcp. In *Theory and Applications of Satisfiability Testing*, pages 424–429. Springer, 2006. xii

[TY84] R. E. Tarjan and M. Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.*, 13(3):566–579, July 1984. 20

[Urq87] A. Urquhart. Hard Examples for Resolution. *J. ACM*, 34(1):209–219, January 1987. 103

[Val79a] L. Valiant. The Complexity of Enumeration and Reliability Problems. *SIAM Journal on Computing*, 8(3):410–421, August 1979. 39

[Val79b] L. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8(2):189 – 201, 1979. 8, 9

[Vat12] M. Vatshelle. *New Width Parameters of Graphs*. PhD thesis, University of Bergen, 2012. 19

[Weg88] I. Wegener. On the complexity of branching programs and decision trees for clique functions. *Journal of the ACM (JACM)*, 35(2):461–471, 1988. 139

[Weg00] I. Wegener. *Branching Programs and Binary Decision Diagrams*. SIAM, 2000. 27, 28, 29, 32, 139, 144, 158

[Žák84] S. Žák. An exponential lower bound for one-time-only branching programs. In *Mathematical Foundations of Computer Science*, pages 562–566. Springer, 1984. 139