Université d'Artois

Habilitation à diriger des recherches

Spécialité Informatique

présentée par

Florent CAPELLI

Algorithmic Applications of Knowledge Compilation

Soutenue publiquement le *15 Juin 2026* devant le jury composé de :

M.	**Gilles**	**Audemard**	Rapporteur
M.	**Olaf**	**Beyersdorff**	Rapporteur
Mme.	**Luce**	**Brotcorne**	Examinatrice
M.	**Hubie**	**Chen**	Rapporteur
M.	**Pierre**	**Marquis**	Garant
M.	**Bruno**	**Zanuttini**	Examinateur

Abstract

This manuscript reviews the research I have pursued since defending my thesis in 2016. The main focus is to study how ideas and data structures from knowledge compilation, traditionally focusing on propositional logic and AI, can be applied to other fields of computer science.

Chapter 1 provides a broad introduction to knowledge compilation and introduces the main data structures and concepts that will be used throughout the manuscript. It also contains a few folklore results on knowledge compilation that have not, to the best of our knowledge, been explicitly proven in the literature.

Chapter 2 introduces a new data structure for knowledge compilation called Tree Decision Diagram (TDD). It can be seen as a generalization of OBDDs to tree-like ordering. We show that TDDs enjoy many interesting properties, such as the existence of a minimal canonical form that can be computed easily from any given TDD, unlocking a very simple bottom-up compilation algorithm. We use this algorithm to recover results such as the fact that bounded treewidth Boolean circuits can be transformed into FPT size deterministic DNNF circuit and that QBF is tractable on bounded treewidth instances. This new approach gives a new perspective on these results. Both can be explained by the fact that bounded treewidth circuits and QBF instances have a small number of subfunctions when following a well-chosen tree structure.

Chapter 3 reviews my work on using knowledge compilation to certify the output of #SAT solvers. We show how one can slightly modify the output of many #SAT-solvers so that instead of outputting only the number K of models, it outputs a certificate that the input formula has indeed K models. The correctness of the certificate can then be checked independently, improving our trust in the output of such complex software. We draw new connections between our early work on the subject and the MICE proof system and explain how all these proof systems can be seen as a way of syntactically certifying the equivalence between a CNF formula and a DNNF circuit.

Chapter 4 studied generalizations of data structures from knowledge compilation to non-binary domain. We show how they can be applied to represent the set of answers of database queries and how they can be used to rederive existing results. We first revisit the Yannakakis algorithm by casting it as a compilation algorithm and then show that a modified version of Exhaustive DPLL yields similar theoretical results. We show how this new perspective allows us to compile a broader class of queries than the Yannakakis algorithm, in particular the class of conjunctive queries with negation.

Finally, Chapter 5 shows how knowledge compilation ideas can be applied to the field of convex optimization. In this chapter, we focus on the notion of extended formulations, where the goal is to write down a set of linear constraints describing a given polyhedron, possibly over a space having a larger dimension. We define the convex hull of Boolean function f as the polyhedron given by the convex hull of its models, where models are seen as points in $[0,1]^n$. We show that when f is given as a DNNF circuit C, then its convex hull can be described with $O(|C|)$ linear constraints, providing a new way of automatically derive extended formulations. We apply this result to the Binary Polynomial Optimization problem and recover many existing tractability results using this new connection with knowledge compilation.

Résumé en français

Ce manuscrit est un résumé des recherches que j'ai menées depuis la soutenance de ma thèse en 2016. Il s'intéresse essentiellement à la manière dont les idées et structures de données issues de la compilation des connaissances, traditionnellement centrées sur la logique propositionnelle et l'IA, peuvent être appliquées à d'autres domaines de l'informatique.

Le premier chapitre propose une introduction générale au domaine de la compilation des connaissances et présente les principales structures de données et notions utilisées dans le manuscrit. Il contient également quelques résultats classiques qui n'ont pas, à notre connaissance, été démontrés explicitement dans la littérature.

Le second chapitre introduit une nouvelle structure de données pour la compilation des connaissances : les *Tree Decision Diagrams*, abbrégé en TDD. Ils peuvent être vus comme une généralisation des OBDD à des ordres arborescents. Nous montrons que les TDDs possèdent de nombreuses propriétés intéressantes, notamment l'existence d'une forme canonique minimale et facilement calculable à partir d'un TDD donné. Cette propriété permet de donner un algorithme de compilation très simple, compilant une formule CNF en un TDD en ajoutant successivement des clauses et en minimisant le diagramme. Nous utilisons cet algorithme pour retrouver plusieurs résultat, comme le fait que des circuits booléens de largeur arborescente bornée peuvent être transformés en circuit DNNF déterministes de taille FPT, ou que le problème QBF peut être efficacement résolu sur des instances de largeur arborescente bornée. Cette nouvelle approche donne un éclairage nouveau sur ces résultats : on peut les réexpliquer par l'existence d'une structure arborescente bien choisi qui découpe le calcul en un petit nombre de sous-fonctions.

Le troisième chapitre présente mes travaux sur l'utilisation de la compilation des connaissances pour certifier la sortie de solveurs #SAT. Nous montrons comment on peut modifier la sortie de nombreux solveurs afin qu'ils ne se contentent pas de produire le nombre K de modèles, mais donnent également un certificat attestant que la formule d'entrée possède bien K modèles. La correction du certificat peut ensuite être vérifiée indépendamment, ce qui renforce la confiance que l'on peut accorder à la sortie de ces logiciels complexes. Nous établissons de nouveaux liens entre nos premiers travaux sur le sujet et le système de preuve MICE, et expliquons comment ces systèmes peuvent être vus comme une manière de certifier, de façon syntaxique, l'équivalence entre une formule CNF et un circuit DNNF.

Le quatrième chapitre étudie la généralisation de structures de données issues de la compilation des connaissances à des domaines non binaires. Nous montrons comment elles peuvent être utilisées pour représenter l'ensemble des réponses à des requêtes de bases de données et comment elles permettent de retrouver des résultats existants. Nous commençons par revisiter l'algorithme de Yannakakis en le reformulant comme un algorithme de compilation, puis montrons qu'une version modifiée de DPLL exhaustif donne des résultats théoriques similaires. Cette nouvelle perspective permet de compiler une plus grande classe de requêtes que l'algorithme de Yannakakis, en particulier, la classes des requêtes conjonctives avec négation.

Enfin, le chapitre 5 montre comment les idées de compilation des connaissances peuvent s'appliquer à l'optimisation convexe. Dans ce chapitre, nous nous intéressons à la notion de formulations étendues, dont le but est d'écrire un ensemble de contraintes linéaires décrivant

un polyèdre donné, éventuellement dans un espace de dimension plus grand. Nous définissons l'enveloppe convexe d'une fonction booléenne f comme le polyèdre défini comme l'enveloppe convexe de ses modèles, où un modèle est vu comme un point de $[0, 1]^n$. Nous montrons que, lorsque f est donnée sous une forme compilée, par exemple, comme un circuit DNNF C, son enveloppe convexe peut être décrite avec un nombre de contraintes linéaires proportionnel à la taille du circuit. Ce résultat fournit une nouvelle manière de dériver automatiquement des formulations étendues. Nous appliquons cette nouvelle connexion au problème d'optimisation polynomiale binaire et retrouvons de nombreux résultats de tractabilité existants.

Remerciements

This manuscript, and more generally, my career, would not have been the same without my colleagues, friends, family and many other people met along the way. I would like to take some space here to acknowledge their importance and influence on my work. This section being of a personal nature, it will mainly be written in French.

I would like to start by warmly thanking Gilles Audemard, Olaf Beyersdorff and Hubie Chen who reviewed this manuscript and managed to do so despite the short time frame and moving deadlines. I am similarly honored by the fact that Bruno Zanuttini and Luce Brotcorne accepted to be part of the jury. I am honored that you accepted my invitation and eager to debate the manuscript during the defense.

Merci à Pierre d'être le garant de cette habilitation. Ma soutenance n'aurait certainement pas pu avoir lieu cette année sans compter sur ton efficacité administrative hors-normes. Mais surtout, je suis heureux de pouvoir travailler avec toi depuis plusieurs années maintenant. Même après tout ce temps, je mesure chaque jour la grande gentillesse et l'humilité dont tu fais systématiquement preuve. Et je reste toujours impressionné par ta disponibilité sans faille lorsque tu as mille autres choses à faire. Merci encore pour ton aide.

Ma carrière scientifique ne serait pas ce qu'elle est sans de nombreuses rencontres faites parfois au hasard et qui ont débouché sur des collaborations scientifiques et, souvent, des amitiés. Il est souvent difficile de séparer les deux et la richesse de ces interactions est certainement ce qui fait de notre métier un plaisir renouvelé. On attribue souvent à Erdös une citation comme quoi un mathématicien serait une machine à transformer le café en théorèmes. Je crois plutôt qu'une chercheuse ou un chercheur est un être capable de transformer une machine à café en espace vivant où les idées s'échangent et les points de vue se transforment. Et cela fait parfois des théorèmes. Qu'ici soient donc remerciés toutes ces personnes croisées au hasard d'un café noir et d'un tableau blanc, où nous avons parlé algorithme, théorème, cuisine, famille ou culture, que cela ait donné ou non, un théorème. Merci particulièrement à Arnaud Durand qui m'a accompagné pendant toute ma thèse, qui fut le vrai commencement de tout cela. Merci à Stefan Mengel, avec qui je collabore depuis de nombreuses années et qui, sans le dire, veille depuis longtemps sur moi. Merci à Jean-Marie Lagniez qui m'a fait découvrir la compilation de connaissances et qui m'a fait venir à Lens une première fois en 2014, manger une faluche et un pain d'chien ! Je serai revenu quelques années plus tard ne serait-ce que pour les trois fromages qui puent. Merci à Yann Strozecki, que j'aime toujours retrouver pour énumérer des choses. Merci aux collègues de LINKS, Sylvain, Sophie, Iovka, Sławek, Charles, Aurélien, Mikaël, Nathalie et Joachim mon ancienne équipe. Vous m'avez bien accueilli lorsque

j'étais un jeune enseignant chercheur, qui, comme beaucoup après leur recrutement, cherchait essentiellement à comprendre comment on peut mener de front tous les aspects de notre métier sans délaisser nos passions. Vos conseils et encouragements, alors même qu'on discutait parfois de mon départ, m'ont été précieux. Merci aux collègues du CRIL pour avoir fait la même chose lorsque je suis arrivé à l'université d'Artois. Merci particulièrement à Daniel pour toute la bienveillance et le soin qu'il met à accueillir chaque nouveau du CRIL. Merci à Anne avec qui j'ai toujours plaisir à discuter. Merci aussi à Virginie, Sandrine et Frédéric. Je n'ai plus eu de problèmes administratifs depuis 2023, et vous en êtes la raison principale, et ce, alors même qu'Etamines et Notilus étaient déployés prématurément.

Je tiens ici à remercier les étudiants que j'ai encadrés en thèse, Nicolas Crosetti et Oliver Irwin. On le dit, on le sait, mais on n'en mesure la richesse qu'une fois qu'on le vit : on apprend certainement autant que les étudiants lorsqu'on les encadre, voire plus. Vos regards neufs sur des objets avec lesquels je suis désormais peut-être trop familier m'ont permis de comprendre beaucoup de choses. Merci en particulier à Nicolas pour la précision de tes analyses et de tes critiques. Merci à Oliver pour ta bienveillance et pour l'énergie que tu es capable d'investir, dans le papier comme dans le béton. Je suis aussi, je l'espère, devenu meilleur pédagogue à vos côtés. Merci aussi à Arthur Ledaguenel et Martín Muñoz. Je vous souhaite à tous une carrière aussi épanouissante que la mienne.

La recherche n'est cependant qu'une partie de mon métier. La partie émergée de ce manuscript. Dessous tout cela, il y a toute une vie, riche en émotions, en échange, en monologues et en dialogues. Dessous tout cela, il y a l'enseignement. Il y a l'UFR des Langues Étrangères Appliquées, il y a la Bande de Roubaix (RBX RPZ). C'est peut-être là que j'ai le plus appris sur moi-même et sur l'université en général. C'est avec elles et eux qu'il a fallu essuyer les nombreux plâtres, de la loi ORE à Parcoursup et des tableaux du PREEL. C'est là qu'il a fallu affronter Hyperplanning et l'axe du mal. Et c'est là, que dans l'épuisement réel, nous trouvions à rire et à partager, un Chélia sur une terrasse ensoleillée, un verre à l'OcKaz ou une chanson en manif. Alors merci à cette belle équipe, merci pour vos luttes et votre énergie. Merci de ne jamais oublier qu'être universitaire, c'est surtout se battre pour la dignité de nos étudiantes et étudiants et celle de toutes et tous nos collègues. Merci de n'être pas loin même si je suis parti. Merci à la Team L1. À Chachou, co-conductrice de tractopelle. À Béné, oracle infatigable qui sait lire la météo dans les entrailles de l'université. À Aurélie pour son efficacité sans faille. Merci à Oliver, Montse, Mag, Angèle, Gabby, Malik, FloB, Delphine (les RICI resteront mes préférés à jamais), Allyson, Cédric, Ouadie, Nico, Maria, Aude, Mehdi, Mallorie. Sans oublier l'incroyable secrétariat ! Merci à Audrey, Cora, Chrystèle, Sophie, Clothilde... En arrivant à l'Université d'Artois, j'ai aussi connu une belle équipe d'enseignants ! Merci particulièrement à Fadoua, Christophe et Gilles pour m'avoir invité dans cette belle aventure qu'est le BIA. À Maxence et Amélie, avec qui construire une scène en Lego n'a jamais été si drôle et inutile.

Merci à mes nombreux amis qui m'ont permis, par leurs petits et grands services, ou simplement par leur présence, d'arriver au bout de ce manuscript. Merci à Chachou, qui, en plus d'être une super voisine, m'a offert un temps précieux quelques mercredis après-midis. Merci à la bande du goûtapéro, où il suffit de taper quelques lignes pour créer un sas de décompression magique. À Camille. Que j'aurais tout aussi bien pu remercier parmi les collègues, mais cela ne ferait pas justice à son importance. À Antoine, Charles, Ambre et leur petits clones, à Lélé,

Zouzou, Nono, Élie et Lou. Au maître Pangolin, Guillaume, qui ne pourra malheureusement pas faire un live tweet savoureux de ma soutenance. Merci à Antoine, Emma, Jimmy, Audrey, Anne-Sophie, François, cette joyeuse bande de parents qui ont gardé Olivia plus d'une fois pour me permettre d'écrire ces lignes. À Sami, qui ne lira jamais ces lignes.

Enfin, je souhaiterai remercier ici ma famille, sans qui rien de tout cela n'aurait jamais été possible. À Jack et Magali, Audrey et Ju, Karine et Steph, pour avoir été de supers Papis-Mamis-Tata-Tonton quand il fallait avancer en période de vacances scolaires. À Yaya, bâtisseur infatigable, qui m'a donné le goût de la recherche depuis l'enfance mais qui est parti trop tôt pour être parmi nous aujourd'hui. À mes parents et à ma sœur, qui m'ont toujours soutenu et encouragé, qui m'ont appris la patience, la bienveillance et à organiser ma pensée. Et qui sont aussi devenus d'incroyables Papou, Mamou et Tatie depuis la naissance d'Olivia, m'offrant de nombreuses occasions de manquer à mon devoir familial pour partir en conférences ! Merci à ma compagne, Aurélia, pour son soutien quotidien tout au long de la rédaction de ce manuscript, pour m'avoir donné le temps dont j'avais besoin malgré ses propres contraintes, de m'avoir prêté un peu de son énergie qui semble inépuisable. Et enfin, merci à Olivia, pour toute la joie et l'amour que tu nous apportes chaque jour. Même si ce “livre n'est pas aussi important que le travail de maman”, j'espère qu'il servira à quelques personnes tout de même.

Foreword

Here we are. Finally writing the first lines of the weirdest document anyone pursuing a career in the French academic system has to undertake, the infamous *Habilitation à diriger des recherches*. An administrative formality for some, a pointless but necessary exercise for others, a scientific document with the noble purpose of enlightening the entire scientific community, a summary of one's work, an academic life work... Each one has his own personal understanding of the exercise and it took me some time to go through this diversity and find my own definition. So, here we are, in this foreword section which will serve, hopefully, as a clarification, for the reader of course, but mainly for myself.

Ten years ago, I defended my PhD where I tried to understand what really made the #SAT problem hard. My first contribution on this topic has been a paper at SAT14 [CDM14] where Arnaud Durand, Stefan Mengel and I defined a new notion of hypergraph acyclicity, namely, disjoint branches acyclicity, and gave a polynomial time algorithm for #SAT when the hypergraph of the input CNF formula is disjoint branches acyclic. After delivering my talk, Jean-Marie Lagniez approached me for discussing the result. I guess this encounter changed the course of my scientific career. Jean-Marie explained to me how his d4 solver works and how it was related to this weird domain known as Knowledge Compilation. I went home and started reading some random papers on DNNF and friends. It is a mouthful I know, and my dear fellows keep making fun of me when new acronyms keep popping up in the middle of my slides (hopefully, the notations from Chapter 4 will make all of this clearer). But that day, upon reading and understanding the definition of DNNF and its restrictions, something clicked. Every #SAT algorithm I had encountered so far was implicitly building such a data structure (an observation partially made in [HD05] already but that I was not aware of it). And it was a very natural and unifying way of explaining why tractable classes of formulas for #SAT where also tractable for other tasks such as enumeration, sampling or weighted model counting. So many hard looking algorithms were now only an encoding away.

This prompted me into looking into other domains of computer science I was familiar with and see where such a unifying framework could be applied, or already has been implicitly. "Decomposability" and "determinism" (whose better name would actually be "unambiguity" but it is too early to start arguing) are notions so natural that the concept of DNNF or close cousins have appeared in many other areas of computer science. The lens of knowledge compilation allows one to simplify, unify and reexplain existing results with a better interface between seemingly disconnected areas of computer science. The present document serves an overview of how I have applied these concepts in my research.

Contents

1 Data structures for Boolean functions 1
1.1 Definition and notations . . . 1
1.2 Binary Decision Diagrams . . . 5
1.3 Negation Normal Form Circuits . . . 10
1.4 The Knowledge Compilation Map . . . 16
1.4.1 Comparing Representations . . . 16
1.4.2 Tractable queries . . . 18
1.4.3 Tractable Transformations . . . 23
1.5 Conclusion . . . 27

2 Applications to Propositional Logic 29
2.1 Tree Decision Diagrams . . . 31
2.1.1 Main definitions . . . 31
2.1.2 Determinism . . . 34
2.1.3 Tractable transformations . . . 36
2.1.4 Minimization and canonicity . . . 41
2.1.5 Comparing d-TDD with other data structures . . . 45
2.2 Compiling Structured CNF formulas . . . 52
2.2.1 Bottom-up compilation of CNF formulas . . . 52
2.2.2 Structured CNF formulas . . . 53
2.2.3 Circuit treewidth . . . 58
2.3 Compiling Structured Quantified CNF formulas . . . 61
2.3.1 Determinizing nTDD . . . 62
2.3.2 Application to quantified Boolean functions . . . 64
2.4 Conclusion . . . 66

3 Applications to Proof Complexity 67
3.1 A naive proof system for #SAT . . . 70
3.2 From d-DNNF circuits to #SAT proof systems . . . 73
3.3 Certified decision-DNNF . . . 74
3.4 Syntactic caching and top-down solver . . . 77
3.4.1 Syntactic entailment . . . 79
3.4.2 MICE proof system . . . 82

3.4.3 Certified D4 . . . 89
3.5 Certified decision-DNNF circuits with learned clauses . . . 91
3.6 Propositional based certification . . . 94
3.7 Conclusion . . . 95

4 Applications to Databases 97
4.1 Preliminaries . . . 99
4.2 Relational Circuits . . . 106
4.2.1 Yet another class of circuits . . . 106
4.2.2 Main definitions . . . 107
4.2.3 Relational circuits, Factorized Databases and DNNFs . . . 109
4.3 Building Relational Circuits Bottom-up . . . 112
4.3.1 Yannakakis Algorithm . . . 112
4.3.2 Acyclic join queries . . . 113
4.3.3 Beyond acyclic queries . . . 116
4.4 Building Relational Circuits Top-down . . . 124
4.4.1 Manipulating Join Queries . . . 125
4.4.2 Exhaustive DPLL . . . 127
4.4.3 Complexity of Exhaustive DPLL . . . 129
4.5 Building Relational Circuits for Signed Join Queries . . . 134
4.5.1 Definitions and notations . . . 134
4.5.2 Exhaustive DPLL for signed join queries . . . 135
4.5.3 Exhaustive DPLL and Conjunctive Queries . . . 140
4.6 Binarization . . . 141
4.7 Using Relational Circuits . . . 143
4.7.1 Constant Delay Enumeration . . . 144
4.7.2 Counting . . . 145
4.7.3 Direct access . . . 146
4.8 Conclusion . . . 148

5 Applications to Optimization Theory 149
5.1 Definitions and notations . . . 151
5.2 Optimization on DNNF circuits . . . 155
5.2.1 Maximizing weighted Boolean Functions . . . 155
5.2.2 Convex hull of OBDD circuits . . . 157
5.2.3 Convex hull of DNNF circuits . . . 159
5.3 Binary Polynomial Optimization . . . 168
5.3.1 Problem definitions . . . 168
5.3.2 Solving BPO via Knowledge Compilation . . . 170
5.3.3 Beyond BPO . . . 174
5.4 Conclusion . . . 177

Extended CV 203

Introduction

Efficient manipulation of Boolean functions has been identified as a fundamental task very early in the history of computing, when it clearly appeared that all information could eventually be encoded as binary. From the seminal algebraic approach of Shannon [Sha38] motivated by the design of circuits, easy to manipulate graph-based representations of Boolean functions have been proposed. In 1959, Lee proposed a representation where functions are represented as successions of switches leading to a decision [Lee59]. Akers [Ake78] later introduced the notion of binary decision diagrams where a Boolean function is implemented as a succession of tests on the values of its input variables, leading to a decision to accept or reject the current state of the input. A natural syntactic restriction of this representation has since been studied by Bryant [Bry86; Bry92] who defined the *ordered binary decision diagrams* (OBDD) where variables have to be tested following a static order, see Fig. 1 for an example computing whether the number of variables set to 1 is odd. Bryant showed that this representation enjoys interesting properties: it can be exponentially more compact than the truth table or the list of models of the function while still allowing efficient manipulation of the underlying function. For example, one can compute the number of models of a Boolean function represented as an OBDD by doing a linear (in the size of the OBDD) number of arithmetic operations. He also observed that OBDDs allow for efficient manipulation. Given an OBDD, one can easily build an OBDD computing the negation of the original one, or one could combine two OBDDs (following the same order) into one accepting the conjunction or disjunction of the two. One could hence see OBDDs as an efficient way of implementing a data structure for Boolean functions offering an interface for computing the number of models, the conjunction of two functions, the enumeration of all models, and so on.

Efficiently manipulating Boolean functions becomes particularly useful when such a Boolean function encodes something one is interested in, such as knowledge on how a system works, a common point of view in the field of declarative artificial intelligence where the knowledge base and the program manipulating it are separated. A natural way of encoding such knowledge has been to use various rule-based [McD82] or logical frameworks, using propositional logic for example [KR87]. These developments led to the notion of *propositional knowledge base* where propositional logic is used to encode knowledge. For example, when building a software, the compilation process may be configured using options depending on what the user needs and can afford. We see each compilation option as a Boolean variable O_i such that enabling the option corresponds to setting O_i to true. However, some options may be incompatible with one another, or may have complicated dependencies. Hence, we would like to be able to express

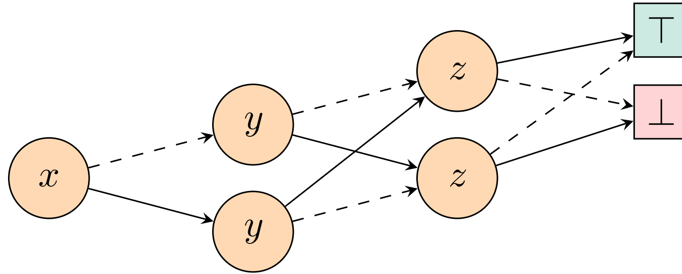


Figure 1: An OBDD accepting assignments having an odd number of variables set to 1.

which combinations of O_i are correct. This can be modeled as a Boolean function K mapping to true each such combination (a *model*, in logic) and to false otherwise. In this case, K is a Boolean function representing a knowledge base, in the sense that it encodes some knowledge on a process. For example, there may be an option O_{GPU} to enable GPU acceleration and an option O_{HD} to enable high-definition graphics. In this case, one wants to ensure that whenever O_{HD} is set to true, one also needs to enforce O_{GPU}. Hence K will evaluate to false as soon as O_{GPU} is set to false and O_{HD} is set to true. Consider this other example: the program may rely on a library for arbitrary precision computation but gives the user the choice between three libraries L_1, L_2, L_3. Each library corresponds to a compilation option $O_{L_1}, O_{L_2}, O_{L_3}$ and the user needs to pick exactly one such library. Hence K should evaluate to false as soon as no library or more than one library is chosen by the user. Finally, let's say that only library L_1 has the necessary features for implementing GPU acceleration, so if O_{GPU} is true, then only L_1 can be used.

Now, assume that such a knowledge base is provided to the user. Its main use will be to be *queried*. The most obvious case is to check whether a given choice of options is consistent with K. In this case, it boils down to evaluating K on the given assignment of options, which seems a relatively easy task. Now, many other types of queries could be imagined. Maybe the user is certain about a subset of options but does not really care about the value of the others and would happily let the computer decide for him. In this case, one needs to check whether K has a model for which a subset of its variables has been set. And in many cases, one may be interested in picking the "best model" compatible with the user's needs, assuming we have some way of comparing each model (e.g., by assigning a cost in $\mathbb{R}_+$ to each option and picking the one which minimizes the total cost of chosen options). Another useful direction for testing would be to sample consistent sets of options for compilation, or to enumerate a few distinctive examples.

The main hurdle with this approach is that the complexity of each task listed above depends on how the knowledge base K is given to the user. In general, K is implicitly described in a "natural form", corresponding to the way experts would describe the problem. More often than not, knowledge is described in propositional logic as a list – that is, from a logical point of view, a conjunction – of constraints the system has to respect. In the example above, we could say

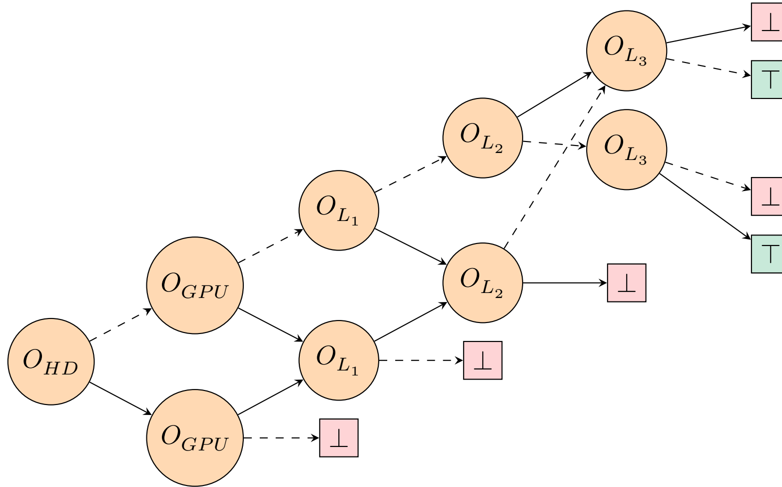


Figure 2: An OBDD representing a knowledge base on compilation options.

that K is defined by the following constraints: $O_{HD} \Rightarrow O_{GPU}$, $O_{L_1} \vee O_{L_2} \vee O_{L_3}$, $\neg O_{L_1} \vee \neg O_{L_2}$, $\neg O_{L_1} \vee \neg O_{L_3}$, $\neg O_{L_2} \vee \neg O_{L_3}$ and $O_{GPU} \Rightarrow O_{L_1}$. Unfortunately, this kind of representation is usually hard to work with. It is indeed well known since the seminal work by Cook and Levin [Lev73; Coo71] that deciding the satisfiability of a formula given in *Conjunctive Normal Form* (CNF) is NP-complete[1]. While this observation may seem depressing at first, it turns out that the underlying system of constraints is often well-behaved enough that dedicated solvers are capable of answering most queries asked by the user in a reasonable amount of time, despite these tasks being NP-hard to answer. But even in this case, calling the solver several times with different assumptions hides the fact that the calls are not independent from one another: they all share the knowledge encoded in K. To take this observation into account, a better approach has been used: given a knowledge base K, construct a data structure representing K which also allows one to efficiently solve the reasoning tasks needed for the problem at hand. In the previous example, one could build the diagram depicted in Fig. 2 which represents K. Now, each time the user needs to query the knowledge base, it can directly perform reasoning on the OBDD instead of calling a dedicated solver on the original representation of K. The complexity of each operation now depends on the size of the new representation, which ideally is small.

Efficient reasoning on propositional knowledge bases by using appropriate data structures has been the main focus of a research area in artificial intelligence known as *Knowledge Compilation*. This use of the term was originally coined by Kautz and Selman in [KS91] as a very general notion studying the influence of precomputation on the complexity of solving a given problem. More precisely, they study the complexity of a function f in the following

[1] Alternatively, the power of conjunction can be directly seen through the Tseitin transforms of Boolean circuits [Tse83] where the computation performed by a circuit is emulated as a conjunction of local constraints expressing the value each gate is supposed to take given the values of its input gates.

framework: given a partial input K to f, can we precompute some data structure so that computing $f(K, \tau)$ for any τ becomes tractable. For example, K could be a propositional knowledge base represented as a CNF, τ could be a partial assignment of the variables of K and f the function that accepts every partial assignment τ that can be extended to a model of K. It was quickly observed by Cadoli, Donini and Schaerf [CDS96] building on an idea of Kautz and Selman [KS92] that almost every interesting function f does not admit this kind of preprocessing (unless NP $\subseteq$ P/poly, which would be very surprising from the perspective of complexity theory). But in practice, the notion remains interesting because some knowledge bases are indeed tractable. The approach morphed into the art of transforming a "bad" representation of a Boolean function into a "better" representation that fits the user's needs. This point of view has been solidified by Darwiche and Marquis in their seminal paper [DM02] giving a systematic study of concrete data structures for representing Boolean functions, comparing their strengths and weaknesses in terms of succinctness and supported queries or transformations. Their approach offers a clear and usable framework, a "map", as they call it, to navigate a complicated space. This point of view encompasses three main research directions:

1. The design and implementation of *compilation algorithms*, whose aim is to transform a knowledge base into a tractable representation. For example, transforming a given Boolean function described as a CNF formula into an OBDD.

2. The design and implementation of algorithms for manipulating these representations. It encompasses two main directions. The first one consists in solving a query on the Boolean function, for example, computing its number of satisfying assignments. The second consists in transforming the data structure to represent a new Boolean function, for example, computing a representation of $\neg f$ from a representation of f.

3. Comparing different representations in terms of succinctness and tractability, for example, establishing that there exist Boolean functions succinctly represented as CNF but which admit only OBDDs of exponential size.

All three directions have generated important contributions in several fields of computer science. The first one has, for example, helped in the design of efficient tools for solving #SAT in practice [LM17; Dar11], a notoriously hard problem. The second one has been implemented to reason in real time on large knowledge bases representing product configurations [ACF10; Par03]. Finally, the last one has uncovered connections with many aspects of computational complexity, e.g., communication complexity [Bov+16; Vin24], proof complexity [Cap19; BHK25; BHS24] or extension complexity [CPG23].

This manuscript summarizes and organizes the different contributions I made in the field of knowledge compilation, more specifically, in applying techniques and ideas from knowledge compilation to other fields of computer science: propositional logic, proof complexity, database theory and convex optimization theory. Every chapter (but the first one containing preliminaries) is dedicated to a different field and all chapters are independent from one another. They are introduced with a general introduction, presenting the field, the main ideas and the chapter's organization. At the end of each introduction, I summarize my main contributions

covered in the chapter. The rest of this introduction gives a general idea of the content of each chapter.

Knowledge compilation and Propositional Logic: tractability of structured CNF formulas. As hinted before, the field of knowledge compilation has mostly focused on compiling Boolean functions, often represented as CNF formulas, hence using propositional logic. It is only natural that knowledge compilation has several applications in logic. For example, the #SAT problem of computing the number of models of a given CNF formula can be solved using tools from knowledge compilation: first, compile the formula F into a data structure C for which we can efficiently compute the number of models and run the counting algorithm directly on C. The total time is then the time to produce C plus the time needed to compute the number of models of C, which is polynomial in the size of C. Since #SAT is a notoriously hard problem to solve, we cannot have guarantees that the compilation phase runs in polynomial time in the size of F but, depending on the structure of F, we can sometimes obtain such guarantees. One extreme example is the following: consider a tree $T = (V, E)$ and the following formula on variables V:

$$F = \bigwedge_{\{x,y\} \in E} (x \vee y).$$

In this case, we can show that, by following the structure of the tree, we can represent F with an OBDD of size linear in E, and in particular, find $\#F$. This is obviously not the only approach to find $\#F$. It is not hard to see that $\#F$ is the number of vertex covers of T which we can find using a dynamic programming algorithm. However, the compilation approach here offers modularity. If one now wants to sample uniformly a model of F, we can either design another dynamic programming algorithm for this specific task or simply invoke the fact that uniformly sampling models of OBDDs is tractable. This modularity is both interesting in theory, to avoid duplicating very similar proofs and in practice, since having efficient implementations to solve several tasks for OBDDs can then be reused for many problems as long as we are able to convert them into OBDDs.

While the previous example is very limited, it can be generalized via the notion of treewidth, which intuitively measures how tree-like a graph is. In this case, OBDD may be too weak to offer interesting complexity guarantees, but it can be recovered by using simple generalizations of OBDD presented in Chapter 1. This approach is the starting point of a fruitful line of research which tries to understand what kind of structures in CNF formulas lead to interesting tractability results [Che04; FMR08; SS10; PSS13; SS13; OPS13; GS17; Bov+15; Ama+20]. In Chapter 2, we focus on this line of research, showing that CNF formulas of bounded treewidth can efficiently be compiled into a tractable representation language, recovering many known tractability results. While the results are not new in themselves, the way we prove them is. Indeed, we introduce a new data structure for representing Boolean functions, that we call Tree Decision Diagrams (TDD), and which generalizes OBDD. They are powerful enough to recover the interesting tractability results of bounded treewidth CNF formulas while retaining most properties of OBDD. In particular, we show that each Boolean function has a canonical

and minimal representation in this formalism and that any representation can be minimized into its canonical form.

Knowledge compilation and Proof Complexity: proof systems for #SAT. When a SAT solver answers that a given CNF formula is satisfiable and outputs a satisfying assignment, it is easy to check that it is indeed the case. Even if the solver has a bug, it may not have been encountered during this specific run and it does not change the correctness of its output. This fact is not symmetrical however: when a SAT solver answers that the input CNF formula is unsatisfiable, it is not straightforward to check and the user is left to trust a possibly buggy software. To circumvent this problem, the community has focused on outputting a proof of unsatisfiability which can efficiently be checked independently from the solver. In a nutshell, the proof of unsatisfiability can often be seen as a list of logical steps one can follow to derive a contradiction from the input formula. This approach quickly became standard for SAT solvers and inspired other tools solving hard to check logical problems to follow a similar path, for example, in the design of MaxSAT solvers [BLM07; Bon+18], where one tries to maximize the number of satisfied clauses in unsatisfiable formulas or QBF solvers [Jus+07; JGM13; BBM21], where one tries to decide the satisfiability of quantified CNF formulas. A notorious exception, however, is the situation with #SAT solvers, outputting the number of satisfying assignments of a given CNF formula. When I started investigating this question in 2019 [Cap19], no #SAT solver was able to output a proof that the output number of satisfying assignments was correct. This can be partially explained by the fact that, in the case of QBF and MaxSAT, proof systems based on generalizations of resolution, a well-understood proof system for SAT, have been described early. No such proof system is known for #SAT and it may explain why no certified tool was created earlier.

To design proof systems for the #SAT problem, we rely on the following observation from [Cap19]: if one wants to certify that a CNF formula F has k satisfying assignments, one can simply exhibit a representation C of the satisfying assignments of F where there is an efficient algorithm to find the number of models of C. C could, for example, be an OBDD as in Fig. 2. The main problem with this approach is that there is no reason to trust the fact that C faithfully represents the models of F. The main idea from [Cap19] has been to introduce a labeling of C which allows to check that it is indeed equivalent to F. Other proof systems based on the same idea (sometimes implicitly at first) have later been proposed, such as MICE [FHR22; BHS24] or CPOG [Bry+25]. We also implemented a similar approach [CLM21] to certify in practice the output of the #SAT solver D4 [LM17]. In Chapter 3, we review this contribution and explain the connections between the many proof systems introduced so far for #SAT.

Knowledge compilation and database theory. The most important task in a database management system, from the user's point of view, is arguably its ability to efficiently list the answers to a given query. One obvious limitation is the fact that the number of answers may be large, leading to important inefficiencies in just printing them out. One unsatisfactory workaround is to limit the number of answers printed. While it works well for presenting the answers to the user, it is hard to guarantee that the sample chosen is representative of the

answer set, nor whether it fits what the user needed. A more useful approach has been to consider enumeration algorithms which build a data structure from which we can extract the answers one after the other in a streaming fashion, while guaranteeing that the time needed to output the next answer is reasonable [BDG07]. This point of view works well when the goal is to perform an action on every tuple from the answer set but may not be satisfactory when the user needs to understand the answer set. Indeed, when this set is too large, it is more interesting to get insights by sampling it, by computing some statistics such as the number of answers sharing values on some attributes or by enumerating targeted parts of the answer set. In this case, the approach is akin to the one used in knowledge compilation: we want to precompute a data structure succinctly representing the answer set, which we can then use to better understand it. Not surprisingly, this approach has emerged independently from knowledge compilation in the database community under the name *factorized databases* [OZ15]. The main difference with the setting of knowledge compilation is that we go from a binary to a larger domain but the data structures introduced in [OZ15] are all straightforward generalizations of data structures already present in the knowledge compilation map [DM02], a fact already observed by Olteanu in [Olt16].

Interestingly, the focus in database theory has specific features that do not naturally arise from the point of view of knowledge compilation. One illustrative example of this is that new tasks are needed which were barely considered until now for knowledge compilation. For example, aggregating tuples with weights in a semiring [KNR16; JPR16] is a task in database that has many applications in database theory as it allows to compute explanations on why tuples are in the answer sets, via the notion of provenance. While the task has been considered in knowledge compilation [KVD17], it has not received the same degree of attention. Another task is to answer direct access queries [Bag+08; Car+20]. In this setting, we assume that there is an order on the answer set and we want to be able to efficiently find the kth answer given k on the input. The complexity of this problem depends on how the answer set is represented and how its order is defined and it has led to a fruitful line of research, classifying the tractable and intractable cases [BCM22; Car+23] that we have revisited and generalized using tools from knowledge compilation [CI24]. Another example of the specific features of the database point of view comes from how the complexity is measured. In the knowledge compilation approach, we mostly focus on the size of the representation as a whole while in the database setting, there are quantities of different orders of magnitude. Indeed, the query is considered to be much smaller than the database itself, and it changes how one understands the notion of tractability. If s is the size of the query and d the size of the database, then a representation of size $2^s \cdot d$ is more useful than a representation of size $s \cdot d^2$. It may seem a triviality now but it actually forces us to study the algorithms in more details and realize that even if two representations of the same answer set support the same query in polynomial time, their dependency on s and d may be different, leading to one being “more tractable” than the other.

In Chapter 4, we present the notion of relational circuits and explain how they can be seen as generalizations of the data structures introduced in Chapter 1. The rest of the chapter is mostly dedicated to compilation algorithms, which transform an input query and database into a relational circuit. We first revisit a celebrated result by Yannakakis [Yan81] as a compilation

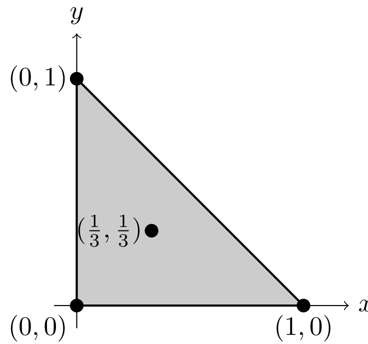


Figure 3: The convex hull of $\neg x \lor \neg y$ with interior point $(1/3, 1/3) = (1/3) \cdot ((0,0) + (0,1) + (1,0))$.

algorithm. We then show that a minor adaptation of exhaustive DPLL [San+04; HD05] to this setting offers comparable complexity guarantees while allowing to compile a larger family of queries.

Knowledge compilation and Convex Optimization. When a Boolean function is succinctly represented using tools from knowledge compilation, it is known that one can solve various optimization problems such as finding the assignment that has maximal value, where the value of an assignment is defined as some linear function over literals. This can be seen as some form of weighted model counting using an appropriate semiring [KVD17]. When the objective function is not linear, however, many optimization problems become hard [Kor+16]. This is partially explained algorithmically: as long as the objective function is linear, it can be decomposed into independent parts, each of them concerning one variable at a time. It allows the design of dynamic programming algorithms, progressing toward the objective value inductively. When the value of the objective function depends on the interaction of more than one variable together, in a non-trivial way, it is not clear how the data structure can be exploited to speed the computation. In Chapter 5, we present a new point of view explaining the tractability of optimizing linear functions over circuits from knowledge compilation using a geometric explanation, which we first presented in [CPG23]. To do so, we consider the models of a Boolean function with n variables as points in the space $\mathbb{R}^n$. In this space, we can define the convex hull of f as the set of points that can be obtained as a convex combination of models of f, that is, a combination of the form $\sum_{i=1}^{N}(\lambda_i \cdot p_i)$ where p_i is a point in $\{0,1\}^n$ corresponding to a model of f and $\lambda_i \in \mathbb{R}_+$ are such that $\sum_{i=1}^{N} \lambda_i = 1$. For example, Fig. 3 shows the convex hull for the Boolean function over variables x, y defined by the propositional formula $\neg x \lor \neg y$ and shows that the point $(1/3, 1/3)$ belongs to this convex hull.

This convex hull is a polyhedron $\mathcal{P} \subseteq \mathbb{R}^n$ and we know that any linear function will reach its optimal value at an extreme point of $\mathcal{P}$, and we can see that the extreme points of $\mathcal{P}$ coincide with the models of f. Now, it says that to optimize a linear function over the models of f, we can optimize over its convex hull and use tools from convex optimization theory. That

said, we do not get anything from this observation alone since the size of the description we have of the convex hull is at least the number of models of f, which can be large. Our key observation is that if f is represented with a small data structure of size s, we can extract from it a description of its convex hull with $O(s)$ linear constraints.

In Chapter 5, we review in detail this connection between knowledge compilation and compact description of the convex hull of Boolean functions. We then apply this connection to the problem of Binary Polynomial Optimization (BPO) where one tries to maximize a multilinear polynomial over $\{0, 1\}^n$ [DK17]. We show that by describing the optimal value of this problem as a Boolean function and using results from knowledge compilation, we are able to recover and generalize many tractability results for BPO.

Chapter 1

Data structures for Boolean functions

In this chapter, we study various data structures for representing Boolean functions and their properties. It serves as a basis for the other chapters, where these data structures are used as a means to derive, explain, and generalize tractability results in other areas of computer science. After giving general definitions and notation used throughout this manuscript, we focus on the main data structures used in knowledge compilation. We review a few algorithms for manipulating these data structures without going into too much detail. We also provide more detailed proofs for a few folklore results that are hard to find explicitly stated in the literature, while keeping the presentation short. Definitions and notions that are only necessary for a given chapter are given in that chapter, to make each of them as self-contained as possible.

Organization of this chapter. We start by giving general-purpose definitions and notation in Section 1.1. We then present data structures based on decision diagrams in Section 1.2 as a warm-up before generalizing to data structures based on syntactic restrictions of Boolean circuits in Section 1.3. We conclude this chapter with Section 1.4, where we study the succinctness and tractability of these data structures.

Personal contributions covered in this chapter. As this chapter is an introduction to knowledge compilation, it is intended as an opinionated survey and not as a presentation of my contributions. That said, some results in this chapter follow from earlier work. In particular, Section 1.4.1 compares the succinctness of different data structures presented earlier. Some separations here follow from [Bov+16], where we established a connection with communication complexity to prove new lower bounds.

1.1 Definition and notations

In this section, we give general definitions and notation that will be used throughout this document.

Assignments. Given two sets A and B, we denote by B^A the set of functions from A to B. When $B = \{0, 1\}$, we will often write 2^A to denote the set of assignments from a set A to $\{0, 1\}$. An element $\tau \in 2^A$ is called a *Boolean assignment on variables* A, and we will often just write "assignment" when it is clear from context that it is Boolean. A *partial (Boolean) assignment on variables* X is an element of 2^Y for some $Y \subseteq X$. Given two assignments $\tau_1 \in 2^X$ and $\tau_2 \in 2^Y$, we say that τ_1 and τ_2 are *compatible*, and write $\tau_1 \simeq \tau_2$ if for every $z \in X \cap Y$, $\tau_1(z) = \tau_2(z)$. In this case, we denote by $\tau_1 \bowtie \tau_2$ the assignment in $2^{X \cup Y}$ defined as $(\tau_1 \bowtie \tau_2)(z) = \tau_1(z)$ if $z \in X$ and $(\tau_1 \bowtie \tau_2)(z) = \tau_2(z)$ if $z \in Y$. When $X \cap Y = \emptyset$, then we necessarily have $\tau_1 \simeq \tau_2$ by definition. In this case, to emphasize this property, we will also use the notation $\tau_1 \times \tau_2$ in place of $\tau_1 \bowtie \tau_2$. We denote by $\langle x/0 \rangle$ (resp. $\langle x/1 \rangle$) the assignment in $2^{\{x\}}$ mapping x to 0 (resp. to 1). We will also use the notation $\langle x_1/b_1, \dots, x_k/b_k \rangle$ to denote the assignment in $2^{\{x_1,\dots,x_k\}}$ mapping x_i to b_i. Given $\tau \in 2^X$ and $Y \subseteq X$, we denote by $\tau|_Y$ the assignment in 2^Y such that $\tau|_Y(y) = \tau(y)$ for every $y \in Y$.

Boolean functions. A *Boolean function f on variables X* is a subset of 2^X. An assignment $\tau \in f$ is said to *satisfy* f and is alternatively called a *satisfying assignment* or a *model*. Given a Boolean function $f \subseteq 2^X$ on variables X and $Y \subseteq X$, we denote by $f|_Y \subseteq 2^Y$ the Boolean function on variables Y defined as $\{\tau|_Y \mid \tau \in f\}$.

Given a Boolean function $f \subseteq 2^X$, the *negation of* f, denoted by $\neg f$, is the Boolean function on variables X defined as $2^X \setminus f$. Given two functions $f \subseteq 2^X$ and $g \subseteq 2^Y$, we denote by $f \wedge g = \{\tau_1 \bowtie \tau_2 \mid \tau_1 \in f, \tau_2 \in g, \tau_1 \simeq \tau_2\} \subseteq 2^{X \cup Y}$. In other words, $\tau \in f \wedge g$ if and only if $\tau|_X \in f$ and $\tau|_Y \in g$. We say that $f \wedge g$ is *the conjunction of f and g*. When $X \cap Y = \emptyset$, we can observe that $f \wedge g$ is isomorphic to the Cartesian product of f and g and we will denote it by $f \times g$, to emphasize this syntactic property. This is also known as a *decomposable conjunction* (see Section 1.3 for more details).

Let $f \subseteq 2^X$ and $g \subseteq 2^Y$. We denote by $f \models g$ if and only if $f \wedge \neg g$ has no model. Or, equivalently, if f and g are defined over the same variables, then every model of f is also a model of g. We also denote by $f \Rightarrow g$ the Boolean function over variables $X \cup Y$ defined as $\neg f \vee g$. Observe that $f \models g$ if and only if $f \Rightarrow g$ is a tautology.

We denote by $f \vee g = \{\tau \in 2^{X \cup Y} \mid \tau|_X \in f \text{ or } \tau|_Y \in g\}$ and call $f \vee g$ the *disjunction of f and g*. When $X = Y$, we can observe that $f \vee g$ is equivalent to $f \cup g$, when seeing f and g as sets of assignments. We will hence sometimes use the $\cup$-symbol to emphasize the fact that f and g are defined on the same set of variables. This is also known as a *smooth disjunction* (see Section 1.3).

Conjunctive and Disjunctive Normal Form Formulas. While there are many ways of specifying Boolean functions, one central concept used throughout this document is that of Conjunctive Normal Form (CNF) formulas. Given a set X of variables, a literal over X is either $x \in X$ or $\neg x$. We let $\mathrm{lit}(X)$ be the set of literals over X, and for $\ell \in \mathrm{lit}(X)$, we denote by $\mathsf{var}(\ell)$ its underlying variable (that is, $x = \mathsf{var}(x) = \mathsf{var}(\neg x)$). For an assignment $\tau \in 2^X$, we naturally extend it to literals by defining $\tau(\neg x) = 1 - \tau(x)$.

A *clause* c is a set of literals. We will often write it as $c = \ell_1 \vee \dots \vee \ell_k$ because a clause is to be interpreted as a disjunction, and we let $\mathsf{var}(c) = \bigcup_{\ell \in c} \mathsf{var}(\ell)$. An assignment τ *satisfies*

a clause c if there exists $\ell \in c$ such that τ is defined on $\mathsf{var}(\ell)$ and $\tau(\ell) = 1$.

A *Conjunctive Normal Form formula* (CNF formula for short) F is a set of clauses. We let $\mathsf{var}(F) = \bigcup_{c \in F} \mathsf{var}(c)$. We will often denote it by $c_1 \wedge \cdots \wedge c_m$ because it is to be interpreted as a conjunction of clauses. An assignment τ *satisfies* F (or is *a model of* F) if for every clause $c \in F$, τ satisfies c. The Boolean function defined by a CNF formula is the Boolean function over $\mathsf{var}(F)$ whose satisfying assignments are the assignments over $\mathsf{var}(F)$ that satisfy F. We will often identify a CNF formula or a clause with the Boolean function it represents. Hence, we will use the notation we defined for Boolean functions directly on formulas. For example, we write $F \models c$ whenever every satisfying assignment of F is also a satisfying assignment for c.

The *size* $\|F\|$ of F is defined as $\sum_{c \in F} |\mathsf{var}(c)|$. We sometimes use the notation $|F|$ to denote the number of clauses in F, but we usually try to be explicit about this number instead of relying on the notation.

A CNF formula can be conditioned by a partial assignment as follows: for $\tau \in 2^Y$, we let $F[\tau]$ be the CNF formula obtained as follows. We remove from F every clause c containing a literal ℓ such that $\tau(\ell) = 1$. In the remaining clauses, we remove every literal ℓ such that $\tau(\ell) = 0$. An assignment $\sigma \in 2^{\mathsf{var}(F) \setminus Y}$ satisfies $F[\tau]$ if and only if $\sigma \times \tau$ satisfies F.

A *unit clause* is a clause having exactly one literal. If F contains a unit clause $c = \{\ell\}$, then every satisfying assignment of F must set ℓ to true because it must satisfy c. Simplifying a CNF formula by iteratively conditioning it on its unit clauses until no unit clause remains is called *unit propagation*.

Example 1.1. Let $F = (x \vee y \vee z) \wedge (\neg x \vee y) \wedge (\neg z \vee x)$. The size of F is $|F| = 3+2+2 = 7$. Its satisfying assignments (or models) are given by:

x	y	z
0	1	0
1	1	0
1	1	1

We have $F[x/1] = y$ and $F[x/0] = (y \vee z) \wedge (\neg z)$. In $F[x/0]$, the clause $(\neg z)$ is a unit clause; hence every model of $F[x/0]$ must set z to 0. We can then perform a unit propagation step and obtain $F[x/0, z/0]$, resulting in the formula y. Performing unit propagation again leads to the empty CNF; hence we find that F has only one model with $x = 0$, namely $\langle x/0, y/1, z/0 \rangle$.

A *Disjunctive Normal Form formula* (DNF formula for short) is the dual of a CNF formula: it is a disjunction of terms, where terms are sets of literals. An assignment τ satisfies a term T if and only if for every literal $\ell \in T$, we have $\tau(\ell) = 1$. An assignment τ satisfies a DNF formula F if and only if there exists a term $T \in F$ such that τ satisfies T. The size $\|F\|$ of a DNF formula is defined as $\sum_{T \in F} |\mathsf{var}(T)|$. Using De Morgan's laws, it is easy to see that the negation of a CNF formula F can be written as a DNF formula of size $\|F\|$.

Graphs. We will use graphs for many different reasons in this text: to describe circuits, to analyze the structure of formulas and queries, etc. We assume the reader is familiar with the basics of graph terminology, and refer to [Die12] for more details. That said, we give here a few useful definitions that we will use throughout the text. A *directed graph* $G = (V, E)$ is defined as a set of *vertices* V and a set $E \subseteq V \times V$ of *edges*. If $e = (u, v) \in E$, we say that e is an edge from u to v, also denoted $u \to v$. A *self-loop* is an edge of the form (u, u) for some $u \in V$. If the graph does not contain self-loops and has the property that for every $(u, v) \in E$, we have $(v, u) \in E$, we say that G is *undirected*. If not specified, we assume graphs to be undirected by default.

Two vertices u, v are called *neighbors* if and only if $(u, v) \in E$ or $(v, u) \in E$. The *neighborhood of* u is the set of neighbors of u together with u. The *open neighborhood of* u is the set of neighbors of u.

We will also use the notion of a *multigraph*. In this setting, the edge set E of a graph $G = (V, E)$ is a multiset; that is, the same edge may occur more than once. This will be particularly useful when describing decision diagrams, where we can have duplicated edges (albeit with distinct labels).

A *path* in a graph $G = (V, E)$ is a sequence $v_1, \ldots, v_k$ of distinct vertices such that $(v_i, v_{i+1}) \in E$ for every $1 \leq i < k$. A *cycle* is a path such that $(v_k, v_1) \in E$.

Given a graph $G = (V, E)$, a *connected component* of G is a subset W such that:

- For every $u, v \in W$, there is a path from u to v.
- If there is a path from $u \in W$ to some $v \in V$, then $v \in W$.

It is straightforward to find all connected components of G in linear time by performing a depth-first search, a fact we will use later. A graph with exactly one connected component is said to be *connected*.

A graph is said to be *acyclic* if it does not contain any cycle. We write *DAG* in place of directed acyclic graph. A connected undirected acyclic graph is called a *tree*. In this case, we can *root* the tree at an arbitrary vertex v and orient every edge toward the root. Observe that every vertex u except the root has exactly one outgoing edge (u, f). In this case, we call f *the parent of* u, and u is called *a child of* f.

Hypergraphs are generalizations of undirected graphs where edges may have an arbitrary number of vertices. More formally, a *hypergraph* $H = (V, E)$ is a pair of sets: V is the set of vertices of H and $E \subseteq 2^V$ is the set of edges of H. An edge $e \in E$ is thus a subset of V, i.e., $e \subseteq V$. We extend most notions from graphs to hypergraphs. In particular, a *path in* H is a sequence of distinct vertices $v_1, \ldots, v_k$ such that for every $1 \leq i < k$, there exists $e \in E$ such that $\{v_i, v_{i+1}\} \subseteq e$. Contrary to graphs, there may be more than one edge connecting v_i and v_{i+1} together. Sometimes, we will therefore describe a path with the explicit sequence of edges it uses, i.e., as a sequence $v_1, e_1, v_2, \ldots, e_{k-1}, v_k$ where $v_1, \ldots, v_k$ are distinct vertices and $e_1, \ldots, e_{k-1}$ are edges of H such that $\{v_i, v_{i+1}\} \subseteq e_i$ for every $1 \leq i < k$.

1.2 Binary Decision Diagrams

Binary Decision Diagrams form a family of graph-based data structures for representing Boolean functions that have a rich history, both in theory and in practice, that dates back to the 1980s [Bry92; Bry86]. We only give a brief introduction. We encourage the interested reader to consult the book by Wegener [Weg00] for a thorough treatment of the topic.

A *Binary Decision Diagram (BDD) C on variables X* is a directed acyclic multigraph such that:

- C contains exactly one vertex with indegree 0 called the *source*,
- Every vertex with outdegree 0 is called *a sink* and is labeled by either $\top$ or $\bot$,
- Each remaining vertex is referred to as a *decision-gate*, is labeled by some $x \in X$, and has at most two outgoing edges: at most one labeled 0 and at most one labeled 1. (Because C is a multigraph, these two edges may share the same endpoints, but they must have different labels.)

We will call the vertices of decision diagrams (and, later, of circuits) *gates*. When drawing decision diagrams such as the one in Fig. 1, we use solid edges to denote 1-labeled edges and dashed edges to denote 0-labeled edges.

We let $\mathsf{edges}(C)$ be the multiset of edges of the underlying DAG of C and $\mathsf{gates}(C)$ be the set of gates of C. We define the *size of* C, denoted by $|C|$, as the number of edges it contains.

Given a gate g of C, we let $\mathsf{var}(g) \subseteq X$ be the set of variables x such that there exists a path from g to a decision-gate g' labeled by x. Alternatively, we can define $\mathsf{var}(g) = \bigcup_{g' : (g,g') \in \mathsf{edges}(C)} \mathsf{var}(g')$.

Given a path $P = g_1, \ldots, g_k$ in C and an assignment $\tau \in 2^{\mathsf{var}(g_1)}$, we say that P *is compatible with* τ if and only if for every $1 \leq i < k$, if g_i is labeled by the variable x, then there is an edge between g_i and g_{i+1} that is labeled by $\tau(x)$. We say that an assignment $\tau \in 2^{\mathsf{var}(g)}$ *satisfies* g or *is a model of* g if and only if there exists a path from g to a $\top$-sink that is compatible with τ. Observe that in this case, the path from g to a $\top$-sink is unique. Indeed, for every i, there is at most one neighbor g' of g_i such that there is an edge (g_i, g') labeled by $\tau(x)$, that is, g_{i+1} is uniquely defined by g_i and $\tau(x)$. This is why we will often say *the* path compatible with τ. A BDD C on variables X and with source s *computes* the Boolean function f_C on variables X defined as $\{\tau \in 2^X \mid \tau \text{ satisfies } s\}$.

Now, BDDs themselves are not our main focus and we usually consider subclasses of them. A BDD is *read-once* if for every g and g' such that there is a path from g to g', the labels of g and g' are distinct. In particular, no variable is tested twice on any source-to-sink path. The read-once property is interesting because it allows us to check the existence of a model very easily. Indeed, if C is read-once then it has a model if and only if there exists a path from the source of C to a $\top$-gate, since we can reconstruct the model from this path by setting $\tau(x) = b$ if the path contains a decision-gate labeled by x and goes through an edge labeled by b. Since it is read-once, there is at most one such gate, leading to a correct definition of τ. Read-once BDDs also appear under the name *Free Binary Decision Diagram* (FBDD) in the literature.

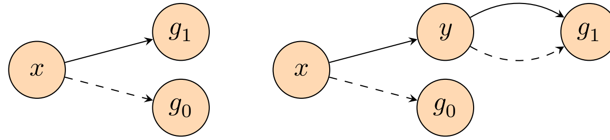


Figure 1.1: Smoothing BDDs by adding dummy gates: here $y \in \mathsf{var}(g_0) \setminus \mathsf{var}(g_1)$, so we add it with a dummy gate.

Observe that FBDDs are *deterministic* by nature, in the sense that given an FBDD C over X and an assignment $\tau \in 2^X$, there is at most one path that is compatible with τ in C. Indeed, at each decision-gate on variable x, there is at most one edge labeled by $\tau(x)$, hence, a path compatible with τ must contain this edge.

A decision diagram is *ordered* if there exists an order $x_1, \dots, x_n$ on X such that for every decision-gate g on variable x_i, if there is a path from g to a decision-gate $g' \neq g$ in C, then g' is labeled by a variable x_j with $j > i$. Observe that by definition, an ordered BDD (OBDD) is also read-once. While OBDDs are exponentially weaker than FBDDs [Weg00, Example 5.9.4], their stronger structure allows more tractable transformations, see Section 1.4.3 for more details.

A BDD may not test every variable on every path because sometimes, the value of a variable may be arbitrarily chosen. While this is allowed in the definition, it is sometimes handy to have BDDs that are *smooth* (also known as *complete*): a BDD C is smooth if there exists a set X of variables such that on every path from the source to a sink, every variable from X is tested. Equivalently, it means that for every decision-gate g and g', g'' such that (g, g') and (g, g'') are edges of C, we have $\mathsf{var}(g') = \mathsf{var}(g'')$. Observe that if C is read-once, then it is equivalent to say that all paths from the source to a sink have the same length.

A BDD C on variables X can always be made smooth and the transformation preserves the fact that it is read-once and/or ordered. To do so, we can simply add dummy gates that test the missing variables after each decision-gate, as illustrated in Fig. 1.1. More precisely, say g is a decision-gate on variable x with one 0-labeled outgoing edge (g, g_0) and one 1-labeled outgoing edge (g, g_1). Assume $\mathsf{var}(g_0) \neq \mathsf{var}(g_1)$ and, without loss of generality, let $y \in \mathsf{var}(g_0) \setminus \mathsf{var}(g_1)$. We can add a decision-gate g_1' on y with two outgoing edges pointing to g_1 and replace the edge (g, g_1) by (g, g_1'). It does not change the function computed by C and now, $\mathsf{var}(g_0) \setminus \mathsf{var}(g_1')$ has diminished in size. We can hence apply this transformation iteratively until $\mathsf{var}(g_1) = \mathsf{var}(g_0)$ for every gate g. The resulting BDD will have size $O(|X| \cdot |C|)$ and be smooth.

Lemma 1.2. *Given a BDD C (resp. read-once, ordered) on variables X, we can construct a smooth (resp. read-once, ordered) BDD C' computing f_C of size at most $O(|X| \cdot |C|)$ in time $O(|X| \cdot |C|)$.*

We illustrate the different properties in Fig. 1.2. Observe that in many cases, we do not really need the $\bot$-sink. Indeed, if a decision-gate g on x has only one outgoing edge labeled

by $b \in \{0,1\}$, it prevents any path going through g from setting $x = 1-b$. We observe that all these properties are syntactic and can be tested in polynomial time.

Lemma 1.3. *Given a BDD C of size s and having n variables, we can test in time $O(ns)$ whether C is read-once and whether it is smooth. We can test in time $O(n+s)$ whether it is ordered.*

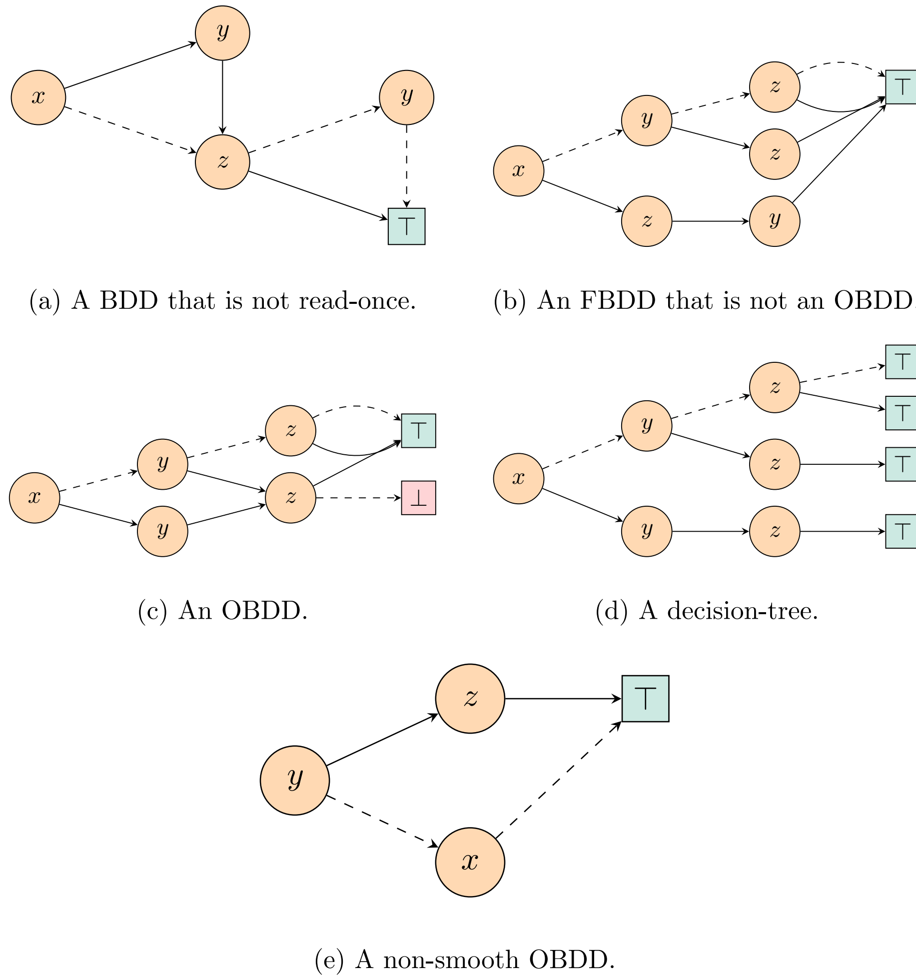
Proof. We start by arbitrarily numbering the variables of C as $x_1, \dots, x_n$ and precompute $\mathsf{var}(g)$ for each gate g of C. To do so, we store at each gate g a table T_g with n entries such that $T_g[i] = 1$ if and only if $x_i \in \mathsf{var}(g)$, and 0 otherwise. This can be done in time $O(ns)$ by recalling that $\mathsf{var}(g) = \bigcup_{g':(g,g') \in \mathsf{edges}(C)} \mathsf{var}(g')$. Hence, we can set $T_g[j] \leftarrow 1$ where x_j is the variable tested at g and $T_g[i] \leftarrow \max_{g':(g,g') \in \mathsf{edges}(C)}(T_{g'}[i])$ for every $i \neq j$. It takes time $O(n)$ per edge of the form (g,g'). Hence the total time is $O(ns)$.

Now, to test whether C is read-once, we only have to check that for every decision-gate g on variable x_i, and g' such that (g,g') is an edge of C, we have $T_{g'}[i] = 0$. This can be done in $O(s)$ by simply going over every edge of C.

To test whether the BDD is smooth, we need to check that for every g and g', g'' such that (g,g') and (g,g'') are edges in C, we have $\mathsf{var}(g') = \mathsf{var}(g'')$. This check can be done in time $O(n)$ after the previous precomputation and we only have to do it at most once per edge of the BDD, so a total time of $O(ns)$.

To test whether the BDD is ordered, we build a directed graph G_C whose vertices are the variables of C as follows: we visit every decision-gate g of C. Let x be the variable labeling g and g' a gate such that (g,g') is an edge of C. If g' is a decision-gate and z labels g', we add an edge $x \to z$ in G_C. Obviously, G_C has at most s edges and n vertices and we can build G_C in time $O(s+n)$ because we only need to visit every edge of C once and each of them generates at most one edge in G_C. Now it is easy to see that a variable x is tested before a variable y in C if and only if there is a path from x to y in G_C. Hence, if G_C is acyclic, we can compute in time $O(|G_C|) = O(s+n)$ a total order $y_1, \dots, y_n$ on X such that if $j > i$ then there is no path in G_C from y_j to y_i. In other words, C is ordered by $y_1, \dots, y_n$. Conversely, if C is ordered for some order $y_1, \dots, y_n$, then clearly G_C is acyclic. Hence, we can test whether C is ordered in time $O(s+n)$. □

Non-determinism. BDDs can be augmented by adding new gates labeled by $\vee$ that represent non-deterministic choice. Such BDDs are called *non-deterministic BDDs*, abbreviated nBDD. We naturally have the corresponding notions of smooth BDD, read-once BDD (nFBDD) and ordered BDD (nOBDD). We adapt the definition of compatible paths as follows (boldface is used to show the change). Given $\tau \in 2^{\mathsf{var}(g_1)}$ and a path $P = g_1, \dots, g_k$ in C, we say that P is compatible with τ if and only if for every $i < k$, if g_i **is a decision-gate labeled by** x, then there is an edge (g_i, g_{i+1}) labeled by $\tau(x)$. For a gate g, we say that $\tau \in 2^{\mathsf{var}(g)}$ is a *model of* g or *satisfies* g if there exists a path from g to a $\top$-labeled sink that is compatible with τ. The Boolean function f_C computed by a non-deterministic BDD C on variables X is the function whose models are $\tau \in 2^X$ such that τ satisfies the source of C.



(a) A BDD that is not read-once.

(b) An FBDD that is not an OBDD.

(c) An OBDD.

(d) A decision-tree.

(e) A non-smooth OBDD.

Figure 1.2: Many shades of BDD, all computing $x \leq y \leq z$.

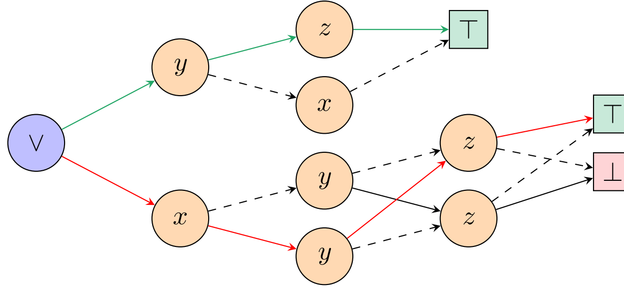


Figure 1.3: A non-deterministic FBDD. The assignment $\langle x/1, y/1, z/1\rangle$ has two compatible paths depicted in red and green.

The definition of compatible paths is roughly the same as for (deterministic) BDDs but this time, given $\tau \in 2^{\mathsf{var}(g)}$, we may have more than one path compatible with it, see Fig. 1.3 for an example.

Non-determinism adds much power to BDDs as there exist functions with polynomial size nFBDD but no FBDD of polynomial size, see Section 1.4.1 for details. However, it comes at the price of weaker tractability. The main difference, as we shall see in Sections 1.4.2 and 1.4.3, is that we cannot compute the negation of an nBDD efficiently and counting the number of models of an nBDD is now #P-hard, that is, very unlikely to admit a polynomial-time algorithm. This can be seen as a straightforward reduction to the #DNF problem since a DNF F can be easily encoded as a small nOBDD having one $\vee$-node as the source corresponding to the outer disjunction of F which is connected to one OBDD per term of F. That said, it was recently established that despite this #P-hardness, the problem of *approximating* the number of models of an nOBDD is tractable, namely, it admits an FPRAS [Are+21a], which was then generalized to the case of nFBDD [MC25]. An interesting subclass of nFBDD still supports tractable counting. The main source of #P-hardness in nFBDD comes from the fact that it is hard to compute the number of models of a given $\vee$-gate because its inputs may have common satisfying assignments. But, in some cases, it may be that these models are disjoint, in which case we say that the $\vee$-gate is *unambiguous* or *deterministic*[1]. More formally, a $\vee$-gate g is said to be *unambiguous* whenever for every $\tau \in 2^{\mathsf{var}(g)}$, there is at most one gate g' such that (g, g') is an edge of C and τ is a model of g'. For example, the $\vee$-gate from Fig. 1.3 is not

[1]There is a bit of a mismatch regarding this terminology in the literature. For BDD and automata, the usual term is unambiguous, but for NNF circuits (see Section 1.3), this has been historically called deterministic [Dar01]. Unambiguous describes the phenomenon more faithfully since determinism is usually understood as the fact that the computational process is determined by its input. When regarding an FBDD as a computation transforming an input assignment into a compatible path from the source to a sink, the path is iteratively constructed in a unique manner, so the computation is deterministic. In other words, the computation performed by a decision-gate is deterministic. Now, when encountering a $\vee$-gate, even if it is unambiguous, it is not clear at all how to either construct the unique compatible path, nor to decide whether such a path exists. The computation has to be performed possibly to the end of the diagram before being able to conclude. For these reasons, we prefer the word unambiguous when describing such $\vee$-gates but we will use the established terminology throughout this manuscript.

unambiguous because there are two compatible paths starting at two distinct successors of g.

A non-deterministic FBDD C is said to be *unambiguous*, uFBDD for short, if every $\vee$-gate of C is unambiguous. Unambiguous $\vee$-gates add a lot of succinctness to FBDD [Weg00].

Contrary to the other property previously introduced, unambiguity is not a syntactic restriction. For example, it cannot be tested as easily as smoothness. In fact, it is actually coNP-hard to check the unambiguity of a given nFBDD. More precisely:

Lemma 1.4. *The problem of deciding, given an nFBDD C and a $\vee$-gate g of C, whether g is unambiguous is coNP-complete.*[2]

Proof. The problem is obviously in coNP since it is enough to give an assignment τ satisfying two distinct successors of g to prove g is ambiguous (i.e., not unambiguous). Since checking that τ is indeed a satisfying assignment of two distinct successors of g can be performed in linear time in $O(|C|)$, this concludes the proof of the problem being in coNP.

To see the completeness, we reduce from UNSAT. Let $F = \bigwedge_{i=1}^{m} c_i$ be a CNF formula over variables $x_1, \dots, x_n$ and consider $F' = \bigwedge_{i=1}^{m} c'_i$ where c'_i is a clause obtained from c_i by replacing variable x_j by a copy $x_j^{(i)}$. It makes the variables of c'_i and c'_k disjoint for $i \neq k$. Hence, F' is computed by an FBDD C' of size $O(|F|)$ which is made from a sequence of OBDDs, each of them computing c'_i.

Now consider $F'' = \bigwedge_{i=1}^{m} (x_i^{(1)} = \dots = x_i^{(m)})$. Obviously F'' can be computed by an OBDD of size $O(nm)$. We now consider the nFBDD made from a $\vee$-gate g with inputs C' and C''. Observe that even if C' and C'' are OBDD, they do not use the same order on variables. We claim that g is unambiguous if and only if F is unsatisfiable.

Indeed, if F is satisfied by an assignment τ then we define τ' by $\tau'(x_i^{(j)}) = \tau(x_i)$. Clearly, τ' is both a satisfying assignment of C' and C'', meaning that g is ambiguous. Similarly, if g is ambiguous then let τ' be an assignment satisfying C' and C''. By definition $\tau'(x_i^{(j)}) = \tau'(x_i^{(k)})$ for every $i \leq n$ and $j, k \leq m$. The assignment τ defined by $\tau(x_i) := \tau(x_i^{(1)})$ for every $i \leq n$ is a satisfying assignment of F. Indeed, since τ' satisfies c'_j for every $j \leq m$, there exists $i \leq n$ such that c'_j contains literal ℓ' over $x_i^{(j)}$ and $\tau'(\ell') = 1$. Now, it means that there is a literal ℓ in c_j such that $\tau(\ell) = \tau'(\ell') = 1$, that is, τ satisfies c_j. □

1.3 Negation Normal Form Circuits

We now turn our attention to Negation Normal Form circuits (NNF circuits for short). In a nutshell, an NNF circuit is a Boolean circuit where every negation has been pushed to the inputs of the circuit, which is always possible using De Morgan's laws. For this reason, NNF circuits are as powerful as Boolean circuits and not particularly interesting from a tractability perspective but we define them anyway to study restrictions thereof.

[2]It seems that this result is folklore and does not appear in the literature. I remember discussing it with Antoine Amarilli while preparing [AC24] though he mentioned this proof, credited to Marcelo Arenas, though the tale may be even older. Let them be credited here anyway!

A *Negation Normal Form circuit* C *over* X (NNF for short) is a directed acyclic multigraph. The nodes of C are called gates and it contains a distinguished gate $\mathsf{out}(C)$ called the *output of* C. Moreover, it respects the following conditions:

- Every gate g with no incoming edges is labeled by either a literal over X or by a constant $b \in \{0, 1\}$, in which case we say that g is an *input of* C.
- Every other gate g is labeled by either $\vee$, in which case we say that g is a $\vee$-gate, or by $\wedge$, in which case we say that g is a $\wedge$-gate.

Observe that the only occurrence of negation in NNF circuits is on the inputs of the circuit, where negative literals can be used, which motivates the name. We usually draw circuits with their inputs at the bottom and their output at the top, see Fig. 1.4 for examples. Hence when describing algorithms or in proofs, *bottom-up* means that we are visiting the gates of the circuit from its inputs to its output; *top-down* means from its output to its inputs.

If (g', g) is an edge of C, we say that g' is an input of g and g is an output of g'. If a gate g' has more than one output, we will often say that g' is "shared" in the circuit, meaning that its computation is reused. We denote by $\mathsf{var}(g) \subseteq X$ the set of variables x such that there is a directed path from an input labeled by x or $\neg x$ to g. Alternatively, we can define $\mathsf{var}(g)$ inductively as follows: if g is an input labeled by literal ℓ, we let $\mathsf{var}(g) = \mathsf{var}(\ell)$. If g is labeled by $b \in \{0, 1\}$, we let $\mathsf{var}(g) = \emptyset$. Otherwise, $\mathsf{var}(g) = \mathsf{var}(g_1) \cup \cdots \cup \mathsf{var}(g_k)$ where $g_1, \dots, g_k$ are the inputs of g.

Given $\tau \in 2^Y$ with $Y \supseteq \mathsf{var}(g)$, we say that τ *satisfies* g or is *a model of* g if the following holds:

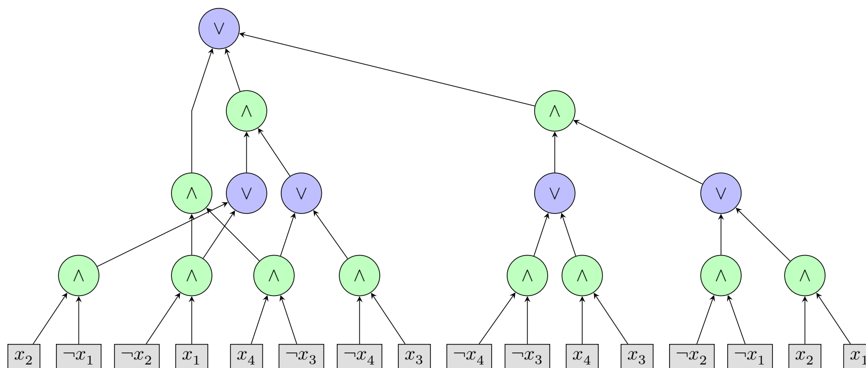
- g is an input gate labeled by either 1 or a literal ℓ such that $\tau(\ell) = 1$.
- g is a $\vee$-gate and τ satisfies at least one input of g,
- g is a $\wedge$-gate and τ satisfies every input of g.

We let f_g be the Boolean function computed by g defined as the set of assignments $\tau \in 2^{\mathsf{var}(g)}$ such that τ satisfies g. Observe that if g has inputs $g_1, \dots, g_k$ then either it is a $\vee$-gate and we have $f_g = f_{g_1} \vee \cdots \vee f_{g_k}$ or it is a $\wedge$-gate and we have $f_g = f_{g_1} \wedge \cdots \wedge f_{g_k}$. The Boolean function f_C computed by C is defined as the Boolean function computed by its output gate.

CNF and DNF formulas can both be seen as particular shallow forms of NNF circuits. For this reason, testing whether f_C has at least one model is NP-complete.

Decomposable $\wedge$-gates. Let C be an NNF circuit. A $\wedge$-gate g of C is decomposable if and only if for every pair $g_1 \neq g_2$ of inputs of g, we have $\mathsf{var}(g_1) \cap \mathsf{var}(g_2) = \emptyset$. An NNF circuit C is said to be *decomposable* if every $\wedge$-gate of C is decomposable. In this case, we also say that C is a *DNNF circuit*.

Decomposability is particularly useful for the following reason: if $g_1, \dots, g_k$ are the inputs of g and $\tau_1, \dots, \tau_k$ are models of $g_1, \dots, g_k$ on $\mathsf{var}(g_1), \dots, \mathsf{var}(g_k)$ respectively, then $\tau_1 \times \cdots \times \tau_k$ is a model of g. Observe that it does not hold for non-decomposable gates because we would need to add the assumption that $\tau_i \simeq \tau_j$. This is precisely what makes the problem SAT



(a) A DNNF circuit.

Figure 1.4: Negation Normal Form Circuits.

hard: finding a common assignment that satisfies every clause. Decomposability sidesteps this hardness and it is now easy to check whether f_C has a satisfying assignment or even enumerate them (see Section 1.4.2).

It is easy to see that DNF formulas (after removing inconsistent terms, that is, terms having opposite literals) are a particular case of DNNF circuits having only one $\vee$-node at the top and one decomposable $\wedge$-gate per term.

Deterministic $\vee$-gates. While DNNF circuits already offer a good trade-off between succinctness and tractability, they cannot be used for counting. Similarly to nFBDD, the complexity stems from the possible non-empty intersection between models of distinct inputs of a $\vee$-gate. As for FBDD, we introduce a specific kind of $\vee$-gate: the deterministic $\vee$-gates. Given an NNF circuit C and a $\vee$-gate g of C with inputs $g_1, \dots, g_k$, we say that g is *deterministic* if and only if for every $\tau \in 2^{\mathsf{var}(g)}$, there is at most one $i \leq k$ such that τ satisfies g_i. In this case, observe that the models (over $\mathsf{var}(g)$) of g_i and g_j for every $i < j \leq k$ are disjoint. A *deterministic DNNF circuit*, d-DNNF circuit for short, is a DNNF circuit where every $\vee$-gate is deterministic. Observe that if g is deterministic with inputs $g_1, \dots, g_k$ and N_i is the number of models of g_i over $\mathsf{var}(g)$, we have that the number of models of g is $\sum_{i=1}^{k} N_i$. It gives a polynomial time algorithm for counting the models of d-DNNF circuits, by inductively computing, for every gate g, the number of satisfying assignments of g over $\mathsf{var}(g)$.

Testing for determinism is coNP-complete and this is a direct consequence of Lemma 1.4 since nFBDDs are, in particular, DNNF circuits.

Corollary 1.5. *The problem of deciding, given a DNNF circuit C and a $\vee$-gate g of C, whether g is deterministic is coNP-complete.*

Often, the determinism of the circuit is either given as a promise on the input or ensured by the correctness of the algorithm that constructed the d-DNNF circuit. For example, the algorithm from [Bov+15] constructs a d-DNNF circuit from structured CNF formulas and the determinism follows from the fact that each time a $\vee$-gate g is created in the circuit with

children g_1 and g_2, then the satisfying assignments of g_1 do not satisfy the same clauses of F as those satisfying g_2.

Decision-gates. In many algorithms for knowledge compilation however, the source of determinism is even more explicit as it is syntactic. Indeed, many knowledge compilers only introduce $\vee$-gates in the form of *decision-gates*, that is, gates similar to the ones used for FBDD and OBDD: the knowledge compiler picks a variable x and creates a decision-gate g on variable x. One input of g accepts the models of g where x is set to 0 while the other input accepts the models of g where x is set to 1, as depicted on Fig. 1.5. As with BDDs, we need a read-once property: if g is a decision-gate labeled by x with input g_0, g_1, then $x \notin \mathsf{var}(g_0)$ and $x \notin \mathsf{var}(g_1)$. Observe that this condition ensures that the $\wedge$-nodes in Fig. 1.5 are decomposable. Formally, let g be a decision-gate labeled by x with inputs g_0, g_1, where the incoming edge (g_0, g) is labeled by 0 and the incoming edge (g_1, g) is labeled by 1. We say that $\tau \in 2^Y$ satisfies g if and only if either $\tau(x) = 0$ and τ satisfies g_0 or $\tau(x) = 1$ and τ satisfies g_1. Observe that with this definition, the $\vee$-gate from Fig. 1.5 is indeed deterministic since the models from its first input must set x to 0, while the models from its second input must set x to 1. A *decision-DNNF circuit* is a circuit having only inputs, decomposable $\wedge$-gates, and decision-gates. Observe that a decision-DNNF circuit can straightforwardly be transformed into a d-DNNF circuit but keeping the structure of decision-gate is often more interesting than using the resulting d-DNNF circuit.

We observe here that the orientation of edges differs from the definition of BDD. In BDD, the edges are going out of the decision-gate while in decision-DNNF circuit, we use the convention that the edges are going in the decision-gate. While this may seem confusing, it is hard to avoid: in Boolean circuits in general, gates are seen as functions producing an output from its inputs, and the inputs are oriented toward the gate. Hence, in this case, decision-gates must have their input oriented toward themselves. In BDD, a decision-gate orients a path toward the next node by deciding on the value of some variable, in which case, the decision-gate is oriented toward the next one. We do not see any way of really bridging the gap between the two notions here. In Chapter 3, we will orient the edges of decision-DNNF circuits as for BDDs, that is, from its output to its inputs. This orientation makes more sense in this chapter because we are often considering properties of paths from the output to the input and it is more natural to orient the circuit this way. When we want to stress that decision-DNNF circuits are just a specific kind of DNNF circuits, we will more often use the DNNF circuits orientation.

Structuredness. The final syntactic property that is useful for NNF circuits is the notion of *structuredness*. In a nutshell, it is a generalization of the ordering property introduced for OBDDs but in the case of NNF circuits. In BDD, a model corresponds to a path in the DAG, hence, it is natural that the additional structural properties of OBDDs restrict how these paths can test variables by imposing a static ordering. In DNNF circuits, a model corresponds to a tree since both inputs of $\wedge$-gates must be satisfied (see Lemma 5.11 later for a more detailed presentation of this fact). Hence, we will add constraints on how these trees can be structured. To do so, we introduce the notion of variable trees. Given a finite set X of variables, a *variable*

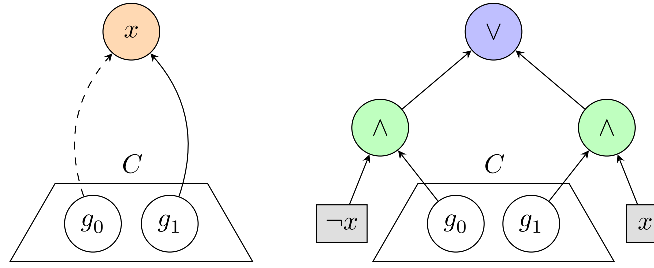


Figure 1.5: A decision-gate on variable x seen as a deterministic $\vee$-gate.

tree (vtree for short) T *on* X is a rooted tree such that every node except the leaves has exactly two children and such that the leaves of T are in one-to-one correspondence with X, that is, each leaf of T is labeled by some $x \in X$ and for every $x \in X$, there is exactly one leaf of T labeled by x. Given a node t of T, we denote by T_t the subtree of T rooted at t and by $\mathsf{var}(t) \subseteq X$ the set of variables labeling the leaves of T_t.

Given a DNNF circuit C on variables X, T a vtree on X and t a node of T with children t_1, t_2, we say that *a* $\wedge$*-gate* g *with inputs* g_1, g_2 *respects* t if and only if $\mathsf{var}(g_1) \subseteq \mathsf{var}(t_1)$ and $\mathsf{var}(g_2) \subseteq \mathsf{var}(t_2)$. If for every $\wedge$-gate g of C, g respects some node t of T, we say that C respects T. If C respects a vtree T, we say that C is a *structured DNNF circuit*. We similarly use terms structured d-DNNF circuit, structured decision-DNNF circuit etc. Structured DNNF circuits have been introduced by Darwiche in [PD10] because, like OBDDs, they can be joined together: from two structured DNNF circuit C_1, C_2 respecting the same vtree, we can build a structured DNNF circuit C' of size at most $|C_1| \cdot |C_2|$ computing $C_1 \wedge C_2$. It gives a method to transform CNF formulas into structured DNNF circuits known as *bottom-up compilation*, which we will cover extensively in Chapter 2 for a subclass of structured d-DNNF circuits.

It is straightforward to see that testing whether a DNNF circuit C on variables X respects a given vtree T can be done in polynomial time. Indeed, we can precompute $\mathsf{var}(g)$ for every gate g in the circuit and then, for every $\wedge$-gate g of C with input g_1, g_2, find the least common ancestors t_1, t_2 in T of the leaves labeled by $\mathsf{var}(g_1)$ and $\mathsf{var}(g_2)$ respectively and check that the least common ancestor t of t_1, t_2 is neither t_1 nor t_2. Computing $\mathsf{var}(g)$ for each gate g of C takes time $O(|X| \cdot |C|)$ and checking one $\wedge$-gate can be done in time $O(|X|)$ hence a total running time of $O(|X| \cdot |C|)$.

Lemma 1.6. *Given a DNNF circuit* C *and a vtree* T *on variables* X*, one can check whether* C *respects* T *in time* $O(|X| \cdot |C|)$.

Normal Forms. We conclude this section with a few observations on NNF circuits. It is often handy to assume a few syntactic properties of the circuit. We show here that some of them can always be ensured. We first turn our attention to constants. While allowing inputs labeled by a constant $b \in \{0, 1\}$ is useful, it is sometimes better to remove them to ensure that every subcircuit contains at least one variable. Fortunately, we can always do this in linear time.

Lemma 1.7. *Let C be an NNF circuit which does not compute a constant Boolean function. We can compute in time $O(|C|)$ an equivalent NNF circuit C' of size at most $|C|$ which does not use constant inputs. Moreover, the transformation preserves structuredness, determinism and decomposability.*

Proof. Let g be an input of C labeled by $b \in \{0,1\}$. If g' is a gate that has g as input, we simplify it as follows:

- If g' is a $\vee$-gate and $b = 0$, or if g' is a $\wedge$-gate and $b = 1$, we remove the edge (g, g').
- If g' is a $\vee$-gate and $b = 1$, or if g' is a $\wedge$-gate and $b = 0$, we remove the edge (g, g') and relabel g' with b.

This transformation may leave some gates isolated (that is, without outputs), in which case, we remove them from the circuit. Similarly, some $\vee$-gate or $\wedge$-gate may not have inputs anymore. We replace $\wedge$-gate without inputs by 1 and $\vee$-gate without inputs by 0.

It is easy to see that this transformation does not change the function computed by the circuit and that it preserves structuredness, determinism and decomposability of every gate in the circuit. Moreover, it decreases the size of the circuit by at least one. Therefore, applying the transformation iteratively converges toward a circuit where it cannot be applied anymore, hence where no constant gates remain. Finally, each transformation takes time proportional to the number of edges connected to g, and it removes them all. Hence, the total time needed to eliminate constants is $O(|C|)$. □

Another useful transformation is to make the circuit smooth. A circuit is smooth if for every $\vee$-gate g and input g' of g, we have $\mathsf{var}(g) = \mathsf{var}(g')$. We can always complete the circuit to make it smooth.

Lemma 1.8. *Given an NNF circuit C on variables X, we can compute a smooth NNF circuit C' of size at most $O(|X| \cdot |C|)$ computing the same function as C in time $O(|X| \cdot |C|)$. The transformation preserves determinism and decomposability.*

Proof. First, observe that for any set of variables $Y \subseteq X$, we can build a smooth deterministic DNNF circuit C_Y of size $O(|Y|)$ accepting every assignment over Y. C_Y is simply $\bigwedge_{y \in Y}(y \vee \neg y)$. Now if g is a $\vee$-gate of C and g' an input of g with $\mathsf{var}(g') \subsetneq \mathsf{var}(g)$, we let $\Delta = \mathsf{var}(g) \setminus \mathsf{var}(g') \neq \emptyset$. We disconnect g' from g and add a $\wedge$-gate g'' computing $g' \wedge C_\Delta$, and connect g'' to g. Clearly, g'' computes $2^\Delta \times f_{g'}$ and is decomposable, hence the function computed by g has not changed. Moreover, $\mathsf{var}(g'') = \mathsf{var}(g)$. Since C_Δ is smooth, we hence have reduced the number of edges (g', g) where g is a $\vee$-gate and $\mathsf{var}(g') \subsetneq \mathsf{var}(g)$. We can iteratively apply this transformation until every $\vee$-gate is smooth. Each edge of C may incur $O(|X|)$ new gates, hence the resulting circuit has size at most $O(|X| \cdot |C|)$ and it can be built in this amount of time. □

In the case of structured NNF, smoothing can still be performed while preserving structuredness but one has to work inductively on the vtree T and ensure that every $\wedge$-gate g respecting a node t of T is such that $\mathsf{var}(g) = X_t$. If not, one can add a similar gadget as

before to ensure it. We can then force $\vee$-gates to be connected to other $\vee$-gates or $\wedge$-gates g' with $\mathsf{var}(g') = X_t$ for some t. Because of structuredness, the gadgets used in different parts of the circuit will share a lot of structure. By using a clever construction, Shih, Van den Broeck, Beame and Amarilli have shown that smoothing structured circuits can actually be done in time and space $O(\alpha(|C|, |X|) \cdot |C|)$ where α is the inverse Ackermann function [Shi+19].

1.4 The Knowledge Compilation Map

We have so far introduced many different data structures that allow us to represent Boolean functions. To compare their respective usefulness, it is first important to understand how they compare with one another. First of all, it is important to understand the succinctness of each language. For example, does forcing an order in OBDDs make them really less succinct than FBDDs or can we always pay a reasonable increase in size to ensure this syntactic property? It turns out that some functions have polynomial size FBDD but no polynomial size OBDD. The second thing that is important to understand is the tasks we are able to solve depending on the representation of a given Boolean function. In their seminal paper [DM02], Darwiche and Marquis have proposed to summarize results as tables explaining for each representation the tasks that could be solved in polynomial time on them and the transformations that are possible.

In this section, we review some results concerning how representation languages compare with one another and present some queries and transformations that will be interesting in the rest of the manuscript and give a high-level description of the algorithms involved.

1.4.1 Comparing Representations

We have introduced several representations of Boolean functions as NNF circuits or decision diagrams. We are now interested in understanding how relatively succinct they are. Given two representation languages $\mathcal{C}_1, \mathcal{C}_2$ of Boolean functions, we will say that $\mathcal{C}_1$ is *at least as succinct* as $\mathcal{C}_2$, denoted by $\mathcal{C}_1 \leq \mathcal{C}_2$ if and only if for every $C \in \mathcal{C}_2$ then there exists $C' \in \mathcal{C}_1$ computing the same function as C and such that $|C'| = \text{poly}(|C|)$. We say that $\mathcal{C}_1$ is *more succinct* than $\mathcal{C}_2$, and write $\mathcal{C}_1 < \mathcal{C}_2$, if we do not have $\mathcal{C}_2 \leq \mathcal{C}_1$. In other words, it means that there exists a sequence $(f_n)_{n \in \mathbb{N}}$ of Boolean functions and a superpolynomial function $w \colon \mathbb{N} \to \mathbb{N}$ such that: for every $n \in \mathbb{N}$, f_n has a representation $C_1 \in \mathcal{C}_1$ of size s_n, with $s_n \to +\infty$ as $n \to +\infty$ and every representation $C_2 \in \mathcal{C}_2$ of f_n has size at least $w(s_n)$.

For example, $FBDD \leq OBDD$ because an OBDD is in particular an FBDD, so for every OBDD C, there exists an FBDD of size polynomial in $|C|$ and computing the same function, in this case, C itself. We also have $FBDD < OBDD$. Indeed, consider functions $ROW(M_n)$ and $COL(M_n)$ which respectively decide whether there is a row and a column of all ones in an $n \times n$ matrix $M_n = (x_{i,j})_{i,j \leq n}$ of Boolean variables and let s be a distinct Boolean variable. Then $f_n = (s \wedge ROW(M_n)) \vee (\neg s \wedge COL(M_n))$ can be represented by an FBDD of size $O(n^2)$ but every OBDD representing f_n must have size at least 2^{cn} for some constant c [Weg00].

It is obvious that FBDD $\leq$ decision-DNNF $\leq$ d-DNNF $\leq$ DNNF $\leq$ NNF because they are a succession of restrictions of NNF circuits. All these succinctness results are strict. Separating

representations $\mathcal{C}_1$ and $\mathcal{C}_2$ often relies on finding the right function that is easy for $\mathcal{C}_1$ but hard for $\mathcal{C}_2$. Proving that a function is easy for $\mathcal{C}_1$ is usually the easiest part as one only has to come up with an algorithm computing the function with the right representation. On the other hand, proving that a function cannot be represented with polynomial size in $\mathcal{C}_2$ requires a precise understanding of the computational process. We give below a few examples:

- Decision-DNNF circuits are more succinct than FBDD [Raz14]. Razgon uses a Boolean function encoding the vertex covers of a graph having n vertices and whose treewidth is $O(\log n)$ and pathwidth is $O(\log^2 n)$. The smallest FBDD for this function is $n^{c \log n}$ for some constant $c > 0$ while we can represent it with a decision-DNNF circuit of size poly(n). The separation is not really exponential and we can actually prove that any decision-DNNF circuit of size s can be represented as an FBDD of size $s^{\log s}$ [Bea+14]. The bound for FBDD actually works even if we allow non-deterministic nodes, showing that nFBDD are not more succinct than decision-DNNF circuits.

- d-DNNF circuits are more succinct than decision-DNNF circuits. We can use a function

$$f_n = (PARITY(M_n) \wedge ROW(M_n)) \vee (\neg PARITY(M_n) \wedge COL(M_n))$$

  to separate them [Bea+13]. Intuitively, the PARITY function implies that the disjunction is deterministic but it is hard to encode with decision-gates.

- DNNF circuits are more succinct than d-DNNF circuits. We prove this result in [Bov+16] using tools from communication complexity and a Boolean function proposed by Sauerhoff [Sau03].

- NNF circuits are more succinct than DNNF circuits. We can actually prove an even stronger separation, namely that there exists a CNF formula F of size s (which are a restricted form of NNF circuit) such that any DNNF circuit computing F has size $2^{O(s)}$. We can prove the separation by relying again on the communication complexity techniques from [Bov+16]. The function used here encodes vertex covers of bounded degree expander graphs [Cap16, Chapter 6]. Observe however that some function can be represented as succinct FBDDs (even OBDDs) but not by succinct CNF formulas. For example, the $PARITY_n$ function is known to have only exponential size circuits of bounded depth [Has86], hence only exponential size CNF formulas. But this function has an OBDD of size $O(n)$.

The goal of this section is mostly to give a feeling about what it means for representations to be more succinct than others and how this can be obtained unconditionally. There is, however, one open question that is, in our opinion, worth investigating, namely:

Open question 1. *Prove a superpolynomial lower bound on the size of d-DNNF circuits representing DNF formulas.*

Indeed, while we can separate d-DNNF from DNNF circuits, we cannot separate DNF formulas from d-DNNF circuits. It is clear that the function $PARITY_n$ has small d-DNNF

circuit but only exponential size DNF formulas, but it is not clear whether there exists DNF formulas having no polynomial size d-DNNF circuits. We observe that if every DNF formula can be transformed into polynomial size d-DNNF circuits, then this transformation is very unlikely to be possible in polynomial time. Indeed, it is known that finding the number of models of a DNF formula is #P-hard while computing the number of models of a d-DNNF circuit C can be done in time polynomial in $|C|$. Hence, a polynomial time transformation from DNF formulas to d-DNNF circuits would give a polynomial time algorithm for a #P-complete problem, which is unlikely.

We also observe that in general, we have separations between the structured and the unstructured version of the same circuit class. For example, OBDDs are exponentially weaker than FBDDs, as mentioned earlier in this section. Similarly, structured d-DNNF circuits are exponentially weaker than d-DNNF circuits. We can even exhibit a Boolean function having small FBDD but only exponential sized structured d-DNNF circuits, see [Cap16, Chapter 6] for an example.

1.4.2 Tractable queries

We are mostly interested in two queries: model enumeration and model counting. Observe that the knowledge compilation map reviews other interesting queries but it will not be the focus of this document.

Model enumeration. Model enumeration is the task of listing every model of a Boolean function f. The quality of enumeration algorithms may be measured along several parameters. The easiest one is the total time needed to perform the operation. In this case, it is customary to take the output size into account when reasoning about the complexity of a given enumeration algorithm. For example, we say that an enumeration problem is *output polynomial* if every solution to the problem can be listed in time polynomial in the size of the input plus the size of the output. That said, one is usually more interested in the time needed to output a new solution. Hence, given an enumeration algorithm producing solutions, we will study its complexity in terms of *preprocessing*, that is, the time needed to output the first solution, and *delay*, that is, the maximal time needed between the output of two distinct solutions. There are many other relevant ways of studying enumeration algorithms, and this has been one focus of my research [CS19; CS21; CS23] but this is not needed within the scope of this manuscript and the interested reader can learn more about it in Yann Strozecki's thesis and habilitation manuscript [Str10; Str23].

Given a representation language $\mathcal{C}$ of Boolean functions, we will say that model enumeration is tractable for $\mathcal{C}$ if we can output every model of $C \in \mathcal{C}$ with preprocessing and delay polynomial in $|C|$. It is clear that model enumeration is not tractable for NNF circuits, as it would mean in particular that we can solve SAT in polynomial time. The most general class of circuits for which model enumeration is tractable is DNNF circuits:

Theorem 1.9. *Given a DNNF circuit C over variables X, we can enumerate the models of C with preprocessing and delay $O(|C| \cdot |X|)$.*

Proof. The algorithm relies on the following observation: given $Y \subseteq X$ and $\tau \in 2^Y$, we can decide in time $O(|C|)$ whether there exists $\sigma \in 2^{X \setminus Y}$ such that $\tau \times \sigma$ is a model of C. To do so, we compute a value $m_g \in \{0,1\}$ at each gate g of C such that $m_g = 1$ if and only if g has a model that is compatible with τ. If g is an input labeled by a literal ℓ over a variable $y \in Y$, then we let $m_g = \tau(\ell)$. If the literal is over $x \in X \setminus Y$, then $m_g = 1$. If g is a $\wedge$-gate with input $g_1, \dots, g_k$, then $m_g = m_{g_1} \times \dots \times m_{g_k}$ and if g is a $\vee$-gate, we let $m_g = \max_{i \leq k} m_{g_i}$. It is readily verified that if o is the output of C then $m_o = 1$ if and only if C has a model compatible with τ.

To enumerate the models of C, we use the following strategy: we first pick an arbitrary order $x_1, \dots, x_n$ on X. Then we recursively define a function $enum(C, \tau)$ for $\tau \in 2^{\{x_1,\dots,x_i\}}$ which enumerates the models of C compatible with τ as follows:

- If $x_i = x_n$, we output τ.
- If C has a model compatible with $\tau \times \langle x_{i+1}/0 \rangle$, we recursively call $enum(C, \tau \times \langle x_{i+1}/0 \rangle)$.
- If C has a model compatible with $\tau \times \langle x_{i+1}/1 \rangle$, we recursively call $enum(C, \tau \times \langle x_{i+1}/1 \rangle)$.

Clearly $enum(C, \langle\rangle)$ enumerates every model of C. More interestingly, we never perform a recursive call without being sure that there is at least one model below, by using the fact that we can test whether C has a model compatible with τ. Hence, after at most n recursive calls, we find at least a new model of C. Since each recursive call takes time $O(|C|)$ (the time needed to test whether $\tau \times \langle x_{i+1}/b \rangle$ can be extended to a full model of C), we have delay $O(|X| \cdot |C|)$. □

One consequence of Theorem 1.9 is that enumeration is tractable for d-DNNF circuits, decision-DNNF circuits and FBDD as well. That said, when the circuit is deterministic, we can improve the delay. Indeed, if C is a d-DNNF circuit, then we can enumerate the models of a gate g using the following recursive approach:

- If g is a $\vee$-gate with inputs $g_1, \dots, g_k$, then we initialize i to 1 and while $i \leq k$, we recursively enumerate the models of g_i. When done, we increment i by 1. Since $g_1, \dots, g_k$ have disjoint models by determinism, we have no duplicates in the output and list every model of g. The delay d_g to enumerate the models of g is equal to $K_1 + \max_{i \leq k} d_{g_i}$ for some constant K_1 representing the overhead of orchestrating the different enumeration algorithms.

- If g is a $\wedge$-gate with inputs g_1, g_2, then we enumerate the models of g_1. Each time a model τ of g_1 is found, we enumerate the models of g_2 and output $\tau \times \sigma$ for every model σ of g_2. When we reach the end of the enumeration of g_2, then we enumerate the next model of g_1 and enumerate again every model of g_2. The delay d_g is then $K_2 + d_{g_1} + d_{g_2}$ where K_2 is again a constant overhead for orchestrating the interlaced enumeration of g_1 and g_2.

For simplicity, we have assumed above that the $\wedge$-gates of C have fan-in 2, but if they have larger fan-in, with inputs $g_1, \dots, g_k$, we can adapt the same idea by seeing it as a tree of $\wedge$-gates of fan-in 2.

It is not hard to see by induction that, if constants have been removed from C in a preprocessing step of $O(|C|)$, the delay of the previously described algorithm is $O(|X| \times h(C))$ where $h(C)$ is the height of C, that is, the longest path from an input to the output of C. Observe that the dependency in $h(C)$ stems purely from the fact that inputs of $\vee$-gates may also be $\vee$-gates and there may be long paths of $\vee$-gates that we need to go through before going on. There are two ways of circumventing this difficulty. The first easy way is to only consider decision-DNNF circuits. In this case, the depth is bounded by $O(|X|)$ once constant gates have been removed, because if g is a gate with input g', then we have $\mathsf{var}(g') \subsetneq \mathsf{var}(g)$. In this case, we can actually show that the previously described algorithm has delay $O(|X|)$.

A more subtle approach for improving the naive $O(|X| \cdot h(C))$ delay has been proposed in [Ama+17] by Amarilli, Bourhis, Jachiet and Mengel. The trick is to use the preprocessing to jump over long paths of $\vee$-gates. The key observation here is that determinism enforces that given a $\vee$-gate g, if we consider the set of $\vee$-gates g' such that there is a path from g' to g made entirely of $\vee$-gates, then this subcircuit must be a tree and its leaves are connected to only $\wedge$-gates $g_1, \dots, g_k$, known as the *exit gates of* g. The models of g are the union of models of its exit gates. Indeed, if there is an undirected cycle, it would break determinism. Using a clever linear time preprocessing, they are able to construct a data structure such that given a $\vee$-gate g of C, one can enumerate its exit gates with constant delay. Using this approach, we can reduce the delay of the enumeration to $O(|X|)$, after a linear preprocessing. We hence have:

Theorem 1.10 (Adapted from [Ama+17])**.** *Given a DNNF circuit C on variables X, we can enumerate the models of C with preprocessing $O(|C|)$ and delay $O(|X|)$ if C is deterministic and $O(|X| \cdot |C|)$ otherwise. The space needed for enumeration is at most $O(|X| \cdot |C|)$.*

We observe that these algorithms can be generalized to weighted models and we now want to enumerate the models in nondecreasing order of weights (as long as the weights of models are computed from independent weights on variables), see [Ama+24]. However, in the ordered enumeration setting, the space consumption of the algorithms is now exponential since we must keep track of the ordered models.

Model Counting. The other main query that we will focus on in this manuscript is the problem of computing the number of models of a Boolean function f, which we will denote by $\#f$. This problem is known to be #P-complete when f is given as a CNF formula, a problem usually referred to as #SAT. We will not need much knowledge of the complexity theory of counting problems in this manuscript which has been introduced by Valiant [Val79]. Hence, we omit the detailed definitions. A good introduction to the topic may be found in [AB09]. The only thing that we need to know is that a problem is in #P if it can be expressed as the number of accepting paths of a non-deterministic Turing machine running in polynomial time. For example, #SAT can be seen as the number of accepting paths of a Turing machine that, given a CNF F on variables X, non-deterministically guesses an assignment of $\tau \in 2^X$ and accepts only if τ is a satisfying assignment of F. The number of accepting paths of this Turing machine is clearly the number of models of F. Now, guessing τ can be done with $O(|X|)$ non-deterministic steps and checking whether τ is a satisfying assignment of F can be done in polynomial time, so it proves that #SAT is a problem in #P.

The notion of #P-completeness is a generalization of NP-completeness: intuitively, we want a problem A to be #P-complete to mean that it is in #P and every problem B in #P reduces to it, in the sense that having an algorithm for A gives an algorithm for B via an easy to perform transformation. Several notions of reduction may be defined and they yield different notions of completeness. The most powerful being parsimonious reduction, where we want that an instance of B can be encoded as an instance of A, of polynomial size, and having exactly the same value. The core of the proof of the Cook-Levin theorem[Coo71; Lev73] being to encode the runs of a polynomial time non-deterministic Turing machine, it gives a parsimonious reduction from any problem in #P to #SAT. We can also consider reductions allowing postprocessing in the following sense: we can transform an instance I_1 of B into an instance I_2 of A such that $B(I_1) = g(A(I_2))$ where g is a polynomial time computable function.

Under such reduction, it gives that the problem #DNF of computing the number of satisfying assignments of a DNF formula is #P-complete. Indeed, we can reduce #SAT to it: given a CNF formula F, we can write $G = \neg F$ as a DNF formula. Now since $\#G = 2^n - \#F$, we have a reduction from #SAT to #DNF. This reduction is not parsimonious and it is very unlikely that such a reduction exists. Indeed, we know that #DNF admits an approximation algorithm [KLM89] (more precisely an FPRAS). A parsimonious reduction from #SAT to #DNF would mean that #SAT also admits an approximation algorithm, which is unlikely unless P=NP [Rot96].

After this loose introduction to the complexity theory of counting problems, we turn our attention to model counting problems when the input is given as an NNF circuit. Since #DNF is #P-complete already, we directly have that #DNNF is also #P-complete because every DNF formula is, in particular, a DNNF circuit. Now, observe that if f is given as a deterministic DNNF circuit C on variables X, then it is tractable to compute $\#C$. Indeed, we can inductively compute $\#g$, the number of models of a gate g over $\mathsf{var}(g)$ as follows: if g is an input then $\#g = 0$ if it is labeled by 0 and $\#g = 1$ otherwise. If g is a $\wedge$-gate with inputs $g_1, \dots, g_k$ then because of decomposability, $\#g = \prod_{i=1}^{k} \#g_i$. Finally, if g is a $\vee$-gate with inputs $g_1, \dots, g_k$, we have $\#g = \sum_{i=1}^{k} 2^{\delta_i} \#g_i$ where $\delta_i = |\mathsf{var}(g) \setminus \mathsf{var}(g_i)|$. Since we can precompute $\mathsf{var}(g)$ for every gate g of C in time $O(|C| \cdot |X|)$, we can compute δ_i in time $O(|X|)$. It shows that we can compute $\#C$ by doing at most $O(|C|)$ arithmetic operations, for a total running time $O(|C| \cdot |X|)$. Observe that if the circuit is smooth, then we do not need to compute δ_i and we have a complexity of $O(|C|)$ arithmetic operations. One has to be careful, however: the integers $\#g$ used in this algorithm may have a value up to $2^{|X|}$ and will be encoded with up to $O(|X|)$ bits. The bit complexity, even in the smooth case, will hence be $O(|C| \cdot |X|)$ on the RAM model.

Theorem 1.11. *Given a d-DNNF circuit C on variables X, we can compute the number of models $\#C$ of C in time $O(|C| \cdot |X|)$.*

We observe that the algorithm for counting the models of C also works when we have weights on literals. In this setting, given a Boolean function f on variables X and a weight function $w \colon X \times \{0, 1\} \to \mathbb{Q}$, we want to compute:

$$w(f) := \sum_{\tau \models f} w(\tau) \qquad \text{where } w(\tau) := \prod_{x \in X} w(x, \tau(x)).$$

In this case, we can use the same dynamic programming as before. For an input labeled by x, we set $\#g = w(x, 1)$ and we set $\#g = w(x, 0)$ for inputs labeled by literal $\neg x$. Also, if C is not smooth, one has to replace 2^{δ_i} by $\prod_{x \in \mathsf{var}(g) \setminus \mathsf{var}(g_i)}(w(x, 0) + w(x, 1))$ when computing the value of a $\vee$-gate.

More interestingly, we can perform model counting over any semiring $(\mathbb{K}, \oplus, \otimes)$. A semiring is an algebraic structure with two commutative and associative operations $\oplus$ and $\otimes$ with respective identity elements $0_{\mathbb{K}}$ and $1_{\mathbb{K}}$ such that for every $a, b, c \in \mathbb{K}$, $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$. Moreover, $a \otimes 0_{\mathbb{K}} = 0_{\mathbb{K}} \otimes a = 0_{\mathbb{K}}$. In this case, we are given a weight function $w \colon X \times \{0, 1\} \to \mathbb{K}$ and we want to compute:

$$w(f) := \bigoplus_{\tau \models f} w(\tau) \qquad \text{where } w(\tau) := \bigotimes_{x \in X} w(x, \tau(x)).$$

This problem, known as *algebraic model counting* in [KVD17], has been shown to be tractable on d-DNNF circuits. Essentially, the same algorithm as the one previously described works, by simply replacing addition with $\oplus$ and multiplication with $\otimes$. Again, in the case of non-smooth circuits, one should normalize by missing variables and multiply by $\bigotimes_{x \in \mathsf{var}(g) \setminus \mathsf{var}(g_i)}(w(x, 0) \oplus w(x, 1))$, which is the weight of the tautology of $\mathsf{var}(g) \setminus \mathsf{var}(g_i)$ since we can distribute $\otimes$ over $\oplus$.

One semiring will be particularly interesting for us in Chapter 5: the tropical semiring $(\mathbb{Q}, \max, +)$. Over this semiring, we have

$$w(f) = \max_{\tau \models f} \sum_{x \in X} w(x, \tau(x)).$$

It is hence an optimization problem where one is looking for the maximal weight of an assignment. The tropical semiring has an interesting property: it is *additively idempotent*. A semiring $(\mathbb{K}, \oplus, \otimes)$ is additively idempotent if for every $a \in \mathbb{K}$, we have $a \oplus a = a$. In the tropical semiring, $\oplus$ corresponds to max and we indeed have $\max(a, a) = a$. For idempotent semirings, determinism is not needed to compute $w(f)$. We will use this fact to solve optimization problems via knowledge compilation in Chapter 5.

Theorem 1.12 ([KVD17])**.** *Given a weight function w valued in a semiring $\mathbb{K}$ and a d-DNNF circuit C, we can compute $w(f_C)$ by doing $O(|C|)$ semiring operations. If $\mathbb{K}$ is additively idempotent, then we can compute $w(f_C)$ with $O(|C|)$ semiring operations even if C is not deterministic.*

We conclude this section on model counting by quickly reviewing recent results concerning the complexity of approximating the model count in classes of circuits where it is #P-hard to compute exactly. As mentioned before, the earliest result in this direction is certainly the fact that #DNF admits an FPRAS [KLM89]. A recent breakthrough by Arenas, Croquevielle, Jayaram and Riveros [Are+21a] gives an FPRAS for counting the number of words of a given length N accepted by a non-deterministic finite automaton. It is not hard to see that this is actually equivalent to having an FPRAS for the #nOBDD problem. They proposed a generalization to tree automata that could be also seen as an FPRAS for structured d-DNNF circuits [Are+21b]. In both works, structuredness is essential for their algorithms to work.

This restriction has been recently lifted by Meel and de Colnet in [MC25] where they show that #nFBDD and, later, #DNNF[MC26] admit an FPRAS.

1.4.3 Tractable Transformations

As before, we will only consider a limited set of transformations in this document that will be relevant for us. We refer to the knowledge compilation map for more details [DM02]. In this section, we use the word "transformation" as a mapping from one Boolean function f (and possibly other parameters) to another Boolean function. For example, we may be interested in the "negation" that maps a Boolean function f on variables X to the Boolean function $\neg f$ on variables X defined as $\neg f(\tau) = 1 - f(\tau)$ for every $\tau \in 2^X$. Similarly, conditioning maps a Boolean function f on variables X and $\tau \in 2^Y$ to the Boolean function $f[\tau]$ on variables $X \setminus Y$ defined as $f[\tau](\sigma) = f(\tau \times \sigma)$ for every $\sigma \in 2^{X \setminus Y}$.

A class of representations $\mathcal{C}$ is said to *support a transformation* T if for every $C \in \mathcal{C}$, there exists a circuit $C' \in \mathcal{C}$ of size polynomial in $|C|$ such that C' computes $T(f_C)$. Note that it does not mean that C' can be constructed in polynomial time from C although it is often the case. In the case where, in addition, we can construct C' in polynomial time, we will always state it explicitly. We will say, in this case, that *the transformation T is tractable for $\mathcal{C}$*.

For example, negation is tractable for OBDD since we can always construct an OBDD for $\neg C$ from C by swapping its $\top$ and $\bot$ sinks. On the other hand, since DNNF circuits and CNF formulas are exponentially separated [Bov+16], we know that DNNF circuits does not support negation. Indeed, let $(F_n)_{n \in \mathbb{N}}$ be a family of CNF formulas such that F_n has no DNNF circuit of size smaller than $2^{c\|F_n\|}$ for some constant c and let $G_n = \neg F_n$. For every $n \in \mathbb{N}$, $G_n = \neg F_n$ can be represented by a DNF and hence a DNNF circuit of size $O(\|F_n\|)$ by simply applying De Morgan's law. But by definition, $\neg G_n = F_n$ cannot be represented by DNNF circuits of size smaller than $2^{c\|F_n\|}$.

In contrast with the case of tractable queries, it is not necessarily true that if a class of representations $\mathcal{C}$ supports a transformation T then every class of representations $\mathcal{C}' \subseteq \mathcal{C}$ also supports the transformation. Indeed, we only know that for every $C \in \mathcal{C}'$, there exists $C' \in \mathcal{C}$ computing $T(f_C)$ but there is no a priori reason that C' is in $\mathcal{C}'$. A notorious example of this phenomenon is the existential projection of variables which is tractable for DNNF circuits but not for OBDDs.

Conditioning. Conditioning is one of the most fundamental transformations. As explained before, given a Boolean function f on variables X and $\tau \in 2^Y$, *f conditioned on τ* is the Boolean function $f[\tau]$ on variables $X \setminus Y$ defined as $f[\tau](\sigma) = f(\tau \times \sigma)$. Every representation introduced so far supports conditioning. In NNF circuits, it is indeed enough to replace an input labeled by a literal ℓ on a variable $y \in Y$ by the constant $\tau(\ell)$. It is straightforward to see that the resulting circuit computes $f[\tau]$ and that the transformation preserves determinism, structuredness, and so on. For BDDs, one simply has to modify every edge (h, g) entering a decision-gate g labeled by $y \in Y$ as (h, g') where g' is the gate such that (g, g') is labeled by $\tau(y)$.

Conditioning is important because it allows us to show that every tractable query on a class of representation $\mathcal{C}$ is still tractable when the truth values of some variables are fixed. For

example, if f is represented as a d-DNNF circuit, we can quickly find the number of models of f extending a partial assignment τ by first conditioning to obtain a d-DNNF circuit computing $f[\tau]$ and then counting the number of models of this new d-DNNF circuit.

We observe here that the literature contains a few examples of representations which do not support conditioning or for which, it is not yet known whether they support conditioning or not. This is the case of canonical SDD [Dar11; VD15] where enforcing canonicity (without changing the vtree) may blow up the size of the circuit exponentially, see Section 2.1.5 for details. It is also the case for representations exploiting symmetries, for example [Bar+14], where subcircuits computing the same function after renaming variables can be shared. In this case, conditioning may break the symmetries and induce a blow-up in the size of the circuit.

Negation. Surprisingly, only a few representations presented so far support negation. It is clear that for OBDD and FBDD, it is sufficient to swap the 1-sink and the 0-sink to construct a new diagram computing the negation. For other representations, it is in general not possible or not known to be possible. We have already explained that the separation between DNNF circuits and CNF is enough to show that there exist DNNF circuits such that their negation does not have any small DNNF circuits. The same argument shows that nOBDD and nFBDD do not support negation either. In many cases, non-determinism is enough to make negation hard. But the case of unambiguity is not settled either. Recently, Vinall-Smeeth has shown that structured d-DNNF circuits do not support negation [Vin24] but the question is still open for d-DNNF circuits:

Open question 2. *Do d-DNNF circuits support negation?*

We can consider an extension of d-DNNF circuit using negation gates. This has been considered in the literature under the name d-D circuits [Mon20] (for deterministic Decomposable circuits) or POG [Bry+25] (for Partitioned Operation Graphs). In this case, d-D circuits obviously support negation since we can simply add a negation gate at the top of the circuit. d-D circuits retain some tractability properties such as model counting or weighted model counting, but not others. For example, algebraic model counting on an arbitrary semiring is not necessarily tractable, though, to the best of our knowledge, no hardness results have been formally shown in this direction. One related open question asked by Mikaël Monet (personal communication) is the following: given a weight function over $(\mathbb{Q}, \max, +)$ and a d-DNNF circuit C computing f, what is the complexity of finding the maximum weight of a model of $\neg f$, in symbols, $\max_{\tau \models \neg f} w(\tau)$?

In Chapter 2, we propose a new data structure which supports negation and is between OBDD and structured d-DNNF circuits.

Existential projection. Existential projection, also called forgetting, is the operation that maps a Boolean function f over variables X and a subset $Y \subseteq X$ to the Boolean function $\exists Y.f$ on variables $X \setminus Y$ such that $\sigma \in 2^{X \setminus Y}$ is a model of $\exists Y.f$ if and only if there exists $\tau \in 2^Y$ such that $\sigma \times \tau$ is a model of f.

By conditioning, it is not too hard to see that if we are given a DNNF circuit C, we can decide whether σ is a model of $\exists Y.f$ by checking whether $C[\sigma]$ has a model. It should therefore

not be surprising that DNNF circuits supports existential projection. The transformation is actually very simple: we replace every input labeled by y and by $\neg y$ for $y \in Y$ by a 1 input. One can show that the resulting circuit computes $\exists Y.C$. A similar transformation also works for nFBDDs or nOBDDs by simply replacing every decision-gate over variable $y \in Y$ by a non-deterministic node $\vee$.

Theorem 1.13. *Given a DNNF circuit (resp. an nOBDD, an nFBDD) C on variables X and $Y \subseteq X$, we can compute in time $O(|C|)$ a DNNF circuit (resp. an nOBDD, an nFBDD) C' computing $\exists Y.C$ with size at most $|C|$.*

This transformation may however break determinism in the following sense: if C is a d-DNNF circuit and we replace every input labeled by y or $\neg y$ by 1, we obtain a DNNF circuit computing $\exists y.C$ but this circuit is no longer deterministic. Indeed, consider for example $(x \wedge \neg y) \vee (x \wedge y)$. The circuit is deterministic but after the transformation, we get $(x \wedge 1) \vee (x \wedge 1)$ which is not deterministic anymore.

We can actually show that neither OBDD, FBDD, structured d-DNNF nor d-DNNF circuits support existential projection. This is actually easy to see using known separations of circuits and we illustrate it on d-DNNF circuits. Indeed, consider a function f that is hard for d-DNNF circuits but easy for DNNF circuits and consider a small DNNF circuit C for f and assume without loss of generality that C has fan-in 2. Replace each $\vee$-gate with inputs g_1, g_2 of C with $(x_g \wedge g_1) \vee (\neg x_g \wedge g_2)$ where x_g is a fresh variable. We obtain a new circuit C' that is deterministic. Let Z be the set of fresh variables x_g that have been introduced in the circuit. It is clear that $\exists Z.C'$ computes f. Hence C' has a small d-DNNF circuit but $\exists Z.C'$ does not.

We observe however that in some cases, this projection actually preserves determinism. Indeed, let f be a Boolean function on variables X, $Z \subseteq X$ and assume that the Z-variables we want to project are defined by $X \setminus Z$ in f, that is, for all models τ_1, τ_2 of f, if $\tau_1|_{X\setminus Z} = \tau_2|_{X\setminus Z}$ then $\tau_1 = \tau_2$. This often happens when the Z-variables are used for encoding constraints, for example, when performing a Tseitin encoding of a Boolean circuit. For example, we could use a z variable to encode $z \Leftrightarrow x_1 \wedge x_2$, in which case, fixing the value of x_1, x_2 completely determines the value of z. We claim that if Z is defined by $X \setminus Z$ in f and if f is represented by a d-DNNF circuit C, then $\exists Z.C$ remains deterministic, see [MW20, Lemma 18]. In a nutshell, the argument is as follows: if some deterministic $\vee$-gate g is connected to the output of C and not deterministic after projecting Z variables, it means that there exists some $\tau_1, \tau_2 \in 2^{\mathsf{var}(g)}$ such that $\tau_1|_{\mathsf{var}(g)\setminus Z} = \tau_2|_{\mathsf{var}(g)\setminus Z}$ and $\tau_1|_Z \neq \tau_2|_Z$. Now, because d-DNNF circuits are decomposable and g can be reached from the root, we know that there exists $\sigma \in 2^{X\setminus \mathsf{var}(g)}$ such that $\sigma \times \tau_1$ and $\sigma \times \tau_2$ are both models of C [Bov+16], which contradicts the fact that Z is defined by $X \setminus Z$ in f.

In Section 2.3.1, we show that in structured circuits, we can sometimes control the blow-up of size in the circuit by bounding some of its parameter by recovering the determinism lost after existential projection. The transformation is akin to the well-known determinization of automata and could actually be applied straightforwardly to OBDDs.

Apply. We conclude this tour of transformations with the apply transformation. So far, the transformations were transforming a given Boolean function. In the case of apply, the goal

is to combine Boolean functions together. More precisely, given Boolean functions f_1, f_2 on variables X and g on variables $\{x_1, x_2\}$, we let $APPLY(f_1, f_2, g)$ be the Boolean function on variables X defined as: $APPLY(f_1, f_2, g)(\tau) = g(f_1(\tau), f_2(\tau))$ for every $\tau \in 2^X$.

We say that a class of representation $\mathcal{C}$ supports the apply operator if and only if for every Boolean function g on $\{x_1, x_2\}$ and $C_1, C_2 \in \mathcal{C}$, there exists a circuit C_3 computing $APPLY(C_1, C_2, g)$ of size poly$(|C_1| + |C_2|)$. In some cases, a class of circuits may support the apply operator only for a handful of functions g.

In practice, we are particularly interested in the case where $g(x_1, x_2) = x_1 \wedge x_2$, in which case, $APPLY(f_1, f_2, g) = f_1 \wedge f_2$, a transformation that we call *conjunction*. Indeed, if a class supports conjunction and if we are able to build a circuit for every clause, then we have an algorithm to build a representation in this class from CNF formulas. We compile each clause of the input CNF formula F into a circuit and combine these circuits using conjunction. The size of the circuit will likely blow up but using clever heuristics or minimization algorithms (as for OBDD), we can sometimes manage to keep the circuit small enough during the computation. This approach is known as "bottom-up" compilation which we study in more detail in Chapter 2.

Structuredness is essential for the apply transformation to be tractable and it is actually the main motivation for it. Indeed, if we are given two OBDDs C_1, C_2 *respecting the same variable order*, then we can build an OBDD computing $C_1 \wedge C_2$ of size at most $|C_1| \cdot |C_2|$. The construction is a product construction: we introduce a gate $g_{a,b}$ for every pair (a, b) of decision-gates and inductively ensure that $g_{a,b}$ computes $f_a \wedge f_b$. Since OBDDs can also be negated, we can implement the apply transformation for any function $g(x_1, x_2)$ since for example, $x_1 \vee x_2 = \neg(\neg x_1 \wedge \neg x_2)$. The transformation is, however, only possible if C_1 and C_2 respect the same variable order. Otherwise, we can build two OBDD C_1, C_2 using different variables order and such that any OBDD computing $C_1 \wedge C_2$ has size exponential in $|C_1| + |C_2|$. One such function is $ROW(M_n) \wedge COL(M_n)$. Both functions can be implemented by OBDD of size $O(n^2)$ but using different orders.

A similar result holds for structured DNNF circuits: if two structured DNNF circuits C_1, C_2 respect the same vtree, then we can compute a structured DNNF circuit of size $O(|C_1| + |C_2|)$ which computes $C_1 \wedge C_2$. The transformation preserves determinism, in the sense that if both C_1 and C_2 are deterministic, then the resulting circuit is also deterministic. Now if C_1 and C_2 have distinct vtrees, the transformation may induce an exponential blow-up, see [Cap16, Chapter 6] for a separation of structured DNNF circuits and FBDDs that can be seen as an example of this phenomenon. Another notable difference between structured d-DNNF circuits and OBDDs is that, since d-DNNF circuits cannot be efficiently negated, then the apply transformation cannot work for arbitrary g. Indeed, we could take $g(x_1, x_2) = \neg x_1$. Applying g results in negating the first DNNF circuit, which is not possible in polynomial blow-up, see [Vin24]. In Chapter 2, we present a restriction of structured d-DNNF circuits where we can both negate and apply conjunction, leading to a tractable apply for any function g.

1.5 Conclusion

This section has presented the main classes of representation that have been used in knowledge compilation and the ones that will be useful for reading this manuscript. Exhaustiveness not being the goal of this section, we have not included many interesting data structures and algorithms that have been studied. For example, Sentential Decision Diagrams (SDD) are really useful data structures, both theoretically and in practice, which nicely generalize OBDD to tree-like decomposition [Dar11]. We present this important data structure in more detail in Chapter 2 because this section is dedicated to another generalization of OBDD. Other ideas have been considered, for example, by using affine functions to represent Boolean functions with Affine Decision Trees [Kor+13], using symmetries [Bar+14], algebraic decision diagrams [DPV20], etc.

We emphasize in this conclusion the fact that the notions of decomposability and determinism (or unambiguity) are really the ones enabling most tractability results, because they allow us to design efficient dynamic programming algorithms that reduce a Boolean function into smaller ones. We argue through this document that if these notions have been isolated and formalized in the domain of knowledge compilation, they are so natural that they actually appear under different disguises and different names in many other parts of computer science: in proof complexity (see Chapter 3), in database theory (see Chapter 4), in optimization theory (see Chapter 5) and many others. One aspect of this manuscript is to review the few places where I came across similar notions and could use this connection to get new insights.

Chapter 2

Applications to Propositional Logic

An obvious application of data structures for efficiently representing Boolean functions is in the field of propositional logic. In this setting, Boolean functions are naturally represented as formulas built from a set of variables called *propositions* glued together using connectives with a precise semantics. This semantics maps a formula to a Boolean function, corresponding to its models. Hence, from a computational point of view, one can see propositional logic as a logical representation of Boolean functions. Unfortunately, this representation is often not tractable, in the sense that even the SAT problem of deciding whether the Boolean function represented in Conjunctive Normal Form (CNF) has a satisfying assignment is NP-complete. A successful line of research has been to identify classes of formulas where SAT and other problems are tractable, that is, classes for which every formula admits a polynomial time algorithm. One way of achieving this is by restricting how the variables and the clauses interact in a CNF formula. One of the earliest results in this direction has been to establish the tractability of SAT for instances having bounded treewidth. The earliest reference for this fact appears to be in a paper by Dantsin from 1979 [Dan79], though it is not specifically stated with the treewidth terminology, later refined by Razborov and Alekhnovich [AR11], where the result is expressed in terms of the equivalent branch-width measure. These results have then been generalized in two main complementary directions: by either generalizing the class of tractable formulas, or by showing that even harder problems than SAT are tractable. For the first direction, more general definitions of treewidth such as incidence treewidth [Sze04] or consensus treewidth [GS17] have been considered, or alternative graph and hypergraph parameters such as β-acyclicity [OPS13] or clique-width [FMR08]. For the second direction, the counting version #SAT of SAT has first been observed to be tractable on bounded treewidth instances by Sang, Bacchus, Beame, Kautz, and Pitassi in [San+04] and later generalized to the more general case of incidence treewidth by Samer and Szeider in [SS10]. Other generalizations to broader graph classes have followed such as modular treewidth [PSS13], clique width [SS13], MIM-width [Vat12], see [Cap16, Chapter 2] for a survey. One contribution here has been to show that most model counting algorithms could be seen as compilation algorithms into "small" d-DNNF circuits. It is implicit in Darwiche's early contribution [Dar04] and explicit in a collaboration with Pipatsrisawat [PD10] for primal treewidth. The case for incidence treewidth has been formally proven along more general results in [Bov+15].

There are two main ways of leveraging the structure of a CNF formula for efficient compilation that can be found in the literature. The first one is based on an algorithm which is a generalization of DPLL known as *exhaustive DPLL*. It was first introduced as an algorithm for #SAT in [San+04]. It has been observed that the trace of the exhaustive DPLL algorithm is a decision DNNF circuit [HD05] and hence, one can use it as a compiler from CNF formulas to decision DNNF circuits. By carefully choosing the order in which variables are set, one can prove that the circuit built this way has polynomial size. This has been observed when the primal graph of the formula has bounded treewidth [San+04] and also when the underlying hypergraph of the formula is β-acyclic [Cap17]. We will cover a generalization of this approach in detail in Chapter 4. The other approach uses dynamic programming algorithm which usually follows a tree structure which inductively decomposes the formula into smaller subformulas. This approach has been dominant in structure-based algorithm for solving #SAT [PSS13; SS13; SS10; STV14] and has been generalized to knowledge compilation in [Bov+15]. This connection and its consequences have been the main topic of my PhD thesis [Cap16] and we refer interested readers to it for more details.

In this chapter, we revisit the dynamic-programming-based approach for compiling structured CNF formulas into deterministic DNNF circuits and offer a novel, more modular way of understanding this result. To this end, we use an approach known in knowledge compilation as a *bottom-up compilation*. The idea is to build the circuit by incorporating clauses one after the other. We start from a data structure D representing the constant Boolean function $\top$ and iteratively update $D \leftarrow D \wedge C$ for every clause C of the CNF formula. This supposes that one has a way of computing a data structure for $D \wedge C$ from D and C, a special case of an operation known as *the apply operation*, which some data structures studied in the literature efficiently support. Moreover, in order to avoid exponential blow-up, we also need to keep the circuit as small as possible at each step. This has previously been made possible by using a minimization operator that minimizes a circuit without changing the computed Boolean function, for example, when compiling OBDDs [Bry92]. In this chapter, we hence introduce a new data structure that generalizes OBDD, in the sense that it is more succinct, while enjoying tractable apply operation and a minimization procedure. We also show that the minimal circuit obtained this way is minimal, meaning that if two circuits compute the same function, then they will have the same minimal circuit. We apply this bottom-up algorithm to show that compilation of bounded treewidth CNF formulas can be recovered as bottom-up compilation on this data structure, avoiding the need for specially-crafted dynamic programming. We also generalize the result to the case of bounded treewidth Boolean circuits. Finally, we also show how to leverage this result to solve the more general problem of QBF on structured instances, revisiting the result of [CM19], itself a generalization of a seminal result by Chen [Che04].

Organization of this chapter. Section 2.1 introduces the new data structure, known as deterministic tree decision diagrams and studies them with a knowledge compilation approach, by explaining the tractable tasks and transformations. In particular, Section 2.1.4 covers the minimization algorithm and introduce a semantic measure on Boolean functions characterizing the size of their smallest TDD. Section 2.2 explains how TDD can be used to perform bottom-up compilation and why this algorithm builds a small circuit on instances having

small treewidth with respect to the chosen vtree respected by the TDD). Finally, Section 2.3 shows how determinism can be recovered from non-deterministic TDDs at the cost of an exponential blow-up in their size. This observation is then applied to universally or existentially project blocks of variables in a TDD and thereby derives from this the tractability of bounded treewidth QBF and generalizations of it.

Personal contributions covered in this chapter. This chapter can be understood as a new perspective on our early work in showing that many classes of CNF formulas can be compiled into succinct deterministic DNNF circuits. It reproves, using a bottom-up approach, a result about compiling CNF formulas of bounded incidence treewidth that can be found in [Bov+15], and its generalization to QBF [CM19] and to Boolean circuits [Ama+20].

2.1 Tree Decision Diagrams

We introduce our new data structure, called Tree Decision Diagrams (TDD). While they can be seen as a syntactic restriction of structured DNNF circuits, we choose a definition that is not directly related to them to make their syntactic properties more salient.

2.1.1 Main definitions

Non-deterministic TDD. Let T be a vtree whose leaves are labeled by the set of variables X. A *non-deterministic Tree Decision Diagram (nTDD for short) $C = (N, E)$ respecting T* is defined as follows:

- $N = \biguplus_{t \in T} N_t$ is a set of nodes, partitioned into disjoint sets N_t for each node t of T. The elements of N_t are called *t-nodes*.
- If t is a leaf labeled by x, then every node in N_t is labeled by either x, $\neg x$, 1 or 0.
- E maps every t-node g to its *inputs*: if t is a leaf, then $E(g) = \emptyset$. Otherwise, if t has children t_1, t_2, $E(g) \subseteq N_{t_1} \times N_{t_2}$, that is, $E(g)$ is a set of pairs (g_1, g_2) such that $g_1 \in N_{t_1}$ is a t_1-node and $g_2 \in N_{t_2}$ is a t_2-node.
- There is one distinghuised r-node $\mathsf{out}(C)$ called the output of C, where r is the root of T.

C computes a Boolean function over X defined inductively as follows. Each t-node g computes a Boolean function $f_g \subseteq 2^{X_t}$ where $X_t = \mathsf{var}(T_t)$:

- if t is a leaf, then g computes the Boolean function defined by its label, that is, if g is labeled by 0 then $f_g = \emptyset$, if g is labeled by 1 then $f_g = 2^{\{x\}}$, and if g is labeled by ℓ for $\ell \in \{x, \neg x\}$, $f_g = \{\tau\}$ where τ is such that $\tau(\ell) = 1$.
- if t is an internal node with input t_1, t_2, then $\tau \in f_g$ if and only if there exists $(g_1, g_2) \in E(g)$ such that $\tau|_{X_{t_1}} \in f_{g_1}$ and $\tau|_{X_{t_2}} \in f_{g_2}$. In other words, $f_g = \bigvee_{(g_1,g_2) \in E(g)} (f_{g_1} \wedge f_{g_2})$. If $E(g)$ is empty, we take the convention that $f_g = \emptyset$ is the 0 constant function.

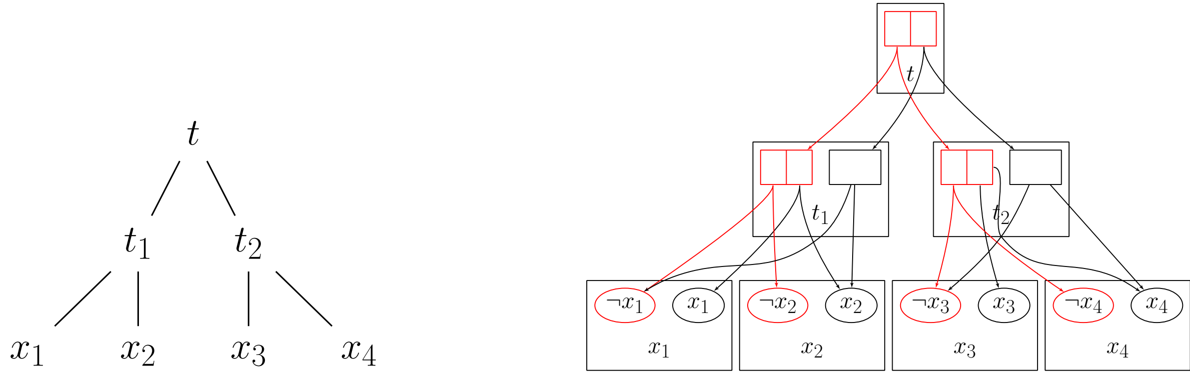


Figure 2.1: A vtree and an nTDD respecting it.

An nTDD C computes the Boolean function f_C defined as $f_{\mathsf{out}(C)}$. An assignment $\tau \in 2^{X_t}$ such that $\tau \models f_g$ for some t-node g is called *a model of* g. For $\tau \in 2^X$ such that $\tau \models C$, we say that τ is *a model of* C.

Example 2.1. Fig. 2.1 shows a vtree T on variables $X = \{x_1, \dots, x_4\}$ and an nTDD C respecting T. We grouped together the set of t-nodes for every node t of T. The assignment defined as $\tau(x) = 0$ for every $x \in X$ is a model of C because it is a model of every node pictured in red.

Another way of characterizing the models of C is via the notion of certificates. Given an assignment $\tau \in 2^X$, a *certificate for* τ *in* C is an nTDD $\mathcal{P}$ formed by picking exactly one t-node $g_t^{\mathcal{P}}$ of C for every node t of T and such that:

- If t is a leaf of T, then $g_t^{\mathcal{P}}$ is either labeled by $\top$ or by a literal ℓ such that $\tau(\ell) = 1$.
- If t is a node of T with children t_1, t_2, then $(g_{t_1}^{\mathcal{P}}, g_{t_2}^{\mathcal{P}}) \in E(g_t^{\mathcal{P}})$.

The red part of Fig. 2.1 represents the certificate for τ, where τ is the assignment setting every variable to 0, which is indeed a model of the circuit C. More generally, a certificate for τ in C is a witness of the fact that τ is a model of C:

Proposition 2.2. *Let T be a vtree over X and C an nTDD respecting T. For every $\tau \in 2^X$, τ is a model of C if and only if there exists a certificate $\mathcal{P}$ for τ in C. In particular, for every node t of T, $\tau|_{X_t}$ satisfies $g_t^{\mathcal{P}}$.*

Proof. First assume τ has a certificate $\mathcal{P}$ in C. By induction, we prove that $\tau|_{X_t}$ is a model of $g_t^{\mathcal{P}}$ for every node t of T. It is true for the leaves of T by definition of certificates. Now if t is a node with children t_1, t_2, then $X_t = X_{t_1} \uplus X_{t_2}$ and $\tau|_{X_t} = \tau|_{X_{t_1}} \times \tau|_{X_{t_2}}$. By induction, $\tau|_{X_{t_i}}$ satisfies $g_{t_i}^{\mathcal{P}}$ for $i \in \{1, 2\}$. By definition of certificates, $(g_{t_1}^{\mathcal{P}}, g_{t_2}^{\mathcal{P}}) \in E(g_t^{\mathcal{P}})$, hence $\tau|_{X_t}$ satisfies $g_t^{\mathcal{P}}$. In particular, τ is a model of $g_r^{\mathcal{P}} = \mathsf{out}(C)$, hence τ is a model of C.

Now let τ be a model of C. We construct a certificate. We let $g_r^{\mathcal{P}} = \mathsf{out}(C)$. Now let t be a node of T and assume that we have constructed $\mathcal{P}$ for every node u on the path from the

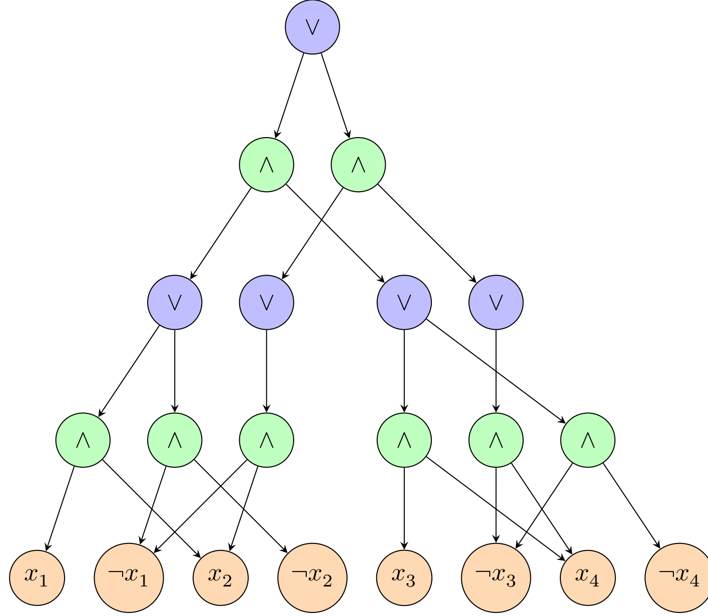


Figure 2.2: A DNNF circuit for the nTDD from Fig. 2.1.

root to t, including t. Let t_1, t_2 be the children of t. Now, since τ is a model of C, there exists $(g_1, g_2) \in E(g_t^{\mathcal{P}})$ such that $\tau|_{X_{t_i}}$ is a model of g_i for $i \in \{1, 2\}$. We let $g_{t_1}^{\mathcal{P}} = g_1$ and $g_{t_2}^{\mathcal{P}} = g_2$. By definition, $(g_{t_1}^{\mathcal{P}}, g_{t_2}^{\mathcal{P}}) \in E(g_t^{\mathcal{P}})$ which corresponds to the definition of certificates.

To show that this process constructs a certificate, it remains to show that it is correct on input nodes. But by definition, if t is a leaf and hence $g_t^{\mathcal{P}}$ is an input, τ is a model of $g_t^{\mathcal{P}}$. Hence, either $g_t^{\mathcal{P}}$ is labeled by 1 or by a literal ℓ such that $\tau(\ell) = 1$, which concludes the proof. □

The *size* $|C|$ of C is defined as $\sum_{n \in N} |E(g)|$. The *width* of C is defined as $\max_{t \in T} |N_t|$. It is straightforward to see from the definition that any nTDD C respecting T can be transformed into a structured DNNF circuit C' in linear time by interpreting every internal t-node g as a $\vee$-gate v_g whose inputs are $\{v_{g_1} \wedge v_{g_2} \mid (g_1, g_2) \in E(g)\}$. If t is a leaf, g is interpreted by an input with the same label in C'. Clearly C' is a DNNF circuit respecting T which computes the same Boolean function as C and has size $\sum_{n \in N} 3|E(g)| \leq 3|C|$ since v_g needs 3 edges to be connected to v_{g_1} and v_{g_2} for every $(g_1, g_2) \in E(g)$. This transformation is mostly a reformulation of nTDDs but the shape of nTDD will make the proofs in this chapter easier to follow.

Theorem 2.3. *For every nTDD C respecting T, there exists a DNNF circuit C' respecting T of size $O(|C|)$ computing the same function as C.*

In Fig. 2.2, we give a DNNF circuit equivalent to the nTDD from Fig. 2.1.

2.1.2 Determinism

We define a syntactic restriction of nTDD which ensures determinism of its underlying DNNF circuit. A *Tree Decision Diagram (TDD)* $C = (N, E)$ is an nTDD satisfying the following property, called *determinism*. For every node t of T:

- If t is a leaf labeled by x, then no two nodes of N_t can be satisfied simultaneously. Syntactically, it means that N_t contains at most one node labeled by x, at most one node labeled by $\neg x$ and at most one node labeled by 1. Moreover, if there is a node labeled by 1, then all other nodes of N_t are labeled by 0.
- Otherwise, for every $g, g' \in N_t$, we have $E(g) \cap E(g') = \emptyset$.

Observe that a TDD of width k has size at most $2|X| \cdot k^2$. Indeed, T has at most $2|X|$ nodes and each t-node can contain at most k^2 pairs.

Contrary to the notation of determinism for DNNF circuit, the determinism of TDD is a syntactic notion. Therefore, it can be checked in polynomial time whether a given nTDD is deterministic. Moreover, it induces a very strong form of determinism, in the following sense:

Theorem 2.4. *Let $C = (N, E)$ be a TDD respecting a vtree T. For every node t of T and any two t-nodes g, g', f_g and $f_{g'}$ have disjoint models.*

Proof. The proof is by induction on T. It is obvious if t is a leaf by assumption. Indeed, either exactly one t-node is labeled by 1 and all the other t-nodes are labeled by 0, in which case the property is clear. Otherwise, we have at most one t-node labeled by x, at most one t-node labeled by $\neg x$ and all others labeled by 0. Again, we have disjoint models.

Now let t be a node of T with children t_1, t_2 and assume the property holds for t_1 and t_2. Let g and g' be two t-nodes. Let τ be a model of g and let $(g_1, g_2) \in E(g)$ be such that $\tau|_{X_{t_1}}$ is a model of g_1 and $\tau|_{X_{t_2}}$ is a model of g_2. Now if τ is also a model of g', there exists $(g'_1, g'_2) \in E(t)$ such that $\tau|_{X_{t_1}}$ is a model of g'_1 and $\tau|_{X_{t_2}}$ is a model of g'_2. By determinism, $(g_1, g_2) \neq (g'_1, g'_2)$. Without loss of generality, assume $g_1 \neq g'_1$. In this case, $\tau|_{X_{t_1}}$ is a model of both g_1 and of g'_1. But this is not possible by induction. Hence τ is not a model of g'. □

A consequence of Theorem 2.4 is that all models of C have unique certificates:

Corollary 2.5. *Let T be a vtree over X and let $C = (N, E)$ be a TDD respecting T. For every model τ of C, there exists a unique certificate $\mathcal{P}_C(\tau)$ for τ in C.*

Proof. We have already observed in Proposition 2.2 that if $\mathcal{P}$ is a certificate for τ in C then $\tau|_{X_t}$ is a model of $g_t^{\mathcal{P}}$. By Theorem 2.4, there is a unique t-node α_t that is satisfied by $\tau|_{X_t}$, hence $\alpha_t = g_t^{\mathcal{P}}$ for every t, that is, $\mathcal{P}$ is unique. □

Interestingly, observe that we can build the certificate for τ in C efficiently in a bottom-up way, or report that τ is not a model for C. To do so, we try to construct the certificate bottom-up: if t is a leaf of T labeled by x, then by determinism, there exists at most one t-node that τ satisfies and we choose it. If no such node exists, we report that τ is not a model of C.

Now assume t is a node of T with children t_1, t_2. Assume we have constructed the unique t_1-node g_1 such that $\tau|_{X_{t_1}}$ is a model of g_1 and the unique t_2-nodes g_2 such that $\tau|_{X_{t_2}}$ is a model of g_2. Either there is a unique t-node g such that $(g_1, g_2) \in E(g)$, in which case we know that g is the t-node from the certificate of τ. Otherwise, no such node exists, in which case, we reject τ.

The exact complexity of this procedure depends on how we exactly represent the t-nodes. If we do it naively, so that each node is a list of pairs, then we may need to go over every t-node to decide whether (g_1, g_2) appears in $E(g)$ for some g. The total time of the procedure would therefore be $O(|C|)$.

Now we can efficiently maintain a data structure for each t-node that allows us to get the only t-node g such that $(g_1, g_2) \in E(g)$ if it exists and fail otherwise. This can, for example, be a hash table mapping pairs (g_1, g_2) to g or a matrix indexed by g_1 and g_2 whose entry at (g_1, g_2) is g. In this case, each check takes constant time (expected average constant time in the case of hash tables) and we can construct the certificate for τ in time $O(|T|) = O(|X|)$.

Lemma 2.6. *Given a TDD C over variables X and $\tau \in 2^X$, we can check that τ satisfies C and construct its certificate in time $O(|X|)$.*

Another consequence of Theorem 2.4 is that the DNNF circuit built from a TDD using Theorem 2.3 is deterministic:

Theorem 2.7. *For every TDD C respecting a vtree T, we can construct a d-DNNF circuit C' respecting T in time $O(|C|)$.*

Proof. Let v_g be one $\vee$-gate from the construction of Theorem 2.3, corresponding to t-node g. It is of the form $\bigvee_{(g_1,g_2)\in E(g)}(v_{g_1} \wedge v_{g_2})$. If v is not deterministic, there exists (g_1, g_2) and (g'_1, g'_2) with $(g_1, g_2) \neq (g'_1, g'_2)$ and τ that is both a model of $(v_{g_1} \wedge v_{g_2})$ and of $(v_{g'_1} \wedge v_{g'_2})$. Assume wlog $g_1 \neq g'_1$. Then τ is a model of g_1 and of g'_1 which contradicts Theorem 2.4. □

In particular, every tractable query for d-DNNF circuits is also tractable for TDD and the algorithm is straightforward since the corresponding d-DNNF circuit has a structure close to the original TDD. We illustrate this for model counting. The counting algorithm for d-DNNF circuit straightforwardly translates directly to the following algorithm. For each t-node g, we inductively compute S_g, the number of models of g over X_t as follows:

- If g is an input labeled by a literal then $S_g = 1$. If it is labeled by 1, then $S_g = 2$. Otherwise $S_g = 0$.
- Otherwise $S_g = \sum_{(g_1,g_2)\in E(g)} S_{g_1} \cdot S_{g_2}$.

Observe that this algorithm returns the number of models of C over X, the underlying set of its vtree. It may be that the circuit does not depend on a given variable x (that is, no input is labeled by a literal on x). If we want the number of models of C on $X \setminus \{x\}$, we can then either divide by 2 or modify the algorithm to set $S_g = 1$ for g a 1-input.

While tractable queries directly transfer from (structured) d-DNNF circuits to TDDs, this is not the case for transformations, as applying a transformation to the TDD seen as a d-DNNF circuit may break the syntactic properties of TDD. In the next section, we explore the tractable transformations for TDD.

2.1.3 Tractable transformations

Constants elimination. Before studying transformations of TDDs, we explain how one can remove constant inputs in TDD. This is necessary to properly explain some transformations such as conditioning or forgetting where some variables are removed from the vtree, which may induce an ambiguity on the semantics of the circuit.

Recall that by convention, a gate g such that $E(g) = \emptyset$ computes the 0-constant Boolean function. We refer to such gates as gates with empty inputs. We explain how one can remove 0 constants and gates with empty inputs.

Theorem 2.8. *Given a TDD C respecting vtree T with at least one model, one can build in linear time a TDD C' respecting T, computing the same function as C and such that C' does not contain any 0-labeled input nor gates with empty inputs. Moreover, the transformation preserves determinism.*

Proof. Let g be a gate that is either a 0-labeled input or a gate with empty inputs. Assume that g is a u-node for some node u of T. If u is the root of C, we simply remove g from the circuit. Indeed, since C has at least one model, g is not the output of C and it is not connected to any other gate, hence we can remove it without changing the function computed by the circuit.

Otherwise, let t be the parent of u in T. We remove g from the circuit and propagate the change upward: for every t-node h, we remove from $E(h)$ any pair containing g. Now for some h, $E(h)$ may become empty, in which case, we remove h from the circuit similarly and propagate the change upward. We can apply this transformation until no 0-input remains nor any gate with empty inputs. Observe that we only remove gates from C, hence the transformation preserves determinism. □

Removing 1-inputs is however more delicate. Indeed, by construction, TDDs have a rigid structure: if one node h has a pair (g, g') as input, with g being a 1-labeled input, then we cannot simply remove g from the pair since it will change the definition of TDD. One way of doing it would be to allow TDD gates to have singletons in their inputs but it will make the definition of determinism slightly more delicate. Hence, we cannot always remove 1-inputs in TDD.

There is a setting where it is possible. Assume that for some variable x, C does not contain any input labeled by x nor by $\neg x$. In this case, we say that C *does not syntactically depend on x*. Now, if ℓ is the leaf of T labeled by x, every ℓ-node of C is labeled by a constant, and, without loss of generality, by Theorem 2.8, we can assume these constants to be 1. These constants can be removed in the circuit by removing x from the vtree and renormalizing it along this new vtree.

Given a vtree T over variables X and $x \in X$, we let $T \setminus x$ be the vtree obtained as follows: we start by removing the leaf labeled by x in T. Now the parent t of this leaf in T has only one child u which is not allowed in a vtree. If t is the root of the circuit, we simply remove t. Otherwise, we remove t and connect u directly to the parent p of t in T. See Fig. 2.3 for an illustration.

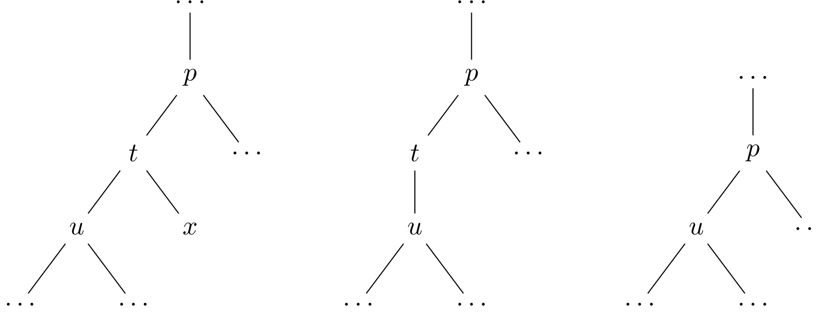


Figure 2.3: Removing x from a vtree T: we first remove the x leaf, then remove the parent of x by plugging its only child u with its parent p.

Theorem 2.9. *Let T be a vtree over X and C a TDD respecting T. Assume there is $x \in X$ such that C does not syntactically depend on x. Then we can build a TDD C' respecting $T \setminus x$ in linear time. Moreover, this transformation preserves determinism.*

Proof. We let l be the leaf of T labeled by x and use the same notation as in Fig. 2.3 for the other nodes of T. First assume that every l-node is labeled by $\bot$. In this case, every t-node h has no model and then, the whole circuit has no model and we can replace it by the TDD computing $\bot$.

Otherwise, there exists exactly one l-node g labeled by $\top$, all the other l-nodes being labeled by $\bot$. Let h be a t-node and let U_h be the set of u-nodes g' such that $(g', g) \in E(h)$ The models of h are exactly the disjoint union of the models of g', for $g' \in U_h$. For every t-node h, we introduce a new u-node g_h whose inputs are $\bigcup_{v \in U_h} E(v)$ and remove every other u-node. It is clear that g_h and h have the same models over $X \setminus \{x\}$.

Now if t is the root of T, the output of C is some t-node h. We remove every t-node from C and choose g_h as the new output. We hence have a new TDD over $T \setminus x$ which computes the same function as C but over $X \setminus \{x\}$.

If t is not the root of T, let p be its parent in T. Let v be a p-node of C. Its inputs are of the form (h, b) with h a t-node. We replace each such input of v by (g_h, b). It does not change the function computed by v but now g_h is a u-node and we have a new TDD over $T \setminus x$ which computes the same function as C but over $X \setminus \{x\}$.

The transformation preserves determinism because if C is deterministic, then $U_h \cap U_{h'} = \emptyset$ for distinct t-nodes h, h'. Hence, $E(g_h) \cap E(g_{h'}) = \emptyset$. □

Conditioning. Conditioning TDDs works similarly as conditioning on DNNF circuits. Let C be a TDD respecting some vtree T over variables X. Let $x \in X$ be a variable and $b \in \{0, 1\}$. Let $\ell = x$ if $b = 1$ and $\ell = \neg x$ otherwise. We let C' be the circuit obtained by simply relabeling every input labeled by ℓ with 1 and every input labeled by $\neg\ell$ with 0. This new circuit almost computes C conditioned by $\langle x/b \rangle$, but not exactly. Indeed, C' is still defined over X. Its models are hence $\{\tau \times \langle x/b' \rangle \mid b' \in \{0, 1\}$ and $\tau \times \langle x/b \rangle \in f_C\}$ while what we would like to

have is a circuit whose models are $\{\tau \in 2^{X \setminus \{x\}} \mid \tau \times \langle x/b \rangle \in f_C\}$. We get such a circuit by observing that C' does not syntactically depend on x. Hence, we can apply Theorem 2.9 to get a TDD structured over $T \setminus x$ whose models are exactly $\{\tau \in 2^{X \setminus \{x\}} \mid \tau \times \langle x/b \rangle \in f_C\}$.

The transformation does not affect the size nor the determinism of the TDD. Indeed, assume that C is deterministic. Since the structure of C' is the same as the structure of C, we still have that for every pair g, g' of internal t-nodes, $E(g) \cap E(g') = \emptyset$. For t a leaf, observe that either there is a t-node labeled by 1 in C, in which case, we have nothing to change since by determinism, no input is labeled by ℓ nor $\neg\ell$. Otherwise, we introduce at most one 1-input and there is no t-node labeled by a literal anymore so the circuit is still deterministic. Finally, the transformation from Theorem 2.9 also preserves determinism. We have proven:

Theorem 2.10. *Given a TDD C of width k, a vtree T over variables X, $x \in X$ and $b \in \{0, 1\}$, one can build in time $O(|C|)$ a TDD C' with size at most $|C|$ and width at most k computing $f_C[x/b]$. Moreover, C' respects $T \setminus x$ and the transformation preserves determinism.*

Negation. Deterministic TDDs can be negated in polynomial time. It is a great advantage over structured deterministic DNNF circuits, making them more akin to OBDD in this respect. Intuitively, we can transform a TDD such that we can add a new t-node that accepts every assignment of X_t that is not accepted by another t-node. If we have this property at the root, then we can merge every t-node that is not $\mathsf{out}(C)$ and set it as the new output to compute $\neg C$.

We say that a TDD $C = (N, E)$ respecting a vtree T is *t-full* if $f_t := \bigvee_{g \in N_t} f_g$ is the 1-constant Boolean function (every assignment $\tau \in 2^{X_t}$ is a model of f_t). It is *full* if it is t-full for every node t of T.

Proposition 2.11. *Let T be a vtree over X. For every TDD $C = (N, E)$ of width k and respecting vtree T, one can build in time $O(k^2|X|)$ a full TDD C' of width $k+1$ respecting T, computing the same function and of size at most $|C| + 2|X| \cdot k^2$.*

Proof. The proof is by induction on T: for every node t, we transform C in a bottom-up way to make it u-full for every node u below t in T.

If t is a leaf, then either there is a t-node labeled by 1 and it is obviously t-full. Otherwise, for every literal ℓ over x such that no t-node is labeled by ℓ, we add a t-node labeled by ℓ. In the end, we have a t-node labeled by x and one by $\neg x$, hence the circuit is now t-full.

Now let t be a node with children t_1, t_2 and assume that the circuit is u-full for any node below t but t itself. We transform the circuit into a t-full circuit. To this end, we add a new t-node g to C that is connected to every pair of $N_{t_1} \times N_{t_2}$ that is not plugged into any other t-node. In other words, we let $E(g) = (N_{t_1} \times N_{t_2}) \setminus \bigcup_{g' \in N_t} E(g')$. We now claim that the circuit is t-full. Indeed, let $\tau \in 2^{X_t}$ and let $\tau_1 = \tau|_{X_{t_1}}$, $\tau_2 = \tau|_{X_{t_2}}$. Since the circuit is t_1-full and t_2-full, we let $(g_1, g_2) \in N_{t_1} \times N_{t_2}$ such that τ_1 is a model of g_1 and τ_2 is a model of g_2. Let $g' \in N_t$ be the t-node such that $(g_1, g_2) \in E(g')$ which exists since every pair of $N_{t_1} \times N_{t_2}$ is now covered. We have that τ is a model of g'. Hence the circuit is now t-full.

Observe that we have added at most one new gate per node t of T in the circuit hence we have increased the width of C by at most 1. The number of edges introduced for each node is

at most k^2 and T has at most $2|X|$ nodes, hence the total size of the circuit is now at most $|C| + 2|X| \cdot k^2$.

To build the circuit efficiently, one can do it as follows: for each t-node, create a $N_{t_1} \times N_{t_2}$ indexed matrix M initialized to 0. Loop over every t-node v and their inputs. Upon seeing $(g_1, g_2) \in E(v)$, update $M[g_1, g_2]$ to 1. The new gates to be added are exactly the entries of M whose value is 0 after having looped over every input of every t-node. The size of M is at most k^2 and we loop over at most k^2 values to build it, hence we need $O(k^2)$ time to treat a t-node. Since there are at most $2|X|$ nodes in T, we need a total time $O(|X|k^2)$. □

Full TDDs can easily be negated:

Proposition 2.12. *Let T be a vtree over X. Given a full TDD $C = (N, E)$ of width k respecting T, one can build in time $O(k)$ a full TDD C' computing $\neg C$ of width at most k and size at most $|C|$.*

Proof. Let r be the root of T. Since C is full, it is r-full, that is, $\bigvee_{g \in N_r} f_g$ is the 1-constant Boolean function over X. Moreover, by Theorem 2.4, this disjunction is deterministic. Hence, $\neg f_C = \bigvee_{g \in N_r, g \neq \mathsf{out}(C)} f_g$. We build C' by merging every r-node g which is not $\mathsf{out}(C)$: that is, we remove every r-node that is not the output and add a new r-node g' such that $E(g') = \bigcup_{g \in N_r, g \neq \mathsf{out}(C)} E(g)$. Obviously, $f_{g'} = \neg f_C$. We hence set $\mathsf{out}(C') = g'$ and C' computes $\neg f_C$. Moreover C' is still full since the previous output gate is still in the circuit and neither the width nor the size have increased. We only need $O(k)$ to perform this transformation as we only need to loop over every r-node but the output and there is at most $k - 1$ such node. □

Applying Proposition 2.11 and Proposition 2.12 successively gives:

Theorem 2.13. *Let T be a vtree over X. Given a TDD C of width k respecting T, there exists a TDD C' of width at most $k + 1$ and of size at most $|C| + 2|X| \cdot k^2$ computing $\neg C$. Moreover, C' can be constructed in time $O(k^2|X|)$.*

Observe that for full TDDs, we can assume that we only have two r-nodes where r is the root of T: one that is the output of C and computes f_C, and one which computes the rest, hence $\neg f_C$. In other words, it resembles an OBDD where we have two sinks, one computing f and the other $\neg f$. We will see in the next section that this connection can actually be made formal: an OBDD can be seen as a TDD if rooted at its sinks.

Apply. We now prove that we can efficiently implement the apply operator over TDDs, and this transformation preserves determinism. The transformation for conjunction is mostly the same product construction as the one for OBDDs or structured DNNF circuits. For disjunction, we simply express $C \vee C'$ as $\neg(\neg C \wedge \neg C')$.

Proposition 2.14. *Let T be a vtree over variables X. Let C_1 and C_2 be two TDDs respecting T of width k and k' respectively. One can build a TDD C respecting T and computing $C_1 \wedge C_2$ in time $O(|C_1| \cdot |C_2|)$. The size of C is at most $|C_1| \cdot |C_2|$ and its width is at most $k \cdot k'$. Moreover, if C_1 and C_2 are deterministic then C is also deterministic.*

Proof. We construct C such that for every node t of T, for every t-node g_1 of C_1 and g_2 of C_2, C has a t-node $v_{\{g_1,g_2\}}$ which computes $f_{g_1} \wedge f_{g_2}$. We proceed by induction on T.

If T only contains a leaf t labeled by x, then for each pair g_1, g_2 of t-nodes in C_1 and C_2, we introduce a gate v_{g_1,g_2} labeled by $e_1 \wedge e_2$ where e_1 is the label of g_1 and e_2 is the label of g_2. The label of v_{g_1,g_2} is in $\{x, \neg x, 0, 1\}$ and v_{g_1,g_2} clearly computes $f_{g_1} \wedge f_{g_2}$. Observe that if C_1 and C_2 are both deterministic, then the resulting TDD is also deterministic. Indeed, assume v_{g_1,g_2} is labeled by 1, then it means that both g_1 and g_2 are labeled by 1 and then every other t-node of C_1 and C_2 are labeled by 0. Therefore, for every pair g_1', g_2' distinct from g_1, g_2, $v_{g_1',g_2'}$ is labeled by 0. Otherwise, assume v_{g_1,g_2} is labeled by a literal ℓ. Then either both g_1, g_2 are labeled by ℓ, in which case v_{g_1,g_2} is the only gate of C labeled by ℓ because C_1 and C_2 do not have any gate labeled by 1. Or g_1 is labeled by 1 and g_2 by ℓ, in which case again, v_{g_1,g_2} is the only gate of C labeled by ℓ because C_1 does not have any gate labeled by ℓ nor $\neg\ell$. The last case where g_1 is labeled by ℓ and g_2 by 1 is symmetrical.

Now let t be a node of T with children t_1, t_2. Assume that we have constructed C up to t_1 and t_2. We construct the t-nodes of C as follows: for every pair g_1, g_2 of t-nodes of C_1 and C_2 respectively, we create a t-node v_{g_1,g_2} and define $E(v_{g_1,g_2}) = \{(v_{a_1,b_1}, v_{a_2,b_2}) \mid (a_1, a_2) \in E(g_1), (b_1, b_2) \in E(g_2)\}$.

By induction, v_{g_1,g_2} computes

$$\begin{aligned}
&\bigvee_{(a_1,a_2)\in E(g_1),(b_1,b_2)\in E(g_2)} (f_{a_1} \wedge f_{b_1}) \wedge (f_{a_2} \wedge f_{b_2}) \\
&\qquad = \bigvee_{(a_1,a_2)\in E(g_1),(b_1,b_2)\in E(g_2)} (f_{a_1} \wedge f_{a_2}) \wedge (f_{b_1} \wedge f_{b_2}) \\
&\qquad = \bigvee_{(a_1,a_2)\in E(g_1)} (f_{a_1} \wedge f_{a_2}) \wedge \bigvee_{(b_1,b_2)\in E(g_2)} (f_{b_1} \wedge f_{b_2}) \\
&\qquad = f_{g_1} \wedge f_{g_2}.
\end{aligned}$$

It remains to show that C is deterministic. Let v_{a_1,b_1} be a t_1-node of C and v_{a_2,b_2} be a t_2-node of C. We claim that there is at most one t-node v_{g_1,g_2} in C such that $(v_{a_1,b_1}, v_{a_2,b_2}) \in E(v_{g_1,g_2})$. Indeed, there is at most one t-node g_1 of C_1 such that $(a_1, a_2) \in E(g_1)$ and at most one t-node g_2 of C_2 such that $(b_1, b_2) \in E(g_2)$. Hence $(v_{a_1,b_1}, v_{a_2,b_2})$ only appears in $E(v_{g_1,g_2})$, that is, C is deterministic.

By construction, it is clear that the width of C is $k \cdot k'$ and its size is at most $|C_1| \cdot |C_2|$. Constructing the circuit takes time $O(|C_1| \cdot |C_2|)$ since we loop once over every pair of gates in C_1, C_2. □

Combining Proposition 2.14 and Theorem 2.13, we can also construct a circuit computing $f_C \vee f_{C'}$ as $\neg(\neg C \wedge \neg C')$ when both C and C' are deterministic. In other words:

Theorem 2.15. *Let T be a vtree over variables X. Let C_1 and C_2 be two TDDs respecting T of width k and k' respectively. Let $f(x_1, x_2)$ be a Boolean function over $\{x_1, x_2\}$. There exists a TDD C respecting T and computing $f(C_1, C_2)$. Moreover, the size of C is at most $O(|C_1| \cdot |C_2|)$, its width is at most $(k + 1) \cdot (k' + 1)$ and it can be built in time $O(|C_1| \cdot |C_2|)$.*

Observe that one corollary of TDDs supporting the apply transformation is the fact that TDDs form a complete language, that is, it can represent every Boolean function over a given set of variables X and with any vtree over X.

Proposition 2.16. *For every Boolean function f over X and vtree T over X, there exists a TDD computing f.*

Proof. Let T be a vtree over X. It is enough to observe that for any $\tau \in 2^X$, there exists a TDD C_τ respecting T that accepts only τ. This is straightforward by induction over T. Now, given a Boolean function f with models $\tau_1, \dots, \tau_p$, we get a TDD C_p for f by constructing it iteratively as follows: $C_1 = C_{\tau_1}$ and $C_i = APPLY(\vee, C_{i-1}, C_{\tau_i})$. □

Proposition 2.16 may seem obvious, but note that enforcing structure in decision-DNNF circuits leads to incomplete language. Indeed, if we fix a balanced vtree T over variables $x_1, \dots, x_n$, then there is no decision-DNNF circuit respecting T and computing $x_1 \vee \cdots \vee x_n$.

Forgetting. We conclude this section with one last transformation, which is notoriously easy for DNNF circuits but not for deterministic DNNF circuits. We have a similar property with TDDs. Given a Boolean function f over X and $Y \subseteq X$, we denote by $\exists Y.f$ the Boolean function whose models are $\{\tau|_{X \setminus Y} \mid \tau \models f\}$. This operation is often called *forgetting* or *existential projection.* Forgetting is easy in TDD but the transformation does not preserve determinism:

Theorem 2.17. *Let T be a vtree over X and $Y \subseteq X$. Given a TDD C respecting T, one can construct in time $O(|C|)$ a TDD C' respecting $T \setminus Y$ computing $\exists Y.f_C$. Moreover, C' has smaller width and size than C.*

Proof. The same transformation as for DNNF circuits works: we construct C' by relabeling in C every input gate labeled by either y or $\neg y$ for some $y \in Y$ with constant 1 and removing the introduced constants as explained in Theorem 2.9. It is easy to see by induction that C' computes $\exists Y.f_C$ with no increase in width nor size. □

The construction from Theorem 2.17 does not preserve determinism however because we could have more than one 1-input gate in the inputs corresponding to y. We will however see in Section 2.3 that TDD can be made deterministic with only an exponential increase in the width, allowing us to prove tractability results on quantified Boolean formulas.

2.1.4 Minimization and canonicity

This section is dedicated to one of the most important aspects of TDDs: they can be minimized in polynomial time and the minimal circuit is unique up to isomorphism, a property that is usually referred to as canonicity. The minimization algorithm is akin to the minimization of OBDD: we identify in the circuits pairs of gates that we call twins and which can be merged without changing the function computed by the circuit. We repeat this merging procedure until no twins can be found. The circuit we obtain is now the minimal TDD computing the same Boolean function. Moreover, we show that we are able to precisely understand what each

gate in the circuit computes. This knowledge will be used in Section 2.2 to recover compilation of bounded treewidth CNF formulas and circuits in a modular way.

In this section, we fix a vtree T over variables X and a TDD C respecting T. Let t_1 be a node of T that is not the root of T, let t be its parent and t_2 its sibling. For a t_1-node g_1 and a t-node g, we define the *siblings of g_1 with respect to g*, denoted by $\mathsf{sib}(g_1, g)$, to be $\{g_2 \mid (g_1, g_2) \in E(g)\}$, that is, the set of t_2-nodes that appears together with g_1 in the set $E(g)$.

Two t_1-nodes g_1, g_1' are said to be *twins* if the following holds: for every t-node g, we have $\mathsf{sib}(g_1, g) = \mathsf{sib}(g_1', g)$. If g_1 and g_1' are twins, we define the *twins contraction of* g_1, g_1' to be the operation where we replace g_1, g_1' in C with a new gate $v_{g_1,g_1'}$ such that: $E(v_{g_1,g_1'}) = E(g_1) \cup E(g_1')$. Moreover, for any t-node g, we replace any pair of the form (g_1, g_2) in $E(g)$ by $(v_{g_1,g_1'}, g_2)$ and remove every pair of the form (g_1', g_2). Observe that since g_1 and g_1' are twins, $(g_1, g_2) \in E(g)$ if and only if $(g_1', g_2) \in E(g)$ by definition. We have:

Lemma 2.18. *After contracting two twins, the function computed by the circuit is not changed. Moreover, the circuit is still deterministic.*

Proof. Let $C' = (N', E')$ be the circuit obtained after contraction. It is clear by definition that $v_{g_1,g_1'}$ in C' computes $f_{g_1} \vee f_{g_1'}$. Since the twin operation does not change the number of t-nodes (only their input), there is a natural one-to-one correspondence between the t-nodes of C and C'. We claim that they compute the same function. Indeed, every pair (g_1, g_2) in $E(g)$ appears together with $(g_1', g_2) \in E(g)$ since g_1 and g_1' are twins. This part of the function computes $(f_{g_1} \wedge f_{g_2}) \vee (f_{g_1'} \wedge f_{g_2})$ which is equal to $(f_{g_1} \vee f_{g_1'}) \wedge f_{g_2}$, that is, the function computed by $v_{g_1,g_1'}$. Hence, replacing pairs (g_1, g_2) and (g_1', g_2) in $E(g)$ with $(v_{g_1,g_1'}, g_2)$ does not change the function g computes. Determinism is preserved since we did not add duplicates. □

We transform C as follows: first, if r is the root of T, we remove every r-node but $\mathsf{out}(C)$. We also remove every node that is not connected to the output of the circuit by a directed path. More precisely, for every t with parent t' in T, we remove every t-node g that is not in any pair of $\bigcup_{g' \in N_{t'}} E(g')$. Clearly, this does not change the function computed by C since these gates are not used in any certificate. Once no such node exists anymore, we apply twins contraction to C until no twins exists anymore. We denote by $m(C)$ the TDD obtained after applying this transformation. This procedure terminates since each twin contraction reduces the total number of nodes in the TDD by at least one, hence it cannot be applied more than the initial number of nodes in C. It even runs efficiently since both identifying and contracting twins can be done in polynomial time in the width of the circuit. The resulting circuit $m(C)$ has width and size smaller than the width and size of C.

To prove the minimality and the canonicity of $m(C)$, we need to understand what are the gates of $m(C)$ with respect to f_C. To this end, we need the notion of subfunctions and factors from [BS17]. Let f be a Boolean function over X and $Y \subseteq X$. A *subfunction (or cofactor) of f induced by Y*[1], or Y-subfunction for short, is a Boolean function over $X \setminus Y$ of the form $f[\tau]$ for some $\tau \in 2^Y$. Observe that f has at most $2^{|Y|} \leq 2^{|X|}$ distinct Y-subfunctions but it could

[1] [BS17] mainly uses the name cofactor also calls them subfunctions, which also appears in the literature over OBDD. We prefer the term subfunction here.

have less. Indeed, two distinct assignments $\tau_1, \tau_2 \in 2^Y$ could be such that $f[\tau_1]$ and $f[\tau_2]$ have the same models over $2^{X \setminus Y}$, hence define the same subfunction. A subfunction is said to be *non-trivial* if it has at least one model. Given a vtree T and a node t of T, we will mostly be interested in the X_t-subfunction of f.

Observe that given $Y \subseteq X$ we can define a relation $\equiv_f^Y$ on 2^Y as $\tau \equiv_f^Y \tau'$ if and only if $f[\tau] = f[\tau']$. This is clearly an equivalence relation, hence partitioning 2^Y into classes $f_1, \dots, f_r$, which could be seen as Boolean functions over Y. Such a function over Y is called a *Y-factor*. In other words, a Y-factor is a Boolean function g over Y such that there exists a Y-factor f' of f and the models of g are exactly the assignments $\tau \in 2^Y$ such that $f[\tau] = f'$.

Example 2.19. Consider the function $EVEN$ over X whose models are the set of assignments $\tau \in 2^X$ such that $|\tau^{-1}(1)|$ is even. For any vtree T over X and any node t of T, $EVEN$ has at most two distinct X_t-subfunctions. Indeed, if $\tau \in 2^{X_t}$ is such that $|\tau^{-1}(1)|$ is even then the models of $f[\tau]$ are the assignments $\sigma \in 2^{X \setminus X_t}$ such that $|\sigma^{-1}(1)|$ is even. Now if $\tau \in 2^{X_t}$ is such that $|\tau^{-1}(1)|$ is odd then the models of $f[\tau]$ are the assignments $\sigma \in 2^{X \setminus X_t}$ such that $|\sigma^{-1}(1)|$ is odd. Hence $EVEN$ has two X_t-factors: the first one accepts assignments of 2^{X_t} with an even number of variables set to 1 and the other accepts those with an odd number of variables set to 1.

Now, we observe that a t-node naturally defines an X_t-subfunction in the following sense:

Lemma 2.20. *For a node t of T and g a t-node of C, let τ_1, τ_2 be two models of g. We have that $f_C[\tau_1]$ and $f_C[\tau_2]$ define the same X_t-subfunction, denoted by sub_g. Moreover, for every model τ of C such that g is in the certificate of τ, $\tau|_{X \setminus X_t}$ is a model of sub_g.*

Proof. Let σ be a model of $f_C[\tau_1]$. In particular, $\sigma \times \tau_1$ is a model of C. The t-node in the unique certificate of $\sigma \times \tau_1$ is g, since τ_1 is a model of g. Now, we can swap the part of the certificate below t with the certificate of τ_2 below t. This way, we get a certificate for $\sigma \times \tau_2$, hence $\sigma \times \tau_2$ is a model of f_C, and then σ is a model of $f_C[\tau_2]$. Symmetrically, we have that every model σ of $f_C[\tau_2]$ is a model of $f_C[\tau_1]$. Hence $f_C[\tau_1]$ and $f_C[\tau_2]$ have the same models and define the same X_t-subfunction. Finally, observe that for every model τ of C whose certificate contains g, we have that $\tau|_{X \setminus X_t}$ is a model of $f_C[\tau|_{X_t}]$ and by the above, $f_C[\tau|_{X_t}]$ is sub_g. □

Lemma 2.20 hence uniquely defines, for every t-node g of C, an X_t-subfunction sub_g of f_C as $f_C[\tau]$ for some model τ of g. Observe that it also gives a lower bound on the number of t-nodes that any circuit representing a Boolean function f must have: the number of t-nodes in C must be at least the number of non-trivial X_t-subfunctions of f_C. Indeed, if $\tau_1, \tau_2 \in 2^{X_t}$ are such that $f_C[\tau_1]$ and $f_C[\tau_2]$ define two distinct X_t-subfunctions, then they cannot be models of the same t-node. Now, if $f_C[\tau_1]$ is non-trivial, then there must be a t-node g_1 such that τ_1 is a model of g_1, since there exists at least one $\sigma \in 2^{X \setminus X_t}$ such that $\sigma \times \tau_1$ is a model of C. Similarly, if $f_C[\tau_2]$ is non-trivial, there is a t-node g_2 such that τ_2 is a model of g_2. Hence we have at least one gate per non-trivial X_t-subfunctions of f_C. We have proven:

Theorem 2.21. *Given a Boolean function f over variables X and T a vtree over X, the smallest TDD computing f has at least S_t t-nodes for every node t of T where S_t is the number of non-trivial X_t-subfunctions of f.*

We now show that $m(C)$ actually matches the lower bound from Theorem 2.21.

Theorem 2.22. *Let T be a vtree over X with root r and C a TDD. Let $m(C)$ be the circuit obtained after having removed every r-node but $\mathsf{out}(C)$, removed every dangling node and contracting all twins. Then $m(C)$ has exactly S_t t-nodes, where S_t is the number of non-trivial X_t-subfunctions of f_C.*

Proof. Assume $m(C)$ does not match the lower bound. It means by the pigeonhole principle that there is a node t of T and two distinct t-nodes g, g' such that $\mathsf{sub}_g = \mathsf{sub}_{g'}$ and sub_g is not trivial. We can assume that t is not the root r of T since $m(C)$ has only one r-node. Moreover, we can assume that for every ancestor u of t in T, there is exactly one u-node per X_u-subfunction. Indeed, if this is not the case, then we could have picked u instead of t.

To summarize, there is a node t_1, with parent t and sibling t_2, and two t_1-node g, g' such that $\mathsf{sub}_g = \mathsf{sub}_{g'}$. Moreover, there is exactly S_t t-nodes, one for each X_t-subfunction of f_C. We claim that in this case, g and g' are twins, contradicting the fact that $m(C)$ does not contain twins. Indeed, let h be a t-node such that $E(h)$ contains a pair (g, γ) for some t_2-node γ. We show that $E(h)$ also contains (g', γ). Consider a model τ of g, a model τ' of g' and σ a model of γ. By definition, $\tau \times \sigma$ is a model of h. Let α be a model of sub_h, that is, $\beta := \alpha \times \tau \times \sigma$ is a model of C. Now it means that $\alpha \times \sigma$ is a model of sub_g, hence, it is also a model of $\mathsf{sub}_{g'}$. In other words, $\beta' := \alpha \times \sigma \times \tau'$ is a model of C. Now the certificate of β' must contain h, because α is a model of sub_h and we assumed that h is the only t-node for this X_t-subfunction. Hence, $(g', \gamma) \in E(h)$. Since the argument is symmetric in g and g', we immediately get $\mathsf{sib}(g, h) = \mathsf{sib}(g', h)$ for any t-node h. In other words, g and g' are siblings which contradicts the construction of $m(C)$.

Hence, for every node t of T and g, g' distinct t-nodes of $m(C)$, $\mathsf{sub}_g \neq \mathsf{sub}_{g'}$. The X_t-subfunction defined by g is non-trivial because we removed dangling nodes. Hence $m(C)$ contains $\sum_{t \in T} S_t$ gates. □

Theorems 2.21 and 2.22 together prove that $m(C)$ has minimal size. Moreover, this minimal circuit is unique because each gate is uniquely defined by the factor it computes. Indeed, each t-node of $m(C)$ computes exactly one X_t-factor of f_C and we need exactly one t-node for each non-trivial factor. Deterministic TDD can therefore be minimized in polynomial time into a canonical minimal circuit.

Observe that the time needed to minimize the circuit can be parametrized by the width of the circuit. Indeed, detecting and contracting twins can be done at each node t in time polynomial in the width of the circuit. Let t be a node of T with children t_1, t_2. If we have two t_1-nodes that are twins, merging them can only create new twins in t_1-nodes or in t_2-nodes. Hence, for every t, we can iterate over t_1-nodes and t_2-nodes until there are no twins anymore among them. This can be done in polynomial time in the width of C. Now, if we proceed from the root of T to its leaves, then when this contraction finishes, the circuit cannot have twins anymore. Hence the total time for this contraction is in time $\mathrm{poly}(k)|X|$ where k is the width of the TDD and X its set of variables. Removing 0-constants and dangling gates can also be done in time $\mathrm{poly}(k)|X|$ for the same reasons. Hence, we have the following:

Theorem 2.23. *Given a TDD C of width k over variables X, we can minimize C in time $\mathrm{poly}(k)|X|$.*

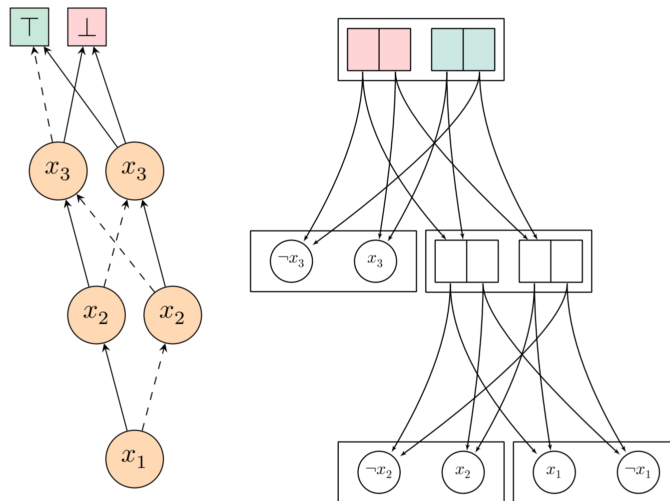


Figure 2.4: An OBDD represented as a TDD. The output of the TDD is represented in green and each node t of the vtree is made explicit by a rectangle around t-nodes.

The exact complexity of the minimization procedure heavily depends on the actual data structures that are used to represent the circuit gates and their inputs. We will not go into such detailed granularity and leave such an analysis for a practical implementation of a TDD compiler.

2.1.5 Comparing d-TDD with other data structures

2.1.5.1 Deterministic TDD and OBDD

Tractable queries and transformations for TDD are similar to the tractable queries and transformations for OBDD. In this section, we argue that TDD can be seen as a natural extension of OBDD to trees. More precisely, OBDD can be seen as TDD on a *linear vtree*. A *linear vtree* is a vtree such that for each internal node t of T, the right child of t is a leaf.

Given a set of variables X and $\pi = (x_1, \dots, x_n)$ an order on X, we define T_π to be the linear vtree defined as follows: T_π has $2n-1$ nodes $\ell_1, \dots, \ell_n$ and $p_n, \dots, p_2$. The root of T_π is p_n. For every $i > 1$, the left child of p_i is p_{i-1}, the right child of p_i is ℓ_i. The left child of p_2 is ℓ_1 and its right child is ℓ_2. The nodes ℓ_i are leaves labeled by x_i.

Similarly, we can extract an order π_T from every linear vtree. Given a linear vtree T on variables X, $\pi_T = (x_1, \dots, x_n)$ is the order defined from T as follows: if T has exactly one node, then it is a leaf labeled by x_1 and the order is simply (x_1). Otherwise, let r be the root of T. We let x_1 be the label of the right child of r, which is a leaf by definition, and we define inductively $(x_2, \dots, x_n) = \pi_{T'}$ where T' is the vtree rooted at the left child of r.

We claim in this section that an OBDD on some order π is precisely a TDD respecting T_π. We illustrate the correspondence with an example in Fig. 2.4 and sketch the correspondence below.

Theorem 2.24. *For every complete OBDD C on variables X and order $\prec = (x_1, \dots, x_n)$, there exists a TDD C' respecting $T_\prec$ of size at most $3|C|$ computing the same function and having the same width as C.*

Proof. Intuitively the TDD corresponds to rooting the OBDD at its 1-sink. We assume without loss of generality that C has exactly two sinks: a 1-labeled sink and a 0-labeled sink. For each decision node v but the source of C, we introduce a node g_v in the TDD. The models of g_v are exactly the set of assignments that are compatible with a path from the source of C to v.

First assume v is a decision-node on x_2. Either every assignment of variable x_1 leads to v (that is, both outgoing edges of the source of C are connected to v), in which case we define g_v as the unique ℓ_1-node labeled by $\top$. Otherwise, we define g_v as the ℓ_1-node labeled by the literal corresponding to the path leading to v. More precisely:

- If both outgoing edges of the source of C are connected to v, then g_v is labeled by 1 (and it is the only ℓ_1-node of C').
- If only the 1-labeled outgoing edge of the source of C is connected to v then g_v is labeled by x_1.
- If only the 0-labeled outgoing edge of the source of C is connected to v then g_v is labeled by $\neg x_1$.

By definition, the models of g_v are clearly the set of assignments of $\{x_1\}$ whose path in C ends in v. Now assume v is a decision-node on x_{i+1} ($i > 1$) and that g_w has been constructed for every node w before v in C. We let g_v be a new p_i-node and we introduce two ℓ_i-nodes L_i^1 and L_i^0 respectively labeled by x_i and $\neg x_i$. The inputs of g_v are defined as follows: it contains a pair (L_i^1, g_w) for every node w such that there is an edge labeled by 1 from w to v and a pair (L_i^0, g_w) for every node w such that there is an edge labeled by 0 from w to v. By induction, the models of g_v are the assignment $\tau \in 2^{\{x_1,\dots,x_i\}}$ such that the path of τ goes to w and such that, $\tau(x_i) = b$ if and only if there is an b-labeled edge between w and v. But this is exactly what it means for the path of τ to ends in v. Hence the induction hypothesis is preserved.

If v is a sink, we do the same construction as above but on variable x_n. That is, we have two ℓ_n-nodes L_n^0 and L_n^1 labeled by $\neg x_n$ and x_n respectively. We introduce a p_n-node g_v whose inputs are pairs (L_n^b, g_w) if (w, v) is an edge of C labeled by $b \in \{0, 1\}$. As before, the models of g_v are exactly the assignments whose path ends in v. Hence, if we choose the output of the C' to be g_v where v is the 1-sink of C, the models of C' are the same as the models of C. Observe that we have not changed the width of the circuit. □

The construction works both ways in the sense that any TDD on a linear vtree can also straightforwardly be transformed into an OBDD:

Theorem 2.25. *For every TDD C respecting a linear vtree T, there exists an OBDD C' of size at most $3|C|$ using order $\pi_T = (x_1, \dots, x_n)$ and computing the same function as C.*

Proof. We only sketch the construction. We have one decision gate $v(g)$ in C' for each t-node g where t is an internal node of T. We define it as follows: Let g be a t-node of C where t

is an internal node of T. Let u be the parent of t in T and ℓ be the sibling of t in T. By definition, ℓ is labeled by some variable x_i since T is linear. We add a decision gate $v(g)$ in the OBDD on variable x_i. The 1-edge of $v(g)$ is connected to $v(h)$ where h is the only u-node containing input (x_i, g) or $(\top, g)$. If no such h exists, we plug the 1-edge of $v(g)$ into a $\bot$-sink. We connect the 0-edge of $v(g)$ similarly to $v(h)$, where h is the only u-node containing input $(\neg x_i, g)$ or $(\top, g)$. □

Theorems 2.24 and 2.25 offer direct construction between OBDD and deterministic (linear) TDD. We observe however that Theorem 2.21 offer a way to obtain the same correspondence in a non (directly) constructive way. Indeed, it is known [HI98] that the width of the minimal OBDD is exactly the maximum number of subfunctions one can get by fixing variables $x_1, \ldots, x_i$ for some $i \leq n$. This exactly corresponds to the number of subfunctions we can have with a linear vtree. The previous constructions can be extended to the case of non-deterministic TDD and non-deterministic OBDD. TDDs and OBDDs are separated, however. This follows as a corollary of [Raz14] and Theorem 2.39 that we will prove later in this manuscript. In [Raz14], Razgon proves that there exists a family of CNF formulas $(F_n)_{n\in\mathbb{N}}$ where F_n has n variables and treewidth $O(\log n)$ such that every OBDD representing F_n must have size $n^{\Omega(\log n)}$, while Theorem 2.39 shows that such instances have a polynomial size TDD.

Theorem 2.26. *There exists a family $(F_n)_{n\in\mathbb{N}}$ of CNF formulas such that F_n can be represented by a polynomial size TDD while every OBDD representing F_n has size at least $n^{c \log n}$ for some constant c.*

The separation given by Theorem 2.26 is only quasi-polynomial and one can wonder whether a truly exponential separation is possible. It may seem possible that TDD can be quasi-polynomially simulated by OBDD, in the same way FBDDs quasi-polynomially simulate decision-DNNF circuits [Bea+13]. We leave this question for future investigation though:

Open question 3. *Given a TDD C of size N, does it exist an OBDD C' of size at most $N^{O(\text{polylog}(N))}$ computing f_C?*

2.1.5.2 Deterministic DNNF

As we have already observed, TDD is a subclass of structured deterministic DNNF circuits. We show in this section that structured deterministic DNNF circuit can be exponentially smaller than TDD. To do so, we will be interested in the *Hidden Weighted Bit* function, which are known to be hard for OBDD [Weg00]. Given $n \in \mathbb{N}$, we define $\mathsf{HWB}_n(x_1, \ldots, x_n)$ to be the Boolean function defined as:

$$\mathsf{HWB}_n(x_1, \ldots, x_n) = x_S \text{ where } S = \sum_{i=1}^{n} x_i$$

In other words, the models of HWB_n are the assignments setting $x_S = 1$ where S is the number of variables set to 1. It can easily be rewritten as follows:

$$\mathsf{HWB}_n(x_1,\dots,x_n) = \bigvee_{i=1}^{n} x_i \wedge \mathsf{HAM}_i(x_1,\dots,x_n)$$

where HAM_i is the function that evaluates to 1 if the number of variables set to 1 is exactly i. Observe that this disjunction is deterministic. Moreover, it is easy to build an OBDD H_i of width n using the variable order $(x_1,\dots,x_n)$ computing $x_i \wedge \mathsf{HAM}_i(x_1,\dots,x_n)$ by simply keeping track of the number of variables set to 1 in each layer, where the layer testing on x_i only has outgoing edges labeled by 1. The last layer accepts iff the number of variables set to 1 is i. In other words, HWB_n can be computed by an unambiguous OBDD of size $O(n^2)$, and in particular, by a structured deterministic DNNF circuit.

Lemma 2.27. *For every $n \in \mathbb{N}$, there exists a structured deterministic DNNF circuit of size $O(n^2)$ computing HWB_n.*

We now show that for every vtree T, the factor-width of HWB_n on T is at least 2^{cn} for some constant c. In other words, we show that $\mathsf{fw}(\mathsf{HWB}_n) \geq 2^{cn}$ and by Theorem 2.21, it gives a 2^{cn} lower bound on the size of any TDD computing HWB_n, separating TDD from structured deterministic DNNF circuits. The proof essentially follows the known lower bound for OBDDs, for example, the one that can be found in [Weg00, Lemma 4.10.1], with small adaptations to go from orders to trees. It relies on this technical lemma (whose proof is delayed at the end of this section).

Lemma 2.28. *Let $n \in \mathbb{N}$ and $I \subseteq [n]$. Assume there exists $p, \ell \in [n]$ such that:*

- $J := I \cap [p; p+\ell]$ *has size* t,
- $\ell \leq n - |I|$,
- $n \geq p + \ell + t/2$.

Then HWB_n has at least $\binom{t}{t/2}$ subfunctions over $X_I := \{x_i \mid i \in I\}$.

It can be used as follows:

Theorem 2.29. *Let $n \in \mathbb{N}$ which is a multiple of 7. We have $\mathsf{fw}(\mathsf{HWB}_n) \geq 2^{cn}$ for some constant c. In particular, any TDD computing HWB_n has size at least 2^{cn}.*

Proof. Let T be a vtree over $\{x_1,\dots,x_n\}$ and let t be a node of T such that $n/3 \leq |X_t| \leq 2n/3$. We let $I = \{i \mid x_i \in X_t\} \subseteq [n]$. We show that we can find a sub-interval $J = [p; p+\ell]$ respecting the conditions of Lemma 2.28 with $t = \beta n$ for some constant β. This is enough to prove the claim as $\binom{\beta n}{\beta n/2} \geq 2^{\frac{\beta n}{2}}$.

We start by splitting $[n]$ into seven segments of size $n/7$: $J_0 = [1, n/7], \dots, J_6 = [6n/7, n]$. Since $J_0 \cup J_6$ has size at most $2n/7$. Hence $I \setminus (J_0 \cup J_6)$ has size at least $|I| - |J_0| - |J_6| \geq (1/3 - 2/7)n = n/21$. It means there must be J_i for $1 \leq i \leq 5$ such that $J_i \cap I$ has size $t \geq (n/21)/5 = n/105$. We let $p = i(n/7)+1$, $\ell = n/7$, $J = J_i = [p; p+\ell]$ and $t = |J \cap I| \geq n/105$.

We have to verify the conditions of Lemma 2.28, namely:

- $\ell \leq n - |I|$. This is straightforward as $\ell = n/7 \leq n/3 = n - 2n/3 \leq n - |I|$ since $|I| \leq 2n/3$.
- $n \geq p+\ell+t/2$. In our case, $p+\ell+t/2 = i(n/7)+1+n/7+n/210 \leq 6n/7+1+n/210 \leq n$.

Hence, by Lemma 2.28, HWB_n has at least $2^{t/2} \geq 2^{n/210}$ subfunctions over X_t. □

Corollary 2.30. *Deterministic TDD are exponentially less succinct than structured deterministic DNNF circuits.*

We finish this section by providing the proof of Lemma 2.28.

Proof of Lemma 2.28. Let A be the set of assignments of X_J setting exactly $t/2$ variables to 1 (and $t/2$ to 0). Given $a \in A$, we define $a^* \in 2^{X_I}$ as follows: we set the largest possible number of variables $X_I \setminus X_J$ to 1 so that the total number of variables set to 1 by a^* does not exceed p. Since $|X_I \setminus X_J| = |I| - t$ and a already sets $t/2$ variables to 1, we have two cases:

- either $|I| - t \geq p - t/2$, in which case, we pick $p - t/2$ variables from $X_I \setminus X_J$, set them to 1 and then the other to 0. In this case, we have that a^* sets exactly p variables to 1.
- or $|I| - t < p - t/2$, in which case, we set every variable from $X_I \setminus X_J$ to 1, in which case, a^* sets $t/2 + |I| - t = |I| - t/2 < p$ variables to 1.

Observe that the number of variables set to 1 by a^* does not depend on a, but only on whether $|I| - t/2 \geq p$. We denote this number by N_1.

We claim that given distinct $a, b \in A$, $\mathsf{HWB}_n[a^*]$ and $\mathsf{HWB}_n[b^*]$ are two different subfunctions of HWB_n over X_I. Indeed, since a and b are distinct, there exists $i \in J$ such that $a(x_i) \neq b(x_i)$, and in particular, $a^*(x_i) \neq b^*(x_i)$. Assume without loss of generality that $a^*(x_i) = 1$ and $b^*(x_i) = 0$. Now if we pick $c \in 2^{X \setminus X_I}$ such that c sets exactly $i - N_1$ variables to 1 (assume for now that this is possible). In this case, $a^* \times c$ and $b^* \times c$ both set exactly i variables to 1. But then exactly $a^* \times c$ and $b^* \times c$ is a model of HWB_n since $a^*(x_i) = 1$ but $b^* \times c$ is not a model of HWB_n. Hence $\mathsf{HWB}_n[a^*]$ and $\mathsf{HWB}_n[b^*]$ are distinct subfunctions. In other words, HWB_n has at least $|A| = \binom{t}{t/2}$ subfunction over X_I.

It remains to show that we can pick $c \in 2^{X \setminus X_I}$ so that it sets $i - N_1$ variables to 1. First, observe that $i - N_1 \geq 0$ since by definition, $N_1 \leq p$ and $i \in [p; p+\ell]$. We now have to prove that $|[n] \setminus I| = n - |I| \geq i - N_1$. Since $i \leq p + \ell$, we will prove $n - |I| \geq p + \ell - N_1$. We distinguish two cases. First assume $|I| - t/2 \geq p$, that is $N_1 = p$. In this case, we need to prove $n - |I| \geq \ell$ but this was part of our assumption. Otherwise, $|I| - t/2 < p$ and $N_1 = |I| - t/2$. We hence have to prove $n - |I| \geq p + \ell - |I| + t/2$, that is $n \geq p + \ell + t/2$ which is also part of our assumption. □

2.1.5.3 Sentential Decision Diagrams

Another class of circuits resembling TDD is the class of Sentential Decision Diagrams (SDD) introduced by Darwiche in [Dar11]. SDD are also a restriction of structured deterministic DNNF circuits with canonical representations, efficient negations and efficient apply. Hence,

they enable bottom-up construction of circuits. An efficient bottom-up compiler has been implemented together with nice heuristics allowing to dynamically change the vtree to minimize the constructed SDD [CD13], something that would be necessary to understand for TDD too if one wants to implement an efficient and practical compiler.

SDD is another natural generalization of OBDD based on the following observation: in an OBDD, a decision gate on variable x can be seen as a way of partitioning the models of the circuit into those satisfying the Boolean function x and the one satisfying the Boolean function $\neg x$. The functions x and $\neg x$ partition the space of Boolean functions over $\{x\}$. SDD generalizes this idea by introducing the notion of X-partition. Given a set of variables X, an X-partition is a set of Boolean function $p_1, \dots, p_k$ over X such that:

- $p_i \wedge p_j = \bot$, that is, p_i and p_j do not have any common model
- $\bigvee_{i=1}^{k} p_i = \top$, that is, every assignment $\tau \in 2^X$ satisfies at least one p_i.

In other words, $p_1, \dots, p_k$ is an X-partition if and only if for every $\tau \in 2^X$, τ satisfies exactly one p_i. Now, gates of SDD decomposes function using X-partitions in the following way. Given a set of variables Z, disjoint subsets $X, Y \subseteq Z$ such that $Z = X \cup Y$, and a Boolean function f over Z , an (X,Y)-*decomposition of* f is a set of pairs $\{(p_1, s_1), \dots, (p_k, s_k)\}$ of Boolean functions such that: $\{p_1, \dots, p_k\}$ is an X-partition and $f = \bigvee_{i=1}^{k}(p_i \wedge s_i)$.

Let T be a vtree over variables X and t a node of T. An SDD C that respects the vtree rooted at t is either:

- A single gate g labeled by a constant $\{0, 1\}$, in which case, it computes the constant Boolean function,
- If t is a leaf of T labeled by x, it can be a single gate g labeled by a literal (x or $\neg x$). In this case, it computes the Boolean function associated with each literal.
- Otherwise, if t is an internal node of T with children t_1, t_2 and C is given by a gate g labeled by $\{(p_1, s_1), \dots, (p_k, s_k)\}$ where for every $i \leq k$, p_i is an SDD respecting t_1, s_i is an SDD respecting t_2 and $\{f_{p_1}, \dots, f_{p_k}\}$ is an X_{t_1}-partition. In this case, it computes $\bigvee_{i=1}^{k}(f_{p_i} \wedge f_{s_i})$.

It is easy to build a deterministic DNNF circuit from an SDD of linear size in the size of the SDD by simply rewriting every internal gate as a disjunction of conjunctions, the disjunction being deterministic because of the partition property, and the conjunction is decomposable because of the underlying vtree structure.

SDD are canonical in the following sense. An SDD is said to be *trimmed* if it does not have an internal gate of the form $\{(\top, \alpha), (\bot, \alpha)\}$ nor $\{(\top, \alpha)\}$. It is easy to trim an SDD in polynomial time by simply replacing such gates by α. Now, we say that an SDD is *compressed* if every internal node labeled by $\{(p_1, s_1), \dots, (p_k, s_k)\}$ has the property that $s_i \neq s_j$ for every $i < j \leq k$. One can transform an SDD to a compressed SDD by duplicating gates that are used several times within internal nodes. This transformation however cannot be done in polynomial space: the duplication may cascade, resulting in an exponential blow-up. The terminology here is a bit misleading as one could think that compressed SDD are smaller than

the others but this is really not the case. We stick to the original terminology from [Dar11], where Darwiche shows that for every Boolean function f over variables X and a vtree T, there is a unique, canonical, trimmed and compressed SDD respecting T and computing f.

The canonicity of SDD is however really different from the canonicity of OBDD and TDD. First of all, the canonical SDD respecting some given vtree T for a Boolean function f is not the smallest SDD respecting T and computing f, as the compressing operation may blow the size of the SDD. This has undesired side effects. Indeed, the main application of canonicity is to check whether two circuits compute the same function. The previous observation can be understood as follows: checking the equivalence of two given SDD C_1, C_2 by comparing their canonical form does not run in polynomial time. Indeed, constructing the canonical representations of C_1 and C_2 to check that they are equal may induce an exponential blow-up.

Another important difference between the canonicity of SDD and that of OBDDs and TDD: the additional syntactic restrictions of trimming and compressing put on SDDs are not stable by conditioning. This means that the canonical SDD of $f[x/1]$ over a vtree T may be very different from the canonical SDD of f over a vtree T which is somehow unexpected. Actually, Darwiche and Van der Broeck have shown [VD15] that an exponential blow-up in the size of the canonical SDD may happen when conditioning (as long as we impose the same vtree). It is however open to prove that the canonical SDD of the conditioned function may be large for any vtree.

We conclude this section with a comparison between SDD and TDD. One can see that SDD can be cast into non-deterministic TDD. Indeed, internal gates compute disjunction in the similar way as for TDD. The only thing that SDD can do which is not allowed in TDD is to have constant inputs that respect internal nodes of the vtree but we can easily pad the TDD with dummy nodes.

Theorem 2.31. *Given an SDD C on variables X, there exists a non deterministic TDD C' computing the same function and has size at most $|X| \cdot |C|$.*

The transformation from Theorem 2.31 does not yield a (deterministic) TDD despite the fact that the TDD constructed this way has a small deterministic DNNF circuit corresponding to it because it retains the partition property from SDD. That said, the notion of determinism for TDD is very different from the notion of partitions. Indeed, an internal gate $\{(p_1, s_1), \dots, (p_k, s_k)\}$ in an SDD cannot use the same p_i twice as the determinism is ensured by the fact that $(p_i)_{i \leq k}$ form a partition. This asymmetry between the left and right part of SDD is not present in TDD but determinism is ensured in a very different way. In a TDD, determinism can be rephrased as the fact that for each node t of the vtree T, t-nodes form an X_t-partition, but neither the left nor the right child are responsible for this.

The results from the previous section can however be used to prove the following:

Theorem 2.32. *SDD and compressed SDD cannot be polynomially simulated by TDD.*

Proof. This follows from a result by Bova [Bov16]. He shows that HWB_n from the previous section can be computed by an SDD of size $O(n^3)$. Hence, the lower bound follows from the lower bound of Theorem 2.29. In the same paper, Bova also builds a Boolean function $F(X, Y)$ such that $F(X, 1, \dots, 1) = \mathsf{HWB}_n(X)$ and such that F has a small compressed SDD.

Since TDD can be conditioned in polynomial time, the smallest TDD for F must be of size polynomial in the size of the smallest TDD of HWB_n. By Theorem 2.29 again, F cannot have polynomial size TDD. □

We however leave the symmetric question open:

Open question 4. *Can TDDs be polynomially simulated by compressed SDD or by SDD?*[2]

2.2 Compiling Structured CNF formulas

We are ready to use TDDs to prove tractability results on CNF formulas having a specific structure. We use a very simple strategy known as *bottom-up compilation*, which has successfully been employed for OBDD already [Som98]. We present this strategy in detail for TDD and then proceed to study its complexity guarantees in the case of structured CNF formulas, namely, CNF formulas with bounded treewidth. We also improve the results from Bova and Szeider [BS17] and provide a new efficient algorithm to construct a TDD from any Boolean circuit.

2.2.1 Bottom-up compilation of CNF formulas

The bottom-up strategy for knowledge compilation usually refers to algorithms that compile a CNF formula F by iteratively using the APPLY operator until a circuit for F is found. In its simplest form, the algorithm proceeds as follows: on input F, it starts by ordering the clauses of F as $c_1, \dots, c_n$ and then builds a circuit B_i for each clause c_i. It then builds a circuit for F by computing the following sequence of circuits: $C_1 = B_1$ and for every $i < n$, $C_{i+1} := APPLY(C_i, B_{i+1}, \wedge)$. It is straightforward to see that C_i computes $c_1 \wedge \dots \wedge c_i$ and hence C_n computes F.

This technique can be applied in a straightforward fashion to any class of circuits supporting the apply transformation and for which every clause can be represented by a small circuit. The former is guaranteed by Theorem 2.15 while we prove the latter now:

Proposition 2.33. *Let T be a vtree over variables X and c be a clause such that $\mathsf{var}(c) \subseteq X$. We can construct in time $O(|X|)$ a TDD C of width 2 respecting T such that the models of C are exactly the models of c over X.*

Proof. We construct a TDD $C = (N, E)$ such that for every node t of T, N_t contains exactly two gates g_0^t and g_1^t such that the models of g_0^t are the models of $\neg c|_{X_t}$ and the models of g_1^t are the models of $c|_{X_t}$. First assume t is a leaf labeled by x. If x does not appear in c and we set g_0^t be a 0-input gate and g_1^t be a 1-input gate. If there is a literal ℓ in c over x, we let g_0^t to be an input labeled by $\neg\ell$ and g_1^t to be an input labeled by ℓ. Now if t is a node of T with children t_1, t_2, we have $c|_{X_t} = c|_{X_{t_1}} \vee c|_{X_{t_2}}$, hence a model of $c|_{X_t}$ is either a model of $c|_{X_{t_1}}$ and $c_{X_{t_2}}$, or a model of only $c|_{X_{t_1}}$ (and of $\neg c|_{X_{t_2}}$) or a model of only $c|_{X_{t_2}}$ (and of $\neg c|_{X_{t_1}}$). Hence we define $E(g_1^t) = \{(g_1^{t_1}, g_1^{t_2}), (g_1^{t_1}, g_0^{t_2}), (g_0^{t_1}, g_1^{t_2})\}$ and $E(g_0^t) = \{(g_0^{t_1}, g_0^{t_2})\}$.

[2]This question is not completely open anymore. SDD can simulate TDD but it is not clear whether compressed SDD can. The final version of this manuscript will contain the proof.

If r is the root of T, we set $\mathsf{out}(C) = g_1^r$, which computes c by induction. □

However, iterating the apply transformation with the complexity guarantees from Theorem 2.15 will quickly lead to an exponential blow-up in size. Indeed, without further simplification of the circuit, the width of C_i will be 2^i. However, this upper bound is easy to get around since we know how to minimize TDDs by Theorem 2.22. Hence we could replace our previous simplistic scheme by the one presented on Algorithm 1, where the circuit is minimized after each apply.

Algorithm 1 Bottom-up compilation into TDD.

Input: A CNF formula $F = c_1 \wedge \cdots \wedge c_m$, a vtree T over $\mathsf{var}(F)$.
Output: A TDD computing F respecting T.

```
1: procedure CNF-TO-TDD(F, T)
2:     C ← TDD computing 1 respecting T
3:     for i=1 to m do
4:         D ← TDD computing c_i
5:         C ← APPLY(C, D, ∧)
6:         minimize(C)
       return C
```

Now, interestingly, our precise understanding of the width of TDDs allows to bound the running time of Algorithm 1 using the number of X_t-subfunctions of the input CNF formula. It naturally corresponds to the definition of *factor-width* from [BS17]: given a vtree T over X and f a Boolean function on variables X, the *factor-width of f with respect to T*, denoted by $\mathsf{fw}(f, T)$, is defined as $\max_t |\mathsf{sub}_t^T(f)|$ where $\mathsf{sub}_t^T(f)$ is the set of distinct X_t-subfunctions of f (which corresponds to the number of X_t-factors, hence the name). We abuse notation and, for a given a CNF formula F and a vtree T over $\mathsf{var}(F)$, we write $\mathsf{fw}(F, T)$ to denote the factor-width of the Boolean function represented by F with respect to T.

Theorem 2.34. *Given a CNF formula F, a vtree T over variables X and an order $c_1, \dots, c_m$ on the clauses of F, Algorithm 1 runs in time $m|X| \cdot \mathrm{poly}(k)$ where $k = \max_{i=1}^m \mathsf{fw}(c_1 \wedge \cdots \wedge c_i, T)$.*

Proof. Constructing B_i for each clause c_i can be done in $O(|X|)$ by Proposition 2.33. Now, by Proposition 2.14 and Theorem 2.23, both the apply and minimization steps run in $|X| \mathrm{poly}(k)$. Indeed, before the apply, C is the minimal TDD computing $c_1 \wedge \cdots \wedge c_i$ which has width at most k by Theorem 2.22. Since B_i has width 2, after the apply, C has width $2k$. Hence, the minimization runs in $|X| \mathrm{poly}(k)$. The total complexity of Algorithm 1 is then $m|X| \cdot \mathrm{poly}(k)$. □

2.2.2 Structured CNF formulas

factor-width is a semantic definition that does not directly relate to the structure of the CNF formula, that is, the way variables and clauses interact. It is hence hard to compare the tractability result from Theorem 2.34 with earlier results on tractable compilation for

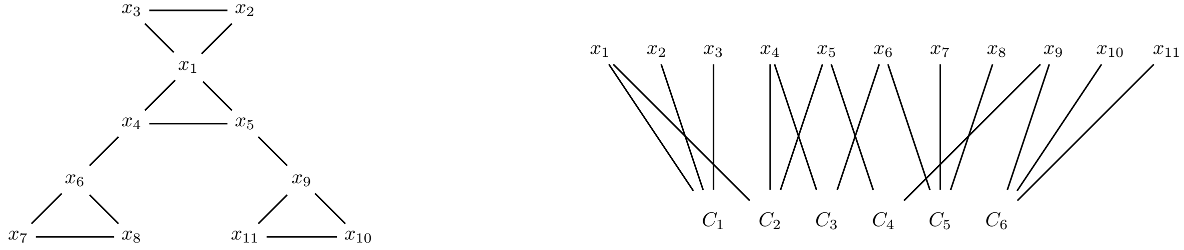


Figure 2.5: The primal $\mathsf{Prim}(F)$ and incidence $\mathsf{Inc}(F)$ graphs of Example 2.35.

structured CNF formulas as in [Bov+15; Cap17]. It turns out that some structural results from [Bov+15] are subsumed by Theorem 2.34 because the structure exploited there implies that the factor-width of any subset of clauses of F is bounded.

In this section, we let F be a CNF formula on variables X. We characterize the structure of a formula using graphs. The first graph we consider is the *primal graph* of F, denoted by $\mathsf{Prim}(F) = (X, E)$. The vertices of $\mathsf{Prim}(F)$ are the variables of F and there is an edge $\{x, y\}$ in E if and only if there is a clause c of F such that $x, y \in \mathsf{var}(c)$. Observe that large clauses induce large cliques in $\mathsf{Prim}(F)$. Hence, the primal graph is hiding clause-variables interactions if some clauses are included in others. An extreme case is a formula containing a clause c with $\mathsf{var}(c) = X$. In this case, $\mathsf{Prim}(F)$ is a $|X|$-clique and the structure of $F \setminus \{c\}$ is hidden in $\mathsf{Prim}(F)$.

A more faithful graph for F is its *incidence graph*, denoted by $\mathsf{Inc}(F) = (X \cup F, E)$. The vertices of $\mathsf{Inc}(F)$ are both the variables and the clauses of F. There is an edge $\{x, c\}$ for $x \in X$ and $c \in F$ in $\mathsf{Inc}(F)$ if and only if $x \in \mathsf{var}(c)$. Observe that $\mathsf{Inc}(F)$ is bipartite. It is not isomorphic to the CNF formula F since the sign of x in c is not visible in $\mathsf{Inc}(F)$. A directed version of the incidence graph has been considered in the literature where we have an edge $x \to c$ if x appears positively in c and $x \leftarrow c$ if x appears negatively in c. The directed incidence graph is isomorphic to the CNF formula.

Example 2.35. Fig. 2.5 represents the primal and incidence graphs of $F = C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5 \wedge C_6$ where $C_1 = (x_1 \vee x_2 \vee x_3), C_2 = (x_1 \vee x_4 \vee x_5), C_3 = (x_4 \vee x_6), C_4 = (\neg x_5 \vee x_9), C_5 = (x_6 \vee x_7 \vee x_8), C_6 = (x_9 \vee x_{10} \vee x_{11})$. One can observe that clauses induce cliques in the primal graph (here, triangles) but the incidence graph is actually acyclic.

Treewidth. One can now study the structure of F by analyzing the structure of $\mathsf{Prim}(F)$ or $\mathsf{Inc}(F)$ using well-studied graph decomposition tools. One of the most studied graph decomposition is without a doubt the notion of *tree decomposition* which intuitively maps a graph to a tree, allowing one to design algorithms exploiting this tree structure. A *tree decomposition* $\mathcal{T}$ *for a graph* $G = (V, E)$ is a tree such that each node t of $\mathcal{T}$ is labeled by a subset B_t of V, called a *bag at* t. Moreover, $\mathcal{T}$ has the following properties:

1. **Connectedness:** for every $x \in V$, the set $\{t \mid x \in B_t\}$ is connected in $\mathcal{T}$,

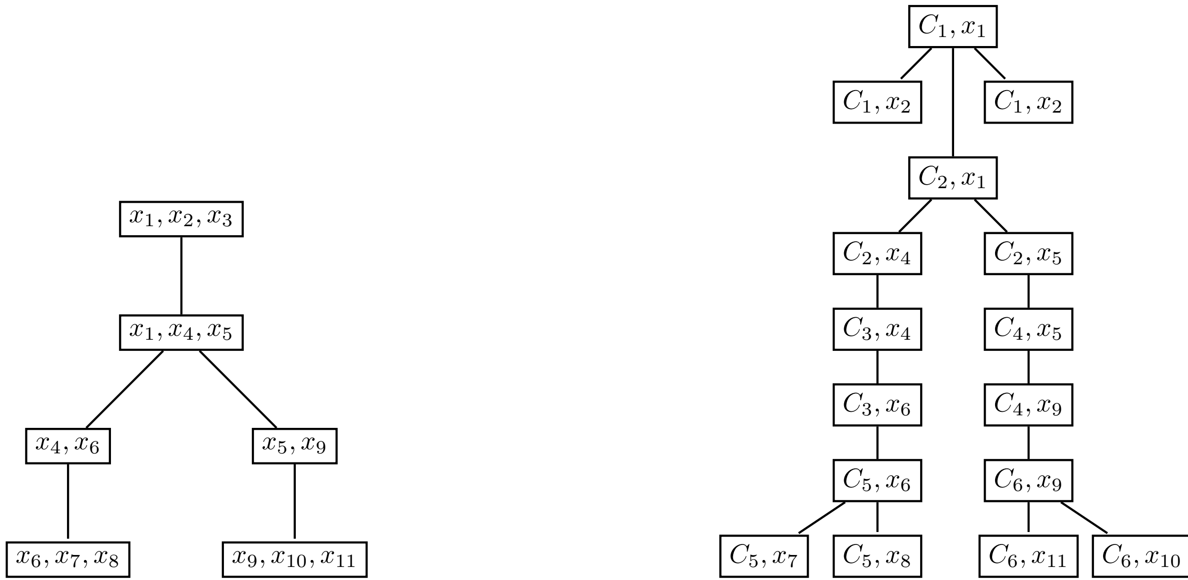


Figure 2.6: Tree decompositions for $\mathsf{Prim}(F)$ (left) and $\mathsf{Inc}(F)$ (right) from Example 2.35.

2. **Completeness:** for every edge e of G, there exists a node t such that $e \subseteq B_t$.

In other words, a tree decomposition fits a graph into a tree such that the structure of the graph is still visible in the tree. For example, if one removes a node t of $\mathcal{T}$, $\mathcal{T}$ is split into two connected components $\mathcal{T}_1, \mathcal{T}_2$. Let V_1, V_2 be the subset of vertices appearing in $\mathcal{T}_1$ and $\mathcal{T}_2$ respectively. It is easy to see by connectedness that $V_1 \cap V_2 = B_t$. Moreover, every edge $e = \{v_1, v_2\}$ with one endpoint $v_1 \in V_1$ and another endpoint in $v_2 \in V_2$ is such that $e \subseteq B_t$ by completeness. In other words, one node of the tree decomposition splits the graph into two disconnected subgraphs G_1, G_2 having at most $|B_t|^2$ edges between V_1 and V_2.

Now, one can obviously represent any graph G with a very simple tree decomposition: $\mathcal{T}$ has one node t labeled by V. This is indeed a valid tree decomposition for G but it does not give any insight on the structure of G. The previous observation hints at the fact that having small bags is desirable because it decomposes the graph into small disconnected graphs having only a few edges between them. The notion of treewidth exactly captures this complexity: the *treewidth of a tree decomposition $\mathcal{T}$ of G*, denoted by $\mathsf{tw}(G, \mathcal{T})$ is defined as $\max_t |B_t| - 1$ and the *treewidth of G*, denoted by $\mathsf{tw}(G)$, is defined to be $\min_{\mathcal{T}} \mathsf{tw}(G, \mathcal{T})$, where $\mathcal{T}$ runs over all valid tree decompositions of G.

Example 2.36. Fig. 2.6 gives an example of tree decompositions of the graphs of Example 2.35, depicted on Fig. 2.5. The fact that $\mathsf{Inc}(F)$ is acyclic is witnessed here by a tree decomposition of width 1, which is equivalent to being acyclic. The tree decomposition of the primal graph pictured on Fig. 2.5 is of width 2 which is also optimal, since it is not acyclic.

We can apply the notion of treewidth to CNF formulas, via either its primal or incidence

graph. The *primal treewidth of a CNF formula F*, denoted by $\mathsf{ptw}(F)$, is defined as the treewidth of $\mathsf{Prim}(F)$ while the *incidence treewidth of a CNF formula F*, denoted by $\mathsf{itw}(F)$, is defined to be the treewidth of $\mathsf{Inc}(F)$. It is not hard to see that for every CNF formula F, we have $\mathsf{itw}(F) \leq \mathsf{ptw}(F)+1$. Moreover, the CNF formula $F_n = (x_1 \vee \cdots \vee x_n)$ with one clause has incidence treewidth 1 (since its incidence graph is a tree) and primal treewidth $n-1$ since its primal graph is an n-clique. Hence the incidence treewidth can be bounded while the primal treewidth is not. We say that incidence treewidth is more general than primal treewidth.

In both cases, we can show that a CNF formula with small treewidth also has small factor-width:

Theorem 2.37. *Given a CNF formula F, we have* $\mathsf{fw}(F) \leq 2^{\mathsf{ptw}(F)}$.

Proof. Let T be a tree decomposition of $\mathsf{Prim}(F)$ of treewidth k. Before describing how to build a vtree, we give an intuition on why F has small factor-width. Let t be a node of T and let $B_{\leq t}$ be the set of variables appearing in a bag below t in T. Let c be a clause of F such that $\mathsf{var}(c) \cap B_{\leq t} \neq \emptyset$. We claim that in this case, either $\mathsf{var}(c) \subseteq B_{\leq t}$ or $\mathsf{var}(c) \cap B_{\leq t} \subseteq B_t$. Indeed, assume that $\mathsf{var}(c) \not\subseteq B_{\leq t}$. That is, c has a variable $y \notin B_{\leq t}$. Let $x \in B_{\leq t} \cap \mathsf{var}(c)$. The edge $\{x, y\}$ is an edge of $\mathsf{Prim}(F)$ and is hence covered in some bag of T. Since $y \notin B_{\leq t}$, the bag covering $\{x, y\}$ is not below t. Hence, by connectivity, $x \in B_t$. This proves that $\mathsf{var}(c) \cap B_{\leq t} \subseteq B_t$.

Now let $\tau \in 2^{B_{\leq t}}$. We claim that $F[\tau]$ only depends on the value of τ over B_t. Indeed, either $F[\tau]$ contains the empty clause and hence it computes the 0-constant function. Otherwise, the clauses in F not satisfied by τ are either clauses of F with no variable in $B_{\leq t}$ or clauses of F having at least one variable in $B_{\leq t}$ and one variable outside $B_{\leq t}$ (clauses having all their variables in $B_{\leq t}$ are satisfied by τ because $F[\tau]$ does not contain the empty clause). In other words, if τ and τ' are such that neither $F[\tau]$ nor $F[\tau']$ contains the empty clause and $\tau_{|B_t} = \tau'_{|B_t}$, then $F[\tau] = F[\tau']$. Hence they define the same subfunction. Since $|B_t| \leq k$, there is at most 2^k such subfunctions.

We conclude this proof by constructing a vtree T' such that for each internal node t' of T', $X_{t'}$ is of the form $B_{\leq t} \setminus U$ for some $U \subseteq B_t$. From what precedes, the number of $X_{t'}$-subfunctions of F will be at most 2^k, since they will only depend on the value of an assignment on variables B_t.

To do so, observe that for each variable x of F, we can find a unique node $t(x)$ in T such that $x \in B_{t(x)}$ and x is not in $X \setminus B_{\leq t(x)}$. In other words, $t(x)$ is the shallowest node of T in which x appears. We build T' by starting from T and attaching a new leaf labeled by x to $t(x)$. We then make the tree binary by replacing nodes with d children with binary tree having d leaves and contract nodes with one child with their child. It is easy to see by construction that T' has the required property and that the factor-width of F wrt T' is at most 2^k. □

We have a similar phenomenon for incidence treewidth:

Theorem 2.38. *Given a CNF formula F, we have* $\mathsf{fw}(F) \leq 2^{\mathsf{itw}(F)}$.

Proof. The proof is similar. Consider a tree decomposition T of $\mathsf{Inc}(F)$ and a node t of T. Let $V_{\leq t} = B_{\leq t} \cap X$, the variables appearing below t. Let c be a clause of F that has variables

both in $V_{\leq t}$ and in $X \setminus V_{\leq t}$. We claim that, in this case, $c \in B_t$ or $\mathsf{var}(c) \cap V_{\leq t} \subseteq B_t$. Let $x \in \mathsf{var}(c) \cap V_{\leq t}$. By definition, c contains a variable $y \in X \setminus V_{\leq t}$. Now, both edges $\{y, c\}$ and $\{x, c\}$ of $\mathsf{Inc}(F)$ have to be covered in some bag of T. The bag covering $\{y, c\}$ cannot be below t because y is not in $V_{\leq t}$. Hence c appears somewhere not below t. If it appears below t, then $c \in B_t$ by connectedness. Otherwise, c does not appear below t and hence, the edge $\{x, c\}$ is not covered below t. But since x appears below t, by connectedness, it has to appear in B_t too.

Now, fix $\tau \in 2^{V_{\leq t}}$, and consider $F[\tau]$. If it contains the empty clause, then it is the 0-constant function. Otherwise, the remaining clauses only depends on the value of τ over B_t and can possibly contain any subset of clauses appearing in B_t. Hence, there is at most 2^k possible CNF formulas for $F[\tau]$. In other words, F has at most 2^k $V_{\leq t}$-subfunctions.

The construction of the vtree T' is similar to the one in Theorem 2.37 to ensure that for every node t' of T', $X_{t'}$ is of the form $V_{\leq t} \setminus U$ for some $U \subseteq B_t$, which concludes the proof. □

Not surprisingly, bounded treewidth CNF formulas have attracted much attention. It has long been known that SAT can be solved in time $2^{O(k)}\|F\|$. The earliest reference of this fact seems to be in a paper by Dantsin from 1979 [Dan79], though it is not specifically stated with the treewidth terminology, later improved by Razborov and Alekhnovich [AR11], where the result is expressed in terms of the equivalent branch-width measure. The generalization for the tractability of #SAT has first been observed by Sang, Bacchus, Beame, Kautz, and Pitassi in [San+04] and later generalized to the more general case of incidence treewidth by Samer and Szeider in [SS10]. The existence of small d-DNNF circuit for such formulas is implicit in Darwiche's early contribution [Dar04] and explicit in a collaboration with Pipatsrisawat [PD10] for primal treewidth. The case for incidence treewidth has been formally proven along more general results in [Bov+15].

The new connection established by Theorems 2.37 and 2.38 allows to nicely recover these tractability results. Indeed, it is straightforward to see that if a CNF formula has primal or incidence treewidth k, then every sub-formula of F has treewidth at most k. Hence, the bottom-up compilation to TDD from Algorithm 1 runs in time $\mathrm{poly}(2^k) \cdot mn = 2^{O(k)} \cdot mn$ on a formula with n variables, m clauses and of primal or incidence treewidth k by Theorem 2.34, provided we start from the vtree given by Theorems 2.37 and 2.38.

Theorem 2.39. *Given a CNF formula F of primal or incidence treewidth k, one can construct a TDD of width at most 2^k computing F in time $2^{O(k)} \cdot mn$.*

We feel that Algorithm 1 gives a conceptually simpler algorithm than the bottom-up dynamic programming on tree decompositions from [SS10] and serves as a nice example of the power of TDDs and minimization. While canonicity in SDD offers similar guarantees [Dar11], we feel that the nice connection between factor-width and the width of the canonical minimal circuit is more natural. Moreover, canonical SDD may be exponentially larger than the smallest SDD representing the same function [VD15] and we feel that the fact that the minimal representation matches the canonical one is more natural.

However, the complexity of this approach is not as good as earlier work where the dependency on the size of the CNF formula is linear. One current weakness of TDDs which prevents them from matching earlier methods' efficiency is that they are *complete*, that is, even if the

Boolean function only depends on a strict subset of variables Y, the resulting circuit will have at least $|X|$ gates. This is particularly hurtful for clauses. Moreover, we could leverage the fact that we only apply clauses and not arbitrary TDDs to improve the construction during the apply and try to minimize at the same time, as is done for OBDDs. Improving this part of the algorithm may lead to interesting practical results. We leave this line of work open. More concretely:

Open question 5. *Can we build a minimal TDDs in time $2^{O(k)}\|F\|$ for CNF formulas of primal and incidence treewidth k using Algorithm 1 by implementing optimized functions for apply and minimization for clauses?*

2.2.3 Circuit treewidth

Another interesting application of Algorithm 1 is related to the compilation of bounded treewidth Boolean circuits, which can be seen as a generalization of incidence treewidth. The treewidth of a Boolean circuit C is defined as the treewidth of its underlying graph. For the notion to make sense, one needs to first assume that for any variable $x \in X$, there is at most one input of the circuit labeled by x (observe that we are considering Boolean circuits here and not their close cousin NNF, hence the inputs are labeled by variables and not by literals; the literal $\neg x$ can be obtained with a negation gate). It turns out that the factor-width of a Boolean function defined as a Boolean circuit of treewidth k is also bounded, which allows us to give a simple algorithm constructing a small TDD computing the same Boolean function of a bounded treewidth Boolean circuit. It was already known that bounded treewidth circuits can be efficiently compiled into deterministic DNNF circuits, either via a direct algorithm [Ama+20], or via a treewidth preserving Tseitin transform into CNF followed by an existential projection of functionally defined variables in a deterministic DNNF circuit, which can be done without increase in the size of the circuit and without losing determinism, since they are functionally defined by other variables, see [MW20, Lemma 18]. The latter approach cannot be found explicitly in any published work to the best of my knowledge, see this Stackoverflow post [htt] based on a discussion with Stefan Mengel. Another approach for compiling bounded treewidth circuit was proposed in [BS17]. Interestingly, this is also based on the relation between factor-width and circuit treewidth but the relation proven in this paper is double-exponential. We improve it to a single exponential in Theorem 2.40.

To explain our approach, we need a few definitions first. Given a partial assignment $\tau \in 2^Y$ for some $Y \subseteq X$ and a gate v of C, we define *the value of v under τ*, denoted by $val(v, \tau)$, as a value in $\{0, 1, \bot\}$. Intuitively, $val(v, \tau)$ is the value of v under the partial assignment, if it can be computed. Otherwise, $val(v, \tau) = \bot$, meaning that its value is undefined yet. A gate may have undefined inputs and still have a fixed value: for example, it is enough for a $\wedge$-gate v to have one input with value 0 to set the value of v to 0. More formally, we define it inductively as follows: if v is an input gate labeled by $y \in Y$, then $val(v, \tau) = \tau(y)$. If $y \notin Y$, then $val(v, \tau) = \bot$. If v is a negation gate with input w and $val(w, \tau) = \bot$, then $val(v, \tau) = \bot$. Otherwise, $val(v, \tau) = 1 - val(w, \tau)$. If v is a $\vee$-gate with input $w_1, \dots, w_k$ then: $val(v, \tau) = 1$ if $val(w_i, \tau) = 1$ for some $i \leq k$, $val(v, \tau) = 0$, if $val(w_i, \tau) = 0$ for every $i \leq k$, and $val(v, \tau) = \bot$ otherwise. If v is a $\wedge$-gate with input $w_1, \dots, w_k$ then: $val(v, \tau) = 0$ if

$val(w_i, \tau) = 0$ for some $i \leq k$, $val(v, \tau) = 1$, if $val(w_i, \tau) = 1$ for every $i \leq k$, and $val(v, \tau) = \bot$ otherwise.

Theorem 2.40. *Let C be a Boolean circuit on variables X computing f_C such that the treewidth of its underlying graph is k. We have $\mathsf{fw}(f_C) \leq 3^{k+2}$.*

Proof. Since the treewidth of C is k, there must exist a tree decomposition of C where each bag has size at most $k+1$. We let $\mathcal{T}$ be this optimal tree decomposition where we have added the output o of C, to every bag. It is still a tree decomposition of C of width $k+1$. Given a node t of $\mathcal{T}$, we denote by $X_{\leq t}$ the set of variables x such that the input gate labeled by x of C appears in a bag below t in $\mathcal{T}$. We claim that for every t, f_C has at most 3^{k+2} distinct $X_{\leq t}$-subfunctions. Indeed, given $\tau \in 2^{X_{\leq t}}$, let b_τ be the function mapping $v \in B_t$ to $val(v, \tau)$. We claim that for $\tau, \nu \in 2^{X_t}$, if $b_\tau = b_\nu$, then τ and ν define the same $X_{\leq t}$-subfunction. Since there are at most $|\{0, 1, \bot\}|^{|B_t|} \leq 3^{k+2}$ possible functions from B_t to $\{0, 1, \bot\}$, it will be enough to prove the desired result.

We prove $b_\tau = b_\nu$ implies that $f_C[\tau] = f_C[\nu]$. Assume toward a contradiction that these functions are different. It means that there exists some $\sigma \in 2^{X \setminus X_t}$ such that $f_C(\tau \times \sigma) \neq f_C(\nu \times \sigma)$. Recall that o denotes the output of C and that by construction, $o \in B_t$. Since we have $val(o, \tau) = val(o, \nu)$ by assumption, we must have $val(o, \tau) = val(o, \nu) = \bot$, otherwise, we would have $val(o, \tau \times \sigma) = val(o, \nu \times \sigma)$, that is $f_C(\tau \times \sigma) = f_C(\nu \times \sigma)$.

Now, it follows that there is at least one g of C such that:

1. $g \in B_t$,
2. $val(g, \tau \times \sigma) \neq val(g, \nu \times \sigma)$ (in particular $val(g, \tau) = val(g, \nu) = \bot$).

Consider a deepest gate v_0 verifying this, that is, $v_0 \in B_t$, $val(v_0, \tau \times \sigma) \neq val(v_0, \nu \times \sigma)$ and no gate below v_0 verifies these two properties. Observe that v_0 cannot be an input gate, otherwise it would be labeled with some variable $x \in X_{\leq t}$ and we would have $\tau(x) = \nu(x)$ since $v_0 \in B_t$, contradicting the fact that $val(v_0, \tau \times \sigma) \neq val(v_0, \nu \times \sigma)$. Now, since $val(v_0, \tau \times \sigma) \neq val(v_0, \nu \times \sigma)$, v_0 must have at least one input v_1 such that $val(v_1, \tau \times \sigma) \neq val(v_1, \nu \times \sigma)$ and $val(v_1, \tau) = \bot$ or $val(v_1, \nu) = \bot$. Indeed, if every input v of v_0 verifies $val(v, \tau \times \sigma) = val(v, \nu \times \sigma)$ then it would contradict the fact that v_0 verifies Item 2. Now let v be an input of v_0 such that $val(v, \tau \times \sigma) \neq val(v, \nu \times \sigma)$ and assume $val(v, \tau) \in \{0, 1\}$ and $val(v, \nu) \in \{0, 1\}$. In this case, we cannot have $val(v, \tau) = val(v, \nu)$. Without loss of generality, assume $val(v, \tau) = 0$ and $val(v, \nu) = 1$. Now, if v_0 is a $\vee$-gate, then $val(v_0, \nu) = 1$, if v_0 is a $\wedge$-gate, then $val(v_0, \tau) = 0$ and if v_0 is a $\neg$-gate, $val(v_0, \tau) = 1$. In any case, it contradicts that $val(v_0, \nu) = val(v_0, \tau) = \bot$.[3]

To sum up, we have shown that there exists an input v_1 of v_0 such that $val(v_1, \tau \times \sigma) \neq val(v_1, \nu \times \sigma)$ and either $val(v_1, \tau) = \bot$ or $val(v_1, \nu) = \bot$. Wlog, we now assume that $val(v_1, \tau) = \bot$. We now construct two paths from v_1 to an input of C:

[3]Observe here that this argument is not valid if we allow the circuit to use other gates such as $\oplus$-gates. We crucially rely on the fact that for any gate g, there is a Boolean value b such that fixing the value of one input of g to b is enough to fix the value of g. We leave open the question of bounding the factor-width of Boolean circuits using other gates than $\{\wedge, \vee, \neg\}$ and unbounded fan-in.

- The first one is built on the following observation: v_1 must have at least one input v_2 such that $val(v_2, \tau \times \sigma) \neq val(v_2, \nu \times \sigma)$, otherwise, we would have $val(v_1, \tau \times \sigma) = val(v_1, \nu \times \sigma)$. By iterating, we build a path $P_1 = (v_1, \ldots, v_p)$ to some input v_p of the circuit such that for every $i \leq p$, $val(v_i, \tau \times \sigma) \neq val(v_i, \nu \times \sigma)$. In particular, v_p is labeled with some variable $x \in X_{\leq t}$ such that $\nu(x) \neq \tau(x)$.
- The second one is built on the following observation: v_1 must have at least one input w_2 such that $val(w_2, \tau) = \bot$. Otherwise, $val(v_1, \tau)$ would take a value in $\{0, 1\}$. By iterating on this observation, we build a path $P_2 = (v_1, w_2, \ldots, w_q)$ to some input w_q of the circuit such that for every $i \leq q$, $val(w_i, \tau) = \bot$. In particular, w_q is labeled with some variable $y \in X \setminus X_{\leq t}$, since it is not assigned by τ.

Finally, consider the path P from v_p to w_q obtained by concatenating P_1 with the reverse of P_2. It only contains gates below v_1 by definition, hence below v_0 in particular. Moreover, v_p being labeled by some $x \in X_{\leq t}$ with $\tau(x) \neq \nu(x)$, it must appear strictly below t in $\mathcal{T}$. And since w_q is labeled by some $y \notin X_{\leq t}$, it does not appear below t. Hence P must contain a gate g in B_t by connectedness of $\mathcal{T}$. Since $val(g, g) = val(g, \nu)$ by assumption, if g is on path P_1, then $val(g, \tau) = val(g, \nu) = \bot$, otherwise, we cannot have $val(g, \tau \times \sigma) \neq val(g, \nu \times \sigma)$. If g is on path P_2, then by construction, we have $val(g, \tau) = \bot$. And since $g \in B_t$, we must also have $val(g, \nu) = \bot$. Recall that g appears in the subcircuit rooted in v_0 by construction. But then, it contradicts the fact that v_0 is a deepest gate satisfying Items 1 and 2 and we must have $f_C[\tau] = f_C[\nu]$.

We obtain the corresponding vtree for f_C from $\mathcal{T}$ inducing the same partitions as $(X_{\leq t}, X \setminus X_{\leq t})$ following the same construction as Theorem 2.37. □

Now given a Boolean circuit C, we say that C' is a subcircuit of C if it is a well-formed connected Boolean circuit containing a subset of edges and gates of C. Given a Boolean circuit C and a subcircuit C' of C, a tree decomposition of C is also a tree decomposition of C'. Hence, if T is the vtree constructed for C in Theorem 2.40 such that $\mathsf{fw}(f_C, T) \leq 3^{k+2}$, then for any connected subcircuit C' of C, we have $\mathsf{fw}(f_{C'}, T) \leq 3^{k+2}$, where $f_{C'}$ is the Boolean function computed by C' (which is well defined since C' is connected). In other words, we have:

Theorem 2.41. *Let C be a Boolean circuit on variables X computing f_C such that the treewidth of its underlying graph is k. There exists a vtree T such that for every connected subcircuit C' of C, we have $\mathsf{fw}(f_{C'}, T) \leq 3^{k+2}$ and T can be computed in polynomial time from a tree decomposition of C of treewidth k.*

Now, this gives a straightforward FPT algorithm to construct a TDD computing f_C respecting a vtree T as given in Theorem 2.41. We visit the gates of C from the inputs to the output and for each gate g, we construct and cache a canonical and full TDD computing f_g. This is easy to do in time $O(|X|)$ for the inputs of C. Now, given a $\wedge$-gate g of C with input $g_1, \ldots, g_k$, assume we have built minimal TDDs $T_{g_1}, \ldots, T_{g_k}$ for $f_{g_1}, \ldots, f_{g_k}$. We build a TDD T_g for f_g by setting $T_1 = T_{g_1}$ and compute T_{i+1} as $T_{i+1} = MINIMIZE(APPLY(T_i, T_{g_{i+1}}, \wedge))$ with the algorithm from Theorem 2.15. For a $\neg$-gate g with input g_1, we simply compute T_g as $\neg T_{g_1}$ in constant time by changing the output in the full TDD g. If g is a $\vee$-gate with

input $g_1, \dots, g_k$, we proceed as for the $\wedge$-case after negating each TDD for T_{g_i} to get a full and canonical TDD computing $\bigwedge_{i=1}^{k} \neg f_{g_i}$ that we negate once more to have T_g computing $\bigvee_{i=1}^{k} f_{g_i}$. In the end of the algorithm, we return T_o where o is the output of C.

By Theorem 2.41 and Theorem 2.22, each intermediate TDD T_g constructed has width at most 3^{k+2} where k is the treewidth of C since they compute Boolean functions of a subcircuit of C. We do a constant number of apply operations per edge of C. Each apply and minimization takes time $O(\text{poly}(3^{k+2})|X|) = 2^{O(k)}|X|$. Hence we have built a full TDD computing f_C in time $2^{O(k)}|X| \cdot |C|$. Since computing optimal tree decompositions can be done in FPT linear time [Bod93], we have:

Theorem 2.42. *Given a Boolean circuit C of treewidth k, we can compute a vtree T and a TDD respecting T computing f_C in time $2^{O(k)}|X| \cdot |C|$.*

As for Theorem 2.38, the construction from Theorem 2.42 is not as good as the compilation to deterministic DNNF circuits proposed in [Ama+20] but we feel the algorithm is more elementary and gives insights on why such circuits are tractable. It would be interesting, in the spirit of Open question 5, to have a more refined algorithm to compile bounded treewidth circuits exploiting the flexibility of TDD while ensuring linear dependency in $|C|$.

We conclude this section with an open question. The proof of Theorem 2.40 does not work if we allow the Boolean circuit to use $\oplus$-gates, but it is not clear whether it is a problem with the proof or with the result itself:

Open question 6. *Given a Boolean circuit C with gates in $\{\vee, \wedge, \neg, \oplus\}$ of size N and bounded treewidth k, is $\mathsf{fw}(f_C)$ bounded by $h(k) \cdot \text{poly}(N)$ for some computable function $h \colon \mathbb{N} \to \mathbb{N}$?*

We leave another question open here. Treewidth is not the most general parameter for which we can build polynomial size deterministic DNNF circuits using a dynamic programming algorithm. CNF formula of bounded MIM-width for example may have unbounded treewidth but have polynomial size deterministic DNNF circuits [Bov+15; STV14]. That said, it is not clear whether they have bounded factor-width. Such a result would allow to show that Algorithm 1 works in polynomial time on bounded MIM-width instances, simplifying the more involved algorithm from the literature.

Open question 7. *Let F be a CNF formula and T a branch decomposition of $\mathsf{Inc}(F)$ of MIM-width k. Do we have $\mathsf{fw}(F) \leq f(k)$ for some computable function $f \colon \mathbb{N} \to \mathbb{N}$?*

2.3 Compiling Structured Quantified CNF formulas

We now show how the notion of width of TDDs can be used to recover and generalize a result by Hubie Chen [Che04] establishing that QBF is FPT when parametrized by treewidth. The TDD data structure introduced in the previous section allows us to recover this result in an interesting way. We show that we can transform any non-deterministic TDD of width k into a TDD of width 2^k. We call such an operation *determinization*. Since existential projection of a block of variables can be done without an increase in size on non-deterministic TDDs, we can alternate existential projection and determinization to build a (deterministic) TDD

representing $Q_1X_1 \dots Q_pX_p.F$ whose width is $\mathsf{tower}_2(w,p)$ from a TDD representation of F of width w, where $\mathsf{tower}_2(w,0)$ is defined as w and $\mathsf{tower}_2(w,i+1) = 2^{\mathsf{tower}_2(w,i)}$.

2.3.1 Determinizing nTDD

In this section, we show that nTDD can be made deterministic by paying an exponential blow-up in their width. The existence of such a (deterministic) TDD can be seen through subfunctions. Indeed, fix a vtree T over variables X and an nTDD C respecting T. For every node t of T, we let N_t be the set of t-nodes of C. Recall that by assumption, $|N_t| \leq k$, where k is the width of C. Given $S \subseteq N_t$, we let f_S^t be the Boolean function whose models are the assignments $\tau \in 2^{X_t}$ such that:

- for every $v \in S$, τ is a model of v,
- for every $v \in N_t \setminus S$, τ is not a model of v.

Given an assignment $\tau \in 2^{X_t}$, we can define the *t-shape $S_t(\tau)$ of τ* (simply called *shape of τ* when t is clear from context) as the set $S_t(\tau) \subseteq N_t$ containing every $g \in N_t$ such that τ is a model of g. Clearly τ is a model of f_S^t if and only if $S = S_t(\tau)$. Moreover, it is not hard to prove by adapting the proof of Lemma 2.20 that if $S(\tau_1) = S(\tau_2)$, then $f[\tau_1]$ and $f[\tau_2]$ define the same X_t-subfunction. This hints at the fact that C cannot have more than $2^{|N_t|} \leq 2^k$ X_t-subfunctions and that they can be defined (possibly with redundancies) by the f_S^t functions. Hence, since nTDD are complete and by Theorem 2.22, there must exist a (deterministic) TDD C' computing the same function as C and of width at most 2^k.

The previous sketch does not, however, give a way of actually constructing the TDD for C. Below, we give a constructive proof allowing to efficiently construct C' from C. Before going into the main proof, we do a few key observations. For a set $S \subseteq N_t$, if there exists some $\tau \in 2^{X_t}$ such that $S = S^t(\tau)$, we say that S is a t-shape of C (without the need of explicitly referring to τ anymore). Let t be an internal node of T with children t_1, t_2 and let S_1 be a t_1-shape and S_2 be a t_2-shape. We say that (S_1, S_2) *generates t-shape S* if there exists $\tau_1 \in 2^{X_{t_1}}$ and $\tau_2 \in 2^{X_{t_2}}$ such $S^{t_1}(\tau_1) = S_1, S^{t_2}(\tau_2) = S_2$ and $S^t(\tau_1 \times \tau_2) = S$. The following lemma shows that (S_1, S_2) generates exactly one t-shape, and this shape is easy to compute from (S_1, S_2):

Lemma 2.43. *For every t_1-shape S_1 and t_2-shape S_2, there exists a unique t-shape S that is generated by S_1 and S_2 defined as $S := \{g \in N_t \mid \exists (g_1, g_2) \in E(g)$ such that $g_1 \in S_1, g_2 \in S_2\}$.*

Proof. Let $\tau_1 \in 2^{X_{t_1}}$ with t_1-shape S_1 and $\tau_2 \in 2^{X_{t_2}}$ with t_2-shape S_2 and let S' be the t-shape of $\tau := \tau_1 \times \tau_2$. We want to show that $S' = S$. Let $g \in S$ and let $(g_1, g_2) \in E(g)$ with $g_1 \in S_1, g_2 \in S_2$ which exists by definition of S. Since g_1 is in S_1 and τ_1 has shape S_1, τ_1 is a model of g_1. Similarly, τ_2 is a model of g_2. Hence, τ is a model of g, that is, $g \in S'$.

Now let $g \in S'$. By definition, τ is a model of g, that is, there exists $(g_1, g_2) \in E(g)$ such that τ_1 is a model of g_1 and τ_2 is a model of g_2. But it then means that $g_1 \in S_1$ and $g_2 \in S_2$ since S_1 and S_2 are the shapes of τ_1 and τ_2 respectively. In other words, $g \in S$ and we have proven $S' = S$. ☐

The next lemma follows directly from definitions and from Lemma 2.43 but it nicely wraps up what will be needed later.

Lemma 2.44. *Let $\tau \in 2^{X_t}$ be an assignment with t-shape S. Let $\tau_1 := \tau|_{X_{t_1}}$, $\tau_2 := \tau|_{X_{t_2}}$, and let S_1, S_2 be their respective shapes. Then (S_1, S_2) generates S. Moreover, if (S_1, S_2) generates S then for every $\tau_1 \in 2^{X_{t_1}}$ with shape S_1 and $\tau_2 \in 2^{X_{t_2}}$ with shape S_2, $\tau_1 \times \tau_2$ has shape S.*

Proof. The first part follows directly from the definition of (S_1, S_2) generates S since $\tau = \tau_1 \times \tau_2$. The second part follows from Lemma 2.43. Indeed, S is the unique t-shape generated by (S_1, S_2) and hence $\tau_1 \times \tau_2$ has shape S by definition. □

We are now ready to prove the main theorem of this section:

Theorem 2.45. *Let T be a vtree over variables X. Let C be an nTDD of width k. One can construct a full TDD C' of width at most 2^k computing the same function as C in time $^{O(k)} \cdot |C|$.*

Proof. As before we let N_t be the set of t-nodes and for $S \subseteq N_t$, we let f_S be the function whose models are assignments whose shape is exactly S. We build C' inductively in a bottom-up fashion. We maintain the following invariant: for every node t of T and $S \subseteq N_t$ such that $f_S \neq \emptyset$, C' contains a t-node v_S^t computing f_S.

If t is a leaf of T labeled by x, then there are exactly two possible assignments: the one mapping x to 1 and the other mapping x to 0. If both assignments satisfy the same set $S \subseteq N_t$ of t-nodes, it means that N_t only contains inputs labeled by 1 or 0. Hence we have exactly one v_S^t node labeled by constant 1. Otherwise, we have two possible sets $S_0, S_1 \subseteq N_t$, the one containing every t-node satisfied by $\tau_0 := \langle x/0 \rangle$ and the one containing every t-node satisfied by $\tau_1 := \langle x/1 \rangle$, that is, S_1 is the set containing every t-node labeled by 1 or x and S_0 is the set containing every t-node labeled by 1 or $\neg x$. We hence have two t-nodes $v_{S_0}^t$ and $v_{S_1}^t$ labeled by $\neg x$ and x respectively. Observe that it is compatible with the determinism conditions on inputs.

We now proceed by induction to show how to build the gate v_S^t for an internal node t of T and $S \subseteq N_t$. We define $E(v_S^t) := \{(v_{S_1}^{t_1}, v_{S_2}^{t_2}) \mid (S_1, S_2) \text{ generates } S\}$. By induction, every model of v_S^t is of the form $\tau_1 \times \tau_2$ where τ_1 has shape S_1, τ_2 has shape S_2 and (S_1, S_2) generates S. Hence by Lemma 2.44, $\tau_1 \times \tau_2$ has shape S. Moreover, if τ has shape S then by Lemma 2.44 again, $\tau = \tau_1 \times \tau_2$ where there exists some shapes (S_1, S_2) generating S and τ_1 has shape S_1, τ_2 has shape S_2. In other words, the models of v_S^t are exactly the set of assignments of 2^{X_t} of shape S, that is, v_S^t computes f_S.

We now argue that we can actually build $E(v_S^t)$ efficiently for every t-shape S. To do so, we can enumerate every t_1-shape S_1 and every t_2-shape S_2. This can be done because we have computed them inductively. Now, we can compute the t-shape S generated by S_1, S_2 using Lemma 2.43 by going every node g in N_t and checking whether $E(g)$ contains a pair (g_1, g_2) with $g_1 \in S_1, g_2 \in S_2$. This can be done in time $O(k^3)$ because N_t contains at most k node. Moreover, each t-node contains at most k^2 pairs since N_{t_1} and N_{t_2} also have size bounded by k. Now, we simply add $(v_{S_1}^{t_1}, v_{S_2}^{t_2})$ as an input of v_S^t.

Since there are at most 2^k possible S_1 and 2^k possible S_2, the construction of every t-node of C' takes time $O(k^3 2^{2k}) = 2^{O(k)}$ and we have to compute this for every internal node t of T. We hence have a total time of $|C| \times 2^{O(k)}$.

We argue that C' is deterministic. The inputs of C' are clearly deterministic since by construction, we cannot have a 1-labeled gate and literals, hence C' respects this constraint. Now, let t be an internal node of T with children t_1, t_2. Let S_1 be a t_1-shape and S_2 be a t_2-shape and let S be the unique shape they generate. Hence the pair $(v^{t_1}_{S_1}, v^{t_2}_{S_2})$ will only be used as an input of v^t_S and no other. That is, C' is deterministic.

Finally, we argue that C' is full. Indeed, let $\tau \in 2^{X_t}$. By definition, τ has a unique shape S, which may possibly be empty. By construction, we have a gate v^t_S, hence v^t_S is satisfied by τ. In other words, every $\tau \in 2^{X_t}$ satisfies exactly one t-node of C', that is, C' is full.

□

It may be surprising that the TDD constructed in Theorem 2.45 is full. This is because $S = \emptyset$ is a t-shape for every node t of the vtree. The shape $S = \emptyset$ catches every assignment in 2^{X_t} that does not satisfy any t-node of C, which is exactly what it means for the circuit to be complete. Interestingly, running the algorithm from Theorem 2.45 on a TDD will produce a full TDD in time $O(\mathrm{poly}(k) \cdot |C|)$. Indeed, the only possibles shape for TDDs are singletons or the empty set. The procedure will add the empty set shape whenever it is missing and plug it accordingly with what is below, which is exactly what the procedure from Theorem 2.13 does.

2.3.2 Application to quantified Boolean functions

We now formalize how we can use determinization as a tool for constructing TDD for quantified Boolean function. We first start by giving a few formal definitions. If f is a Boolean function over variables X and $Y \subseteq X$, we define the Boolean function $\exists Y.f$ to be the Boolean function over variables $X \setminus Y$ such that $\tau \in 2^{X \setminus Y}$ is a model of $\exists Y.f$ if and only if there exists $\sigma \in 2^Y$ such that $\tau \times \sigma$ is a model of f. In this case, we say that Y *is existentially projected in* f. Similarly, we define $\forall Y.f$ to be the Boolean function over $X \setminus Y$ such that $\tau \in 2^{X \setminus Y}$ is a model of $\forall Y.f$ if and only if for every $\sigma \in 2^Y$, $\tau \times \sigma$ is a model of f. In this case, we say that Y *is universally projected in* f.

We will often write $Q_1 X_1 \dots Q_p X_p.f$ for $Q_i \in \{\exists, \forall\}$ and $X_i \subseteq X$ to denote the Boolean function over variables $X \setminus \bigcup_{i \leq p} X_i$ obtained by iteratively projecting the variables, universally or existentially depending on the nature of f. We say that $g := Q_1 X_1 \dots Q_p X_p.f$ is a *quantified Boolean function* and that $Q_1 X_1 \dots Q_p X_p$ is the *quantifier prefix of* g. Observe that if $\bigcup_{i \leq p} X_i = X$, that is, every variable of f has been projected, then g either has one model (the only element of $2^{\emptyset}$), in which case, we say it is *satisfiable* or no model, in which case we say it is *unsatisfiable*.

We abuse the prefix $Q_1 X_1 \dots Q_p X_p.f$ notation and use it in front of any representation of a Boolean function. For example, if F is a CNF formula, we write $Q_1 X_1 \dots Q_p X_p.F$ as a representation of the Boolean function $Q_1 X_1 \dots Q_p X_p.f$ where f is the Boolean function represented by F. If C is a Boolean circuit, we use notation $Q_1 X_1 \dots Q_p X_p.C$ in a similar way.

Determinization of nTDDs allows us to prove the following:

Theorem 2.46. *Let T be a vtree over variables X, $Y \subseteq X$ and C be a full TDD respecting T of width k. One can build a TDD C' of width at most 2^k computing $QY.C$ in time $2^{O(k)}|C|$ for $Q \in \{\exists, \forall\}$.*

Proof. We start with the case $Q = \exists$. By Theorem 2.17, we can build an nTDD C_1 computing $\exists Y.C$ of width at most k and size at most $|C|$ in time $O(|C|)$ by simply relabeling every input gate labeled by y or $\neg y$ with 1. This transformation does not preserve determinism however but we can use Theorem 2.45 to build C' from C_1 in time $2^{O(k)}|C_1| \leq 2^{O(k)}|C|$. By construction, C' computes $\exists Y.C$, is deterministic and has width at most 2^k which is what we wanted.

We now do the case $Q = \forall$. Since C is full, there is a gate g in C computing $\neg C$. We let C_0 be the circuit obtained by choosing g as the output of C, which can be done in constant time, and compute a full TDD C_1 of width 2^k computing $\exists Y.C_0$ in time $2^{O(k)}|C|$ as in the previous paragraph. Now, since C_1 is full, there is a gate g' in C_1 computing $\neg C_1 = \neg\exists Y \neg C$ which is exactly the set of models of $\forall Y.C$. Hence, we build C' by changing the output gate of C_1 to g', which is what we wanted. □

The *Quantified Boolean Formula problem* (QBF for short) asks, given a CNF formula F over variables X and a quantifier prefix $Q_1X_1, \dots, Q_pX_p$ whether $Q_1X_1, \dots, Q_pX_p.F$ is satisfiable. Observe that in this case, we can always assume that $X = \bigcup_{i \leq p} X_i$, since $Q_1X_1, \dots, Q_pX_p.F$ is satisfiable if and only if $\exists X_0 Q_1X_1, \dots, Q_pX_p.F$ is satisfiable, where $X_0 = X \setminus \bigcup_{i \leq p} X_i$. We will also be interested in the counting version of QBF, denoted as #QBF, which asks, given a CNF formula and a quantifier prefix $Q_1X_1, \dots, Q_pX_p$, to compute the number of models of $Q_1X_1, \dots, Q_pX_p.F$.

QBF is a notoriously hard problem and was shown early on to be PSPACE-complete by Stockmeyer and Meyer [SM73]. That said, as for SAT, treewidth helps in the case of QBF. Feder and Kolaitis observed that the parametrized tractability of QBF on bounded treewidth instances can be obtained via Courcelle's theorem [FK06], though this approach blurs the exact complexity of the algorithm. Chen gave a relatively simple and straightforward fixed-parameter tractable algorithm [Che04]. Not surprisingly, the complexity has an awful complexity regarding treewidth: it runs in time $\mathsf{tower}(p+1, O(k))\|F\|$. This dependency is however optimal in the sense that under ETH, one cannot hope for better than a tower of exponentials.

We now give yet another proof of the tractability of QBF with bounded treewidth by building a TDD for $Q_1X_1, \dots, Q_pX_p.F$.

Theorem 2.47. *Let F be a CNF formula with n variables, m clauses and incidence treewidth k. Let $Q_1X_1, \dots, Q_pX_p$ be a quantifier prefix. We can build a TDD of width $\mathsf{tower}_2(p+1, k)$ computing $Q_1X_1, \dots, Q_pX_p.F$ in time $2^{O(k)}mn + n \cdot \mathsf{tower}_2(p+1, O(k))$.*

Proof. We start by building a full TDD computing F of width at most 2^k in time $2^{O(k)}mn$, using Theorem 2.39. We then inductively apply the closure from Theorem 2.46. Assume that we have built a full TDD for $Q_{i+1}X_{i+1}, \dots, Q_pX_p.F$ of width $\mathsf{tower}_2(p+1-i, k)$ in time $n \cdot \mathsf{tower}_2(p+1-i, bk)$ for some constant b.

We then compute a full TDD for $Q_iX_i, \dots, Q_pX_p.F$ using Theorem 2.46. The resulting circuit has width at most $2^{\mathsf{tower}_2(p+1-i,k)} = \mathsf{tower}_2(p+2-i, k)$. It can be computed in time

$2^{b\mathsf{tower}_2(p+1-i,k)}\mathsf{tower}_2(p+1-i,k)n$ for some constant b, that is, in time at most $\mathsf{tower}(p+2-i, k(\log^{p+2-i} b))\mathsf{tower}(p+1-i,k) \leq \mathsf{tower}(p+2-i, bk)$. When $i = 0$, we have the desired TDD. The total time to compute it is bounded by $2^{O(k)}mn$ for the first phase and by $\sum_{i=1}^{p} \mathsf{tower}_2(i+1, bk) \leq 2\mathsf{tower}_2(p+1, bk) \leq \mathsf{tower}_2(p+1, O(k))$ as desired. □

Observe that the complexity stated in Theorem 2.47 is not quite linear in $\|F\|$ which is a bit disappointing. This comes from the fact that in this chapter, we decided to use the bottom-up approach to build the TDD for F and this has quadratic complexity. Other, more direct, methods could be used to build the first TDD or a similar data structure in linear time as we do it in [CM19] but the goal of this chapter was to illustrate from scratch the usefulness of TDDs and to rediscover these results through the bottom-up approach. Recall also that we believe that a more careful implementation and analysis of the bottom-up algorithm would give a linear time compilation for bounded treewidth CNF formulas (see Open question 5), but we have not yet settled this question.

The tractability of QBF, but also #QBF, on bounded treewidth CNF, directly follows from Theorem 2.47 and the fact that satisfiability and counting are tractable on TDDs. Finally, we remark that Theorem 2.47 also works when F is a bounded treewidth Boolean circuit since we can build a TDD of size $2^{O(k)}n$ in this case thanks to Theorem 2.42.

2.4 Conclusion

In this chapter, we have introduced a new data structure for representing Boolean functions that offers advantages similar to those of OBDD but can handle bounded treewidth instances. The main advantage of this data structure over the existing ones, such as deterministic DNNF circuits or SDDs is that they can be minimized into a canonical circuit for which the size and width can be easily understood. While SDDs also have a canonical representation, they do not correspond to the minimal representation and the canonical representation may be exponentially larger than the minimal one [VD15]. Our approach allows us to recover known compilation results about bounded treewidth CNF formulas and circuits with a classical bottom-up algorithm, which gives, in our opinion, an interesting and fresh perspective on these results. It is not clear whether TDDs are interesting in practice but we chose this presentation here to show an alternative way to understand results my previous research have contributed to. In particular, even though the results proven in this chapter are slightly weaker than some my previous work, the simplicity of the underlying theory and its unifying approach is appealing. In particular, we recover the compilation algorithm from [Bov+15] as far as treewidth is concerned (though this paper presents a more general algorithm for instances of bounded PS-width), the algorithm from [Ama+20] and the one from [CM19] (though again, this paper presents an algorithm for bounded PS-width).

Chapter 3

Applications to Proof Complexity

The results presented in this chapter arise from a simple but surprisingly little studied question: how can a user verify the output of a #SAT solver? More concretely, if a #SAT solver says that an input CNF formula has 25 models, how can we check that this is indeed correct? There is no reason to be more confident in a #SAT solver returning 25 than in a SAT solver returning UNSAT, yet modern SAT solvers are expected to output an independently verifiable proof that a formula is unsatisfiable. It may seem like a nitpick, but previous misfortunes have shown us that some implementation bugs may lurk undetected for some time in software for solving hard problems and there is no reason the same would not apply to #SAT solvers. To improve the trust a user can have in a tool, a first solution, with a long and beautiful history from logic to software engineering, would be to implement a #SAT solver in a proof assistant such as Rocq or Lean, prove that its behavior is correct, and extract a program implementing the proven specifications. This approach has been used, for example, to develop the CompCert [Ler06] compiler formally guaranteed to output an assembly program that follows the specification of an input C program, and also for SAT solvers [FW20; Bla+18]. This approach, however, has two main shortcomings. First, it is expensive to implement because it would mean developing a #SAT solver from scratch, with significant effort required to prove its behavior correct. Second, a formally verified implementation makes low-level optimizations harder to create, which may hinder the performance of the solver.

Another cheaper yet powerful and more realistic approach in our scenario is to slightly tweak the behavior of the solver so that instead of only providing the result, it also outputs a certificate that allows one to efficiently verify the correctness of the result with a hopefully simpler independent program. This is the approach favored for SAT solvers. In the case of a satisfiable formula, a proof could simply be a satisfying assignment, which a user can then easily and efficiently verify with a program simple enough to be trusted. Things get harder when it comes to convincing the user that a formula is not satisfiable. In this case, most SAT solvers can output a proof of unsatisfiability that can be independently checked by the user. It can be viewed intuitively as a list of facts implied by the original formula, all easy to check from the input formula and previously derived facts, until one can deduce a contradiction. While the list of facts necessary to reach a contradiction may be long, one can check the correctness of the proof in polynomial time in the length of the formula and in the length of the proof.

This approach predates modern SAT solvers and was originally introduced by Cook and Reckhow [CR79] under the name *propositional proof systems* with the original goal of proving (or disproving) that $\mathsf{NP} \neq \mathsf{coNP}$. One of the earliest proof systems to be studied is the *resolution proof system* [DP60]. It is based on the resolution rule that states that if $x \vee C$ and $\neg x \vee D$ are two clauses of a CNF formula F, then F logically entails the clause $C \vee D$. It is easy to see by case analysis, since every model of F has to set the variable x to some Boolean value. If x is set to true, then the model must necessarily satisfy D, while if x is set to false, then the model must satisfy C. It turns out that if a formula is unsatisfiable, then there is a way of deriving the empty clause by simply applying the resolution rule. Since each clause derived via the resolution rule is entailed by the original formula, this can be seen as a proof of unsatisfiability: a step-by-step guide of easy-to-apply rules allowing one to reach a contradiction. It was observed early on that some unsatisfiable CNF formulas do not have short proofs [Hak85; Urq87]. Many proof systems, more powerful than resolution, have been studied since then, for example, extended resolution, cutting planes or Polynomial Calculus [CCT87; CEI96]. Despite being more powerful than resolution, for each of them, one is able to build unsatisfiable CNF having no short proof. However, studying them is helpful to understand why SAT solvers fail on some instances (e.g., because disproving the satisfiability using resolution is hard) and what tricks could be added to improve their performance on such instances. In theory, CDCL SAT solvers are able to produce proofs of unsatisfiability using extended resolution [PD11] but reconstructing this proof from their run would significantly hinder their practical performance. In practice, proof systems tailored to how CDCL solvers work, such as DRAT [WHH14], are used, so that a certificate boils down to logging and trimming some steps performed by the solver which can then be independently verified.

Now, the proof systems used for SAT solvers are not suited for #SAT solvers. Propositional proof systems have been designed to prove unsatisfiability but proving that a formula has exactly k models requires new tools. Moreover, an ideal proof system for #SAT should be close enough to how solvers work so that existing tools can easily be modified in order to output a certificate along with their result, in the same way extended resolution does not fit the way CDCL solvers work and solvers therefore use tailored proof systems such as DRAT. Practical #SAT solvers are either based on a variant of DPLL that explores every branch with a caching system to avoid redoing expensive computations or on using bottom-up compilation algorithms with appropriate data structures as presented in Chapter 2. In the former case, it has been observed by Huang and Darwiche [HD05] that the trace of the algorithm corresponds to a decision-DNNF circuit, while in the latter case, the procedure is explicitly based on constructing a (restricted) d-DNNF circuit corresponding to the input formula. Hence, in both cases, it seems that a good certificate attesting that a formula F has exactly k models could be a d-DNNF circuit C computing F and having k models. Since model counting on d-DNNF circuits is tractable, one could check that the number of models returned by the solver is indeed correct by simply rerunning the counting algorithm on C. However, the user still has to trust the solver that C indeed represents the same Boolean function as F, which is coNP-hard to check. In this chapter, we introduce proof systems for #SAT that are based on “certified” d-DNNF circuits, that is, a d-DNNF circuit for which one can check that it is indeed equivalent to a given CNF formula. This chapter is devoted to exploring this point of

view in detail and comparing different proof systems for #SAT.

Organization of the chapter. This chapter starts by building a naive proof system for #SAT that essentially lists every model of the formula and a proof that no other model exists (see Section 3.1). We then observe that the set of models does not have to be explicitly given as long as we can count them, and explain how tools from knowledge compilation are well suited for this in Section 3.2. We explain why DNNF circuits are not enough and propose a first solution to overcome this challenge in Section 3.3, whose original idea is from [Cap19]. In Section 3.4, we then turn our attention to proof systems which can easily be implemented inside top-down knowledge compilers, trying to mimic what has worked for SAT solvers. We revisit our contribution on certifying D4 [CLM21] and also show how the MICE proof system [FHR22; BHS24] fits into this framework, though it is slightly less general. We finally explore another variation of proof systems based on knowledge compilation in Section 3.5, returning to the original idea from Section 3.3 with an improvement which drastically strengthens the system. We conclude in Section 3.6 by quickly reviewing another very interesting line of work which complements our approach by using propositional proof systems to guarantee the equivalence between DNNF circuits and CNF formulas directly [Bry+25], but in which we have no direct contribution.

Personal contributions covered in this chapter. This chapter revisits our contributions to the field originally published in SAT'19 [Cap19], which was the first paper to define proof systems for #SAT, in light of subsequent results that improved on our ideas [FHR22; BHS24; Bey+24; Bry+25]. In AAAI'21 [CLM21], we presented a proof of concept for these systems by designing a tool to certify the output circuit of the D4 knowledge compiler [LM17], though this chapter will not cover this part of our work in detail.

A remark on the content of this chapter. This chapter was mostly written in August 2025. Section 3.4 contains results that were intended as a new contribution, in particular, the reformulation of MICE as a version of the syntactic proof system (see Corollary 3.24) from [CLM21]. While finalizing this chapter, I contacted Olaf Beyersdorff, Tim Hoffmann, Luc Spachmann and Kaspar Kasche regarding one of their contributions in the field [BHS24] and, seeing similarities between their current line of work and what I was describing in my email, they shared the preprint [BHK25] (not available at that time and to appear at AAAI26) which independently proves the same connection and offers an even deeper analysis of the links between proof systems and #SAT solvers. The terminology and proof systems used in their version are different from the one we created in this chapter but we decided to leave this chapter as it was originally written. We encourage readers interested in this topic to use the terminology from [BHK25] because it will appear in a venue and it contains more results than those in Section 3.4.

3.1 A naive proof system for #SAT

In this section, we formalize what we mean by "proof systems" for #SAT and show a simple but naive way of leveraging any propositional proof system into a proof system for #SAT.

Cook-Reckhow Proof Systems. Cook-Reckhow proof systems [CR79] can be seen as a means to bridge the gap between $\mathcal{P}$ and harder complexity classes. More formally, let Σ, Γ be finite alphabets and $L \subseteq \Sigma^*$ a language. A *Cook-Reckhow proof system* for L is a **surjective polynomial-time computable function** $\mathsf{Check}_L \colon \Gamma^* \to L$. By definition, for every $w \in L$, there exists at least one $c \in \Gamma^*$ such that $\mathsf{Check}_L(c) = w$. We will call such c a *certificate of* w or a *proof of* $w \in L$. The name comes from the following high-level interpretation of Check_L: assume that we have a black box that, given an input w, tells us whether $w \in L$. If the black box also outputs c such that $\mathsf{Check}_L(c) = w$ whenever $w \in L$, then the user can **check** the correctness of the black box on this input by computing $\mathsf{Check}_L(c)$ in polynomial time (in $|c|$).

This definition of Cook-Reckhow proof systems is neat but may sometimes be confusing. An more verbose equivalent definition is that of a computable function $\mathsf{Check}_L \colon \Sigma^* \times \Gamma^* \to \{0, 1\}$ such that:

- Check_L can be computed in polynomial time,
- for every $w \in \Sigma^*$, there exists (at least one) $c \in \Gamma^*$ such that $\mathsf{Check}_L(w, c) = 1$ if and only if $w \in L$.

Propositional Proof Systems. A *propositional proof system* is a Cook-Reckhow proof system for the problem UNSAT, which takes a CNF formula as input and accepts if and only if the CNF formula is unsatisfiable. As mentioned in the introduction, several such proof systems exist. The only one we explicitly use is the *resolution proof system* [DP60], but we can also mention other more powerful propositional proof systems such as cutting planes [CCT87] or Polynomial Calculus [CEI96].

Resolution is based on the resolution rule. Given two clauses $c_1 = \{x\} \cup c_1'$ and $c_2 = \{\neg x\} \cup c_2'$, the resolvent of c_1 and c_2, denoted $res(c_1, c_2)$, is the clause defined as $c_1' \cup c_2'$. Given an unsatisfiable CNF formula F, a resolution refutation ρ of F is a list of clauses $c_1, \dots, c_p$ such that $c_p = \emptyset$ and for every $i \leq p$, either c_i is a clause of F or $c_i = res(c_j, c_k)$ for some $j, k < i$. The size of ρ is defined as the number of clauses in ρ. We say that a clause c_j of ρ is derivable from F. It is not too hard to prove by induction that for every $i \leq p$, $F \models c_i$. Hence, if F has a refutation, it is unsatisfiable because it entails the empty clause. Now, if F is unsatisfiable, it is not too hard to see that a brute-force algorithm iteratively computing every possible resolvent will eventually derive the empty clause. In other words:

Theorem 3.1. *Resolution is a propositional proof system.*

We denote the resolution proof system by Res. We refer the reader to [Kra19] for more details on propositional proof complexity. In this chapter, we will need a few facts about resolution refutations. First, we need to be able to extend clauses in refutations. More precisely, we consider the proof system $\mathsf{Res} + \mathsf{sub}$ as follows: a refutation of F in the $\mathsf{Res} + \mathsf{sub}$

proof system is a list $c_1, \ldots, c_p$ of clauses where $c_p = \emptyset$ and such that for every $i \leq p$, either $c_i \in F$, or $c_i = res(c_j, c_k)$ for $j, k < i$, or $c_i = c_j \vee d$ for some clause d and $j < i$, which we call a subsumption. This is clearly a sound proof system because if $F \models c$ then $F \models c \vee d$ for any clause d. Now, one can easily see that adding this rule does not allow shorter refutations:

Theorem 3.2. *Given a* $\mathsf{Res{+}sub}$ *refutation* ρ *of* F *of size* p*, one can construct in time* $\mathrm{poly}(p)$ *a resolution refutation of* F *of size at most* p*.*

Proof. Let $\rho = c_1, \ldots, c_p$ and let $c_i = c_a \vee d$ for $a < i$ be the last subsumption step in ρ. Let $c_k = res(c_i, c_b)$ be the first step in which c_i is used in the proof after being introduced. Since there are no subsumptions after step i, it is necessarily a resolution step.

If the complementary literal in c_i and c_b is a literal of d, then we can remove c_i from the clause list and replace $c_k = res(c_i, c_b)$ by the subsumption $c_k = c_a \vee d'$ where $d' = res(d, c_b)$. If the complementary literal in c_i and c_b is a literal of c_a, we can remove c_i from the clause list and let $c'_k = res(c_a, c_b)$ and, if $c'_k \neq c_k$, we let $c_k = c'_k \vee (c_k \setminus c'_k)$.

Now, this transformation preserves the fact that ρ is a refutation of size at most p of F. However, the subsumption moves strictly to the right each time. Hence, at some point, $c'_k = c_k$ and we can remove the subsumption. By iteratively applying this transformation, we remove every subsumption rule from ρ. □

Finally, we need to be able to extract from a refutation of F, a smaller refutation of $F[\tau]$ for any partial assignment τ of $\mathsf{var}(F)$. Proof systems with this property are said to be closed under restriction [MS24]. Resolution is known to be closed by restriction:

Theorem 3.3. *Given a resolution refutation* ρ *of* F *of size* p *and a partial assignment* τ *of* $\mathsf{var}(F)$*, we can construct in time* $\mathrm{poly}(p)$ *a resolution refutation of* $F[\tau]$ *of size at most* p*.*

Proof. Replace every clause c in ρ by $c[\tau]$. We claim that this yields a $\mathsf{Res{+}sub}$-refutation of $F[\tau]$. Indeed, if $c_i \in F$ then $c_i[\tau] \in F[\tau]$. Now, if $c_i = res(c_j, c_k)$ and the complementary literal of c_j and c_k is not assigned by τ, we have $c_i[\tau] = res(c_j[\tau], c_k[\tau])$. Otherwise, we have $c_i[\tau] = c_j[\tau] \vee c_k[\tau]$, hence we can obtain $c_i[\tau]$ via subsumption. Now, by Theorem 3.2, we can transform $\rho[\tau]$ into a resolution refutation of $F[\tau]$ of size at most p. □

Proof Systems for #SAT. We have so far defined proof systems for decision problems. For problems such as #SAT, the definition of Cook-Reckhow does not directly apply. We therefore use the usual transformation from functions to decision problems to generalize the definition. Indeed, the problem #SAT can be seen as the problem of deciding, given a CNF formula F and an integer $k \in \mathbb{N}$, whether $\#F = k$. Hence, a proof system for #SAT is a proof system for the decision problem $\{(F, k) \mid F \text{ is a CNF and } \#F = k\}$. In other words, it is a polynomial time computable function Check that takes as input a CNF formula F, an integer k and a proof c. It accepts only if $\#F = k$. Moreover, for every F and k such that $\#F = k$, there exists at least one certificate c such that $\mathsf{Check}(F, k, c)$ accepts.

Table Proof Systems for #SAT. As an illustration, we now explain how to transform any propositional proof system into a proof system for #SAT and give a concrete example using resolution. Given a propositional proof system Check_{prop}, we define the *table proof system (for #SAT) induced by* Check_{prop}, denoted by $\mathsf{Table}(\mathsf{Check}_{prop})$, as follows. For a CNF F with k models $\tau_1, \ldots, \tau_k$, a certificate c_F for F in this proof system is given as:

- The list $[\tau_1, \ldots, \tau_k]$ of its models,
- A certificate c^F_{prop} in Check_{prop} such that $G = F \wedge T_1 \wedge \cdots \wedge T_k$ is unsatisfiable, where T_i is the clause whose only non-satisfying assignment is τ_i (formally, $T_i = \bigvee_{x \in \mathsf{var}(F), \tau_i(x)=0} x \vee \bigvee_{x \in \mathsf{var}(F), \tau_i(x)=1} \neg x$).

To check a certificate in $\mathsf{Table}(\mathsf{Check}_{prop})$, one must check that the list of models contains k elements, check that for every i, $\tau_i \models F$. Finally, build G and check that $\mathsf{Check}_{prop}(G, c^F_{prop}) = 1$. All these steps can be performed in time polynomial in $|c_F|$, in the bit-length of k and in $\|F\|$. Moreover, F has k models if and only if this procedure accepts, because G can only be refuted if $\tau_1, \ldots, \tau_k$ are exactly the set of models of F. Hence, we just described a proof system for #SAT.

Example 3.4. In this example, we use the resolution proof system and the CNF formula F defined by

$$(x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z) \wedge (x \vee \neg y) \wedge (\neg x \vee y).$$

We denote by c_1, c_2, c_3, c_4 the clauses of F in the order above.

It can be checked that $\tau_1 := \langle x/0, y/0, z/1 \rangle$ and $\tau_2 := \langle x/1, y/1, z/0 \rangle$ are the only models of F. A proof of this fact in the proof system $\mathsf{Table}(\mathsf{Res})$ could be given as the list $[\tau_1, \tau_2]$ and the following refutation:

- $c_5 := res(c_1, T_1) = x \vee y$, where $T_1 = x \vee y \vee \neg z$ corresponds to τ_1.
- $c_6 := res(c_2, T_2) = \neg x \vee \neg y$, where $T_2 = \neg x \vee \neg y \vee z$ corresponds to τ_2.
- $c_7 := res(c_3, c_5) = x$.
- $c_8 := res(c_4, c_5) = y$.
- $c_9 := res(c_8, c_6) = \neg x$.
- $c_{10} := res(c_9, c_7) = \emptyset$.

This certificate establishes that F has exactly two models.

The obvious limitation of table proof systems is that the size of any proof for F is bounded below by $\#F$, which can be large even for trivial formulas. Consider, for example, the case of the formula F containing exactly one clause $x_1 \vee \cdots \vee x_n$. It is clear that F has $2^n - 1$ models. Hence, every proof for $\#F$ in table proof systems will have size at least $2^n - 1$, even

with a powerful underlying propositional proof system. In the next section, we show that we can overcome this limitation by giving a more succinct representation of the models of F than its list of models.

3.2 From d-DNNF circuits to #SAT proof systems

In this section, we present proof systems based on the following idea: given a CNF formula F, a certificate for the fact that F has exactly k models is a d-DNNF circuit C such that F and C have the same models. Checking that F has k models, which can be done in polynomial time in the size of the certificate C, hence boils down to checking that C has k models

As it is currently described, this proof system is flawed. Indeed, to correctly check the certificate, one also needs to check that C is indeed a d-DNNF circuit and that it represents the same set of models as F. This is hard in general for the following reason: a CNF F on variables X is unsatisfiable if and only if it represents the Boolean function $\bot_X$, defined as $\bot_X(\tau) = 0$ for every $\tau \in 2^X$. This function has a trivial d-DNNF circuit: a circuit consisting of a single 0-input gate. Hence, checking whether a CNF formula and a d-DNNF circuit represent the same set of models is as hard as checking whether a CNF is unsatisfiable, the standard coNP-complete problem. This observation is formalized below (and is shown for OBDD, a data structure even weaker than d-DNNF circuits):

Proposition 3.5. *Given a CNF formula F and an OBDD C both over variables X, it is coNP-complete to check whether $F \models C$, that is, to check whether every model of F is also a model of C.*

Proof. The problem is obviously in coNP. Indeed, the problem of checking whether F has a model that is not a model of C is in NP since the model can be guessed and then checked. We prove hardness by reducing from the problem UNSAT of deciding whether a CNF formula is unsatisfiable. The reduction is immediate: we reduce F to the problem of checking whether $F \models C$ where C is the OBDD consisting of a single 0-sink. $\square$

One can observe that Proposition 3.5 only establishes coNP-completeness of checking whether every model of F is also a model of C. This is because the converse direction is actually tractable, even for the larger class of DNNF circuits:

Proposition 3.6. *Given a CNF formula F and a DNNF circuit C both over variables X, one can check in polynomial time whether $C \models F$, that is, whether every model of C is also a model of F.*

Proof. For a clause $c \in F$, we denote by τ_c the unique assignment of $\mathsf{var}(c)$ such that τ_c does not satisfy c. We claim that there exists $\tau \in 2^X$ such that $\tau \models C$ and $\tau \not\models F$ if and only if there exists some clause $c \in F$ such that $C[\tau_c]$ has at least one model. Indeed, if such τ exists then by definition it means that there is some clause $c \in F$ such that $\tau \not\models c$, hence $\tau|_{\mathsf{var}(c)} = \tau_c$. Moreover, since $\tau \models C$, $C[\tau_c]$ has at least one model since $\tau|_{X \setminus \mathsf{var}(c)}$ is a model of $C[\tau_c]$. Conversely, assume that there is some clause c such that $C[\tau_c]$ has one model τ'. Let

$\tau = \tau' \cup \tau_c$. Then τ is a model of C by definition but $\tau \not\models F$ because c is not satisfied by τ. Hence, the claim holds.

By contraposition, $C \models F$ if and only if for every $c \in F$, $C[\tau_c]$ is unsatisfiable. Since one can check whether $C[\tau_c]$ has at least one model in time $O(|C|)$ for a given $c \in F$, one can check whether $C \models F$ in time $O(m|C|)$ where m is the number of clauses of F. □

Proposition 3.5 and Proposition 3.6 suggest that the coNP-hardness of checking whether a DNNF circuit is equivalent to a given CNF formula stems from checking $F \models C$. Equivalently, this boils down to checking whether $\neg C \models \neg F$, that is, every assignment not satisfying the DNNF circuit should also violate at least one clause of F. In the next subsections, we explain how the coNP-hardness of this task can be circumvented.

3.3 Certified decision-DNNF

The first non-trivial proof system for #SAT, introduced in [Cap19], is based on the notion of certified decision-DNNF circuits which offers a workaround to the complexity of Proposition 3.5. The idea is to annotate 0-input gates with an explanation of why they contradict a given CNF formula. More formally, a *certified decision-DNNF circuit* C is a decision-DNNF circuit such that every 0-gate g is labeled with a clause c_g.

In this chapter, we use the convention that decision-DNNF circuits are oriented from the output to the sinks, in the same way as FBDDs. A certified decision-DNNF circuit C on variables X is *correct* if for every $\tau \in 2^X$, if there exists a path from the output of C to a 0-gate g of C compatible with τ, then $\tau \models \neg c_g$, that is, $\tau(\ell) = 0$ for every literal ℓ of c_g. We denote by $Z(C)$ the set of 0-gates of a certified decision-DNNF circuit C. We clearly have:

Proposition 3.7. *Let C be a correct certified decision-DNNF circuit. Then $\bigwedge_{g \in Z(C)} c_g \models C$.*

Proof. We prove the contraposition, that is, $\neg C \models \bigvee_{g \in Z(C)} \neg c_g$. Let $\tau \in 2^X$ be such that $\tau \models \neg C$. Since τ does not satisfy C, there must exist a path from the output of C to a 0-gate $h \in Z(C)$ that is compatible with τ. Now since C is correct, $\tau \models \neg c_h$ by definition. Hence, $\tau \models \bigvee_{g \in Z(C)} \neg c_g$. □

A direct consequence of Proposition 3.7 is the following:

Corollary 3.8. *Let F be a CNF formula and C be a correct certified decision-DNNF circuit such that for every $g \in Z(C)$, c_g is a clause of F. Then $F \models C$.*

Proof. It directly follows from the fact that $F \models \bigwedge_{g \in Z(C)} c_g$ since c_g is a clause of F for every $g \in Z(C)$. □

Checking whether a Boolean circuit with decision-gates is a decision-DNNF circuit can be done in polynomial time since one only has to check that the circuit contains no $\vee$-gates or no negations and that each $\wedge$-gate is decomposable. The latter can be done by computing $\mathsf{var}(g)$ for every g bottom-up in time $O(|X| \cdot |C|)$. Now, checking that a Boolean circuit is a *correct* certified decision-DNNF circuit may look harder since correctness is defined semantically.

However, there is an equivalent definition of correctness that is purely syntactic and allows for efficient checking:

Proposition 3.9. *Given a Boolean circuit C with labeled 0-input gates, we can decide in polynomial time whether C is a correct certified decision-DNNF circuit.*

Proof. One first checks that C is a decision-DNNF circuit as explained in the preceding paragraph. Now observe that C is not correct if and only if there exists a 0-input gate g, a literal $\ell \in c_g$ on variable x, an assignment $\tau \in 2^X$ with $\tau(\ell) = 1$ and a path P from the output of C to g compatible with τ. Then either P contains a decision-gate on variable x and P follows the edge labeled by the value compatible with ℓ since it is compatible with τ, or P does not contain a decision-gate on variable x. Thus, if C is not correct, there is a path from the output of C to a 0-input gate g and a literal $\ell \in c_g$ such that P does not contain any edge $g_1 \to g_2$ where g_1 is a decision gate on $\mathsf{var}(\ell)$ and $g_1 \to g_2$ is labeled by the value compatible with $\neg\ell$.

We claim that this is actually an equivalence, that is, if such a path P exists, then C is not correct. Indeed, if such a path exists, consider an assignment τ compatible with P. This assignment exists since each variable is tested at most once on P, and we can simply choose the value of $\tau(x)$ compatible with the corresponding edge in P. For any other variable x, we arbitrarily choose $\tau(x) = 0$. Now, if $\tau(\ell) = 0$, then it means that $\mathsf{var}(\ell)$ does not appear on P because otherwise, by definition, P would contain the edge compatible with ℓ. Hence, we can flip the value of $\tau(\ell)$ so that $\tau(\ell) = 1$ and still have an assignment compatible with P. Which proves that C is not correct.

We hence have a syntactic characterization of correctness: C is not correct if and only if there exists a 0-input gate g, a literal $\ell \in c_g$ and a path P from the output of C to g which does not contain an edge $g_1 \to g_2$ where g_1 is a decision gate on $\mathsf{var}(\ell)$ and $g_1 \to g_2$ is labeled by the value compatible with $\neg\ell$.

The existence of such a path can be checked in polynomial time as follows: for every 0-input gate g and literal $\ell \in c_g$, remove every edge of the form $g_1 \to g_2$ where g_1 is a decision-gate on $\mathsf{var}(\ell)$ and $g_1 \to g_2$ is labeled by the value compatible with $\neg\ell$ then check in polynomial time whether there is a path from g to the output of C in this subgraph. If it does then C is not correct. If no such path exists for every $g \in Z(C)$ and $\ell \in c_g$, then C is correct. □

We are now ready to describe our first proof system for #SAT which we call kcps, for *Knowledge Compilation Proof System.* Given a CNF formula F, a kcps-certificate for F is a pair (C, k) such that:

1. C is a correct certified decision-DNNF circuit.
2. For every $g \in Z(C)$, c_g is a clause of F.
3. $C \models F$
4. $k = \#\{\tau \in 2^{\mathsf{var}(F)} \mid \tau \models C\}$.

The following establishes that kcps is indeed a proof system for #SAT:

Theorem 3.10. *Given a CNF formula F, there exists a* kcps*-certificate (C, k) for F if and only if $\#F = k$. Moreover, given (C, k), one can check in time polynomial in $|C| + \|F\|$ whether (C, k) is a* kcps*-certificate for F.*

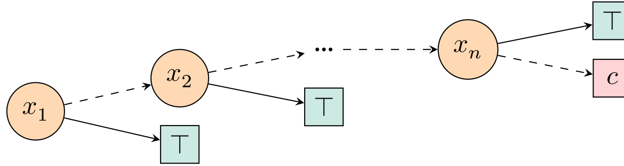
Proof. First, observe that if F has a kcps-certificate (C, k), then $F \Leftrightarrow C$. Indeed, one direction is ensured by Item 3 of the definition. The other direction follows from Item 1 and Item 2 which ensure $F \models C$ by Corollary 3.8. Now, since $F \Leftrightarrow C$, F and C have the same set of models over $\mathsf{var}(F)$, hence, $\#F = k$ by Item 4.

Conversely, let $k = \#F$ and let T be a full decision tree testing every variable of F. Each leaf is either labeled by 1 if the path to this leaf corresponds to a satisfying assignment of F and otherwise by a clause c that is falsified by the corresponding assignment. It is easy to check that (k, T) is a kcps-certificate for F (which has size exponential in $\|F\|$).

It remains to show that one can check in polynomial time whether a given pair (C, k) is a kcps-certificate for F. Item 1 can be verified in polynomial time by Proposition 3.9, Item 2 is trivial to check, Item 3 can be checked in polynomial time by Proposition 3.6 and finally Item 4 can be checked in polynomial time by computing $\#C$ using the counting algorithm described in Chapter 1. □

We now illustrate the kcps-proof system on two examples:

Example 3.11. Consider again the CNF formula F with one clause $c := x_1 \vee \cdots \vee x_n$, which admits no polynomial-size certificate in the Table(Check) proof system for any propositional proof system Check. We give a polynomial size kcps-certificate for F as $(C, 2^n - 1)$ where C is the following correct certified decision-DNNF circuit:



Example 3.12. We consider here a more involved example. Consider the CNF formula F defined as the conjunction of:

- $c_0 := \neg x \vee \neg y_1 \vee y_2$
- $c_1 := \neg x \vee y_1 \vee \neg y_2$
- $c_2 := \neg x \vee \neg z_1 \vee z_2$
- $c_3 := \neg x \vee z_1 \vee \neg z_2$
- $c_4 := \neg x \vee \neg z_2 \vee z_3$

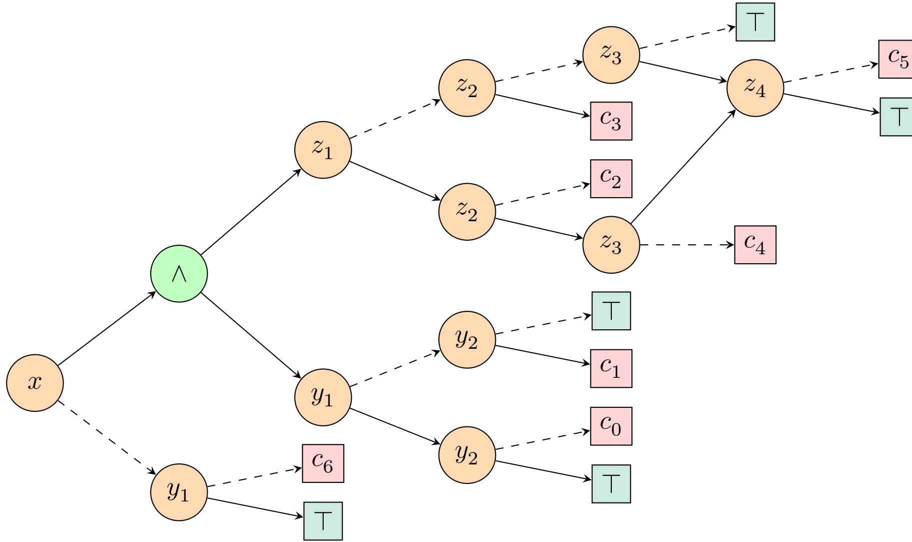


Figure 3.1: A correct certified decision-DNNF circuit for Example 3.12.

- $c_5 := \neg x \vee \neg z_3 \vee z_4$
- $c_6 := x \vee y_1$

We let C be the certified decision-DNNF circuit from Fig. 3.1. One can easily check that it is correct, most 0-input gates having at most one path leading to it, so Item 1 is satisfied. Moreover, every clause used for labeling 0-input gates in C is a clause of F. Hence, we have $F \models C$, so Item 2 is satisfied. We can also check that $C \models F$ by applying Proposition 3.6. For example, we can plug $\neg c_0 = x \wedge y_1 \wedge \neg y_2$ into the circuit and observe that the resulting decision-DNNF circuit has no models. It means that Item 3 is satisfied. Finally, one can check that C has 40 models over $\{x, y_1, y_2, z_1, z_2, z_3, z_4\}$: 32 models mapping x to 0 and 8 models mapping x to 1 (because there are two models of y_1, y_2 on one side of the decomposable $\wedge$-gate and 4 models on the other side: namely, the model $z_1 = z_2 = z_3 = z_4 = 1$ and three models where $z_1 = z_2 = 0$). Therefore, $(C, 40)$ is a certificate for F, and we can conclude that $\#F = 40$.

A direct consequence of Example 3.11 is that kcps can have exponentially more succinct proofs than Table(Check) for any propositional proof system Check, which happens mainly because F has an exponential number of models and hence, any Table(Check) certificate for F would need to list them explicitly.

3.4 Syntactic caching and top-down solver

While certified decision-DNNF circuits allow for efficiently checking that $F \models C$, it does not exactly match the way top-down #SAT solvers such as D4 work. These solvers are based on

the following inductive algorithm to compute $\#_X F$, the number of assignments in 2^X that satisfy F ($\mathsf{var}(F) \subseteq X$):

- If F contains no clause, return $\#_X F = 2^{|X|}$,
- If F contains the empty clause, return 0,
- If $F = F_1 \wedge F_2$ with $\mathsf{var}(F_1) \cap \mathsf{var}(F_2) = \emptyset$, return $\#_{X \setminus X_2} F_1 \times \#_{X_2} F_2$ where $X_2 = \mathsf{var}(F_2)$,
- Otherwise, choose $x \in X$ and return $\#_{X \setminus \{x\}} F[x/0] + \#_{X \setminus \{x\}} F[x/1]$. The choice is made following a heuristic, but some choices are of course more sensible than others, for example, if there is a unit clause x in the formula, then we know that $\#F[x/0] = 0$, and we only have to explore one branch.

The previous algorithm is improved via two main optimizations. The first one, that we will call *syntactic caching*, avoids redoing expensive computations by using the following syntactic caching mechanism: each time a recursive call $\#_Y G$ terminates, the value $\#_{\mathsf{var}(G)} G = \frac{\#_Y G}{2^{Y \setminus \mathsf{var}(G)}}$ is cached with G as key. Upon a recursive call $\#_Z H$, a lookup is made in the cache to check whether $\#_{\mathsf{var}(H)} H$ has already been computed. If this is the case, then we can directly return $2^{Z \setminus \mathsf{var}(H)} \cdot \#_{\mathsf{var}(H)} H$. Otherwise, the algorithm proceeds as described above.

The second optimization made by most top-down solvers is to run a SAT solver on G before trying to compute $\#_Y G$. If the SAT solver returns UNSAT, then we know that $\#_Y G = 0$ and we can directly return the value. Otherwise, the recursive call proceeds as described above. This optimization is interesting for two reasons: first, it leverages the efficiency of existing SAT solvers to avoid spending time on uninteresting branches. Second, if done correctly, each oracle call can give more insights than the simple fact that G is satisfiable or not. Indeed, SAT solvers based on the *Conflict Driven Clause Learning* algorithm (CDCL-solver) can learn clauses implied by the formula. These clauses can be kept to detect conflicts earlier or unit propagation steps.

It is not hard to see, as observed in [HD05], that the trace of the previously sketched top-down algorithm, even with both optimizations, is a decision-DNNF circuit. Without syntactic caching, the trace is a decision-DNNF circuit whose underlying graph isg a tree (since in this case, the output of each gate is used exactly once). The second optimization yields a smaller circuit by directly detecting unsatisfiable formulas, represented as a 0-input gate in the circuit, instead of a possibly large circuit with 0 models.

Certified decision-DNNF circuits are however not powerful enough to account for any of these optimizations. It is easy to see that for the second optimization: upon calling a top-down solver on an unsatisfiable formula F, the second optimization will directly terminate the computation, and a decision-DNNF circuit containing only a 0-input gate will be returned. In a certified decision-DNNF circuit, the proof of unsatisfiability of F has to be somehow hard-coded in the circuit using the original clauses of F. An easy fix, that is discussed in detail in Section 3.5, is to use clauses implied by F in the labels of the certified decision-DNNF circuit, as long as we have a proof that such clauses are indeed implied by F. As we will see in Section 3.5, this approach is surprisingly very powerful.

(a) A decision-DNNF circuit produced by a top-down solver with syntactic caching.

(b) A correct certified decision-DNNF circuit, preventing caching.

Figure 3.2: Decision-DNNF circuits for $(x \vee \neg x_1 \vee \neg x_2) \wedge (\neg x \vee \neg x_1 \vee \neg x_2)$.

In this section, we focus on the limitations of certified decision-DNNF circuits regarding syntactic caching. To see why certified decision-DNNF circuits are not powerful enough, we consider the following example.

Example 3.13. Let F be the CNF formula $F = (x \vee \neg x_1 \vee \neg x_2) \wedge (\neg x \vee \neg x_1 \vee \neg x_2)$. We let $c_0 = \neg x \vee \neg x_1 \vee \neg x_2$ and $c_1 = x \vee \neg x_1 \vee \neg x_2$. Any top-down #SAT solver picking x as the first branching variable would build the decision-DNNF circuit depicted on Fig. 3.2a. This is because $F[x/0]$ and $F[x/1]$ are syntactically the same CNF formula $\neg x_1 \vee \neg x_2$. This equivalence would be detected and caching would happen on the second recursive call. Now, observe that the decision-DNNF circuit from Fig. 3.2a cannot be certified with clauses from F. Indeed, if we label the only 0-input gate of the circuit with c_1, then the path starting with the edge labeled by $\neg x$ does not contradict c_1. A similar phenomenon happens if we label the 0-gate with c_0. To build a correct certified decision-DNNF circuit for F, one would hence need to duplicate the circuit computing $\neg x_1 \vee \neg x_2$ as depicted on Fig. 3.2b.

3.4.1 Syntactic entailment

This limitation of certified decision-DNNF circuits regarding syntactic caching has originally been observed in [CLM21] where we proposed a certified version CD4 of the D4 knowledge compiler. The CD4 tool builds on the observation that the decision-DNNF circuits output by top-down solvers use only syntactic caching and the consistency of this caching policy can be checked efficiently a posteriori. This led us to the definition of *syntactic equivalence* between a CNF formula and a decision-DNNF circuit: a strengthening of the notion of equivalence which can be checked in polynomial time in the size of the formula and the circuit. The notion we present here is not exactly the same as the one in [CLM21]. First, we only care about syntactic entailment since we know by Proposition 3.6, one way of the equivalence is always easy to check. Also, the original definition was tailored to D4, taking into account some specificities of the tool to store the minimal amount of extra information. We discuss the differences between the definition given here and the one from [CLM21] in Section 3.4.3.

Given a CNF formula F and a subset X of variables of F, the *X-connected component of F* is defined as the set of clauses c of F such that either $\mathsf{var}(c) \cap X \neq \emptyset$ or if there exists $x \in X$ and $y \in \mathsf{var}(c)$ and a path from x to y in the primal graph of F.

Let C be a decision-DNNF circuit. A *syntactic CNF-labeling of C* is a labeling (F_g) of the gates of C such that for every gate g of C, F_g is a CNF formula and:

- If g is a $\wedge$-gate with input $g_1, \dots, g_k$, then for every $i \leq k$, F_{g_i} is the $\mathsf{var}(g_i)$-connected component of F_g.
- If g is a decision-gate on variable x and g_1 is the input of g corresponding to assigning x to $b \in \{0,1\}$, $F_{g_1} = F_g[x/b]$.

A syntactic CNF-labeling of C is said to be *consistent* if for every 0-input gate g of C, F_g is unsatisfiable.

We say that C *syntactically entails* a CNF formula F if there exists a consistent syntactic CNF-labeling of C with output r such that $F_r = F$.

Observe that to be correct, a CNF-labeling has to be consistent with circuit caching. Indeed, if g has two outputs g_1, g_2, then F_g has to be syntactically the same if derived from F_{g_1} or from F_{g_2}. We illustrate this phenomenon on the following example:

Example 3.14. Let F be the CNF-formula $F = (x \vee y_1) \wedge (\neg x \vee y_2) \wedge (y_1 \vee \neg y_2) \wedge (\neg y_1 \vee y_2)$. It is easy to see that F has two models:

- $\langle x/0, y_1/1, y_2/1 \rangle$ and,
- $\langle x/1, y_1/1, y_2/1 \rangle$.

Hence, both decision-DNNF circuits from Fig. 3.3 are actually equivalent to F. However, only the one in Fig. 3.3b is syntactically entailed by F. Indeed, assume there is a CNF-labeling of the circuit from Fig. 3.3a whose output is labeled by F. Then the decision-gate on y_1 has a label F' that must satisfy both $F' = F[x/0]$ and $F' = F[x/1]$ but $F[x/0] = y_1 \wedge (y_1 \vee \neg y_2) \wedge (\neg y_1 \vee y_2)$ while $F[x/1] = y_2 \wedge (y_1 \vee \neg y_2) \wedge (\neg y_1 \vee y_2)$. Hence, F' cannot exist, that is, F does not syntactically entail the circuit. It can be checked however that this does not happen in Fig. 3.3b. In this case, both decision gates on y_1 can be labeled by $F[x/0]$ and $F[x/1]$ respectively, and the decision-gate on y_2 will be labeled with $F[x/0, y_1/1] = F[x/1, y_1/1] = y_2$.

The circuit from Fig. 3.3a is semantically entailed by F but the sharing on variable x is not witnessed syntactically even if it is semantically correct. Observe that if we consider the CNF formula $F' = (y_1 \vee \neg y_2) \wedge (\neg y_1 \vee y_2) \wedge (x \vee y_1) \wedge (\neg x \vee y_1)$ instead of F, then F' now syntactically entails the circuit from Fig. 3.3a because $F'[x/0]$ and $F'[x/1]$ are both equal to $y_1 \wedge (y_1 \vee \neg y_2) \wedge (\neg y_1 \vee y_2)$.

Syntactic entailment is related to entailment:

Lemma 3.15. *Let C be a decision-DNNF circuit and (F_g) be a consistent syntactic CNF-labeling of C. Then for every gate g of C, the set of models of F_g over $\mathsf{var}(F_g)$ is a subset of the models of g over $\mathsf{var}(F_g)$.*

(a) F does not syntactically entail the circuit.

(b) F syntactically entails the circuit.

Figure 3.3: decision-DNNF circuits for $F = (x \vee y_1) \wedge (\neg x \vee y_2) \wedge (y_1 \vee \neg y_2) \wedge (\neg y_1 \vee y_2)$.

Proof. The proof is by bottom-up induction on the circuit. By definition, if g is a constant input, then this is clear: if g is a 0-input gate, then F_g is unsatisfiable and hence both g and F_g have an empty set of models. If g is a 1-input gate, then every model of F_g over $\mathsf{var}(F_g)$ is a model of g since g computes a tautology.

We now proceed by structural induction. If g is a decision-gate on variable x, we let g_b be the input of g labeled with $b \in \{0, 1\}$. By definition, $F_{g_0} = F_g[x/0]$ and $F_{g_1} = F_g[x/1]$. Hence, F_g is equivalent to $(x \wedge F_{g_1}) \vee (\neg x \wedge F_{g_0})$ and $\mathsf{var}(F_g) = \mathsf{var}(F_{g_1}) \cup \mathsf{var}(F_{g_0}) \cup \{x\}$. By induction, the models of F_{g_b} are included in the models g_b on $\mathsf{var}(F_{g_b})$. Hence, the models of F_g on $\mathsf{var}(F_g)$ are included in the model of g over $\mathsf{var}(F_g)$ since the models of g are by definition $\{\langle x/0 \rangle\} \times f_{g_0} \cup \{\langle x/1 \rangle\} \times f_{g_1}$.

Finally, assume that g is a $\wedge$-gate with input $g_1, \dots, g_k$. By definition, the set of clauses of F_g is the disjoint union of clauses of F_{g_i} since $\mathsf{var}(g_i)$ are pairwise disjoint and hence induce disjoint connected components in F_g. Every model of F_{g_i} is also a model of g_i by induction. Now a model of F_g is in particular a model of every clause of F_g, therefore of F_{g_i} for every i. By induction, it is thus a model of g_i for every i, and therefore of g. □

A direct corollary is the following:

Corollary 3.16. *If F syntactically entails a decision-DNNF circuit C, then $F \models C$.*

Proof. It is a direct application of Lemma 3.15 to the output of C. □

Now, it remains to show that one can efficiently decide whether a CNF formula F syntactically entails a decision-DNNF circuit C. It is not too hard to see that there is at most one correct CNF-labeling when we fix the output-label. Indeed, if we label the output of C with CNF formula F, then the rest of the CNF-labeling for the circuit can be obtained by following the definition of CNF-labeling. Now, one has to be careful to check that it indeed gives a correct labeling of the circuit. Indeed, a gate g with two outputs g_1, g_2 may be labeled differently depending on whether we define F_g from F_{g_1} or from F_{g_2}, as it happens on Fig. 3.3a. So one has to check that for every gate g and output gate g' of g, F_g is independent of the choice of g'. This can be done in polynomial time, in a top-down fashion. If this step is successful, we obtain a CNF-labeling of C with its output labeled by F. We will refer to this CNF-labeling as the *canonical syntactic CNF-labeling of C with respect to F* (or simply the canonical labeling if C and F are clear from the context). The previous discussion establishes the following:

Proposition 3.17. *Given a decision-DNNF circuit C and a CNF-formula F, we can check in polynomial time whether the canonical syntactic CNF-labeling of C with respect to F exists.*

However, this is not yet enough to check syntactic entailment as the canonical labeling may not be consistent. Checking for consistency is actually coNP-hard for the same reasons as in Proposition 3.5. To build an actual proof system for #SAT, we need to add proofs of unsatisfiability for each 0-input gate. We hence introduce the following notion of certificate. Given a propositional proof system $\mathcal{P}$, a sync-decDNNF($\mathcal{P}$) *proof* for a CNF formula F is a pair (C, ρ, k) where C is a decision-DNNF circuit, $k \in \mathbb{N}$ and ρ maps each 0-input gate to $\mathcal{P}$-proofs such that:

- The canonical CNF-labeling (F_g) of C with respect to F is well-defined.
- For every 0-input gate g of C, ρ_g is a $\mathcal{P}$-refutation of F_g.
- $C \models F$.
- C has k models over $\mathsf{var}(F)$.

Theorem 3.18. *For every propositional proof system $\mathcal{P}$ and CNF formula F, there exists a* sync-decDNNF($\mathcal{P}$) *proof (C, ρ, k) for F if and only if $\#F = k$. Moreover, given (C, ρ, k), one can check in polynomial time in $|C| + |\rho| + \|F\|$ whether (C, ρ, k) is a* sync-decDNNF($\mathcal{P}$) *proof for F.*

Proof. Let F be a CNF formula and (C, ρ, k) a sync-decDNNF($\mathcal{P}$) proof for F. Then we have $\#F = k$. Indeed, the existence of the canonical CNF-labeling and the correctness of refutations in ρ implies that the labeling is a consistent CNF-labeling of C. Hence, by Corollary 3.16, $F \models C$. Since we also have $C \models F$, C and F have the same models over $\mathsf{var}(F)$ and hence $\#F = k$.

The completeness of the system can be established as in Theorem 3.10 by considering a complete decision-tree computing F. In this case, ρ is trivial for each $g \in Z(C)$ since by definition, F_g contains an empty clause.

Checking the correctness of a certificate can be done in polynomial time using Proposition 3.17 to check for the existence of a canonical CNF-labeling, then by checking the correctness of ρ_g for every g in polynomial time (which is given by the fact that $\mathcal{P}$ is a propositional proof system). Computing the number of models of C is tractable because C is a decision-DNNF circuit. Finally, checking $C \models F$ is tractable because of Proposition 3.6. □

3.4.2 MICE proof system

The idea of using syntactic entailment between a circuit and a formula, originally defined in [CLM21] as syntactic equivalence, has independently and implicitly appeared in another line of work about #SAT solvers, namely the MICE proof system [FHR22] of Fichte, Hecherk and Roland. The connection has not been made explicit, which is the goal of this section. Each MICE proof for a CNF formula F can be used to reconstruct a decision-DNNF circuit equivalent to F, an observation that has been made by Beyersdorff, Hoffmann and Spachmann

in [BHS24]. In this section, we show that this decision-DNNF circuit is syntactically entailed by F.

We only present here the simplification of the MICE proof system known as MICE' and introduced in [BHS24]. The MICE' proof system is a rule-based proof system whose rules are presented on Fig. 3.4. MICE' proofs are based on the notion of claims. A claim is a triple (F, A, c) such that A is an assignment of variables in $\mathsf{var}(A) \subseteq \mathsf{var}(F)$. A claim is correct if $c = \#\{\tau \in 2^{\mathsf{var}(F)} \mid \tau \models F[A]\}$. A *MICE' trace* is a sequence $(I_1, \dots, I_k)$ of claims and a sequence $(\rho_1, \dots, \rho_k)$ of resolution refutations such that for every i:

- I_i is either an axiom,
- or I_i is a claim derived using rule (Join'), (Comp') or (Ext') such that the premises of each rule are claims from the set $\{I_j \mid j < i\}$,
- if I_i is derived using the (Comp') rule then ρ_i is a propositional proof of absence of models statement. Otherwise, ρ_i is empty. [1]

A *MICE' proof* of F is a MICE' trace where $I_k = (F, \emptyset, c)$. The size $s(\mathcal{I})$ of a MICE' proof $\mathcal{I}$ is defined as the number of inference steps plus the total number of clauses appearing in the resolution proofs $(\rho_1, \dots, \rho_k)$.

It has been proven in [BHS24] that MICE' is equivalent to the original MICE proof system from [FHR22], that is, every proof in MICE can be transformed into a polynomial-size proof in MICE' and vice-versa. Moreover, it is sound and complete:

Theorem 3.19 ([BHS24; FHR22])**.** *MICE' is a sound and complete proof system for #SAT.*

The inference rules of MICE' should be reminiscent of the algorithm for counting the models of a d-DNNF circuit to the reader of this manuscript, which is heavily biased towards knowledge compilation. This has been weakly formalized in [BHS24] as follows:

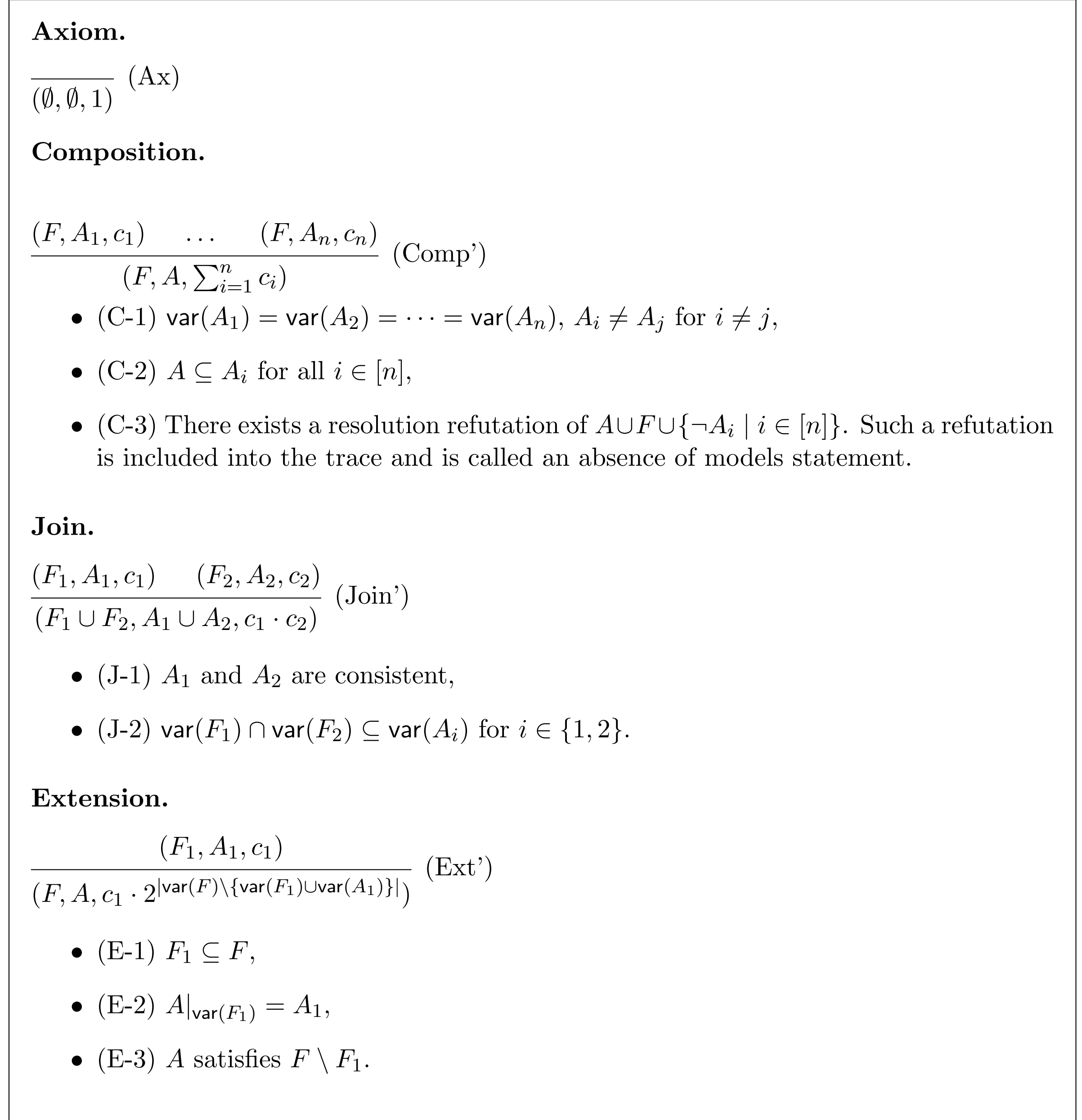
Theorem 3.20 (Theorem 6.1 in [BHS24])**.** *Let F be a CNF formula with a MICE' proof $\mathcal{I}$ using k inference rules. There exists a decision-DNNF circuit computing F of size $k \cdot (|\mathsf{var}(F)| + 1) + 1$.*

We strengthen this result by showing that we can actually extract a decision-DNNF circuit computing F that is syntactically entailed by F from a MICE' proof. To make the translation more straightforward, we start by normalizing MICE' proofs so that they only use the admissible (UComp) and (UP) rules defined on Fig. 3.5.

We start by proving that we can use the rules (UComp) and (UP) in a MICE' proof:

Proposition 3.21. *The rules (UComp) and (UP) from Fig. 3.5 are admissible in the MICE' proof system.*

[1] The original definition from [BHS24] considers only one sequence of either claims or pairs of claims and resolution proof depending on the rule applied; we prefer this presentation when we do not have to distinguish between the rules.



Axiom.

$$\frac{}{(\emptyset, \emptyset, 1)} \text{ (Ax)}$$

Composition.

$$\frac{(F, A_1, c_1) \quad \dots \quad (F, A_n, c_n)}{(F, A, \sum_{i=1}^{n} c_i)} \text{ (Comp')}$$

- (C-1) $\mathsf{var}(A_1) = \mathsf{var}(A_2) = \cdots = \mathsf{var}(A_n)$, $A_i \neq A_j$ for $i \neq j$,
- (C-2) $A \subseteq A_i$ for all $i \in [n]$,
- (C-3) There exists a resolution refutation of $A \cup F \cup \{\neg A_i \mid i \in [n]\}$. Such a refutation is included into the trace and is called an absence of models statement.

Join.

$$\frac{(F_1, A_1, c_1) \quad (F_2, A_2, c_2)}{(F_1 \cup F_2, A_1 \cup A_2, c_1 \cdot c_2)} \text{ (Join')}$$

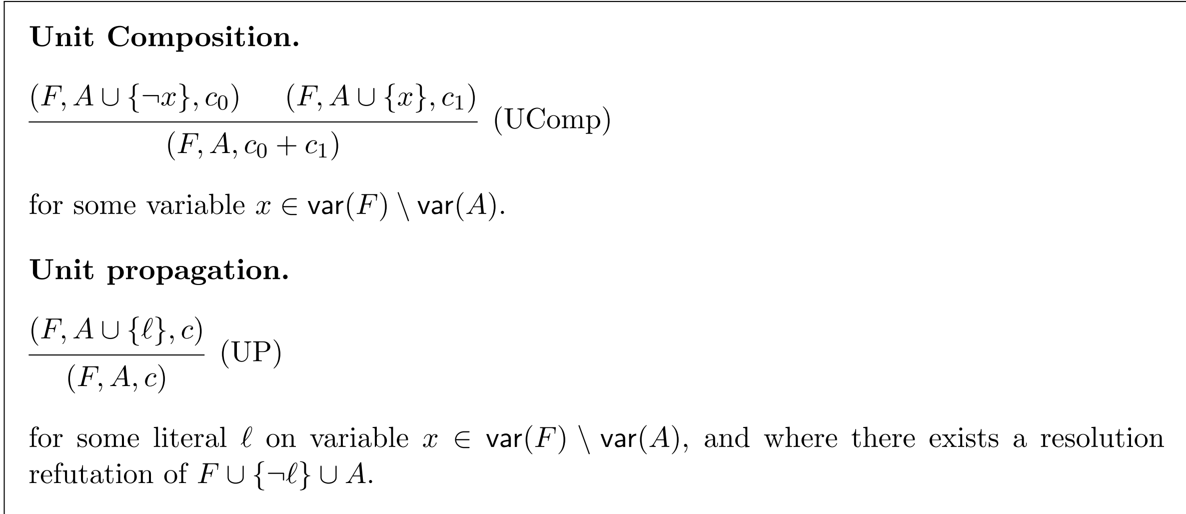
- (J-1) A_1 and A_2 are consistent,
- (J-2) $\mathsf{var}(F_1) \cap \mathsf{var}(F_2) \subseteq \mathsf{var}(A_i)$ for $i \in \{1, 2\}$.

Extension.

$$\frac{(F_1, A_1, c_1)}{(F, A, c_1 \cdot 2^{|\mathsf{var}(F) \setminus \{\mathsf{var}(F_1) \cup \mathsf{var}(A_1)\}|})} \text{ (Ext')}$$

- (E-1) $F_1 \subseteq F$,
- (E-2) $A|_{\mathsf{var}(F_1)} = A_1$,
- (E-3) A satisfies $F \setminus F_1$.

Figure 3.4: Inference rules for MICE'. Reproduced from [BHS24].



Unit Composition.

$$\frac{(F, A \cup \{\neg x\}, c_0) \qquad (F, A \cup \{x\}, c_1)}{(F, A, c_0 + c_1)} \text{ (UComp)}$$

for some variable $x \in \mathsf{var}(F) \setminus \mathsf{var}(A)$.

Unit propagation.

$$\frac{(F, A \cup \{\ell\}, c)}{(F, A, c)} \text{ (UP)}$$

for some literal ℓ on variable $x \in \mathsf{var}(F) \setminus \mathsf{var}(A)$, and where there exists a resolution refutation of $F \cup \{\neg \ell\} \cup A$.

Figure 3.5: Simplifying (Comp').

Proof. (UComp) and (UP) correspond to the special cases of the (Comp') rule when $\mathsf{var}(A_i) = \mathsf{var}(A) \cup \{x\}$ for some x. Concerning (UComp), there is no need for absence of model statement since we can derive x from A and $\neg A \cup \{\neg x\}$, and we can derive $\neg x$ similarly. Hence, we can derive the empty clause by resolving x and $\neg x$. □

We now show that we can replace one application of rule (Comp') by applications of only (UComp) or (UP). We call a MICE' proof that uses (UComp) and (UP) and does not use (Comp') a *unit MICE' proof.*

Proposition 3.22. *Given a MICE' proof $\mathcal{I}$ with p derivation steps, we can construct a unit MICE' proof with at most $(2p-1)p$ derivation steps, that does not use the (Comp') rule and has size $O(p^2 s(\mathcal{I}))$.*

Proof. We show that given a valid application of the (Comp') rule with ρ as absence of model statement:

$$\frac{(F, A_1, c_1) \quad \dots \quad (F, A_n, c_n)}{(F, A, \sum_{i=1}^{n} c_i)} \text{ (Comp')}$$

We can build a derivation with the same hypothesis and conclusion that consists of at most $2n-1$ applications of (UComp) and (UP). We do this by induction on $|\mathsf{var}(A_i) \setminus \mathsf{var}(A)|$. The case $|\mathsf{var}(A_i) \setminus \mathsf{var}(A)| = 1$ is trivial because in this case, (Comp') is either a direct application of (UComp) or of (UP). Since $n \geq 1$, we have $2n - 1 \geq 1$ and we have the right number of applications of (UComp) or (UP).

Otherwise, we first assume $n > 1$ and pick any variable $x \in \mathsf{var}(A_i) \setminus \mathsf{var}(A)$ such that there exist A_i and A_j with $A_i(x) \neq A_j(x)$ (this always exists because the A_i are distinct). We

reorder $A_1, \dots, A_n$ into $A_1, \dots, A_k$ and $A_{k+1}, \dots, A_n$ so that $A_i(x) = 0$ for every $i \leq k$ and $A_i(x) = 1$ for every $i > k$.

Now, consider the following derivation:

$$\frac{(F, A_1, c_1) \quad \dots \quad (F, A_k, c_k)}{(F, A \cup \{\neg x\}, \sum_{i=1}^{k} c_i)} \text{(Comp')}$$

It is a valid application of (Comp') since $A_i \subseteq A \cup \{\neg x\}$ for $i \leq k$. Moreover, we know by absence of model statement that we have a resolution refutation ρ of $F \cup A \cup \{\neg A_i \mid i \in [n]\}$. In particular, by Theorem 3.3, we can construct a refutation of $F \cup A \cup \{\neg A_i \mid i \in [n]\} \cup \{\neg x\}$ of size at most that of ρ from ρ. Every clause $\neg A_i$ for $i > k$ is made trivial by adding $\neg x$ in them, so they are useless for the derivation of $F \cup A \cup \{\neg A_i \mid i \in [n]\} \cup \{\neg x\}$. Hence, we have a resolution refutation ρ_0 of $F \cup A \cup \{\neg x\} \cup \{\neg A_i \mid i \in [k]\}$.

Hence, we can apply the induction hypothesis on this application of (Comp') and get a derivation T_0 of claim $(F, A \cup \{\neg x\}, \sum_{i=1}^{k} c_i)$ from the hypotheses $(F, A_1, c_1), \dots, (F, A_k, c_k)$ which does not use rule (Comp'). We get a similar derivation T_1 of $(F, A \cup \{x\}, \sum_{i=k+1}^{n} c_i)$ from hypotheses $(F, A_{k+1}, c_{k+1}), \dots, (F, A_n, c_n)$. We can hence remove the use of (Comp') as follows:

$$\frac{\dfrac{\dfrac{(F, A_1, c_1) \quad \dots \quad (F, A_k, c_k)}{\vdots\, T_0}}{(F, A \cup \{\neg x\}, \sum_{i=1}^{k} c_i)} \qquad \dfrac{\dfrac{(F, A_{k+1}, c_{k+1}) \quad \dots \quad (F, A_n, c_n)}{\vdots\, T_1}}{(F, A \cup \{x\}, \sum_{i=k+1}^{n} c_i)}}{(F, A, \sum_{i=1}^{n} c_i)} \text{(UComp)}$$

The number of applications of the (UComp) and (UP) rules in T_0 is at most $2k - 1$ by induction. In T_1, it is bounded by $2(n - k) - 1$ by induction again. Hence, the total number of application is bounded by $2k - 1 + 2(n - k) - 1 + 1 = 2n - 1$ which concludes this part of the induction.

If $n = 1$, we pick any literal $\ell \in A$ and replace the (Comp') rule by:

$$\cfrac{\cfrac{(F, A_1, c_1)}{(F, A \cup \{\ell\}, c_1)} \text{(Comp')}}{(F, A, c_1)} \text{(UP)}$$

Observe that the refutation ρ for $F \cup A \cup \{\neg A_1\}$ is also a refutation of $F \cup A \cup \{\ell\} \cup \{\neg A_1\}$ so, by induction, we can transform the application of (Comp') by a derivation tree T with hypothesis (F, A_1, c_1) and conclusion $(F, A \cup \{\ell\}, c_1)$. Now it remains to give a refutation of $F \cup A \cup \{\neg \ell\}$ to properly apply the (UP) rule. This can be directly obtained from ρ. Indeed, since $\neg \ell$ is a literal of $\neg A_1$, we can use subsumption to go from $\neg \ell$ to $\neg A_1$ and just use ρ directly. By Theorem 3.2, we can then construct a proof of the same size and without the subsumption.

Observe that after applying this transformation for every application of the (Comp') rule in $\mathcal{I}$, we have a proof without (Comp') rule and using (UComp) and (UP) instead. Each removal of (Comp') adds at most $2k-1$ new steps where k is the number of premises of the (Comp') rule.

Hence, $k \leq p$, where p is the number of steps in $\mathcal{I}$. This procedure is applied at most p times, meaning that we have a proof with at most $(2p-1)p$ steps. Moreover, each resolution refutation we use is obtained from a resolution refutation of $\mathcal{I}$ by applying either the subsumption rule or a restriction. Hence, each refutation has a size no larger than that of a refutation in the original $\mathcal{I}$. The total size of the new proof is at most $2(p-1)p \times s(\mathcal{I}) = O(p^2 s(\mathcal{I}))$. □

We are now ready to build a decision-DNNF circuit from any MICE' proof. The idea is to simply transform it into a unit MICE' proof and then turn every (Ax) rule into a 1-input gate, every (Join) rule into a decomposable $\wedge$-node, every application of (UComp) into a decision-node and every application of the (UP) rule into a decision-node with one edge pointing to a 0-input gate. Gates are linked to the premises of the corresponding rules by ignoring (Ext') rules. We show that the resulting decision-DNNF circuit C computes F and that we can show that F syntactically entails C.

Proposition 3.23. *Given a unit MICE' proof $\mathcal{I}$ for F with k inference rules, there exists a decision-DNNF circuit C of size at most $2k$ computing F with a consistent CNF-labeling (F_g) of C. Moreover, each 0-input gate v of C can be labeled with a resolution refutation ρ_v of F_v and the output of C is labeled with F.*

Proof. The MICE' proof $\mathcal{I} = (I_1, \dots, I_k)$ naturally induces a DAG: each claim I_i of $\mathcal{I}$ that is not derived as an axiom is derived using some other claim I_j as a premise. In this case we add an oriented edge from I_j to I_i.

We use this graph to build a Boolean circuit. We introduce gates $(v_1, \dots, v_k)$ as follows:

- If I_i is derived using an axiom, then v_i is a 1-input gate.
- If I_i is derived with rule (Join) with claims I_a and I_b as premises, we let v_i be a $\wedge$-gate with inputs v_a, v_b.
- If I_i is derived with rule (UComp) from claims $I_a = (F, A \cup \{x\}, c_1)$ and $I_b = (F, A \cup \{\neg x\}, c_0)$, then v_i is a decision-gate on variable x with 1-edge connected to v_a and 0-edge connected to v_b.
- If I_i is derived with rule (UP) from claim $I_a = (F, A \cup \{\ell\}, c)$, then v_i is a decision-gate on variable $\mathsf{var}(\ell)$. Moreover, if ℓ is positive, then the 1-edge of v_i is connected to v_a and the 0-edge of v_i is connected to a 0-input gate. If ℓ is negative, then the 1-edge of v_i is connected to a 0-input gate and the 0-edge of v_i is connected to v_a.
- If I_i is derived with rule (Ext') from claim I_a, then v_i and v_a are identified.

We claim that the Boolean circuit C obtained is such that each gate v_i corresponding to a claim (F', A, c) computes the same set of models as $F'[A]$. Moreover, the variables below v_i are exactly $\mathsf{var}(F'[A])$. The proof is by induction. This is obviously true for the claim derived by the axiom rule since the empty CNF formula has exactly one model and has no variables.

Now assume $I_i = (F, A, c)$ is obtained using the (Join) rule on claims $I_a = (F_1, A_1, c_1)$ and $I_b = (F_2, A_2, c_2)$. By induction, v_a computes $F_1[A_1]$ and v_b computes $F_2[A_2]$. Hence, v_i computes $F_1[A_1] \wedge F_2[A_2] = (F_1 \wedge F_2)[A_1 \cup A_2] = F[A]$. Moreover, by induction, we

know that $\mathsf{var}(v_a) = \mathsf{var}(F_1[A_1])$ and $\mathsf{var}(v_b) = \mathsf{var}(F_2[A_2])$. First observe that $\mathsf{var}(F_2[A_2]) \cap \mathsf{var}(F_1[A_1]) = \emptyset$ by (J-2) (see Fig. 3.4) hence v_i is decomposable. Also, $\mathsf{var}(v_i) = \mathsf{var}(v_a) \cup \mathsf{var}(v_b) = \mathsf{var}(F_1[A_1]) \cup \mathsf{var}(F_2[A_2]) = \mathsf{var}(F[A_1 \cup A_2]) = \mathsf{var}(F[A])$.

If $I_i = (F, A, c)$ is obtained using the (UComp) rule on claims $I_a = (F, A \cup \{x\}, c_1)$ and $I_b = (F, A \cup \{\neg x\}, c_0)$, then v_i computes by definition $(x \wedge f_a) \vee (\neg x \wedge f_b)$ where f_a and f_b are the functions computed by v_a and v_b respectively. By induction, they are $F[A \cup \{x\}]$ and $F[A \cup \{\neg x\}]$ respectively. Hence, v_i computes $(F[A \cup \{x\}] \wedge x) \vee (F[A \cup \{\neg x\}]) = F[A]$. Moreover, x is not in $\mathsf{var}(v_a)$ nor $\mathsf{var}(v_b)$ since they are equal to $\mathsf{var}(F[A \cup \{x\}])$ and $\mathsf{var}(F[A \cup \{\neg x\}])$ respectively. Hence, this is a valid decision-gate and $\mathsf{var}(v_i) = \{x\} \cup \mathsf{var}(F[A \cup \{x\}]) \cup \mathsf{var}(F[A \cup \{\neg x\}]) = \mathsf{var}(F[A])$.

The case where $I_i = (F, A, c)$ is obtained via the (UP) rule is similar since one side is not satisfiable (and we have a resolution proof of this fact).

It remains to show if $I_i = (F, A, c)$ is derived from $I_a = (F_1, A_1, c_1)$ using the extension rule, then $F[A] = F_1[A_1]$. This is clear from (E-1), (E-2) and (E-3) which concludes the induction.

We now give a consistent CNF-labeling of C. We label v_i with $F'[A']$ for every i where $I_i = (F', A', c)$. Moreover, every 0-input gate of C arises, by construction, from a (UP) derivation. Let g be a 0-input gate constructed because of a (UP) derivation with premise $(F', A' \cup \{\ell\}, c)$. We label g with $F'[A' \cup \{\neg \ell\}]$. Observe that the resolution refutation from the (UP) derivation is, by definition, a refutation of $F'[A' \cup \{\neg \ell\}]$ and hence, we have a correct refutation of the label of g.

The rules of unit MICE' allow us to show that this is a consistent CNF-labeling of C. Indeed, let v_i be a gate labeled by $F'[A']$ with input v_j labeled by $F''[A'']$. If v_i is a decision gate, then either I_i is derived using (UP) or (UComp). In any case, we have $F' = F''$ and $A'' = A' \cup \{\ell\}$ for some literal $\ell \in \{x, \neg x\}$ and the edge from $v_j \to v_i$ is labeled by $b \in \{0, 1\}$ which corresponds to the sign of literal ℓ. Hence, v_i is labeled by $F'[A']$ and v_j is labeled by $F'[A' \cup \{\ell\}] = F'[A'][x/b]$ which corresponds to the condition for syntactic labeling.

Similarly, if v_i is a $\wedge$-gate and let v_a, v_b be its inputs. By construction, $I_i = (F_1 \wedge F_2, A_1 \cup A_2, c)$ is derived from $I_a = (F_1, A_1, c_1)$ and $I_b = (F_2, A_2, c_2)$ and $\mathsf{var}(F_1) \cap \mathsf{var}(F_2) \subseteq \mathsf{var}(A_1)$ and $\mathsf{var}(F_1) \cap \mathsf{var}(F_2) \subseteq \mathsf{var}(A_2)$. Then v_i is labeled by $F[A]$ where $F = F_1 \wedge F_2$ and $A = A_1 \cup A_2$ and v_a is labeled by $F_1[A_1]$ and v_b by $F_2[A_2]$. Observe that since A_1 and A_2 are consistent and their shared variables cover $\mathsf{var}(F_1) \cap \mathsf{var}(F_2)$, we have $(F_1 \wedge F_2)[A_1 \cup A_2] = F_1[A_1] \wedge F_2[A_2]$ and this conjunction is decomposable. In particular, we have that the $\mathsf{var}(F_1)$-connected component of $F[A]$ is $F_1[A_1]$ and its $\mathsf{var}(F_2)$-connected component is $F_2[A_2]$ which concludes the fact that we have a syntactic CNF-labeling of C.

Finally, observe that the output of C is by definition labeled by F. Moreover, every 0-input gate of C us generated using the (UP) rule in the case when $F'[A \cup \{\neg \ell\}]$ is UNSAT, and we have a resolution proof of this fact. Hence, we have a consistent CNF-labeling of C whose output is labeled by F and have resolution refutations of its 0-input gate labels. □

We can now conclude regarding the link between the MICE proof system and syntactic caching based proof systems:

Corollary 3.24. *Given a MICE' proof $\mathcal{I}$ of F with p derivation steps, we can construct a* sync-decDNNF(Res) *proof (C, ρ, k) for F of size $O(p^2 s(\mathcal{I}))$.*

Proof. We start by normalizing the MICE' proof and use Proposition 3.23 to build a decision-DNNF circuit C of size $O(p^2)$ computing F. We let $k = \#C$ be the number of models of C over $\mathsf{var}(F)$ and for each 0-input gate g, we let ρ_g be the resolution refutation of Proposition 3.23. By construction, $C \models F$. Moreover, Proposition 3.23 shows that the canonical CNF-labeling of C with respect to F is well-defined (that is F syntactically entails C) and ρ_g is a correct refutation of the canonical CNF-labeling. Hence, (C, ρ, k) is a correct sync-decDNNF(Res) proof. The size of the circuit is p^2 and the total size of the resolution refutations is bounded by $O(p^2 s(\mathcal{I}))$. □

The natural follow-up question now is to ask whether given a sync-decDNNF(Res) proof for F, we can build a polynomial-size MICE proof. It is indeed not straightforward that MICE can p-simulate sync-decDNNF(Res). Indeed, syntactic caching may leverage the fact that $F_1[A_1] = F_2[A_2]$ even if F_1 and F_2 are really different. MICE has only a degenerated version of it (the extension rule) which seems too weak to encode proper syntactic caching, but it does not rule out the possibility that MICE can p-simulate sync-decDNNF(Res) entirely. While we left the question open in a draft version of this chapter, it has been solved in the meantime by Beyersdorff, Hoffmann and Kasche [BHK25], who provided an exponential separation of both systems[2]. Interestingly, they also show that MICE can be augmented by a simple rule allowing the identification of proofs for (F, A) and (F', A') as long as $F[A] = F'[A']$, which is exactly what syntactic caching does. This new proof system then gives a proof system as powerful as sync-decDNNF(Res).

3.4.3 Certified d4

In this section, we quickly discuss the differences between the syntactic entailment presented in Section 3.4 and the way it is used in [CLM21]. The starting point is the same, and we observed that the decision-DNNF circuits produced by D4 are indeed syntactically entailed by the input CNF formula. For aesthetics reasons though, we use a slightly different but equivalent definition. Given a decision-DNNF circuit C, we map each gate g to an edge $e_g = (g, g')$ of the circuit called the *canonical arc*. It allows us to define a unique labeling of the decision-DNNF circuit: the output is labeled with the input CNF formula F. Moreover, F_g is defined from $F_{g'}$ in the same way as we do for syntactic entailment: if g' is a decision-gate, then $F_g = F_{g'}[\ell]$ where ℓ is the label of the canonical arc (g, g'). If g' is a $\wedge$-gate, then F_g is the $\mathsf{var}(g)$-connected component of $F_{g'}$. This labeling always exists. If C is syntactically entailed by F, then F_g is the canonical CNF-labeling as defined in Section 3.4. Otherwise, the checking algorithm from [CLM21] failed because it was impossible to establish syntactic entailment. That is, there is a gate g with an output g'', such that the CNF formula obtained from $F_{g''}$ by following arc (g, g'') is different from F_g, the one we get by following the canonical arc. In other words, the canonical CNF-labeling does not exist. Hence canonical arcs or canonical CNF-labeling give the same notion of syntactic entailment but in one scenario we fail because the canonical CNF-labeling is ill defined, in the other we fail because the induced labeling is not correct. These are two flavors of the same definition. In practice however, the canonical

[2]As mentioned in the introduction of this chapter, this paper came to our attention late in the writing process and hence uses a different but equivalent theoretical framework than the one presented here.

arc allows us to map each gate to a canonical set of literals that basically represent the state of the solver when it reached the 0-gate. This integrates well with the second improvement that we now describe.

The work in [CLM21] is mainly practical, as we were aiming to certify a precise tool, D4. To this end, we focused on designing certificates that could be generated with the least amount of modification of D4 while still being easy to check. In practice, the most costly part in the certificate is to check the resolution refutation for each 0-input gate. In theory, when a top-down solver returns a 0-gate, it either has detected a conflict with one clause of F, or a SAT solver call has identified a conflict, or one learned clause by a previous SAT solver call has a conflict with the current assignment. In the two first cases, we can basically report either a trivial resolution proof or the one that has been generated by the SAT solver. In the third case, we however have to reconstruct a refutation from the learned clause. It would incur a high cost in practice: first, it means that we need to store, during the execution of D4, more information so that we keep track of how the SAT solver has learned the clause to be able to reconstruct the refutation. Second, we may duplicate several times the same refutation or similar refutation in the circuit, which may be costly in practice. Last, to check the correctness of a certificate, one would need to check each refutation independently, which can be costly. To circumvent this difficulty, we just return the set of clauses R that have been learned through SAT solver oracle calls and append them in the certificate. The nature of clauses learned by SAT solver allows us to efficiently check that $F \models R$, that is, every clause in R is entailed by F. This can be done using dedicated efficient tools. Now, interestingly, each time D4 returns a 0-input gate, it is because one clause in $F \cup R$ is not compatible with the current assignment of the solver. Using the canonical arcs from the previous paragraph, we can recover this canonical assignment and plug it into $F \cup R$ and check whether it contains the empty clause. This is enough to guarantee that the 0-gate is legitimate and it allows us to factorize parts of resolution derivation where the same clause may be used in several places without being repeated.

There is however a minor technical twist when doing this. In [CLM21], we have to force each internal gate of the circuit to compute a non-zero Boolean function. This is not a problem since D4 will never explore an unsatisfiable branch because it calls a SAT solver at each recursive call. However, without this assumption, we would not have a correct certificate. Indeed, learned clauses may interact badly with component decomposition. Indeed, assume that $F = F_1 \wedge F_2$ with $\mathsf{var}(F_1) \cap \mathsf{var}(F_2) = \emptyset$. Assume that F_2 is unsatisfiable but not F_1. Hence we could prove that $F \models c_1$ for any clause c_1 over $\mathsf{var}(F_1)$, for example $c_1 = x$ is a unit clause. However, it may be the case that $F_1 \not\models x$, that is, F_1 has at least one model where $x = 0$. Hence, if we have a recursive call on F_1, we could wrongly use c_1 to perform a unit propagation on x. It does not happen, however, when F is satisfiable, since if $F \models c_1$ and c_1 is over $\mathsf{var}(F_1)$, then we necessarily have $F_1 \models c_1$. Hence, in [CLM21], we have an extra condition that each internal gate of the decision-DNNF circuit is satisfiable.

It is not hard to see that this does not make the proof system more powerful than $\mathsf{sync\text{-}decDNNF(Res)}$ in theory. It just makes the certificate possibly a bit less verbose but the gain will not be polynomial. We can still reproduce full refutations from R in each 0-input gate and have a $\mathsf{sync\text{-}decDNNF(Res)}$. Conversely, given a $\mathsf{sync\text{-}decDNNF(Res)}$, we can show

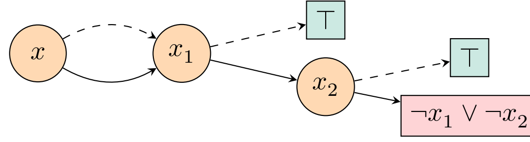


Figure 3.6: A certified decision-DNNF circuit for $(x \vee \neg x_1 \vee \neg x_2) \wedge (\neg x \vee \neg x_1 \vee \neg x_2)$ using the clause $\neg x_1 \vee \neg x_2$ derived by resolution.

that we can first simplify every gate without models into a 0-input. Then we can pick an arbitrary canonical arc. For each 0-gate, we can use the refutation of the canonical labeling to construct a clause that evaluates to $\emptyset$ by following the canonical arc. We do not give full details, but we note that:

Theorem 3.25. *The proof systems from [CLM21] and* sync-decDNNF(Res) *are p-equivalent.*

While the general approach for certifying D4 sketched here and presented in [CLM21] is sound, our implementation has a bug which was uncovered by Bryant, Nawrocki, Avigad and Heule [Bry+25]. To explain 0-input gates, we were relying on the clauses learned with oracle SAT solver calls but were checking them using the DRAT-TRIM tool [WHH14] which uses the DRAT proof system. The problem is that the correctness of our proof system relies on the fact that we only use clauses entailed by the formula while the DRAT proof system may introduce new clauses that are not entailed. The key property of the DRAT proof system is that if c is a clause derived from F with DRAT, $F \wedge c$ is satisfiable if and only if F is satisfiable. However, c may not be entailed by F which can be maliciously used to make CD4 accept an incorrect proof. In practice, this bug can be circumvented by using RUP certificates that have the desired property.

3.5 Certified decision-DNNF circuits with learned clauses

We now turn our attention to an extension of the kcps proof system presented in Section 3.3. Recall that one limitation of kcps, which we depicted on Fig. 3.2, is that labeling 0-input gates with conflicting clauses may prevent some natural caching in the circuit. In Fig. 3.2a, there is no way to label the unique 0-input gate with a clause of F that is conflicting with every path from the output to this gate, as required by any kcps-certificate. However, we could label this gate with the clause $\neg x_1 \vee \neg x_2$ as depicted on Fig. 3.6. This gives a correct certified decision-DNNF circuit, but it is not a kcps-certificate because $\neg x_1 \vee \neg x_2$ is not a clause of F. That said, $\neg x_1 \vee \neg x_2$ is entailed by F because this clause could be obtained as the resolvent of $x \vee \neg x_1 \vee \neg x_2$ and $\neg x \vee \neg x_1 \vee \neg x_2$. Hence, if C denotes the certified decision-DNNF circuit from Fig. 3.6 and $F = (x \vee \neg x_1 \vee \neg x_2) \wedge (\neg x \vee \neg x_1 \vee \neg x_2)$, we know by Proposition 3.7 that $(\neg x_1 \vee \neg x_2) \models C$ and since $F \models (\neg x_1 \vee \neg x_2)$ as shown above, we have $F \models C$. More generally, we can prove this generalization of Corollary 3.8:

Corollary 3.26. *Let F be a CNF formula and C be a correct certified decision-DNNF circuit such that for every $g \in Z(C)$, c_g is a clause such that $F \models c_g$. Then $F \models C$.*

Proof. It directly follows from the fact that $F \models \bigwedge_{g \in Z(C)} c_g$ since c_g is a clause entailed by F for every $g \in Z(C)$. $\square$

We can then augment kcps-certificates with any propositional proof system $\mathcal{P}$ so that we allow labeling 0-input gates of the certified decision-DNNF circuit with clauses entailed by F, as long as we are given a proof in $\mathcal{P}$ that F entails them. More precisely, given a propositional proof system $\mathcal{P}$, a $\mathsf{kcps}(\mathcal{P})$-certificate is given as (C, ρ, k) such that:

- C is a correct certified decision-DNNF circuit,
- For every $g \in Z(C)$, ρ_g is a $\mathcal{P}$-refutation of $F \wedge \neg c_g$.
- $C \models F$
- C has k models over $\mathsf{var}(F)$.

From what precedes, we immediately have:

Theorem 3.27. *For every propositional proof system $\mathcal{P}$ and CNF formula F, there exists a $\mathsf{kcps}(\mathcal{P})$-certificate (C, ρ, k) for F if and only if $\#F = k$. Moreover, given (C, ρ, k), one can check in polynomial time in $|C| + \|F\| + |\rho|$ whether (C, ρ, k) is a $\mathsf{kcps}(\mathcal{P})$-certificate for F.*

The idea of $\mathsf{kcps}(\mathcal{P})$ was first sketched in [Cap19], but was not investigated. In [Bey+24], a thorough analysis of kcps(Res) (denoted by kcps^+ in the paper) is performed. Surprisingly, kcps^+ is more general than MICE. In other words, adding the power of resolution to certified decision-DNNF circuits is enough to handle syntactic caching. This is exemplified by the example from Fig. 3.6. In Fig. 3.2a, it was clear that the decision-DNNF circuit could not be used as a kcps-certificate, but Fig. 3.6 shows that using resolution is enough to circumvent this limitation. In other words:

Theorem 3.28 ([Bey+24, Theorem 4.1])**.** *Given a MICE-proof $\mathcal{I}$ for F, there exists a* kcps(Res)*-certificate of size* $\mathrm{poly}(s(\mathcal{I}))$ *for F.*

We do not reproduce a proof of the result here but only explain intuitively why it works. We explain how we can tag the decision-DNNF circuit from Corollary 3.24 to get a kcps(Res)-certificate. The key observation is that a proof in MICE can be reused only if it has the same CNF F and the same partial assignment A. Hence, assume that two different paths in the circuit lead to the same 0-gate g in the decision-DNNF circuit built in Proposition 3.23. Let A_1 and A_2 be the corresponding assignments. Now, by construction, the 0-gate can only be created because of a unit propagation in the circuit. The decision-gate g' leading to it must correspond to a (UP) rule in the MICE-proof labeled by some $F' \subseteq F$ and A' such that $A' \cup \{\neg\ell\} \subseteq A_1$, $A' \cup \{\neg\ell\} \subseteq A_2$ where ℓ is the literal labeling the edge from g' to g. Moreover, the (UP) rule is given with a refutation of $F' \wedge A' \wedge \{\neg\ell\}$ which can be turned into a resolution proof that $F' \models c_g$ where $c_g = \neg A' \vee \ell$. Every path to g must be consistent with A', hence

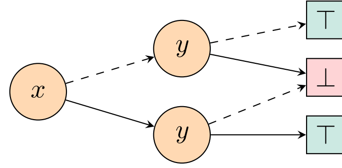


Figure 3.7: A decision-DNNF circuit syntactically equivalent to $(x \vee \neg y) \wedge (\neg x \vee y)$ illustrating why sync-decDNNF(Res) may be hard to capture with a kcps(Res)-certificate. Credits to Olaf Beyersdorff, Kasper Kasche, Tim Hoffmann and Luc Spachmann (personal communication).

tagging g with c_g and proceeding in the same way for every other 0-gates in the circuit yields a kcps(Res)-certificate for F.

The previous simulation works because the construction from Proposition 3.23 constructs a decision-DNNF circuit with a very weak caching rule. It is not clear, however, whether it can be extended to show that a polynomial size kcps(Res)-certificate can be obtained from a sync-decDNNF(Res) proof. To illustrate why this is not straightforward, consider the example from Fig. 3.7. This is syntactically equivalent to $F = (x \vee \neg y) \wedge (\neg x \vee y)$ but the two paths leading to the 0-gate refutes distinct clauses. Moreover, we can only derive the tautological clause from F using resolution. Hence, to build a kcps(Res)-certificate for F, one needs to use a different underlying decision-DNNF circuit. It is not clear whether the transformation can be done in polynomial time (or at least space). We leave this question open:

Open question 8. *Can* sync-decDNNF(Res) *be p-simulated by* kcps(Res)*?*

On the other hand, syntactic caching may sometimes be too weak to encode kcps-certificate since decision-DNNF circuits may cache some values for semantic reasons rather than for syntactic ones. To get a feeling of why, consider the example given on Fig. 3.8 representing a correct certified decision-DNNF circuit C for $F' = (x \vee y_1) \wedge (\neg x \vee y_2) \wedge (y_1 \vee \neg y_2) \wedge (\neg y_1 \vee y_2) \wedge y_1$. F does not syntactically entail C. Indeed, $F'[x] = y_2 \wedge (y_1 \vee \neg y_2) \wedge (\neg y_1 \vee y_2) \wedge y_1$ while $F'[\neg x] = (y_1 \vee \neg y_2) \wedge (\neg y_1 \vee y_2) \wedge y_1$. Now, clearly, $F'[x]$ and $F'[\neg x]$ define the same Boolean function, but this is not syntactically straightforward. This example is obviously simplified and not realistic since any reasonable top-down solver would branch on y_1 first. However, it serves as an example to show how syntactic entailment and clause certificates are two orthogonal mechanisms allowing to prove that a CNF formula entails a decision-DNNF circuit. By Theorem 3.28, both mechanisms are encompassed by kcps(Res).

While the previous example is not enough to establish an exponential separation between MICE and kcps(Res), [Bey+24] actually establishes that:

- MICE does not p-simulate kcps, that is, there is a family (F_n) of CNF formulas such that any MICE proof for F_n has size exponential in n but F_n has a polynomial size kcps-certificate.
- kcps does not p-simulate MICE.
- kcps(Res) p-simulates kcps and MICE.

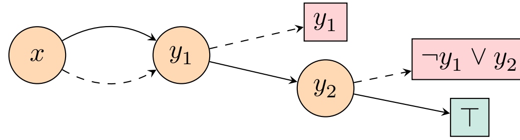


Figure 3.8: A correct certified decision-DNNF circuit for $F = (x \vee y_1) \wedge (\neg x \vee y_2) \wedge (y_1 \vee \neg y_2) \wedge (\neg y_1 \vee y_2) \wedge y_1$ that is not syntactically entailed by F.

We do not reproduce the full proofs of these results here and refer the reader to [Bey+24] for details. That said, we give a few hints on how the separations are established. Showing that kcps does not p-simulate MICE is actually not too hard. For unsatisfiable CNF formulas, it is clear that MICE can use the full power of resolution, by using (Comp') on an empty assignment and a resolution refutation of F. Now, any kcps-certificate for an unsatisfiable formula is a decision-DNNF circuit where each $\wedge$-gate must have one unsatisfiable side. By keeping only this side for each $\wedge$-gate and removing unreachable gate, we end up with a decision-DNNF circuit without $\wedge$-gates and only 0-input gates, each labeled by a clause of F, refuted by any path leading to it. This is a well-known characterization of regular resolution, a proof system weaker than resolution where each variable can be resolved at most once on any path, see for example [Juk12, Theorem 18.1]. Regular resolution is known to be exponentially weaker than resolution, hence any family of unsatisfiable CNF with no small regular-resolution but small resolution refutations will exponentially separate kcps from MICE.

To prove a lower bound on MICE with small kcps-certificate, [Bey+24] construct a formula F in a way that prevents syntactic caching from happening, leading to an exponential lower bound on the size of any decision-DNNF circuit that is syntactically entailed by F. However, this formula has a small certified decision-DNNF circuit that uses non-syntactic caching.

3.6 Propositional based certification

Before concluding this chapter and for completeness, we quickly present one last possible way of overcoming the coNP-completeness from Proposition 3.5 that has been proposed by Bryant, Nawrocki, Avigad and Heule [Bry+25]. In this work, the authors simply encode the fact that $\neg(F \Leftrightarrow C)$ for a tractable circuit C into a propositional formula, by using a Tseitin encoding. In this case, a certificate is a refutation of $\neg(F \Leftrightarrow C)$ using a propositional proof system.

This idea is interesting in that it allows to use circuit classes beyond decision-DNNF circuits. Indeed, the previously presented proof systems heavily depended on the decision-DNNF circuit structure. In the propositional encoding approach, one can use any class of circuits as long as model counting is tractable. However, sometimes, one would need an extra proof that the given circuit is in the expected class. Indeed, suppose that the user is given a d-DNNF circuit C and a proof that $F \Leftrightarrow C$. There is no way for the user to check that C is indeed deterministic, since determinisism checking is a coNP-hard (see Corollary 1.5). The most general class of circuit considered in [Bry+25] is the class of Partitioned-Operation Graphs (POG), which can be seen defined as d-DNNF circuits with negation gates, a class of circuits which

support efficient model counting.

Another strong point of the approach is that proofs can be generated from the circuit alone. Either by calling a SAT solver to prove that $\neg(F \Leftrightarrow C)$ is unsatisfiable and getting a DRAT proof (the *monolithic approach* [Bry+25]) or by some ad-hoc algorithm exploiting the underlying structure of the circuit (the *structural approach*). Both ideas were presented and successfully exploited in [Bry+25] to assess the correctness of D4 output on a given benchmark.

POG-based proof systems can hence be parametrized by both the underlying propositional proof system $\mathcal{P}$ it uses and by the class of circuits that is considered for representing the CNF formula. For a given propositional proof system $\mathcal{P}$ and a class of Boolean circuits $\mathcal{C}$, we denote by $\mathsf{CPOG}^{\mathcal{C}}_{\mathcal{P}}$ (for Certified POG) the proof system defined as follows: a $\mathsf{CPOG}^{\mathcal{C}}_{\mathcal{P}}$-certificate for F is a circuit $C \in \mathcal{C}$, and $\mathcal{P}$-proofs ρ_1, ρ_2, ρ_3 such that:

- ρ_1 is a $\mathcal{P}$-refutation of a Tseitin encoding of $\neg(F \Rightarrow C)$,
- ρ_2 is a $\mathcal{P}$-refutation of a Tseitin encoding of $\neg(C \Rightarrow F)$,
- ρ_3 is a proof that $C \in \mathcal{C}$.

From what precedes, it may be surprising that ρ_2 and ρ_3 are needed in a $\mathsf{CPOG}^{\mathcal{C}}_{\mathcal{P}}$. This is because for decision-DNNF circuits, they are not necessary since both problems can be solved in polynomial time. However, for larger circuit classes, this may not be true. For d-DNNF circuit, Proposition 3.6 ensures that ρ_2 is not needed as the problem can be solved in polynomial time. However, checking that a given DNNF circuit is a deterministic is coNP-hard (see Lemma 1.4). Hence, in this case, ρ_3 is needed to allow one to check the certificate in polynomial time. Observe that even in the case of general POG, that is, d-DNNF circuits with negation gates, ρ_2 is not really necessary as checking $C \models F$ when C is a POG can be done in polynomial time, with the same algorithm as the one proposed in Proposition 3.6. In [Bry+25], ρ_2 is however introduced for symmetry purposes. Indeed, the CPOG mechanism from the paper has been formally verified in Lean, hence, having a unique framework to express equivalence is more practical than having to prove an ad hoc algorithm. That said, in practice, a polynomial size refutation ρ_2 is extracted from the circuit using Proposition 3.6.

Interestingly, when $\mathcal{C}$ is the class of decision-DNNF circuits, $\mathsf{CPOG}^{\mathcal{C}}_{\mathsf{Res}}$ offers a proof system that is more powerful than $\mathsf{kcps}(\mathsf{Res})$ and $\mathsf{sync\text{-}decDNNF}(\mathsf{Res})$ because ρ_1 allows to express both syntactic caching and (proven) semantic caching. That said, as exemplified with CD4, $\mathsf{sync\text{-}decDNNF}(\mathsf{Res})$ can be extracted directly during the compilation process while POG-based certificates have to be constructed after compilation, leading to different advantages.

3.7 Conclusion

In this chapter, we tried to give an overview of the current state of research concerning #SAT proof systems, certainly biased toward the knowledge compilation point of view, and tried to explain how our contributions have helped create this line of research. This connection between knowledge compilation and proof complexity runs, in our opinion, deeper than the simple connections we have drafted here. Indeed, by transforming a CNF formula into a

certified circuit, not only can we verify that the number of models is correct, but we can also leverage it for any other kind of tractable operation on the circuit. This encompasses weighted model counting, maximal models, etc. In [Cap19], we proposed proof systems for MaxSAT using knowledge compilation. The idea is to twist the input CNF formula by adding selector variables on each of its clauses so that the maximum number of clauses simultaneously satisfiable can be found in the maximal satisfying assignment of the new CNF formula. By compiling it into a certified circuit where such maximal satisfying assignments are easy to find, we can construct proof systems for MaxSAT. The relation between these new proof systems and other proof systems for MaxSAT such as MaxSAT resolution [BLM07] is not yet well understood and is left as future work.

We would like to finish this chapter on the following observation: it may seem from this chapter that proof systems for #SAT are only a minor change of perspective regarding the connection between #SAT and knowledge compilation. We would like to argue that this new framework actually enables a more precise understanding of this connection. The connection between knowledge compilation and #SAT was used in particular to prove lower bounds on the runtime of solvers. By proving a lower bound on the size of decision-DNNF circuits representing a given CNF formula F, using for example the framework from [Bov+16], we also prove a lower bound on the runtime of any top-down #SAT solver on the input. However, this approach is still a crude approximation of the internals of a top-down #SAT solver. Studying knowledge compilation with this proof complexity perspective gives a finer understanding of the internals of the solver: for example, syntactic caching is a limitation of these solvers for constructing some decision-DNNF circuit but proving lower bounds for it has been enabled by the formalization we propose in this chapter and which has been independently proposed in [BHK25] while this chapter was written.

Chapter 4

Applications to Databases

We now leave the realm of propositional logic to deal with another kind of logical framework: databases. Of course, research on databases is a huge field and we will focus here on a tiny part of the existing literature. We will be interested in the following question and its applications: how can we represent the answer set of a database query? This is a fundamental question, akin to the one raised in Chapter 1 for Boolean functions. Depending on the representation, previously seemingly intractable tasks now become feasible. It turns out that a large part of the database theory literature on aggregation problems has focused on finding queries where the tasks of enumerating or aggregating their answers are tractable. One of the earliest and most influential work in this branch is the seminal paper by Yannakakis [Yan81] where he shows that model checking, that is, checking whether the query has a model, can be decided in linear time (in the size of the database) when the query is *an α-acyclic conjunctive query*. This work has been the initial point of several fruitful research directions. One of them has been to understand which tasks can be solve in linear time in the size of the database on α-acyclic conjunctive queries while proving, under reasonable complexity assumptions, that α-acyclic conjunctive queries are the only conjunctive queries enjoying this property. In [BDG07], Bagan, Durand and Grandjean made an important step toward this goal: they proved that the Yannakakis algorithm can be extended to enumerate the answers of acyclic conjunctive queries[1]. The first phase of their algorithm is a preprocessing step that builds a data structure in time linear in the size of the database. The second phase uses the data structure to produce the answer of the query with a small delay, depending only on the size of the query, between two distinct outputs. Their algorithm is often described as a linear preprocessing and constant delay enumeration algorithm, in the sense that, if we consider the query to be a constant, then the first answer is output after a time linear in the input size and the delay between two outputs is constant. They show that, under reasonable assumptions, only acyclic queries can be enumerated with such complexity guarantees.

Several follow-up work continued to establish new tractability results for acyclic conjunctive queries. Pichler and Skritek showed that one can compute the number of answers of a

[1] Technically, to account for projections, one has to consider so-called free-connex acyclic conjunctive queries, but we leave this detail out for now.

(free-connex) acyclic conjunctive query in linear time [PS13][2], which has later been generalized to aggregation over arbitrary semirings by Khamis, Ngo, Hung and Rudra [KNR16] and Joglekar, Puttagunta, and Ré [JPR16]. Carmeli, Tziavelis, Gatterbauer, Kimelfeld and Riedewald [Car+23] have shown that the enumeration algorithm can be transformed into a *direct access* algorithm for a specific lexicographical order: after a linear preprocessing, one can find, for any i, the ith answer of the query in polylogarithmic time. All these results can actually be generalized to arbitrary query if one is ready to spend more than linear time in the preprocessing phase. In this case, the preprocessing complexity is measured by analyzing the structure of the query, using parameters such as hypertree width or fractional hypertree width [GLS99; GM14].

All these tractability results share a very similar preprocessing phase: a tree-like data structure representing is built from which we can extract the answers of the query efficiently. In a way, the data structure can be seen as a factorized representation of the answer set. This has been made formal by Olteanu and Závodný [OZ15] who coined the term "factorized database", describing a general data structure which can represent relations in a factorized way while still allowing insight into the represented relations, such as computing its size or enumerating its tuples. Olteanu and Závodný show that linear-size data structure can be computed for acyclic conjunctive queries, allowing them to recover existing results in a more modular way.

Their representation is based on two atomic relations: Cartesian products and disjoint unions. It is not hard to see that, if the domain is $\{0, 1\}$, the former correspond to decomposable $\wedge$-gates and the latter to (smooth) deterministic $\vee$-gates. The main difference here is that we can use a larger domain than $\{0, 1\}$. In this chapter, we hence revisit these tractability results through the prism of knowledge compilation. We introduce the notion of relational circuits, a generalization of DNNF circuits to larger domains and show how one can reinterpret Yannakakis algorithm as a compilation algorithm from acyclic conjunctive queries to relational circuits. We then show how a classical compilation algorithm for CNF formulas, namely exhaustive DPLL [San+04], can be straightforwardly adapted to compile conjunctive queries. We show that it also runs in linear time on acyclic conjunctive queries and that it generalizes to conjunctive queries where atoms can be negated, a setting where Yannakakis does not work.

Organization of this chapter. Section 4.1 introduces the necessary preliminaries and Section 4.2 contains a generalization of NNF circuits to non-binary domains. Section 4.3 shows how to adapt Yannakakis algorithm into a compilation algorithm and Section 4.4 shows that exhaustive DPLL can also be adapted into a database setting. Finally, Section 4.7 shows that we can recover many tractability results with this approach.

Personal contributions covered in this chapter. Our main contribution to this chapter can be found in [CI24] where we adapt exhaustive DPLL to conjunctive queries and conjunctive queries with negation. In this paper, we provide a full analysis of the complexity of exhaustive

[2] The upper bound given in the paper on the runtime of their algorithm is quadratic but a more careful analysis shows that we can do it in linear time.

DPLL depending on a static order on the variables, which allows us to generalize direct access algorithms from [Car+23; BCM22] to conjunctive queries with negations. This result can be seen as a generalization of earlier work on the specific subcase of β-acyclic queries [Cap17; BCM15]. Compared to [CI24], the focus of this chapter is more on the compilation part and less on the direct access problem. To this end, we focus on explaining how one can see Yannakakis algorithm as a compilation algorithm, which we think has never been really made explicit in the literature (even if [OZ15] does similar things, their compilation algorithm is still slightly different from Yannakakis itself). The complexity analysis of the DPLL algorithm being tedious, we do not cover all details in this chapter and refer to [CI24] for details.

4.1 Preliminaries

In this section, we give the main definitions needed to understand this chapter. We do not aim for exhaustivity regarding databases and will present most notions in a biased way, to ease comparison with knowledge compilation. One notorious example is that we only use the named tuple convention, that is, tuples are seen as mapping from variables (names) to values, while databases sometimes prefer positional convention, where tuples are k-uplets of values.

Tuples and relations. We extend the notion of assignment from the Boolean point of view to the notion of tuples. A *tuple on variables X and domain D* is an element of D^X, that is, a mapping from X to D. We denote by $\mathsf{var}(\tau) = X$ the variables assigned by τ. Assignment, as defined in Chapter 1, can be seen as tuples over a binary domain. We extend every notation from assignment to tuples: for example, we denote by $\tau|_Y$ the tuple over $Z := \mathsf{var}(\tau) \cap Y$, such that for every $z \in Z$, $\tau|_Y(z) = \tau(z)$. Similarly, we write $\sigma \simeq \tau$ if σ and τ take the same value over their common variables. If $\sigma \in D^Y$ and $\tau \in D^X$ are defined such that $\tau \simeq \sigma$, we write $\tau \bowtie \sigma$ for the tuple over $D^{X \cup Y}$ defined as:

$$(\sigma \bowtie \tau)(x) = \begin{cases} \sigma(x) \text{ if } x \in Y \\ \tau(x) \text{ otherwise.} \end{cases}$$

which is well defined since $\sigma(z) = \tau(z)$ for every $z \in X \cap Y$. When $X \cap Y = \emptyset$, we always have $\sigma \simeq \tau$ and use notation $\sigma \times \tau$ instead to emphasize the disjointness.

A *relation R on variables X and domain D* is a set of tuples, that is, a subset of D^X. When not explicitly stated, we use the notation $\mathsf{dom}(R)$ to denote the domain R is defined on, that is, $\{\tau(x) \mid \tau \in R, x \in X\}$. We extend the notation from Boolean functions to relations: if $R \subseteq D^X$ and $S \subseteq D^Y$ are two relations, we define the natural join of R and S, denoted by $R \bowtie S$ as $R \bowtie S := \{\tau \bowtie \sigma \mid \tau \in R, \sigma \in S, \tau \simeq \sigma\}$. If X and Y are disjoint, we write $R \times S$ to emphasize their disjointness. Observe that $|R \times S| = |R| \cdot |S|$. If both R and S are over the same set of variables X, we let $R \cup S$ be the union of the two relations, that is, $\tau \in R \cup S$ if and only if $\tau \in R$ or $\tau \in S$. If, in addition, R and S have disjoint tuples, that is, $R \cap S = \emptyset$, we will write $R \uplus S$ to emphasize it. In this case, we have $|R \uplus S| = |R| + |S|$.

Also, given a set of variables Y and a relation R over X, we denote by $R|_Y = \{\tau|_Y \mid \tau \in R\}$ (which is sometimes denoted by $\pi_Y(R)$ in relational algebra). Observe that, by definition, $R|_Y$

is a relation over $X \cap Y$. Finally, given a tuple τ over Y, we let R/τ be the relation over $X \setminus Y$ defined as $\{\sigma \mid \sigma \times (\tau|_X) \in R\}$, which is often denoted in relational algebra by $\pi_{X \setminus Y}(\sigma_{Y=\tau}(R))$. While this notation may be surprising at first, observe that it is natural as it is some kind of "inverse" of Cartesian product, hence we denote it with a division symbol. One way of understanding the notation on an intuitive level is as follows: we have $\sigma \in R/\tau$ if and only if $\sigma \times \tau \in R$, which can be seen as "multiplying" by τ on both sides.

We do a few observations on the notation:

Lemma 4.1. *Let R and S be relations over variables X and Y respectively and let τ be a tuple over Z. We have $(R \bowtie S)/\tau = (R/\tau) \bowtie (S/\tau)$. Moreover, if $Z \supseteq X \cap Y$, $(R \bowtie S)/\tau = (R/\tau) \times (S/\tau)$.*

Proof. Let $\sigma \in (R \bowtie S)/\tau$. By definition, σ is over $(X \cup Y) \setminus Z$ and $\alpha := \sigma \times \tau|_{X \cup Y} \in (R \bowtie S)$. That is $\alpha|_X \in R$ and $\alpha|_Y \in S$. By definition, $\alpha|_X = \alpha|_{X \setminus Z} \times \alpha|_{X \cap Z}$. But since σ does not assign any variable from Z, we have $\alpha|_{X \cap Z} = \tau|_{X \cap (X \cup Y)} = \tau|_X$. In other words, $\alpha|_X = \alpha|_{X \setminus Z} \times \tau|_X$, hence $\alpha|_{X \setminus Z} \in R/\tau$. Similarly, we can show $\alpha|_{Y \setminus Z} \in S/\tau$. But then $\sigma = \alpha|_{X \setminus Z} \bowtie \alpha|_{Y \setminus Z}$ which shows $\sigma \in (R/\tau) \bowtie (S/\tau)$.

For the other way around, let $\sigma \in (R/\tau) \bowtie (S/\tau)$. We need to show that $\sigma \times \tau|_{X \cup Y} \in R \bowtie S$. In other words, $\sigma_1 := \sigma|_{X \setminus Z} \in R/\tau$ and $\sigma_2 := \sigma|_{Y \setminus Z} \in S/\tau$. Hence $\alpha_1 := \sigma_1 \times \tau|_X \in R$, $\alpha_2 := \sigma_2 \times \tau|_Y \in S$. We can conclude since $\sigma \times \tau|_{X \cup Y} = \alpha_1 \bowtie \alpha_2 \in R \bowtie S$.

If $Z \supseteq X \cap Y$, then observe that R/τ and S/τ are relations defined over variables $X' := X \setminus Z$ and $Y' := Y \setminus Z$ respectively and that $X' \cap Y' \subseteq (X \cap Y) \setminus Z = \emptyset$. Hence the righthandside of the equality is actually a Cartesian product. □

Given what we have said on notation R/τ, Lemma 4.1 may seem surprising since $\frac{a}{c} \times \frac{b}{c}$ is not equal to $\frac{a \times b}{c}$ for rational numbers. But it makes more sense if one recalls that a tuple is also a product of unit tuples $\langle x/d \rangle$. In this case, if τ is a tuple over $Z \subseteq X \cup Y$ and if we see τ decomposed as $\tau_1 \bowtie \tau_2$ where $\tau_1 = \tau|_X$ and $\tau_2 = \tau|_Y$, then for relations R and S over X and Y respectively, the equality from Lemma 4.1 can be rewritten as $(R \bowtie S)/(\tau_1 \bowtie \tau_2) = (R/\tau_1) \bowtie (S/\tau_2)$ which then corresponds to the intuition that $\frac{a}{c} \times \frac{b}{d} = \frac{ab}{cd}$.

We will need this last equality:

Lemma 4.2. *Let R be a relation over variables X and τ be a tuple over variables $Z \subseteq X$ and let Y be a set of variables such that $Z \subseteq Y \subseteq X$. We have*

$$R/\tau = \biguplus_{\substack{\sigma \in R|_Y \\ \sigma \simeq \tau}} (R/\sigma) \times \sigma|_{Y \setminus Z}.$$

Proof. Let $\alpha \in R/\tau$. Let $\alpha_1 = \alpha|_{X \setminus Y}$ and $\alpha_2 = \alpha|_{Y \setminus Z}$. Since $\alpha = \alpha_1 \times \alpha_2$, we need to show that $\alpha_1 \in R/\sigma$ and $\alpha_2 = \sigma|_{Y \setminus Z}$ for some $\sigma \in R|_Y$ with $\sigma \simeq \tau$.

To this end, we let $\sigma = (\alpha \times \tau)|_Y$. Clearly, $\sigma \simeq \tau$. Moreover, since $\alpha \in R/\tau$, we have $\alpha \times \tau \in R$ and hence $\sigma \in R|_Y$. Finally, we clearly have $\alpha_2 = \sigma|_{Y \setminus Z}$ and $\alpha_1 \times \sigma = \alpha \times \tau$, hence $\alpha_1 \times \sigma \in R$, that is, $\alpha_1 \in R/\sigma$.

Now let $\sigma \in R|_Y$, with $\sigma \simeq \tau$ and let $\beta \in R/\sigma$. We need to show that $\beta \times \sigma|_{Y \setminus Z} \in R/\tau$. Since $Z \subseteq Y$, $\sigma|_Z = \tau$. Hence $\sigma|_{Y \setminus Z} \times \tau = \sigma$. In other words, $\beta \times \sigma|_{Y \setminus Z} \times \tau = \beta \times \sigma \in R$ since $\beta \in R/\sigma$, which concludes the proof of equality.

It remains to prove the union is disjoint. Assume there exists σ_1, σ_2 and some α such that $\alpha \in (R/\sigma_1) \times \sigma_1|_{Y \setminus Z}$ and $\alpha \in (R/\sigma_2) \times \sigma_2|_{Y \setminus Z}$. We immediately have $\sigma_1|_{Y \setminus Z} = \sigma_2|_{Y \setminus Z}$. Now, $\sigma_1|_Z = \tau = \sigma_2|_Z$ by definition. Hence $\sigma_1 = \sigma_2$. □

Join queries. While databases have a rich history of separating database schemas and the actual data, and define how this data can be queried using logical devices, we will deviate heavily from this point of view for clarity reasons. Our main goal in this chapter is not to offer a thorough introduction to the theory of databases (interested readers can consult [Are+22] or other good references such as [Lib04; AHV95]) but to explain how one can represent relations succinctly and how it connects to some previous work in database theory. To this end, we try to keep the required definitions as minimal as possible. Hence, we will not properly define the notion of database schemas (i.e., a collection of relation names with arity), nor what exactly is a database over such a schema (usually understood as an interpretation of said relation names by actual relations). We will only peer through them via the notion of join and conjunctive queries.

A *join query* $Q = R_1, \ldots, R_m$ is a finite list of relations $R_1, \ldots, R_m$ over respective variable sets $X_1, \ldots, X_m$ and domain D. The *answers of* Q, denoted by $[\![Q]\!] \subseteq D^X$, is the relation over variables $\mathsf{var}(Q) := X_1 \cup \cdots \cup X_m$ and domain $\mathsf{dom}(Q) := \mathsf{dom}(R_1) \cup \cdots \cup \mathsf{dom}(R_m)$ defined as $R_1 \bowtie \ldots \bowtie R_m$. It is easy to check that $[\![Q]\!]$ can alternatively be defined as the set of tuples $\tau \in D^X$ such that for every $i \leq m$, $\tau|_{X_i} \in R_i$. We will often write a join query as $R_1(X_1), \ldots, R_m(X_m)$ to implicitly name the variables of each relation in Q. The order of atoms in Q being not relevant to its semantics, we often identify Q with a multiset too, and use related notations, like $R \in Q$ to denote a relation appearing in Q. Such a relation $R \in Q$ is called *an atom of* Q.

We extend some notation on relations over join queries. For a set of variables Y, we let $Q|_Y = \{R|_Y \mid R \in Q\}$. We have to be a bit careful with this notation however since $[\![Q]\!]|_Y \subseteq [\![Q|_Y]\!]$ but equality does not hold. For example, let $Q = R, S$ with $R = \{\langle x/0 \rangle, \langle y/0 \rangle\}$ and $S = \{\langle x/0 \rangle, \langle y/1 \rangle\}$. Clearly $[\![Q]\!] = \emptyset$ but $[\![Q|_{\{x\}}]\!] = \{\langle x/0 \rangle\}$. For a tuple τ, we also let $Q/\tau = \{R/\tau \mid R \in Q\}$. This time, however, we have:

Lemma 4.3. *For every join query Q and tuple τ, $[\![Q/\tau]\!] = [\![Q]\!]/\tau$.*

Proof. If $\sigma \in [\![Q/\tau]\!]$, then $(\sigma \times \tau)|_{\mathsf{var}(R)} \in R$ for every $R \in Q$, hence $\sigma \in [\![Q]\!]/\tau$.

Similarly, if $\sigma \in [\![Q]\!]/\tau$, then $\sigma \times \tau \in [\![Q]\!]$. In particular, $(\sigma \times \tau)|_{\mathsf{var}(R)} \in R$ for every $R \in Q$, hence $\sigma|_{\mathsf{var}(R)} \in R/\tau$, that is, $\sigma \in [\![Q]\!]/\tau$. □

A *conjunctive query* $Q(S)$ is given by a join query Q on variables X and a subset S of X, called the *free variables* of Q. The *answers of* $Q(S)$, denoted by $[\![Q(S)]\!] \subseteq D^S$, is defined as the projection of $[\![Q]\!]$ onto the variables in S, that is, $\{\tau|_S \mid \tau \in [\![Q]\!]\}$.

We consider two notions of sizes for join queries, inherited from the original separation between the logical layer and the data layer. The *query size of* Q, denoted by $|Q|$ is defined as $\sum_{R \in Q} |\mathsf{var}(R)|$. The query size does not take into account the content of the relation, but only their structure, for which the number of variables is a proxy. The *data size of* Q, denoted by $\|Q\|$, is defined as $\sum_{R \in Q} |\mathsf{var}(R)| \cdot |R|$. The data size is sensitive to the number of tuples

R	x	y
	0	0
	1	1
	2	2
	0	1

S	y	z
	2	0
	2	1
	1	1

T	x	z
	1	1
	2	1
	2	0

$[\![Q]\!]$	x	y	z
	1	1	1
	2	2	0
	2	2	1

$[\![Q(x,y)]\!]$	x	y
	1	1
	2	2

Figure 4.1: Relations as tables and the answers of a join query $Q = R \bowtie S \bowtie T$ and of the conjunctive query $Q(x, y)$.

in each relation. If we assume the RAM model and that each value of the domain fits in one register, then it corresponds (up to some multiplicative constant) to the number of registers needed to encode the join query. We extend these definitions to conjunctive queries, where $|Q(S)|$ is defined as $|Q|$ and $\|Q(S)\|$ as $\|Q\|$.

As for CNF formulas, the way variables and relations interact with one another in a join query may be used to design faster algorithms for computing $[\![Q]\!]$. Given a join query Q, we will mostly study its structure via its *hypergraph* $\mathcal{H}(Q)$ defined as follows: the variables of $\mathcal{H}(Q)$ are the variables $\mathsf{var}(Q)$ of Q and the edges are $\{\mathsf{var}(R) \mid R \in Q\}$. Observe that the size of $\mathcal{H}(Q)$ is exactly the query size of Q.

Example 4.4. Fig. 4.1 shows three relations $R(x, y)$, $S(y, z)$ and $T(x, z)$ over domain $\{0, 1, 2\}$ depicted as tables together with the answers of join query $Q = R(x, y), S(y, z), T(x, z)$ and conjunctive query $Q(x, y)$. The query size $|Q|$ of Q is $3 \times 2 = 6$ and the data size is $2 \times 4 + 2 \times 3 + 2 \times 3 = 20$.

Observe that relations are not multisets, hence even if there are two tuples in $[\![Q]\!]$ setting both $x = 2$ and $y = 2$, $[\![Q(x, y)]\!]$ contains exactly one tuple with $\langle x/2, y/2\rangle$. This observation hints at something important in this chapter: while deciding whether $[\![Q]\!] \neq \emptyset$ is equivalent to deciding whether $[\![Q(x, y)]\!] \neq \emptyset$, the associated counting problems of computing $|[\![Q]\!]|$ and computing $|[\![Q(x, y)]\!]|$ may be different.

Observation 1. *In this chapter, we will slightly deviate from the usual terminology of knowledge compilation. The term query being employed in both databases and knowledge compilation for different notions, we will reserve the word query for databases queries. When referring to "tractable queries" on a class of circuits, in the meaning from knowledge compilation, we will use the word* task.

Combined and data complexity. In a database setting, we are interested in evaluating join queries that have the same hypergraph but on relations of varying sizes. In a typical

scenario, the query size of Q is far smaller than its data size. With this observation, it is often assumed in complexity analysis, that the query size is constant and only the data size is taken into account. A complexity bound on an algorithm given under this assumption will be referred to as the *data complexity of the algorithm*. In particular, in data complexity, the number of variables and atoms of a query are considered constant. When the complexity bound does not make this assumption, and when it is necessary to be explicit about it, we will say that it is the *combined complexity of the algorithm*.

As an illustration, we observe that the problem of deciding, given a join query Q, whether $[\![Q]\!] \neq \emptyset$, is NP-complete. That said, the problem has a polynomial time algorithm in data complexity since we can simply brute force over every possible answer tuple. There are at most $|\mathsf{dom}(Q)|^{|\mathsf{var}(Q)|}$ such tuples, which is polynomial if $|\mathsf{var}(Q)|$ is considered constant.

While the data complexity assumption is reasonable in many cases, we find it unsatisfactory as it may hide (or lead the reader to think it hides) an exponential dependency in the query size. Moreover, if a theorem only states the data complexity of some problem, it makes the result hard to reuse outside the realm of databases where the data complexity assumption may not be as relevant. For example, one can easily encode a 3-clause into a relation with three variables and of size at most 7. Hence, one can encode a 3-CNF F into a join query Q_F, but in this case, the query size of Q_F is linear in the size of F. If one wants to use an algorithm for join queries on 3-CNF formulas using this reduction, data complexity is not precise enough to pinpoint the exact complexity of the approach. For these reasons, we will work at a somewhat intermediate level in this chapter, which is sometimes referred to as the *fine-grained complexity*. The complexity bounds that we explicitly give will always be stated in combined complexity but we will try to separate, if possible, the query size from the data size in the notation so that the data complexity is easy to extract. Moreover, we will try to be as precise as possible on the exponent of the data size part, while being looser on the query size. To this end, we will use notation $\mathrm{O}_{\mathrm{poly}}(f)$ to ignore polynomial factors in $|Q|$, that is, $g = \mathrm{O}_{\mathrm{poly}}(f)$ if there exists a polynomial p such that $g \leq p(|Q|) \cdot f$. We sometimes write $\mathrm{poly}(|Q|)$ explicitly to insist on the fact that the function is polynomial in $|Q|$. For example, we will write $\mathrm{O}_{\mathrm{poly}}(\|Q\|)$ to describe the complexity of an algorithm that is linear in data complexity and polynomial in $|Q|$.

Sometimes, the complexity depends on factors polynomial in $\log \|Q\|$. These factors are sometimes induced by the model of computation we choose to implement the algorithms on. For example, sorting an array of size n may be done with $O(n \log n)$ comparisons but can be achieved in time $O(n)$ on the RAM model using radix sort. Or these factors are induced by the algorithm itself, for example, via a binary search or because of a particular encoding. In this chapter, we will try to be as explicit as possible on the source of polylogarithmic factors. That said, from the fine-grained perspective, we are mostly interested in the exponent of $\|Q\|$, hence, we can ignore these factors. We therefore write $g = \tilde{\mathrm{O}}(f)$ instead of $\mathrm{poly}(\log \|Q\|) \cdot f$, meaning that there exists a polynomial p such that $g \leq p(\log \|Q\|) \cdot f$. We combine both of the previous notation as $g = \tilde{\mathrm{O}}_{\mathrm{poly}}(f)$ to denote that $g \leq p(\mathrm{poly}(|Q|) \log \|Q\|) \cdot f$ for some polynomial p.

Acyclic hypergraphs and queries. The complexity of answering a join query Q largely depends on its structure, that we study through its hypergraph $\mathcal{H}(Q)$. An important class of queries in the literature which enjoys efficient algorithms is the class of *acyclic queries* which have been introduced by Yannakakis [Yan81]. Acyclic queries are defined via the notion of tree decomposition of their hypergraphs which we already partially covered for graphs when defining treewidth in Chapter 2. We extend the notion to hypergraphs as follows: given a hypergraph $H = (V, E)$, a *tree decomposition* $\mathcal{T}$ of H is a tree where each node t is labeled by $B_t \subseteq V$ and respecting the following properties:

- For every $e \in E$, there is a node t in $\mathcal{T}$ such that $e \subseteq B_t$,
- For every $x \in V$, the set $\{t \mid x \in B_t\}$ is a connected subtree of $\mathcal{T}$ (we refer to this property as the "connectedness property").

A *join tree* for a hypergraph $H = (V, E)$ is a tree decomposition $\mathcal{T}$ of H such that $\mathcal{T}$ has exactly $|E|$ nodes and for every e, there exists exactly one node t of $\mathcal{T}$ such that $B_t = e$. In other words, a join tree is a way of organizing the edges of H into a tree which respects the connectedness property. A hypergraph is said to be α-acyclic if it has a join tree. A join query Q is said to be α-acyclic if its hypergraph is α-acyclic.

Example 4.5. Consider query

$$Q = R(x, y, z), T_1(x, y), T_2(y, z), T_3(z, x), S_1(x, y, t), S_2(x, z, u).$$

Fig. 4.2 depicts a join tree for Q. It is labeled with the name of the atoms to make the mapping explicit though it is not required by the definition. One can check that the connectedness property holds.

Now consider query $Q' = T_1(x, y), T_2(y, z), T_3(z, x)$. We will show that Q' is not α-acyclic. Indeed, assume that there is a join tree $\mathcal{T}$ for Q' and assume it is rooted in the node labeled by $\mathsf{var}(T_1) = \{x, y\}$. Now, either this node has two children labeled respectively by $\mathsf{var}(T_2), \mathsf{var}(T_3)$. In this case, the connectedness of z is not ensured since it is not in $\mathsf{var}(T_1)$. Otherwise, the root has one child t'. Assume t' is labeled by $\mathsf{var}(T_2)$. Then t' has one child labeled by $\mathsf{var}(T_3)$. In this case, x being in $\mathsf{var}(T_1) \cap \mathsf{var}(T_3)$ but not in $\mathsf{var}(T_2)$, its connectedness is not respected. If t' is labeled by $\mathsf{var}(T_3)$, we have a symmetric situation where the connectedness of y is not respected.

One consequence of Example 4.5 is that α-acyclicity is not hereditary. Indeed, observe that $Q' \subseteq Q$, but Q is α-acyclic and Q' is not. This is a surprising phenomenon as the intuition regarding acyclicity we have from graphs is that acyclicity should be preserved when we take subhypergraphs. It makes sense from a complexity point of view however. It may be that subqueries are harder to answer than the query itself. Consider, for example, any join query Q over variables X and a similar query $Q' = Q, R'(X)$. Observe in particular that Q' is α-acyclic: we can build a join tree whose root is labeled by X and which has one child for each atom of Q. The root containing every variable, the connectedness property is respected. But more

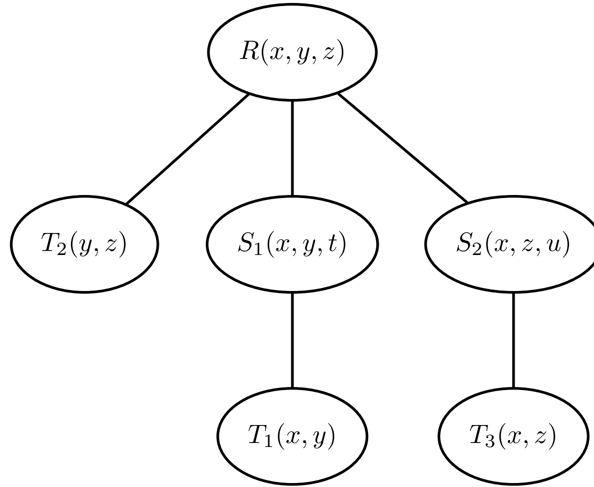


Figure 4.2: A join tree for query $Q = R(x,y,z), T_1(x,y), T_2(y,z), T_3(z,x), S_1(x,y,t), S_2(x,z,u)$.

importantly, observe that Q' is trivial to solve: one can simply list every tuple in $R'(X)$ and check whether they are answers of Q.

In some cases however, we will need a stronger notion of acyclicity that is hereditary. Namely, the β-acyclicity. A hypergraph H is *β-acyclic* if every subhypergraph $H' \subseteq H$ (in the sense that H' contains a subset of the edges of H) is α-acyclic. A query is *β-acyclic* if $H(Q)$ is β-acyclic. Observe that Q from Example 4.5 is not β-acyclic because $Q' \subseteq Q$ is not α-acyclic.

This definition of β-acyclicity is however not really usable algorithmically since it does not give much insight into the structure of the query. A characterization of β-acyclicity is more interesting with this respect. Given a hypergraph $H = (V, E)$, a *nest point* $x \in V$ is a vertex of H such that the set $E_x := \{e \in E \mid x \in e\}$ forms a chain for the inclusion, that is, it is equal to $\{e_1, \dots, e_k\}$ with $e_1 \subseteq \cdots \subseteq e_k$. A *$\beta$-elimination order* is an ordering of $V = (v_1, \dots, v_n)$ such that for every $i \leq n$, v_i is a nest point of $H[v_i, \dots, v_n]$. We have:

Lemma 4.6 ([Bee+83])**.** *A hypergraph is β-acyclic if and only if it has a β-elimination order.*

This characterization can be seen as the following well-known characterization of graph acyclicity: a graph is acyclic if and only if one can iteratively remove leaves (ie vertices of degree 1) in the graph until it is empty. We have the same phenomenon here but with a different notion of "leaf". An equivalent characterization actually exists for α-acyclicity. Given a hypergraph $H = (V, E)$, a *α-leaf* $x \in V$ is a vertex such that there exists $f \in E_x := \{e \in E \mid x \in e\}$ such that for every $e \in E_x$, $e \subseteq f$. A *α-elimination order* is an ordering of $V = (v_1, \dots, v_n)$ such that for every $i \leq n$, v_i is an α-leaf of $H[v_i, \dots, v_n]$. We have:

Lemma 4.7 ([Bee+83; TY84])**.** *A hypergraph is α-acyclic if and only if it has an α-elimination order.*

Example 4.8. Consider query $Q = R(x_1, x_2, x_3), S(x_2, x_3), T(x_1, x_3)$ and observe that (x_1, x_2, x_3) is a β-elimination order of $\mathcal{H}(Q)$ but not (x_3, x_1, x_2) because x_3 is contained in edges $\{x_2, x_3\}$ and $\{x_1, x_3\}$ which are not included in one another. However, (x_3, x_1, x_2) is an α-elimination order because $\{x_1, x_2, x_3\}$ includes both $\{x_1, x_2\}$ and $\{x_2, x_3\}$ hence x_3 is an α-leaf in $\mathcal{H}(Q)$.

It is not too hard to see that removing a nest point or an α-leaf can only create more nest points or α-leaves. Hence, a greedy algorithm which iteratively removes nest points (respectively α-leaf) gives a polynomial time algorithm to test for β-acyclicity (respectively α-acyclicity). This approach is, however, not the best known, as it is known that finding β-elimination orders and join trees can be done more efficiently:

Theorem 4.9 ([TY84])**.** *There is an algorithm that, given $H = (V, E)$, constructs an α-elimination order and a join tree of H in time $O(|V|+|E|)$ if H is α-acyclic and fails otherwise.*

Theorem 4.10 ([PT87])**.** *There is an algorithm that given $H = (V, E)$ constructs a β-elimination order of H in time $O(|H| \cdot |V|)$ if H is β-acyclic and fails otherwise.*

Both α-acyclicity and β-acyclicity degenerate to the usual graph acyclicity when applied to hypergraphs having edges of size 2 (that is, graphs). Other generalizations of acyclicity for hypergraphs have been studied in the literature, and we invite the reader to consult the seminal paper by Fagin [Fag83] and the survey [Bra14] by Brault-Baron which contains a very thorough presentation of these notions, their different characterizations and their relations. The vast majority of results in database theory concerning acyclic queries being about α-acyclic queries, the prefix is often omitted. We will follow the same convention in this chapter, meaning that, unless specified otherwise, an acyclic query is defined as an α-acyclic query.

4.2 Relational Circuits

4.2.1 Yet another class of circuits

In this section, we introduce our main tool: the notion of *relational circuits*. For the reader of this document, it will appear obvious that these circuits are unnecessary generalizations of the restricted version of Boolean circuits introduced in the previous chapters to finite domains larger than $\{0, 1\}$. Indeed, we can always encode any value from a finite domain of size d with $(\log d)$ bits, hence using as many Boolean variables.

We would like to take the time, in this subsection, to explain a few reasons why we feel it is needed to go over the definitions presented in Chapter 1 again, but with a different perspective. Most of them are not scientific but pedagogical and cultural.

The first reason is that the notion of Boolean circuits comes with some implicit assumptions that may not be obvious from the perspective of relational algebra. The usual point of view on DNNF circuits and their cousins is that, as restrictions of Boolean circuits, they compute Boolean functions. In databases, one of the main objects of study has been the notion of relations and their algebraic nature, at least since the seminal paper of Codd [Cod70]. We

will therefore see the computation of a relational circuit not as the computation of a function but as a device to *succinctly represent a relation* by representing it as a factorized algebraic expression. This is a nitpick but it has some impact. One illustrative example is the case of $\vee$-gates. In the Boolean function view, $f \vee g$ is naturally defined as a Boolean function over $X \cup Y$ if f is over variables X and g is over variables Y. In the relational view, $f \vee g$ corresponds to building the union of two tables. From the relational algebra perspective, the union is only defined if both relations are defined on the same attributes, which corresponds to smoothness in NNF circuits. This is because in the case of relational algebra, the domain is not fixed, hence, it has to be made explicit for an operator such as $\vee$ to make sense. The Boolean circuit approach may introduce some confusion for communities that are used to manipulate relations and tuples instead, not because it is conceptually hard, but because it sometimes fails to be explicit about some implicit assumptions.

To be extremely explicit, we introduce a new operator on relations, that we call the *extended union*. Given relations $R \subseteq D^X$ and $S \subseteq D^Y$, the *extended union of R and S over domain D*, denoted by $R \mathbin{\bar{\cup}}_D S$ is the relation over $X \cup Y$ defined as $(R \times D^{Y \setminus X}) \cup (S \times D^{X \setminus Y})$. We may simply write $R \mathbin{\bar{\cup}} S$ when D is clear from context.

Another reason for duplicating the definitions in some ways is that the historical terminology and notations from knowledge compilation are often puzzling for newcomers. One example we already mentioned in Chapter 2 is that the way the word determinism is used for Boolean circuits is actually not what most people understand by determinism and is closer to what other communities have called *unambiguity*. Moreover, the multiplication of acronyms to denote classes of circuits may not be the most welcoming habit for newcomers, especially when many terms start with letter d. In this chapter, we will try to name classes of circuits in a more readable and transparent way, a habit we tried to enforce in earlier work and survey [AC24; CI24]. We will systematically name classes of circuits by explicitly listing the gates they are allowed to use.

4.2.2 Main definitions

For the reasons described in the previous section, we take the time to properly introduce a relational interpretation of DNNF circuit for non-binary domains. As for DNNF circuits, we start with a very general notion of circuits only to refine it later. A $\{\bowtie, \bar{\cup}\}$-circuit on variables X and domain D is a circuit whose gates are labeled as follows:

- Every input is labeled by either $\top$, $\bot$, or $\langle x/d \rangle$ for $x \in X$ and $d \in D$,
- Every internal gate is labeled by either $\cup$ or by $\bowtie$.

Let C be a $\{\bowtie, \bar{\cup}\}$-circuit on variables X and domain D. Given a gate g of C, we let $\mathsf{var}(g)$ be the variables labeling the inputs of C_g, that is, the subcircuit of C rooted at g. Each gate of C computes a relation over $D^{\mathsf{var}(g)}$ defined inductively as follows:

- Input gates labeled by $\top$ compute the relation containing only the empty tuple, those labeled by $\bot$ compute the empty relation and those labeled by $\langle x/d \rangle$ compute the relation $R = \{\langle x/d \rangle\}$.

- If g is a $\bar{\cup}$-gate with inputs $g_1, \dots, g_k$, then g computes $\mathsf{rel}(g_1) \bar{\cup}_D \dots \bar{\cup}_D \mathsf{rel}(g_k)$.
- If g is a $\bowtie$-gate with inputs $g_1, \dots, g_k$, then g computes $\mathsf{rel}(g_1) \bowtie \dots \bowtie \mathsf{rel}(g_k)$.

As for Boolean circuits, by their very definition, $\{\bowtie, \bar{\cup}\}$-circuit can be conditioned efficiently, that is, given $Y \subseteq X$ and $\tau \in D^Y$, we can compute the set of tuples of $\mathsf{rel}(C)$ which agree with τ. Indeed, we simply need to relabel any input gate labeled by $y = d$ with $y \in Y$ and $d \neq \tau(y)$ with the $\bot$ label. In relational algebra, this operation is called a selection and is denoted by $\sigma_{Y=\tau}(R)$.

Smoothness. As hinted in the previous section, smoothness is very natural in the case of relational circuits and removes the need to be explicit about the domain. A $\{\bowtie, \bar{\cup}\}$-circuit is smooth if for every $\bar{\cup}$-gate g with input $g_1, \dots, g_k$, we have for every $i \leq k$, $\mathsf{var}(g) = \mathsf{var}(g_i)$. In this case, observe that $\mathsf{rel}(g) = \mathsf{rel}(g_1) \cup \dots \cup \mathsf{rel}(g_k)$ and this interpretation does not depend on making the domain D explicit as needed for $\bar{\cup}$. We will indicate smoothness in the notation by writing $\{\bowtie, \cup\}$-circuit to denote smooth $\{\bowtie, \bar{\cup}\}$-circuit.

Decomposability. Since the combined complexity of deciding whether $R_1 \bowtie \dots \bowtie R_m$ is not empty is NP-complete, $\{\bowtie, \bar{\cup}\}$-circuits are not particularly useful as a way of representing relations, mirroring Boolean circuits, in that they are too general to allow for any tractable tasks. We get some tractability by enforcing decomposable $\bowtie$-gates. Formally, a $\{\bowtie, \bar{\cup}\}$-circuit is decomposable if for every $\bowtie$-gate g with input $g_1, \dots, g_k$, we have that for every $1 \leq i < j \leq k$, $\mathsf{var}(g_i) \cap \mathsf{var}(g_j) = \emptyset$. From a relational-algebra point of view, a decomposable $\bowtie$-gate computes the relation $\mathsf{rel}(g_1) \times \dots \times \mathsf{rel}(g_k)$. We therefore use the following notation: an $\{\times, \bar{\cup}\}$-circuit is a $\{\bowtie, \bar{\cup}\}$-circuit such that every $\bowtie$-gate is decomposable. Following our naming convention, an $\{\times, \cup\}$-circuit is a $\{\bowtie, \bar{\cup}\}$-circuit that is both smooth and decomposable.

As for DNNF circuits, decomposability is enough to ensure that one can efficiently test emptiness of the relation represented by an $\{\times, \bar{\cup}\}$-circuit. It also suffices to allow for efficient enumeration of the tuples in $\mathsf{rel}(C)$ [Ama+24].

Determinism. In Boolean circuits, determinism translates to the fact that an assignment satisfies at most one input of a given $\vee$-gate. From the relational perspective, it means that for a $\cup$-gate, a tuple may belong to at most one of its input relations. In other words, it means that the union is disjoint. More explicitly, a $\bar{\cup}$-gate g with input $g_1, \dots, g_k$ is deterministic if for every $\tau \in \mathsf{rel}(g)$, there exists a unique $i \leq k$ such that $\tau \in D^{\Delta_i} \times \mathsf{rel}(g_i)$ where $\Delta_i = \mathsf{var}(g) \setminus \mathsf{var}(g_i)$. We will denote an extended disjoint union and a disjoint union by $\bar{\uplus}$ and $\uplus$ respectively. As before, a $\{\times, \uplus\}$-circuit is a deterministic $\{\bowtie, \cup\}$-circuit, i.e., a smooth, decomposable and deterministic $\{\bowtie, \bar{\cup}\}$-circuit.

As for DNNF circuits, determinism makes the computation of $|\mathsf{rel}(C)|$ tractable as one can evaluate it with a naive bottom-up dynamic programming algorithm.

Decision-gates. Determinism is a semantic property and in practice, it is often enforced syntactically via decision-gates, as in OBDD or decision-DNNF circuits. From a purely syntactic point of view, a decision-gate g on variable x is simply a $\uplus$-gate such that each input

of g is of the form $\langle x/d\rangle \times g_i$ where $d \in D$ and such that no other input of g uses the same value d. It will be handy though to have a specific type of gates for such behavior. Hence, we allow our circuit to have another type of gate, called *decision-gate*, and are defined as follows: a *decision-gate g on variable x and domain D* is a gate labeled by x and whose incoming edges are labeled by elements from D. For each $d \in D$, there is at most one incoming edge of g labeled d. The relation computed by g is defined as $\langle x/d_1\rangle \times \mathsf{rel}(g_1) \bar{\cup} \ldots \bar{\cup} \langle x/d_k\rangle \times \mathsf{rel}(g_k)$ where the input edges of g are $(g_1, g), \ldots, (g_k, g)$ and are labeled by $d_1, \ldots, d_k$ respectively and where we assume that $x \notin \mathsf{var}(g_i)$.

A $\{\times, \overline{\mathsf{dec}}\}$-circuit is a circuit using only $\times$-gates and decision-gates. We do not necessarily enforce smoothness, though sometimes, it will be handy to have it, or it will be guaranteed by our compilation algorithm. In this case, we will write $\{\times, \mathsf{dec}\}$-circuit, omitting the overline on dec.

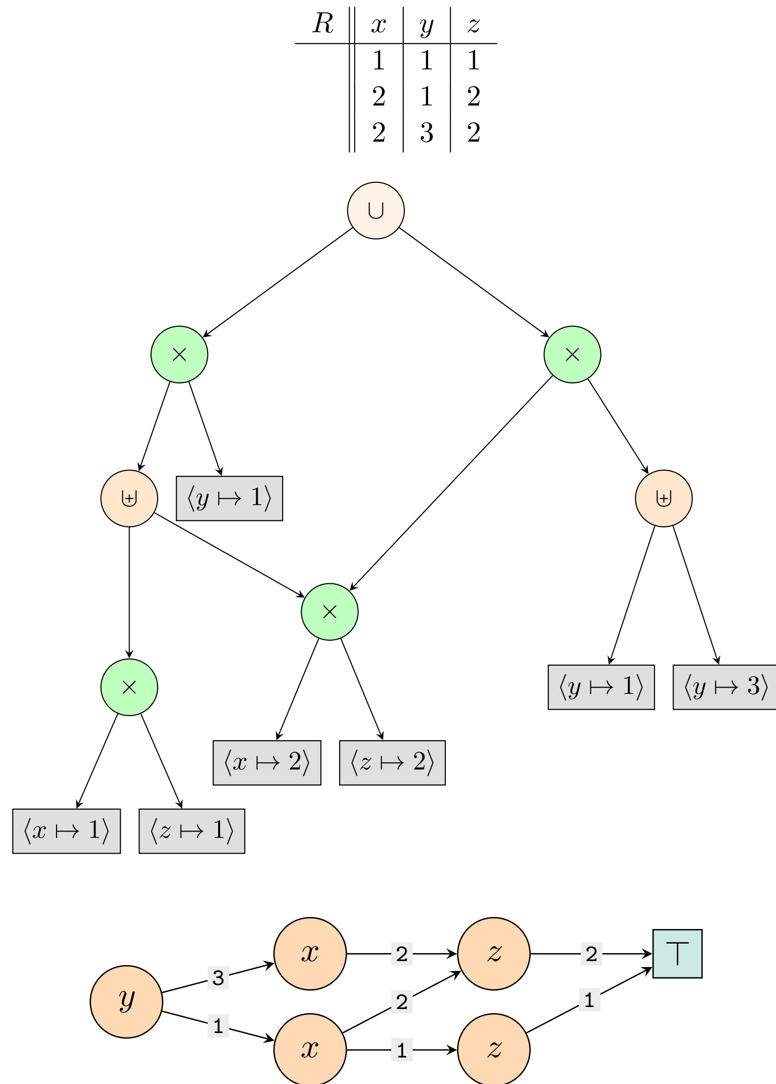
Structure and order. We extend the notion of structuredness of DNNF circuits to $\{\times, \bar{\cup}\}$-circuits. We say that a $\{\times, \bar{\cup}\}$-circuit C on variables X *respects a vtree T over variables X* if for every $\times$-gate g of C, g has exactly two inputs g_1, g_2 and there exists a node t of T with children t_1, t_2 such that $\mathsf{var}(g_1) \subseteq \mathsf{var}(t_1)$ and $\mathsf{var}(g_2) \subseteq \mathsf{var}(t_2)$.

Similarly, we have a structure restriction for $\{\times, \mathsf{dec}\}$-circuits akin to the one for OBDD. By default, $\{\times, \mathsf{dec}\}$-circuits may use different orders on different branches. We introduce a syntactic restriction of $\{\times, \mathsf{dec}\}$-circuits called *ordered $\{\times, \mathsf{dec}\}$-circuit* as follows: we say that a *$\{\times, \mathsf{dec}\}$-circuit C on variables X respects an order $\pi = (x_1, \ldots, x_n)$ on X* if for every decision-gate g of C labeled by variable x_i and with inputs $g_1, \ldots, g_k$, we have that $\mathsf{var}(g_j) \subseteq \{x_{i+1}, \ldots, x_n\}$. If there exists an order π such that C respects π, we say that C is an ordered $\{\times, \mathsf{dec}\}$-circuit.

Example 4.11. We illustrate the previous definitions on Fig. 4.3. We give two circuits computing the same relation R depicted in the figure as well. Observe that the output of the first circuit is a $\cup$-gate which is not a $\uplus$-gate because the relation computed by its two input gates both contain the tuple $\langle x/2, y/1, z/2\rangle$. The second circuit only has decision-gates and respects order $\pi = (y, x, z)$.

4.2.3 Relational circuits, Factorized Databases and DNNFs

Knowledge compilation approaches in databases have been heavily studied since the seminal paper of Olteanu and Závodný [OZ12], under the general concept of *factorized databases.* In this document, we have decided to use the name relational circuits to describe the objects used to represent relations, in contrast to the term factorized databases which, in our opinion, describe a collection of techniques covering, in particular, relational circuit. The notion of $\{\times, \cup\}$-circuit already appears under the name *factorized representation with definitions*, or d-representations, in [OZ12; OZ15], while $\{\times, \uplus\}$-circuits have been introduced as deterministic d-representations. In [OZ15], the definition explicitly enforces smoothness as unions are only allowed over representations having the same attributes. We still decided to deviate from the



R	x	y	z
	1	1	1
	2	1	2
	2	3	2

Figure 4.3: A $\{\times, \cup\}$-circuit and a $\{\times, \mathsf{dec}\}$-circuit computing the same relation R.

original terminology because we think the name d-representation lacks precision. Indeed, it is used in [OZ15] both as a way of designing arbitrary $\{\times, \cup\}$-circuit and a particular canonical d-representation that is produced from a conjunctive query Q, a database $\mathbb{D}$ and a so-called d-tree $\mathcal{T}^{\uparrow}$.

This algorithm actually constructs a $\{\times, \mathsf{dec}\}$-circuit[3] and since it follows a d-tree which organizes the attributes of the query in a tree, one can observe that the resulting circuit is actually structured along a vtree. Hence, [OZ15] shows a way of constructing a $\{\times, \mathsf{dec}\}$-circuit respecting a vtree T and representing $Q(\mathbb{D})$, given a conjunctive query Q, a database $\mathbb{D}$ and a d-tree $\mathcal{T}^{\uparrow}$, which embeds both the structure of the vtree T and some dependency information that can be used in the compilation algorithm (see Sections 4.3.1 and 4.4 for alternative ways of obtaining the same result). But the result is not presented in this way. Instead, in [OZ15, Section 5.4], a unique canonical circuit, denoted by $\mathcal{T}^{\uparrow}(Q(\mathbb{D}))$ is defined, and a compilation algorithm to produce this canonical representation is given.

In our opinion, the terminology from [OZ15] creates confusion between the properties of the circuit produced (structured, using only decision and $\times$-gates), some sort of canonical representation and the compilation algorithm itself. This is why, in this work, we decided to decouple the notion of *relational circuits* that simply represent relations, sometimes with specific semantics or syntactic restrictions, and how such relational circuits are produced.

NNFs and relational circuits. The connection with knowledge compilation and factorized databases has been made explicit in [Olt16] but this work does not contain a thorough comparison of factorized databases and their Boolean circuits counterpart and some confusion remains around how these definitions are connected. We provide a comparison table in Table 4.1 using the notations from the previous section. Each line illustrates the correspondence between a Boolean circuit and relational circuits as follows: when considering circuits on the right column on domain $D = \{0, 1\}$, they correspond exactly to the Boolean circuit on the left.

The correspondence actually works both ways: for non-binary domain $D = \{d_0, \dots, d_{k-1}\}$, we can encode D with bitstrings over $b = \lceil \log k \rceil$ bits by representing d_i with the binary representation of i over b bits. Hence, given a $\{\bowtie, \bar{\cup}\}$-circuit C over variables $X = \{x_1, \dots, x_n\}$ and domain $D = \{d_0, \dots, d_{k-1}\}$, we can translate it into an NNF over variables $\widetilde{x}^b = \{x_1^1, \dots, x_1^b, \dots, x_n^1, \dots, x_n^b\}$ by replacing every input $\langle x_i / d_j \rangle$ by a circuit over variables $x_i^1, \dots, x_i^b$ accepting only if they encode j in binary. This can easily be done by an OBDD of size $2b$, hence, it can be encoded in any NNF subclass from Table 4.1, showing that every relational circuit on variables X and domain D from the right column can be naturally cast into an NNF from the left column with an additional increase of at most $|X| \log |D|$. We actually show in Section 4.6 that this binary encoding can actually sometimes allow to produce more compact circuits by improving which parts can be shared.

[3] It is in line with what happens in knowledge compilation too: we do not really know any algorithm capable of producing arbitrary DNNF circuits, it always produces a subclass of DNNF circuits, be it decision-DNNF circuits or structured deterministic DNNF circuits [OD17]

Knowledge Compilation	**Relational circuits**
NNF circuit	$\{\bowtie, \bar{\cup}\}$-circuit
smooth NNF circuit	$\{\bowtie, \cup\}$-circuit
DNNF circuit	$\{\times, \bar{\cup}\}$-circuit
smooth DNNF circuit	$\{\times, \cup\}$-circuit
d-DNNF circuit	$\{\times, \bar{\uplus}\}$-circuit
smooth d-DNNF circuit	$\{\times, \uplus\}$-circuit
dec-DNNF circuit	$\{\times, \overline{\mathsf{dec}}\}$-circuit
smooth dec-DNNF circuit	$\{\times, \mathsf{dec}\}$-circuit
FBDD	$\{\overline{\mathsf{dec}}\}$-circuit
smooth (complete) FBDD	dec-circuit

Table 4.1: Rosetta Stone of Tractable Circuits.

4.3 Building Relational Circuits Bottom-up

In this section, we review an algorithm for building relational circuits succinctly representing the answer set of a join query Q. It works bottom-up by exploiting a decomposition of the query. It is, in a way, a simple reformulation of the original Yannakakis algorithm [Yan81], more specifically of the counting algorithm from Skritek and Pichler [PS13], where the algebraic operations are simply translated into constructing the circuit.

4.3.1 Yannakakis Algorithm

In this section, we revisit Yannakakis Algorithm [Yan81] in terms of circuit construction, as a warm-up and an illustration of how relational circuits can be obtained. This construction is an adaptation of the counting algorithm from [PS13] and it has already been framed as a compilation algorithm by Olteanu and Závodný in [OZ15] though not in an explicit way. The Yannakakis algorithm was originally designed to test whether the answer set of a join query Q is empty or not[4] in time $\text{poly}(|Q|) \cdot \|Q\|$, hence in linear time with respect to data complexity.

Yannakakis algorithm roughly works as follows: it evaluates the join by adding relations from the leaves to the root of the join tree. If materialized explicitly, each intermediate join (and, in particular, the final one) will have a large number of answers, which would not give the desired linear complexity. The catch is to observe that while evaluating the join of the relations appearing below some node t of the join tree, every variable appearing below t but

[4]This is already a stretched but common reinterpretation of the original contribution of Yannakakis. The original paper gives an algorithm to test the consistency of a database schema but this is roughly the same problem if one considers the join query to use every relation in the schema.

not in B_t itself is irrelevant for the rest of the join computation, since they will not appear elsewhere in the tree decomposition. Hence, we can simply project the answer set up to t onto the variables in B_t and it is enough information to reconstruct each answer of Q. Now, since B_t corresponds to an atom R of Q, the intermediate join projected onto B_t will necessarily be a subset of R, hence it is of size at most $\|Q\|$, explaining why we can work in linear time in $\|Q\|$.

Before explaining the compilation algorithm, we need a small handy notation and make some observation about it. Given a relation R over X and τ a tuple over Y, we let R/τ be the relation over $X \setminus Y$ defined as $\{\sigma \mid \sigma \times (\tau|_X) \in R\}$. The notation should be understood as a "division" by τ, when seeing the Cartesian product of tuples as a product. We have the following:

4.3.2 Acyclic join queries

The goal of this section is to show the following:

Theorem 4.12. *Let Q be a join query and $\mathcal{T}$ a join tree for Q. We can build a $\{\times, \mathsf{dec}\}$-circuit computing $[\![Q]\!]$ of size $O(\|Q\|)$ in time $O(\|Q\| \cdot \mathrm{poly}(|Q|))$.*

Circuit construction. We start by describing the compilation algorithm that takes an acyclic join query Q and builds a $\{\times, \mathsf{dec}\}$-circuit computing $[\![Q]\!]$ in time $O(\mathrm{poly}(|Q|) \cdot \|Q\|)$.

Let Q be an acyclic query and $\mathcal{T}$ a join tree for Q, rooted at r. For a node t of $\mathcal{T}$, we let R_t be the atom of Q labeling node t and $Q_{\leq t}$ to be the query whose atoms are the atoms appearing below t, formally $\{R_u \mid u \in \mathcal{T}_t\}$. Observe that if t has children $t_1, \dots, t_k$, we have by definition $Q_{\leq t} = R_t, Q_{\leq t_1}, \dots, Q_{\leq t_k}$. Moreover, since $\mathcal{T}$ is a tree decomposition, $\mathsf{var}(Q_{\leq t_i}) \cap \mathsf{var}(Q_{\leq t_j}) \subseteq \mathsf{var}(R_t)$.

We build a $\{\times, \mathsf{dec}\}$-circuit computing $[\![Q]\!]$ inductively as follows: for each node t of $\mathcal{T}$, we build a circuit C_t with the following property: for every $\tau \in R_t$, there is a gate v_t^τ in C_t which computes $[\![Q_{\leq t}]\!]/\tau := \{\sigma \mid \sigma \times \tau \in [\![Q_{\leq t}]\!]\}$.

If t is a leaf of $\mathcal{T}$, then $Q_{\leq t} = \{R_t\}$ contains exactly one atom. Hence, for every $\tau \in R_t$, $[\![Q_{\leq t}]\!]/\tau = R_t/\tau = \{\langle\rangle\}$, is the singleton relation containing the empty tuple. Hence C_t is simply a $\top$-gate g and $v_t^\tau := g$ for every $\tau \in R_t$.

Now let t be an internal node of $\mathcal{T}$ with children $t_1, \dots, t_k$ and let $\tau \in R_t$. Observe that

$$\begin{aligned}
[\![Q_{\leq t}]\!]/\tau &= [\![R_t, Q_{\leq t_1}, \dots, Q_{\leq t_k}]\!]/\tau \\
&= [\![R_t]\!]/\tau \bowtie [\![Q_{\leq t_1}]\!]/\tau \bowtie \dots \bowtie [\![Q_{\leq t_k}]\!]/\tau \text{ by Lemma 4.1} \\
&= [\![Q_{\leq t_1}]\!]/\tau \bowtie \dots \bowtie [\![Q_{\leq t_k}]\!]/\tau \text{ since } \tau \in R_t \\
&= [\![Q_{\leq t_1}]\!]/\tau \times \dots \times [\![Q_{\leq t_k}]\!]/\tau \text{ since } \mathsf{var}(Q_{\leq t_i}) \cap \mathsf{var}(Q_{\leq t_j}) \subseteq \mathsf{var}(R_t).
\end{aligned} \tag{4.1}$$

So, to compute $[\![Q_{\leq t}]\!]/\tau$, we need to compute $[\![Q_{\leq t_i}]\!]/\tau$ for every $i \leq k$. Observe that it has not yet been precomputed in the circuits inductively built C_{t_i} since τ is not a tuple of R_{t_i}. But we can use Lemma 4.2 for this as follows: we first let $Z_i = \mathsf{var}(R_t) \cap \mathsf{var}(Q_{\leq t_i})$, $Y_i = \mathsf{var}(R_{t_i})$

and $\tau_i = \tau|_{Z_i}$. Observe that by connectedness, $Z_i \subseteq Y_i$. Moreover, $[\![Q_{\leq t_i}]\!]/\tau = [\![Q_{\leq t_i}]\!]/\tau_i$ since $\tau|_{\mathsf{var}(Q_{\leq t_i})} = \tau_i$. Now by Lemma 4.2, applied to τ_i, $Z_i \subseteq Y_i$, we have

$$[\![Q_{\leq t_i}]\!]/\tau = \biguplus_{\substack{\sigma \in [\![Q_{\leq t_i}]\!]|_{Y_i} \\ \sigma \simeq \tau_i}} ([\![Q_{\leq t_i}]\!]/\sigma) \times \sigma|_{Y_i \setminus Z_i}$$

Now, observe that $[\![Q_{\leq t_i}]\!]|_{Y_i} \subseteq R_{t_i}$ since R_{t_i} is an atom of $Q_{\leq t_i}$ over variables Y_i. Hence, the previous equality becomes

$$[\![Q_{\leq t_i}]\!]/\tau = \biguplus_{\substack{\sigma \in R_{t_i} \\ \sigma \simeq \tau_i}} ([\![Q_{\leq t_i}]\!]/\sigma) \times \sigma|_{Y_i \setminus Z_i}$$

Let $A_i = \{\sigma|_{Y_i \setminus Z_i} \mid \sigma \in R_{t_i}, \sigma \simeq \tau_i\}$. Observe that in this case, $A_i = R_{t_i}/\tau_i$. Hence we can once again rewrite the previous equality as:

$$[\![Q_{\leq t_i}]\!]/\tau = \biguplus_{\alpha \in R_{t_i}/\tau_i} ([\![Q_{\leq t_i}]\!]/(\alpha \times \tau_i)) \times \alpha. \tag{4.2}$$

Eq. (4.2) is interesting because by induction, we have gates in the circuit computing $[\![Q_{\leq t_i}]\!]/(\alpha \times \tau_i)$ for every $\alpha \in R_{t_i}/\tau_i$ since $\alpha \times \tau_i \in R_{t_i}$. Hence, by adding a few decision-gates to compute α, we can build a circuit computing $[\![Q_{\leq t_i}]\!]/\tau$.

Let us be more precise. Observe that the right-hand side of Eq. (4.2) only depends on $\tau_i = \tau|_{Z_i}$. So if $\tau, \sigma \in R_t$ are such that $\tau|_{Z_i} = \sigma|_{Z_i}$, we have $[\![Q_{\leq t_i}]\!]/\tau = [\![Q_{\leq t_i}]\!]/\sigma$. It suggests the following construction: for every $\beta \in R_t|_{Z_i} \cap R_{t_i}|_{Z_i}$, we build a gate r_i^β computing $[\![Q_{\leq t_i}]\!]/\beta$ as follows: r_i^β is the root of a decision tree on variables $Y_i \setminus Z_i$ whose leaves are in one to one correspondence with R_{t_i}/β. The leaf corresponding to $\alpha \in R_{t_i}/\beta$ is then identified with $v_{t_i}^{\alpha \times \beta}$, which has been constructed inductively in the circuit and computes $[\![Q_{\leq t_i}]\!]/(\alpha \times \beta)$. Gate r_i^β therefore computes

$$\bigcup_{\alpha \in R_{t_i}/\beta} ([\![Q_{\leq t_i}]\!]/(\alpha \times \beta)) \times \alpha$$

which is, by Eq. (4.2), $[\![Q_{\leq t_i}]\!]/\beta$.

Now for every $\tau \in R_t$, we define v_t^τ as follows:

- if there is some $i \leq k$ such that $\tau|_{Z_i} \notin R_{t_i}$ where $Z_i = \mathsf{var}(R_t) \cap \mathsf{var}(R_{t_i})$, then it is clear that $[\![Q_{\leq t}]\!]/\tau = \emptyset$. Hence we let $v_t^\tau = \bot$ and v_t^τ indeed computes the empty relation, which is $[\![Q_{\leq t}]\!]/\tau$.
- Otherwise, we let v_t^τ be a new $\times$-gate and connect it to gates $r_i^{\tau_i}$ that we have defined above for every $i \leq k$. By the above, it computes $\bigtimes_{i \leq k} [\![Q_{\leq t_i}]\!]/\tau_i = [\![Q_{\leq t}]\!]/\tau$ by (Eq. (4.1)) which is what we wanted.

Now, applying this construction to the root r of the tree decomposition, we obtain a $\{\times, \mathsf{dec}\}$-circuit C such that for every $\tau \in R_r$, it has a gate v_r^τ computing $[\![Q_{\leq r}]\!]/\tau = [\![Q]\!]/\tau$. Since $[\![Q]\!] = \biguplus_{\tau \in R_r} \tau \times ([\![Q]\!]/\tau)$, we just add a decision-tree on $\mathsf{var}(R_r)$ on top of C whose leaves are in one-to-one correspondence with R_r and plug the leaf corresponding to τ to v_r^τ to obtain a new gate in the circuit computing $[\![Q]\!]$.

Size analysis. It remains to analyze the size of the circuit produced by the previous algorithm. It contains at most $|R_t|$ $\times$-gates for each node t of the join tree, that is, the number of $\times$-gates is bounded by $\sum_{R \in Q} |R| \leq \|Q\|$. It remains to count the number of decision-gates in the circuit. For this, one has to be a bit more careful. Let t be a node with children $t_1, \dots, t_k$. Every decision-gate we add are part of the decision tree rooted at r_i^β we build for each $\beta \in R_{t_i}|_{Z_i}$. If we consider the leaves of all this decision trees, there is at most one per tuple of R_{t_i}, hence at most $|R_{t_i}|$ leaves. It means that the total number of decision-gates they contain is $|Y_i \setminus Z_i| \cdot |R_{t_i}| \leq |Y_i| \cdot |R_{t_i}|$. Hence in total, we need at most $\sum_t |\mathsf{var}(R_t)| \cdot |R_t| \leq \|Q\|$ decision-gates. In total, we hence have at most $2\|Q\| + 1$ gates in the circuit (the 1 being for the needed $\top$-gate).

Complexity. This circuit can be built in time $O(\|Q\| \log \|Q\|)$. Indeed, each step of the algorithm boils down to grouping the tuples in R_{t_i} depending on their values of Z_i and building a decision-tree for them. This can be done easily after sorting the tuples in R_{t_i}. In the RAM model, we can sort them in linear time in the size of each element using radix sort [Cor+22, Section 6.3]. The rest of the operations only depends on the structure of the join tree, mainly for navigating it and can all be done in poly($|Q|$).

Structure. Two structural properties can be enforced in the circuit built by this algorithm. The $\{\times, \mathsf{dec}\}$-circuit constructed is actually structured. Indeed, when processing node t with children $t_1, \dots, t_k$, the $\times$-gates built have k children whose variables are respectively $\mathsf{var}(Q_{\leq t_i}) \setminus \mathsf{var}(R_t)$ and the decision-tree below contains variables in $\mathsf{var}(R_{t_i}) \setminus \mathsf{var}(R_t)$. That is, we can consider the following vtree: we keep the structure of $\mathcal{T}$. On each edge (u, t) of $\mathcal{T}$ where t is the parent of u, we attach leaves labeled by $\mathsf{var}(R_{t_i}) \setminus \mathsf{var}(R_t)$. It gives a vtree and one can check that the circuit produced by the previous algorithm is structured along it (as long as we keep the same order in the decision-trees we build and in the vtree).

The second property that we will investigate in more details in the next section is that we can find a total order $x_1, \dots, x_n$ on the variables of Q such that if g is a decision-gate on x_i, then every decision-gate below g are on variables x_j with $j > i$. This follows from the previous observation: in each decision tree, if we follow the order given by the vtree constructed in the previous paragraph, it gives us a partial order on the variables that we can now extend into a full order.

Example 4.13. We illustrate the previous algorithm on the example given in Fig. 4.4. In this example, we assume we have performed the construction in the subtrees rooted in t_1 and t_2 respectively. Hence, we have constructed gates $v_{t_1}^{100}, v_{t_1}^{121}, v_{t_1}^{200}, v_{t_1}^{221}$ and $v_{t_2}^{114}, v_{t_2}^{113}, v_{t_2}^{213}, v_{t_2}^{302}, v_{t_2}^{401}$ corresponding to every tuple of R_{t_1} and R_{t_2} respectively. The only common variable between R_t and R_{t_1} (denoted by Z_1 in the previous proof) is x_1 which has been grayed in the figure. Hence, the first step of the construction starts by grouping every tuple in R_{t_1} having the same projection on x_1 and building a decision tree for them. In the example, it means that we group $\langle x_1/1, x_3/0, x_4/0\rangle$ and $\langle x_1/1, x_3/2, x_4/1\rangle$ together and hence build a decision tree for the relation $\{\langle x_3/0, x_4/0\rangle, \langle x_3/2, x_4/1\rangle\}$. This

decision tree is depicted on the bottom left corner of Fig. 4.4. Observe that its leaves are connected to $v_{t_1}^{100}$ and $v_{t_1}^{121}$ respectively. The same is done for $\langle x_1/2, x_3/0, x_4/0\rangle$ and $\langle x_1/2, x_3/2, x_4/1\rangle$ but we connect the leaves to $v_{t_1}^{200}$ and $v_{t_1}^{221}$. Observe that we built the same decision-tree twice because the tuples of R_{t_1} where $x_1 = 1$ projected on x_3, x_4 are the same as the tuples of R_{t_1} where $x_1 = 2$ projected on x_3, x_4. Yet, we do not share the subtree because, below, they may have used x_1 in different ways.

We proceed similarly with R_{t_2}. Observe that in this case, despite having 4 distinct values of x_1 in R_{t_2}, we only construct decision-trees for three of them because the case $x_1 = 4$ has no possible extension in R_t. Now, we connect the root of these decision trees using a $\times$-gate if they agree on the value for x_1 and correspond to a tuple of R_t. That is, we connect together the decision tree corresponding to $\sigma_{x_1=1}(R_{t_1})$ and $\sigma_{x_1=1}(R_{t_2})$ because R_t has two tuples where $x_1 = 1$. This $\times$-gate now computes $Q_{\leq t}/\tau$ for every $\tau \in \sigma_{x_1=1}(R_t)$. Observe that we introduce a similar $\times$-gate for v_t^{12}. They could be merged since $Q_{\leq t}/\langle x_1/1, x_2/0\rangle$ and $Q_{\leq t}/\langle x_1/1, x_2/2\rangle$ are the same. While this optimization could be performed, it is not needed to reach the linear size complexity guarantees, so we decided to ignore it.

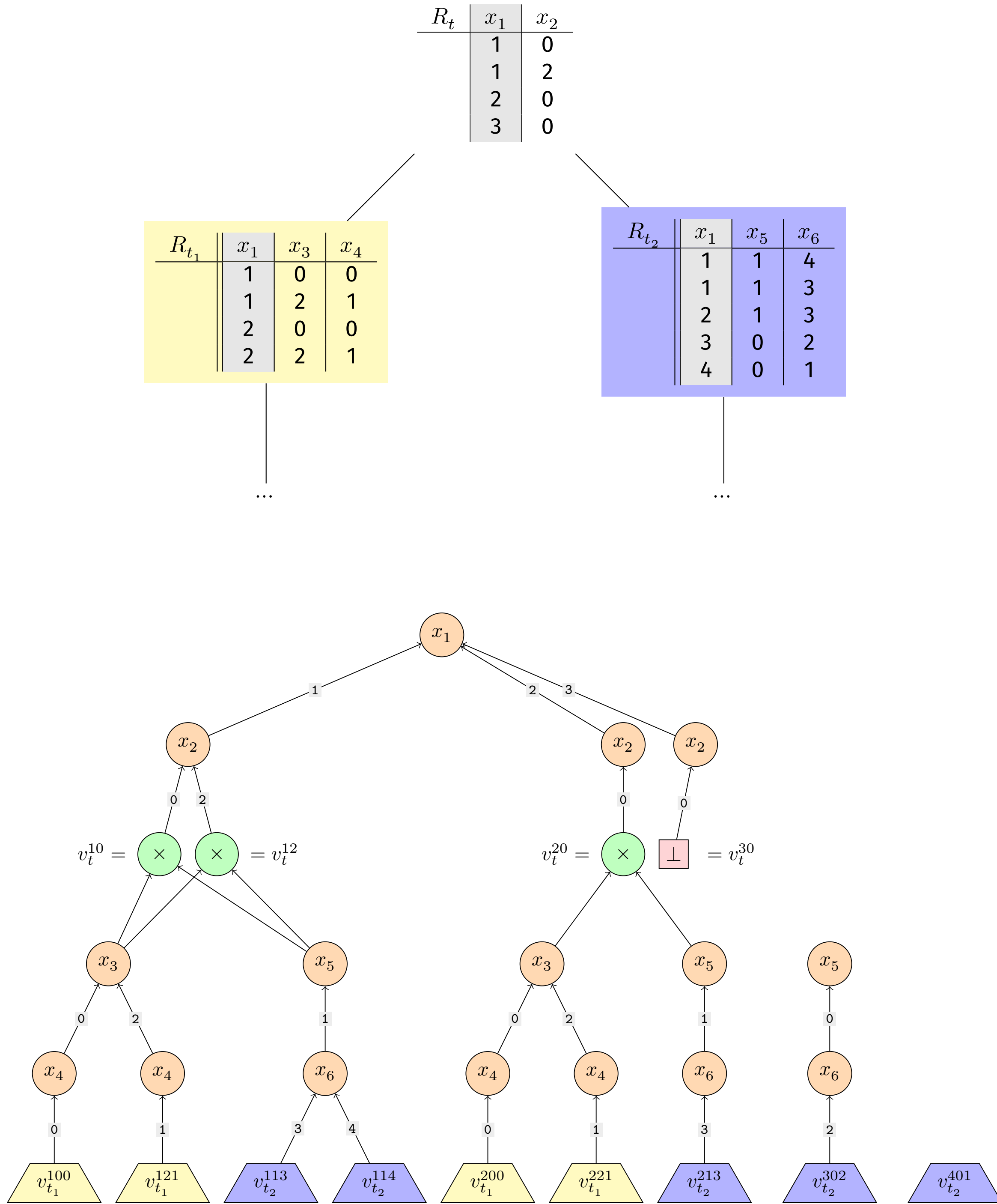
Observe that the case $x_1 = 3$ is interesting because it both has corresponding tuples in R_t and R_{t_2} but not in R_{t_1}. In this case, we know that $[\![Q_{\leq t}]\!]/\langle x_1/3\rangle = \emptyset$ because $R_{t_1}/\langle x_1/3\rangle = \emptyset$. Hence, we set v_t^{30} to be a $\bot$-gate and the root of the join tree corresponding to $\langle x_5/0, x_6/2\rangle$ is dangling in the circuit and will never be used. This is a typical case of the overcomputation performed by many dynamic programming algorithms: we precompute some values that may have relevance locally but not later in the circuit. The same happens to the tuple where $x_1 = 4$. Neither R_t nor R_{t_1} has such tuples, hence, we have a dangling tuple again (observe that in this case, we do not even construct the decision-tree). Observe that we can always post-process the circuit C in $O(|C|) = O(\|Q\|)$ to remove dangling gates and $\bot$-inputs (by propagation).

Finally, if t is also the root of the join tree, then it remains to project away the variables x_1 and x_2 in the circuit done in the last layer of the circuit.

4.3.3 Beyond acyclic queries

One problem with the previously described algorithm is that it only works for acyclic join queries. This is not particularly inherent to compilation algorithms but with join trees in general. They only work for acyclic join queries because otherwise, they do not exist. Fortunately, it is not hard to generalize the notion of join trees to any query. What makes the Yannakakis algorithm efficient on join trees is that there is only a linear number of ways to set the variables in each bag so that they can be extended to a full answer of Q. If we relax this linearity condition, we can straightforwardly generalize the algorithm to any tree decomposition of the hypergraph of Q. This is what motivated the definition of (generalized) hypertree width in the first place in a seminal paper by Gottlob, Leone and Scarcello [GLS99].

Given a hypergraph $H = (V, E)$ and $W \subseteq V$, the *cover number of* W denoted by $\rho_H(W)$ is defined as the minimal number of edges from H one needs to cover W. In other words, ρ_H is the optimal value of the following integer linear program:



R_t	x_1	x_2
	1	0
	1	2
	2	0
	3	0

R_{t_1}	x_1	x_3	x_4
	1	0	0
	1	2	1
	2	0	0
	2	2	1

R_{t_2}	x_1	x_5	x_6
	1	1	4
	1	1	3
	2	1	3
	3	0	2
	4	0	1

Figure 4.4: Local construction of Yannakakis style compilation algorithm. See Example 4.13.

$$\begin{aligned} \min \sum_{e \in E} \lambda_e \\ \forall x \in W, \sum_{e : x \in e} \lambda_e \geq 1 \\ \lambda_e \in \{0,1\} \end{aligned} \tag{4.3}$$

Now given a tree decomposition $\mathcal{T}$ of H, the *generalized hypertree width of* $\mathcal{T}$, denoted by $\mathsf{htw}(H,\mathcal{T})$, is defined as $\max_{t \in \mathcal{T}} \rho_H(B_t)$. That is, if the generalized hypertree width of $\mathcal{T}$ is k, it means that each bag of $\mathcal{T}$ can be covered by the union of at most k edges of H. The *generalized hypertree width of* H[5] is defined as $\mathsf{htw}(H) := \min_{\mathcal{T}} \mathsf{htw}(H,\mathcal{T})$ where $\mathcal{T}$ runs over every tree decomposition of H.

Clearly, an α-acyclic hypergraph H has generalized hypertree width 1 since every bag in the join tree being an edge of H, it has cover-number 1. This is actually an equivalence:

Lemma 4.14. *A hypergraph $H = (V, E)$ is α-acyclic if and only if $\mathsf{htw}(H) = 1$.*

Proof. A join tree has the required property so if H is α-acyclic, it clearly has a tree decomposition as required.

Now let $\mathcal{T}$ be a tree decomposition of H of generalized hypertree width 1. Let t be a node of $\mathcal{T}$ such that $B_t \subsetneq e$ for some $e \in E$. By definition, there is another node u of $\mathcal{T}$ such that $e \subseteq B_u$. Consider the path from t to u in $\mathcal{T}$ and let t' be a node on this path. Clearly, $e \subseteq B_{t'}$, otherwise, it would contradict the connectedness of the tree decomposition. Hence, we can replace the label of B_t with e without changing the fact that $\mathcal{T}$ is a tree decomposition of generalized hypertree width 1. Now we have a tree decomposition where each node is labeled by one edge of H. This is not yet a join tree because some edges may be missing and some edges may be duplicated. We show that we can normalize it into a join tree.

First, assume some f is missing from the tree decomposition. Since $f \subseteq B_t$ for at least some node t, we can add a new node u labeled by f and plug it to t. It does not break connectedness and we now have a bag labeled with f.

Now assume that there are two bags B_u and B_t labeled with e. As before, any node on the path from u to t must contain e. Moreover, the path from u to t contains either the parent of u or the parent of t. Assume wlog that it contains the parent p of u. From what we said, $B_p \supseteq e = B_u$. Hence, we can remove u from the tree, plug all its children directly to p without changing the fact that $\mathcal{T}$ is a tree decomposition of H: it does not affect connectedness nor the fact that each edge is covered. Moreover, we now have one less node labeled by e. By iterating this transformation, we get a join tree for H. □

We now explain how generalized hypertree width can be exploited to build relational circuits. We prove the following result:

[5] Generalized here is with respect to another definition of hypertree width where there is an extra condition on the tree decomposition. We will not directly use it in this chapter and refer the reader to [GLS99] for the exact definition. The notions are related by a constant so they define the same class of hypergraphs.

Theorem 4.15. *Let Q be a query and $\mathcal{T}$ a hypertree decomposition of Q of generalized hypertree width k. We can build a structured $\{\times, \mathsf{dec}\}$-circuit computing $[\![Q]\!]$ of size $O(\|Q\|^k)$ in time* $\text{poly}(|Q|) \cdot \|Q\|^k$.

Proof. We use $\mathcal{T}$ to build a new query Q' such that:

- $\mathcal{T}$ is a join tree of Q',
- $\|Q'\| \leq \|Q\|^k$,
- $[\![Q]\!] = [\![Q']\!]$.

Before explaining how to construct Q', we explain why it is enough: Q' being acyclic, we can apply the algorithm from Section 4.3.1 to build a circuit of size $O(\|Q'\|) = O(\|Q\|^k)$ and computing $[\![Q']\!] = [\![Q]\!]$, hence, we build a circuit computing $[\![Q]\!]$.

Given a node t of $\mathcal{T}$ labeled by W_t, we let Q_t be the join query on variables W_t defined as $Q_t = \{R|_{W_t} \mid R \in Q\}$. Since there exists $R_1, \dots, R_p \in Q$ such that $p \leq k$ and $\bigcup_{i=1}^p \mathsf{var}(R_i) \supseteq W_t$ by definition of generalized hypertree width, we know that $[\![Q_t]\!] \subseteq R_1|_{W_t} \bowtie \dots \bowtie R_p|_{W_t}$. Hence $[\![Q_t]\!]$ contains at most $\prod_{i=1}^k \|R_i\| \leq \|Q\|^k$ tuples. Now, observe that we can materialize $[\![Q_t]\!]$ in time $\text{poly}(|Q|)\|Q\|^k$: we can brute-force every subset of at most k atoms of Q to find a covering of B_t with at most k relations. Then we join them in time $\|Q\|^k$ using a classical join algorithm and filter tuples according to the other relations in Q_t.

Now define $Q' = \{Q_t \mid t \in \mathcal{T}\}$. Clearly, $\mathcal{T}$ is a join tree for Q'. Moreover, we can materialize each Q_t in time $\text{poly}(|Q|) \cdot \|Q\|^k$ and hence compute Q' in the same amount of time. Finally, by definition, the size of Q' is at most $|\mathcal{T}| \cdot \text{poly}(|Q|) \cdot \|Q\|^k = \text{poly}(|Q|) \cdot \|Q\|^k$.

It remains to show $[\![Q]\!] = [\![Q']\!]$. By definition, every tuple in $[\![Q]\!]$ projected over W_t is in $[\![Q_t]\!]$. Hence, $[\![Q]\!] \subseteq [\![Q']\!]$. For the other way around, recall that for every relation R of Q, there is a node t of $\mathcal{T}$ such that $\mathsf{var}(R) \subseteq W_t$. In this case, $[\![Q_t]\!]|_{\mathsf{var}(R_t)} \subseteq R_t$. Hence $[\![Q']\!] \subseteq [\![Q]\!]$. □

An easy corollary of Theorem 4.15 is that for any join query Q, there is a structured $\{\times, \mathsf{dec}\}$-circuit computing $[\![Q]\!]$ of size $O(\|Q\|^{\mathsf{htw}(\mathcal{H}(Q))})$. Unfortunately, if one wants to construct this circuit, an optimal tree decomposition of $\mathcal{H}(Q)$ is needed, and we know that even deciding whether $\mathsf{htw}(Q) \leq 2$ is NP-hard [GMS09; FGP18]. Hence, we cannot hope for polynomial combined complexity for this problem but can brute-force over tree decompositions $\mathcal{H}(Q)$ and have a $O(\|Q\|^{\mathsf{htw}(\mathcal{H}(Q))})$ algorithm in data complexity. Another possible approach is to use the fact that one can compute tree decompositions of width at most $3\mathsf{htw}(\mathcal{H}(Q))$ in time $|Q|^{O(\mathsf{htw}(\mathcal{H}(Q)))}$ [AGG07] enabling a polynomial time yet non-optimal compilation algorithm in combined complexity if $\mathsf{htw}(\mathcal{H}(Q))$ is considered constant. In a nutshell:

Theorem 4.16. *Let Q be a query. We can build a structured $\{\times, \mathsf{dec}\}$-circuit computing $[\![Q]\!]$ of size $O(\|Q\|^{3k})$ in time* $\text{poly}(|Q|) \cdot \|Q\|^{O(k)}$ *where $k = \mathsf{htw}(\mathcal{H}(Q))$.*

The technique in Theorem 4.15 relies on two ingredients: first, we need to bound the size of $[\![Q_t]\!]$ in each bag and then, we need to materialize Q_t in time $O(|[\![Q_t]\!]|)$. It turns out that the upper bound given by the cover number is not optimal. One minimal and popular example is the triangle query defined as $Q_\Delta = R(x, y), S(y, z), T(z, x)$ for relations R, S, T of size at

most N each. The cover number of $\{x, y, z\}$ in $H(Q_\Delta)$ is two: one atom is not enough to cover every variable while any pair of atoms would do. Hence, the upper bound on $|[\![Q_\Delta]\!]|$ given by the cover number is N^2, but it is known that $|[\![Q_\Delta]\!]| \leq N^{1.5}$ [GM14; AGM13]. The proof of this fact is based on a relaxation of the cover number to so called *fractional cover number.* Indeed, one can consider the definition of $\rho_H(W)$ from (4.3) and relax the integer constraint:

$$\begin{aligned} \min \sum_{e \in E} \lambda_e \\ \forall x \in W, \sum_{e : x \in e} \lambda_e \geq 1 \\ \lambda_e \in [0,1] \end{aligned} \tag{4.4}$$

The optimal value of (4.4) is called the *fractional cover number of* W, which we denote by $\rho^*_H(W)$. Intuitively, we aim at covering variables from W but are now allowed to use the edges in a fractional way, as long as each variable of W is covered by edges summing to at least 1.

Example 4.17. Consider the triangle query Q_Δ again and its hypergraph $\mathcal{H}(Q_\Delta)$ and let $W = \{x, y, z\}$. 4.4 rewrites as:

$$\begin{aligned} \min\ & \lambda_{\{x,y\}} + \lambda_{\{y,z\}} + \lambda_{\{x,z\}} \\ \text{s.t. } & \lambda_{\{x,y\}} + \lambda_{\{x,z\}} \leq 1 \\ & \lambda_{\{x,y\}} + \lambda_{\{y,z\}} \leq 1 \\ & \lambda_{\{x,z\}} + \lambda_{\{y,z\}} \leq 1 \\ & \lambda_{\{x,y\}} \leq 1, \lambda_{\{y,z\}} \leq 1, \lambda_{\{x,z\}} \leq 1. \end{aligned}$$

The solution $\lambda_{\{x,y\}} = \lambda_{\{y,z\}} = \lambda_{\{x,z\}} = 1/2$ is optimal since every constraint is satisfied at equality, proving that $\rho^*_{\mathcal{H}(Q_\Delta)}(\{x, y, z\}) = 1.5$.

The main result about fractional cover number we need now is the following bound, known as the AGM bound from the names of Atserias, Grohe and Marx [AGM13] who proved it:

Theorem 4.18. *Let Q be a join query over variables X, let $N = \max_{R \in Q} |R|$ and $\rho^* = \rho^*_{\mathcal{H}(Q)}(X)$. We have*

$$|[\![Q]\!]| \leq N^{\rho^*}.$$

Actually, [AGM13] also proves the optimality of Theorem 4.18 in the sense that for every N, one can build a join query Q^* such that every relation in Q^* has size at most N, $\mathcal{H}(Q) = \mathcal{H}(Q^*)$ and $|[\![Q^*]\!]| \geq \frac{N^{\rho^*}}{\text{poly}(|Q|)}$.

Now this result suggests the following generalization of generalized hypertree width: given a hypergraph H and a tree decomposition $\mathcal{T}$ of H, the *fractional hypertree width of* $\mathcal{T}$, denoted by $\mathsf{fhtw}(H, \mathcal{T})$, is defined as $\max_{t \in \mathcal{T}} \rho^*_H(B_t)$ and the *fractional hypertree width of* H, denoted by $\mathsf{fhtw}(H)$, as $\min_{\mathcal{T}} \mathsf{fhtw}(H, \mathcal{T})$.

Now, given a tree decomposition $\mathcal{T}$ of H of fractional hypertree width k, we can define for every node t of $\mathcal{T}$ the query Q_t and Q' as in the proof of Theorem 4.15. From Theorem 4.18, $|[\![Q_t]\!]| \leq \|Q\|^k$ and, following the same reasoning as in Theorem 4.15, $|[\![Q']\!]| \leq \|Q\|^k$. This suggests the following:

Theorem 4.19. *Let Q be a query and $\mathcal{T}$ a hypertree decomposition of Q of fractional hypertree width k. We can build a structured $\{\times, \mathsf{dec}\}$-circuit computing $[\![Q]\!]$ of size $O(\|Q\|^k)$ in time* $\mathrm{poly}(|Q|) \cdot \|Q\|^k$.

While the proof of Theorem 4.19 outlined above is enough to prove the existence of a circuit of size $\mathrm{poly}(|Q|) \cdot \|Q\|^k$ computing $[\![Q]\!]$, there is one bit missing. Indeed, for the proof to work, we need to materialize Q_t in each bag. With the cover number, we use a cover of k relations to do that. It is not as clear how one should proceed in the case of fractional hypertree width. Again, consider the triangle query Q_Δ. If one tries to compute $[\![Q_\Delta]\!]$ by iteratively joining its relations (as would most database management systems proceed), then the first join between say $R(x, y)$ and $S(y, z)$ may contain N^2 tuples, defeating the advantage of fractional cover number already.

Fortunately, there is a way to compute the answers of a join query Q over variables X in time $\mathrm{poly}(|Q|) \cdot \|Q\|^k$ where $k = \rho^*_{\mathcal{H}(Q)}(X)$. There is a rich literature around this problem: given a query Q such that we have an upper bound U on its number of answers, when can we materialize the answers in time $\mathrm{poly}(|Q|)U$, that is, linear in U in the data complexity model. These kind of algorithms have been studied under the name "worst-case optimal", in the sense that, if $|[\![Q]\!]|$ reaches the upper bound U (hence the worst case), then the algorithm is optimal because it needs at least time $O(U)$ to output every answer of Q.

One of the first worst-case optimal join algorithm which matches the upper bound given by the fractional cover number is due to Ngo, Porat, Ré and Rudra in 2012 [Ngo+12; Ngo+18] where they prove:

Theorem 4.20. *Given a join query Q over variables X, we let $N = \max_{R \in Q} |R|$ and $k = \rho^*_{\mathcal{H}(Q)}(X)$. We can compute $[\![Q]\!]$ in time* $\mathrm{poly}(|Q|)\|Q\|^k$.

Theorem 4.20 is enough to complete the proof of Theorem 4.19 sketched above. Without entering the details, we give a quick overview of the literature related to Theorem 4.20. The approach of [Ngo+12] is based on joining the relations of Q by joining partitions of their tuples that are guaranteed to have a small enough number of answers and recombine them to recover $[\![Q]\!]$. There were conceptually simpler algorithms later, for example LeapFrog Trie Join [Vel14] and Generic Join [Ngo18]. These algorithms are based on a branching algorithm where the answers of Q are enumerated by extending a candidate one variable at a time. With Oliver Irwin and Sylvain Salvati, we propose an even simpler version of this branching algorithm, with a simple analysis in [CIS25]. Much interesting work has gone beyond the worst case offered by fractional cover number by considering queries with constraints on the database which make the worst case smaller [Got+12; KNS17; Kha+24; KNS25]. While they could be used in our setting, we will not cover this literature in detail in this document and refer the reader to the excellent survey of Dan Suciu for more details on the main results and proof techniques [Suc23].

As before, computing a tree decomposition for $\mathcal{H}(Q)$ with optimal fractional hypertree width is hard since deciding whether a hypergraph has fractional hypertree width smaller than 2 is NP-hard [FGP18]. We hence cannot hope for polynomial combined complexity to construct a circuit of size $O(\|Q\|^{\mathsf{fhtw}(\mathcal{H}(Q))})$. As before, though in a less satisfactory way, we can approximate fractional hypertree width: given a hypergraph H, we can construct a tree decomposition of H of fractional hypertree width at most $O(k^3)$ where $k = \mathsf{fhtw}(H)$ in time $O(|H|^{O(k^3)})$ [Mar10, Theorem 4.1]. In other words:

Theorem 4.21. *Let Q be a query of fractional hypertree width k. We can build a structured $\{\times, \mathsf{dec}\}$-circuit computing $[\![Q]\!]$ of size $O(\|Q\|^{O(k^3)})$ in time* $\mathrm{poly}(|Q|) \cdot \|Q\|^{O(k^3)}$.

4.3.3.1 Conjunctive queries

We conclude this section about Yannakakis algorithm to show how to handle conjunctive queries. Compiling conjunctive queries is significantly harder than compiling join queries. This can be seen via the complexity of computing $|[\![Q]\!]|$ since building a $\{\times, \mathsf{dec}\}$-circuit C for Q allows to compute $|[\![Q]\!]|$ in time $O(|C|)$: we know from [PS13] that there exists a conjunctive query $Q(Z)$ such that $\mathcal{H}(Q)$ is α-acyclic but such that computing $|[\![Q(Z)]\!]|$ cannot be done in linear time under reasonable complexity assumptions, even if Z contains every variable of Q but one. It is in stark contrast with what Theorem 4.12 implies for acyclic **join** queries. The intuition behind this discrepancy comes from the fact that one can encode many things into projected variables. For example, the star example from [DM15], of the form, $Q_k(x_1, \dots, x_k) = R_1(x_1, y), \dots, R_k(x_k, y)$ is used to encode the number (up to a constant) of sets of vertices $x_1, \dots, x_k$ in a graph G that are not k-cliques in G. Hence, counting them is unlikely to be tractable.

Observe that any tree decomposition for Q_k has its free variables interlaced with the projected y variables. This can actually be seen as the source of hardness, which has been formalized under the name of *star-size* by Durand and Mengel [DM15]. Similarly, in [BDG07], Bagan, Durand and Grandjean characterize the conjunctive queries whose answers can be enumerated with constant delay after linear time preprocessing (in data complexity). It turns out that the main condition for tractability, in addition to the underlying join query to be acyclic, is that the free variables have to be somewhat separated from the other variables in the tree decomposition, a property known as free-connex.

To understand what motivates the definition of free-connexity, we consider the following scenario: let Q be an acyclic join query over variables X and $Z \subseteq X$. Assume you have a tree decomposition $\mathcal{T}$ of $\mathcal{H}(Q)$ and a node t such that every variable in Z appears below t and only those variables. Then, when we deal with node t in Theorem 4.12, the circuit rooted in v_t^τ for every $\tau \in R_t$ contains only variables in Z. If we want to compute $[\![Q(X \setminus Z)]\!]$ instead of $[\![Q]\!]$, observe that at this point in the algorithm, we do not really care at the relation computed at v_t^τ but only whether it is empty or not. Hence we can adapt the algorithm to compute $[\![Q(X \setminus Z)]\!]$ instead.

We formalize the notion below: let $H = (V, E)$ be a hypergraph and $\mathcal{T}$ be a tree decomposition of H. Given $S \subseteq V$, we say that $\mathcal{T}$ is *ext-S-connex*[6] if there exists a connected subset

[6]The notion of S-connexity of tree decompositions has originally been defined in [BDG07] for join trees.

A of nodes of $\mathcal{T}$ such that $\bigcup_{t\in A} B_t = S$.

Now, if $Q(S)$ is a conjunctive query and $\mathcal{T}$ is an ext-S-connex tree decomposition of $\mathcal{H}(Q)$ of fractional hypertree width k, then we can follow a similar approach as in Theorem 4.15: we build a new join query Q' over variables S such that Q' is acyclic and $[\![Q']\!] = [\![Q(S)]\!]$ and $\|Q'\| \leq \|Q\|^k$ and use Theorem 4.12 to build a $\{\times, \mathsf{dec}\}$-circuit computing $[\![Q']\!] = [\![Q(S)]\!]$ and has $\|Q\|^k$, hence generalizing Theorem 4.19 to the case of conjunctive queries.

The connection between ext-S-connexity and conjunctive queries is the following result:

Lemma 4.22. *Let $Q(S)$ be a conjunctive query and $\mathcal{T}$ be an ext-S-connex tree decomposition of $\mathcal{H}(Q)$ of fractional hypertree width k. We can construct a join query Q' over variables S in time* $\mathrm{poly}(|Q|) \cdot \|Q\|^k$ *such that:*

- *Q' is acyclic,*
- *$[\![Q']\!] = [\![Q(S)]\!]$ and*
- *$\|Q'\| \leq \|Q\|^k$.*

Proof. We start by materializing each bag as in Theorem 4.15 to construct a join query Q'' of size at most $\|Q\|^k$ such that $\mathcal{T}$ is a join tree of Q'' and $[\![Q'']\!] = [\![Q]\!]$.

Let A be the subset of nodes of $\mathcal{T}$ such that $\bigcup_{t\in A} B_t = S$. We root $\mathcal{T}$ in some node t of A and let $\bar{A}$ be the set of nodes u such that u is not in A but the parent t of u is in A.

First, observe that $Q'' = \{R_t \mid t \in A\} \cup \bigcup_{u\in\bar{A}} Q''_{\leq u}$ where R_t is the atom of Q'' labeling node t. We let $Q_0 = \{R_t \mid t \in A\}$. Observe that by definition, $\mathsf{var}(Q_0) = S$. Let $u, u' \in \bar{A}$ be distinct nodes and t the parent of u, then $\mathsf{var}(Q''_{\leq u}) \cap \mathsf{var}(Q''_{\leq u'}) \subseteq \mathsf{var}(R_t) \subseteq S$. It implies that

$$[\![Q''(S)]\!] = [\![Q_0]\!] \bowtie_{u\in\bar{A}} [\![Q''_{\leq u}(S)]\!].$$

Indeed, if $\tau \in [\![Q''(S)]\!]$, then there is σ over $X \setminus S$ such that $\tau \times \sigma$ is an answer of Q''. In particular, it is an answer of every atom of Q'', hence of Q_0 and $Q_{\leq u}$ for every $u \in \bar{A}$. Conversely, we let τ be an answer of $[\![Q_0]\!] \bowtie_{u\in\bar{A}} [\![Q''_{\leq u}(S)]\!]$. By definition, for every $u \in \bar{A}$, we have σ_u such that $(\tau \times \sigma_u)|_{\mathsf{var}(Q''_{\leq u})}$ is an answer of $Q''_{\leq u}(S)$. From what precedes, for distinct $u, u' \in \bar{A}$, σ_u and $\sigma_{u'}$ do not share variables, hence we can define $\sigma = \bigtimes_{u\in\bar{A}} \sigma_u$ and verify that $\tau \times \sigma$ is an answer of Q'', hence τ is an answer of $Q''(S)$.

For $u \in \bar{A}$, we let $R'_u = [\![Q''_{\leq u}(S)]\!]$ and we let $Q' = \{R_t \mid t \in A\} \cup \{R'_u \mid u \in \bar{A}\}$. From what precedes, $[\![Q']\!] = [\![Q''(S)]\!] = [\![Q(S)]\!]$. Moreover, if we relabel node u of $\mathcal{T}$ by R'_u and remove every node of $\mathcal{T}$ that is not in $A \cup \bar{A}$, we have a join tree for Q'.

It remains to show that $\|Q'\| \leq \|Q\|^k$ and to show that it can be materialized. There is nothing to do for R_t for $t \in A$ since R_t has been materialized already when constructing Q'' and we know that this relation has size at most $\|Q\|^k$.

For $u \in \bar{A}$ with parent $t \in A$, we have $\mathsf{var}(Q''_{\leq u}) \cap S \subseteq \mathsf{var}(Q''_{\leq u}) \cap \mathsf{var}(R_t) \subseteq \mathsf{var}(R_u)$ by connectedness of $\mathcal{T}$. Hence $[\![Q''_{\leq u}(S)]\!] \subseteq R_u|_S$. The construction from Theorem 4.12 at node u

In this case, one is not allowed to use bags that are not atoms. This prevents imposing strict conditions on the bags and the notion of Z-connexity works around this rigidity. The same paper relaxes the notion to tree decomposition under the name ext-S-connexity which is the only one really needed and which is equivalent. We keep the original name.

gives us exactly the set of tuples in $R_u|_S$ that can be extended below as an answer of $[\![Q''_{\leq u}(S)]\!]$. We hence only have to filter $R_u|_S$ to keep the relevant tuples. It construct R'_u, of size at most $|R_u| \leq \|Q\|^k$ in time $\text{poly}(|Q|) \cdot \|Q\|^k$, which concludes the proof. □

Now we can apply Theorem 4.12 to Q' from Lemma 4.22 to construct a circuit computing $[\![Q(S)]\!]$ and have:

Theorem 4.23. *Let $Q(S)$ be a conjunctive query and $\mathcal{T}$ be an ext-S-connex tree decomposition of $\mathcal{H}(Q)$ of fractional hypertreewidth k. We can build a structured $\{\times, \mathsf{dec}\}$-circuit computing $[\![Q(S)]\!]$ of size $O(\|Q\|^k)$ in time $\text{poly}(|Q|) \cdot \|Q\|^k$.*

We conclude this section with an observation. In the proof of Lemma 4.22, the fractional hypertree width of $\mathcal{T}$ is used differently for the nodes corresponding to A and for the nodes in $\bar{A}$. In the former case, they are used as a way of building the final circuits. In the latter, only to materialize $[\![Q''_{\leq u}(S)]\!]$. But in this case, we could use any other join algorithm. While in general, we do not know better algorithms than the ones exploiting fractional hypertree width, we can sometimes use external knowledge on the structure of Q to speed up computation, such that functional dependencies or degree constraints or a submodular width [Mar13; KNS25; KNS16] (a generalization of fractional hypertree width that does not allow counting but efficient model checking), which in some cases, may speed up the computation of the circuit.

Finally, we conclude this section by remarking that we have solely focused on upper bounds in this section. It is not clear from what we presented here that fractional hypertree width is an optimal measure to build $\{\times, \mathsf{dec}\}$-circuit, but these kinds of questions have been studied in the literature. The focus of this document being mainly on algorithmic applications of knowledge compilation techniques, we will not cover the details of this literature but give a few pointers for theinterested readers. Olteanu and Závodný have already studied this direction in their seminal paper [OZ15], showing that, if we fix the vtree T the circuit must respect, then, essentially, we can build a database for which the smallest circuit respecting T will have size $\|Q\|^k$ where k is the fractional hypertree width of some tree decomposition induced by the vtree. It does not mean that we cannot, however, find a better vtree for this particular database nor that smaller unstructured circuits cannot be built for it. Vinall-Smeeth and Berkholz give such lower bounds in [BV23; BV25] for $\{\times, \cup\}$-circuit based on the more general notion of submodular width.

4.4 Building Relational Circuits Top-down

The previous section has mainly focused on how to build circuits using a dynamic programming algorithm which iteratively joins relations following a join tree from its leaves to its root, hence the name "bottom-up", since the circuit is also built from its inputs to its output. But many knowledge compilers for CNF formulas such as D4 build the circuit in a top-down fashion, from its output to its inputs. One simple illustration of this is when building a decision tree representing a CNF formula F. One can start by picking one variable x, add a decision-gate on x and branch the 0-edge to a recursively computed decision tree for $F[x/0]$ and the 1-edge to a decision tree for $F[x/1]$. While this idea is simple, it has been refined in [San+04; HD05]

under the name #DPLL or exhaustive DPLL to the point of being practical. While these kinds of branching approaches are well known in knowledge compilation and in constraint programming in general, they have been less explicitly exploited in databases.

In this section, we revisit exhaustive DPLL in the context of databases and show how it can be seen as a way of performing the Yannakakis algorithm upside down. We also show that, contrary to algorithms based on tree decompositions, we can use this algorithm to build $\{\times, \mathsf{dec}\}$-circuits for conjunctive queries having negated atoms, which is not possible using only tree decompositions, at least, not in a way that is polynomial in the combined complexity. This section revisits some results from [CI24].

4.4.1 Manipulating Join Queries

Before explaining the algorithm, we need to introduce a few notations and notions that will be needed later. Let $Q = R_1, \dots, R_m$ be a join query and τ a tuple over variables Y. If there is a relation $R \in Q$ such that $R/\tau = \emptyset$, we say that τ is *inconsistent with* Q. In this case, we clearly have $[\![Q/\tau]\!] = \emptyset$.

In this section, we will need to represent $R(y_1, \dots, y_k)/\tau$ for an atom $R(y_1, \dots, y_k)$ of a join query. We will often do it explicitly by writing $R(\tau^*(y_1), \dots, \tau^*(y_k))$ where

$$\tau^*(y) = \begin{cases} \tau(y) \text{ if } \tau \text{ is defined on } y, \\ y \text{ otherwise.} \end{cases}$$

For example, $R(x_1, x_2, x_3)/\langle x_1/0, x_2/2\rangle$ will be denoted by $R(0, 2, x_3)$.

Representing join queries. Our algorithm needs to efficiently get representations of Q/τ and to this end we need a good representation of Q. We assume that we have an order $\{x_1, \dots, x_n\}$ on variables X that we will always be using in the algorithm. Moreover, we assume that the relations of Q are ordered as $R_1(X_1), \dots, R_m(X_m)$. We represent each relation $R_i(X_i) \in Q$ as a *trie*. A node of the trie is labeled by $x \in X$ and a hash table mapping domain elements $d \in D$ to another trie node. If $X_i = \{x_{i_1}, \dots, x_{i_k}\}$ with $i_1 < \dots < i_k$, we store $R_i(X_i)$ as a trie whose first node is labeled by x_{i_1} and a hash table h. Then, for each value d of x_{i_1} appearing in R_i, $h[d]$ contains a pointer to a node representing a trie for $R_i/\langle x_{i_1}/d\rangle$. We represent Q as a table T_Q of length m where $T_Q[i]$ is a trie representing R_i.

It is easy to see that if each relation of Q is originally given in the input as a list of tuples, then we can build this trie representation of Q in time $O(\|Q\|)$ by sorting its tuples in the lexicographical order over D^{X_i} induced by $x_1, \dots, x_n$ (recall that sorting can be performed in linear time in the RAM model of computation we are considering).

Now observe that if we want a representation of $Q/\langle x_1/d\rangle$, we simply have to scan T_Q and for each relation $R_i(X_i) \in Q$ such that $x_1 \in X_i$, we move to the next node in the trie representation. This can be done in expected time $O(|Q|)$ (expected time comes from the use of the hash table), but we could also trade it for $O(|Q| \log |Q|)$ by storing the hash table as a sorted array and performing binary search.

More generally, if τ is a tuple over variables $x_1, \dots, x_p$ with $p < n$, then we can get a representation of Q/τ in time $O(p|Q|) = \text{poly}(|Q|)$. It is important to observe here that it

does not depend on the data size of Q. We can also detect in $O(\text{poly}(|Q|))$ whether there is an $R \in Q$ such that $R/\tau = \emptyset$, that is, Q is inconsistent with τ. Indeed, while navigating the trie of R, if $R/\tau = \emptyset$, we will hit a missing value in the hash table associated with it.

Subqueries and caching. Given two subqueries Q_1, Q_2 of Q and two tuples τ_1, τ_2 , we say that (Q_1, τ_1) and (Q_2, τ_2) are *syntactically equivalent* if:

1. Both Q_1 and Q_2 are consistent with τ_1 and τ_2.
2. Let Q_1' be the set of atoms R of Q_1 such that $\mathsf{var}(R) \subseteq \mathsf{var}(\tau_1)$, and define Q_2' similarly. Then $Q_1^* := Q_1 \setminus Q_1'$ and $Q_2^* := Q_2 \setminus Q_2'$ have the same atoms and for every $R \in Q_1^* = Q_2^*$, we have $\tau_1|_{\mathsf{var}(R)} = \tau_2|_{\mathsf{var}(R)}$.

In other words, (Q_1, τ_1) and (Q_2, τ_2) are syntactically equivalent if, after removing every atom from Q_1 already satisfied by τ_1, and every atom from Q_2 already satisfied by τ_2, Q_1 and Q_2 have the same atoms, and for each such atom R, the projection of τ_1 over $\mathsf{var}(R)$ is the same as the projection of τ_2 over $\mathsf{var}(R)$. Obviously, it implies $R/\tau_1 = R/\tau_2$ and hence, $[\![Q_1/\tau_1]\!] = [\![Q_2/\tau_2]\!]$.

Now let (Q_1, τ_1) be a pair such that $\mathsf{var}(\tau_1) \cap \mathsf{var}(Q_1) = \{x_1, \ldots, x_i\} \cap \mathsf{var}(Q_1)$ for some i, that is, every variable of Q_1 is assigned up to some index i. The canonical representation of (Q_1, τ_1) is an array T of size m where $T[i]$ contains a pointer to the node of the trie representing R/τ_1 if R is an atom of Q_1^* and $T[i]$ is a 0-pointer otherwise. Observe that such representation can be stored in $O(|Q|)$ registers and hence hashed efficiently.

We have the following:

Lemma 4.24. *(Q_1, τ_1) and (Q_2, τ_2) are syntactically equivalent if and only if they have the same canonical representation. Moreover, we can check syntactic equivalence in $O(m)$ and maintain a hash table indexed by canonical representation in $O(1)$ expected time.*

Proof. By definition, the atoms in the canonical representation of (Q_1, τ_1) and (Q_2, τ_2) are exactly Q_1^* and Q_2^*. Moreover, there is a one-to-one correspondence between the nodes of the trie representing R and tuples $\tau_1|_{\mathsf{var}(R)}$, hence the equality of representations. We can hence check syntactic equivalence of (Q_1, τ_1) and (Q_2, τ_2) in time $O(m)$ by checking each entry of the table. Moreover, the size of the canonical representation being polynomially bounded in the size of the input, we can hash them (for example, by hashing the concatenation of the values stored in T) and maintain a hash table indexed by these hash values in $O(1)$ expected time. □

Example 4.25. Let $Q = R_1(x_1, x_2), R_2(x_2, x_3), R_3(x_2, x_4)$ with

- $R_1 = \{\langle x_1/0, x_2/0\rangle, \langle x_1/1, x_2/1\rangle, \langle x_1/0, x_2/1\rangle\}$,
- $R_2 = \{0, 1\}^{\{x_2, x_3\}}$,
- $R_3 = \{0, 1\}^{\{x_2, x_4\}}$.

Let $\tau_1 = \langle x_1/0, x_2/0\rangle$, $\tau_2 = \langle x_1/0, x_2/1\rangle$ and $\tau_3 = \langle x_1/1, x_2/1\rangle$. $Q/\tau_1 = R_2/\tau_1, R_3/\tau_1 = R_2(0, x_3), R_3(0, x_4)$. Since τ_1 assigns every variable of R_1 in a consistent manner, R_1 does not appear in Q/τ_1. It is represented in memory as an array T_1 of size 3 such that $T_1[1]$ is a 0-pointer (because R_1 is not in Q/τ_1), $T_1[2]$ contains a pointer to the node of the trie representing $R_2/\langle x_2/0\rangle$ and $T_1[3]$ contains a pointer to a node of the trie representing $R_3/\langle x_2/0\rangle$.

Now $Q/\tau_2 = R_2(1, x_3), R_3(1, x_4)$. One can observe that Q/τ_1 and Q/τ_2 have the same answer set, but (Q, τ_1) and (Q, τ_2) are not syntactically equivalent because they differ on the value of x_2. In contrast, $Q/\tau_3 = R_2(1, x_3), R_3(1, x_4) = Q/\tau_2$, hence (Q, τ_2) and (Q, τ_3) are syntactically equivalent. It can be checked that they have the same representation in memory, allowing efficient hashing.

Connected components. Observe that if the atoms of a join query Q can be partitionned into $Q_1 \cup Q_2$ such that $\mathsf{var}(Q_1) \cap \mathsf{var}(Q_2) = \emptyset$, then we have $[\![Q]\!] = [\![Q_1]\!] \times [\![Q_2]\!]$. A generalization of this fact can be written as follows: if $Q = Q_1 \cup Q_2$ and τ is a tuple over variables Y such that $\mathsf{var}(Q_1) \cap \mathsf{var}(Q_2) \subseteq Y$. In this case, we have $[\![Q]\!]/\tau = [\![Q_1]\!]/\tau \times [\![Q_2]\!]/\tau$, as shown in Lemma 4.1. Moreover, if τ is consistent with Q, then it is consistent with both Q_1 and Q_2. It motivates the following definition: let Q be a join query and τ a tuple over variables Y that is consistent with Q. Let G_Q^τ be the graph whose vertices are the set of atoms R of Q such that $\mathsf{var}(R) \setminus Y \neq \emptyset$. In G_Q^τ, there is an edge between two atoms $R, S \in Q$ if and only if $(\mathsf{var}(R) \cap \mathsf{var}(S)) \setminus Y \neq \emptyset$. We denote by $\mathsf{cc}(Q, \tau) = \{C_1, \dots, C_k\}$ the set of connected components of G_Q^τ. Each connected component corresponds to a subset of atoms of Q and $\mathsf{var}(C_i) \cap \mathsf{var}(C_j) \subseteq Y$. Hence, we have:

Lemma 4.26. *For every τ consistent with Q, $[\![Q]\!]/\tau = \bigtimes_{C \in \mathsf{cc}(Q,\tau)}([\![C]\!]/\tau)$. Moreover, we can compute $\mathsf{cc}(Q, \tau)$ in time polynomial in $|Q|$.*

Observe that Lemma 4.26 is only true if τ is consistent with Q, because we have removed from G_Q^τ every atom $R \in Q$ such that $\mathsf{var}(R) \subseteq Y$. Hence, if Q is not consistent with τ, we have $[\![Q]\!]/\tau = \emptyset$ but the right-hand side relation in Lemma 4.26 may be nonempty.

Example 4.27. We let

$$Q = R_1(x_1, x_2), R_2(x_2, x_3), R_3(x_1, x_4), R_4(x_3, x_5), R_5(x_4, x_6)$$

and τ be a tuple over $\{x_1, x_2\}$ consistent with Q. Then G_Q^τ is the graph with vertices R_2, R_3, R_4, R_5 and edge $\{R_2, R_4\}$ because $x_3 \in (\mathsf{var}(R_2) \cap \mathsf{var}(R_4)) \setminus \{x_1, x_2\}$ and edge $\{R_3, R_5\}$. Then $\mathsf{cc}(Q, \tau) = \{\{R_2, R_4\}, \{R_3, R_5\}\}$ and we indeed have $Q/\tau = Q_1/\tau \times Q_2/\tau$ with $Q_1 = \{R_2, R_4\}$, and $Q_2 = \{R_3, R_5\}$.

4.4.2 Exhaustive DPLL

We are now ready to describe exhaustive DPLL. We assume that we are starting from a join query $Q = R_1, \dots, R_m$, with variables X ordered as $\pi = (x_1, \dots, x_n)$, represented as

explained in the previous section (after a linear preprocessing in $\|Q\|$). The algorithm computes an ordered $\{\times, \mathsf{dec}\}$-circuit computing $[\![Q]\!]$ recursively. Each recursive call of the algorithm has two inputs: Q', a subquery of Q, and τ, a tuple over variables $\{x_1, \dots, x_p\} \cap \mathsf{var}(Q')$ for some $p \leq n$, and it returns a gate $v(Q', \tau)$ in a $\{\times, \mathsf{dec}\}$-circuit that computes $[\![Q']\!]/\tau$. Before returning, the algorithm updates a hash table called the *cache* mapping the canonical representation of (Q', τ) to a gate v computing $[\![Q']\!]/\tau$. The goal of the cache is to avoid redoing the same computation twice: if at some point the algorithm is called with parameters (Q'', τ') and the canonical representation of (Q'', τ') is already in the cache and mapped to a gate v, then it immediately returns v which computes $[\![Q'']\!]/\tau'$ by construction.

We describe the algorithm $\mathsf{DPLL}(Q, \tau, \pi)$ and prove its correctness at the same times below. On input (Q', τ), $\mathsf{DPLL}(Q', \tau, \pi)$ does the following:

1. If Q' is inconsistent with τ, then return a $\bot$-gate. It is correct since $[\![Q']\!]/\tau = \emptyset$.

2. If τ assigns every variable of Q', then return a $\top$-gate. It is correct since $[\![Q']\!]/\tau = \{\langle\rangle\}$.

3. If the canonical representation of (Q', τ) is in the cache and maps to gate v, then it returns v. It is correct since by induction, the cache is correct and hence v computes Q'/τ.

4. Otherwise, let $\mathsf{cc}(Q', \tau) = \{C_1, \dots, C_k\}$.

   (a) If $k > 1$: we create $\times$-gate v and for $i = 1, \dots, k$, we let $v_i \leftarrow \mathsf{DPLL}(C_i, \tau, \pi)$ to recursively construct a gate v_i computing $[\![C_i]\!]/\tau$. We add v_i as an input of v. Now v computes $\bigtimes_{i=1}^{k} [\![C_i]\!]/\tau = [\![Q']\!]/\tau$ by Lemma 4.26 hence it is correct. The algorithm updates the cache to map the canonical representation of (Q', τ) with v and returns v.

   (b) If $k = 1$: let x_i be the smallest variable in $\mathsf{var}(Q')$ that is not set by τ. We add a new decision-gate v on variable x_i. For every $d \in \mathsf{dom}(Q)$, we let $v_d \leftarrow \mathsf{DPLL}(Q', \tau_d, \pi)$ where $\tau_d = \tau \times \langle x_i/d \rangle$ to recursively construct a gate v_d computing $[\![Q']\!]/\tau_d$ and add an edge labeled by d between v and v_d. By construction, v computes $\bigcup_{d \in \mathsf{dom}(Q)} [\![Q']\!]/(\tau \times \langle x_i/d \rangle) = [\![Q']\!]/\tau$. The algorithm updates the cache to map Q'/τ with v and returns v.

We hence have:

Theorem 4.28. *On input Q, $\mathsf{DPLL}(Q, \langle\rangle, \pi)$ builds a $\{\times, \mathsf{dec}\}$-circuit C computing $[\![Q]\!]$. Moreover, C respects order π.*

Example 4.29. Several interactive examples of this algorithm can be found at https://florent.capelli.me/algorithms/dpll/. We detail an example below. We consider $Q = R(x_0, x_1), S(x_0, x_2), T(x_1, x_3)$ with R, S, T given below.

R	x_0	x_1
	0	1
	1	0
	1	1

S	x_0	x_2
	0	1
	1	0
	1	1

T	x_1	x_3
	0	0
	1	1

With $\pi = (x_0, x_1, x_2, x_3)$, $\mathsf{DPLL}(Q, \tau, \pi)$ produces the circuit shown in Fig. 4.5. Each gate is labeled by the number of the recursive call which produced it. For example, v_1 is produced by the first recursive call on input $(Q, \langle x_0/1\rangle, \pi)$. This recursive call has canonical representation

$$R(0, x_1), S(0, x_2), T(x_1, x_3).$$

We can observe that $G_{Q,\pi}^{\langle x_0/1\rangle}$ has two connected components: the one containing $R(x_0, x_1)$ and $T(x_1, x_3)$ and the one containing $S(x_0, x_2)$. Indeed, the first two atoms are linked by an edge because they share variable x_1, while $S(x_0, x_2)$ is isolated because no other atom contains x_2. Hence a $\times$-gate is created and linked to the result of the first recursive call producing the circuit rooted in v_2 and the one rooted in v_7.

The edge between v_{11} and v_4 is particularly interesting because it corresponds to a cache hit. Indeed, the recursive call producing v_{11} is done with parameters $(Q_1, \langle x_0/1\rangle)$ with $Q_1 = R(x_0, x_1), T(x_1, x_3)$. Since there is exactly one connected component, a decision-gate on x_1 is created and a recursive call with tuple $\langle x_0/1, x_1/0\rangle$ is first made, leading to the creation of the circuit rooted at v_{12}. Then, another recursive call is made with parameters (Q_1, τ_1) where $\tau_1 = \langle x_0/1, x_1/1\rangle$. In this case, the canonical representation of (Q_1, τ_1) is $T(1, x_3)$. But it is the same canonical representation as the one which produced v_4 since the call which created v_4 had parameters (Q_1, τ_0) with $\tau_0 = \langle x_0/0, x_1/1\rangle$ which has canonical representation $T(1, x_3)$. Hence, v_4 is directly returned when the cache is queried for after call (Q_1, τ_1).

4.4.3 Complexity of Exhaustive DPLL

In this section, we study the complexity of Exhaustive DPLL and relate it to the structure of the hypergraph of the query. We leave out most of the proofs as they are quite technical and can be found in [Cap+26]. Given a join query Q, the complexity of $\mathsf{DPLL}(Q, \langle\rangle, \pi)$ depends on the order $\pi = (x_1, \ldots, x_n)$ chosen for the variables. The goal here is to measure how good π is for a join query Q on variables $x_1, \ldots, x_n$.

The case of Acyclic Join Queries. We start by giving some intuition in the case where Q is acyclic. Let $\mathcal{T}$ be a join tree for Q. Assume it is rooted in some atom R_0 and let $\mathcal{T}_1, \ldots, \mathcal{T}_k$ be the subtrees attached to it. Assume $\mathcal{T}_i$ is rooted in atom R_i and let $X_i = \mathsf{var}(R_i)$. Consider an order π on $\mathsf{var}(Q)$ which starts with X_0, that is, we start by assigning variables in the root of the join tree. First, observe that whenever the variables of X_0 are fixed to a value that is not a tuple of R_0, Exhaustive DPLL returns $\bot$. Hence, there will be at most $|R_0|$ different values for X_0 in every recursive call that does not directly return $\bot$.

Now, assume the variables of X_0 have been fixed to $\tau \in R_0$. Observe that $\mathsf{cc}(Q, \tau)$ contains at least k connected components: one for each subtree $\mathcal{T}_i$ attached to the root of $\mathcal{T}$. Hence,

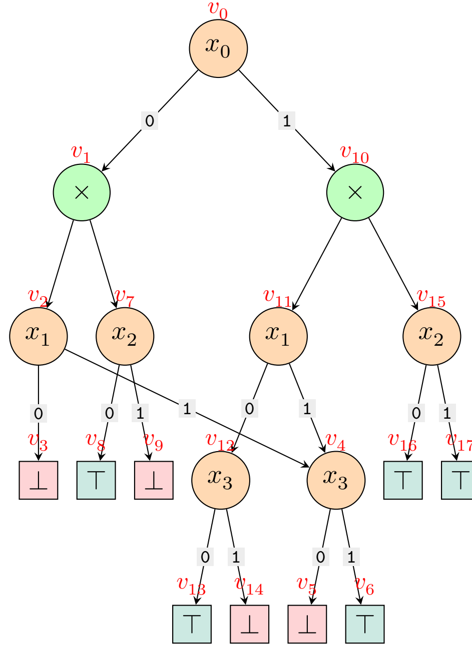


Figure 4.5: The circuit produced from Example 4.29

Exhaustive DPLL will build a $\times$-gate and recursively construct a circuit for $Q_1/\tau_1, \dots, Q_k/\tau_k$ where Q_i is the query corresponding to the atoms in $\mathcal{T}_i$ and $\tau_i = \tau|_{X_i}$.

If later the variables of X_0 are fixed to $\tau' \in R_0$, Exhaustive DPLL will again build a Cartesian product and recursively build a circuit for $Q_1/\tau'_1, \dots, Q_k/\tau'_k$. Now if $\tau_i = \tau'_i$ for some $i \leq k$, then (Q_i, τ_i) and (Q_i, τ'_i) are syntactically equivalent. Hence, the recursive call with parameters (Q_i, τ'_i) will directly return a gate computing Q_i/τ'_i retrieved from the cache.

The previous observation hints at the following: if Q is acyclic with join tree $\mathcal{T}$, then it seems interesting to run $\mathsf{DPLL}(Q, \langle\rangle, \pi)$ with an order π that removes every variable from a bag before removing the ones below. This notion can be formalized as follows: given a rooted join tree $\mathcal{T}$ for Q and a variable $x \in \mathsf{var}(Q)$, we let t_x be the first node of $\mathcal{T}$ where x appears, starting from the root of $\mathcal{T}$. By connectedness, this bag is unique. Now we write $x \preceq_{\mathcal{T}} y$ if t_y is below t_x in $\mathcal{T}$. It is easy to see that this defines a preorder on $\mathsf{var}(Q)$. Now, if π is compatible with $\preceq_{\mathcal{T}}$, meaning that if $x_i \preceq_{\mathcal{T}} x_j$ then we do not have $i > j$, then we can show that $\mathsf{DPLL}(Q, \langle\rangle, \pi)$ runs in time near linear in $\|Q\|$.

However, a join tree is just an artifact here since the algorithm does not depend on it. Recall from Lemma 4.7 that acyclicity can also be characterized in terms of elimination orders: Q is α-acyclic if and only if there exists an α-elimination order $(x_n, \dots, x_1)$ of $\mathcal{H}(Q)$. It turns out that if $\pi^{-1} = (x_n, \dots, x_1)$ is an α-elimination order then $\pi = (x_1, \dots, x_n)$ is an interesting order for $\mathsf{DPLL}(Q, \langle\rangle, \pi)$. Indeed, we can prove the following:

Theorem 4.30 ([CI24]). *Let Q be a join query on domain D and $\pi = (x_1, \dots, x_n)$ an order*

such that $\pi^{-1} = (x_n, \dots, x_1)$ *is an* α*-elimination order of* $\mathcal{H}(Q)$*. Then* $\mathsf{DPLL}(Q, \langle\rangle, \pi)$ *builds a* $\{\times, \mathsf{dec}\}$*-circuit* C *respecting* π *and computing* $[\![Q]\!]$ *in time* $\mathrm{poly}(|Q|) \cdot \|Q\| \cdot |D|$*. In particular,* C *is of size* $\mathrm{poly}(|Q|) \cdot \|Q\| \cdot |D|$*.*

Theorem 4.30 mirrors Theorem 4.12 in the sense that the circuit built is almost linear-size when the input join query is acyclic and a good decomposition is given as input. In the case of Theorem 4.12, the required decomposition is a join tree while in the case of Theorem 4.30, the required decomposition is a reversed α-elimination order. The intuition we provided on the complexity of Exhaustive DPLL when one follows an ordered $\prec_{\mathcal{T}}$ induced by a join tree can be made formal by observing that one can reconstruct a tree decomposition $\mathcal{T}$ of width 1 from π such that π is compatible with $\prec_{\mathcal{T}}$. Theorem 4.30 can be proven without requiring the connection to join trees but join trees offer an interesting insight on why the construction works, see [CI24] for details.

The complexity from Theorem 4.30 is however not fully satisfactory as there is an extra $|D|$ factor in the complexity, which makes the final complexity super linear in the size of the database. In Section 4.6, we explain how a simple trick can be used to transform the $|D|$ dependency into $\log |D|$. The bound is still not as good as that of Theorem 4.12, but we will see that Exhaustive DPLL can be extended into directions where join trees are too limited, for example, to compile join queries where some atoms may be negated, see Section 4.5 for details. We now turn our attention to the non-acyclic case. Our goal is now to measure how good an order π is for Q. To this end, we use a well-known connection between fractional hypertree width and elimination orders.

Hypergraph decomposition and elimination orders. Relations between width of join trees and elimination orders in hypergraphs have been made numerous times in the literature, see for example [ANR15, Appendix C] or [Fic+18; Gan+22; KNR16] but also as incompatibility number in [BCM22, Proposition 14], or via the notion of static width in [Kar+25]. These results generalize the connection between α-acyclicity and α-elimination orders from Lemma 4.7 to any tree decompositions.

Given a hypergraph $H = (V, E)$ and an ordering $\pi = (x_1, \dots, x_n)$ of V, we define the *fractional hyperorder width of* π, denoted by $\mathsf{fhow}(H, \pi)$, is defined as follows: given $i \leq n$, let $\mathcal{N}_{H,\pi}(i)$ be the set of x_j with $j \geq i$ such that there is a path $x_i, x_{i_1}, \dots, x_{i_k}, x_j$ in H such that $i_s \leq i$ for every $s \leq k$. In other words, $\mathcal{N}_{H,\pi}(i)$ is the set of vertices x_j with $j \geq i$ that can be reached from x_i by following only variables in $\{x_1, \dots, x_i\}$. The fractional hyperorder width of π is then defined as $\mathsf{fhow}(H, \pi) = \max_{i \leq n} \rho^*_H(\mathcal{N}_{H,\pi}(i))$ and the fractional hyperorder width of H is defined as $\mathsf{fhow}(H) = \min_\pi \mathsf{fhow}(H, \pi)$. We define similarly the *hyperorder width of* π, denoted by $\mathsf{how}(H, \pi)$, by replacing the fractional cover number ρ^* by the cover number ρ in the definition. The hyperorder width of H is then $\mathsf{how}(H) = \min_\pi \mathsf{how}(H, \pi)$.

Example 4.31. Consider hypergraph H on vertices $\{x_1, \dots, x_6\}$ and edges $\{x_1, x_2\}, \{x_1, x_3\}, \{x_1, x_4\}, \{x_3, x_4\}, \{$
Consider $\pi = (x_1, \dots, x_6)$.

We have $\mathcal{N}_{H,\pi}(1) = \{x_1, x_2, x_3, x_4\}$ because all these vertices are neighbors of x_1 and we can only use x_1 to reach other vertices. We have $\rho^*(\mathcal{N}_{H,\pi}(1)) = 2$ since it is covered

by $\{x_1, x_2\} \cup \{x_3, x_4\}$. Now $\mathcal{N}_{H,\pi}(2) = \{x_2, x_3, x_4, x_5, x_6\}$. Indeed, x_5, x_6 are neighbors of x_2 and x_3, x_4 can be reached from x_2 by path x_2, x_1, x_3 and x_2, x_1, x_4 respectively. Again, $\rho^*(\mathcal{N}_{H,\pi}(2)) = 2$ since $\{x_3, x_4\} \cup \{x_2, x_5, x_6\}$ covers $\mathcal{N}_{H,\pi}(2)$. We can already conclude that $\mathsf{fhow}(H, \pi) = \mathsf{how}(H, \pi) = 2$ because $\mathcal{N}_{H,\pi}(2)$ contains every vertex but x_1, hence $\mathcal{N}_{H,\pi}(j)$ for $j > 2$ can only be smaller.

Consider order $\pi' = (x_5, x_6, x_2, x_1, x_3, x_4)$. In this case,

- $\mathcal{N}_{H,\pi'}(1) = \{x_2, x_5, x_6\}$,
- $\mathcal{N}_{H,\pi'}(2) = \{x_2, x_6\}$,
- $\mathcal{N}_{H,\pi'}(3) = \{x_1, x_2\}$,
- $\mathcal{N}_{H,\pi'}(4) = \{x_1, x_3, x_4\}$,
- $\mathcal{N}_{H,\pi'}(5) = \{x_3, x_4\}$ and
- $\mathcal{N}_{H,\pi'}(6) = \{x_4\}$.

It is easy to check that $\rho(\mathcal{N}_{H,\pi'}(i)) = \rho^*(\mathcal{N}_{H,\pi'}(i)) = 1$ for $i \in \{1, 2, 3, 5, 6\}$, $\rho^*(\mathcal{N}_{H,\pi'}(4)) = 3/2$ and $\rho(\mathcal{N}_{H,\pi'}(4)) = 2$. This establishes that $\mathsf{fhow}(H, \pi') = 3/2$ and $\mathsf{how}(H, \pi') = 2$.

In the context of join queries only, fractional hyperorder width has been called the incompatibility number of π (with respect to Q), denoted by $\iota(Q, \pi)$ in [BCM22]. In this paper, ι is only defined for join queries and is technically not defined on hypergraphs. Moreover, it does not have a non-fractional version. The term fractional hyperorder width is defined in [CI24], where we wanted to stress that the decomposition of the hypergraph is not a tree but an order. In this paper, we use a slightly different but equivalent definition: fractional hyperorder width is defined via an elimination procedure of a vertex which removes a vertex from the hypergraph and adds a new edge corresponding to its neighborhood. Consider hypergraphs $H_1, \dots, H_n$ obtained by removing $x_1, \dots, x_n$ successively. The width of the order is the maximal fractional cover number with respect to H of the neighborhood of x_i in H_i. It is not hard to check by induction that for every i, the neighborhood of x_i in H_i is exactly the set of vertices x_j with $j \geq i$ such that there is a path from x_i to x_j which follows only variables in $\{x_1, \dots, x_i\}$, hence, it is equal to the previously defined $\mathcal{N}_{H,\pi}(i)$. With this definition, it is easy to see that:

Theorem 4.32. *Let H be a hypergraph and $\pi = (x_1, \dots, x_n)$ an order on its vertices. Then the following are equivalent:*

- $\mathsf{fhow}(H, \pi) = 1$,
- $\mathsf{how}(H, \pi) = 1$,
- *π is an α-elimination order of H.*

Fractional hyperorder width and fractional hypertree width are actually the same. More precisely:

Theorem 4.33. *Let $H = (V, E)$ be a hypergraph.*

- *For every order π on V, there exists a tree decomposition $\mathcal{T}_\pi$ of H such that* $\mathsf{fhtw}(H, \mathcal{T}_\pi) \leq \mathsf{fhow}(H, \pi)$ *and* $\mathsf{htw}(H, \mathcal{T}_\pi) \leq \mathsf{how}(H, \pi)$.

- *For every tree decomposition $\mathcal{T}$ of H, there exists an order $\pi_\mathcal{T}$ of H such that* $\mathsf{fhtw}(H, \mathcal{T}) \leq \mathsf{fhow}(H, \pi_\mathcal{T})$ *and* $\mathsf{htw}(H, \mathcal{T}) \leq \mathsf{how}(H, \pi_\mathcal{T})$.

Proof. We only sketch the proof here but it can be found in [ANR15, Appendix C]. To construct the tree decomposition from the order π, we can proceed by induction: we construct for $i = n$ down to 1, a tree decomposition $\mathcal{T}_i$ for H_i as described in the previous paragraph. To build $\mathcal{T}_i$ from $\mathcal{T}_{i+1}$, observe that we only need to add x_i and we need a bag covering $\mathcal{N}_{H,\pi}(i)$. But by definition, the edge $\mathcal{N}_{H,\pi}(i) \setminus \{x_i\}$ is in H_{i+1}, hence there is a bag for it in $\mathcal{T}_{i+1}$. We attach a new leaf labeled by $\mathcal{N}_{H,\pi}(i)$ to it. This edge has fractional cover number at most $\mathsf{fhow}(H, \pi)$ by definition and it does not break the connectedness since x_i only appears here. Also, every edge of H_{i+1} being already covered in $\mathcal{T}_{i+1}$, we only need to show that every edge containing x_i in H_i is covered. But since by definition $\mathcal{N}_{H,\pi}(i)$ is the neighborhood of x_i in H_i, the bag labeled by $\mathcal{N}_{H,\pi}(i)$ covers every edge containing x_i. Moreover, $\rho^*_H(\mathcal{N}_{H,\pi}(i)) \leq \mathsf{fhow}(H, \pi)$ hence $\mathsf{fhtw}(H_i, \mathcal{T}_i) \leq \mathsf{fhow}(H, \pi)$. And, similarly, $\mathsf{htw}(H_i, \mathcal{T}_i) \leq \mathsf{how}(H, \pi)$. The result follows from the fact that $\mathcal{T}_0$ is a tree decomposition of H.

For the other way around: consider a leaf t of $\mathcal{T}$ with parent u. If $B_t \subseteq B_u$, we can remove t from $\mathcal{T}$ without changing the fact that it is a tree decomposition of H of the same width. Repeat until there is at least one leaf t whose bag is not included in its parent bag. In this case, there is some $x_1 \in B_t \setminus B_u$. Since t is a leaf of $\mathcal{T}$, t is the only bag of $\mathcal{T}$ where x_1 appears by connectedness. Hence, the neighborhood N_1 of x_1 has to be covered by B_t hence $\rho^*_H(N_1) \leq \mathsf{fhtw}(H, \mathcal{T})$. Now we build $\mathcal{T}_1$ by removing x_1 from $\mathcal{T}$ and H_1 by remonving x from H and adding a new edge $N_1 \setminus \{x_1\}$ to H_1. $\mathcal{T}_1$ is a tree decomposition of H_1 of width at most $\mathsf{fhtw}(H, \mathcal{T})$. Hence we can proceed recursively and build $\pi = (x_1, \dots, x_n)$, hypergraphs $H_1, \dots, H_n$ and $N_1, \dots, N_n$ where N_i is the neighborhood of x_i in H_i and $\rho^*_H(N_i) \leq \mathsf{fhtw}(H, \mathcal{T})$. We can show that $N_i = \mathcal{N}_{H,\pi}(i)$ for every i, hence establishing that $\mathsf{fhow}(H, \pi) \leq \mathsf{fhtw}(H, \mathcal{T})$ (we similarly have $\mathsf{how}(H, \pi) \leq \mathsf{htw}(H, \mathcal{T})$). □

Fractional hyperorder width is particularly relevant for Exhaustive DPLL because it allows to show a similar result as Theorem 4.19 but for Exhaustive DPLL, by using elimination orders only. We can indeed show the following:

Theorem 4.34 ([CI24]). *Let Q be a join query over domain D and $\pi = (x_1, \dots, x_n)$ an order. Let $k = \mathsf{fhow}(\mathcal{H}(Q), \pi^{-1})$ where $\pi^{-1} = (x_n, \dots, x_1)$. Then $\mathsf{DPLL}(Q, \langle\rangle, \pi)$ produces a $\{\times, \mathsf{dec}\}$-circuit C computing $[\![Q]\!]$ in time* $\mathrm{poly}(|Q|) \cdot |D| \cdot \|Q\|^k$ *such that C respects π and is of size* $\mathrm{poly}(|Q|) \cdot |D| \cdot \|Q\|^k$.

As before, the $|D|$ factor can be removed, see Section 4.6. Theorem 4.34 mirrors Theorem 4.19 since we have a direct connection between the fractional hypertree width of a tree decomposition $\mathcal{T}$ and the existence of an order π having an equivalent fractional hyperorder width by Theorem 4.33.

4.5 Building Relational Circuits for Signed Join Queries

4.5.1 Definitions and notations

We now show how the Exhaustive DPLL algorithm from Section 4.4.2 is more flexible than the approach of the Yannakakis algorithm. We illustrate it by showing that with only a slight modification, we can handle join queries having negated atoms. A *signed join query* $Q = (Q_+, Q_-)$ is given by two lists of atoms: the positive part Q_+ and the negative part Q_-. The answer set $[\![Q]\!]_D$ of Q over domain $D \supseteq \mathsf{dom}(Q_+) \cup \mathsf{dom}(Q_-)$ is defined as the set of tuples $\tau \in D^{\mathsf{var}(Q)}$ such that $\tau|_{\mathsf{var}(Q_+)} \in [\![Q_+]\!]$ and for every relation $R \in Q_-$, $\tau|_{\mathsf{var}(R)} \notin R$. Observe in this case that we need to be explicit on the domain here because otherwise, there may be an infinite number of answers. For example, if a variable only appears in the negative part, then we could define $\tau(x)$ to any value. A query is said to be *safe* if $\mathsf{var}(Q_-) \subseteq \mathsf{var}(Q_+)$. In this case, the answers of Q are necessarily over domain $\mathsf{dom}(Q_+)$ and hence we can omit the domain in the definition of the answers of Q. Unsafe queries can always be made safe by adding a positive unary atom $R_x(x)$ for every variable $x \in \mathsf{var}(Q_-) \setminus \mathsf{var}(Q_+)$ and defining $R_x(x) = D$ with $D \supseteq \mathsf{dom}(Q)$. In this case, for any domain $D' \supseteq D$, $[\![Q]\!]_{D'} = [\![Q]\!]_D$, hence for safe signed join queries, we can simply write $[\![Q]\!]$ to denote the set of answers of Q. Following this discussion, we will always assume the signed queries are safe (since if they are not, we can always normalize them as explained above).

Another way of seeing signed join query is to consider $Q_- = R_1(X_1), \dots, R_m(X_m)$ as the join query $Q'_- = (D^{X_1} \setminus R_1), \dots, (D^{X_m} \setminus R_m)$. In this case, we have $[\![Q]\!]_D = [\![Q_+, Q'_-]\!]$. We extend every notation over join queries to signed join queries. For example, we let $\mathsf{var}(Q) = \mathsf{var}(Q_+) \cup \mathsf{var}(Q_-)$, $\mathsf{dom}(Q) = \mathsf{dom}(Q_+) \cup \mathsf{dom}(Q_-)$, etc. We also let $|(Q_+, Q_-)| = |Q_+| + |Q_-|$ and $\|(Q_+, Q_-)\| = \|Q_+\| + \|Q_-\|$. Observe that the size of a signed join query only makes sense for safe queries. In the case of non-safe queries, we should take into account the size of the domain they are evaluated on since it may have an effect on the output size. This is actually how the size of signed join queries is defined in [Lan23] and [CI24].

We often write Q as $R_1, \dots, R_m, \neg T_1, \dots, \neg T_p$ where $Q_+ = R_1, \dots, R_m$ and $Q_- = T_1, \dots, T_p$. The notation is motivated by the fact that the answers of Q are the tuples that satisfy the atoms in Q_+ but do not satisfy those in Q_-, hence the negation symbol.

Example 4.35. Let Q be the signed join query

$$R(x,y), S(y,z), T(x,z), \neg U(x,y,z)$$

where R, S, T are the same relations as in Fig. 4.1 and U is the relation containing two tuples $\langle x/0, y/0, z/0\rangle$ and $\langle x/2, y/2, z/1\rangle$. We have $Q_+ = R(x,y), S(y,z), T(x,z)$ and $Q_- = U(x,y,z)$. The set of answers of Q_+ is given in Fig. 4.1, hence we can deduce that the set of answers of Q is $\{\langle x/1, y/1, z/1\rangle, \langle x/2, y/2, z/0\rangle\}$ because the answer $\langle x/2, y/2, z/1\rangle$ of Q_+ is a tuple of U, hence not an answer of Q.

Tractable classes of join queries. The tractable classes of signed join queries are different from the tractable classes of join queries. Indeed, the case of α-acyclic signed join queries is

actually as hard as the general case. To see this, consider any signed join query Q over variables X and consider $Q' = Q, \neg R(X)$ where R is the empty relation. Clearly, $[\![Q]\!] = [\![Q']\!]$ but the (unsigned) hypergraph whose edges are the atoms of Q' is α-acyclic because it has an edge covering every variable (the one corresponding to atom $\neg R$).

The trick used in the previous reduction could be used for any negative atom of a signed join query Q: we can always assume the relation of a negative atom to be empty without changing the hypergraph of Q. Hence, it seems that the complexity of finding the answers of Q, if we only consider the structure of its hypergraph, is the worst complexity of finding the answers of a subquery of Q, obtained by taking every positive atom of Q and a subset of negative atoms.

It suggests that β-acyclicity is a better suited property to yield tractable classes for signed join queries, because, by definition, every subquery is α-acyclic, hence every subquery should be easy. Brault-Baron has formalized this intuition and shown that if the join query is purely negative (that is, $Q_+ = \emptyset$), then one can check whether it has a model in linear time (in data complexity) if and only if it is β-acyclic [Bra12]. For the case of signed join queries, with a nonempty positive part, a generalization can be found in his thesis [Bra13, Chapter 6]. In this case, the linear time characterization holds for queries such that for every subset S of negative atom, the hypergraph obtained by keeping every positive atom and every negative atom from S is α-acyclic. A recent treatment of this previously unpublished characterization can be found in [Zha+24]. The corresponding acyclicity, unnamed in Brault-Baron's thesis, has been called "signed-acyclicity" here.

Brault-Baron observed that counting the number of answers to such queries can be solved in linear time using inclusion-exclusion. The downside of this approach is that it is exponential in the number of atoms of the query. It is not clear how Yannakakis algorithm can be used to get polynomial combined complexity for such instances since we do not know a characterization of signed-acyclicity or β-acyclicity using a unique join tree. The only useful definitions for algorithmic purposes we have are formulated in terms of elimination order as illustrated in Lemma 4.6. Indeed, the join trees of the original query and its subqueries can be really different. Consider for example the hypergraph H_n whose edges are $e_n := \{x_1, \dots, x_n\}$ and $p_i := \{x_i, x_{i+1}\}$ for $1 \leq i < n$. A join tree for H_n is a star whose center is labeled by e_n and each branch is labeled by p_i while a join tree for the subhypergraph containing only $p_1, \dots, p_{n-1}$ must be a path.

For counting, we circumvented this problem by designing algorithms leveraging β-elimination orders in [BCM15; Cap17]. The technique used in [Cap17] can actually be understood as a form of Exhaustive DPLL. We now explain how the algorithm from Section 4.4.2 can be slightly modified so it works for signed join queries. In particular, we will prove that given a signed-acyclic join query as input, this modified version of exhaustive DPLL produces a $\{\times, \mathsf{dec}\}$-circuit of size $\|Q\| \cdot |\mathsf{dom}(Q)|$. The $|\mathsf{dom}(Q)|$ factor will be removed in Section 4.6.

4.5.2 Exhaustive DPLL for signed join queries

Recall that exhaustive DPLL takes a join query Q' and a tuple τ' as input and returns a gate in a relational circuit computing $[\![Q']\!]/\tau'$. To do so, we use a syntactic representation so that we can define a syntactic equivalence between (Q', τ') and (Q'', τ'') that guarantees

$[\![Q']\!]/\tau' = [\![Q'']\!]/\tau''$.

In the following, we fix a signed join query $Q = (Q_+, Q_-)$ where $Q_+ = R_1, \ldots, R_p$ and $Q_- = S_1, \ldots, S_q$ and an order $\pi = (x_1, \ldots, x_n)$ on its variables. We let $m = p + q$ to be the total number of atoms in Q. We represent each atom of Q (negative and positive) in memory using the trie-based representation from Section 4.4.1 and Q is represented as two arrays P, N of size p and q respectively, where $P[i]$ holds a pointer to the trie representing R_i and $N[i]$ a pointer to the trie representing S_i.

We also define $Q/\tau = (Q_+/\tau, Q_-/\tau)$. As before, we can get a trie representation of Q/τ easily by simply navigating the tries, as long as τ assigns a prefix $(x_1, \ldots, x_i)$ of variables. The only thing that we need to update is the definition of inconsistency, the definition of syntactic equivalence and the definition of $G^i_{Q,\pi}$.

The crucial observation to adapt all these definitions is the following: let $R \in Q_-$ be a negative atom of Q and assume that $R/\tau = \emptyset$. It means that for every $\sigma \in R$, $\sigma \not\simeq \tau$. In other words, $[\![Q/\tau]\!] = [\![Q'/\tau]\!]$ where Q' is the signed join query obtained by removing R from Q_-. On the other hand, if τ assigns every variable in R and $\tau|_{\mathsf{var}(R)} \in R$, then $[\![Q]\!]/\tau = \emptyset$ because a negative atom is satisfied. We therefore say that τ is inconsistent with Q if either τ is inconsistent with Q_+, or there exists $R \in Q_-$ such that $\tau|_{\mathsf{var}(R)} \in R$. We can still check for inconsistency in time poly($|Q|$) since to check the latter condition, we only need to check for every atom of Q_-, whether we hit a leaf of the trie after fixing the variables in τ.

Now, given two subqueries Q_1 and Q_2 of Q (that is, they are formed by subsets of positive and negative atoms of Q) and two tuples τ_1, τ_2, we say that (Q_1, τ_1) and (Q_2, τ_2) are syntactically equivalent if the following holds:

1. Both Q_1 and Q_2 are consistent with τ_1 and τ_2.
2. Let P_1 be the set of positive atoms R of Q_1 such that $\mathsf{var}(R) \subseteq \mathsf{var}(\tau_1)$ and let N_1 be the set of negative atoms R of Q_1 such that $R/\tau_1 = \emptyset$. Define P_2 and N_2 similarly. Let $Q_1^* := ((Q_1)_+ \setminus P_1, (Q_1)_- \setminus N_1)$ and $Q_2^* := ((Q_2)_+ \setminus P_2, (Q_2)_- \setminus N_2)$. We have that Q_1^* and Q_2^* have the same positive and negative atoms and for every atom $R \in Q_1^*$ (equivalently, Q_2^*), we have $\tau_1|_{\mathsf{var}(R)} = \tau_2|_{\mathsf{var}(R)}$.

The only difference between the definition of syntactic equivalence now is that we remove from Q_1 and Q_2 the negative atoms that cannot be satisfied by extending τ_1 or τ_2, hence which do not change the answers of Q_1 or Q_2 that are compatible with τ_1, τ_2 respectively. As before, we have $[\![Q_1]\!]/\tau_1 = [\![Q_2]\!]/\tau_2$. We also define the canonical representation of (Q_1, τ_1) as being two tables P', N' with p and q pointers respectively. We let $P'[i]$ be a pointer to the trie node representing R_i/τ_1 if $R_i \in Q_1^+$ and a null pointer otherwise. And we let $N'[i]$ be a pointer to the trie node representing S_i/τ_1 if $S_i \in Q_1^-$ and a null pointer otherwise. Clearly, (Q_1, τ_1) and (Q_2, τ_2) are syntactically equivalent if and only if they have the same canonical representation.

Example 4.36. Let Q be the signed join query

$$R(x_1, x_2), S(x_1, x_3), T(x_1, x_4), \neg U(x_1, x_2, x_3, x_4)$$

with order $\pi = (x_1, x_2, x_3, x_4)$ and $R = \{0,1\}^{\{x_1,x_2\}}$, $S = \{0,1\}^{\{x_1,x_3\}}$, $T = \{0,1\}^{\{x_1,x_4\}}$, $U = \{\langle x_1/0, x_2/0, x_3/0, x_4/0\rangle\}$.

Let $\tau_1 = \langle x_1/1\rangle$. Observe that $U/\tau_1 = \emptyset$. Hence the canonical representation of Q/τ_1 is $R(1, x_2), S(1, x_3), T(1, x_4)$. Now if $\tau_2 = \langle x_1/0\rangle$, the canonical representation of Q/τ_2 is $R(0, x_2), S(0, x_3), T(0, x_4), \neg U(0, x_2, x_3, x_4)$.

We need one last slight modification: the definition of $\mathsf{cc}(Q, \tau)$. Indeed, we should take into account that negative atoms R such that $R/\tau = \emptyset$ should not be used in the definition of $\mathsf{cc}(Q, \tau)$ since removing them does not change the answer set. Hence, given a signed join query Q and a tuple τ such that τ is consistent with Q, we define $G^\tau_{Q,\pi}$ to be the graph whose vertices are the positive atoms $R \in Q_+$ such that $\mathsf{var}(R) \setminus \mathsf{var}(\tau) \neq \emptyset$ and the negative atoms $R \in Q_-$ such that $R/\tau \neq \emptyset$. There is an edge between two atoms R, S in $G^\tau_{Q,\pi}$ if and only if $(\mathsf{var}(R) \cap \mathsf{var}(S)) \setminus \mathsf{var}(\tau) \neq \emptyset$. We let $\mathsf{cc}(Q, \tau)$ be the connected components of $G^\tau_{Q,\pi}$.

Example 4.37. Consider the query and tuples from Example 4.36. We have that $\mathsf{cc}(Q, \tau_1)$ has three connected components, one per positive atom while $\mathsf{cc}(Q, \tau_2)$ has one connected component containing every atom of Q. This is because in the first case, $\neg U(x_1, x_2, x_3, x_4)$ is not in $G^{\tau_1}_{Q,\pi}$ which disconnects the graph.

The exhaustive DPLL algorithm is as described in Section 4.4.3 with the adapted notion of consistency, syntactic equivalence and connected components. Its correctness is proven similarly as before. Indeed, we still have that syntactic equivalence between (Q', τ') and (Q'', τ'') implies that $[\![Q']\!]/\tau' = [\![Q'']\!]/\tau''$ and that $[\![Q]\!]/\tau' = \bigtimes_{C \in \mathsf{cc}(Q',\tau')} [\![C]\!]/\tau'$.

Width measures for signed queries. As hinted before, the complexity of Exhaustive DPLL should take into account the fact that some negative atoms could be empty, and hence, any result concerning the tractability of signed join queries must differentiate the role of positive and negative atoms. Given a signed query Q, considering the hypergraph formed by all of its atoms is hence too crude to understand correctly the landscape of tractability. To account for negative atoms, we introduce the notion of *signed hypergraph* (which is called bicolor hypergraph in [Bra13]): a signed hypergraph $H = (V, E_+, E_-)$ is defined as a set of vertices V and two sets of edges $E_+ \subseteq 2^V$ and $E_- \subseteq 2^V$ called the positive and negative edges of H respectively. Given a signed join query $Q = (Q_+, Q_-)$, we define the *signed hypergraph* $\mathcal{H}(Q)$ *of* Q (or simply the hypergraph of Q, when it is clear from context that Q is signed) as the hypergraph whose vertices are $\mathsf{var}(Q)$, positive edges $E_+ = \{\mathsf{var}(R) \mid R \in Q_+\}$ and negative edges $E_- = \{\mathsf{var}(R) \mid R \in Q_-\}$.

We now generalize the notions of fractional hyperorder width to signed hypergraphs. From what precedes, for this notion to make sense regarding the tractability of signed join queries, it has to be hereditary: that is, if H' is a signed hypergraph obtained by removing negative edges from H, we want the width of H' to be smaller than the width of H. Observe that it does not hold for hypertree width as removing a single atom can increase the width arbitrarily. In the context of hypergraphs, β-acyclicity is a good example on how one can address the hereditary problem: a hypergraph H is β-acyclic if every subhypergraph $H' \subseteq H$ is α-acyclic.

The notion of β-acyclicity is hence hereditary by its very definition, but it is not a measure. This has been straightforwardly adapted as a measure as follows: the *β-hypertree width* of a hypergraph H, $\beta\text{-}\mathsf{htw}(H)$ for short, is defined as $\max_{H' \subseteq H} \mathsf{htw}(H')$ [GP04]. We define similarly the *fractional β-hypertree width $\beta\text{-}\mathsf{fhtw}(H)$ of H* as $\max_{H' \subseteq H} \mathsf{fhtw}(H')$. Unfortunately, these measures have not yet led to insights regarding the combined tractability of join queries because their definition does not provide any decomposition that could be leveraged into an algorithm. The only exception is β-acyclicity: its characterization by β-elimination order (see Lemma 4.6) has been used to establish several tractability results for signed join queries [BCM15], CNF-formulas [Cap17] or other optimization problems [DD23] (see Chapter 5 for details regarding this result).

A key observation regarding β-elimination orders is that they are α-elimination orders for every subhypergraph. More precisely, we have the following:

Lemma 4.38. *Let $H = (V, E)$ be a hypergraph and $\pi = (x_1, \dots, x_n)$ an order on V. We have that π is a β-elimination order if and only if for every $H' \subseteq H$, π is an α-elimination order of H'.*

Observe that it does not work if we try to replace elimination orders by tree decomposition in the following sense: consider the star hypergraph H_n on vertices $[n]$ and edges $e_i = \{0, i\}$ for $1 \leq i \leq n$ and $e_0 = [n]$. There is no tree decomposition $\mathcal{T}$ such that for every $H' \subseteq H$, $\mathsf{htw}(H', \mathcal{T}) = 1$. Indeed, any tree decomposition of H must contain a bag labeled by $[n]$ to cover e_0. However, if we consider $H' = H \setminus e_0$, this bag is not covered by one edge anymore, hence $\mathsf{htw}(H', \mathcal{T}) > 1$. Elimination orders are more flexible than tree decompositions in this case, because they are less constrained.

Lemma 4.38 serves as an inspiration for the following definition: given an order π on V, we define the *signed fractional hyperorder width of H with respect to* π, denoted by $\mathsf{sfhow}(H, \pi)$, as $\max_{E' \subseteq E_-} \mathsf{fhow}((V, E_+ \cup E'), \pi)$. That is, we consider every hypergraph we can build by taking the union of the positive edges with a subset of negative edges and take the one which has the largest fractional hyperorder width with respect to π. The signed fractional hyperorder width of $H = (V, E_+, E_-)$ is defined as $\mathsf{sfhow}(H) := \min_\pi \mathsf{sfhow}(H, \pi)$ where π runs over every ordering of V. Similarly, we define the *signed hyperorder width* as $\mathsf{show}(H, \pi) := \max_{E' \subseteq E_-} \mathsf{how}((V, E_+ \cup E'), \pi)$. Observe that by Lemma 4.38, for a signed hypergraph H having only negative edges, $\mathsf{sfhow}(H, \pi) = 1$ if and only if π is a β-elimination order of H (seen as a normal hypergraph). Similarly, we let $\mathsf{show}(H) = \min_\pi \mathsf{show}(H, \pi)$.

Example 4.39. Consider hypergraph H on vertices $\{v_1, v_2, v_3\}$ and positive edges $e_3 = \{v_1, v_2\}$, $e_1 = \{v_2, v_3\}$ and $e_2 = \{v_1, v_3\}$ and negative edges $e_0 = \{v_1, v_2, v_3\}$ and $\pi = (v_1, v_2, v_3)$. Then we have $\mathsf{fhow}(H, \pi) = \mathsf{how}(H, \pi) = 1$ because edge e_0 covers every vertex. However, for $H' = H \setminus \{e_0\}$, we have $\mathsf{fhow}(H, \pi) = 3/2$ and $\mathsf{how}(H, \pi) = 2$. Consequently, $\mathsf{sfhow}(H, \pi) = 3/2$ and $\mathsf{show}(H, \pi) = 2$. The vertices being completely symmetric, we conclude that $\mathsf{sfhow}(H) = 3/2$ and $\mathsf{show}(H) = 2$.

In a way, signed hyperorder width can be seen as a simplification of β-hypertree width when we pick a decomposition that is "good" for every subhypergraph while, in β-hypertree

width, the best decomposition of each subhypergraph can be different. Moreover, β-acyclicity corresponds to these widths to be equal to 1 because of Lemma 4.38. To compare such notions, defined on hypergraphs, and notions defined on signed hypergraphs, we cast a hypergraph $H = (V, E)$ into the signed hypergraph $H_- = (V, \emptyset, E)$ defined as the hypergraph with no positive edges and whose negative edges are H. We have:

Theorem 4.40. *For every hypergraph $H = (V, E)$, we have β-$\mathsf{fhtw}(H) \leq \mathsf{sfhow}(H_-)$ and β-$\mathsf{htw}(H) \leq \mathsf{show}(H_-)$. Moreover, H is β-acyclic iff β-$\mathsf{fhtw}(H) = 1$, iff β-$\mathsf{htw}(H) = 1$, iff $\mathsf{sfhow}(H_-) = 1$, iff $\mathsf{show}(H_-) = 1$.*

Actually, for a signed hypergraph, the case $\mathsf{sfhow}(H) = \mathsf{show}(H) = 1$ corresponds to the notion of signed acyclicity from [Bra13; Zha+24].

Another generalization of β-acyclicity, the nest-set width, has been introduced in [Lan23]. Let $H = (V, E)$ be a hypergraph. A *nest set of* H is a subset $S \subseteq V$ such that $E_S := \{e \setminus S \mid e \in E, e \cap S \neq \emptyset\}$ is ordered by inclusion, that is, for every $e_1, e_2 \in E_S$, either $e_1 \subseteq e_2$ or $e_2 \subseteq e_1$. In other words, $E_S = \{e_1, \dots, e_q\}$ with $e_1 \subseteq \dots \subseteq e_q$. Observe that a nest set is a generalization of the notion of nest point, in the sense that if x is a nest point in H, then $\{x\}$ is a nest set of H. A *nest set elimination order* Π is a sequence $S_1, \dots, S_p$ such that $S_1, \dots, S_p$ is a partition of V (that is $S_i \cap S_j = \emptyset$ for every $i \neq j$ and $\bigcup_i S_i = V$) and for every $i \leq p$, S_i is a nest set of $H \setminus \bigcup_{j<i} S_j$. The *nest set width* of H w.r.t. Π is defined as $\mathsf{nsw}(H, \Pi) := \max_{i \leq p} |S_i|$. The nest set width of H is defined as $\mathsf{nsw}(H) = \min_\Pi \mathsf{nsw}(H, \Pi)$ where Π runs over every nest set elimination order of H.

Example 4.41. Let H be a hypergraph with vertices $\{v_1, \dots, v_5\}$ and edges $e_1 = \{v_2, v_3\}, e_2 = \{v_1, v_3\}, e_3 = \{v_1, v_2\}, e_4 = \{v_1, \dots, v_4\}, e_5 = \{v_1, \dots, v_5\}$. H is not β-acyclic since it contains the triangle v_1, v_2, v_3 as a subhypergraph. But $S_1 = \{v_1, v_2\}$ is a nest set because $e_3 \setminus S_1 = \emptyset \subseteq e_1 \setminus S_1 = e_2 \setminus S_1 = \{v_3\} \subseteq e_4 \setminus S_1 = \{v_3, v_4\} \subseteq e_5 \setminus S_1 = \{v_3, v_4, v_5\}$. Hence $S_1, \{v_3\}, \{v_4\}, \{v_5\}$ is a nest set elimination order of width 2.

It is not hard to see that nest set width is hereditary. Indeed, a nest set elimination order for H is still a nest set elimination order for $H' \subseteq H$, hence $\mathsf{nsw}(H') \leq \mathsf{nsw}(H)$. Moreover, given a nest set elimination order $\Pi = (S_1, \dots, S_p)$, if $\pi = (x_1, \dots, x_n)$ is an order obtained by concatenating arbitrary orders on $S_1, \dots, S_p$, then $\mathsf{show}(H_-, \pi) \leq \mathsf{nsw}(H, \Pi)$ (see [CI24] for a proof), establishing:

Theorem 4.42. *For every hypergraph H, $\mathsf{show}(H_-) \leq \mathsf{nsw}(H)$.*

Complexity of Exhaustive DPLL on signed join queries. It turns out that we can generalize Theorem 4.34 to signed queries by considering the signed hyperorder width. We have the following:

Theorem 4.43 ([CI24])**.** *Let Q be a signed join query on domain D and $\pi = (x_1, \dots, x_n)$ an order. Let $k = \mathsf{show}(\mathcal{H}(Q), \pi^{-1})$ where $\pi^{-1} = (x_n, \dots, x_1)$. Then $\mathsf{DPLL}(Q, \langle\rangle, \pi)$ produces a $\{\times, \mathsf{dec}\}$-circuit C computing $[\![Q]\!]$ in time $|Q|^{O(k)} \cdot |D| \cdot \|Q\|^k$ such that C respects π and is of size $|Q|^{O(k)} \cdot |D| \cdot \|Q\|^k$.*

Hence, Exhaustive DPLL runs in polynomial time on instances having bounded signed hyperorder width, even when we consider combined complexity. It generalizes several results concerning the tractability of signed and negative join queries (i.e., signed join queries without positive atoms), such as the tractability of counting answers of β-acyclic join queries [BCM15] or the tractability of model checking on queries of bounded nest-set width [Lan23].

One stark contrast with Theorem 4.34 however is that we do not consider the *fractional* signed hyperorder width. The proof technique used in [CI24] allows to prove only the partial result:

Theorem 4.44 ([CI24]). *Let Q be a signed join query on domain D with m negative atoms and $\pi = (x_1, \dots, x_n)$ an order. Let $k = \mathsf{sfhow}(\mathcal{H}(Q), \pi^{-1})$ where $\pi^{-1} = (x_n, \dots, x_1)$. Then $\mathsf{DPLL}(Q, \langle\rangle, \pi)$ produces a $\{\times, \mathsf{dec}\}$-circuit C computing $[\![Q]\!]$ in time $2^m \operatorname{poly}(|Q|) \cdot |D| \cdot \|Q\|^k$ such that C respects π and is of size $2^m \operatorname{poly}(|Q|) \cdot |D| \cdot \|Q\|^k$.*

In other words, when we relax the notion of signed hyperorder width to signed fractional hyperorder width, the current proof technique does not allow to prove a polynomial runtime in combined complexity. We hence leave the following question open:

Open question 9. *Is the runtime of $\mathsf{DPLL}(Q, \langle\rangle, \pi)$ polynomial in $\|Q\|$ when $\mathsf{sfhow}(\mathcal{H}(Q), \pi)$ is bounded?*

One interesting direction regarding this question is to clarify the relation between signed hyperorder width and signed fractional hyperorder width. While we know they are separated because there are classes of hypergraphs having bounded fractional hyperorder width but unbounded hyperorder width [GM14], it is not clear whether these measures are separated when looking at hypergraphs having only negative edges. Indeed, we know that bounded fractional β-hypertree width is equivalent to bounded β-hypertree width [CRZ20]. A similar phenomenon could also occur for signed hyperorder width on hypergraphs with no positive edges. Actually, it seems that the proof of Theorem 4.43 still proves that Exhaustive DPLL has polynomial combined complexity for a hybrid of signed fractional hyperorder width and signed hyperorder width where positive edges can be used fractionally to cover vertices, but negative edges can only be used with weight 1.

Our current focus on polynomial combined complexity may seem artificial in this chapter about databases where one is often more interested in data complexity but let us quickly go back to the realm of propositional logic to explain why it matters. A clause $C = \ell_1 \vee \dots \vee \ell_k$ over variables $x_1, \dots, x_k$ may be seen as the complement of a relation R_C over $X = \{x_1, \dots, x_k\}$, domain $\{0, 1\}$, containing the only assignment $\tau \in 2^X$ that does not satisfy C. Hence, we can map a CNF formula $F = \bigwedge_{i=1}^m C_i$ to a signed join query $Q_F = \neg R_{C_1}, \dots, \neg R_{C_m}$ such that $[\![Q]\!]_F$ is exactly the set of models of F. A polynomial algorithm for combined complexity would hence directly tanslate into tractability for SAT, #SAT etc.

4.5.3 Exhaustive DPLL and Conjunctive Queries

We have seen in Theorem 4.23 that the Yannakakis Algorithm can be adapted for conjunctive queries as long as the tree decomposition is free-connex, that is, the free variables form a connected subtree. The main insight is the following:

Theorem 4.45. *Given a* $\{\times, \mathsf{dec}\}$*-circuit* C *respecting* $\pi = (x_1, \dots, x_n)$ *and* $i \leq n$*, we can construct a* $\{\times, \mathsf{dec}\}$*-circuit* C' *of size at most* $|C|$*, respecting* $\pi' = (x_1, \dots, x_i)$ *and computing* $\mathsf{rel}(C)|_{x_1,\dots,x_i}$ *in time* $O(|C|)$*.*

Proof. First observe that since C respects π, for every gate g of C, we have $\mathsf{var}(g) \subseteq \{x_j, \dots, x_n\}$ for some $j \leq n$. We build C' by simply replace every gate g of C such that $\mathsf{var}(g) \subseteq \{x_{i+1}, \dots, x_n\}$ by $\top$ if $\mathsf{rel}(g) \neq \emptyset$ and by $\bot$ otherwise. We leave the rest of the circuit unchanged. We claim that for every gate g' of C' corresponding to one gate g of C, we have that g' computes $\mathsf{rel}(g)|_{\{x_1,\dots,x_i\}}$. It is clear for every gate g' of C' that have been constructed by replacing a gate g of C by $\top$ or $\bot$. For every other gates, it can be shown by bottom-up induction. If g' is a $\times$-gate with input $g'_1, \dots, g'_k$ corresponding to a gate g of C with input $g_1, \dots, g_k$, we know by induction that $\mathsf{rel}(g'_j) = \mathsf{rel}(g)|_{\{x_1,\dots,x_i\}}$ and it is straightforward to see that $\mathsf{rel}(g') = \bigtimes_{j=1}^{k} \mathsf{rel}(g_j)|_{\{x_1,\dots,x_i\}} = (\bigtimes_{j=1}^{k} \mathsf{rel}(g_j))|_{\{x_1,\dots,x_i\}} = \mathsf{rel}(g)|_{\{x_1,\dots,x_i\}}$. We have a similar induction when g is a decision-gate (necessarily on a variable x_j with $j \leq i$). □

Now a direct corollary of this is that if $Q(S)$ is a signed join query on variables X and $\pi = (x_1, \dots, x_n)$ is an order such that $S = \{x_1, \dots, x_i\}$ for some $i \leq n$, then we can run $\mathsf{DPLL}(Q, \langle\rangle, \pi)$ to construct a $\{\times, \mathsf{dec}\}$-circuit computing $[\![Q]\!]$ and project it onto S using Theorem 4.45 to build a circuit computing $[\![Q(S)]\!]$. The projection being linear, we get the same guarantees as in Theorem 4.34 and Theorem 4.43, as long as the order π starts with the free variables S.

Interestingly, in the case of (positive) conjunctive queries, we can show that from an ext-S-connex tree decomposition of Q of fractional hypertree width k, we can extract an order π starting with S such that $\mathsf{fhow}(\mathcal{H}(Q), \pi^{-1}) = k$. Conversly, we can also construct a tree decomposition $\mathcal{T}$ from an order π starting with S such that $\mathsf{fhow}(\mathcal{H}(Q), \pi^{-1}) = \mathsf{fhtw}(\mathcal{H}(Q), \mathcal{T})$. In other words, we can generalize Theorem 4.33 so that ext-S-connex tree decompositions correspond to elimination orders ending with S.

Another way of interpreting this circuit transformation is to directly modify the Exhaustive DPLL procedure. Indeed, we can decide that whenever we perform a recursive call on a subquery Q' and assignment τ' on variables Z with $(\mathsf{var}(Q') \setminus Z) \cap S = \emptyset$, that is, where Q' does not contain unassigned free variables anymore, then instead of building a circuit computing $[\![Q']\!]/\tau'$, we can simply perform a model checking algorithm on Q'/τ' to check whether $[\![Q']\!]/\tau'$ is empty or not and return $\top$ or $\bot$ accordingly. This approach has a disadvantage: we may lose some speedup given by the caching mechanism of Exhaustive DPLL. On the other hand, depending on the structure of Q', it may be more advantageous to run to simply run an efficient model checking algorithm than to build a complete circuit for $[\![Q']\!]/\tau'$.

4.6 Binarization

The complexity bounds from Theorem 4.34 and Theorem 4.43 depend on the size of the domain which makes them weaker than what we would obtained from tree decompositions. We explain here a simple trick to remove this dependency and trade it for a factor polynomial in $\log |D|$ instead. We start by observing that this factor is not an over-estimation in the analysis of the algorithm but that it can effectively appear on a concrete example.

Example 4.46. Consider the query $Q := A(x_1) \wedge B(x_2) \wedge \neg R(x_1, x_2)$ (which has already been considered in [Zha+24] for similar reasons). Consider the order x_1, x_2 and take $A(x_1) = [d]$, $B(x_2) = [d]$ and $R(x_1, x_2) = \{(v, v) \mid v \in [d]\}$ (i.e., $\neg R(x_1, x_2)$ enforces $x_1 \neq x_2$). We have $|R|+|A|+|B| = O(d)$, and the signed hyperorder width of Q is 1, but DPLL will produce a quadratic size circuit. Indeed, $\neg R(x_1, x_2)$ prevents DPLL from creating any Cartesian product. Moreover, for each $d_1 \leq d$, $Q/\langle x_1/d_1 \rangle$ is $B(x_2) \wedge \neg R(d_1, x_2)$. Hence, for $d_1 \neq d_1'$, $Q/\langle x_1/d_1 \rangle$ and $Q/\langle x_1/d_1' \rangle$ are not syntactically equivalent since they do not assign the same value to x_1. It means that for each $d_1 \leq d$, the recursive calls on $\langle x_1/d_1 \rangle$ do not trigger the cache. From this recursive call, one call on $\langle x_1/d_1, x_2/d_2 \rangle$ for every $d_2 \neq d_1$ will be done, yielding a circuit of size $O(d^2)$.

To overcome this inefficiency, we rely on a simple trick which consists in encoding the domain in binary and transforming the query and the database accordingly. It turns out that this transformation can be done in a way that preserves the structure of the query.

In this section we fix a query Q on set of variables X on domain $D = [d]$ and let $b = \lceil \log(d) \rceil$. For $v \in [d]$, we denote by $\widetilde{v}^b$ the binary encoding of v over b bits and let $\widetilde{v}^b[i]$ be the ith bit of $\widetilde{v}^b$ for every $1 \leq i \leq b$. For a variable $x \in X$, we introduce fresh variables $x^1, \dots, x^b$ and let $\widetilde{X}^b = \{x^i \mid x \in X, 1 \leq i \leq b\}$. For a tuple $\tau \in D^Y$, we let $\widetilde{\tau}^b$ be the tuple in $D^{\widetilde{Y}^b}$ such that for every $y \in Y$ and $1 \leq i \leq b$, $\widetilde{\tau}^b(y^i) = \widetilde{\tau(y)}^b[i]$. For a relation $R \subseteq D^Y$, we let $\widetilde{R}^b = \{\widetilde{\tau}^b \mid \tau \in R\} \subseteq D^{\widetilde{Y}^b}$. For an atom A (positive or negative) on a set of variables Y, we let $\widetilde{A}^b$ be the corresponding atom on variables set $\widetilde{Y}^b$. We let $\widetilde{Q}^b$ be the query on a set of variables $\widetilde{X}^b$ whose atoms are $\widetilde{R}^b$ for each atom R of Q.

Example 4.47. Let $Q := A(x_1), B(x_2), \neg R(x_1, x_2)$ and $D = [3]$. Then $\widetilde{Q}^2$ is

$$\widetilde{A}^2(x_1^1, x_1^2), \widetilde{B}^2(x_2^1, x_2^2), \neg\widetilde{R}^2(x_1^1, x_1^2, x_2^1, x_2^2).$$

The contents of each atom of Q and their binarized versions are given in Fig. 4.6a and Fig. 4.6b respectively.

Binarization is interesting since there is a natural isomorphism between $[\![Q]\!]$ and $[\![\widetilde{Q}^b]\!]$. Indeed, given $\tau \in [\![Q]\!]$, it is easy to see that $\widetilde{\tau}^b$ is an answer of $\widetilde{Q}^b$. Similarly, given $\tau \in \{0,1\}^{\widetilde{X}^b}$, we let $\bar{\tau}^b \in D^X$ be the tuple defined by, for every $x \in X$, $\bar{\tau}^b = \sum_{i=1}^b 2^{i-1}\tau(x^i)$. It is clear that for every $\tau \in [\![\widetilde{Q}^b]\!]$, $\bar{\tau}^b \in [\![Q]\!]$. Hence there is a one-to-one correspondence between the answers of Q and of $\widetilde{Q}^b$ which simply consists in writing each domain value in binary, on b bits.

Moreover, for an order $\pi = (x_1, \dots, x_n)$ on X, we denote by $\widetilde{\pi}^b$ the order on $\widetilde{X}^b$ defined by $(x_1^b, \dots, x_1^1, \dots, x_n^b, \dots, x_n^1)$. It turns out that this transformation preserves hyperorder width:

Theorem 4.48. *For every query Q and $b \in \mathbb{N}$, $\mathsf{show}(\mathcal{H}(\widetilde{Q}^b), \widetilde{\pi}^b) = \mathsf{show}(\mathcal{H}(Q), \pi)$ and $\mathsf{sfhow}(\mathcal{H}(\widetilde{Q}^b), \widetilde{\pi}^b) = \mathsf{sfhow}(\mathcal{H}(Q), \pi)$.*

If we take $b = \lceil \log |D| \rceil$, $\widetilde{Q}^b$ is a signed join query with $O(|X| \cdot \log |D|)$ variables, hence $|\widetilde{Q}^b| = O(|Q| \log |D|)$ and $\|\widetilde{Q}^b\| = O(\|Q\| \log |D|)$. Hence, if instead of running $\mathsf{DPLL}(Q, \langle\rangle, \pi)$,

A	x_1
	0
	1
	2

B	x_2
	1
	2
	3

R	x_1	x_2
	0	0
	1	1
	2	2
	3	3

(a) Original data

$\widetilde{A}^2$	x_1^1	x_1^2
	0	0
	1	0
	0	1

$\widetilde{B}^2$	x_2^1	x_2^2
	1	0
	0	1
	1	1

$\widetilde{R}^2$	x_1^1	x_1^2	x_2^1	x_2^2
	0	0	0	0
	1	0	1	0
	0	1	0	1
	1	1	1	1

(b) Binarized data with $b = 2$

Figure 4.6: Binarization of join queries

we run $\mathsf{DPLL}(\widetilde{Q}^b, \langle\rangle, \widetilde{\pi}^b)$, we construct a circuit representing $[\![Q]\!]$ in binary, hence the complexity from Theorem 4.34 and Theorem 4.43 will no longer depend linearly on $|D|$.

Theorem 4.49 ([CI24]). *Let Q be a join query on domain D and $\pi = (x_1, \dots, x_n)$ an order. Let $k = \mathsf{fhow}(\mathcal{H}(Q), \pi^{-1})$ where $\pi^{-1} = (x_n, \dots, x_1)$. Then $\mathsf{DPLL}(\widetilde{Q}^b, \langle\rangle, \widetilde{\pi}^b)$ produces a $\{\times, \mathsf{dec}\}$-circuit C computing $[\![\widetilde{Q}^b]\!]$ in time $\mathrm{poly}(|Q| \log |D|) \cdot \|Q\|^k$ such that C respects $\widetilde{\pi}^b$ and is of size $\mathrm{poly}(|Q| \log |D|) \cdot \|Q\|^k$.*

Theorem 4.50 ([CI24]). *Let Q be a signed join query on domain D and $\pi = (x_1, \dots, x_n)$ an order. Let $k = \mathsf{show}(\mathcal{H}(Q), \pi^{-1})$ where $\pi^{-1} = (x_n, \dots, x_1)$. Then $\mathsf{DPLL}(\widetilde{Q}^b, \langle\rangle, \widetilde{\pi}^b)$ produces a $\{\times, \mathsf{dec}\}$-circuit C computing $[\![\widetilde{Q}^b]\!]$ in time $(|Q| \log |D|)^{O(k)} \cdot \|Q\|^k$ such that C respects $\widetilde{\pi}^b$ and is of size $(|Q| \log |D|)^{O(k)} \cdot \|Q\|^k$.*

Example 4.51. Consider again the instance $Q = A(x_1) \wedge B(x_2) \wedge (x_1 \neq x_2)$ from Example 4.46 where we proved that caching would never occur with this instance, resulting in a circuit of size at least $O(d^2)$. However, when running DPLL on $\widetilde{Q}^b$, we can see that caching will now often happen. Indeed, assume that the algorithm is in a state where every copy $x_1^b, \dots, x_1^1$ of x_1 has already been set to values $v_b, \dots, v_1$ and the algorithm is now setting variables $x_2^b, \dots, x_2^1$. Observe that as soon as some copy x_2^i is set to the value $1 - v_i$, then $\neg\widetilde{R}^b$ is satisfied since the value assigned to x_1 is necessarily different from the value assigned to x_2. Hence, the atom $\neg\widetilde{R}^b$ is now simplified and recursive calls happen with only one atom B. Hence caching occurs for every subset of $\widetilde{B}^b$ where some prefix of $x_2^b, \dots, x_2^1$ is set, which yields $O(d \log d)$ values, less than the $O(d^2)$ from Example 4.46.

4.7 Using Relational Circuits

In this section, we review a few applications of relational circuits and show how they allow us to recover existing results in database literature.

4.7.1 Constant Delay Enumeration

One of the earliest extensions of the Yannakakis algorithm has been made by Bagan, Durand and Grandjean [BDG07], who observed that the answers of a free-connex acyclic conjunctive query could be enumerated with constant delay after a linear preprocessing. It seems to be more powerful than what we can do with deterministic DNNF circuits since enumeration algorithms for such circuits have at least a delay that depends on their depth. But one can observe that this is actually possible to recover the results from circuits alone. Indeed, in this case, constant delay is to be understood as constant in terms of data complexity, that is, the complexity may depend on the number of variables of the query. From this point of view, a $\{\times, \mathsf{dec}\}$-circuit can be seen as a constant depth circuit. Technically, $\{\times, \mathsf{dec}\}$-circuits can be of arbitrary depth, but only if they have paths formed by an unbounded number of $\times$-gates. The number of decision-gates from the output of the circuit to one of its inputs is bounded by the total number of variables since no variable can appear twice on the same path. In other words, we have the following:

Lemma 4.52. *Let C be a $\{\times, \mathsf{dec}\}$-circuit on variables X such that no $\times$-gate has another $\times$-gate as an input. The depth of C is at most $2|X|$.*

Now, we present an algorithm allowing to enumerate the tuples in a relation represented by a $\{\times, \uplus\}$-circuits with a delay between two tuples that only depends on the number of variables and its depth, a result that can be found in [OZ15]. From this, we will directly recover the result from [BDG07] and extend it to signed queries since on input Q, exhaustive DPLL constructs a circuit whose depth is at most $2|\mathsf{var}(Q)|$. The construction of the circuit is then the preprocessing needed for the algorithm to work and it can be done in linear time on acyclic join queries.

Enumeration algorithm. The enumeration algorithm works inductively as follows: we explain how to enumerate $\mathsf{rel}(g)$ for every g in the circuit, provided that we know how to enumerate $\mathsf{rel}(g')$ for every g' below g in the circuit. The delay of our algorithm is $O(|\mathsf{var}(g)| \cdot d(g))$ where $d(g)$ is the depth of g, that is, the longest path from g to an input of the circuit.

To enumerate the tuples from an input gate, we simply return the only tuple labeling it and end the algorithm. For constant input gates, we return $\langle\rangle$ for $\top$-gates and nothing for $\bot$-gates. We have $O(1)$ delay which is $O(\max(1, |\mathsf{var}(g)|) \cdot d(g))$ since both are bounded by 1 for input gates.

To enumerate $\mathsf{rel}(g)$ for a $\uplus$-gate g with inputs $g_1, \dots, g_k$, we simply enumerate $\mathsf{rel}(g_1)$, then $\mathsf{rel}(g_2)$, ..., and $\mathsf{rel}(g_k)$. The delay of this algorithm is $\max_{i \le k} \delta_i$ where δ_i is the delay needed to enumerate $\mathsf{rel}(g_i)$ inductively. By induction, it is $O(|\mathsf{var}(g_i)| d(g_i))$. But we have $\mathsf{var}(g_i) = \mathsf{var}(g)$ and $d(g) = 1 + \max_{i \le k} d(g_i)$. Hence the delay is $O(\max_{i \le k} |\mathsf{var}(g)| \cdot d(g_i)) = O(|\mathsf{var}(g)| \cdot \max_{i \le k} d(g_i)) = O(|\mathsf{var}(g)| \cdot d(g))$.

To enumerate $\mathsf{rel}(g)$ for a $\times$-gate g with inputs $g_1, \dots, g_k$, we proceed by induction on k: for $k = 1$, we simply enumerate $\mathsf{rel}(g_1)$. Otherwise, assume we have an algorithm to enumerate $\mathsf{rel}(g_1 \times \dots \times g_k)$ with delay $O(W_k \cdot d(g))$ where $W_k = \sum_{i=1}^{k} |\mathsf{var}(g_i)|$. To enumerate $\mathsf{rel}(g_1 \times \dots \times g_{k+1})$, we run the enumeration of $\mathsf{rel}(g_1 \times \dots \times g_k)$. Each time a tuple

$\tau \in \mathsf{rel}(g_1 \times \cdots \times g_k)$ is output, we run the enumeration of $\mathsf{rel}(g_{k+1})$ with delay $O(d(g_{k+1}) \cdot |\mathsf{var}(g_{k+1})|)$. Each time it outputs $\sigma \in \mathsf{rel}(g_{k+1})$, we output $\tau \times \sigma$.

The delay between two tuples is the delay we need to construct τ plus the delay we need to construct σ. Hence it is at most $O(W_k \cdot d(g)) + O(|\mathsf{var}(g_{k+1})| \cdot d(g_{k+1})) \leq O(W_{k+1} \cdot d(g))$. Now, recall that we have assumed that every $\times$-gate has inputs that have at least one tuple and that are not $\top$-gates. Hence $|\mathsf{var}(g)| = \sum_{i=1}^{k+1} |\mathsf{var}(g_i)| = W_{k+1}$. Hence, our algorithm has the desired complexity.

In a nutshell, we have:

Theorem 4.53 ([OZ15]). *Let C be a $\{\times, \uplus\}$-circuit on variables X, let d be its depth. After a preprocessing $O(|C|)$, we can enumerate $\mathsf{rel}(C)$ with delay $O(|X| \cdot d)$.*

The preprocessing simply consists in propagating constant gates and simplifying $\times$-gates accordingly. In particular, we recover:

Theorem 4.54 ([BDG07]). *Given a free-connex acyclic conjunctive query Q, we can enumerate $[\![Q]\!]$ with delay $O(n^2)$ where $n = |\mathsf{var}(Q)|$ after preprocessing $\mathrm{poly}(|Q|) \cdot \|Q\|$.*

To do this, we can either use the construction of Yannakakis from Section 4.3.1 or exhaustive DPLL from Section 4.4 to build the circuit and then use the previously outlined enumeration algorithm. Both circuits have depth $O(|\mathsf{var}(Q)|)$ and are on variables $\mathsf{var}(Q)$, which are constant. Using exhaustive DPLL introduces logarithmic factors because of binarization that Yannakakis algorithm does not have but works on signed join queries. Brault-Baron already observed in his thesis [Bra13] that signed-acyclic queries could be enumerated with constant delay after linear preprocessing but our general result about the complexity of DPLL allows to generalize it to any join query with a preprocessing depending on $\mathsf{sfhow}(Q, \pi^{-1})$. We observe however that even if the preprocessing depends on the structure of the query during the construction of the circuit, the enumeration algorithm always has $O(n^2)$ delay.

4.7.2 Counting

Pichler and Skritek [PS13] were the first to observe that Yannakakis algorithm could be extended to counting (though their complexity analysis is not optimal and they have a quadratic dependence, mostly because they did not focus on this aspect of the work). In other words, they show that given an acyclic join query Q, we can compute $|[\![Q]\!]|$ in time $\mathrm{poly}(|Q|) \cdot \|Q\|$. They observe that it generalizes to bounded fractional hypertree width join queries. The fact that one can compute $|[\![Q]\!]|$ for signed-acyclic queries was initially proven in Brault-Baron's thesis [Bra13] but using an inclusion-exclusion approach that makes it exponential in $|Q|$. We proposed a polynomial time algorithm for combined complexity for β-acyclic instances in [BCM15] using an approach based on eliminating variables from the query. It could be adapted for signed-acyclic with minor modifications but we think that the circuit approach is more powerful.

Indeed, given a $\{\times, \uplus\}$-circuit C, and by interpreting every $\uplus$-gate as an arithmetic $+$, every $\times$-gates as an arithmetic $\times$ and input gates as constants 1 (but for $\bot$-gates which should be interpreted as 0), we end up with an arithmetic circuit whose value is $|\mathsf{rel}(C)|$. Every

intermediate value needed to evaluate the circuit will be bounded by d^n where d is the size of the domain of C and n is the number of variables. We can represent each intermediate result with n registers containing value encoded over $\log d$ bits. Hence, in the RAM model, we can perform arithmetic operations on such integers in time $\mathrm{poly}(n)$. Hence, we have:

Theorem 4.55. *Given a $\{\times, \uplus\}$-circuit C, we can compute $|\mathsf{rel}(C)|$ with $O(|C|)$ arithmetic operations and in time $\mathrm{poly}(n) \cdot |C|$.*

Now, combined with compilation algorithms for /(signed) join queries, we easily get as a corollary:

Theorem 4.56 ([PS13])**.** *Given an acyclic join query Q, one can compute $|[\![Q]\!]|$ in time $\mathrm{poly}(|Q|) \cdot \|Q\|$.*

Of course, we directly get generalizations to join queries of bounded fractional hypertree width and to signed join queries having bounded hyperorder width, or, if one is only interested in data complexity, of signed join queries having bounded fractional hyperorder width. We get similar upper bounds for conjunctive queries, by considering a free-connex order.

Theorem 4.57. *Given a conjunctive query Q and a free-connex order π on $\mathsf{var}(Q)$ such that $\mathsf{fhow}(Q) = k$, one can compute $|[\![Q]\!]|$ in time $\tilde{\mathrm{O}}_{\mathrm{poly}}(\|Q\|^k)$.*

We can generalize to signed queries, but as before, the combined complexity will be polynomial only if we use signed hyperorder width as a parameter, and not the fractional version.

Theorem 4.58. *Given a signed conjunctive query Q and a free-connex order π on $\mathsf{var}(Q)$ such that $\mathsf{show}(Q) = k$, one can compute $|[\![Q]\!]|$ in time $\mathrm{poly}(|Q|)^{O(k)} \cdot \tilde{\mathrm{O}}(\|Q\|^k)$.*

4.7.3 Direct access

Given a join query Q, we assume that its answers $[\![Q]\!]$ are sorted according to some order $\prec$ and, for each $1 \leq i \leq N = |[\![Q]\!]|$, we let $[\![Q]\!](i)$ be the ith answer of Q according to $\prec$. A *direct access task (for order $\prec$)* is the problem of returning, given i as input, the ith answer $[\![Q]\!](i)$ of Q if $i \leq N$, or an error if $i > N$.

Now, of course, if $[\![Q]\!]$ is materialized as an array of tuples, we can sort it according to $\prec$ and return $[\![Q]\!](i)$ in constant time. In this case, we can see the materialized array as a data structure to efficiently answer direct access tasks. This data structure is however expensive to compute: even in the case of acyclic join queries, the array will take at least N steps to be materialized, which can be of size up to $\|Q\|^n$ where $n = |\mathsf{var}(Q)|$, contrasting with the linear complexity of finding one answer or counting them.

The *direct access problem for Q and order $\prec$* is the problem of computing, given a join query Q, a data structure allowing to answer direct access tasks efficiently. We hence split the complexity of solving a direct access problem into two parameters:

1. The *preprocessing time*, which is the time needed to build the data structure on input Q.

2. The *access time*, which is the time needed to compute $[\![Q]\!](i)$ from the data structure for any i.

In the naive approach of materializing $[\![Q]\!]$, we have preprocessing time of at least $O(|[\![Q]\!]|)$ (possibly worse as some query may have few answers that are hard to find) and access time $O(1)$. This can however be improved in many cases. In [Bag+08], Bagan, Durand, Grandjean and Olive studied the problem for first-order logic. In [Car+20], Carmeli, Zeevi, Berkholz, Kimelfeld and Schweikardt, observed that for an acyclic join query Q (or free-connex acyclic conjunctive query), there exists an order on $[\![Q]\!]$ such that we can solve the direct access problem for Q with linear preprocessing and logarithmic access time. In this work, they use an order defined as a lexicographic order induced by some order on $\mathsf{var}(Q)$ as follows: given an order $\pi = (x_1, \ldots, x_n)$ on $\mathsf{var}(Q)$ and assuming $D := \mathsf{dom}(Q)$ is ordered by $<$, we define the *lexicographical order induced by* π, denoted by $\prec_\pi$, as the order on D^X defined as $\tau \prec_\pi \sigma$ if and only if there exists some $j \leq n$ such that $\tau(x_i) = \sigma(x_i)$ for every $i < j$ and $\tau(x_j) < \sigma(x_j)$. In [Car+20], π is (implicitly) extracted from the tree decomposition and the direct access problem is solved over $\prec_\pi$. Not every order π over $\mathsf{var}(Q)$ allows us to solve the direct access problem for Q and order $\prec_\pi$ with linear preprocessing and logarithmic access time. Indeed, the lexicographical orders allowing such complexity guarantees have been fully characterized by Carmeli, Tziavelis, Gatterbauer, Kimelfeld, Riedewald in [Car+23]. They exactly correspond to revrsals of α-elimination orders. This tractability is generalized in [BCM22] where it is shown that we can solve the direct access problem for Q with a lexicographical order induced by an order π on $\mathsf{var}(Q)$ with preprocessing $\mathrm{poly}(|Q|) \cdot \|Q\|^k$ and access time $\mathrm{poly}(|Q|) \cdot \log(\|Q\|)$ where $k = \mathsf{fhow}(Q, \pi)$. The paper shows a matching fine-grained lower bounds, establishing that under reasonable complexity assumptions, one cannot improve on this bound for preprocessing as long as we ask for polylogarithmic access time.

Interestingly, the above upper bounds match the upper bounds we have on $\{\times, \mathsf{dec}\}$-circuit computing $[\![Q]\!]$ and respecting order π. We can actually recover the upper bound from [BCM22] by using $\{\times, \mathsf{dec}\}$-circuit. It is indeed a direct corollary of the circuit construction from Theorem 4.34 and the following theorem, which we proved in [CI24]:

Theorem 4.59 ([CI24])**.** *Given a $\{\times, \mathsf{dec}\}$-circuit C on variables X and domain D, respecting order $\pi = (x_1, \ldots, x_n)$, we can solve the direct access problem for $\mathsf{rel}(C)$ and order $\prec_\pi$ with preprocessing $O(|C| \cdot \mathrm{polylog}\,|D|)$ and access time $O(n \log |D|)$ on a RAM machine.*

The proof of Theorem 4.59 can be found in [CI24] but we give a quick sketch of the main idea. The proof relies on the fact that we can precompute for every decision-gate g on variable x_i and value $d \in D$, the number of tuples τ in $\mathsf{rel}(g)$ such that $\tau(x_i) \leq d$. After this precomputation step, if we want to find the kth tuple τ of $\mathsf{rel}(C)$, we start by finding the value of $\tau(x_1)$ by finding the smallest value d such that $\mathsf{rel}(C)$ contains at least k tuples σ with $\sigma(x_1) \leq d$. In this case, we know that $\tau(x_1) = d$. We can find this value by doing a binary search on the precomputed values on a decision-gate testing x_1, hence in time $O(\log |D|)$. Once we know the value of τ on x_1, we can condition C on $\langle x_1/d \rangle$ and find $\tau(x_2)$ inductively in a similar manner.

4.8 Conclusion

In this section, we have seen how tools from knowledge compilation have interesting applications in the realm of databases, through the notion of relational circuits. We have revisited one of the most celebrated algorithm for joining relations as a compilation procedure and shown that one of the most natural algorithms for compiling CNF formulas into decision-DNNF circuits allows to prove new results. For example, it allows to efficiently compute the join of signed queries where traditional approaches are less efficient. While appealing in theory, one of the main practical downsides of this approach is that current database management systems are not designed to efficiently implement such branching algorithms.

In this chapter, we have hidden the technical details regarding the complexity analysis of Exhaustive DPLL on join queries which can be found in [CI24]. While the algorithm is easy to explain, proving its runtime guarantees is still challenging and we are still working on finding new ways of presenting these results in a simplified manner. Also, many questions remain unsolved concerning the complexity of signed join queries. First of all, while signed hyperorder width offers interesting insights into the complexity of Exhaustive DPLL, it is not clear how it can be computed efficiently. Even the complexity of computing $\mathsf{show}(H, \pi)$ given H and π as input is not known. These questions suggest at the fact that we have currently only scratched the surface of the algorithmic applications of Exhaustive DPLL in the setting of databases. An interesting direction is to understand how database constraints such as functional dependencies or degree constraints, in the same way as it has been done for enumeration [CK18] or in worst case optimal algorithms [KNS25].

Chapter 5

Applications to Optimization Theory

Given an assignment τ on variables X and a weighting function $w\colon X \times \{0,1\} \to \mathbb{R}_+$ on the literals over X, we can define the weight of τ as $\prod_{x \in X} w(x, \tau(x))$. When $w(x,0)+w(x,1) = 1$, this value corresponds to the probability of sampling τ when choosing the value of each variable independently, with $\Pr(x = 1) = w(x,1)$, when variables of X are independent variables over $\{0,1\}$ having probability $w(x,1)$ of being 1. From this, a natural question is to find a model of a Boolean function f that has maximum probability. It is not hard to see that a simple dynamic programming algorithm over a DNNF circuit can achieve it in linear time, by simply propagating at each gate the most probable model. For the maximization problem, because max is idempotent, that is $\max(a,a) = a$, we do not even need determinism. This observation has been made in [KVD17] where it is proven that algebraic model counting over idempotent semirings is tractable on DNNF circuits.

The tractability of finding the best model of a DNNF circuit can be generalized to finding the k-best models [Bou+22] or to ordered enumeration with application in logics [Ama+24]. In a recent contribution with Silvia Di Gregorio and Alberto Del Pia [CPG23], we used it to solve a problem known as Boolean Polynomial Optimization, where we try to maximize the value of a multilinear polynomial P over $\{0,1\}^n$. The idea, presented in Section 5.3, is to encode P as a CNF formula F_P with weights on its literals, representing the coefficient of P, so that there is a one-to-one correspondence between the value of P over a tuple τ and the weight of τ in F_P. More precisely, we see a multilinear polynomial $P = \sum_{m \in M} \alpha_m \prod_{i \in m} x_i$ as a Boolean function f_P on variables $X \cup M$ such that τ is a model of f_P if and only if $\tau(m) = \prod_{i \in m} \tau(x_i)$ and encode it as a CNF formula. We show that this transformation preserves some structure of the polynomial in how variables and monomials interact that we can use to compute a small DNNF circuit for F_P. Once the circuit C is computed, we can then find the optimal value for P by solving the maximization problem directly on C. We show that this approach allows us to recover many known results concerning the tractability of BPO over structured polynomials, such as polynomials whose underlying hypergraph is β-acyclic or has bounded treewidth. It also allows us to find new tractable classes of instances for BPO and to solve even more complex problems such as BPO with constraints by modifying directly the DNNF

circuit.

This connection actually runs deeper than a simple reduction from one optimization problem to another. Solving the BPO problem is not always what one is interested in, but the multilinear polynomial is only a part of a larger optimization problem. In this case, it is often interesting to be able to replace the polynomial by a system of linear constraints so that we can use more traditional methods for solving it, even when combined with other linear constraints. In the case of BPO, it boils down to the following: for a multilinear polynomial, we focus on its so-called multilinear set, which corresponds to the convex hull of the models of the previously described Boolean function f_P. A significant part of the literature has focused on finding so called *extended formulations* of the multilinear set, that is, a polyhedron $\mathcal{P}$ over $\mathbb{R}^Z$ with $Z \supseteq X \cup M$ that is described by a small number of linear constraints and such that, when projected onto $X \cup M$, coincides with the multilinear set of P. This line of work has focused on building polynomial size extended formulations for known tractable classes of BPO [DK17; DK18; DD23; DK23; DK24]. We show in Section 5.2.3 that all these results can be explained uniformly by giving extended formulations for the convex hull generated by the models of a DNNF circuit. This connection sheds new light on the problem of optimizing on DNNF circuits, by showing that this problem can be integrated into linear programming solvers. On the other hand, it also shows that knowledge compilers can be used to generate extended formulations, using techniques orthogonal to those traditionally used in optimization.

Organization of this chapter. We open this chapter in Section 5.1 with some necessary preliminaries on linear programming and extended formulations. We then explain how to get extended formulations of the convex hull of the models of a DNNF circuit in Section 5.2. We slowly work toward this result by first reviewing the optimization algorithm outlined earlier in Section 5.2.1, then by showing how it can be done on OBDD in Section 5.2.2 before finally generalizing to DNNF circuits in Section 5.2.3. We then explain how we can leverage tractable CNF formulas for knowledge compilation to solve the BPO problem in Section 5.3 and how the previous result can be used to also construct extended formulations in this context and go beyond the classical setting of BPO in Section 5.3.3.

Personal contributions covered in this chapter. The main contribution about extended formulations from DNNF circuits and their application for the Binary Polynomial Optimization problem are extracted from [CPG23], which is currently under review at Mathematical Programming (MAPR). The extended formulation for OBDDs presented in Section 5.2.2 is original but is inspired by discussions I had with Alberto Del Pia when I was visiting him at the University of Wisconsin-Madison in October 2023, when trying to understand how to construct extended formulation from DNNF circuits. The content of this chapter is connected to previous work from Nicolas Crosetti's PhD thesis [Cro23; Cap+24] which I co-supervised with Joachim Niehren, Jan Ramon and Sophie Tison. We only briefly mention this connection in the conclusion without entering into too many details. The main reason is that we are not yet able to make the connection fully formal yet though the approaches are very similar.

5.1 Definitions and notations

In this section, we give a few definitions from optimization theory needed to understand what follows. We will not go into much details and interested readers may find a detailed introduction to the topic in [CCZ14]. We start by defining linear constraints. For convenience with notations, in this chapter, we always assume that finite set of variables are ordered and we often write $X = \{x_1, \dots, x_n\}$.

Formally, given a set of variables $X = \{x_1, \dots, x_n\}$, a *linear constraint over* X is a set of real numbers $(a_1, \dots, a_n, b) \in \mathbb{R}^{n+1}$ together with an operator $\mathsf{op} \in \{\leq, \geq, =\}$. Informally, we will write it as $\sum_{i \leq n} a_i x_i \ \mathsf{op}\ b$. A *solution* of a linear constraint over X is a mapping $\tau \in \mathbb{R}^X$ such that $\sum_{i \leq n} a_i \tau(x_i) \ \mathsf{op}\ b$ holds. For example, $3x + 2y - z \leq 4$ is a linear constraint and $\langle x/1, y/2, z/3 \rangle$ is a solution of this constraint since $3 \times 1 + 2 \times 2 - 3 = 4 \leq 4$. It is not, however, a solution of the linear constraint $3x + 2y - z < 4$. A *system* $\mathcal{S}$ *of linear constraints over* X is a set of linear constraints over X. A solution of $\mathcal{S}$ is a mapping $\tau \in \mathbb{R}^X$ such that τ is a solution of every linear constraint of $\mathcal{S}$. We denote by $\mathcal{P}(\mathcal{S})$ the set of solutions of $\mathcal{S}$. We often write a system of linear constraints in the form of a matrix. Indeed, we can always rewrite a linear constraint in the form $\sum_{j=1}^{n} a_j x_j \geq b$, that is, $aX \geq b$ where a is a row vector $(a_1, \dots, a_n)$ and X is the column vector $(x_1, \dots, x_n)$. Hence, for a set of linear constraints, we will sometimes write $AX \geq b$ where A is a matrix and b is a column vector such that the ith row $(a_1, \dots, a_n)$ of A corresponds to the linear constraint $\sum_{j=1}^{n} a_{i,j} x_j \geq b_i$.

Observe that the set of solutions of $\mathcal{S}$ forms a *convex subset* of $\mathbb{R}^X$, that is, for every $\tau_1, \tau_2 \in \mathcal{P}(\mathcal{S})$ and $\lambda \in [0,1]$, we have $\lambda\tau_1 + (1-\lambda)\tau_2 \in \mathcal{S}$. Indeed, if τ_1, τ_2 are two solutions of the constraint $\sum_{i \leq n} a_i x_i \ \mathsf{op}\ b$, we have $\sum_{i \leq n} \lambda\tau_1(x_i) + (1-\lambda)\tau_2(x_i) \ \mathsf{op}\ \lambda b + (1-\lambda) b = b$. Hence $\lambda\tau_1 + (1-\lambda)\tau_2$ is also a solution of this constraint. A convex subset of $\mathbb{R}^X$ defined by a (finite) system of linear constraints is called a *polyhedron*. We say that a subset of $\mathbb{R}^X$ is *bounded* if it is contained $[-B, B]^X$ for some $B > 0$.

Some points in a convex set, called *extreme points*[1], are particularly interesting: given a convex set $\mathcal{S}$, an *extreme point* $\tau \in \mathcal{S}$ is a point of $\mathcal{S}$ that cannot be obtained as the linear combination of two other distinct points of $\mathcal{S}$. More formally, τ is an extreme point of $\mathcal{S}$ if and only if for every $\tau_1, \tau_2 \in \mathcal{S}$ such that there exists $\lambda \in [0,1]$ and $\tau = \lambda\tau_1 + (1-\lambda)\tau_2$, we have $\tau_1 = \tau_2 = \tau$.

We say that w is a *convex combination of vectors* $\tau_1, \dots, \tau_m \in \mathbb{R}^X$ if $w = \sum_{i=1}^{m} \lambda_i \tau_i$ for some $\lambda_i \in [0,1]$ such that $\sum_{i=1}^{m} \lambda_i = 1$. For a set $V \subseteq \mathbb{R}^X$, we define the *convex hull* $\mathrm{conv}(V)$ *of* V as the set of points obtained as a convex combination of a finite number of points of V.

In the setting of this chapter, we are interested in extreme points and convex hulls because of the following connection, known as the Minkowski-Weyl Theorem:

Theorem 5.1 (Minkowski-Weyl [CCZ14, Theorem 3.37])**.** *Let $\mathcal{P}$ be a convex subset of $\mathbb{R}^X$. The following are equivalent:*

- *$\mathcal{P}$ is a bounded polyhedron,*
- *$\mathcal{P}$ is the convex hull of finitely many points of $\mathbb{R}^X$,*

[1] Also sometimes called *vertices*, a term that we prefer to avoid in this document because of its interference with the notion of graph vertices.

- *$\mathcal{P}$ is the convex hull of its extreme points.*

We will mainly use the Minkowski-Weyl Theorem to prove the equality of two bounded polyhedra. Indeed, this theorem basically says that a bounded polyhedron can be identified by its extreme points. Hence, if we prove that two polyhedra have the same set of extreme points, then we can conclude that the polyhedra are equal.

A *linear program* is a pair $L = (f, \mathcal{S})$ where f is a linear form called the *objective function* and $\mathcal{S}$ is a system of linear constraints, where the goal is to maximize the value of f over $\mathcal{S}$. We often write it in the following form:

$$\begin{aligned} &\max cX \\ &\text{s.t.} \quad AX \geq b \end{aligned}$$

where c is the row vector representing f and $AX \geq b$ is the matrix representation of $\mathcal{S}$. A *feasible point of* L is simply a point of $\mathcal{P}(\mathcal{S})$ and a *solution of* L is a feasible point τ that maximizes f, and the *optimal value of* L is the value $f(\tau)$. If a linear program has a finite optimal value, then there is a solution which is also an extreme point of $\mathcal{P}(\mathcal{S})$. While it is true for every linear program, we explain why it is easy to see on bounded polyhedra using the Minkowski-Weyl theorem. Let $\tau \in \mathcal{P}(\mathcal{S})$ be an optimal solution of L. If τ is not an extreme point, then τ can be obtained as a convex combination of the extreme points $\tau_1, \ldots, \tau_m$ of $\mathcal{P}(\mathcal{S})$, that is $\tau = \sum_{i=1}^m \lambda_i \tau_i$. Let $\tau^* = \text{argmax}_{i=1}^m f(\tau_i)$. We have $f(\tau) = \sum_{i=1}^m \lambda_i f(\tau_i) \leq \sum_{i=1}^m \lambda_i f(\tau^*) = f(\tau^*)$. Hence τ^* is also a solution of L and is an extreme point. The problem of finding a solution (and hence the optimal value) of a given linear program L can be solved in polynomial time [Kha79; Kar84].

Example 5.2. Consider the following linear program on $\{x, y\}$:

$$\begin{aligned} &\max x + y \\ &\text{s.t.} \quad x - y \geq 0 \\ &\qquad x + y \leq 3 \\ &\qquad x - y \leq 2 \\ &\qquad x \geq 0 \end{aligned}$$

The polyhedron $\mathcal{P}$ defined by its feasible points is depicted in gray on Fig. 5.1. Its extreme points are $\langle x/0, y/0\rangle, \langle x/2, y/0\rangle, \langle x/2.5, y/0.5\rangle$ and $\langle x/1.5, y/1.5\rangle$. The optimal value of this linear program is 3: indeed, constraint $x + y \leq 3$ upper bounds the optimal value and it is attained, for example, at $\langle x/2.5, y/0.5\rangle$. Observe that the optimal value is actually attained everywhere on the edge of $\mathcal{P}$ following the yellow line corresponding to the constraint $x + y \geq 3$, but in particular, it is attained on the two extreme points on this line. If one considers objective function $(x + 2y)$, then in this case, the optimal value is attained only at $\langle x/1.5, y/1.5\rangle$ which is an extreme points of $\mathcal{P}$.

Another property of polyhedra is the *integrality*. A polyhedron $\mathcal{P} \subseteq \mathbb{R}^X$ is said to be *integral* if all its extreme points have integer coordinates, that is, if for every extreme point

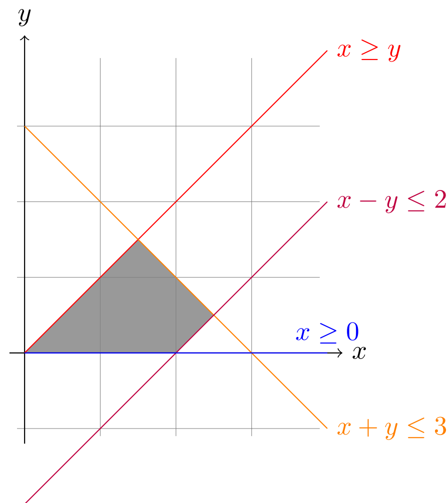


Figure 5.1: The polyhedron from Example 5.2.

$\tau \in \mathcal{P}$, we have $\tau \in \mathbb{Z}^X$. For example, the polyhedron from Example 5.2 is not integral, but the polyhedron defined by $0 \leq y \leq 2$ and $0 \leq x \leq 2$ is integral since its extreme points are $\langle x/0, y/0\rangle, \langle x/2, y/0\rangle, \langle x/2, y/2\rangle, \langle x/0, y/2\rangle$.

Integrality is useful when one is trying to solve a linear program with the additional constraint that the solution must have integer coordinates. This problem is called *integer linear programming*. In this case, the problem becomes NP-complete and one needs to resort to heuristic methods to solve it, in a similar way as SAT. That said, some integer linear programs can be solved in polynomial time. When the polyhedron defined by their linear constraints is integral, then we know that it has an integral optimal solution because there is an optimal solution which is also an extreme point, and which is integral by the integrality of the underlying polyhedron.

There have been several sufficient conditions for integrality of polyhedron described in the literature. One of the best-known is the notion of total unimodularity. A matrix A is said to be *totally unimodular* if and only if every square submatrix B of A has a determinant in $\{-1, 0, 1\}$. We have:

Theorem 5.3 ([HK56]). *If A is totally unimodular and $b \in \mathbb{Z}^X$, then the polyhedron induced by the system of linear constraints $AX \geq b$ is integral.*

As we shall see, total unimodularity is only a sufficient condition for integrality, and sometimes, it is not sufficient. We will later need a more involved sufficient condition known as *total dual integrality*. We say that a rational system of linear inequalities $AX \geq b, X \geq 0$ is *totally*

dual integral, abbreviated *TDI* if b is integral, the minimum in the LP-duality formulation:

$$\min Z^{\mathsf{T}} b$$
$$ZA \geq c$$
$$Z \geq 0$$

has an integral optimum solution Z^* for each integral vector c for which the minimum is finite. Total dual integrality is a sufficient condition for integrality:

Theorem 5.4 ([EG77], see also [CCZ14, Theorem 4.26])**.** *If $\mathcal{S} = AX \geq b, X \geq 0$ is a totally dual integral system then the polyhedron of $\mathcal{S}$ is integral.*

Extended formulations. We conclude this chapter with some complexity considerations. Indeed, we can see a system of linear constraints as a way of representing a convex subset of $\mathbb{R}^X$. This representation is often called the *perfect formulation.* A perfect formulation may not be necessarely succinct. Now, if the goal is to solve a linear program over some polyhedron, then having a perfect formulation is not always necessary. Indeed, assume that we want to find the optimal value of a linear form $f(X)$ over a polyhedron $\mathcal{P} \subseteq \mathbb{R}^X$ and that we have a system of linear constraints $\mathcal{S}$ over variables $X \cup Y$, that is, $\mathcal{S}$ is of the form:

$$A \begin{bmatrix} X \\ Y \end{bmatrix} \geq b$$

Moreover, assume that $\mathcal{P} = \{x \in \mathbb{R}^X \mid \exists y \in \mathbb{R}^Y, (x, y) \in \mathcal{P}(\mathcal{S})\}$, that is, $\mathcal{P}$ is the projection of $\mathcal{P}(\mathcal{S})$ over the X-variables. In this case, we say that $\mathcal{S}$ is an *extended formulation of* $\mathcal{P}$.

Extended formulations are useful because they may allow to reduce the number of constraints when solving an optimization problem. Indeed, if one can find an extended formulation $\mathcal{S}$ of $\mathcal{P}$ that is more succinct than a perfect formulation, then we can solve linear programs directly over the extended formulation. Indeed, for a linear form f over variables X, the optimal value of f over $\mathcal{P}$ can be found by solving the following linear program:

$$\max f(X)$$
$$A \begin{bmatrix} X \\ Y \end{bmatrix} \geq b.$$

A perfect formulation may be exponentially larger than a corresponding extended one. A notorious example is the case for a regular polygon over $\mathbb{R}^2$ with n sides. A perfect formulation for it would require at least n inequalities while an extended formulation with $O(\log n)$ inequalities is known [BN01], see [CCZ10] for a survey and examples of polyhedra having smaller extended formulations than their perfect formulation.

Interestingly, extended formulation can sometimes be extracted from algorithms. Indeed, there is a known connection between the fact that an optimization problem can be solved via a dynamic programming algorithm in polynomial time and the existence of a polynomial sized extended formulation. This has been formalized in [MRC90]. In this chapter, we recover a similar relation between the existence of a DNNF circuit computing a Boolean function and the existence of a small extended formulation expressing the convex hull of the models of the Boolean function itself, when seen as points with $\{0, 1\}$ coordinates.

5.2 Optimization on DNNF circuits

In this section, we study very general relationships between DNNF circuits and optimization problems, that we later use to recover known results in optimization theory in a unified (and generalized) way.

5.2.1 Maximizing weighted Boolean Functions

We start by studying an innocent-looking problem with a simple DNNF-based solution. We will show that this problem has deeper connections to optimization theory than expected in the next sections. We are interested in finding the optimal model of a Boolean function given a mapping from its models to $\mathbb{Q}$. To do so, we consider *literal weight functions*. A *literal weight function* w on variables X is a mapping $w : X \times \{0,1\} \to \mathbb{Q}$. Alternatively, it can be seen as a mapping from literals over X to $\mathbb{Q}$ and we will sometimes use notation $w(\neg x)$ for $w(x,0)$ and $w(x)$ for $w(x,1)$. A literal weight function over X naturally induces a weight on 2^X: for $\tau \in 2^X$, we let $w(\tau) = \sum_{x\in X} w(x,\tau(x))$. Now, given a Boolean function f on variables X, we let $w(f) = \max_{\tau\in f} w(\tau)$. That is

$$w(f) = \max_{\tau\in f} \sum_{x\in X} w(x,\tau(x)),$$

where we define $w(f) = -\infty$ when f has no model. The *Boolean Maximization Problem* is defined as the problem of computing $w(f)$ given a Boolean function f and a literal weight function w as input. If f is given as a CNF formula, there is a straightforward reduction to SAT, and hence the problem is NP-hard. Now, if f is given as a DNNF circuit C, then there is a straightforward dynamic programming algorithm computing the optimal value and optimal model by doing $O(|C|)$ comparisons in $\mathbb{Q}$. The Boolean Maximization Problem can be seen as a form of model counting over the $(\max,+)$-semiring, and such problems over DNNF circuits have been studied explicitly in [KVD17] and in [Bou+22]. We reproduce a proof here to be self contained. In the next statement, the *size* of w, denoted by $\|w\|$, is defined as $\sum_{x\in X}\|w(x)\|$ where the size of a rational number is defined as the number of bits needed to encode it.

Theorem 5.5. *Let (f,w) be a Boolean Maximization Problem and C a DNNF circuit such that $f = f_C$. We can find an optimal solution $\tau^* \in f$ for the weight function w in strongly polynomial time, i.e., with $O(|C|)$ arithmetic operations on numbers of size* $\mathrm{poly}(\|w\|)$.

Proof (sketch). The idea is to compute inductively, for each gate v of C, an optimal solution τ_v^* of the Boolean Maximization Problem problem (f_v, w), that is, a solution $\tau_v^* \in f_v$ such that $w(\tau_v^*)$ is maximal. We recall that $w(\tau_v^*)$ is equal to $\sum_{x\in\mathsf{var}(v)} w(x,\tau_v^*(x))$. If v is a leaf labeled by literal ℓ, then f_v contains exactly one assignment, which is hence optimal by definition.

Now let v be an internal node of C and let $v_1,\dots,v_k$ be its inputs. Assume that for each v_i, an optimal solution $\tau_{v_i}^*$ of (f_{v_i}, w) has been precomputed. If v is a $\wedge$-node, then we have that $f_v = \prod_{i=1}^k f_{v_i}$, by the fact that the circuit is decomposable. It can be easily verified that $\tau_v^* := \prod_{i=1}^k \tau_{v_i}^*$ is an optimal solution of f_v which proves the induction step in this case. Otherwise, v must be a $\vee$-node and in this case, $f_v = \bigcup_{i=1}^k f_{v_i} \times 2^{\mathsf{var}(v)\setminus\mathsf{var}(v_i)}$.

An optimal solution σ_i^* for $2^{\mathsf{var}(v)\setminus\mathsf{var}(v_i)}$ can be obtained by taking $\sigma_i^*(x) = b \in \{0,1\}$ if $w(x,b) \geq w(x,1-b)$ for every $x \in \mathsf{var}(v) \setminus \mathsf{var}(v_i)$. In this case $\tau_{v_i}^* \times \sigma_i^*$ is an optimal solution of $f_{v_i} \times 2^{\mathsf{var}(v)\setminus\mathsf{var}(v_i)}$. Finally, observe that an optimal solution of f_v has to be an optimal solution for $f_{v_i} \times 2^{\mathsf{var}(v)\setminus\mathsf{var}(v_i)}$ for some i. Hence, $\tau_v^* = \mathrm{argmax}_{i=1}^k w(\tau_{v_i}^* \times \sigma_i^*)$ is an optimal solution of f_v.

At each gate, we have to find the maximum of k numbers, where k is the number of inputs. Hence, we have at most one comparison per edge of the circuit. Each number has bit-size $\mathrm{poly}(\|w\|)$. We have an algorithm that computes an optimal solution of $f = f_C$ with $O(|C|)$ arithmetic operations on numbers of bit-size at most $\mathrm{poly}(\|w\|)$. □

Convex hull of Boolean functions. We now revisit the Boolean Maximization Problem as a convex optimization problem. To do so, we see a Boolean function f over variables X as a set of points in $\mathbb{R}^X$, that is, each model $\tau \in f$ is seen as a point in $\{0,1\}^X$. This allows us to naturally define the *convex hull of* f, denoted by $\mathrm{conv}(f)$, as the set of points of $\mathbb{R}^X$ that are convex combinations of models of f seen as $\mathbb{R}^X$ points. More formally, $\mathrm{conv}(f)$ is the following convex subset of $\mathbb{R}^X$:

$$\{\sum_{\tau \in f} \lambda_\tau \tau \mid \lambda_\tau \in [0,1], \sum_{\tau \in f} \lambda_\tau = 1\}.$$

By definition, it is clear that $\mathrm{conv}(f)$ is a convex subset of $\mathbb{R}^X$. Moreover, the extreme points of $\mathrm{conv}(f)$ are exactly the points $\tau \in f$: indeed, every other point in $\mathrm{conv}(f)$ are, by definition, obtained by a convex combination of points from f, and hence, are not extreme. Moreover, given $\tau \in f$, we can show it is an extreme point as follows: assume $\tau = \lambda\tau_1 + (1-\lambda)\tau_2$ and assume $\tau_1 \neq \tau_2$. Let x be such that $\tau_1(x) \neq \tau_2(x)$. Without loss of generality, we assume $\tau_1(x) > \tau_2(x)$. We hence have $0 \leq \tau_2(x) < \tau(x) < \tau_1(x) \leq 1$ which is impossible since $\tau(x) \in \{0,1\}$.

Now, given a literal weight function w over X, we have a natural mapping to the following linear form: $\phi_w(y_1, \dots, y_n) := \sum_{i=1}^n w(x_i) \cdot y_i + w(\neg x_i) \cdot (1 - y_i)$. It is straightforward to see that $\phi_w(\tau) = w(\tau)$ for every $\tau \in 2^X$. We can then reframe the Boolean Maximization Problem as the following convex optimization problem:

$$\begin{aligned} &\max \ \phi_w(\tau) \\ &\text{s.t.} \quad \tau \in \mathrm{conv}(f) \end{aligned} \tag{5.1}$$

It may seem that Eq. (5.1) may have an optimal value greater than the corresponding Boolean Maximization Problem with input f and w since non-integral points are allowed in the solution space. However, the objective function of Eq. (5.1) is linear and $\mathrm{conv}(f)$ is convex and bounded, hence we know that ϕ_w has a maximum reached at some extreme point of $\mathrm{conv}(f)$ as explained in Section 5.1. Since we just proved that the extreme points of $\mathrm{conv}(f)$ are exactly the points $\tau \in f$, we know that the optimal value of Eq. (5.1) is reached at some τ for $\tau \in f$. In other words, the optimal value of Eq. (5.1) is $w(f)$.

While Eq. (5.1) is a reformulation of the Boolean Maximization Problem as a convex optimization problem, it does not provide a concrete way of solving it since $\mathrm{conv}(f)$ is currently

described by the set of models of f. In particular, if one wants to pass Eq. (5.1) to a solver, one needs to enumerate every model of f first in order to describe conv(f), which defeats the purpose of using a solver at all, since one could simply keep track of the optimal model while enumerating all of them. To make Eq. (5.1) usable algorithmically, one needs to find a more compact way of representing conv(f). One possible way of doing this is via writing extended formulations of conv(f) as hinted in the preliminaries.

In the next section, we show that circuits from knowledge compilation can be understood as a way to build an extended formulation of the convex hull of the Boolean function they compute.

5.2.2 Convex hull of OBDD circuits

As a warm-up illustrating our approach, we start by showing that the convex hull of Boolean functions computed by OBDDs have small extended formulations. We show that it relates to the notion of flow in graphs. Extended formulations have historically been known to follow from dynamic programming algorithms. In a way, our result is similar since OBDD circuits can be seen as some trace of a dynamic programming algorithm. Actually, Martin, Rardin and Campbell gave a very general framework in [MRC90] that could be used directly on OBDD but we prefer to present a direct proof in this section to build intuition on what is happening in Section 5.2.3.

Consider an OBDD C over variables $X = \{x_1, \dots, x_n\}$. To simplify the presentation, we assume that C is complete (that is, every variable from X is tested from the source to a sink), and that C has exactly one source and one sink, denoted respectively by s and t, with t labeled by 1. The latter can be enforce by merging every 1-sink into one and removing every 0-sink and every edge going into it. In this normalized OBDD, there is a one-to-one correspondence between paths from s to t and the models of C.

Now remember that we want to express conv(f_C), the convex hull of the Boolean function computed by C, succinctly. We start by writing a linear program whose integral solutions are in one-to-one correspondence with the models of f_C. To do so, we introduce a variable y_e for each edge e of C. We want to encode that if we consider the edges e such that $y_e = 1$ in a solution, then they form a source-to-sink path. To do so, we want to express that at least one edge going out of s is picked and that whenever an edge is picked and it enters a non-sink node, then one edge going out of g is picked too. This leads to the following system of linear constraints $\mathcal{S}_y(C)$ defined over y's variables as follows:

$$\begin{aligned}
\sum_{e \in \mathsf{out}_\mathsf{E}(s)} y_e &= 1 && \\
\sum_{e \in \mathsf{in}_\mathsf{E}(g)} y_e &= \sum_{e \in \mathsf{out}_\mathsf{E}(g)} y_e && \text{for every non-sink node } g \\
y_e &\geq 0 && \text{for every edge } e.
\end{aligned}$$

where $\mathsf{in}_\mathsf{E}(g)$ is the set of edges entering g (that is, edges of the form $h \to g$ for some h) and $\mathsf{out}_\mathsf{E}(g)$ is the set of edges going out of g (that is, edges of the form $g \to h$). We let $\mathcal{P}_y(C)$ be

the polyhedron defined by the solutions of $\mathcal{S}_y(C)$, that is,

$$\mathcal{P}_y(C) = \{\mathbf{y} \in \mathbb{R}^{\mathsf{edges}(C)} \mid y \text{ satisfies } \mathcal{S}_y(C)\}.$$

Now, it is easy to see that every integral point $\mathbf{y} \in \mathcal{P}_y(C)$ verifies that $\mathbf{y}(e) \in \{0, 1\}$, by induction on the distance between e and the source s. Hence, it gives that the set of e such that $\mathbf{y}(e) = 1$ forms a path from s to t in C. Similarly, every source to sink path P in C gives an integral point $\mathbf{y}_P$ of $\mathcal{P}_y(C)$ by defining $\mathbf{y}_P(e) = 1$ if and only if e is in P.

Hence, every integral point of $\mathcal{P}_y(C)$ uniquely corresponds to a model of f_C and every model of f_C can be mapped to a point of $\mathcal{P}_y(C)$. To recover the model associated to an integral point of $\mathcal{P}_y(C)$, we consider the system of linear constraints $\mathcal{S}_{x,y}(C)$ obtained by adding the following constraint over variables $x_1, \dots, x_n$:

$$x_i - \sum_{e \in \mathsf{edges}(C,x)} y_e = 0 \tag{5.2}$$

where $\mathsf{edges}(C, x)$ are the set of edges of C of the form $e = (u, v)$ where u is a decision-gate on x and e is labeled by 1. We let $\mathcal{P}_{x,y}(C)$ be the polyhedron defined as the solutions of $\mathcal{S}_{x,y}(C)$. From what precedes, an integral point $(\mathbf{x}, \mathbf{y}) \in \mathcal{P}_{x,y}(C)$ defines a unique source to sink path $P_{\mathbf{y}}$ in C. Moreover, we claim in this case that $\mathbf{x} \in 2^X$ is an assignment compatible with $P_{\mathbf{y}}$. Indeed, let $e = (u, v)$ be an edge on $P_{\mathbf{y}}$. In this case, u is a decision-gate on some variable x_i and it is the only one on $P_{\mathbf{y}}$. If e is labeled by 1, then we must have $\mathbf{x}(x_i) = 1$ by Eq. (5.2). If e is labeled by 0, then since u is the only decision gate on x_i in $P_{\mathbf{y}}$, by Eq. (5.2) again, $\mathbf{x}(x_i) = 0$. Since C is complete, every variable is tested on $P_{\mathbf{y}}$, and $\mathbf{x}$ is then a well-defined assignment of 2^X, compatible with $P_{\mathbf{y}}$.

Hence, we have shown the following:

Theorem 5.6. *For every OBDD C, the models of f_C are exactly $\mathcal{P}_{x,y}(C) \cap \mathbb{Z}^{X \cup \mathsf{edges}(C)}$ projected onto X.*

Consider the polyhedron $\mathcal{P}_x(C) = \{\mathbf{x} \mid \exists \mathbf{y}.(\mathbf{x}, \mathbf{y}) \in \mathcal{P}_{x,y}(C)\}$, that is, $\mathcal{P}_x$ is the projection of $\mathcal{P}_{x,y}(C)$ onto X. Another way of interpreting Theorem 5.6 is to observe that since $\mathcal{P}_x(C)$ contains at least every point corresponding to the models of f_C and since it is convex, we have $\text{conv}(f_C) \subseteq \mathcal{P}_x(C)$.

We claim that it actually holds that $\text{conv}(f_C) = \mathcal{P}_x(C)$. One could show the inclusion $\mathcal{P}_x(C) \subseteq \text{conv}(f_C)$ by explicitly reconstructing every point of $\mathcal{P}_x(C)$ as a convex combination of models of f_C.

This is actually possible but a bit tedious: each point $(\mathbf{x}, \mathbf{y}) \in \mathcal{P}_{x,y}(C)$ induces several paths from s to t in C that are weighted according to $\mathbf{y}$. Each path P corresponds to a model τ_P of f_C and one can use the values of $\mathbf{y}$ on its edges to assign a weight λ_P to each path such that $\mathbf{x} = \sum_P \lambda_P \tau_P$, where λ_P is, intuitively, the probability of picking P by growing it from the source to the sink and locally deciding the next edge based on the values of $\mathbf{y}(e)$. While this would give an interesting insight on the problem, this approach is a bit tedious and heavy.

There is another interesting way of proving the same connection by observing that $\mathcal{S}_y(C)$ is actually a very well studied linear program. Indeed, the constraints in $\mathcal{S}_y(C)$ can be seen as

flow conservation constraints [FF56]. The resulting polyhedron $\mathcal{P}_y(C)$ is known to be integral. In other words, every extreme point of $\mathcal{P}_y(C)$ has integer values. This can be proven using the fact that the matrix associated to the system $\mathcal{S}_y(C)$ can be shown to be totally unimodular, that is, every square submatrix has determinant in $\{-1, 0, 1\}$, a well-known necessary condition for a linear system to define an integral polyhedron [HK56]. Hence, we have:

Lemma 5.7. *For every OBDD C, $\mathcal{P}_y(C)$ is integral.*

Using Lemma 5.7, we can show that $\mathcal{P}_{x,y}(C)$ is integral:

Lemma 5.8. *For every OBDD C, $\mathcal{P}_{x,y}(C)$ is integral.*

Proof. Let $(\mathbf{x}, \mathbf{y})$ be a vertex of $\mathcal{P}_{x,y}(C)$. We need to show that $(\mathbf{x}, \mathbf{y})$ is integral. There exist $|X| + |\mathsf{edges}(C)|$ linearly independent constraints among $\mathcal{S}_{x,y}(C)$ such that $(\mathbf{x}, \mathbf{y})$ is the unique vector that satisfies these $|X| + |\mathsf{edges}(C)|$ constraints at equality. Note that constraints (5.2) are the only ones containing variables x, and each such constraint contains exactly one x variable with nonzero coefficient. Thus, $|X|$ of the linearly independent constraints defining $(\mathbf{x}, \mathbf{y})$ must be (5.2). The remaining $|\mathsf{edges}(C)|$ linearly independent constraints defining $(\mathbf{x}, \mathbf{y})$ must be in $\mathcal{S}_y(C)$, and these constraints only involve y variables. Hence, $\mathbf{y}$ is a vertex of $\mathcal{P}_y(C)$. By assumption Lemma 5.7, $\mathcal{P}_y(C)$ is integral, so $\mathbf{y}$ is integral. Since $\mathbf{x}$ can be obtained from $\mathbf{y}$ using equations (5.2), we obtain that $\mathbf{x}$ is integral too. □

By Lemma 5.8, the extreme points of $\mathcal{P}_{x,y}(C)$ are all integral. In particular, the extreme points of $\mathcal{P}_x(C)$, which are all projections of extreme points of $\mathcal{P}_{x,y}(C)$, are integral. Now, since $\mathcal{P}_x(C)$ is bounded, it is the convex hull of its extreme points by Minkowski-Weyl Theorem [CCZ14, Corollary 3.14]. Combined with Theorem 5.6, we conclude that every extreme point of $\mathcal{P}_x(C)$ correspond to a model of f_C. Hence, $\mathcal{P}_x(C)$ is the convex hull of its extreme points, that is, of some models of f_C. In other words, $\mathcal{P}_x(C) \subseteq \mathrm{conv}(f_C)$, hence $\mathcal{P}_x(C) = \mathrm{conv}(f_C)$ because the reversed inclusion directly follows from Theorem 5.6.

We just have established the following:

Theorem 5.9. *Given a complete OBDD C on variables X, one can construct in time $O(|C|)$ a system $\mathcal{S}_{x,y}(C)$ with $|C| + |X|$ variables and $O(|C|)$ constraints such that* $\mathrm{conv}(f_C) = \{\mathbf{x} \mid \exists \mathbf{y}, (\mathbf{x}, \mathbf{y})$ *satisfies* $\mathcal{S}_{x,y}(C)\}$.

In other words, Theorem 5.9 states that we can efficiently construct an extended formulation of $\mathrm{conv}(f_C)$ of size $O(|C|)$.

Observe that we never really used the ordered property of OBDDs nor their determinism in what precedes. Hence, the same proof would also work for non-deterministic FBDDs. In the next section, we actually give an even more general result by writing linear size extended formulations capturing the convex hull of DNNF circuits.

5.2.3 Convex hull of DNNF circuits

We now turn our attention to the case of Boolean functions computed by DNNF circuits. The goal of this section is to show a generalization of Theorem 5.9 to DNNF circuits. As

mentioned earlier, [MRC90] contains a very general framework to derive extended formulations from dynamic programming algorithms. It builds on a very generic formalization of dynamic programming algorithms and we tweak it to cover the case of DNNF circuits. That said, this approach does not give much insight on what is going on with DNNF circuits and the translation to their framework would not be much shorter than deriving the proof from scratch. It turns out that the other direction is also possible: we could go from the framework of [MRC90] and reformulate it as DNNF circuits. We think that our theorem, formulated directly on DNNF circuits, allows for more direct connections with existing results, see Section 5.3 for an illustration.

We start by stating our main theorem before dedicating this section to proving it:

Theorem 5.10. *Given a smooth DNNF circuit C on variables X, one can construct in time $O(|C|)$ a system $\mathcal{S}_{x,y}(C)$ with $|C|+|X|$ variables and $O(|C|)$ constraints such that* $\mathrm{conv}(f_C) = \{\mathbf{x} \mid \exists \mathbf{y}, (\mathbf{x}, \mathbf{y})$ *satisfies* $\mathcal{S}_{x,y}(C)\}$.

The proof of Theorem 5.10 is very similar to the proof of Theorem 5.9, but we have to replace paths in OBDD with their counterparts in DNNF circuits: *certifcates*, which are syntactic witnesses of the models of a DNNF circuit. We will start by writing a simple linear program $\mathcal{S}_y(C)$ over variables y_e for every $e \in \mathsf{edges}(C)$ whose integral solutions are exactly the certificates of the circuit C. Then, we will augment it as $\mathcal{S}_{x,y}(C)$ where the model of C defined by the certificate described in the y variables is recovered in the x variables. Finally, we will show that $\mathcal{S}_{x,y}(C)$ is integral and this will allow us to conclude as before.

In this section, we therefore fix DNNF circuit C on variables $X = \{x_1, \dots, x_n\}$ and normalize it as follows, w.l.o.g: we first assume that C does not contain any 0-labeled input, which we ensure by propagating and simplifying it in the circuit. Then, we assume C is smooth since every DNNF circuit C' can be smoothed into a DNNF circuit of size at most $n|C|$. Remember that a DNNF circuit is smooth whenever for every $\vee$-gate g and g' a child of g, we have $\mathsf{var}(g) = \mathsf{var}(g')$. We also assume that the output gate of C, denoted by o, is a $\vee$-gate with no outgoing edges and that there is a path from every gate g of C to o. This can be ensured by iteratively removing every gate that does not satisfy this property. Finally, w.l.o.g, we assume that for every literal ℓ, there is at most one input of C labeled by ℓ. This can be easily ensured by merging all gates labeled by ℓ. We denote by $\mathsf{edges}(C, \ell)$ the edges going out of the only input labeled by ℓ.

Certificates. We start by defining the set of certificates of C, which may be seen as a generalization of the notion of certificates for TDDs from Chapter 2. Intuitively, in a DNNF circuit, a model can be found by starting from the output of C and then repeating the following process until input gates are reached: if the current gate g is a $\vee$-gate, we pick one input of g and proceed with it. If g is a $\wedge$-gate, we pick every input of g and proceed with them. This process constructs a subcircuit whose leaves are literals with disjoint variables. If the circuit is smooth and without 0-input, we have exactly one literal per variable, and this represents one model of C. This is what we call a certificate. More formally, a *certificate of* C is a subcircuit T of C such that the output gate of C is in T and such that:

- for every $\vee$-gate g of T, exactly one input g' of g is in T with edge (g', g).

- for every $\wedge$-gate g of T, every input g' of g is in T with edge (g', g).
- for every g in T that is not the output of C, g is the input of at least one other gate in T.

A certificate is depicted in red with double edges in Figure 5.2.

We denote by $\mathrm{cert}(C)$ the set of certificates of C. If C is smooth[2], each $T \in \mathrm{cert}(C)$ represents a unique model of C as follows (see [Cap16, Section 6.1.1] for details): if C is a smooth DNNF circuit, one can easily check that for every certificate T and every variable $x_i \in X$, there is exactly one input gate labeled with a literal involving variable x_i (that is, either there is exactly one input labeled x_i or exactly one input labeled $\neg x_i$). For a certificate T, we denote by σ_T the assignment defined as $\sigma_T(x_i) = 1$ if an input labeled by x_i is in T and $\sigma_T(x_i) = 0$ if an input labeled by $\neg x_i$ is in T. We have that:

Lemma 5.11. *For every smooth DNNF circuit C, we have $\{\sigma_T \mid T \in \mathrm{cert}(C)\}$ is the set of models of C.*

The certificate from Figure 5.2 witnesses that the assignment $x_1 = 1, x_2 = 1$ is a satisfying assignment of the circuit. Observe that if C is not deterministic, we can have more than one certificate for the same model, in the same way as a model of a non-deterministic OBDD may be accepted by more than one path.

As for OBDD, we now encode the notion of certificate into a system of linear inequality constraints $\mathcal{S}_y(C)$ using variables y_e for each $e \in \mathsf{edges}(C)$. We will use the following notations: for a gate g of C, we denote by $\mathsf{in}_\mathsf{E}(g)$ the set of edges going in g and by $\mathsf{out}_\mathsf{E}(g)$ the set of edges going out of g. We similarly denote by $\in [g]$ the set of input gates of g, that is, the gates h such that (h, g) is an edge of C and by $\mathsf{out}(g)$ the set of outputs of g, that is, the gates h such that (g, h) is an edge of C.

From now on, we assume that the output o of C is a $\vee$-gate. This can be ensured by adding a dummy $\vee$-gate connected only to the output of C and picking this new gate as the output. It obviously does not change the models of C. While this normalization is not completely necessary, it removes one edge case in the definition of the linear program and hence simplifies the presentation. We define $\mathcal{S}_y(C)$ as follows:

$$\sum_{e \in \mathsf{in}_\mathsf{E}(o)} y_e = 1, \qquad \text{where } o \text{ is the output of } C \tag{5.3a}$$

$$\sum_{e \in \mathsf{in}_\mathsf{E}(g)} y_e - \sum_{f \in \mathsf{out}_\mathsf{E}(g)} y_f = 0 \qquad \text{for every } \vee\text{-gate } g \neq o, \tag{5.3b}$$

$$y_e - \sum_{f \in \mathsf{out}_\mathsf{E}(g)} y_f = 0 \qquad \text{for every } \wedge\text{-gate } g \text{ and } e \in \mathsf{in}_\mathsf{E}(g) \tag{5.3c}$$

$$y_e \geq 0 \qquad \text{for every } e \in \mathsf{edges}(C). \tag{5.3d}$$

[2] Smoothness here is only needed to ensure that there is exactly one model accepted by the certificate. Without smoothness, some variables may be missing in the leaves of a certificate. In this case, a certificate T containing literals over variables $Y \subseteq X$ would represent every model of the form $\sigma \times 2^{X \setminus Y}$ for some $\sigma \in 2^Y$ which corresponds to the literals of T.

Figure 5.2: A smooth DNNF circuit C where $\mathcal{S}_y(C)$ is not totally unimodular. A certificate is depicted in red with double edges.

We then define $\mathcal{P}_y(C)$ as the polyhedron:

$$\mathcal{P}_y(C) := \{\mathbf{y} \in \mathbb{R}^{\mathsf{edges}(C)} : \mathbf{y} \text{ satisfies } \mathcal{S}_y(C)\}.$$

We start by giving an example of $\mathcal{P}_y(C)$:

Example 5.12. Figure 5.2 depicts a smooth DNNF circuit C. The system of linear constraints $\mathcal{S}_y(C)$ is hence defined as:

$$y_{12} + y_1 = 1 \tag{5.4a}$$
$$y_4 + y_2 = y_1 \tag{5.4b}$$
$$y_6 = y_{12} \tag{5.4c}$$
$$y_{13} = y_{12} \tag{5.4d}$$
$$y_3 = y_2 \tag{5.4e}$$
$$y_{10} = y_4 \tag{5.4f}$$
$$y_{11} = y_4 \tag{5.4g}$$
$$y_5 = y_3 \tag{5.4h}$$
$$y_7 = y_3 \tag{5.4i}$$
$$y_8 = y_{10} + y_5 \tag{5.4j}$$
$$y_9 = y_7 + y_6 \tag{5.4k}$$
$$0 \leq y. \tag{5.4l}$$

It turns out that every integral point $\mathbf{y}$ of $\mathcal{P}_y(C)$ corresponds to a certificate of C. For example, the certificate depicted in Fig. 5.2 corresponds to the point $\mathbf{y}$ such that $\mathbf{y}(y_1) = \mathbf{y}(y_2) = \mathbf{y}(y_3) = \mathbf{y}(y_5) = \mathbf{y}(y_7) = \mathbf{y}(y_8) = \mathbf{y}(y_9) = 1$ and $\mathbf{y}(e) = 0$ otherwise, that is, we explicitly pick the red edges and discard the others. This can easily be generalized as follows: given a certificate $T \in \mathsf{cert}(C)$, we let $\mathbf{y}_T \in 2^{\mathsf{edges}(C)}$ be defined as: $\mathbf{y}_T(e) = 1$ if e is an edge in T and $\mathbf{y}_T(e) = 0$ otherwise. We have:

Lemma 5.13. *For every certificate $T \in \mathrm{cert}(C)$, $\mathbf{y}_T \in \mathcal{P}_y(C)$.*

Proof. It directly follows from the definition of $\mathcal{P}_y(C)$. Since the output of C is a $\vee$-gate, there is exactly one edge connected to it in T, and this edge is mapped to 1 by $\mathbf{y}_T$. Hence Eq. (5.3a) is satisfied. Now, if g is a gate of C that is not in T, none of the edges connected to it are in T, hence for every such edge e, we have $\mathbf{y}_T(e) = 0$. In this case, Eq. (5.3b) or Eq. (5.3c) (depending on whether g is a $\vee$-gate or a $\wedge$-gate) are clearly satisfied. Finally if g is a $\vee$-gate of T, then by definition of certificates, exactly one ingoing and one outgoing edge of g are in T. Hence Eq. (5.3b) is satisfied since both sums evaluate to 1. If g is a $\wedge$-gate of T, then by definition of certificates, all its ingoing edges and exactly one outgoing edge of g are in T. Hence Eq. (5.3c) is satisfied since $\mathbf{y}(e) = 1$ for each ingoing edge e of g and the sum evaluate to 1. Eq. (5.3d) is also clearly satisfied, which concludes the proof. □

More interestingly, the converse of Lemma 5.13 also holds, in the following sense:

Lemma 5.14. *For every integral point $\mathbf{y}$ of $\mathcal{P}_y(C)$, there exists $T \in \mathrm{cert}(C)$ such that $\mathbf{y} = \mathbf{y}_T$.*

Proof. Consider an integral point $\mathbf{y} \in \mathcal{P}_y(C)$. First, observe that for every gate g and edge $e \in \mathsf{in}_\mathsf{E}(g)$, by (5.3b) if g is a $\vee$-gate or by (5.3c) if g is a $\wedge$-gate, we have:

$$\mathbf{y}(e) \leq \sum_{f \in \mathsf{out}_\mathsf{E}(g)} \mathbf{y}(f). \tag{5.5}$$

We start by proving that for every gate g, $\sum_{f \in \mathsf{out}_\mathsf{E}(g)} \mathbf{y}(f) \in \{0,1\}$, by induction on the depth of g, that is, the length of the longest path from g to the output o of C. If the depth of g is 1, it means that g is directly connected to o through edge $e = (g, o)$ and this edge is the only edge going out of g since otherwise, there would be a longer path from g to o. Now by (5.3a), $\mathbf{y}(e) \leq 1$ which implies $\sum_{f \in \mathsf{out}_\mathsf{E}(g)} \mathbf{y}(f) = \mathbf{y}(e) \in \{0,1\}$ since $\mathbf{y}(e)$ is an integer.

Now assume that the induction hypothesis holds for every gate of depth at most d and let g be a gate of depth $d+1$. Let $e = (g, g') \in \mathsf{out}_\mathsf{E}(g)$. Since $e \in \mathsf{in}_\mathsf{E}(g')$, we have by (5.5) that $\mathbf{y}(e) \leq \sum_{f \in \mathsf{out}_\mathsf{E}(g')} \mathbf{y}(f) \in \{0,1\}$ by induction, since g' has depth at most d. In other words, $\mathbf{y}(e) \in \{0,1\}$.

It remains to show that there is at most one edge e in $\mathsf{out}_\mathsf{E}(g)$ such that $\mathbf{y}(e) > 0$. Assume toward a contradiction that two such edges $e, e' \in \mathsf{out}_\mathsf{E}(g)$ exist, that is, $\mathbf{y}(e) = \mathbf{y}(e') = 1$. By (5.5) and by induction, if a gate g' with depth at most d has an incoming edge f with $\mathbf{y}(f) = 1$, then it has an outgoing edge f' with $\mathbf{y}(f') = 1$. Hence, from g, we can define two paths going toward the output o, one starting with e and the other with e', and both following only edges f with $\mathbf{y}(f) = 1$. Both paths have to meet at some gate g'', with two incoming edges $e_1 = (g_1, g'')$ and $e_2 = (g_2, g'')$ with $\mathbf{y}(e_1) = \mathbf{y}(e_2) = 1$. Now observe that g'' cannot be a $\wedge$-gate, otherwise both inputs of g'' would share a variable (through the subcircuit rooted in g), contradicting the decomposability assumption. Hence g'' is a $\vee$-gate of depth at most d. But by (5.3b), it would imply $\sum_{f \in \mathsf{out}_\mathsf{E}(g'')} \mathbf{y}(f) \geq \mathbf{y}(e_1) + \mathbf{y}(e_2) \geq 2$, which contradicts the induction hypothesis, if g'' is not the output of C. If $g'' = o$, it contradicts (5.3a). Hence g has at most one outgoing edge e with $y_e = 1$ which establishes the induction at depth $d+1$.

We have established that for every g, $\sum_{f \in \mathsf{out}_\mathsf{E}(g)} \mathbf{y}(f) \in \{0,1\}$. One direct consequence is that $\mathbf{y}(e) \in \{0,1\}$ for every edge e. Now, consider the subcircuit T of C defined as the

one induced by the edges e of C such that $\mathbf{y}(e) = 1$. We prove that T is a certificate of C. It is enough to check that every condition of certificates is respected by T. First, let g be a $\vee$-gate in T. By definition, it is in at least one edge e with $\mathbf{y}(e) = 1$. By (5.3b), $\sum_{f \in \mathsf{in}_\mathsf{E}(g)} \mathbf{y}(f) = \sum_{f \in \mathsf{out}_\mathsf{E}(g)} \mathbf{y}(f)$ and, by what precedes, the RHS of this equality must be equal to 1. In other words, there is exactly one ingoing edge e_1 in g and exactly one outgoing edge e_2 from g such that $\mathbf{y}(e_1) = \mathbf{y}(e_2) = 1$. This proves the first condition of certificate. Now, if g is a $\wedge$-gate, again, at most one edge e containing g is such that $\mathbf{y}(e) = 1$ by definition of T. From what precedes, we hence must have $\sum_{f \in \mathsf{out}_\mathsf{E}(g)} \mathbf{y}(f) = 1$. Hence, by (5.3c), every ingoing edge e of g is such that $\mathbf{y}(e) = 1$. It remains to show that the third condition holds. Assume toward a contradiction that there is a gate g such that there is an edge $e = (g', g)$ with $\mathbf{y}(e) = 1$ but such that for every outgoing edge $e' = (g, g'')$, $\mathbf{y}(e') = 0$. It directly contradicts (5.5). Hence the third condition of certificate is respected which concludes the proof. □

Since every integral point $\mathbf{y}$ of $\mathcal{P}_y(C)$ corresponds to a certificate T of C, we can reconstruct the corresponding model σ_T of C easily via the following linear constraints over variables X and (y_e):

$$x_i - \sum_{e \in \mathsf{out}_\mathsf{E}(g_i)} y_e = 0 \qquad \text{for every } x_i \in X \tag{5.6}$$

$$\tag{5.7}$$

where g_i is the only input gate of C labeled by literal x_i. We let $\mathcal{S}_{x,y}(C)$ be the linear program whose constraints are those from $\mathcal{S}_y(C)$ together with the constraints from (5.6). It is clear that for every certificate T of C, the previous constraint will force x_i to be set to $\sigma_T(x_i)$. Let $\mathcal{P}_{x,y}(C)$ be the polyhedron:

$$\mathcal{P}_{x,y}(C) = \{(\mathbf{x}, \mathbf{y}) \in \mathbb{R}^{X \cup \mathsf{edges}(C)} : (\mathbf{x}, \mathbf{y}) \text{ satisfies } \mathcal{S}_{x,y}(C)\}$$

and

$$\mathcal{P}_x(C) = \{\mathbf{x} \in \mathbb{R}^X : \exists \mathbf{y}.(\mathbf{x}, \mathbf{y}) \in \mathcal{P}_{x,y}(C)\}$$

its projection onto X. From Lemmas 5.13 and 5.14 and what precedes, we have shown:

Lemma 5.15. *The projection of $\mathcal{P}_{x,y}(C) \cap \mathbb{Z}^{X \cup \mathsf{edges}(C)}$ onto X coincides with f_C.*

Lemma 5.15 is an analogue of Theorem 5.6 for DNNF circuits. As before, another interpretation of it is that it proves $\mathrm{conv}(f_C) \subseteq \mathcal{P}_x(C)$.

Example 5.16. The full system of linear constraints $\mathcal{S}_{x,y}(C)$ for the circuit C depicted in Fig. 5.2 is defined as $\mathcal{S}_y(C)$ from Example 5.12, together with the following constraints:

$$x_1 - y_8 = 0 \tag{5.8a}$$

$$x_2 - y_9 = 0 \tag{5.8b}$$

Now considering the certificate from Fig. 5.2 and the solution $\mathbf{y}_T$ of $\mathcal{S}_y(C)$ associated to it, we see that the two new constraints force $x_1 = 1$ and $x_2 = 1$. Hence it gives a

corresponding solution $(\mathbf{x}_T, \mathbf{y}_T)$ of $\mathcal{S}_{x,y}(C)$ and we have $\mathbf{x}_T = \sigma_T$.

It remains to show that $\mathcal{P}_x(C) \subseteq \text{conv}(f_C)$. The problem is really analogous to what happens in OBDD. We could explicitly show how every point in $\mathcal{P}_x(C)$ can be obtained as a convex combination of models of f_C by seeing a point in $\mathcal{P}_{x,y}(C)$ as weighted certificates from which one can extract the convex combination of models. As for OBDD, this construction would be possible but tedious. Hence, we adopt a similar strategy as in the previous section. We show that $\mathcal{P}_{x,y}(C)$ is integral. This is enough for our purpose since in this case, every extreme point of $\mathcal{P}_x(C)$ will be a projection of an extreme point of $\mathcal{P}_{x,y}(C)$, that is, will correspond to a model of f_C by Lemma 5.15. Hence, it will imply $\mathcal{P}_x(C) \subseteq \text{conv}(f_C)$. As in Lemma 5.8, we observe that integrality for $\mathcal{P}_y(C)$ is enough:

Lemma 5.17. *If $\mathcal{P}_y(C)$ is integral, then so is $\mathcal{P}_{x,y}(C)$.*

Proof. Let $(\mathbf{x}, \mathbf{y})$ be an extreme point of $\mathcal{P}_{x,y}(C)$. We need to show that $(\mathbf{x}, \mathbf{y})$ is integral. There exist $|X| + |\mathsf{edges}(C)|$ linearly independent constraints among $\mathcal{S}_{x,y}(C)$ such that $(\mathbf{x}, \mathbf{y})$ is the unique vector that satisfies these $|X|+|\mathsf{edges}(C)|$ constraints at equality. Note that constraints (5.6) are the only ones containing variables x, and each such constraint contains exactly one x variable with nonzero coefficient. Thus, $|X|$ of the linearly independent constraints defining $(\mathbf{x}, \mathbf{y})$ must be (5.6). The remaining $|\mathsf{edges}(C)|$ linearly independent constraints defining $(\mathbf{x}, \mathbf{y})$ must be in $\mathcal{S}_y(C)$, and these constraints only involve y variables. Hence, $\mathbf{y}$ is a vertex of $\mathcal{P}_y(C)$. By assumption, $\mathcal{P}_y(C)$ is integral, so $\mathbf{y}$ is integral. Since $\mathbf{x}$ can be obtained from $\mathbf{y}$ using equations (5.6), we obtain that $\mathbf{x}$ is integral too. ☐

It remains to show that $\mathcal{P}_y(C)$ is an integral polyhedron. This proves slightly more difficult than in the case of OBDD. Indeed, one cannot use total unimodularity as we can construct smooth DNNF circuits where the matrix of $\mathcal{S}_y(C)$ is not totally unimodular:

Example 5.18. Consider again the example from Example 5.12. The matrix obtained from the system $\mathcal{S}_y(C)$ of this example by keeping the rows corresponding to constraints 5.4a,5.4b,5.4c,5.4f,5.4h,5.4i,5.4j and 5.4k and the columns corresponding to variables $y_1, y_3, y_4, y_5, y_6, y_7, y_{10}, y_{12}$ is given by:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & -1 \\ 0 & 0 & -1 & 0 & 0 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & -1 & 0 & 0 \end{pmatrix}$$

This is a square matrix with determinant equal to 2. Therefore, the constraint matrix of the system $\mathcal{S}_y(C)$ is not totally unimodular.

We now proceed to show that $\mathcal{S}_y(C)$ is totally dual integral, which by Theorem 5.4, implies that $\mathcal{P}_y(C)$ is integral. Interestingly, we will see that the dual of $\max\{c^\mathsf{T}\mathbf{y} \mid \mathbf{y} \text{ satisfies } \mathcal{S}_y(C)\}$

has an interpretation that resembles the dynamic programming algorithm from Theorem 5.5 to maximize linear forms over DNNF circuits.

Theorem 5.19. *Let C be a smooth DNNF circuit. The system $\mathcal{S}_y(C)$ is TDI and the polyhedron $\mathcal{P}_y(C)$ is integral.*

Proof. It suffices to prove that the system $\mathcal{S}_y(C)$ is TDI by Theorem 5.4.

Dual problem. Let $c \in \mathbb{Z}^{\mathsf{edges}(C)}$. To write the dual of the LP problem $\max\{c^\mathsf{T} y \mid y \text{ satisfies } \mathcal{S}_y\}$, we associate a variable z_o to constraint (5.3a), variables z_g, for every $\vee$-gate g, to constraints (5.3b), variables $z_{g,e}$, for every $\wedge$-gate g and $e \in \mathsf{in}_\mathsf{E}(g)$, to constraints (5.3c). It will be useful to define, for every gate g, $\mathsf{z}(g)$ as the set of z variables corresponding to g. Note that sets $\mathsf{z}(g)$, for every gate g, partition all the z variables. Furthermore, if g is an input gate we have $\mathsf{z}(g) = \emptyset$, if g is a $\vee$-gate we have $\mathsf{z}(g) = \{z_g\}$, and if g is a $\wedge$-gate we have $\mathsf{z}(g) = \{z_{g,e} : e \in \mathsf{in}_\mathsf{E}(g)\}$. The dual of the LP problem $\max\{c^\mathsf{T} y : y \text{ satisfies } \mathcal{S}_y\}$ is

$$\begin{array}{lrl}
\min & z_o & \\
\text{s.t.} & z_h \geq c_e & \forall e = (g,h) \in \mathsf{edges}(C) \text{ with } g \text{ input, } h\ \vee\text{-gate,} \\
& z_{h,e} \geq c_e & \forall e = (g,h) \in \mathsf{edges}(C) \text{ with } g \text{ input, } h\ \wedge\text{-gate,} \\
& -z_g + z_h \geq c_e & \forall e = (g,h) \in \mathsf{edges}(C) \text{ with } g\ \vee\text{-gate, } h\ \vee\text{-gate,} \\
& -z_g + z_{h,e} \geq c_e & \forall e = (g,h) \in \mathsf{edges}(C) \text{ with } g\ \vee\text{-gate, } h\ \wedge\text{-gate,} \\
& -\sum_{f \in \mathsf{in}_\mathsf{E}(g)} z_{g,f} + z_h \geq c_e & \forall e = (g,h) \in \mathsf{edges}(C) \text{ with } g\ \wedge\text{-gate, } h\ \vee\text{-gate,} \\
& -\sum_{f \in \mathsf{in}_\mathsf{E}(g)} z_{g,f} + z_{h,e} \geq c_e & \forall e = (g,h) \in \mathsf{edges}(C) \text{ with } g\ \wedge\text{-gate, } h\ \wedge\text{-gate.}
\end{array}$$

Algorithm. Next, we give an algorithm that, as we will show later, constructs an optimal solution to the dual which is integral. The algorithm recursively assigns values to the variables starting from the inputs, and proceeding towards the output. One variable is assigned its value only when all variables associated with its inputs have already been assigned. More precisely, a variable in $\mathsf{z}(h)$ is assigned its value only when all variables $\mathsf{z}(g)$, for each input g of h have already been assigned.

At the very beginning, the algorithm considers the gates h such that every input of h is an input gate, since g is an input gate if and only if $\mathsf{z}(g) = \emptyset$. As a warm-up, we describe the algorithm in this simpler case. If h is a $\vee$-gate, there can be several constraints in the dual lower-bounding z_h:

$$z_h \geq c_e \qquad \forall e \in \mathsf{in}_\mathsf{E}(h).$$

Thus we assign $z_h^* := \max\{c_e : e \in \mathsf{in}_\mathsf{E}(h)\}$. If h is a $\wedge$-gate, there is exactly one constraint in the dual lower-bounding each variable $z_{h,e}$, for $e \in \mathsf{in}_\mathsf{E}(h)$:

$$z_{h,e} \geq c_e.$$

Thus we assign $z_{h,e}^* := c_e$ for each $e \in \mathsf{in}_\mathsf{E}(h)$.

Next, we describe the general iteration of the algorithm. Let h be a gate such that, for each input g of h, the variables in $\mathsf{z}(g)$ have already been assigned. Consider first the case

where h is a $\vee$-gate. The constraints in the dual lower-bounding z_h are:

$$\begin{aligned} z_h &\geq c_e && \forall e = (g,h) \in \mathsf{edges}(C) \text{ with } g \text{ input}, \\ z_h &\geq c_e + z_g && \forall e = (g,h) \in \mathsf{edges}(C) \text{ with } g \ \vee\text{-gate}, \\ z_h &\geq c_e + \sum_{f \in \mathsf{in}_\mathsf{E}(g)} z_{g,f} && \forall e = (g,h) \in \mathsf{edges}(C) \text{ with } g \ \wedge\text{-gate}. \end{aligned} \tag{5.9}$$

Note that all variables on the right-hand side of the above constraints have already been assigned by the algorithm. Thus we assign z_h^* as follows:

$$z_h^* := \max \begin{Bmatrix} c_e & \forall e = (g,h) \in \mathsf{edges}(C) \text{ with } g \text{ input}, \\ c_e + z_g^* & \forall e = (g,h) \in \mathsf{edges}(C) \text{ with } g \ \vee\text{-gate}, \\ c_e + \sum_{f \in \mathsf{in}_\mathsf{E}(g)} z_{g,f}^* & \forall e = (g,h) \in \mathsf{edges}(C) \text{ with } g \ \wedge\text{-gate} \end{Bmatrix}. \tag{5.10}$$

Next, consider the case where h is a $\wedge$-gate, and let $e = (g,h) \in \mathsf{in}_\mathsf{E}(h)$. There is exactly one constraint in the dual lower-bounding $z_{h,e}$, which is:

$$\begin{aligned} z_{h,e} &\geq c_e && \text{if } g \text{ input}, \\ z_{h,e} &\geq c_e + z_g && \text{if } g \ \vee\text{-gate}, \\ z_{h,e} &\geq c_e + \sum_{f \in \mathsf{in}_\mathsf{E}(g)} z_{g,f} && \text{if } g \ \wedge\text{-gate}. \end{aligned} \tag{5.11}$$

Also in this case, all variables on the right-hand side of the above constraints have already been assigned by the algorithm. Thus we assign $z_{h,e}^*$ as follows:

$$z_{h,e}^* := \begin{cases} c_e & \text{if } g \text{ input}, \\ c_e + z_g^* & \text{if } g \ \vee\text{-gate}, \\ c_e + \sum_{f \in \mathsf{in}_\mathsf{E}(g)} z_{f,e}^* & \text{if } g \ \wedge\text{-gate}. \end{cases} \tag{5.12}$$

This concludes the description of the algorithm. Note that, since c is integral, z^* is integral.

Termination. Our algorithm assigns a value to each variable. This follows from the structure of the directed graph C, since C is acyclic and the inputs of C are precisely the gates of C that have no ingoing edge. Moreover, C is a connected graph since we assumed there is a path from every gate g to the output o.

Feasibility. It is simple to see from the definition of our algorithm that the solution z^* constructed is feasible for the dual. In fact, in each step, the component of z^* considered is defined in a way that it satisfies all inequalities lower-bounding it. On top of that, each constraint in the dual is considered at some point in the algorithm.

Optimality. To prove that the solution z^* constructed by the algorithm is optimal for the dual, we show that any z feasible for the dual must have objective value greater than or equal to that of z^*, which is z_o^*. To do so, we give a procedure that selects a number of constraints of the dual. Summing together the selected constraints yields the inequality $z_o \geq z_o^*$. We remark that the procedure is strongly connected to the concept of certificates.

The procedure is recursive and starts from the gate o, which in particular is a $\vee$-gate. We give the general recursive construction for a gate h.

- If h is an input gate, there is nothing to do.
- If h is a $\vee$-gate, we select one constraint among (5.9) that achieves the maximum in (5.10). This essentially amounts to selecting an edge $e = (g, h) \in \mathsf{edges}(C)$, and therefore a new gate g. We then apply recursively the construction to the gate g.
- If h is a $\wedge$-gate, we select, for every $e \in \mathsf{in_E}(h)$, the one constraint (5.11). This essentially amounts to selecting all edges $e = (g, h) \in \mathsf{edges}(C)$, and therefore all new gates g that are inputs of h. We then apply recursively the construction to all inputs g of h.

It is simple to check that summing all the inequalities selected by the above recursive procedure yields $z_o \geq z_o^*$. □

From what precedes, Theorem 5.19 is enough to imply that $\mathcal{P}_x \subseteq \mathrm{conv}(f_C)$ and we therefore have shown that $\mathcal{P}_x(C) = \mathrm{conv}(f_C)$. We summarize our main theorem as follows:

Theorem 5.20. *Given a smooth DNNF circuit C on variables X, one can construct in time $O(|C|)$ a system $\mathcal{S}_{x,y}(C)$ with $|C| + |X|$ variables and $O(|C|)$ constraints such that* $\mathrm{conv}(f_C) = \{\mathbf{x} \mid \exists \mathbf{y}, (\mathbf{x}, \mathbf{y})$ *satisfies* $\mathcal{S}_{x,y}(C)\}$.

In other words, Theorem 5.20 states that we can efficiently construct an extended formulation of $\mathrm{conv}(f_C)$ of size $O(|C|)$ for any smooth DNNF circuit (and of size $O(n|C|)$ for non-smooth DNNF circuits, after having smoothed the circuit). In the rest of this section, we revisit several known results establishing the existence of polynomial size extended formulation for problems from the literature of optimization theory by exhibiting a DNNF circuit that computes the set of answers.

5.3 Binary Polynomial Optimization

5.3.1 Problem definitions

In this section, we explore the consequences of the results from Section 5.2 for the optimization problem known as Binary Polynomial Optimization. This problem asks to maximize a multilinear polynomial over binary points. More formally, we are given a multilinear polynomial $P = \sum_I a_I \prod_{x \in I} x$ over variables X, with rational coefficients $a_I \in \mathbb{Q}$ and we want to find the assignment $\tau \in 2^X$ such that the value $P(\tau) := \sum_I a_I \prod_{x \in I} \tau(x)$ is maximal. This is an NP-hard problem, even when we restrict the degree of the monomials to be 2 [GJS76], as one can see a direct reduction to the problem of finding a minimum-size vertex cover of a graph $G = (V, E)$. Indeed, one can see that the polynomial $|V|(\sum_{(x,y) \in E} OR(x, y) - |E|) - \sum_{x \in V} x$ is maximized when we map x variables to a minimum size vertex cover of G. It has interesting applications, in classic operations research, computer vision, communication engineering, and theoretical physics [Ber87; BH02; Sch09; Lie+10; Ish11].

One direction that has been considered heavily is akin to what we did for CNF formulas in Chapter 2: one maps the polynomial to a graph or a hypergraph representing the interaction between monomials and variables. The structure of this hypergraph is then leveraged for finding tractable classes. More formally, we see an instance of this problem as a pair (H, p)

where H is the hypergraph $H = (V, E)$ and p is a profit function $p: E \to \mathbb{Q}$. The *Binary Polynomial Optimization* problem for (H, p) is defined as finding an *optimal solution* $x^* \in \{0,1\}^V$ to the following maximization problem: $\max_{x \in \{0,1\}^V} P_{(H,p)}(x)$ where $P_{(H,p)}$ is the polynomial defined as $P_{(H,p)}(x) = \sum_{e \in E} p(e) \prod_{v \in e} x(v)$. We denote this problem as $\mathsf{BPO}(H, p)$. The value $P_{(H,p)}(x^*)$ is called the *optimal value* of $\mathsf{BPO}(H, p)$. To the best of our knowledge, three main polynomially-solvable classes of BPO based on the structure of the hypergraph have been identified so far in the literature: the case when the primal treewidth of H [CHJ90], denoted by $\mathsf{ptw}(H)$, is bounded, the case when H is β-acyclic and the case when H is a cycle hypergraph [DD22; DD23].

One natural approach to solve the $\mathsf{BPO}(H, p)$ problem in practice is to reduce it to an integer linear program with the same optimal value. A direct reduction can be seen as follows, by introducing a new variable y_e for each edge of H which encodes whether the associated monomial takes value 1 or 0. The full reduction is presented below:

$$
\begin{aligned}
\max_{x} \quad & \sum_{e \in E} p(e) y(e) \\
\text{s.t.} \quad & y(e) = \prod_{v \in e} x(v) \\
& x(v) \in \{0,1\} \\
& y(e) \in \{0,1\}
\end{aligned}
$$

which can be further reduced to the following linear program:

$$
\begin{aligned}
\max_{x} \quad & \sum_{e \in E} p(e) y(e) \\
\text{s.t.} \quad & y(e) \leq x(v) \text{ for all } v \in e \\
& \sum_{v \in e} x(v) \leq |e| - 1 + y(e) \\
& 0 \leq x(v) \leq 1 \\
& 0 \leq y(e) \leq 1
\end{aligned}
$$

The integer solutions of this linear program are obviously in one-to-one correspondence with the solutions of the original $\mathsf{BPO}(H, p)$ problem, which gives a way of solving it in practice, though not polynomial time, using integer linear programing solvers such as SCIP, Gurobi or CPLEX [Gur23; Bes+21; Cpl09]. Unfortunately, these tools rarely exploit the underlying structure of how the variables and the constraints interact together. In this section, we show that there is another natural reduction to the problem of maximizing a weighted Boolean function. We will then show how to efficiently encode this Boolean function as a CNF formula while retaining the structural properties of the $\mathsf{BPO}(H, p)$ instance. We will then be able to recover the known tractability results and generalize them by using knowledge compilation and Theorem 5.5.

5.3.2 Solving BPO via Knowledge Compilation

We consider a Boolean function that has been originally introduced by Del Pia and Khajavirad in [DK17]: given a hypergraph $H = (V, E)$, the *multilinear set* f_H *of* H is the Boolean function on variables $V \cup E$ such that τ is a model of f_H if and only if for every $e \in E$, $\tau(e) = \prod_{v \in e} \tau(v)$.

Moreover, given a profit function $p\colon E \to \mathbb{Q}$, we define a weight function $w_p\colon (V \cup E) \times \{0, 1\} \to \mathbb{Q}$ as:

- for every $e \in E$, $w_p(e, 1) = p(e)$ and $w_p(e, 0) = 0$
- for every $v \in V$ and $b \in \{0, 1\}$, $w_p(v, b) = 0$.

Now, observe that $\tau \in 2^{V \cup E}$ is a solution of f_H if and only if $w_p(\tau) = P_{(H,p)}(\tau|_V)$. Moreover, for every $x \in 2^V$, there exists a unique $y \in 2^E$, defined as $y(e) := \prod_{v \in e} x(v)$ such that $x \times y$ is a solution of f_H. By what precedes, $w_p(x \times y) = P_{(H,p)}(x)$. Hence, we have a bijection between the weight of the solutions of f_H and the possible values of the BPO problem (H, p). In particular, it holds for the optimal solutions:

Theorem 5.21. *Let (H, p) be an instance of BPO. The set of optimal solutions of* $\mathsf{BPO}(H, p)$ *is equal to $\{\tau^*|_V \mid \tau^*$ is an optimal solution of $(f_H, w_p)\}$.*

Hence, if the optimal solutions of (f_H, w_p) can be found efficiently, we directly get an efficient algorithm for $\mathsf{BPO}(H, p)$. We now show that a lot of such instances are tractable by compiling them into DNNF circuits. To relate with the results from Chapter 2, we first need to express f_H as a CNF formula. We start by doing the obvious reduction and observe that it preserves both primal and incidence treewidth. We start by observing that f_H can be expressed as the following Boolean formula: $\bigwedge_{e \in E}(e \iff \bigwedge_{v \in e} v)$, which can obviously be expanded to the following CNF formula obtained from the conjunction of the following clauses:

- for every $e \in E$, we have a clause R_e defined as $e \vee \bigvee_{v \in e} \neg v$,
- and for every $e \in E$ and for every $v \in e$, we have the clause $L_{v,e}$ defined as $\neg e \vee v$.

Example 5.22. Let us consider the multilinear expression $-3v_1v_2v_3 + 4v_4v_5 + 5v_2v_3v_4v_5v_6$. The typical hypergraph that represents it is $H = (V, E)$ with $V = \{v_1, v_2, v_3, v_4, v_5, v_6\}$ and E containing only three edges $\{v_1, v_2, v_3\}$, $\{v_4, v_5\}$, $\{v_2, v_3, v_4, v_5, v_6\}$.

Now let us construct its CNF encoding, which involves 9 variables and 13 clauses. Specifically, it is:

$$(\neg e_1 \vee v_1) \wedge (\neg e_1 \vee v_2) \wedge (\neg e_1 \vee v_3) \wedge (\neg v_1 \vee \neg v_2 \vee \neg v_3 \vee e_1) \wedge$$
$$(\neg e_2 \vee v_4) \wedge (\neg e_2 \vee v_5) \wedge (\neg v_4 \vee \neg v_5 \vee e_2) \wedge$$
$$(\neg e_3 \vee v_2) \wedge (\neg e_3 \vee v_3) \wedge (\neg e_3 \vee v_4) \wedge (\neg e_3 \vee v_5) \wedge (\neg e_3 \vee v_6) \wedge$$
$$(\neg v_2 \vee \neg v_3 \vee \neg v_4 \vee \neg v_5 \vee \neg v_6 \vee e_3).$$

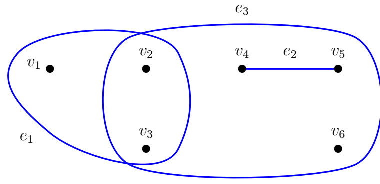


Figure 5.3: Hypergraph H representing $x_1x_2x_3 + x_4x_5 + x_2x_3x_4x_5x_6$.

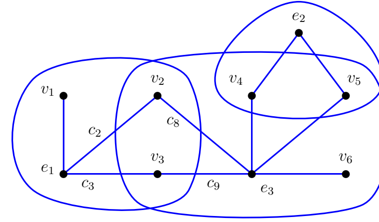


Figure 5.4: Hypergraph representing first CNF construction.

The corresponding hypergraph is depicted in Figure 5.4. ◇

Formula F_H clearly encodes f_H and has size linear in $|H|$. But more interestingly, it preserves the incidence treewidth:

Lemma 5.23. *For every hypergraph H, the satisfying assignments of F_H are exactly f_H. Moreover, $\|F_H\| = O(|H|)$, F_H can be constructed in time $O(|H|)$ and $\mathsf{itw}(F_H) \leq 2 \cdot (1 + \mathsf{itw}(H))$.*

Proof. The first part of the lemma is straightforward from the definition of F_H. We focus on the incidence treewidth of F_H. The proof is based on the following observation. Let $G' = (V', E')$ be the graph obtained from $\mathsf{Inc}(H)$ as follows: we rename every edge $e \in E$ as R_e. We connect to R_e a fresh vertex labeled by e. We also connect this new vertex to every $v \in e$. Observe that R_e and e are modules in G', that is, they have the same neighborhood. It is then easy to see that the treewidth of G' is at most $2k+1$ where k is the incidence treewidth of H. Indeed, given a tree decomposition of $\mathsf{Inc}(H)$, we can add a vertex R_e to every bag where e appears and it gives a tree decomposition of G' of width at most $2(k+1)-1 = 2k+1$.

Now, one can get $\mathsf{Inc}(F_H)$ by splitting the edge $\{e, v\}$ of G' with a node $L_{v,e}$. Hence $\mathsf{Inc}(F_H)$ is obtained from G' by splitting edges, an operation that preserves treewidth. Indeed, in any tree decomposition of G', there is a bag covering the edge $\{e, v\}$. We can attach two leaves to this bag: one labeled $\{L_{v,e}, e\}$ and the other labeled $\{L_{v,e}, v\}$. This gives a tree decomposition of G' of the same width. Hence, we proved $\mathsf{itw}(F_H) \leq \mathsf{tw}(G') \leq 2(1 + \mathsf{itw}(H))$. □

We observe here that bounded incidence treewidth also captures a class of hypergraphs that is known to be tractable for BPO: the class of cycle hypergraphs [DD21]. We define a *cycle hypergraph* as a hypergraph $H = (V, E)$ such that $E = \{e_0, \dots, e_{n-1}\}$ and $e_i \cap e_j \neq \emptyset$ if and only if $j = i + 1 \mod n$ or $j = i - 1 \mod n$. We remark that this definition is slightly more general than the one in [DD21]. While the primal treewidth of a cycle hypergraph can be arbitrarily large (since an edge of size k induces a k-clique in the primal graph and hence treewidth larger than k), the incidence treewidth of cycle hypergraphs is 2. We omit the proof, which is elementary and can be found in [CPG23].

Lemma 5.24. *For every cycle hypergraph $H = (V, E)$, we have $\mathsf{itw}(H) \leq 2$.*

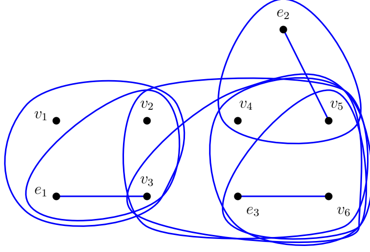
Unfortunately, the previous CNF encoding does not preserve β-acyclicity for which the tractability of $\mathsf{BPO}(H, p)$ is known [DD22; DD23] and for which we know how to compile [Cap17].

To see this, we use the example from Figure 5.3. We remark that H is β-acyclic since $\{v_1, \dots, v_6\}$ is a β-elimination order, but F_H, depicted in Figure 5.4, is not β-acyclic, since it contains, for example, the β-cycle formed by the vertices $\{v_2, e_1, v_3, e_3\}$ and edges $\{c_2, c_3, c_9, c_8\}$. We can nevertheless design a CNF encoding of f_H for $H = (V, E)$ that preserves β-acyclicity, as follows:

Definition 5.25. *Given a total order $\prec$ on V, we let $F_H^{\prec}$ be a CNF on variables $V \cup E$ with having clauses:*

- *for every $e \in E$, there is a clause $R_e = e \vee \bigvee_{v \in e} \neg v$,*
- *for every $e \in E$ and $v \in e$, there is a clause $L_{v,e}^{\prec} = \neg e \vee v \vee \bigvee_{w \in e, v \prec w} \neg w$.*

The CNF encoding from Example 5.22 following the variable order $(v_1, \dots, v_6)$ and its hypergraph are depicted in Figure 5.5. It is easy to see that this hypergraph is β-acyclic.



$$
\begin{gathered}
(\neg e_1 \vee v_1 \vee \neg v_2 \vee \neg v_3) \wedge \\
(\neg e_1 \vee v_2 \vee \neg v_3) \wedge (\neg e_1 \vee v_3) \\
\wedge (\neg v_1 \vee \neg v_2 \vee \neg v_3 \vee e_1) \wedge \\
(\neg e_2 \vee v_4 \vee \neg v_5) \wedge (\neg e_2 \vee v_5) \\
\wedge (\neg v_4 \vee \neg v_5 \vee e_2) \wedge \\
(\neg e_3 \vee v_2 \vee \neg v_3 \vee \neg v_4 \vee \neg v_5 \\
\vee \neg v_6) \wedge (\neg e_3 \vee v_3 \vee \neg v_4 \vee \neg v_5 \\
\vee \neg v_6) \wedge (\neg e_3 \vee v_4 \vee \neg v_5 \vee \neg v_6) \\
\wedge (\neg e_3 \vee v_5 \vee \neg v_6) \wedge (\neg e_3 \vee v_6) \\
\wedge (\neg v_2 \vee \neg v_3 \vee \neg v_4 \vee \neg v_5 \vee \\
\neg v_6 \vee e_3).
\end{gathered}
$$

Figure 5.5: Hypergraph and CNF for β-acyclic preserving encoding.

As before, the satisfying assignments of $F_H^{\prec}$ are the same as f_H since R_e encodes the constraint $\bigwedge_{v \in e} v \Rightarrow e$ and $\bigwedge_{v \in e} L_{v,e}^{\prec}$ encodes $e \Rightarrow \bigwedge_{v \in e} v$. Indeed, if we write $e = \{v_1, \dots, v_k\}$ with $v_1 \prec \dots \prec v_k$, we have that $L_{v_k,e}^{\prec}$ is $e \Rightarrow v_k$, then $L_{v_{k-1},e}^{\prec}$ is $(e \wedge v_k) \Rightarrow v_{k-1}$. Together, these constraints are equivalent to $e \Rightarrow (v_1 \wedge \dots \wedge v_k)$. It is an easy induction to see that $\bigwedge_{v \in e} L_{v,e}^{\prec}$ is then equivalent to $e \Rightarrow \bigwedge_{v \in e} v$.

Lemma 5.26. *For every hypergraph $H = (V, E)$ and order $\prec$ on V, the satisfying assignments of $F_H^{\prec}$ are exactly f_H. We have $\|F_H^{\prec}\| = O(|V| \cdot |H|)$ and $F_H^{\prec}$ can be constructed in time $O(|V| \cdot |H|)$. Moreover, if $\prec$ is a β-elimination order of H, then $F_H^{\prec}$ is β-acyclic.*

Proof. The first part is easy, so we focus on proving β-acyclicity of $F_H^{\prec}$. We claim that $(e_1, \dots, e_m, v_1, \dots, v_n)$ is a β-elimination order of $H(F_H^{\prec})$, where $V = \{v_1, \dots, v_n\}$ with $v_i \prec v_j$ for any $i < j$ and $E = \{e_1, \dots, e_m\}$ is an arbitrary order on E. First we observe that for any $e \in E$, e is a nest point of $H(F_H^{\prec})$. Indeed, e is in R_e and $L_{v,e}^{\prec}$ for every $v \in e$. Now, e is a nest point because $\mathsf{var}(L_{v,e}^{\prec}) \subseteq \mathsf{var}(L_{w,e}^{\prec})$ if $w \prec v$ and $\mathsf{var}(L_{v,e}^{\prec}) \subseteq \mathsf{var}(R_e)$ by definition.

Hence, we can eliminate $e_1, \dots, e_m$ from $H(F_H^{\prec})$. Let H' be the resulting hypergraph. Observe that the vertices of H' are $\{v_1, \dots, v_n\}$. Our goal is to show that $(v_1, \dots, v_n)$ is a

β-elimination order for H' and we will do it using the fact that $(v_1, \ldots, v_n)$ is a β-elimination order of H.

Now consider v_1. It is by definition a nest point of H. We show that it is also a nest point of H'. By definition, v_1 is in $R_e \setminus \{e\}$ for every $e \in H$ containing v_1. It is also in $L^{\prec}_{w,e} \setminus \{e\}$ for every $w \preceq v_1$. But by definition, it only contains v_1. Now observe that $L^{\prec}_{v_1,e} \setminus \{e\} = R_e \setminus \{e\}$ and they contain exactly the vertices in e. Hence v_1 is a nest point in H' because it is in the same edges as in H.

It is now easy to see by induction that once $v_1, \ldots, v_i$ have been removed, then for every e of H containing v_{i+1}, for every $w \preceq v_{i+1}$, v_{i+1} is in $R_e \setminus \{e, v_1, \ldots, v_i\} = L^{\prec}_{w,e} \setminus \{e, v_1, \ldots, v_i\} = e \setminus \{v_1, \ldots, v_i\}$. Hence v_{i+1} is a nest point in H' because it is a nest point in H. $\square$

Theorem 5.5 becomes interesting for our purposes since it is known that both β-acyclic CNF formulas [Cap17] and bounded incidence treewidth CNF formulas can be represented as polynomial size DNNF circuits, by Theorem 2.39 (or see [Bov+15]).

Theorem 5.27 ([Cap17, Theorem 8])**.** *Let F be a β-acyclic CNF formula. One can construct a smooth*[3] *DNNF circuit C of size $O(\|F\|)$ in time $O(\|F\|)$ such that f_C is the set of satisfying assignments of F.*

As a consequence of Theorem 5.27 and Theorem 2.39, we can represent the multilinear set of a BPO problem (H, p) with a polynomial size DNNF circuit if the incidence treewidth of H is bounded:

Theorem 5.28. *Let $H = (V, E)$ and $t = \mathsf{itw}(H)$. The multilinear set f_H of H can be computed by a deterministic TDD and hence a smooth DNNF circuit of size $O(16^t \cdot (|V| + |E|))$ that can be constructed in time $O(16^t \cdot \text{poly}(|V| + |E|))$.*

Proof. We construct F_H from H. It can be constructed in time $O(\|H\|) = O(|V| \cdot |E|)$, has $n = |V| + |E|$ variables, $m \leq |E|(1 + |V|)$ clauses and incidence treewidth at most $2(t + 1)$ by Lemma 5.23. Hence the construction follows by applying Theorem 2.39 to F_H. $\square$

Symmetrically, if H is β-acyclic we get:

Theorem 5.29. *Let $H = (V, E)$ be a β-acyclic hypergraph. The multilinear set f_H of H can be computed by a smooth DNNF circuit of size $O(|H| \cdot |V|)$ that can be constructed in time $O(|H| \cdot |V|)$.*

Hence, combined with Theorem 5.5 and the reduction from Theorem 5.21, we directly get the tractability of $\mathsf{BPO}(H, p)$ for bounded incidence treewidth, a result generalizing existing ones and for β-acyclicity. More interestingly, these results are recovered using a single technique. In a sense, the knowledge compilation approach shows that the tractability of $\mathsf{BPO}(H, p)$ in these cases can be explained by the fact that the underlying polynomial can be factorized efficiently in a way that allows for dynamic programming.

[3] Smoothness is not explicit in [Cap17] but the construction can be made smooth while preserving linear size by observing that if one variable is missing then it comes from a clause that has been satisfied, so we can add a trivial input for this variable.

Theorem 5.30. *Let (H, p) be an instance of* $\mathsf{BPO}(H, p)$. *If H has incidence treewidth t, then we can solve* $\mathsf{BPO}(H, p)$ *in time* $2^{O(t)} \cdot \text{poly}(|H|)$.

Theorem 5.31. *Let (H, p) be an instance of* $\mathsf{BPO}(H, p)$. *If H is β-acyclic, then we can solve* $\mathsf{BPO}(H, p)$ *in strongly polynomial time.*

More interestingly, we get more than tractability via our technique. We also get extended formulations of the multilinear set via Theorem 5.10. Polynomial size extended formulations for $\mathsf{BPO}(H, p)$ are more interesting than ad-hoc tractability algorithms because they allow for better integration into existing solvers.

Extracting polynomial size extended formulations from tractable classes of $\mathsf{BPO}(H, p)$ has been a delicate matter, often requiring a separate paper to establish them, see [DK24; DK17; DK18; WJ04; Lau09; BM18; DK23]. These constructions become simple corollaries of Theorem 5.10 and the existence of a polynomial size smooth DNNF circuit computing the multilinear set, as given by Theorems 5.30 and 5.31.

Theorem 5.32. *Let $H = (V, E)$ and $t = \mathsf{itw}(H)$. There exists an extended formulation of the multilinear set f_H of H of size $2^{O(t)} \cdot \text{poly}(|H|)$ that can be constructed in time $2^{O(t)} \cdot \text{poly}(|H|)$.*

Theorem 5.33. *Let $H = (V, E)$ be a β-acyclic hypergraph. There exists an extended formulation of the multilinear set f_H of H of size polynomial in $|H|$ that can be constructed in polynomial time.*

While the main advantage of using Theorem 5.10 is to have more modular proofs, we observe that Theorem 5.32 is actually new since polynomial size extended formulations were only known for cycle hypergraphs and bounded primal treewidth hypergraphs.

5.3.3 Beyond BPO

We review in this section extensions of BPO that can easily be dealt with thanks to the DNNF circuit framework. We do not go into detail but illustrate the approach in a high-level manner. We refer the interested reader to [CPG23].

Cardinality Constraints. One problem with BPO is that it does not allow to add constraints. This is something that can be dealt with in some cases using a DNNF circuit representation of the multilinear set. Indeed, assume one wants to find the optimal solution of a BPO instance that also respects some constraints $\mathcal{C}$ over the variables x_i. One could start by computing a DNNF circuit C computing the multilinear set of the BPO instance and then transform C into a DNNF circuit so that it only accepts the models of C respecting $\mathcal{C}$. If C is a subclass of DNNF circuits supporting the apply operator, such as TDD or OBDD, one way of doing so would be to compile $\mathcal{C}$ into a similar circuit C' and then compute $\mathsf{apply}(C, C', \wedge)$. This approach would be interesting for bounded treewidth instances since we can compute a polynomial size TDD. In the case of β-acyclicity, the technique does not generalize. That said, we can focus on a specific set of constraints known as *cardinality constraints* that can always be enforced in DNNF circuit.

A cardinality constraint is a constraint that restricts the value of $\sum_{v \in V} v$. For example, we may want to enforce an upper bound U on the number of variables we are allowed to set to 1 by adding constraint $\sum_{v \in V} v \leq U$. Interestingly, we can always enforce such cardinality constraints in DNNF circuits by multiplying the size of the circuit by n. This technique is known under the name "homogenization" for arithmetic circuits [Bür00, Lemma 2.14] and can straightforwardly be lifted to DNNF circuits. Given $\tau \in 2^X$, the *Hamming weight of* τ is defined as $|\tau^{-1}(1)|$, that is, the number of variables mapped to 1. We have:

Lemma 5.34. *Let C be a DNNF circuit with n variables. One can build a DNNF circuit C' of size at most $n|C|$ in time $O(n|C|)$ such that for every $i \leq n$, C' has a gate o^i computing the set of models of C having Hamming weight i.*

Proof. We only sketch the proof. We assume the circuit to be smooth to simplify. The trick is to introduce a copy v^i of each gate v of C computing the models of v of Hamming weight i (over $\mathsf{var}(v)$). This is straightforward for inputs as they have exactly one model. If v is a $\vee$-gate with inputs $w_1, \dots, w_k$, then $v^i = \bigvee_{j=1}^{k} w_j^i$. Indeed, a model of v having Hamming weight i must be a model of Hamming weight i of one of its input. If v is a $\wedge$-gate with inputs w_1, w_2, we have $v^i = \bigvee_{k=0}^{i} w_1^k \wedge w_2^{i-k}$. Observe that each $w_1^k \wedge w_2^{i-k}$ is still a decomposable $\wedge$-gate, hence C' is indeed a DNNF circuit. □

Now, for any $S \subseteq [n]$, we can add a $\vee$-gate $o_S = \bigvee_{i \in S} o^i$ in the circuit from Lemma 5.34 and we have a gate that accepts exactly the models of C whose Hamming weight is in S. Hence, we can force *extended cardinality constraints* in DNNF circuits, where an extended cardinality constraint is a constraint of the form $\sum_{v \in V} v \in S$ for some $S \subseteq [|V|]$.

A direct consequence of Lemma 5.34 is that bounded incidence treewidth or β-acyclic instances of BPO with one extended cardinality constraint can be solved in polynomial time.

Top-k for BPO. Another low-hanging fruit of the circuit-based approach for BPO is that DNNF circuits allow to recover, for any k, the top k best models. This can be useful heuristically when one is interested in getting a good answer that fits other constraints that are harder to formalize or to enforce. In this case, one can have a look at a larger set of near optimal instances and check whether some of these answers are suitable. Given a DNNF circuit C and a weight function w of its variables, we can adapt the dynamic programming algorithm from Theorem 5.5 so that it computes top-k set of models, that is, a set $T_k = \{\tau_1, \dots, \tau_k\}$ of distinct models of C such that $w(\tau_1) \geq \dots \geq w(\tau_k)$ and such that for every model τ of C that is not in T_k, we have $w(\tau) \leq w(\tau_k)$. This has been observed in particular in [Bou+22].

Theorem 5.35 (Reformulated from [Bou+22])**.** *Given a DNNF circuit C on variables Z and a weight function w on Z, one can compute a top-k set for the Boolean Maximization Problem (f_C, w) in time $O(|C| \cdot k \log k)$.*

Hence, as soon as the multilinear set of a BPO instance can be expressed as a DNNF circuit, we can find k best models in polynomial time in k and in the size of the circuit. In particular, this establishes the tractability of top-k BPO in the case of bounded incidence treewidth or β-acyclicity.

Solving Binary Polynomial Optimization with Literals. We conclude this tour about BPO extensions with a natural extension in expressivity. The current form of BPO only allows for polynomials expressed as an explicit sum of monomials. An easy consequence of reencoding BPO as a weighted Boolean function is that values 0 and 1 are completely symmetric. Hence, nothing prevents us from encoding polynomials where the monomials both use x and $(1-x)$. More precisely, we identify a set of literals L over variables V with a monomial over V via $m(L) := \prod_{\ell \in L} \sigma(\ell)$ where $\sigma(v) = v$ and $\sigma(\neg v) = 1 - v$. A *polynomial with literals* is defined as a pair (M, p) where M is a set of sets of literals and a profit function $p \colon M \to \mathbb{Q}$. The size $|M|$ is defined as $\sum_{L \in M} |L|$, the variables $V(M)$ are defined as $\bigcup_{L \in M} \mathsf{var}(L)$ and *the polynomial* $P_{(M,p)}$ *induced by* M is defined as $\sum_{L \in M} p(L)m(L)$. The *Binary Polynomial Optimization Polynomial with Literal problem* ($\mathrm{BPO_L}$for short) is defined as the problem of finding the assignment $\tau \in 2^{V(M)}$ which maximizes the value $P_{(M,p)}(\tau)$. As before, we can define the *multilinear set of a* BPO_L *instance* as the Boolean function over variables $M \cup V$ whose models are the assignments τ such that $\tau(L) = \prod_{\ell \in L} \tau(\ell)$ where $\tau(\neg x) := 1 - \tau(x)$ by definition.

To construct such a DNNF circuit, we proceed as before: we encode the multilinear set of a $\mathrm{BPO_L}$ instance as the CNF formula F_M as before:

- $R'_L = L \vee \bigvee_{\ell \in L} \neg \ell$ for every $L \in M$ and every $\ell \in L$ (where $\neg\neg\ell = \ell$),
- $L'_{e,v} = \neg e \vee \ell$ for every $L \in M$ and $\ell \in L$.

It is clear that the models of F_M encode the multilinear set of (M, p). Moreover, if we weight the variables as $w_p(L, 1) = p(L)$ and $w_p(x, b) = 0$ for every other value, then for every model τ of F_M, we have $w_p(\tau) = P_{(M,p)}(\tau|_{V(M)})$. Hence, as before, having a small DNNF circuit computing F_M is enough to ensure the tractability of $\mathrm{BPO_L}$.

Given a polynomial with literals (M, p), we define its incidence graph $\mathsf{Inc}(M)$ as the bipartite graph over vertex set $V(M) \cup M$ such that there is an edge between $v \in V(M)$ and $L \in M$ if and only if $v \in \mathsf{var}(L)$. The *incidence treewidth of* (M, p) is defined as the treewidth of its incidence graph. With the same proof as Lemma 5.23, we can show that the incidence treewidth of F_M is at most $2k+1$ where k is the incidence treewidth of (M, p). We can hence efficiently construct a DNNF circuit computing F_H and conclude:

Theorem 5.36. *Let* (M, p) *be an instance of* BPO_L. *If* M *has incidence treewidth* t, *one can construct a DNNF circuit computing the multilinear set of* (M, p) *in time* $2^{O(t)}\,\mathrm{poly}(|M|)$ *and of size* $2^{O(t)}\,\mathrm{poly}(|M|)$. *In particular, we can solve* BPO_L *in time* $2^{O(t)}\,\mathrm{poly}(|M|)$ *and construct an extended formulation of the multilinear set* f_M *of size* $2^{O(t)}\,\mathrm{poly}(|M|)$.

Given a polynomial with literals (M, p), we define its hypergraph $H(M)$ as the hypergraph over vertex set $V(M)$ with edges $\{\mathsf{var}(L) \mid L \in M\}$. As before, we need an alternative encoding to preserve β-acyclicity in instances. If $\prec$ is an order on V, we define $F_M^{\prec}$ as the conjunction of:

- $R'_e = y_e \vee \bigvee_{v \in e} \neg \sigma_e(x_v)$ for every $e \in E$ (where we replace $\neg\neg x$ by x),
- $L'_{e,v,\prec} = \neg y_e \vee \sigma_e(x_v) \vee \bigvee_{w \in e, v \prec w} \neg \sigma_e(x_w)$ for every $e \in E$ and $v \in e$ (where we replace $\neg\neg x$ by x).

As before, $F_M^{\prec}$ is clearly a CNF encoding of the multilinear set of M and if $\prec$ is a β-elimination order of $H(M)$, then $F_M^{\prec}$ is β-acyclic. Hence, we have:

Theorem 5.37. *Let (M, p) be an instance of BPO_L. If $H(M)$ is β-acyclic, one can construct a DNNF circuit computing the multilinear set of (M, p) in time* $\text{poly}(|M|)$ *and of size* $\text{poly}(|M|)$*. In particular, we can solve BPO_L in time* $\text{poly}(|M|)$ *and construct an extended formulation of the multilinear set f_M of size* $\text{poly}(|M|)$*.*

Theorems 5.36 and 5.37 are generalizations of Theorems 5.28 and 5.29. They are based on the same technique of solving an optimization problem through DNNF circuit compilation. Hence, the same generalizations as mentioned in this section also work in the case of BPO_L. For example, we can find the k-best solutions of a BPO_L instance of bounded incidence treewidth, even if one has an extended cardinality constraint.

5.4 Conclusion

This chapter has presented an interesting connection between extended formulations and knowledge compilation, which arose as an unexpected consequence of an interesting collaboration with Silvia Di Gregorio and Alberto Del Pia. In a way, extended formulations have the same purpose as knowledge compilation: representing an interesting set in a succinct yet tractable way. For BPO, the interesting set is the multilinear set, which is given on input in the form of a multilinear polynomial, which is not a good representation for optimization purposes. Extended formulations unlock this tractability, at the cost of rewriting the input into a friendlier format, that may sometimes be of exponential size, but sometimes not, depending on the structure of the original polynomial.

In the case of extended formulations, tractability is measured by the fact that we can find the optimal value of a linear form over the set in polynomial time in the size of its representation, but we could go one step further and study extended formulations of $\text{conv}(f)$ for a Boolean function f as a way of representing Boolean functions. As we have shown, this gives a representation that is at least as compact as DNNF circuits, possibly exponentially more compact than DNNF circuits though we have not proved an exponential separation yet, which is, in itself, an interesting question that is certainly worth exploring. Extended formulation can be conditioned without size increase, and variables can be existentially projected (without doing anything but projecting additional variables beyond those already projected).

Another interesting direction would be to use this connection to solve more complex optimization problems on DNNF circuits. Indeed, if one only wants to maximize a linear form over the models of a DNNF circuit C, then extended formulations are not necessary as the usual dynamic programming algorithm is good enough. That said, if we now want to solve a more challenging optimization problem, we can use the extended formulation to embed it into a linear programming solver together with the other constraints.

Interestingly, exponential lower bounds on the size of extended formulations of some convex set have been proven in the literature, the most notorious being the one by Yannakakis [Yan91], who connected the size of the smallest extended formulation of a convex set with the nonnegative rank of its so-called slack matrix. Interestingly, lower bounds on this rank have been

proven using rectangle covers and techniques from communication complexity, see [CCZ10, Section 6]. This approach is actually similar to the approach we have used to prove lower bounds on DNNF circuit size [Bov+16]. We leave the exploration of this connection for future work.

We conclude by observing that extended formulations obtained from circuits may have other applications that we are planning to explore. The first one is from a result by Kolman, Koutecký and Tiwary about extended formulations from Monadic Second Order (MSO) formulas [KKT20]. In this work, they show that some polyhedron defined from the answers of an MSO formula on bounded treewidth graphs admits extended formulation of FPT size. It seems that this result is connected to the result presented in this chapter since the answer set of MSO formulas is known to have FPT size deterministic DNNF circuits [Ama+17]. It seems possible to recover the extended formulation from this DNNF circuit directly. Interestingly, the proof from [KKT20] bears some resemblance with the DNNF approach. For example, one building block for the proof is showing that the Cartesian product of two polyhedra $\mathcal{P}_1, \mathcal{P}_2$ has an extended formulation that can be obtained from extended formulations of $\mathcal{P}_1$ and $\mathcal{P}_2$ respectively, which is basically what we do for decomposable $\wedge$-gates.

A last possible connection between this chapter and previous work is the main result of Nicolas Crosetti's PhD thesis [Cro23], supervised by Joachim Niehren, Jan Ramon, Sophie Tison, and me, which can also be found in [Cap+24]. In this work, we were interested in solving linear programs whose variables are the answers of some join query Q. We develop a query language to write linear constraints parametrized by queries, which can then be interpreted as linear programs when we solve the query Q. Since the number of answers of Q may be large, the resulting linear program may have a large number of variables. We propose a rewriting of the linear program into a much smaller one, having the same optimal value but having $O(\|Q\|^k)$ variables instead of $|[\![Q]\!]|$, where k is the fractional hypertree width of Q. In [Cap+24], the smaller linear program is extracted directly from a tree decomposition of Q but in Nicolas Crosetti's thesis [Cro23], an alternative proof is given where the rewritten linear program is directly extracted from a relational circuit. The core of the rewriting corresponds exactly to the extended formulation we give for the convex hull of DNNF circuits in Theorem 5.20. We believe that there is a way of restating the main result from [Cap+24] as an optimization problem on some notion of convex hull of query answers, for which the results of Section 5.2.3 would provide an extended formulation. That said, the precise connection still eludes us and we leave this research direction for future work.

Conclusion

We arrive at the end this manuscript and we use this conclusion to draw parallels between the different chapters and offer some perspective on future work. This manuscript is a review of our work in the domain of knowledge compilation and how the tools it provides can be used outside their original purpose of representing knowledge bases.

In our opinion, the main advantage of knowledge compilation is that it offers a comfortable framework to show new tractability results. It can often be used by slightly adapting the data structures to the needed application, such as in Chapter 4 where we introduce non-binary domains to be closer to what is needed. It can also be used by using existing tools after having encoded the problem as a Boolean function, as in Chapter 5, where we use this kind of reduction to show tractability results for the Binary Optimization Problems (BPO). In both cases, transforming the input into a circuit allows one not only to show the tractability of the original problem but also to extend it in several directions that would need extra work with a dedicated algorithm. For example, in the case of databases, the Yannakakis algorithm, originally devised for model checking, once revisited through the lens of compilation, directly offers insight on why constant delay enumeration, direct access or counting is tractable on acyclic conjunctive queries. These three extensions of the original algorithm are straightforward on the circuits but took time to mature. Similarly, in the case of BPO, we have seen that once the multilinear set is compiled, we can work directly on the circuit to take into account cardinality constraints, or enumerate the k best solutions, contributions that would have been possible without the knowledge compilation setting but would have required an intimate understanding of the optimization algorithm to adapt them. This also allows us to extract extended formulations directly, which showcases the modularity of the approach. Many tractability results for BPO were first obtained by an ad-hoc polynomial time algorithm before being generalized as the construction of an extended formulation, for example [DD22] proves tractability of β-acyclic instances whereas [DK23] gives extended formulations.

Another advantage of knowledge compilation is that it can be used as a formal setting to better understand the runtime of specific algorithms. Indeed, if we know that the trace of an algorithm implicitly builds some representation of the input, then any lower bound on the representation size translates to a lower bound on the runtime of the algorithm. For example, exhaustive DPLL has originally been described as an algorithm for #SAT [San+04] but it quickly became clear that a run of the algorithm actually builds a decision-DNNF circuit representing the input formula [HD05]. Hence, if the smallest decision-DNNF circuit representing the satisfying assignments of the CNF formula F has size s, we know that exhaustive

DPLL will run in time at least $\Omega(s)$. This only provides a lower bound on the running time of exhaustive DPLL. Indeed, in practice, not every decision-DNNF circuit can be constructed by this algorithm since caching can only happen for syntactic reasons. Using a formal setting to model the behavior of an algorithm is not specific to knowledge compilation and it is a long tradition in proof complexity to study this kind of question, by trying to formalize a proof system matching the complexity of a solver. The approach from Chapter 3 draws from this idea and allows to get a finer understanding of the real complexity of exhaustive DPLL by modeling the behavior of this algorithm more faithfully. Indeed, some decision-DNNF circuit computing a CNF formula F may never be found by exhaustive DPLL as it comes with its own restrictions, such as syntactic caching. Interestingly, recent work by Berkholz and Vinall-Smeeth [BV25] proves lower bounds on the size of the relational circuit representing the answer set of conjunctive queries depending on their fractional hypertree width which directly transfers to a lower bound on the runtime of Yannakakis or Exhaustive DPLL by the results of Chapter 4. Another recent result suggests that if one only wants to decide whether a query has a model, building the circuit is not optimal and such algorithms can be sped up by taking advantage of matrix multiplication [KHS25]. We have a similar phenomenon in Chapter 5 where some tractability results rely on purely algebraic restrictions of the input polynomial, and for which it is not clear whether knowledge compilation can help.

Last but not least, the communities of knowledge compilation and model counting maintain tools such as D4 [LM17], SDD [Dar11] or SharpSAT-TD [KJ21], capable of exploiting non-trivial structure in the input allowing them to solve complex tasks. Similarly to SAT solvers, their efficiency can be quite surprising and they sometimes beat dedicated solvers for other problems as long as the encoding into propositional logic is sufficiently enough. This manuscript tries to collect examples where the connection between knowledge compilation and other fields of computer science have interesting and sometimes deep applications in theory, but it also suggests that this connection could also be exploited in practice. Take for example the BPO problem from Chapter 5. There is a straightforward translation of the problem to integer linear programming. Trying to solve a problem of bounded treewidth with a MILP solver often does not perform well because, for historical reasons, these tools prioritize algebraic approaches over the combinatorial ones. On the other hand, many knowledge compilation tools will have good performances on such instances because they have the ideal structure. It suggests that hybridizing techniques may be of practical value.

All these reasons shape the research directions we currently pursue. One axis that is particularly appealing is to make the tools from knowledge compilation easier to use. A notorious example is the PySAT library[4] which offers an API to many SAT solvers directly into Python, allowing to access different solvers in a unified way. This approach has been partly implemented for knowledge compilation with the PySDD library[5] which allows to manipulate SDD directly into Python by wrapping the SDD compiler with a Python API. Having such libraries but for other tools from knowledge compilation, ideally with a unified API, would encourage adoption of these tools, and foster richer interactions with other fields of computer science.

[4] https://pysathq.github.io/

[5] https://pysdd.readthedocs.io/en/latest/

The new data structure presented in Chapter 3 has strong potential for practical use. We are planning to implement a bottom-up compiler building TDDs. We need to refine the procedure that allows us to apply a clause to a given deterministic TDD and find good data structures to efficiently identify and contract twins.

For more theoretical investigation, getting a better understanding of proof systems for #SAT solvers may help us gain to better understanding of how such solvers could be improved. [BHK25] suggests that moving from syntactic to semantic caching may be an interesting direction as it would allow solvers to discover smaller representations. Such caching strategies seem costly to implement in practice suggesting that there is still room for improvement in finding other caching strategies, better than syntactic but cheaper than the fully semantic ones.

Bibliography

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach.* 1st. New York, NY, USA: Cambridge University Press, 2009 (cit. on p. 20).

[AC24] Antoine Amarilli and Florent Capelli. "Tractable Circuits in Database Theory". In: *SIGMOD Rec.* 53.2 (2024), pp. 6–20. DOI: 10.1145/3685980.3685982. URL: https://doi.org/10.1145/3685980.3685982 (cit. on pp. 10, 107).

[ACF10] Jean Marc Astesana, Laurent Cosserat, and Hélene Fargier. "Constraint-based vehicle configuration: A case study". In: *Tools with Artificial Intelligence (ICTAI).* Vol. 1. IEEE. 2010, pp. 68–75 (cit. on p. xiv).

[AGG07] Isolde Adler, Georg Gottlob, and Martin Grohe. "Hypertree width and related hypergraph invariants". In: *European Journal of Combinatorics* 28.8 (2007), pp. 2167–2181 (cit. on p. 119).

[AGM13] Albert Atserias, Martin Grohe, and Dániel Marx. "Size bounds and query plans for relational joins". In: *SIAM Journal on Computing* 42.4 (2013), pp. 1737–1767 (cit. on p. 120).

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases.* Vol. 8. Addison-Wesley Reading, 1995 (cit. on p. 101).

[Ake78] Akers. "Binary decision diagrams". In: *IEEE Transactions on computers* 100.6 (1978), pp. 509–516 (cit. on p. xi).

[Ama+17] Antoine Amarilli et al. "A Circuit-Based Approach to Efficient Enumeration". In: *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, Warsaw, Poland, July 10-14, 2017.* Ed. by Ioannis Chatzigiannakis et al. Vol. 80. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 111:1–111:15. DOI: 10.4230/LIPICS.ICALP.2017.111. URL: https://doi.org/10.4230/LIPIcs.ICALP.2017.111 (cit. on pp. 20, 178).

[Ama+20] Antoine Amarilli et al. "Connecting Knowledge Compilation Classes and Width Parameters". In: *Theory Comput. Syst.* 64.5 (2020), pp. 861–914. DOI: 10.1007/S00224-019-09930-2. URL: https://doi.org/10.1007/s00224-019-09930-2 (cit. on pp. xv, 31, 58, 61, 66).

[Ama+24] Antoine Amarilli et al. “Ranked Enumeration for MSO on Trees via Knowledge Compilation”. In: *27th International Conference on Database Theory, ICDT 2024, Paestum, Italy, March 25-28, 2024*. Ed. by Graham Cormode and Michael Shekelyan. Vol. 290. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024, 25:1–25:18. DOI: 10.4230/LIPICS.ICDT.2024.25. URL: https://doi.org/10.4230/LIPIcs.ICDT.2024.25 (cit. on pp. 20, 108, 149).

[ANR15] Mahmoud Abo Khamis, Hung Q Ngo, and Atri Rudra. “FAQ: questions asked frequently”. In: *arXiv preprint arXiv:1504.04044* (2015) (cit. on pp. 131, 133).

[AR11] Michael Alekhnovich and Alexander Razborov. “Satisfiability, Branch-Width and Tseitin tautologies”. In: *Computational Complexity* 20.4 (Nov. 2011), pp. 649–678 (cit. on pp. 29, 57).

[Are+21a] Marcelo Arenas et al. “#NFA Admits an FPRAS: Efficient Enumeration, Counting, and Uniform Generation for Logspace Classes”. In: *J. ACM* 68.6 (2021), 48:1–48:40. DOI: 10.1145/3477045. URL: https://doi.org/10.1145/3477045 (cit. on pp. 9, 22).

[Are+21b] Marcelo Arenas et al. “When is approximate counting for conjunctive queries tractable?” In: *STOC ’21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021*. Ed. by Samir Khuller and Virginia Vassilevska Williams. ACM, 2021, pp. 1015–1027. DOI: 10.1145/3406325.3451014. URL: https://doi.org/10.1145/3406325.3451014 (cit. on p. 22).

[Are+22] Marcelo Arenas et al. *Database Theory*. Open source at https://github.com/pdm-book/community, 2022 (cit. on p. 101).

[Bag+08] Guillaume Bagan et al. “Computing the jth solution of a first-order query”. In: *RAIRO-Theoretical Informatics and Applications* 42.1 (2008), pp. 147–164 (cit. on pp. xvii, 147).

[Bar+14] Anicet Bart et al. “Symmetry-Driven Decision Diagrams for Knowledge Compilation”. In: *ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22 August 2014, Prague, Czech Republic - Including Prestigious Applications of Intelligent Systems (PAIS 2014)*. Ed. by Torsten Schaub, Gerhard Friedrich, and Barry O’Sullivan. Vol. 263. Frontiers in Artificial Intelligence and Applications. IOS Press, 2014, pp. 51–56. DOI: 10.3233/978-1-61499-419-0-51. URL: https://doi.org/10.3233/978-1-61499-419-0-51 (cit. on pp. 24, 27).

[BBM21] Olaf Beyersdorff, Joshua Blinkhorn, and Meena Mahajan. “Building strategies into QBF proofs”. In: *Journal of Automated Reasoning* 65.1 (2021), pp. 125–154 (cit. on p. xvi).

[BCM15] Johann Brault-Baron, Florent Capelli, and Stefan Mengel. “Understanding Model Counting for beta-acyclic CNF-formulas”. In: *32nd International Symposium on Theoretical Aspects of Computer Science*. Vol. 30. LIPIcs. Schloss Dagstuhl, 2015, pp. 143–156 (cit. on pp. 99, 135, 138, 140, 145).

[BCM22] Karl Bringmann, Nofar Carmeli, and Stefan Mengel. "Tight Fine-Grained Bounds for Direct Access on Join Queries". In: *Proceedings of the 41st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 2022, pp. 427–436 (cit. on pp. xvii, 99, 131, 132, 147).

[BDG07] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. "On Acyclic Conjunctive Queries and Constant Delay Enumeration". In: *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL, Lausanne, Switzerland, September 11-15, 2007, Proceedings*. Ed. by Jacques Duparc and Thomas A. Henzinger. Vol. 4646. Lecture Notes in Computer Science. Springer, 2007, pp. 208–222. DOI: 10.1007/978-3-540-74915-8_18. URL: https://doi.org/10.1007/978-3-540-74915-8%5C_18 (cit. on pp. xvii, 97, 122, 144, 145).

[Bea+13] Paul Beame et al. "Lower Bounds for Exact Model Counting and Applications in Probabilistic Databases". In: *Proceedings of the Twenty-Ninth Conference on Uncertainty in Artificial Intelligence*. 2013 (cit. on pp. 17, 47).

[Bea+14] Paul Beame et al. "Counting of Query Expressions: Limitations of Propositional Methods". In: *Proc. 17th International Conference on Database Theory (ICDT)*. 2014, pp. 177–188 (cit. on p. 17).

[Bee+83] Catriel Beeri et al. "On the Desirability of Acyclic Database Schemes". In: *J. ACM* 30.3 (July 1983), pp. 479–513 (cit. on p. 105).

[Ber87] J. Bernasconi. "Low autocorrelation binary sequences: statistical mechanics and configuration space analysis". In: *J. Physique* 141.48 (1987), pp. 559–567 (cit. on p. 168).

[Bes+21] Ksenia Bestuzheva et al. *The SCIP Optimization Suite 8.0*. Technical Report. Optimization Online, Dec. 2021. URL: http://www.optimization-online.org/DB_HTML/2021/12/8728.html (cit. on p. 169).

[Bey+24] Olaf Beyersdorff et al. "The relative strength of #SAT proof systems". In: *27th International Conference on Theory and Applications of Satisfiability Testing (SAT 2024)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. 2024, pp. 5–1 (cit. on pp. 69, 92–94).

[BH02] E. Boros and P.L. Hammer. "Pseudo-Boolean optimization". In: *Discrete applied mathematics* 123.1 (2002), pp. 155–225 (cit. on p. 168).

[BHK25] Olaf Beyersdorff, Tim Hoffmann, and Kaspar Kasche. "Proof Systems That Tightly Characterise Model Counting Algorithms". In: *Electron. Colloquium Comput. Complex.* TR25-127 (2025). ECCC: TR25-127. URL: https://eccc.weizmann.ac.il/report/2025/127 (cit. on pp. xiv, 69, 89, 96, 181).

[BHS24] Olaf Beyersdorff, Tim Hoffmann, and Luc Nicolas Spachmann. "Proof Complexity of Propositional Model Counting". In: *J. Satisf. Boolean Model. Comput.* 15.1 (2024), pp. 27–59. DOI: 10.3233/{SAT}-231507. URL: https://doi.org/10.3233/sat-231507 (cit. on pp. xiv, xvi, 69, 83, 84).

[Bla+18] Jasmin Christian Blanchette et al. "A Verified SAT Solver Framework with Learn, Forget, Restart, and Incrementality". In: *J. Autom. Reason.* 61.1-4 (2018), pp. 333–365. DOI: 10.1007/S10817-018-9455-7. URL: https://doi.org/10.1007/s10817-018-9455-7 (cit. on p. 67).

[BLM07] María Luisa Bonet, Jordi Levy, and Felip Manyà. "Resolution for Max-SAT". In: *Artificial Intelligence* 171.8–9 (June 2007), pp. 606–618 (cit. on pp. xvi, 96).

[BM18] Daniel Bienstock and Gonzalo Muñoz. "LP formulations for polynomial optimization problems". In: *SIAM Journal on Optimization* 28.2 (2018), pp. 1121–1150 (cit. on p. 174).

[BN01] Aharon Ben-Tal and Arkadi Nemirovski. "On polyhedral approximations of the second-order cone". In: *Mathematics of Operations Research* 26.2 (2001), pp. 193–205. DOI: 10.1287/moor.26.2.193.10561 (cit. on p. 154).

[Bod93] Hans 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. ACM, 1993, pp. 226–234 (cit. on p. 61).

[Bon+18] Maria Luisa Bonet et al. "MaxSAT resolution with the dual rail encoding". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 32. 1. 2018 (cit. on p. xvi).

[Bou+22] Pierre Bourhis et al. "Pseudo Polynomial-Time Top-k Algorithms for d-DNNF Circuits". In: *CoRR* abs/2202.05938 (2022). arXiv: 2202.05938. URL: https://arxiv.org/abs/2202.05938 (cit. on pp. 149, 155, 175).

[Bov+15] Simone Bova et al. "On Compiling CNFs into Structured Deterministic DNNFs". In: *Theory and Applications of Satisfiability Testing*. Lecture Notes in Computer Science. Springer International Publishing, Sept. 2015, pp. 199–214 (cit. on pp. xv, 12, 29–31, 54, 57, 61, 66, 173).

[Bov+16] Simone Bova et al. "Knowledge Compilation Meets Communication Complexity". In: *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*. Ed. by Subbarao Kambhampati. IJCAI/AAAI Press, 2016, pp. 1008–1014. URL: http://www.ijcai.org/Abstract/16/147 (cit. on pp. xiv, 1, 17, 23, 25, 96, 178).

[Bov16] Simone Bova. "SDDs Are Exponentially More Succinct than OBDDs". In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*. Ed. by Dale Schuurmans and Michael P. Wellman. AAAI Press, 2016, pp. 929–935. DOI: 10.1609/AAAI.V30I1.10107. URL: https://doi.org/10.1609/aaai.v30i1.10107 (cit. on p. 51).

[Bra12] Johann 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*. Vol. 16. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl, 2012, pp. 137–151 (cit. on p. 135).

[Bra13] Johann Brault-Baron. "De la pertinence de l'énumération : complexité en logiques". Thèse de doctorat dirigée par Grandjean, Étienne Informatiquei Caen 2013. PhD thesis. Université de Caen Normandie, 2013, 1 vol. (VII–157 p.) URL: http://www.theses.fr/2013CAEN2056 (cit. on pp. 135, 137, 139, 145).

[Bra14] Johann Brault-Baron. "Hypergraph Acyclicity Revisited". In: *ArXiv e-prints* (Mar. 2014) (cit. on p. 106).

[Bry+25] Randal E. Bryant et al. "Certified Knowledge Compilation with Application to Formally Verified Model Counting". In: *J. Artif. Intell. Res.* 82 (2025). DOI: 10.1613/JAIR.1.15958. URL: https://doi.org/10.1613/jair.1.15958 (cit. on pp. xvi, 24, 69, 91, 94, 95).

[Bry86] Randal E Bryant. "Graph-based algorithms for boolean function manipulation". In: *Computers, IEEE Transactions on* 100.8 (1986), pp. 677–691 (cit. on pp. xi, 5).

[Bry92] Randal E Bryant. "Symbolic Boolean manipulation with ordered binary-decision diagrams". In: *ACM Computing Surveys* 24 (1992), pp. 293–318 (cit. on pp. xi, 5, 30).

[BS17] Simone Bova and Stefan Szeider. "Circuit Treewidth, Sentential Decision, and Query Compilation". In: *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*. Ed. by Emanuel Sallinger, Jan Van den Bussche, and Floris Geerts. ACM, 2017, pp. 233–246. DOI: 10.1145/3034786.3034787. URL: https://doi.org/10.1145/3034786.3034787 (cit. on pp. 42, 52, 53, 58).

[Bür00] Peter Bürgisser. *Completeness and Reduction in Algebraic Complexity Theory*. Vol. 7. Algorithms and Computation in Mathematics. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000. (Visited on 12/26/2015) (cit. on p. 175).

[BV23] Christoph Berkholz and Harry Vinall-Smeeth. "A Dichotomy for Succinct Representations of Homomorphisms". In: *50th International Colloquium on Automata, Languages, and Programming, ICALP 2023, Paderborn, Germany, July 10-14, 2023*. Ed. by Kousha Etessami, Uriel Feige, and Gabriele Puppis. Vol. 261. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 113:1–113:19. DOI: 10.4230/LIPICS.ICALP.2023.113. URL: https://doi.org/10.4230/LIPIcs.ICALP.2023.113 (cit. on p. 124).

[BV25] Christoph Berkholz and Harry Vinall-Smeeth. "Factorised Representations of Join Queries: Tight Bounds and a New Dichotomy". In: *CoRR* abs/2503.20438 (2025). DOI: 10.48550/ARXIV.2503.20438. arXiv: 2503.20438. URL: https://doi.org/10.48550/arXiv.2503.20438 (cit. on pp. 124, 180).

[Cap+24] Florent Capelli et al. "Linear Programs with Conjunctive Database Queries". In: *Log. Methods Comput. Sci.* 20.1 (2024). DOI: 10.46298/LMCS-20(1:9)2024. URL: https://doi.org/10.46298/lmcs-20(1:9)2024 (cit. on pp. 150, 178).

[Cap+26] Florent Capelli et al. *Direct Access for Conjunctive Queries with Negations*. 2026. arXiv: 2310.15800 [cs.DB]. URL: https://arxiv.org/abs/2310.15800 (cit. on p. 129).

[Cap16] Florent Capelli. "Structural restrictions of CNF-formulas: applications to model counting and knowledge compilation". PhD thesis. Université Paris Diderot, Sorbonne Paris Cité, 2016. URL: https://florent.capelli.me/publi/these_capelli.pdf (cit. on pp. 17, 18, 26, 29, 30, 161).

[Cap17] Florent Capelli. "Understanding the complexity of#SAT using knowledge compilation". In: *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE. 2017, pp. 1–10 (cit. on pp. 30, 54, 99, 135, 138, 171, 173).

[Cap19] Florent Capelli. "Knowledge Compilation Languages as Proof Systems". In: *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*. Ed. by Mikolás Janota and Inês Lynce. Vol. 11628. Lecture Notes in Computer Science. Springer, 2019, pp. 90–99. DOI: 10.1007/978-3-030-24258-9_6. URL: https://doi.org/10.1007/978-3-030-24258-9%5C_6 (cit. on pp. xiv, xvi, 69, 74, 92, 96).

[Car+20] Nofar Carmeli et al. "Answering (unions of) conjunctive queries using random access and random-order enumeration". In: *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 2020, pp. 393–409 (cit. on pp. xvii, 147).

[Car+23] Nofar Carmeli et al. "Tractable Orders for Direct Access to Ranked Answers of Conjunctive Queries". In: *ACM Transactions on Database Systems* (Jan. 2023). DOI: 10.1145/3578517. URL: https://doi.org/10.1145/3578517 (cit. on pp. xvii, 98, 99, 147).

[CCT87] William Cook, Collette R Coullard, and Gy Turán. "On the complexity of cutting-plane proofs". In: *Discrete Applied Mathematics* 18.1 (1987), pp. 25–38 (cit. on pp. 68, 70).

[CCZ10] Michele Conforti, Gérard Cornuéjols, and Giacomo Zambelli. "Extended formulations in combinatorial optimization". In: *4OR* 8.1 (2010), pp. 1–48 (cit. on pp. 154, 178).

[CCZ14] Michele Conforti, Gérard Cornuéjols, and Giacomo Zambelli. "Integer programming models". In: *Integer Programming*. Springer, 2014, pp. 45–84 (cit. on pp. 151, 154, 159).

[CD13] Arthur Choi and Adnan Darwiche. "Dynamic Minimization of Sentential Decision Diagrams". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 27.1 (June 2013), pp. 187–194. DOI: 10.1609/aaai.v27i1.8690. URL: https://ojs.aaai.org/index.php/AAAI/article/view/8690 (cit. on p. 50).

[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*. 2014, pp. 399–414 (cit. on p. viii).

[CDS96] Marco Cadoli, Francesco M Donini, and Marco Schaerf. "Is intractability of non-monotonic reasoning a real drawback?" In: *Artificial intelligence* 88.1-2 (1996), pp. 215–251 (cit. on p. xiv).

[CEI96] Matthew Clegg, Jeffery Edmonds, and Russell Impagliazzo. "Using the Groebner basis algorithm to find proofs of unsatisfiability". In: *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*. 1996, pp. 174–183 (cit. on pp. 68, 70).

[Che04] Hubie Chen. "Quantified constraint satisfaction and bounded treewidth". In: *ECAI*. Vol. 16. Citeseer. 2004, p. 161 (cit. on pp. xv, 30, 61, 65).

[CHJ90] Yves Crama, Pierre Hansen, and Brigitte Jaumard. "The Basic Algorithm For Pseudo-Boolean Programming Revisited". In: *Discrete Applied Mathematics* 29 (1990), pp. 171–185 (cit. on p. 169).

[CI24] Florent Capelli and Oliver Irwin. "Direct Access for Conjunctive Queries with Negations". In: *27th International Conference on Database Theory, ICDT 2024, Paestum, Italy, March 25-28, 2024*. Ed. by Graham Cormode and Michael Shekelyan. Vol. 290. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024, 13:1–13:20. DOI: 10.4230/LIPICS.ICDT.2024.13. URL: https://doi.org/10.4230/LIPIcs.ICDT.2024.13 (cit. on pp. xvii, 98, 99, 107, 125, 130–134, 139, 140, 143, 147, 148).

[CIS25] Florent Capelli, Oliver Irwin, and Sylvain Salvati. "A Simple Algorithm for Worst Case Optimal Join and Sampling". In: *28th International Conference on Database Theory, ICDT 2025, Barcelona, Spain, March 25-28, 2025*. Ed. by Sudeepa Roy and Ahmet Kara. Vol. 328. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025, 23:1–23:19. DOI: 10.4230/LIPICS.ICDT.2025.23. URL: https://doi.org/10.4230/LIPIcs.ICDT.2025.23 (cit. on p. 121).

[CK18] Nofar Carmeli and Markus Kröll. "Enumeration Complexity of Conjunctive Queries with Functional Dependencies". In: *21st International Conference on Database Theory, ICDT 2018, Vienna, Austria, March 26-29, 2018*. Ed. by Benny Kimelfeld and Yael Amsterdamer. Vol. 98. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 11:1–11:17. DOI: 10.4230/LIPICS.ICDT.2018.11. URL: https://doi.org/10.4230/LIPIcs.ICDT.2018.11 (cit. on p. 148).

[CLM21] Florent Capelli, Jean-Marie Lagniez, and Pierre Marquis. "Certifying Top-Down Decision-DNNF Compilers". In: *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*. AAAI Press, 2021, pp. 6244–6253. DOI: 10.1609/AAAI.V35I7.16776. URL: https://doi.org/10.1609/aaai.v35i7.16776 (cit. on pp. xvi, 69, 79, 82, 89–91).

[CM19] Florent Capelli and Stefan Mengel. "Tractable QBF by knowledge compilation". In: *36th International Symposium on Theoretical Aspects of Computer Science (STACS 2019)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. 2019, pp. 18–1 (cit. on pp. 30, 31, 66).

[Cod70] Edgar F Codd. "A relational model of data for large shared data banks". In: *Communications of the ACM* 13.6 (1970), pp. 377–387 (cit. on p. 106).

[Coo71] Stephen A Cook. "The complexity of theorem-proving procedures". In: *Proceedings of the third annual ACM symposium on Theory of computing*. ACM. 1971, pp. 151–158 (cit. on pp. xiii, 21).

[Cor+22] Thomas H Cormen et al. *Introduction to algorithms*. MIT press, 2022 (cit. on p. 115).

[CPG23] Florent Capelli, Alberto Del Pia, and Silvia Di Gregorio. "A Knowledge Compilation Take on Binary Polynomial Optimization". In: *CoRR* abs/2311.00149 (2023). DOI: 10.48550/ARXIV.2311.00149. arXiv: 2311.00149. URL: https://doi.org/10.48550/arXiv.2311.00149 (cit. on pp. xiv, xviii, 149, 150, 171, 174).

[Cpl09] IBM ILOG Cplex. "V12. 1: User's Manual for CPLEX". In: *International Business Machines Corporation* 46.53 (2009), p. 157 (cit. on p. 169).

[CR79] Stephen A Cook and Robert A Reckhow. "The relative efficiency of propositional proof systems". In: *The journal of symbolic logic* 44.1 (1979), pp. 36–50 (cit. on pp. 68, 70).

[Cro23] Nicolas Crosetti. "Enhancing and solving linear programs with conjunctive queries. (Enrichir et résoudre des programmes linéaires avec des requêtes conjonctives)". PhD thesis. University of Lille, France, 2023. URL: https://tel.archives-ouvertes.fr/tel-04064287 (cit. on pp. 150, 178).

[CRZ20] Clément Carbonnel, Miguel Romero, and Stanislav Zivný. "Point-Width and Max-CSPs". In: *ACM Trans. Algorithms* 16.4 (2020), 54:1–54:28. DOI: 10.1145/3409447. URL: https://doi.org/10.1145/3409447 (cit. on p. 140).

[CS19] Florent Capelli and Yann Strozecki. "Incremental delay enumeration: Space and time". In: *Discret. Appl. Math.* 268 (2019), pp. 179–190. DOI: 10.1016/J.DAM.2018.06.038. URL: https://doi.org/10.1016/j.dam.2018.06.038 (cit. on p. 18).

[CS21] Florent Capelli and Yann Strozecki. "Enumerating models of DNF faster: Breaking the dependency on the formula size". In: *Discret. Appl. Math.* 303 (2021), pp. 203–215. DOI: 10.1016/J.DAM.2020.02.014. URL: https://doi.org/10.1016/j.dam.2020.02.014 (cit. on p. 18).

[CS23] Florent Capelli and Yann Strozecki. "Geometric Amortization of Enumeration Algorithms". In: *40th International Symposium on Theoretical Aspects of Computer Science, STACS 2023, Hamburg, Germany, March 7-9, 2023*. Ed. by Petra Berenbrink et al. Vol. 254. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 18:1–18:22. DOI: 10.4230/LIPICS.STACS.2023.18. URL: https://doi.org/10.4230/LIPIcs.STACS.2023.18 (cit. on p. 18).

[Dan79] E Dantsin. "Parameters defining the time of tautology recognition by the splitting method". In: *Semiotics and information science* 12 (1979), pp. 8–17 (cit. on pp. 29, 57).

[Dar01] Adnan Darwiche. "On the Tractable Counting of Theory Models and its Application to Truth Maintenance and Belief Revision". In: *Journal of Applied Non-Classical Logics* 11.1-2 (2001), pp. 11–34 (cit. on p. 9).

[Dar04] Adnan Darwiche. "New Advances in Compiling CNF into Decomposable Negation Normal Form". In: *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004*. 2004, pp. 328–332 (cit. on pp. 29, 57).

[Dar11] Adnan Darwiche. "SDD: A New Canonical Representation of Propositional Knowledge Bases". In: *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*. Ed. by Toby Walsh. IJCAI/AAAI, 2011, pp. 819–826. DOI: 10.5591/978-1-57735-516-8/IJCAI11-143. URL: https://doi.org/10.5591/978-1-57735-516-8/IJCAI11-143 (cit. on pp. xiv, 24, 27, 49, 51, 57, 180).

[DD21] Alberto Del Pia and Silvia Di Gregorio. "Chvátal rank in binary polynomial optimization". In: *INFORMS Journal on Optimization* 3.4 (2021), pp. 315–349 (cit. on p. 171).

[DD22] Alberto Del Pia and Silvia Di Gregorio. "On the complexity of binary polynomial optimization over acyclic hypergraphs". In: *Proceedings of SODA 2022*. 2022, pp. 2684–2699 (cit. on pp. 169, 171, 179).

[DD23] Alberto Del Pia and Silvia Di Gregorio. "On the complexity of binary polynomial optimization over acyclic hypergraphs". In: *Algorithmica* 85 (2023), pp. 2189–2213 (cit. on pp. 138, 150, 169, 171).

[Die12] Reinhard Diestel. *Graph Theory, 4th Edition*. Vol. 173. Graduate texts in mathematics. Springer, 2012. ISBN: 978-3-642-14278-9 (cit. on p. 4).

[DK17] Alberto Del Pia and Aida Khajavirad. "A polyhedral study of binary polynomial programs". In: *Mathematics of Operations Research* 42.2 (2017), pp. 389–410 (cit. on pp. xix, 150, 170, 174).

[DK18] Alberto Del Pia and Aida Khajavirad. "The Multilinear polytope for acyclic hypergraphs". In: *SIAM Journal on Optimization* 28.2 (2018), pp. 1049–1076. DOI: 10.1137/16M1095998 (cit. on pp. 150, 174).

[DK23] Alberto Del Pia and Aida Khajavirad. "A polynomial-size extended formulation for the multilinear polytope of beta-acyclic hypergraphs". In: *Mathematical Programming, Series A* (2023) (cit. on pp. 150, 174, 179).

[DK24] Alberto Del Pia and Aida Khajavirad. "The pseudo-Boolean polytope and polynomial-size extended formulations for binary polynomial optimization". In: *Mathematical Programming, Series A* (2024) (cit. on pp. 150, 174).

[DM02] Adnan Darwiche and Pierre Marquis. "A Knowledge Compilation Map". In: *Journal of Artificial Intelligence Research* 17 (2002), pp. 229–264 (cit. on pp. xiv, xvii, 16, 23).

[DM15] Arnaud Durand and Stefan Mengel. "Structural Tractability of Counting of Solutions to Conjunctive Queries". In: *Theory Comput. Syst.* 57.4 (2015), pp. 1202–1249. DOI: 10.1007/S00224-014-9543-Y. URL: https://doi.org/10.1007/s00224-014-9543-y (cit. on p. 122).

[DP60] Martin Davis and Hilary Putnam. "A Computing Procedure for Quantification Theory". In: *J. ACM* 7.3 (July 1960), pp. 201–215 (cit. on pp. 68, 70).

[DPV20] Jeffrey Dudek, Vu Phan, and Moshe Vardi. "ADDMC: weighted model counting with algebraic decision diagrams". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 34. 2020, pp. 1468–1476 (cit. on p. 27).

[EG77] Jack Edmonds and R. Giles. "A min-max relation for submodular functions on graphs". In: *Annals of Discrete Mathematics* 1 (1977), pp. 185–204 (cit. on p. 154).

[Fag83] R. Fagin. "Degrees of acyclicity for hypergraphs and relational database schemes". In: *Journal of the ACM* 30.3 (1983), pp. 514–550 (cit. on p. 106).

[FF56] Lester R Ford Jr and Delbert R Fulkerson. "Maximal flow through a network". In: *Canadian journal of Mathematics* 8 (1956), pp. 399–404 (cit. on p. 159).

[FGP18] Wolfgang Fischl, Georg Gottlob, and Reinhard Pichler. "General and fractional hypertree decompositions: Hard and easy cases". In: *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. ACM. 2018, pp. 17–32 (cit. on pp. 119, 122).

[FHR22] Johannes Klaus Fichte, Markus Hecher, and Valentin Roland. "Proofs for Propositional Model Counting". In: *25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022, August 2-5, 2022, Haifa, Israel*. Ed. by Kuldeep S. Meel and Ofer Strichman. Vol. 236. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 30:1–30:24. DOI: 10.4230/LIPICS.{SAT}.2022.30. URL: https://doi.org/10.4230/LIPIcs.%7BSAT%7D.2022.30 (cit. on pp. xvi, 69, 82, 83).

[Fic+18] Johannes K Fichte et al. "An SMT approach to fractional hypertree width". In: *Principles and Practice of Constraint Programming: 24th International Conference, Lille, France, August 27-31, 2018, Proceedings 24*. Springer. 2018, pp. 109–127 (cit. on p. 131).

[FK06] Tomás Feder and Phokion G Kolaitis. "Closures and dichotomies for quantified constraints". In: *Electronic Colloquium on Computational Complexity, Report TR06-160*. 2006 (cit. on p. 65).

[FMR08] E. Fischer, J.A. Makowsky, and E.V. Ravve. "Counting truth assignments of formulas of bounded tree-width or clique-width". In: *Discrete Applied Mathematics* 156.4 (2008), pp. 511–529 (cit. on pp. xv, 29).

[FW20] Mathias Fleury and Christoph Weidenbach. "A Verified SAT Solver Framework including Optimization and Partial Valuations". In: *LPAR 2020: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Alicante, Spain, May 22-27, 2020*. Ed. by Elvira Albert and Laura Kovács. Vol. 73. EPiC Series in Computing. EasyChair, 2020, pp. 212–229. DOI: 10.29007/96WB. URL: https://doi.org/10.29007/96wb (cit. on p. 67).

[Gan+22] Robert Ganian et al. "Threshold treewidth and hypertree width". In: *Journal of Artificial Intelligence Research* 74 (2022), pp. 1687–1713 (cit. on p. 131).

[GJS76] M.R. Garey, D.S. Johnson, and L. Stockmeyer. "Some simplified NP-Complete graph problems". In: *Theoretical Computer Science* (1976), pp. 237–267 (cit. on p. 168).

[GLS99] Georg Gottlob, Nicola Leone, and Francesco Scarcello. "Hypertree decompositions and tractable queries". In: *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM. 1999, pp. 21–32 (cit. on pp. 98, 116, 118).

[GM14] Martin Grohe and Dániel Marx. "Constraint solving via fractional edge covers". In: *ACM Transactions on Algorithms (TALG)* 11.1 (2014), p. 4 (cit. on pp. 98, 120, 140).

[GMS09] Georg Gottlob, Zoltán Miklós, and Thomas Schwentick. "Generalized Hypertree Decompositions: NP-hardness and Tractable Variants". In: *J. ACM* 56.6 (Sept. 2009) (cit. on p. 119).

[Got+12] Georg Gottlob et al. "Size and Treewidth Bounds for Conjunctive Queries". In: *J. ACM* 59.3 (2012), 16:1–16:35. DOI: 10.1145/2220357.2220363. URL: https://doi.org/10.1145/2220357.2220363 (cit. on p. 121).

[GP04] G. Gottlob and R. Pichler. "Hypergraphs in Model Checking: Acyclicity and Hypertree-Width versus Clique-Width". In: *SIAM Journal on Computing* 33.2 (2004) (cit. on p. 138).

[GS17] Robert Ganian and Stefan Szeider. "New Width Parameters for Model Counting". In: *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*. Ed. by Serge Gaspers and Toby Walsh. Vol. 10491. Lecture Notes in Computer Science. Springer, 2017, pp. 38–52. DOI: 10.1007/978-3-319-66263-3_3. URL: https://doi.org/10.1007/978-3-319-66263-3%5C_3 (cit. on pp. xv, 29).

[Gur23] Gurobi Optimization, LLC. *Gurobi Optimizer Reference Manual*. 2023. URL: https://www.gurobi.com (cit. on p. 169).

[Hak85] Armin Haken. "The intractability of resolution". In: *Theoretical Computer Science*. Third Conference on Foundations of Software Technology and Theoretical Computer Science 39 (1985), pp. 297–308 (cit. on p. 68).

[Has86] John Hastad. "Almost optimal lower bounds for small depth circuits". In: *Proceedings of the eighteenth annual ACM symposium on Theory of computing*. 1986, pp. 6–20 (cit. on p. 17).

[HD05] Jinbo Huang and Adnan Darwiche. "DPLL with a Trace: From SAT to Knowledge Compilation". In: *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*. 2005, pp. 156–162 (cit. on pp. viii, xviii, 30, 68, 78, 124, 179).

[HI98] Kazuyoshi Hayase and Hiroshi Imai. "OBDDs of a monotone function and its prime implicants". In: *Theory of Computing Systems* 31.5 (1998), pp. 579–591 (cit. on p. 47).

[HK56] A.J. Hoffman and J.B. Kruskal. "Integral boundary points of convex polyhedra". In: *Linear Inequalities and Related Systems* (1956). Ed. by H.W. Kuhn and A.W. Tucker, pp. 223–246 (cit. on pp. 153, 159).

[htt] holf (https://cstheory.stackexchange.com/users/42948/holf). *Treewidth relations between Boolean formulas and Tseitin encodings*. Theoretical Computer Science Stack Exchange. URL:https://cstheory.stackexchange.com/q/51345 (version: 2022-04-19). eprint: https://cstheory.stackexchange.com/q/51345. URL: https://cstheory.stackexchange.com/q/51345 (cit. on p. 58).

[Ish11] Hiroshi Ishikawa. "Transformation of general binary MRF minimization to the first-order case". In: *IEEE Trans. Pattern Anal. Mach. Intell.* 33.6 (2011), pp. 1234–1249. DOI: 10.1109/TPAMI.2010.91 (cit. on p. 168).

[JGM13] Mikoláš Janota, Radu Grigore, and Joao Marques-Silva. "On QBF proofs and preprocessing". In: *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer. 2013, pp. 473–489 (cit. on p. xvi).

[JPR16] Manas R Joglekar, Rohan Puttagunta, and Christopher Ré. "Ajar: Aggregations and joins over annotated relations". In: *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 2016, pp. 91–106 (cit. on pp. xvii, 98).

[Juk12] Stasys Jukna. *Boolean Function Complexity - Advances and Frontiers*. Vol. 27. Algorithms and combinatorics. Springer, 2012 (cit. on p. 94).

[Jus+07] Toni Jussila et al. "A first step towards a unified proof checker for QBF". In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2007, pp. 201–214 (cit. on p. xvi).

[Kar+25] Ahmet Kara et al. "Conjunctive queries with free access patterns under updates". In: *Logical Methods in Computer Science* 21 (2025) (cit. on p. 131).

[Kar84] Narendra Karmarkar. "A new polynomial-time algorithm for linear programming". In: *Proceedings of the sixteenth annual ACM symposium on Theory of computing*. 1984, pp. 302–311 (cit. on p. 152).

[Kha+24] Mahmoud Abo Khamis et al. "Join Size Bounds using l_p-Norms on Degree Sequences". In: *Proc. ACM Manag. Data* 2.2 (2024), p. 96. DOI: 10.1145/3651597. URL: https://doi.org/10.1145/3651597 (cit. on p. 121).

[Kha79] Leonid Genrikhovich Khachiyan. "A polynomial algorithm in linear programming". In: *Doklady Akademii Nauk*. Vol. 244. Russian Academy of Sciences. 1979, pp. 1093–1096 (cit. on p. 152).

[KHS25] Mahmoud Abo Khamis, Xiao Hu, and Dan Suciu. "Fast Matrix Multiplication meets the Submodular Width". In: *Proc. ACM Manag. Data* 3.2 (2025), 98:1–98:26. DOI: 10.1145/3725235. URL: https://doi.org/10.1145/3725235 (cit. on p. 180).

[KJ21] Tuukka Korhonen and Matti Järvisalo. "Integrating Tree Decompositions into Decision Heuristics of Propositional Model Counters". In: *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*. Ed. by Laurent D. Michel. Vol. 210. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 8:1–8:11. ISBN: 978-3-95977-211-2. DOI: 10.4230/LIPIcs.CP.2021.8. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CP.2021.8 (cit. on p. 180).

[KKT20] Petr Kolman, Martin Koutecký, and Hans Raj Tiwary. "Extension Complexity, MSO Logic, and Treewidth". In: *Discret. Math. Theor. Comput. Sci.* 22.4 (2020). DOI: 10.23638/DMTCS-22-4-8. URL: https://doi.org/10.23638/DMTCS-22-4-8 (cit. on p. 178).

[KLM89] Richard M Karp, Michael Luby, and Neal Madras. "Monte-Carlo approximation algorithms for enumeration problems". en. In: *Journal of Algorithms* 10.3 (Sept. 1989), pp. 429–448. ISSN: 01966774. DOI: 10.1016/0196-6774(89)90038-2. URL: https://linkinghub.elsevier.com/retrieve/pii/0196677489900382 (visited on 11/12/2021) (cit. on pp. 21, 22).

[KNR16] Mahmoud Abo Khamis, Hung Q Ngo, and Atri Rudra. "FAQ: questions asked frequently". In: *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 2016, pp. 13–28 (cit. on pp. xvii, 98, 131).

[KNS16] Mahmoud Abo Khamis, Hung Q. Ngo, and Dan Suciu. "Computing Join Queries with Functional Dependencies". In: *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. Ed. by Tova Milo and Wang-Chiew Tan. ACM, 2016, pp. 327–342. DOI: 10.1145/2902251.2902289. URL: https://doi.org/10.1145/2902251.2902289 (cit. on p. 124).

[KNS17] Mahmoud Abo Khamis, Hung Q. Ngo, and Dan Suciu. "What Do Shannon-type Inequalities, Submodular Width, and Disjunctive Datalog Have to Do with One Another?" In: *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May*

14-19, 2017. Ed. by Emanuel Sallinger, Jan Van den Bussche, and Floris Geerts. ACM, 2017, pp. 429–444. DOI: 10.1145/3034786.3056105. URL: https://doi.org/10.1145/3034786.3056105 (cit. on p. 121).

[KNS25] Mahmoud Abo Khamis, Hung Q. Ngo, and Dan Suciu. "PANDA: Query Evaluation in Submodular Width". In: *TheoretiCS* 4 (2025). DOI: 10.46298/THEORETICS.25.12. URL: https://doi.org/10.46298/theoretics.25.12 (cit. on pp. 121, 124, 148).

[Kor+13] Frédéric Koriche et al. "Knowledge Compilation for Model Counting: Affine Decision Trees." In: *IJCAI*. 2013, pp. 947–953 (cit. on p. 27).

[Kor+16] Frédéric Koriche et al. "Fixed-Parameter Tractable Optimization Under DNNF Constraints". In: *ECAI 2016 - 22nd European Conference on Artificial Intelligence, 29 August-2 September 2016, The Hague, The Netherlands - Including Prestigious Applications of Artificial Intelligence (PAIS 2016)*. Ed. by Gal A. Kaminka et al. Vol. 285. Frontiers in Artificial Intelligence and Applications. IOS Press, 2016, pp. 1194–1202. DOI: 10.3233/978-1-61499-672-9-1194. URL: https://doi.org/10.3233/978-1-61499-672-9-1194 (cit. on p. xviii).

[KR87] Johan de Kleer and Raymond Reiter. "Foundations for assumption-based truth maintenance systems: Preliminary report". In: *Proc. American Assoc. for Artificial Intelligence Nat. Conf.* 1987, pp. 183–188 (cit. on p. xi).

[Kra19] Jan Krajıéček. *Proof complexity*. Vol. 170. Cambridge University Press, 2019 (cit. on p. 70).

[KS91] Henry Kautz and Bart Selman. "A general framework for knowledge compilation". In: *International Workshop on Processing Declarative Knowledge*. Springer. 1991, pp. 287–300 (cit. on p. xiii).

[KS92] Henry A. Kautz and Bart Selman. "Forming Concepts for Fast Inference". In: *Proceedings of the 10th National Conference on Artificial Intelligence, San Jose, CA, USA, July 12-16, 1992*. Ed. by William R. Swartout. AAAI Press / The MIT Press, 1992, pp. 786–793. URL: http://www.aaai.org/Library/AAAI/1992/aaai92-122.php (cit. on p. xiv).

[KVD17] Angelika Kimmig, Guy Van den Broeck, and Luc De Raedt. "Algebraic model counting". In: *Journal of Applied Logic* 22 (2017), pp. 46–62 (cit. on pp. xvii, xviii, 22, 149, 155).

[Lan23] Matthias Lanzinger. "Tractability beyond β-acyclicity for conjunctive queries with negation and SAT". In: *Theoretical Computer Science* 942 (2023), pp. 276–296 (cit. on pp. 134, 139, 140).

[Lau09] M. Laurent. "Sums of Squares, Moment Matrices and Optimization Over Polynomials". In: *Emerging Applications of Algebraic Geometry*. Vol. 149. The IMA Volumes in Mathematics and its Applications. Springer, 2009, pp. 157–270 (cit. on p. 174).

[Lee59] Chang-Yeong Lee. “Representation of switching circuits by binary-decision programs”. In: *The Bell System Technical Journal* 38.4 (1959), pp. 985–999 (cit. on p. xi).

[Ler06] Xavier Leroy. “Formal certification of a compiler back-end, or: programming a compiler with a proof assistant”. In: *33rd ACM symposium on Principles of Programming Languages*. ACM Press, 2006, pp. 42–54. URL: http://xavierleroy.org/publi/compiler-certif.pdf (cit. on p. 67).

[Lev73] Leonid A Levin. “Universal sequential search problems”. In: *Problemy Peredachi Informatsii* 9.3 (1973), pp. 115–116 (cit. on pp. xiii, 21).

[Lib04] Leonid Libkin. *Elements of finite model theory*. Vol. 41. Springer, 2004 (cit. on p. 101).

[Lie+10] Frauke Liers et al. “A non-disordered glassy model with a tunable interaction range”. In: *Journal of Statistical Mechanics: Theory and Experiment* 2010.05 (May 2010), p. L05003. DOI: 10.1088/1742-5468/2010/05/L05003. URL: https://dx.doi.org/10.1088/1742-5468/2010/05/L05003 (cit. on p. 168).

[LM17] Jean-Marie Lagniez and Pierre Marquis. “An Improved Decision-DNNF Compiler”. In: *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*. Ed. by Carles Sierra. ijcai.org, 2017, pp. 667–673. DOI: 10.24963/IJCAI.2017/93. URL: https://doi.org/10.24963/ijcai.2017/93 (cit. on pp. xiv, xvi, 69, 180).

[Mar10] Dániel Marx. “Approximating fractional hypertree width”. In: *ACM Trans. Algorithms* 6.2 (2010), 29:1–29:17. DOI: 10.1145/1721837.1721845. URL: https://doi.org/10.1145/1721837.1721845 (cit. on p. 122).

[Mar13] Dániel Marx. “Tractable Hypergraph Properties for Constraint Satisfaction and Conjunctive Queries”. In: *J. ACM* 60.6 (Nov. 2013) (cit. on p. 124).

[MC25] Kuldeep S. Meel and Alexis de Colnet. “An FPRAS for Model Counting for Non-Deterministic Read-Once Branching Programs”. In: *28th International Conference on Database Theory, ICDT 2025, Barcelona, Spain, March 25-28, 2025*. Ed. by Sudeepa Roy and Ahmet Kara. Vol. 328. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025, 30:1–30:21. DOI: 10.4230/LIPICS.ICDT.2025.30. URL: https://doi.org/10.4230/LIPIcs.ICDT.2025.30 (cit. on pp. 9, 23).

[MC26] Kuldeep S Meel and Alexis de Colnet. “#CFG and#DNNF admit FPRAS”. In: *Proceedings of the 2026 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM. 2026, pp. 5978–6010 (cit. on p. 23).

[McD82] John McDermott. “R1: A rule-based configurer of computer systems”. In: *Artificial Intelligence* 19.1 (Sept. 1982), pp. 39–88. ISSN: 0004-3702. DOI: 10.1016/0004-3702(82)90021-2. URL: http://dx.doi.org/10.1016/0004-3702(82)90021-2 (cit. on p. xi).

[Mon20] Mikaël Monet. "Solving a Special Case of the Intensional vs Extensional Conjecture in Probabilistic Databases". In: *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2020, Portland, OR, USA, June 14-19, 2020*. Ed. by Dan Suciu, Yufei Tao, and Zhewei Wei. ACM, 2020, pp. 149–163. DOI: 10.1145/3375395.3387642. URL: https://doi.org/10.1145/3375395.3387642 (cit. on p. 24).

[MRC90] R. Kipp Martin, Ronald L. Rardin, and Brian A. Campbell. "Polyhedral characterization of discrete dynamic programming". In: *Operations Research* 38.1 (1990), pp. 127–138 (cit. on pp. 154, 157, 160).

[MS24] Meena Mahajan and Gaurav Sood. "QBF merge resolution is powerful but unnatural". In: *Logical Methods in Computer Science* 20 (2024) (cit. on p. 71).

[MW20] Stefan Mengel and Romain Wallon. "Revisiting Graph Width Measures for CNF-Encodings". In: *J. Artif. Intell. Res.* 67 (2020), pp. 409–436. DOI: 10.1613/JAIR.1.11750. URL: https://doi.org/10.1613/jair.1.11750 (cit. on pp. 25, 58).

[Ngo+12] Hung Q. Ngo et al. "Worst-case optimal join algorithms: [extended abstract]". In: *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2012, Scottsdale, AZ, USA, May 20-24, 2012*. Ed. by Michael Benedikt, Markus Krötzsch, and Maurizio Lenzerini. ACM, 2012, pp. 37–48. DOI: 10.1145/2213556.2213565. URL: https://doi.org/10.1145/2213556.2213565 (cit. on p. 121).

[Ngo+18] Hung Q Ngo et al. "Worst-case optimal join algorithms". In: *Journal of the ACM (JACM)* 65.3 (2018), pp. 1–40. DOI: 10.1145/3180143 (cit. on p. 121).

[Ngo18] Hung Q. Ngo. "Worst-Case Optimal Join Algorithms: Techniques, Results, and Open Problems". In: *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. SIGMOD/PODS '18: International Conference on Management of Data. Houston TX USA: ACM, May 27, 2018, pp. 111–124. ISBN: 978-1-4503-4706-8. DOI: 10.1145/3196959.3196990 (cit. on p. 121).

[OD17] Umut Oztok and Adnan Darwiche. "On Compiling DNNFs without Determinism". In: *CoRR* abs/1709.07092 (2017). arXiv: 1709.07092. URL: http://arxiv.org/abs/1709.07092 (cit. on p. 111).

[Olt16] Dan Olteanu. "Factorized Databases: A Knowledge Compilation Perspective". In: *Beyond NP, Papers from the 2016 AAAI Workshop, Phoenix, Arizona, USA, February 12, 2016*. Ed. by Adnan Darwiche. Vol. WS-16-05. AAAI Technical Report. AAAI Press, 2016. URL: http://www.aaai.org/ocs/index.php/WS/AAAIW16/paper/view/12638 (cit. on pp. xvii, 111).

[OPS13] S. Ordyniak, D. Paulusma, and S. Szeider. "Satisfiability of acyclic and almost acyclic CNF formulas". In: *Theoretical Computer Science* 481 (2013), pp. 85–99 (cit. on pp. xv, 29).

[OZ12] Dan Olteanu and Jakub Zavodny. “Factorised representations of query results: size bounds and readability”. In: *15th International Conference on Database Theory, ICDT ’12, Berlin, Germany, March 26-29, 2012*. Ed. by Alin Deutsch. ACM, 2012, pp. 285–298. DOI: 10.1145/2274576.2274607. URL: https://doi.org/10.1145/2274576.2274607 (cit. on p. 109).

[OZ15] Dan Olteanu and Jakub Závodný. “Size Bounds for Factorised Representations of Query Results”. en. In: *ACM Transactions on Database Systems* 40.1 (Mar. 2015), pp. 1–44. ISSN: 03625915. (Visited on 07/25/2018) (cit. on pp. xvii, 98, 99, 109, 111, 112, 124, 144, 145).

[Par03] Bernard Pargamin. “Extending cluster tree compilation with non-boolean variables in product configuration: A tractable approach to preference-based configuration”. In: *Proceedings of the IJCAI*. Vol. 3. Citeseer. 2003 (cit. on p. xiv).

[PD10] Knot Pipatsrisawat and Adnan Darwiche. “Top-Down Algorithms for Constructing Structured DNNF: Theoretical and Practical Implications.” In: *ECAI*. 2010, pp. 3–8 (cit. on pp. 14, 29, 57).

[PD11] Knot Pipatsrisawat and Adnan Darwiche. “On the power of clause-learning SAT solvers as resolution engines”. In: *Artificial intelligence* 175.2 (2011), pp. 512–525 (cit. on p. 68).

[PS13] Reinhard Pichler and Sebastian Skritek. “Tractable counting of the answers to conjunctive queries”. In: *Journal of Computer and System Sciences*. JCSS Foundations of Data Management 79.6 (Sept. 2013), pp. 984–1001 (cit. on pp. 98, 112, 122, 145, 146).

[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*. 2013, pp. 55–66 (cit. on pp. xv, 29, 30).

[PT87] Robert Paige and Robert E Tarjan. “Three partition refinement algorithms”. In: *SIAM Journal on Computing* 16.6 (1987), pp. 973–989 (cit. on p. 106).

[Raz14] Igor Razgon. “No Small Nondeterministic Read-Once Branching Programs for CNFs of Bounded Treewidth”. In: *Parameterized and Exact Computation - 9th International Symposium, IPEC*. 2014, pp. 319–331 (cit. on pp. 17, 47).

[Rot96] D. Roth. “On the hardness of approximate reasoning”. In: *Artificial Intelligence* 82.1–2 (1996), pp. 273–302 (cit. on p. 21).

[San+04] Tian Sang et al. “Combining Component Caching and Clause Learning for Effective Model Counting.” In: *Theory and Applications of Satisfiability Testing* 4 (2004), 7th (cit. on pp. xviii, 29, 30, 57, 98, 124, 179).

[Sau03] Martin Sauerhoff. “Approximation of boolean functions by combinatorial rectangles”. In: *Theor. Comput. Sci.* 1-3.301 (2003), pp. 45–78 (cit. on p. 17).

[Sch09] M Schroeder. *Number theory in Science and Communication*. 5th ed. Springer-Verlag Berlin Heidelberg, 2009 (cit. on p. 168).

[Sha38] Claude E Shannon. “A symbolic analysis of relay and switching circuits”. In: *Electrical Engineering* 57.12 (1938), pp. 713–723 (cit. on p. xi).

[Shi+19] Andy Shih et al. “Smoothing Structured Decomposable Circuits”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach et al. Vol. 32. Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper_files/paper/2019/file/940392f5f32a7ade1cc201767cf83e31-Paper.pdf (cit. on p. 16).

[SM73] Larry J Stockmeyer and Albert R Meyer. “Word problems requiring exponential time (preliminary report)”. In: *Proceedings of the fifth annual ACM symposium on Theory of computing*. 1973, pp. 1–9 (cit. on p. 65).

[Som98] Fabio Somenzi. “CUDD: CU decision diagram package release 2.3. 0”. In: *University of Colorado at Boulder* 621 (1998) (cit. on p. 52).

[SS10] M. Samer and S. Szeider. “Algorithms for propositional model counting”. In: *Journal of Discrete Algorithms* 8.1 (2010), pp. 50–64 (cit. on pp. xv, 29, 30, 57).

[SS13] F. Slivovsky and S. Szeider. “Model Counting for Formulas of Bounded Clique-Width”. In: *Algorithms and Computation - 24th International Symposium, ISAAC*. 2013, pp. 677–687 (cit. on pp. xv, 29, 30).

[Str10] Y. Strozecki. “Enumeration complexity and matroid decomposition”. PhD thesis. Université Paris Diderot - Paris 7, 2010 (cit. on p. 18).

[Str23] Yann Strozecki. “Enumeration complexity: incremental time, delay and space”. In: *arXiv preprint arXiv:2309.17042* (2023) (cit. on p. 18).

[STV14] S. Hortemo Sæther, J.A. Telle, and M. Vatshelle. “Solving MaxSAT and #SAT on Structured CNF Formulas”. In: *Theory and Applications of Satisfiability Testing*. 2014, pp. 16–31 (cit. on pp. 30, 61).

[Suc23] Dan Suciu. “Applications of Information Inequalities to Database Theory Problems”. In: *38th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2023, Boston, MA, USA, June 26-29, 2023*. IEEE, 2023, pp. 1–30. DOI: 10.1109/LICS56636.2023.10175769. URL: https://doi.org/10.1109/LICS56636.2023.10175769 (cit. on p. 121).

[Sze04] Stefan Szeider. “On fixed-parameter tractable parameterizations of SAT”. In: *Theory and Applications of Satisfiability, 6th International Conference*. Ed. by Enrico Giunchiglia and Armando Tacchella. Vol. 2919. LNCS. Springer, 2004, pp. 188–202 (cit. on p. 29).

[Tse83] Grigori S Tseitin. “On the complexity of derivation in propositional calculus”. In: *Automation of reasoning: 2: Classical papers on computational logic 1967–1970* (1983), pp. 466–483 (cit. on p. xiii).

[TY84] Robert E. Tarjan and Mihalis Yannakakis. “Simple Linear-time Algorithms to Test Chordality of Graphs, Test Acyclicity of Hypergraphs, and Selectively Reduce Acyclic Hypergraphs”. In: *SIAM J. Comput.* 13.3 (July 1984), pp. 566–579 (cit. on pp. 105, 106).

[Urq87] Alasdair Urquhart. “Hard Examples for Resolution”. In: *J. ACM* 34.1 (Jan. 1987), pp. 209–219. (Visited on 11/26/2015) (cit. on p. 68).

[Val79] L. Valiant. “The Complexity of Enumeration and Reliability Problems”. In: *SIAM Journal on Computing* 8.3 (Aug. 1979), pp. 410–421. (Visited on 01/28/2016) (cit. on p. 20).

[Vat12] M. Vatshelle. “New Width Parameters of Graphs”. PhD thesis. University of Bergen, 2012 (cit. on p. 29).

[VD15] Guy Van den Broeck and Adnan Darwiche. “On the role of canonicity in knowledge compilation”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 29. 2015 (cit. on pp. 24, 51, 57, 66).

[Vel14] Todd Veldhuizen. “Triejoin: A Simple, Worst-Case Optimal Join Algorithm”. In: *Proceedings of the 17th International Conference on Database Theory (ICDT), Athens, Greece, 2014* 17.13 (Mar. 2014). Ed. by Nicole Schweikardt, Vassilis Christophides, and Vincent Leroy, pp. 96–106. DOI: 10.5441/002/ICDT.2014.13 (cit. on p. 121).

[Vin24] Harry Vinall-Smeeth. “Structured d-DNNF Is Not Closed under Negation”. In: *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI 2024, Jeju, South Korea, August 3-9, 2024*. ijcai.org, 2024, pp. 3593–3601. URL: https://www.ijcai.org/proceedings/2024/398 (cit. on pp. xiv, 24, 26).

[Weg00] Ingo Wegener. *Branching Programs and Binary Decision Diagrams*. SIAM, 2000 (cit. on pp. 5, 6, 10, 16, 47, 48).

[WHH14] Nathan Wetzler, Marijn JH Heule, and Warren A Hunt Jr. “DRAT-trim: Efficient checking and trimming using expressive clausal proofs”. In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2014, pp. 422–429 (cit. on pp. 68, 91).

[WJ04] M.J. Wainwright and M.I. Jordan. *Treewidth-Based Conditions for Exactness of the Sherali-Adams and Lasserre Relaxations*. Tech. rep. 671. University of California, 2004 (cit. on p. 174).

[Yan81] Mihalis Yannakakis. “Algorithms for Acyclic Database Schemes”. In: *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*. VLDB Endowment, 1981, pp. 82–94 (cit. on pp. xvii, 97, 104, 112).

[Yan91] Mihalis Yannakakis. “Expressing Combinatorial Optimization Problems by Linear Programs”. In: *JOURNAL OF COMPUTER AND SYSTEM SCIENCES* 43 (1991), p. 441466. DOI: 10.1016/0022-0000(91)90024-Y (cit. on p. 177).

[Zha+24] Hangdong Zhao et al. “Conjunctive Queries with Negation and Aggregation: A Linear Time Characterization”. In: *Proc. ACM Manag. Data* 2.2 (2024), p. 75. DOI: 10.1145/3651138. URL: https://doi.org/10.1145/3651138 (cit. on pp. 135, 139, 142).

Extended CV

Career Summary and education

2023–Present **Assistant professor (Chaire de professeur Junior)**, *Université d'Artois*, Lens.

2017–2023 **Assistant professor**, *Université de Lille*, Lille.

2016–2017 **Postdoctoral position with Igor Razgon**, *Birkbeck College*, London.

2013–2016 **PhD under the supervision of Arnaud Durand**, *Université Paris Diderot*, Paris.

"Structural restriction of CNF-formulas : application to model counting and knowledge compilation", June 2016.

Supported by a national grant.

2012–2013 **"Agrégation de mathématiques"**, *ENS Lyon*, Lyon.

French competitive exam for teachers in high schools and universities. Specialized in Computer Science. Rank : 15/320.

2010–2012 **Master degree in Theoretical Computer Science, with highest honors, ranked first**, *ENS Lyon*, Lyon.

Publications

Peer-reviewed international journal papers

2026 Enumeration Theory through the Lens of Database Challenges, with Nofar Carmeli, Alessio Conte, Benny Kimelfeld, Reinhard Pichler, Nikolaos Tziavelis. *PODS Companion 26 : Companion of the 45th Symposium on Principles of Database Systems*

2026 Direct Access for Conjunctive Queries with Negations, with Nofar Carmeli, Oliver Irwin, Sylvain Salvati. *Logical Methods in Computer Science (LMCS)*

2024 Tractable Circuits in Database Theory , with Antoine Amarilli *ACM SIGMOD Record*

2024 Linear Programs with Conjunctive Database Queries , with Nicolas Crosetti Joachim Niehren Jan Ramon *Logical Methods in Computer Science (LMCS)*

2021 *Enumerating models of DNF faster : breaking the dependency on the formula size*, with Yann Strozecki, *Discrete Applied Mathematics (DAM), Volume 303*

2019 *Connecting Knowledge Compilation Classes and Width Parameters*, with Antoine Amarilli, Mikaël Monet, and Pierre Senellart, *Theory Of Computing Systems (TOCS 19)*

2019 *Counting Minimal Transversals of β-Acyclic Hypergraphs*, with Benjamin Bergougnoux and Mamadou Kanté, *Journal of Computer and System Sciences, 2019 (JCSS 19)*

2019 *Incremental delay enumeration : Space and time*, with Yann Strozecki, *Discrete Applied Mathematics, Volume 268*

2015 *The Arithmetic Complexity of Tensor Contraction*, with Arnaud Durand and Stefan Mengel, *Theory of Computing Systems, STACS Special Issue, 2015 (TOCS 15)*

Peer-reviewed international conference papers

2026 A canonical generalization of OBDD, with YooJung Choi, Stefan Mengel, Martín Muñoz, Guy Van den Broeck. *Theory and Applications of Satisfiability Testing (SAT 26)*

2026 Building Relational Circuits (paper for the invited ICDT Lecture)

2025 Dynamic direct access of MSO query evaluation over strings , with Pierre Bourhis Stefan Mengel Cristian Riveros *International Conference on Database Theory (ICDT 25)*

2025 A Simple Algorithm for Worst-Case Optimal Join and Sampling , with Oliver Irwin Sylvain Salvati *International Conference on Database Theory (ICDT 25)*

2024 Direct Access for Conjunctive Queries with Negation , with Oliver Irwin *International Conference on Database Theory (ICDT 24)* , **Best Newcomer Paper Award**

2024 Ranked Enumeration for MSO on Trees via Knowledge Compilation , with Antoine Amarilli Pierre Bourhis Mikaël Monet *International Conference on Database Theory (ICDT 24)*

2024 A Top-Down Tree Model Counter for Quantified Boolean Formulas , with Jean-Marie Lagniez Andreas Plank Martina Seidl *IJCAI*

2023 Geometric Amortization of Enumeration Algorithms , with Yann Strozecki *Symposium on Theoretical Aspects of Computer Science (STACS 23)*

2022 *Linear Programs With Conjunctive Queries*, with Nicolas Crosetti, Joachim Niehren and Jan Ramon : *International Conference on Database Theory (ICDT 22)*, **Best Newcomer Paper Award**

2021 *Certifying Top-Down Decision-DNNF Compilers.*, with Jean-Marie Lagniez, and Pierre Marquis *Thirty-Fifth AAAI Conference on Artificial Intelligence (AAAI 21)*

2019 *Knowledge compilation languages as proof systems*, *Theory and Applications of Satisfiability Testing, 2019 (SAT 19)*

2019 *Tractable QBF by Knowledge Compilation*, with Stefan Mengel, *36th Symposium on Theoretical Aspects of Computer Science, 2019 (STACS 19)*

2017 *Understanding the Complexity of* #SAT *using Knowledge Compilation*, *Symposium on Logic in Computer Science, 2017 (LICS 17)*

2016 *Knowledge Compilation Meets Communication Complexity*, with Simone Bova, Stefan Mengel, Friedrich Slivovsky, *International Joint Conference on Artificial Intelligence, 2016 (IJCAI 16)*

2015 *On Compiling CNFs into Structured Deterministic DNNF*, with Simone Bova, Stefan Mengel, Friedrich Slivovsky, *Theory and Applications of Satisfiability Testing, 2015 (SAT 15)*

2015 *Understanding Model Counting for β-acyclic CNF-formulas*, with Johann Brault-Baron and Stefan Mengel, *32th Symposium on Theoretical Aspects of Computer Science, 2015 (STACS 15)*

2014 *Hypergraph Acyclicity and Propositional Model Counting*, with Arnaud Durand and Stefan Mengel, *Theory and Applications of Satisfiability Testing, 2014 (SAT 14)*

2013 *The Complexity of Tensor Contraction*, with Arnaud Durand and Stefan Mengel, *30th Symposium on Theoretical Aspects of Computer Science, 2013 (STACS 13)*

Preprints

2023 A Knowledge Compilation Take on Binary Polynomial Optimization , with Silvia Di Gregorio Alberto Del Pia *To appear in Mathematical Programming (MAPR)*

Research activities

Projects and fundings

2023-2026 **Project Investigator** of the CPJ funding **TRUKC** on proof systems for Knowledge Compilation

2020-2026 **Project Investigator** of the **ANR JCJC KCODA** on Knowledge Compilation

2025- Postdoc grant for *PIA4 MAIA*

2025- Postdoc grant for *CPER CornelIA*

2026- Project member of **ANR CERADOC**

2019-2023 Project member of **ANR PING/ACK**

2018-2020 Project member of **SACRe KOCOON**.

2016-2021 Project member of **ANR HEADWORK**

2014-2019 Project member of **ANR AGGREG**

Administrative duties

Since 2025 **Member of the scientific comittee of Interstice**

2021-2024 **Member of CER Inria Lille (Commission des Emplois de Recherche)**

2018-2020 **Responsible of GT IM-IA**, a working group at the interface between GDR IM (Computer Science and Mathematics) and GDR IA (Artificial Intelligence).

Program committees

2027 ***International Conference on Database Theory***.

2018, 2019, 2021, 2022, 2023, 2025 ***International Joint Conference on Artificial Intelligence***, *IJCAI*, Distinguished PC Member.

2024 ***International Joint Conference on Automated Reasoning***.

2021 ***AAAI Conference on Artificial Intelligence***.

2020 ***International Conference on Theory and Applications of Satisfiability Testing.***.

2018 ***International Workshop on Quantified Boolean Formulas and Beyond***.

Peer-review activities

2026 ICDT (Outstanding reviewer award), SAT, DAM

2025 STACS,PODS,SAT,IJCAI,ESA,IPEC

2024 STACS,SAT,ICDT,IJCAR,Constraints, JAIR, DAM,

2023 STOC, PODS, IJCAI, SAT, MFCS, AMAI,

2022 IJCAI, PODS, STACS

2021 STACS, IJCAI, AAAI, ARTINT, TODS

2020 TALG, STACS, SAT, AAAI, JAIR

2019 IPEC, TOCS, STACS, IJCAI, ISAAC, TOCT, LICS, LATA

2018 TOCS, STACS, MSCS, IJCAI, ISAAC, SAT, LICS, JSAT, PODS, MFCS

2017 CSR, STACS, SAT, LICS

2016 ICDT

Scientific event organisation

2022 **TUDASTIC 2022**, 1st edition of the thematic school TUtorials on DAta Structures for Text Indexation and Compression, with Camille Marchet, Charles Paperman, Lille.

2019 **International workshop** on Knowledge Compilation in Arras, SACRe Project **Kocoon** with Pierre Bourhis, Pierre Marquis and Stefan Mengel.

2018 **GT ALGA days** of GDR IM in Lille.

2018 **Graph and Constraints** workshop during CP 2018 conference in Lille.

2017 **École de Recherche** *An overview of Knowledge Compilation* à l'ENS Lyon.

PhD Supervisions

December 2022 – March 2026 — **Oliver Irwin**, *Branching and Circuits : Algorithmic Techniques for Efficient Query Evaluation*. Co-supervised with Sylvain Salvati. ICDT 2024 best new comer award.

September 2018 – February 2022 — **Nicolas Crosetti**, *Enhancing and solving linear programs with conjunctive queries*. Co-supervised with Joachim Niehren, Sophie Tison and Jan Ramon. ICDT 2022 best new comer award.

Other Supervisions

2022 — **Master thesis** of Oliver Irwin (Université de Lille) : *Approximation schemes for counting*

2020 — **Master thesis** of Claire Soyez-Martin (Université de Lille) : *Hypergraph decompositions for #SAT*

June/July 2019 — **Third-year internship** of Fanny Canivet (ENS Ulm) : *Proof systems based on restricted Boolean circuits*

February 2019 — **Third-year internship** of Georges Lebellier (Central Lille) : *Incompleteness in databases*

2018 — **Master thesis** of Nicolas Crosetti : *Dependency Weighted Aggregation*

2017 — **Project** of Nicolas Crosetti : *Yannakakis Algorithm and Factorized databases*

PhD Commitee

2022 — Commitee Member for the PhD of Alexis de Colnet, Université d'Artois.

2017 — Commitee Member for the PhD of Mikaël Monet, Telecom Paris.

Teaching activities

Teaching Experience

Since 2024 — **Junior Professor**, *Université d'Artois*, UFR des Sciences.

- *An introduction to AI*, 2nd year students in Biology, Physics or Chemistry
- *An introduction to Python*, 2nd year students in Biology, Physics or Chemistry
- *Logic*, 1st year of Master
- *An introduction to networks*, 1st year students

Since 2017 — **Maître de conférences (Assistant Professor)**, *Université de Lille*, UFR des Langues Étrangères Appliquées.

- *Algorithm and Their Complexity*, for Master students (Centrale Lille and Université de Lille)
- *Website creation in Wordpress*, for Master students
- *Website creation in HTML/CSS*, for third year students
- *Libreoffice Writer et Calc*, for second year students
- *Access*, for Master students

2013–2016 — **Teaching assistant**, *Université Paris Diderot*, Paris.

Preparation of the exercice sheets and supervision of the exercice sessions in the following topics :

- *History of computation* for first year math and computer science students
- *Databases* for third year students in social science
- *C programming and algorithmic* for math graduate students with a programming project

2015 — Participation to recreational scientific presentations in primary school, Paris.

2010–2011 — **Oral exam training in mathematics**, *Lycée Fénelon*, Paris.

For undergraduate students

Other teaching duties

August 2026 **Reasoning Web Summer School 2026 : *Knowledge Compilation : power and limits***, Vilus, Lithuania.

Since 2023 CAPES NSI jury (French exam to become high school teacher in computer science).

December 2017 **Research school *An overview of Knowledge Compilation***, *ENS Lyon*, Lyon.
with Jean-Marie Lagniez et Pierre Marquis

Administrative duties

2020-2022 • Responsible for **first year students** : roughly 800 students.

2018 • Responsible for **organising teaching sessions and mentoring** for first year students learning difficulties.

2018-2021 • Elected member of the LEA department council.

2018-2022 • Responsible for **Parcoursup jury**.