RICE UNIVERSITY

# Planning and Execution for Discrete Integration

by

**Jeffrey M. Dudek**

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

**Doctor of Philosophy**

Approved, Thesis Committee:

Moshe Vardi
Moshe Vardi (Jul 9, 2021 08:59 CDT)

Moshe Y. Vardi, Chair
Karen Ostrum George Distinguished
Service Professor in Computational
Engineering

Devika Subramanian

Devika Subramanian
Professor of Computer Science and
Electrical Engineering

[illegible]
Illya Hicks (Jul 10, 2021 10:07 CDT)

Illya V. Hicks
Professor of Computational and Applied
Mathematics

[illegible]
Leonardo Dueñas-Osorio (Jul 11, 2021 12:02 CDT)

Leonardo Dueñas-Osorio
Professor of Civil and Environmental
Engineering

Houston, Texas
August, 2021

ABSTRACT

# Planning and Execution for Discrete Integration

by

Jeffrey M. Dudek

Discrete integration is a fundamental problem in artificial intelligence with numerous applications, including probabilistic reasoning, planning, inexact computing, engineering reliability, and statistical physics. The task is to count the total weight, subject to a given weight function, of all solutions to given constraints. Developing tools to compute the total weight for applied problems is an area of active research.

Over the last ten years, hundreds of thousands of research hours have been poured into low-level computational tools and compilers for neural network training and inference. Simultaneously, there has been a surge in high-level reasoning tools based on graph decompositions, spurred by several competitions. While some existing tools for discrete integration (*counters*) tightly integrate with these low-level computational or high-level reasoning tools, no existing counter is able to leverage both together.

In this dissertation, we demonstrate that a clean separation of high-level reasoning (*planning*) and low-level computation (*execution*) leads to scalable and more flexible counters. Instead of building tightly on any particular tool, we target APIs that can be fulfilled by multiple implementations. This requires novel theoretical and algorithmic techniques to use existing high-level reasoning tools in a way consistent with the options available in popular low-level computational libraries. The resulting counters perform well in many hardware settings (singlecore, multicore, GPU).

# Acknowledgments

I want to thank my advisor, Moshe Vardi, who took me in as an undergrad that had only the barest sense of how to do research. He gave me freedom and power to explore my own ideas, yet was always ready with support and guidance when things didn't go as planned. I couldn't have asked for a better mentor than Moshe.

I also want to thank Devika Subramanian, Illya Hicks, and Leonardo Dueñas-Osorio for serving on my thesis committee and for their many excellent suggestions.

I have been very lucky to collaborate with amazing researchers: Leonardo Dueñas-Osorio, Dror Fried, Kuldeep Meel, Vu Phan, and Moshe Vardi. Their ideas and drive were fundamental in everything I did in my PhD. I am also grateful to the entirety of my research group, LAPIS, where I could always find someone to debate an idea, to sharpen endless iterations of presentations, or simply to relax with after work.

I am also grateful to the friends I made along the way– through the QGSA and GradGames, in Duncan Hall and Valhalla, on campus and off. Rice has been my home for 10 years, and I will miss the people the most.

I am especially grateful to my partner, Karl, who supported me from near and far for many long years. I could not be more excited about the future.

Finally, I want to thank my parents and siblings. Without their constant love, support, and understanding, I would not be where I am today.

# Contents

# Illustrations

# Tables

# Chapter 1

# Introduction

Discrete integration is a fundamental problem in artificial intelligence, with applications in probabilistic reasoning, planning, inexact computing, engineering reliability, and statistical physics [1–4]. In discrete integration, the task is to count the total weight, subject to a given weight function, of the solutions of given constraints [3]. If the input constraints are given as a propositional formula over independent variables, the problem is called *model counting*. If dependent variables (i.e., existential variables) are also allowed in the input constraints, the problem is called *projected model counting*. This dissertation primarily focuses on exact techniques for model counting and projected model counting, but other works have considered approximate techniques [5] or other classes of discrete-integration queries (e.g., allowing constraints with alternating quantifiers [6, 7], or allowing continuous variables [8]).

As is common, we restrict our attention to discrete-integration queries where the weight function is *log-linear* (i.e., literal-weighted), which captures a wide variety of probabilistic distributions [9]. We call a discrete-integration query *weighted* if the weight function is log-linear and *unweighted* if the weight function is a constant function (and thus the task is simply to count the number of solutions to the input constraints). Note that unweighted discrete integration is a special case of weighted discrete integration. Unweighted model counting is also called #SAT, while unweighted projected model counting is also called #∃SAT.

Even the simplest case of discrete integration, #SAT, is #P-Complete [10]. In fact, every problem in the polynomial hierarchy can be answered by a single #SAT query [11]. As a simple example, a propositional formula is satisfiable if and only if its unweighted model count is nonzero. Weighted model counting (with rational, log-linear weights) is #P-Complete as well, through a reduction to #SAT that encodes the weights in additional variables [12, 13]. But projected model counting is thought to be harder: #∃SAT is not contained in #P unless the polynomial hierarchy collapses to $\Sigma_2^P$ [14].

Despite the theoretical difficulty of discrete integration, a variety of tools for discrete integration (called *counters*) exist that can handle large sets of constraints and so solve large, useful applied problems. The earliest counters, e.g. `CDP` [15], were based on search. The key idea is to take an algorithm for exploring the entire solution space of a set of constraints (e.g. DPLL [16, 17], and later CDCL [18]) and augment it to enumerate the number of partial solutions encountered. Modern solvers such as `cachet` [19] and `SharpSAT` [20] follow this approach. Another class of counters (e.g. `miniC2D` [21] and `d4` [22]) are based on knowledge compilation, where the idea is to compile the set of constraints into an alternative representation on which a discrete-integration query may be answered in polynomial time. For example, the set of constraints may be compiled into a single binary decision diagram (BDD) [23] or a sentential decision diagram (SDD) [24].

A third class of counters [25–31] are based on dynamic programming. Dynamic programming is a powerful technique that has been applied across computer science [32], including to database-query optimization [33], satisfiability solving [34–36], and quantified Boolean formula (QBF) evaluation [37]. The key idea is to solve a sequence of smaller subproblems, formed by partitioning the input constraints, and then incrementally combine these solutions into the final result. The techniques developed in this dissertation are all based on dynamic programming.

The key idea of this dissertation is that dynamic-programming algorithms for discrete integration can be cleanly separated into two phases: a *planning phase* of high-level reasoning to construct subproblems, followed by an *execution phase* of low-level computation to solve subproblems. Existing dynamic-programming-based counters perform both high-level reasoning and low-level planning, but in an intermixed way that is often tightly coupled to a single existing library. Explicitly separating planning and execution enables us to separately reason about, implement, and optimize each phase. Instead of building tightly on any particular library, we target APIs that can be fulfilled by multiple planning or execution implementations. The resulting counters scale to large problem instances and run flexibly in a variety of hardware environments.

## 1.1 Planning: High-Level Reasoning

In the planning phase, we aim to leverage graph decompositions as a tool for high-level reasoning. *Graph decompositions* [38–41] decompose a graph into subgraphs in such a way that the decomposition can aid algorithms running on the graph. For example, a *tree decomposition* [38, 39] decomposes a graph into a tree structure that captures information on graph cycles. Algorithms based on graph decompositions have been successful across computer science [33, 42–44].

The success of graph-decomposition algorithms in practice relies on finding good decompositions of arbitrary graphs. This has spurred the development of a variety of heuristics and tools for efficiently finding graph decompositions [45–48]. Moreover, recent competitions [49] ensured that many graph decomposition tools can be used through a single unified API and thus, with careful design, can be interchanged.

A variety of dynamic-programming algorithms [50, 51] and counters [25–28] exist for performing discrete integration using graph decompositions. These counters are typically tightly integrated with a a single graph-decomposition tool. For example `gpuSAT2` [27, 28] is a tool for weighted model counting that is tightly integrated with the tree-decomposition solver `htd` [45]. While `gpuSAT2` also includes some low-level computational optimizations, it uses handwritten GPU kernel calls instead of leveraging low-level computational libraries.

## 1.2 Execution: Low-Level Computation

In the execution phase, we aim to leverage two existing classes of low-level computational libraries: tensors and decision diagrams.

*Tensors* are a tool used across quantum physics and computer science for describing and reasoning about quantum systems, big-data processing, and more [52–54]. In particular, tensors are heavily used in neural network training and inference, where the key operation is *tensor contraction*. [55–58]. Consequently, there is massive practical work across machine learning and high-performance computing on tensor contraction [55–58]. This includes a variety of high-performance libraries that can perform tensor contraction [59–61] in a variety of hardware settings (i.e., on GPUs [62, 63] or other specialized hardware [64]) with a single unified API. Several works have applied these libraries towards discrete integration [65, 66], but these approaches are limited to unweighted model counting on a single CPU core.

*Decision diagrams* are a group of data structures used to sparsely represent sets and functions [23, 67–69]. In particular, we aim to use *Algebraic Decision Diagrams (ADDs)* [67] for discrete integration, which have been used in stochastic model checking [70] and stochastic planning [71]. An ADD is a compact representation of a real-valued function as a directed acyclic graph. Moreover, there are several high-performance libraries for efficiently manipulating ADDs [72, 73]. ADDs were used for weighted model counting in `ADDMC` [29, 30], which tied for first place of the weighted track of the 2020 Model Counting Competition [74]. While `ADDMC` also includes some high-level reasoning, it uses tightly-integrated custom-built heuristics [75–78] instead of leveraging existing high-level reasoning tools.

## 1.3 Contributions

The main contribution of this dissertation is a separation of dynamic-programming algorithms for discrete integration into a planning phase of high-level reasoning, followed by an execution phase of low-level computation.

We first introduce this separation in a new model counter, `TensorOrder`, which uses graph decompositions for planning and tensors for execution. This includes a new reduction from weighted model counting to tensor network contraction. We consider two planning algorithms– **LG** and **FT**– based on graph decompositions. While **LG** is based on existing tensor network (and constraint satisfaction) techniques, we contribute a new analysis that more closely matches the memory usage of existing tensor libraries. **FT** is a novel planning algorithm, tailored for constrained counting, that factors tensor networks based on tree decompositions. We evaluate `TensorOrder` and show that **FT** dramatically outperforms **LG** on benchmarks from probabilistic inference. Overall, `TensorOrder` is able to solve many benchmarks solved by no other (exact) counter.

Next, we show that adding this separation improves the existing model counter `ADDMC` [29, 30]. We unify a variety of approaches into a single conceptual framework using project-join trees. We show that replacing the existing constraint-satisfaction planner in `ADDMC` with **LG** (a planner based on graph decompositions) leads to a faster model counter. Moreover, we compare (dense) tensors with (sparse) algebraic decision diagrams in the execution phase and find that algebraic decision diagrams outperform tensors on single CPU cores. Overall, `DPMC` is the fastest weighted model counter on a significant number of benchmarks.

Next, we show that this separation can be exploited to develop parallel model counters. We build a novel, high-performance parallel model counter `TensorOrder2` by separately parallelizing the planning and execution phases in `TensorOrder`. We parallelize the planning phase by introducing an algorithmic portfolio of planners, motivated by success in the SAT community [79]. We parallelize the execution phase through the use of parallel tensor libraries to run on multiple CPU cores, on a GPU, and even on a TPU (i.e., a Tensor Processing Unit [64]). To handle limited-memory environments as on a GPU, we introduce a novel technique for parallel model counting based on variable marginalization. The resulting tool `TensorOrder2` is the fastest parallel weighted model counter on a significant number of benchmarks.

Finally, we apply our separation into planning and execution to the problem of projected model counting in a new projected model counter, `ProCount`. We present a novel algorithm for performing projected model counting through a planning and execution phase using graded project-join trees. As part of this, we introduce a novel planning algorithm that builds graded project-join trees by using a planner for standard project-join trees as a black-box. The resulting tool `ProCount` makes a significant contribution to a portfolio of exact weighted projected model counters.

## 1.4 Tools

The following open source tools have been developed as part of this dissertation:

`TensorOrder` and `TensorOrder2`: https://github.com/vardigroup/TensorOrder

`DPMC` and `ProCount`: https://github.com/vardigroup/DPMC

## 1.5 Organization

The remainder of this dissertation is organized as follows:

Chapter 2 presents notation and background on model counting, existing high-level planning tools (graph decompositions and heuristics from constraint satisfaction), and existing low-level computational tools (tensors and decision diagrams).

Chapter 3 presents an algorithm for weighted model counting based on tensor networks. The resulting tool `TensorOrder` uses graph decompositions for planning and tensors for execution. Most results in this chapter appear in [80].

Chapter 4 presents an algorithm for weighted model counting based on project-join trees. The resulting tool `DPMC` uses graph decompositions for planning and decision diagrams for execution. Most results in this chapter appear in [81].

Chapter 5 parallelizes the techniques from Chapter 3 to run on multiple CPUs, a GPU, and on a TPU. Most results in this chapter appear in [82].

Chapter 6 generalizes the techniques from Chapter 4 to weighted projected model counting. Most results in this chapter appear in [83].

Finally, Chapter 7 summarizes the main contributions of this dissertation and describes several possible future directions.

# Chapter 2

# Preliminaries

In this chapter, we introduce notations and preliminaries needed to present and understand our work.

## 2.1 Boolean Formulas and Pseudo-Boolean Functions

Let $X$ be a set of propositional variables and let $\varphi$ be a Boolean formula defined over $X$. We use $\texttt{Vars}(\varphi)$ to indicate the set of variables of $\varphi$. An *assignment* to $X$ is an element of $2^X$, and further $\tau \in 2^X$ is a *solution* to $\varphi$ if $\varphi$ evaluates to true under $\tau$.

A *literal* is a variable (e.g., $x$) or the negation of a variable (e.g., $\neg x$). A Boolean formula is a *(CNF) clause* if it is the disjunction of literals (e.g., $x \vee y \vee \neg z$). A Boolean formula is a *CNF formula* (also called *in CNF*) if it is the conjunction of CNF clauses. For example $(x \vee y) \wedge (z \vee \neg x)$ is a CNF formula with 2 clauses and 4 solutions ($\{y\}$, $\{y, z\}$, $\{x, z\}$, and $\{x, y, z\}$). If $\varphi$ is a CNF formula, we often treat $\varphi$ as the set of its clauses and so write $C \in \varphi$ to mean that $C$ is one of the CNF clauses of $\varphi$. Thus $\varphi = \bigwedge_{C \in \varphi} C$.

A generalization of Boolean formulas are pseudo-Boolean functions.

**Definition 2.1** (Pseudo-Boolean function)**.** *Let $X$ be a set of Boolean variables. A* pseudo-Boolean function *over $X$ is a function $f : 2^X \rightarrow \mathbb{R}$. We use $\texttt{Vars}(f)$ to indicate the set of variables of $f$.*

Pseudo-Boolean functions are also known as *factors*. A pseudo-Boolean function can naturally represent a Boolean formula. In detail, for a Boolean formula $\varphi$ define $[\varphi] : 2^{\texttt{Vars}(\varphi)} \rightarrow \mathbb{R}$ to be a pseudo-Boolean function where, for all $\tau \in 2^{\texttt{Vars}(\varphi)}$, if $\tau$ is a solution of $\varphi$ then $[\varphi](\tau) \equiv 1$ else $[\varphi](\tau) \equiv 0$. As an abuse of notation, we often omit the brackets for simplicity and define $\varphi(\tau) \equiv 1$ if $\tau$ is a solution of $\varphi$ and $\varphi(\tau) \equiv 0$ otherwise.

Operations on pseudo-Boolean functions include *product*, *$\Sigma$-projection* and *$\exists$-projection*. First, we define product.

**Definition 2.2** (Product). *Let $X$ and $Y$ be sets of Boolean variables. The* product *of functions $f : 2^X \to \mathbb{R}$ and $g : 2^Y \to \mathbb{R}$ is the function $f \cdot g : 2^{X \cup Y} \to \mathbb{R}$ defined for all $\tau \in 2^{X \cup Y}$ by $(f \cdot g)(\tau) \equiv f(\tau \cap X) \cdot g(\tau \cap Y)$.*

Product generalizes conjunction of Boolean formulas: if $\alpha$ and $\beta$ are Boolean formulas, then $[\alpha] \cdot [\beta] = [\alpha \wedge \beta]$. Thus if $\varphi$ is a CNF formula then $[\varphi] = \prod_{C \in \varphi} [C]$.

We next define $\Sigma$-projection.

**Definition 2.3** ($\Sigma$-projection). *Let $X$ be a set of Boolean variables, and let $x \in X$. The* $\Sigma$-projection *of a function $f : 2^X \to \mathbb{R}$ w.r.t. $x$ is the function $\Sigma_x f : 2^{X \setminus \{x\}} \to \mathbb{R}$ defined for all $\tau \in 2^{X \setminus \{x\}}$ by $(\Sigma_x f)(\tau) \equiv f(\tau) + f(\tau \cup \{x\})$.*

$\Sigma$-projection is also known as *additive projection* or *marginalization*. Finally, we define $\exists$-projection.

**Definition 2.4** ($\exists$-projection). *Let $X$ be a set of Boolean variables, and let $x \in X$. The* $\exists$-projection *of a function $f : 2^X \to \mathbb{R}$ w.r.t. $x$ is the function $\exists_x f : 2^{X \setminus \{x\}} \to \mathbb{R}$ defined for all $\tau \in 2^{X \setminus \{x\}}$ by $(\exists_x f)(\tau) \equiv \max(f(\tau), f(\tau \cup \{x\}))$.*

$\exists$-projection is also called *disjunctive projection* and generalizes existential quantification: if $\alpha$ is a Boolean formula and $x \in \texttt{Vars}(\alpha)$, then $\exists_x[\alpha] = [\exists x\ \alpha]$.

In general, $\Sigma$-projection does not commute with $\exists$-projection. For example, if $f(x, y) = [x \oplus y]$ (XOR), then $\Sigma_x \exists_y f \neq \exists_y \Sigma_x f$. Nevertheless, $\Sigma$-projection and $\exists$-projection are each independently commutative. That is, for all $x, y \in X$ and $f : 2^X \to \mathbb{R}$, we have that $\Sigma_x \Sigma_y f = \Sigma_y \Sigma_x f$ and $\exists_x \exists_y f = \exists_y \exists_x f$. Thus, for all $X = \{x_1, \ldots, x_n\}$, define $\Sigma_X f \equiv \Sigma_{x_1} \ldots \Sigma_{x_n} f$ and $\exists_X f \equiv \exists_{x_1} \ldots \exists_{x_n} f$. We also take the convention that $\Sigma_\varnothing f \equiv \exists_\varnothing f \equiv f$.

## 2.2 Discrete Integration

In discrete integration (also called constrained counting) the task is to count the total weight, subject to a given weight function, of the set of solutions of input constraints [3]. We represent weight functions as pseudo-Boolean functions. As is standard, we focus on literal-weight functions:

**Definition 2.5** (Literal-Weight Function)**.** *A pseudo-Boolean function $W : 2^X \to \mathbb{R}$ is* literal-weight *function if there exist pseudo-Boolean functions $W_x : 2^{\{x\}} \to \mathbb{R}$ for all $x \in X$ such that $W = \prod_{x \in X} W_x$.*

Literal-weight functions are also called *log-linear* functions and can be employed to capture a wide variety of probabilistic distributions that arise from graphical models, conditional random fields, skip-gram models, and the like [9]. $W_x(\{x\})$ indicates the weight of the positive literal $x$ in $W$ and $W_x(\varnothing)$ indicates the weight of the negative literal $\neg x$ in $W$.

This dissertation primarily focuses on two subclasses of discrete integration: model counting and projected model counting.

### 2.2.1 Model Counting

In (weighted) model counting, the input constraints are given as a propositional formula, typically in CNF. Formally:

**Definition 2.6** (Weighted Model Count)**.** *Let $\varphi$ be a Boolean formula over Boolean variables $X$ and let $W : 2^X \to \mathbb{R}$ be a pseudo-Boolean function. We say that $(X, \varphi, W)$ is an instance of* weighted model counting. *The $W$-*weighted model count *of $\varphi$ is*

$$W(\varphi) \equiv \sum_{\tau \in 2^X} \varphi(\tau) \cdot W(\tau).$$

The *unweighted model count* of a Boolean formula $\varphi$ is the number of solutions of $\varphi$, i.e. the $W$-weighted model count of $\varphi$ where $W(\tau) = 1$ for all $\tau \in 2^X$.

Many discrete-integration queries can be phrased and solved as a weighted model counting instance [3]. This includes problems from probabilistic reasoning [1,2], verification [4], and statistical physics [84].

Existing approaches to weighted model counting can be split broadly into three categories: *search*, *knowledge compilation*, and *dynamic programming*. In counters based on search (e.g., `cachet` [19] and `SharpSAT` [20]), the idea is to take an algorithm for exploring the entire solution space of $\varphi$ (e.g. DPLL [16,17], and later CDCL [18]) and augment it to enumerate the number of partial solutions encountered. In counters based on knowledge compilation (e.g. `miniC2D` [21] and `d4` [22]), the idea is to compile $\varphi$ into an alternative representation on which counting is easy, for example into a single binary decision diagram (BDD) [23] or a sentential decision diagram (SDD) [24]. In counters based on dynamic programming (e.g. `ADDMC` [29] and `gpuSAT2` [27, 28]), the idea is to traverse the constraint structure of $\varphi$ and reason about subsets of constraints. The techniques developed in this dissertation are all based on dynamic programming.

### 2.2.2 Projected Model Counting

In (weighted) projected model counting, the input constraints are given as a propositional formula, typically in CNF, together with a set of irrelevant, existential variables. Formally:

**Definition 2.7** (Weighted Projected Model Count). *Let $\varphi$ be a Boolean formula, $\{X, Y\}$ be a partition of* $\texttt{Vars}(\varphi)$*, and $W : 2^X \to \mathbb{R}$ be a pseudo-Boolean function. We say that $(X, Y, \varphi, W)$ is an instance of* weighted projected model counting*. The* $W$-weighted $Y$-projected model count *of $\varphi$ is*

$$\texttt{WPMC}(\varphi, W, Y) \equiv \sum_{\tau \in 2^X} \left( W(\tau) \cdot \max_{\sigma \in 2^Y} \varphi(\tau \cup \sigma) \right).$$

Variables in $X$ are called *relevant* or *additive*, and variables in $Y$ are called *irrelevant* or *disjunctive*. Notice that model counting is a special case of projected model counting where all variables are relevant. Thus $W(\varphi) = \texttt{WPMC}(\varphi, W, \varnothing)$ for all Boolean formulas $\varphi$ and weight functions $W : 2^{\texttt{Vars}(\varphi)} \to \mathbb{R}$.

Many discrete-integration queries require projected variables and so are typically phrased and solved as a weighted projected model counting instance. This includes problems from planning [85], formal verification [86], and reliability estimation [87].

There are several recent tools for weighted projected model counting. For example, $\texttt{D4}_{\texttt{P}}$ is based on compilation to decision decomposable negation normal form [88], `projMC` is based on search (to build disjunctive decompositions) [88], and `reSSAT` is based on search [89]. While our focus in Chapter 6 is on deterministic exact weighted projected model counting, it is worth mentioning that various relaxations have also been studied, e.g., probabilistic [90], approximate [91–93], or unweighted [14,85,94,95] projected model counting.

## 2.3 Graphs

A *graph* $G$ has a nonempty set of vertices $\mathcal{V}(G)$, a set of (undirected) edges $\mathcal{E}(G)$, a function $\delta_G : \mathcal{V}(G) \to 2^{\mathcal{E}(G)}$ that gives the set of edges incident to each vertex, and a function $\epsilon_G : \mathcal{E}(G) \to 2^{\mathcal{V}(G)}$ that gives the set of vertices incident to each edge. Each edge must be incident to exactly two vertices, but multiple edges can exist between two vertices. If $E \subset \mathcal{E}(G)$, let $\epsilon_G[E] = \bigcup_{e \in E} \epsilon_G(e)$. Similarly, if $V \subset \mathcal{V}(G)$, let $\delta_G[V] = \bigcup_{v \in V} \delta_G(v)$. An *edge clique cover* of a graph $G$ is a set $A \subseteq 2^{\mathcal{V}(G)}$ such that (1) every vertex $v \in \mathcal{V}(G)$ is an element of some set in $A$, and (2) every element of $A$ is a clique in $G$ (that is, for every $C \in A$ and every pair of distinct $v, w \in C$ there is an edge between $v$ and $w$ in $G$).

### 2.3.1 Trees

A *tree* is a simple, connected, and acyclic graph. A *leaf* of a tree $T$ is a vertex of degree one, and we use $\mathcal{L}(T)$ to denote the set of leaves of $T$. For every edge $a$ of $T$, deleting $a$ from $T$ yields exactly two trees, whose leaves define a partition of $\mathcal{L}(T)$.

Let $C_a \subseteq \mathcal{L}(T)$ denote an arbitrary element of this partition. Throughout this work, we often refer to a vertex of a tree as a *node* and an edge as an *arc*, since our proofs will frequently work simultaneously with a graph and an associated tree.

A *rooted tree* is a tree $T$ together with a distinguished node $r \in \mathcal{V}(T)$ called the *root*. In a rooted tree $(T, r)$, each node $n \in \mathcal{V}(T)$ has a (possibly empty) set of *children*, denoted $\mathcal{C}_{T,r}(n)$, which contains all nodes $n'$ adjacent to $n$ s.t. all paths from $n'$ to $r$ contain $n$.

A *rooted binary tree* is a rooted tree where either $|\mathcal{V}(T)| = 1$ or the root has degree two and every non-root node has degree three or one. If $|\mathcal{V}(T)| > 1$, the *immediate subtrees of* $T$ are the two rooted binary trees that are the connected components of $T$ after the root is removed.

### 2.3.2 Graph Decompositions

In this work, we use three decompositions of a graph as a tree: carving decompositions [41], branch decompositions [40], and tree decompositions [40]. All decompose the graph into an *unrooted binary tree*, which is a tree where every vertex has degree one or three. First, we describe carving decompositions [41]:

**Definition 2.8** (Carving Decomposition)**.** *Let $G$ be a graph. A* carving decomposition *for $G$ is an unrooted binary tree $T$ whose leaves are the vertices of $G$, i.e. $\mathcal{L}(T) = \mathcal{V}(G)$.*

*The* width *of $T$, denoted $width_c(T)$, is the maximum number of edges in $G$ between $C_a$ and $\mathcal{V}(G) \setminus C_a$ for all $a \in \mathcal{E}(T)$, i.e.,*

$$width_c(T) = \max_{a \in \mathcal{E}(T)} \left| \left( \bigcup_{v \in C_a} \delta_G(v) \right) \cap \left( \bigcup_{v \in \mathcal{V}(G) \setminus C_a} \delta_G(v) \right) \right|.$$

*The width of a carving decomposition $T$ with no edges is 0.*

Recall that $C_a \subseteq \mathcal{L}(T)$ is an arbitrary element of the partition of $\mathcal{L}(T)$ induced by deleting $a$ from $T$. The *carving width* of a graph $G$, denoted $width_c(G)$, is the minimum width across all carving decompositions for $G$. Note that an equivalent definition of carving decompositions allows for degree two vertices within the tree as well. Branch decompositions are the dual of carving decompositions and hence can be defined by swapping the role of $\mathcal{V}(G)$ and $\mathcal{E}(G)$ in Definition 2.8.

Finally, we define tree decompositions [40]:

**Definition 2.9** (Tree Decomposition). *A* tree decomposition *for a graph $G$ is an unrooted binary tree $T$ together with a labeling function $\chi : \mathcal{V}(T) \to 2^{\mathcal{V}(G)}$ that satisfies the following three properties:*

1. *Every vertex of $G$ is contained in the label of some node of $T$. That is, $\mathcal{V}(G) = \bigcup_{n \in \mathcal{V}(T)} \chi(n)$.*
2. *For every edge $e \in \mathcal{E}(G)$, there is a node $n \in \mathcal{V}(T)$ whose label is a superset of $\epsilon_G(e)$, i.e. $\epsilon_G(e) \subseteq \chi(n)$.*
3. *If $n$ and $o$ are nodes in $T$ and $p$ is a node on the path from $n$ to $o$, then $\chi(n) \cap \chi(o) \subseteq \chi(p)$.*

*The* width *of a tree decomposition, denoted $width_t(T, \chi)$, is the maximum size (minus 1) of the label of every node, i.e.,*

$$width_t(T, \chi) = \max_{n \in \mathcal{V}(T)} |\chi(n)| - 1.$$

The *treewidth* of a graph $G$, denoted $width_t(G)$, is the minimum width across all tree decompositions for $G$. The treewidth of a tree is 1. Treewidth is bounded by thrice the carving width [96]. The treewidth (plus 1) of graph is no smaller than the branchwidth and is bounded from above by 3/2 times the branchwidth [40].

Given a CNF formula $\varphi$, a variety of associated graphs have been considered. The *incidence graph* of $\varphi$ is the bipartite graph where both variables and clauses are vertices and edges indicate that the variable appears in the connected clause. The *primal graph* of $\varphi$ is the graph where variables are vertices and edges indicate

| **Database Concept** | **Factor Graph Concept** | **Tensor Concept** |
|---|---|---|
| Attribute | Variable | Index |
| Table | Pseudo-Boolean Function | Tensor |
| Project-Join Query | Factor Graph | Tensor Network |
| Join Tree | Dtree | Contraction Tree |
| Width | Largest Cluster Size | Contraction Complexity |
| - | Largest Separator Size | Max Rank |

Table 2.1 : An analogy between the language of databases, the language of factor graphs, and the language of tensors.

that two variables appear together in a clause. There are fixed-parameter tractable model counting algorithms with respect to the treewidth of the incidence graph and the primal graph [51]. If the treewidth of the primal graph of a formula $\varphi$ is $k$, the treewidth of the incidence graph of $\varphi$ is at most $k+1$ [97].

## 2.4 An Introduction to Tensors and Tensor Networks

In this section, we introduce tensors, tensor networks, and the problem of tensor-network contraction. To aid in exposition, along the way we build an analogy between the language of databases [98], the language of factor graphs [75, 99, 100], and the language of tensors in Table 2.1.

### 2.4.1 Tensors

*Tensors* are a generalization of vectors and matrices to higher dimensions– a tensor with $r$ dimensions is a table of values each labeled by $r$ indices. An index is analogous to a variable in constraint satisfaction or an attribute in database theory.

Fix a set **Ind** and define an *index* to be an element of **Ind**. For each index $i$ fix a finite set $[i]$ called the *domain* of $i$. An index is analogous to a variable in constraint satisfaction.

An *assignment* to a set of indices $I \subseteq \mathbf{Ind}$ is a function $\tau$ that maps each index $i \in I$ to an element of $[i]$. Let $[I]$ denote the set of assignments to $I$, i.e.,

$$[I] = \{\tau : I \to \bigcup_{i \in I} [i] \text{ s.t. } \tau(i) \in [i] \text{ for all } i \in I\}.$$

We now define tensors as multidimensional arrays of values, indexed by assignments to a set of indices:[*]

**Definition 2.10** (Tensor)**.** *A* tensor $A$ *over a finite set of indices (denoted* $\mathcal{I}(A)$*) is a function* $A : [\mathcal{I}(A)] \to \mathbb{C}$ *(where* $\mathbb{C}$ *is the set of complex numbers).*

The *rank* of a tensor $A$ is the cardinality of $\mathcal{I}(A)$. The memory to store a tensor (in a dense way) is exponential in the rank. For example, a scalar is a rank 0 tensor, a vector is a rank 1 tensor, and a matrix is a rank 2 tensor. An example of a higher-rank tensor is the *copy tensor* on a set of indices $I$, which is the tensor $\text{COPY}_I : [I] \to \mathbb{C}$ such that, for all $\tau \in [I]$, $\text{COPY}_I(\tau) \equiv 1$ if $\tau$ is a constant function on $I$ and $\text{COPY}_I(\tau) \equiv 0$ otherwise [101]. Note that a pseudo-Boolean function can be seen as a special case of a tensor, where each index has a domain of size 2 and every tensor entry lies in $\mathbb{R}$.

It is common to consider sets of tensors closed under contraction (see Section 2.4.2), e.g. tensors with entries in $\mathbb{R}$ as in Section 3.2. Database tables under bag-semantics [102], i.e., multirelations, are tensors with entries in $\mathbb{N}$. Probabilistic database tables [103] are tensors with entries in $[0, 1]$ that sum to 1.

Many tools exist (e.g. `numpy` [59]) to efficiently manipulate tensors. In Section 3.5, we use these tools to implement tensor-network contraction, defined next.

[*]In some works, a tensor is defined as a multilinear map and Definition 2.10 would be its representation in a fixed basis.

### 2.4.2 Tensor Networks

A *tensor network* defines a complex tensor by combining a set of simpler tensors in a principled way. This is analogous to how a database query defines a resulting table in terms of a computation across many tables.

**Definition 2.11** (Tensor Network)**.** *A* tensor network $N$ *is a nonempty set of tensors across which no index appears more than twice.*

*Free indices* of $N$ are indices that appear once, while *bond indices* of $N$ are indices that appear twice. We denote the set of free indices of $N$ by $\mathcal{F}(N)$ and the set of bond indices of $N$ by $\mathcal{B}(N)$. The *bond dimension* of $N$ is the maximum size of $[i]$ for all bond indices $i$ of $N$. We focus in this work on tensor networks with relatively few (or no) free indices and hundreds or thousands of bond indices, and where the bond dimension is small (i.e., 2). Such tensor networks are obtained in a variety of applications [53, 104], including the reduction from model counting to tensor network contraction [65] (see Section 3.2).

The problem of *tensor-network contraction*, given an input tensor network $N$, is to compute the *contraction* of $N$ by marginalizing all bond indices:

**Definition 2.12** (Tensor Network Contraction)**.** *The* contraction *of a tensor network $N$ is a tensor $\mathcal{T}(N)$ with indices $\mathcal{F}(N)$ (the set of free indices of $N$), i.e. a function $\mathcal{T}(N) : [\mathcal{F}(N)] \to \mathbb{C}$, that is defined for all $\tau \in [\mathcal{F}(N)]$ by*

$$\mathcal{T}(N)(\tau) \equiv \sum_{\rho \in [\mathcal{B}(N)]} \prod_{A \in N} A((\rho \cup \tau)\big|_{\mathcal{I}(A)}). \tag{2.1}$$

In Definition 2.12, $\rho$ is an assignment to the bond indices of $N$, while $\tau$ is an assignment to the free indices of $N$. Thus $\rho \cup \tau$ is an assignment to all indices of $N$.

For example, the contraction of the tensor network $\{\text{COPY}_I, \text{COPY}_J\}$ with $I \cap J \neq \varnothing$ is the tensor $\text{COPY}_{I \oplus J}$ (where $I \oplus J$ is the symmetric difference of $I$ and $J$). Notice that if a tensor network has no free indices then its contraction is a rank 0 tensor.

A tensor network $N'$ is a *partial contraction* of a tensor network $N$ if there is a surjective function $f : N \to N'$ s.t. for every $A \in N'$ we have $\mathcal{T}(f^{-1}(A)) = A$; that is, if every tensor in $N'$ is the contraction of some tensors of $N$. If $N'$ is a partial contraction of $N$, then $\mathcal{T}(N') = \mathcal{T}(N)$.

Let $A$ and $B$ be tensors. Their *contraction* $A \cdot B$ is the contraction of the tensor network $\{A, B\}$. If $\mathcal{I}(A) = \mathcal{I}(B)$, their *sum* $A + B$ is the tensor with indices $\mathcal{I}(A)$ whose entries are given by the sum of the corresponding entries in $A$ and $B$.

Following our analogy, given a tensor network containing database tables (under bag-semantics) as tensors, its contraction is the join of those tables followed by the projection of all shared attributes. Thus a tensor network is analogous to a project-join query. A tensor network can also be seen as a variant of a factor graph [99] with the additional practical restriction that no variable appears more than twice. The contraction of a tensor network corresponds to the marginalization of a factor graph [105], which is a a special case of the sum-of-products problem [75,106] and the FAQ problem [7]. The restriction on variable appearance is heavily exploited in tools for tensor-network contraction and in this work, since it allows tensor contraction to be implemented as matrix multiplication and so leverage significant work in high-performance computing on matrix multiplication on CPUs [107] and GPUs [108].

## 2.5 Decision Diagrams

An *algebraic decision diagram (ADD)* is a compact representation of a pseudo-Boolean function as a directed acyclic graph [67]. For functions with logical structure, an ADD representation can be exponentially smaller than the explicit representation. Originally designed for matrix multiplication and shortest path algorithms, ADDs have also been used for Bayesian inference [109,110], stochastic planning [111], model checking [112], and model counting [29,113]. Formally:

**Definition 2.13** (ADD). *An* algebraic decision diagram (ADD) *is a tuple* $(X, S, \sigma, G)$ *where* $X$ *is a set of Boolean variables,* $S$ *is an arbitrary set (called the* carrier set*),*

*$\sigma : X \to \mathbb{N}$ is an injection (called the* diagram variable order*), and $G$ is a rooted directed acyclic graph satisfying the following three properties:*

1. *Every leaf node of $G$ is labeled with an element of $S$,*

2. *Every internal node of $G$ is labeled with an element of $X$ and has two outgoing edges, labeled 0 and 1, and*

3. *For every path in $G$, the labels of internal nodes must occur in increasing order under $\sigma$.*

In this work, we only need to consider ADDs with the carrier set $S = \mathbb{R}$. An ADD $(X, S, \sigma, G)$ is a compact representation of a function $f : 2^X \to S$. Although there are many ADDs representing $f$, for each injection $\sigma : X \to \mathbb{N}$, there is a unique minimal ADD that represents $f$ with $\sigma$ as the diagram variable order, called the *canonical ADD*. ADDs can be minimized in polynomial time, so it is typical to only work with canonical ADDs.

One challenge in using ADDs is choosing the diagram variable order. The choice of diagram variable order can have a dramatic impact on the size of the ADD; some variable orders may produce ADDs that are exponentially smaller than others for the same real-valued function. A variety of techniques exist in prior work to heuristically find diagram variable orders [77, 78, 114]. Moreover, since binary decision diagrams (BDDs) [115] can be seen as ADDs with carrier set $S = \{0, 1\}$, there is significant overlap with the techniques to find variable orders for BDDs.

Several packages exist for efficiently manipulating ADDs. For example, `CUDD` [72] implements product, $\Sigma$-projection and $\exists$-projection on ADDs in polynomial time (in the size of the ADD representation).

# Chapter 3

# TensorOrder: Tensors and Graph Decompositions with Tensor Networks

*Tensor networks* are a tool used across quantum physics and computer science for describing and reasoning about quantum systems, big-data processing, and more [52–54]. A tensor network describes a complex tensor as a computation on many simpler tensors, and the problem of *tensor-network contraction* is to perform this computation. Contraction is a key operation in neural network training and inference [55–58], and, as such, many of the optimizations for neural networks also apply to tensor-network contraction [62,63,116]. Moreover, there is a known reduction from unweighted model counting to tensor-network contraction [65]. Tensor networks are therefore a natural bridge between high-performance computing and discrete integration, but this connection has not been fully explored.

In this chapter, we exploit this bridge to build `TensorOrder`, a new tool for weighted model counting through tensor network contraction. `TensorOrder` uses a 3-phase algorithm to perform weighted model counting. First, in the *reduction* phase, a weighted model counting instance is reduced to a tensor-network problem using a novel reduction. Second, in the *planning* phase, an order to contract tensors in the network is determined. Finally, in the *execution* phase, tensors in the network are contracted with state-of-the-art tensor libraries using the determined order.

The key theoretical challenge of this approach lies in the planning phase: the determined contraction order should minimize the computational cost of the execution phase. Since the number of possible contraction orders grows exponentially in the number of tensors, cost-based exhaustive algorithms, e.g. [117], cannot scale to handle the large tensor networks required for the reduction from constrained counting. Instead, recent work [66] gave heuristics that can sometimes find a “good-enough” contraction order through structure-based optimization. Finding efficient contraction orders for tensor networks remains an area of active research [118].

The primary theoretical contribution of this chapter is the application of heuristic graph-decomposition techniques to find efficient contraction orders for tensor networks. Algorithms based on graph decompositions have been successful across computer science [33, 44], and their success in practice relies on finding good decompositions of arbitrary graphs. This, along with several recent competitions [49], has spurred the development of a variety of heuristics and tools for efficiently finding graph decompositions [45–47]. While we do not establish new parameterized complexity results for model counting (as fixed-parameter algorithms for model counting are well-known for a variety of parameters [50, 51]), we combine these theoretical results with high-performance tensor network libraries and with existing heuristic graph-decomposition tools to produce a competitive tool for weighted model counting.

We first discuss the **Line-Graph** planner (**LG**) for finding efficient contraction orders through structure-based graph analysis. First applied to tensor networks by Markov and Shi [119], we contribute a new analysis that more closely matches the memory usage of existing tensor libraries. Our analysis combines two theoretical insights: (1) memory-efficient contraction orders are equivalent to low-width carving decompositions (first observed in [120]), and (2) tree decompositions can be used to find carving decompositions. **LG** has previously been implemented using exact tools for finding tree decompositions [121], but its implementation using heuristic tools for tree decompositions is largely unexplored.

Although **LG** is a general-purpose technique for finding contraction orders, **LG** cannot handle high-rank tensors and so cannot solve many existing counting benchmarks. We therefore contribute a novel structure-based method for finding efficient contraction orders, tailored for constrained counting: the **Factor-Tree** planner (**FT**). **FT** factors high-rank, highly-structured tensors as a preprocessing step, leveraging prior work on Hierarchical Tucker representations [122].

In order to compare our approaches against other counters (weighted model counters `cachet` [19], `miniC2D` [21], and `d4` [22], and unweighted model counters `dynQBF` [25], `dynasp` [26], and `SharpSAT` [20]), as well as other tensor-based planning techniques, we implemented **LG** and **FT** in our new tool `TensorOrder` using three state-of-the-art heuristic tree-decomposition solvers. **LG** outperforms other model counters and tensor-based planning techniques on a set of unweighted benchmarks, while **FT** improves the virtual best solver on 21% of a standard set of weighted model counting benchmarks. `TensorOrder` is thus useful as part of a portfolio of weighted model counters. All code, benchmarks, and detailed data of benchmark runs are available at https://github.com/vardigroup/TensorOrder.

The rest of the chapter is organized as follows. We introduce a framework for solving the problem of weighted model counting with tensor networks in Section 3.1. We discuss a novel reduction from weighted model counting to tensor-network contraction in Section 3.2. We discuss algorithms for contracting a tensor network in Section 3.3. We discuss algorithms for constructing contraction trees in Section 3.4, including the **Line-Graph** planner and the **Factor-Tree** planner. We implement these techniques in `TensorOrder` and analyze its performance experimentally in Section 3.5. Finally, we conclude in Section 3.6.

## 3.1 Separating Planning and Execution

In this section, we discuss the algorithm for literal-weighted model counting using tensor networks. This algorithm is presented as Algorithm 3.1 and has three phases.

First, in the *reduction* phase the input formula $\varphi$ and weight function $W$ is transformed into a tensor network $N$. The goal of the reduction phase is that the contraction of $N$, $\mathcal{T}(N)$, should be equal to the $W$-weighted model count of $\varphi$. We discuss

**Algorithm 3.1:** Computing the weighted model count with a TN

```
  Input: $\varphi$: a CNF formula
  Input: $W$: a literal-weight function
  Output: $W(\varphi)$, the weighted model count of $\varphi$ w.r.t. $W$
1 $N \leftarrow$ Reduce$(\varphi, W)$
2 repeat
3     $M, T \leftarrow$ Plan$(N)$                          /* e.g., LG or FT */
4 until $\alpha \cdot$ TimeCost$(M, T) <$ elapsed time in seconds
5 return Execute$(M, T)$
```

in more detail in Section 3.2.

Second, in the *planning* phase an order $T$ to contract tensors in the network is determined. $T$ is a *contraction tree* [123]:

**Definition 3.1** (Contraction Tree)**.** *Let $N$ be a tensor network. A* contraction tree *for $N$ is a rooted binary tree $T$ whose leaves are the tensors of $N$.*

In our database analogy from Section 2.4, a contraction tree for a tensor network representing a project-join query is a join tree of that query (with projections done as early as possible). In our factor-graph analogy from Section 2.4, a contraction tree corresponds to a *dtree* [124], to an elimination order [100], and to a binary join tree [125] where factors are assigned to leaf nodes.

There are two subtleties in the planning process. Firstly, the planning phase is allowed to modify the input tensor network $N$ as long as the new tensor network $M$ has the same contraction as $N$. This allows the planning phase to tweak the tensor network to enable better contraction trees. Secondly, planning in Algorithm 3.1 is an anytime process: we heuristically generate better contraction trees $T$ until one is "good enough" to use (governed by a parameter $\alpha \in \mathbb{R}$). This allows us to incorporate heuristic, anytime high-level reasoning tools into a planning algorithm. In this work we discuss two planning techniques: the **Line-Graph** planner (see Section 3.4.2) and the **Factor-Tree** planner (see Section 3.4.3).

Third, in the *execution* phase the chosen contraction tree $T$ is used to contract the tensor network $M$. The goal of the execution phase is to compute the contraction of $M$ and thus compute the $W$-weighted model count of $\varphi$. The computation cost of the execution phase is heavily determined by the chosen contraction tree. We discuss this phase in more detail in Section 3.3.

We assert the correctness of Algorithm 3.1 in the following theorem.

**Theorem 3.2.** *Let $\varphi$ be a CNF formula and let $W : 2^{\texttt{Vars}\varphi} \to \mathbb{R}$ be a literal-weight function. Assume:*

1. `Reduce`$(\varphi, W)$ *returns a tensor network $N$ s.t. $\mathcal{T}(N)(\varnothing) = W(\varphi)$,*
2. `Plan`$(N)$ *returns a tensor network $M$ and a contraction tree $T$ for $M$ s.t. $\mathcal{T}(M) = \mathcal{T}(N)$, and*
3. `Execute`$(M, T)$ *returns $\mathcal{T}(M)$ for all tensor networks $M$ and contraction trees $T$ for $M$.*

*Then Algorithm 3.1 returns $W(\varphi)$.*

*Proof.* By Assumption 3, Algorithm 3.1 returns $\mathcal{T}(M)(\varnothing)$. By Assumption 2, this is equal to $\mathcal{T}(N)(\varnothing)$, which by Assumption 1 is exactly $W(\varphi)$. □

Theorem 3.2 proves that Algorithm 3.1 indeed computes the weighted model count of $\varphi$, as long the reduction, planning, and execution phases all satisfy the stated assumptions. We verify Assumption 1 in Section 3.2, Assumption 2 in Section 3.4, and Assumption 3 in Section 3.3.

Organizationally, note that we discuss the execution phase in Section 3.3 before we discuss the planning phase in Section 3.4. This is because we must understand how plans are used before we can evaluate various planning algorithms.

## 3.2 Reduction Phase: From Weighted Model Counting to Tensor Networks

Existing reductions from model counting to tensor-network contraction [65, 66] focus on unweighted model counting. Since we are interested in the more general problem of weighted model counting, we prove that the reduction can be easily extended:

**Theorem 3.3.** *Let $\varphi$ be a CNF formula over Boolean variables $X$ and let $W : 2^X \to \mathbb{R}$ be a literal-weight function. One can construct in polynomial time a tensor network $N_\varphi$ such that $\mathcal{F}(N_\varphi) = \varnothing$ and the contraction of $N_\varphi$ is $W(\varphi)$.*

*Proof.* The key idea is to include in $N_\varphi$ a tensor $A_x$ for each variable $x \in X$ and a tensor $B_C$ for each clause $C \in \varphi$ such that the tensors share an index if and only if the corresponding variable appears in the corresponding clause.

Define $I = \{(x, C) : C \in \varphi, x \in \texttt{Vars}(C)\}$ to be a set of indices, each with domain $\{0, 1\}$. That is, $I$ has an index for each appearance of each variable in $\varphi$. We use $I$ as the set of indices in $N_\varphi$.

Next, recall that since $W$ is a literal-weight function there exist pseudo-Boolean functions $W_x : 2^{\{x\}} \to \mathbb{R}$ for all $x \in X$ such that $W = \prod_{x \in X} W_x$. For each $x \in X$, let $\text{dep}(x)$ be the set of clauses that contain $x$. Define $A_x : [\{x\} \times \text{dep}(x)] \to \mathbb{R}$ to be the tensor defined by

$$A_x(\tau) \equiv \begin{cases} W_x(\{x\}) & \text{if } \tau((x, C)) = 1 \text{ for all } C \in \text{dep}(x) \\ W_x(\varnothing) & \text{if } \tau((x, C)) = 0 \text{ for all } C \in \text{dep}(x) \\ 0 & \text{otherwise.} \end{cases}$$

Next, for each $C \in \varphi$, let $B_C : [\texttt{Vars}(C) \times \{C\}] \to \mathbb{R}$ be the tensor defined by

$$B_C(\tau) \equiv \begin{cases} 1 & \text{if } \{x : x \in \texttt{Vars}(C) \text{ and } \tau((x, C)) = 1\} \text{ satisfies } C \\ 0 & \text{otherwise.} \end{cases}$$

Finally, let $N_\varphi = \{A_x : x \in X\} \cup \{B_C : C \in \varphi\}$. Each index $(x, C) \in I$ appears exactly twice in $N_\varphi$ (in $A_x$ and $B_C$) and so is a bond index. Thus $N_\varphi$ is a tensor

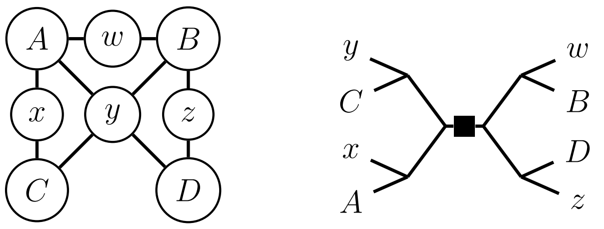


Figure 3.1 : The tensor network (left) produced by Theorem 3.3 on $\varphi = (w \vee x \vee \neg y) \wedge (w \vee y \vee z) \wedge (\neg x \vee \neg y) \wedge (\neg y \vee \neg z)$, consisting of 8 tensors and 10 indices. Vertices in this diagram are tensors, while edges indicate that the tensors share an index. The weight function affects the entries of the tensors for $w$, $x$, $y$, and $z$. This tensor network has a contraction tree (right) of max rank 4, but no contraction trees of smaller max rank.

network with $\mathcal{F}(N_\varphi) = \varnothing$. We now compute $\mathcal{T}(N_\varphi)$, the contraction of $N_\varphi$. By Definition 5,

$$\mathcal{T}(N_\varphi)(\varnothing) = \sum_{\rho \in [I]} \prod_{x \in X} A_x(\rho|_{\{x\}\times \mathrm{dep}(x)}) \cdot \prod_{C \in \varphi} B_C(\rho|_{\texttt{Vars}(C)\times\{C\}}).$$

To compute the term of this sum for each $\rho \in [I]$, we examine if there exists some $\tau_\rho \in [X]$ such that $\rho((x, C)) = \tau_\rho(x)$ for all $(x, C) \in I$. If so, then by construction $A_x(\rho|_{\{x\}\times \mathrm{dep}(x)}) = W_{x,\tau_\rho(x)}$ (where $W_{x,\tau_\rho(x)} = W_x(\{x\})$ if $\tau_\rho(x) = 1$ and $W_x(\varnothing)$ if $\tau_\rho(x) = 0$) and $B_C(\rho|_{\texttt{Vars}(C)\times\{C\}}) = C(\tau_\rho|_{\texttt{Vars}(C)})$. On the other hand, if no such $\tau_\rho$ exists then there is some $y \in X$ such that $\rho$ is not constant on $\{y\} \times \mathrm{dep}(y)$. Thus by construction $A_y(\rho|_{\{y\}\times \mathrm{dep}(y)}) = 0$ and so the term in the sum for $\rho$ is 0. Hence

$$\mathcal{T}(N_\varphi)(\varnothing) = \sum_{\tau \in [X]} \prod_{x \in X} W_{x,\tau(x)} \cdot \prod_{C \in \varphi} C(\tau|_{\texttt{Vars}(C)}) = W(\varphi).$$

□

Theorem 3.3 proves that $\texttt{Reduce}(\varphi, W) = N_{\varphi,W}$ satisfies Assumption 1 in Theorem 3.2. See Figure 3.1 for an example of the reduction. This reduction is closely related to the formulation of model counting as the marginalization of a factor graph representing the constraints. Unlike the reduction to factor-graph marginalization, which only assigns factors to clauses, we must also assign a tensor to each variable $x$.

For example, if $x$ has weights $W(x, 0) = W(x, 1) = 1$ then the tensor assigned to $x$ is a copy tensor.

While we focus in this work on weighted model counting with CNF formulas, note that the proof of Theorem 3.3 made no use of the fact that each clause is a CNF clause. This reduction can thus be easily extended beyond CNF formulas to apply to to conjunctions of other types of constraints, e.g. parity or cardinality constraints.

## 3.3 Execution Phase: Contracting the Tensor Network

The goal in the execution phase is to compute the contraction of an input tensor network $N$, using an input contraction tree as a guide for the computation. Recall from Defintion 2.12 that the contraction of $N$ is a function $\mathcal{T}(N) : [\mathcal{F}(N)] \to \mathbb{C}$, that is defined for all $\tau \in [\mathcal{F}(N)]$ by

$$\mathcal{T}(N)(\tau) \equiv \sum_{\rho \in [\mathcal{B}(N)]} \prod_{A \in N} A((\rho \cup \tau)\big|_{\mathcal{I}(A)}). \tag{3.1}$$

For formulas $\varphi$ with hundreds of clauses, the corresponding tensor network produced by Theorem 3.3 has hundreds of bond indices. Directly following Equation 3.1 in this case is infeasible, since it sums over an exponential number of terms (one for each assignment in $[\mathcal{B}(N)]$).

Instead, Algorithm 3.2 shows how to compute $\mathcal{T}(N)$ for a tensor network $N$ using a contraction tree $T$ as a guide [123]. The key idea is to repeatedly choose two tensors $A_1, A_2 \in N$ (according to the structure of $T$) and contract them. One can prove inductively that Algorithm 3.2 satisfies Assumption 3 of Theorem 3.2.

Each contraction in Algorithm 3.2 contains exactly two tensors and so can be implemented as a matrix multiplication with a variety of tensor libraries (e.g. `numpy` [59] or `TensorFlow` [60]). Although the choice of contraction tree does not affect the correctness of Algorithm 3.2, it may have a dramatic impact on the running-time and memory usage. We explore this further in the following section.

**Algorithm 3.2:** Recursively contracting a tensor network

```
Input: A tensor network $N$
Input: A contraction tree $T$
Output: $\mathcal{T}(N)$, the contraction of $N$ as given in Definition 2.12
function Execute($N, T$):
    if $|N| = 1$
        return the tensor contained in $N$
    else
        $T_1, T_2 \leftarrow$ immediate subtrees of $T$
        $A_1 \leftarrow$ Execute($\mathcal{L}(T_1), T_1$)
        $A_2 \leftarrow$ Execute($\mathcal{L}(T_2), T_2$)
        return $A_1 \cdot A_2$
```

## 3.4 Planning Phase: Building a Contraction Tree

The task in the planning stage is, given a tensor network, to find a contraction tree that minimizes the computational cost of Algorithm 3.2. In this section, we first discuss various characterizations of the computational cost of Algorithm 3.2 and conclude that, in this context, *max rank* is the appropriate measure. We then discuss two planning techniques in detail: an existing approach, the **Line-Graph** planner, and a novel technique specialized for weighted model counting, the **Factor-Graph** planner. We contribute a new analysis of each planner in terms of max rank.

### 3.4.1 Measuring Planner Quality

There are a variety of existing planning techniques that each aim to minimize different notions of computational cost of Algorithm 3.2.

Firstly, there are several *cost-based* approaches aim for minimizing the total number of floating point operations (specifically, in Line 8 of Algorithm 3.2). Examples include the `Netcon` algorithm [117], the `einsum` package in `numpy` [59], and the Tensor Contraction Engine [56], which performs additional optimizations of Algorithm 3.2 at the compiler level. Such approaches typically focus on tensor networks with relatively few tensors (tens) but large indices (hundreds of thousands of elements in each domain). Most cost-based approaches are therefore unsuitable for weighted model counting, where we encounter tensor networks with many tensors (thousands) but where each index has a domain of size 2.

Instead, we focus on *structure-based* approaches to planning, which analyze the rank of intermediate tensors that appear during recursive calls of Algorithm 3.2. These ranks indicate the amount of memory and computation required at each recursive stage. Moreover, these ranks are more amenable to analysis.

One line of work [119, 121] uses graph decompositions to analyze the *contraction complexity* of a contraction tree: the maximum over all recursive calls of the sum (minus 1) of the rank of the two tensors contracted in Line 8 of Algorithm 3.2. In our factor graph analogy, contraction complexity of a contraction tree corresponds to the *width* of a dtree (i.e., the size of the largest cluster) [100]. Contraction complexity measures the memory required when Line 8 is computed by summing over the shared indices separately, i.e., first contracting $A_1$ and $A_2$ as if they have no common indices (producing and storing a rank $|\mathcal{I}(A_1)| + |\mathcal{I}(A_2)|$ tensor) and then sequentially contracting each pair of shared indices. In this approach, the memory required is exponential in the contraction complexity. Modern tensor packages (e.g. `numpy`), however, instead sum over all shared indices simultaneously without storing an intermediate result. This alternative approach requires the same number of floating-point operations, but often requires significantly less intermediate memory. Thus contraction complexity overestimates the memory requirements of many contraction trees.

Instead, another line of planning approaches analyzes the maximum rank over all recursive calls of the result of Line 8 (and Line 3). We call this the *max rank* of a contraction tree. In our factor graph analogy, max rank of a contraction tree corresponds to the size of the largest separator of a dtree [100] and is closely connected to the performance of the Shenoy-Shafer algorithm [125, 126]. Max rank measures the memory required when Line 8 is computed by summing over all shared indices simultaneously without storing an intermediate result: the memory required is exponential in the max rank. Thus, max rank estimates the memory usage of modern tensor packages. While the max rank is no more than linearly smaller than the contraction complexity, both appear in the exponent of the required memory. Therefore even small differences between max rank and contraction complexity can indicate very different memory usage.

We therefore choose max rank as the most appropriate measure of contraction tree quality in this setting. Recent work [66] introduced three planning techniques that heuristically minimize the max rank: a greedy approach (called **greedy**), an approach using graph partitioning (called **metis**), and an approach using community-structure detection (called **GN**). Note that **metis** is analogous to earlier work on using graph partitioning to construct dtrees with small separators [100].

### 3.4.2 The Line-Graph Planner

The **Line-Graph** planner for finding contraction trees for a tensor network $N$ applies graph-decomposition techniques to a particular graph constructed from $N$. Prior work [119] on tensor networks with no free indices constructed a graph from a tensor network where tensors correspond to vertices and indices shared between tensors correspond to edges. In the context of constraint networks [114], this is analogous to the dual constraint graph (if multiple edges are drawn between constraints with multiple variables in common).

Although tensor networks constructed from weighted model counting instances do not have free indices, we utilize tensor networks with free indices as part of the preprocessing in Section 3.4.3 and so we need a more general graph construction that can handle free indices. Other works, e.g. [127], extend the graph construction of [119] to tensor networks with free indices by treating free indices as "half-edges" (i.e., edges incident to one vertex), but decompositions of such graphs are not well-studied. In order to cleanly extend our decomposition-based analysis to tensor networks with free indices, in this work we instead add an extra vertex incident to all free indices, which we call the *free vertex*. We call the resulting graph the *structure graph* of a tensor network:

**Definition 3.4** (Structure Graph)**.** *Let $N$ be a tensor network. The* structure graph *of $N$ is the graph denoted struct$(N)$ whose vertices are the tensors of $N$ and a fresh vertex $z$ (called the* free vertex*) and whose edges are the indices of $N$. Each tensor is incident to its indices, and $z$ is incident to all free indices. That is, with $G = \mathit{struct}(N)$ we have $\mathcal{V}(G) = N \sqcup \{z\}$, $\mathcal{E}(G) = \mathcal{B}(N) \cup \mathcal{F}(N)$, $\delta_G(A) = \mathcal{I}(A)$ for all $A \in N$, and $\delta_G(z) = \mathcal{F}(N)$.*

If $N$ has no free indices, the free vertex has no incident edges and the remaining graph is exactly the graph analyzed in prior work. Intuitively, the structure graph of a tensor network $N$ captures how indices are shared by the tensors of $N$. For example, on a CNF formula $\varphi$ Theorem 3.3 produces a tensor network $N_\varphi$ whose structure graph is exactly the *incidence graph* of $\varphi$.

It is also convenient to define the *internal structure graph* of a tensor network $N$, denoted structInt$(N)$, to be struct$(N)$ with the free vertex (and all incident edges) removed.

The structure graph of $N$ contains all information needed to compute the max rank of a contraction-tree of $N$, as formalized in the following lemma.

**Lemma 3.5.** *Let $N$ be a tensor network with structure graph $G$. If $N' \subseteq N$ is nonempty, then $N'$ is a tensor network and $\mathcal{F}(N') = \delta_G[V] \cap \delta_G[\mathcal{V}(G) \setminus N']$.*

*Proof.* $N'$ is a tensor network since $N$ is. Let $z$ be the free vertex of $G$. An index $i$ is free in $N'$ if and only if $i$ appears in $N'$ and either $i$ is free in $N$ or $i$ also appears in $N \setminus N'$. Thus

$$\mathcal{F}(N') = \bigcup_{A \in N'} \mathcal{I}(A) \cap \left( \mathcal{F}(N) \cup \bigcup_{B \in N \setminus N'} \mathcal{I}(B) \right).$$

Since $\mathcal{I}(A) = \delta_G(A)$ for all $A \in N$ and $\mathcal{F}(N) = \delta_G(z)$, we conclude that

$$\mathcal{F}(N') = \delta_G[N'] \cap (\delta_G(z) \cup \delta_G[N \setminus N']) = \delta_G[N'] \cap \delta_G[\mathcal{V}(G) \setminus N'].$$

□

Contraction trees are closely connected to decompositions of the structure graph. In particular, contraction trees of a tensor network correspond to carving decompositions of its structure graph, where max rank corresponds exactly to carving width. This correspondence was first proven for tensor networks with no free indices by de Oliveira Oliveira [120]. Theorem 3.6 extends this correspondence to tensor networks with free indices as well:

**Theorem 3.6.** *Let $N$ be a tensor network with structure graph $G$ and let $w \in \mathbb{N}$. Then $N$ has a contraction tree of max rank $w$ if and only if $G$ has a carving decomposition of width $w$. Moreover, given one of these objects the other can be constructed in $O(|N|)$ time.*

*Proof.* Let $z$ be the free vertex of $G$. First, let $T$ be a contraction tree of $N$ of max rank $w$. Construct $T'$ from $T$ by adding $z$ as a leaf to the root of $T$.

$T'$ is a carving decomposition of $G$, since $T'$ is an unrooted binary tree with $\mathcal{L}(T') = \mathcal{L}(T) \sqcup \{z\} = \mathcal{V}(G)$. For each $a \in \mathcal{E}(T')$, removing $a$ from $T'$ produces two connected components, both trees. Let $T_a$ be the connected component that does not contain $z$ and note that $T_a$ is a contraction tree for $\mathcal{L}(T_a) \subseteq N$.

This gives us a bijection between $\mathcal{E}(T')$ and the set of recursive calls of Algorithm 3.2, where each $a \in \mathcal{E}(T')$ corresponds to the recursive call where $T_a$ is the input contraction tree and $\mathcal{T}(\mathcal{L}(T_a))$ is the output tensor. Thus

$$width_c(T') = \max_{a \in \mathcal{E}(T')} |\delta_G[\mathcal{L}(T_a)] \cap \delta_G[\mathcal{V}(G) \setminus \mathcal{L}(T_a)]| = \max_{a \in \mathcal{E}(T')} |\mathcal{F}(\mathcal{L}(T_a))| = w$$

where the middle equality is given by applying Lemma 3.5 with $N' = \mathcal{L}(T_a)$.

Conversely, let $S$ be a carving decomposition of $G$ of width $w$. Construct $S'$ from $S$ by removing the leaf $z$ (and its incident arc) from $S$. $S'$ is a contraction tree of $N$, since $S$ is a rooted binary tree (whose root is the node previously attached by an arc to $z$) and $\mathcal{L}(S') = \mathcal{L}(S) \setminus \{z\} = N$. Moreover, applying the construction in the first half of the proof produces $S$ and so the max rank of $S'$ is $width_c(S) = w$. $\square$

One corollary of Theorem 3.6 is that tensor networks with isomorphic structure graphs have contraction trees of equal max rank. This corollary is closely related to Theorem 1 of [123].

Carving decompositions have been studied in several settings. For example, there is an algorithm to find a carving decomposition of minimal width of a planar graph in time cubic in the number of edges [128]. It follows that if the structure graph of a tensor network $N$ is planar, one can construct a contraction tree of $N$ of minimal max rank in time $O(|\mathcal{B}(N) \cup \mathcal{F}(N)|^3)$. This may be of interest in domains where problems can be seen as planar or near-planar, e.g. in circuit analysis or infrastructure reliability [104, 119], but most benchmarks we consider in Section 3.5 do not result in planar structure graphs.

There is limited work on the heuristic construction of "good" carving decompositions for non-planar graphs. Instead, we leverage the work behind finding tree decompositions to find carving decompositions and subsequently find contraction trees of small max rank.

One technique for join-query optimization [33, 129] focuses on analysis of the *join graph*. The *join graph* of a project-join query consists of all attributes of a database as vertices and all tables in the join as cliques. In this approach, tree decompositions for the join graph of a query are used to find optimal join trees. The analogous technique on factor graphs analyzes the *primal graph* of a factor graph, which consists of all variables as vertices and all factors as cliques. Similarly, tree decompositions of the primal graph can be used to find variable elimination orders [130]. The graph analogous to join graphs and primal graphs for tensor networks is the *line graph* of the structure graph:

**Definition 3.7** (Line Graph)**.** *The* line graph *of a graph $G$ is a graph Line$(G)$ whose vertices are the edges of $G$, and where the number of edges between each $e, f \in \mathcal{E}(G)$ is $|\epsilon_G(e) \cap \epsilon_G(f)|$, the number of endpoints shared between $e$ and $f$.*

This technique was applied in the context of tensor networks by Markov and Shi [119], who proved that tree decompositions for Line$(G)$ (where $G$ is the structure graph of a tensor network $N$) can be transformed into contraction trees for $N$ of small contraction complexity. Specifically, tree decompositions of optimal width $w$ yield contraction trees of contraction complexity $w + 1$.

In the following theorem we analyze the max rank of the resulting contraction trees, which has not previously been studied. We present this result as a new relationship between carving width and treewidth:

**Theorem 3.8.** *Let $G$ be a graph with $\mathcal{E}(G) \neq \varnothing$. Given a tree decomposition $T$ for Line$(G)$ of width $w \in \mathbb{N}$, one can construct in polynomial time a carving decomposition for $G$ of width no more than $w + 1$.*

In order to prove this theorem, it is helpful to first state and prove a lemma on simplifying the internal structure of a tree decomposition. In particular, we show that given an edge clique cover of a graph $G$, it is sufficient to only consider tree decompositions whose leaves are labeled only by elements of the edge clique cover.

**Lemma 3.9.** *Let $G$ be a graph, let $(T, \chi)$ be a tree decomposition of $G$, and let $A$ be a finite set. If $f : A \to 2^{\mathcal{V}(G)}$ is a function whose image is an edge clique cover of $G$, then we can construct in polynomial time a tree decomposition $(S, \psi)$ of $G$ and a bijection $g : A \to \mathcal{L}(S)$ with $width_t(S, \psi) \leq width_t(T, \chi)$ and $\psi \circ g = f$.*

*Proof.* Consider an arbitrary $a \in A$ and define $\chi_a : \mathcal{V}(T) \to 2^{\mathcal{V}(G)}$ by $\chi_a(n) \equiv \chi(n) \cap f(a)$ for all $n \in \mathcal{V}(T)$. Notice that $(T, \chi_a)$ is a tree decomposition of $G \cap f(a)$.

Since $f$ is an edge clique cover, $G \cap f(a)$ is a complete graph with $|f(a)|$ vertices and thus has treewidth $|f(a)| - 1$. It follows that $width_t(T, \chi_a) \geq |f(a)| - 1$. That is, there is some $n_a \in \mathcal{V}(T)$ such that $|\chi_a(n_a)| \geq |f(a)|$. It follows that $\chi_a(n_a) = f(a)$ and so $f(a) \subseteq \chi(n_a)$. Choose an arbitrary arc $b \in \epsilon_T(n_a)$ and construct $T'$ from $T$ by attaching a new leaf $\ell_a$ at $b$ (and introducing a new internal node). We can extend $\chi$ into a labeling $\chi' : \mathcal{V}(T') \to 2^{\mathcal{E}(G)}$ by labeling the new internal node with $\chi(n_a)$ and labeling the new leaf node with $f(a)$. Note that $(T', \chi')$ is still a tree decomposition of $G$ of width $width_t(T, \chi)$, that all labels of leaves of $(T, \chi)$ are still labels of leaves of $(T', \chi')$, and that the new leaf of $(T'\chi')$, namely $\ell_a$, is labeled by $f(a)$.

By repeating this process for every $a \in A$, by induction we produce in polynomial time a tree decomposition $(T', \chi')$ of width $width_t(T, \chi)$. Moreover, define the function $g : A \to \mathcal{L}(T')$ for all $a \in A$ by $g(a) = \ell_a$ (the new leaf attached at each step). By construction, $\chi' \circ g = f$ (since each $\ell_a$ was labeled by $f(a)$) and moreover $f$ is an injection (since a new leaf was introduced at each step). It remains to make $f$ a bijection by removing leaves of $T'$.

Since $f$ is an edge clique cover, properties (1) and (2) of a tree decomposition can be satisfied purely looking at the nodes of $T'$ in the range of $g$. Moreover, removing leaves of $T'$ cannot falsify property (3) of a tree decomposition. Thus we can repeatedly remove leaves of $T'$ not in the range of $g$ until we eventually reach a tree decomposition $(S, \psi)$ for $G$ whose leaves are exactly the range of $g$. After this process, $g$ is a bijection as a function onto $\mathcal{L}(S)$ and $\psi \circ f = \delta_G$. Moreover, since $\psi(\mathcal{V}(S)) \subseteq \chi'(\mathcal{V}(T'))$ it follows that $width_t(S, \psi) \leq width_t(T', \chi') = width_t(T, \chi)$ as desired. □

We now use this lemma to prove Theorem 3.8.

*Proof of Theorem 3.8.* First, observe that the image of $\delta_G : \mathcal{V}(G) \to \mathcal{E}(G)$ is an edge clique cover of Line$(G)$. It follows by Lemma 3.9 that we can construct a tree decomposition $(S, \psi)$ of Line$(G)$ and a bijection $g : \mathcal{V}(G) \to \mathcal{L}(S)$ such that $\psi \circ g = \delta_G$ and $width_t(S, \psi) \leq width_t(T, \chi)$.

Construct $T'$ from $S$ by replacing every leaf $\ell \in \mathcal{L}(S)$ with $g^{-1}(\ell)$. Since $g$ is a bijection, $\mathcal{L}(T') = \mathcal{V}(G)$ and so $T'$ is a carving decomposition. In order to compute the carving width of $T'$, consider an arbitrary arc $a \in \mathcal{E}(T')$ and let $C_a$ be an element of the partition of $\mathcal{V}(G)$ defined by removing $a$.

For every edge $e \in \delta_G[C_a] \cap \delta_G[\mathcal{L}(T') \setminus C_a]$, there must be vertices $v \in C_a$ and $w \in \mathcal{L}(T') \setminus C_a$ that are both incident to $e$ and so $e \in \delta_G(v) \cap \delta_G(w)$. Since $\chi \circ g = \delta_G$, it follows that $e \in \chi(g(v)) \cap \chi(g(w))$. Property 3 of tree decompositions implies that $e$ must also be in the label of every node in the path from $g(v)$ to $g(w)$ in $T$; in particular, $e$ must be in the label of both endpoints of $a$.

Thus every element of $\delta_G[C_a] \cap \delta_G[\mathcal{L}(T') \setminus C_a]$ must be in the label of both endpoints of $a$. It follows that $|\delta_G[C_a] \cap \delta_G[\mathcal{L}(T') \setminus C_a]| \leq width_t(T, \chi) + 1$. Hence $width_c(T') \leq width_t(T, \chi) + 1$ as desired. $\square$

An alternative proof of Theorem 3.8 uses Theorem 2.4 of [131] to construct a carving decomposition $T'$ of $G$ from $T$ whose vertex congestion is $w + 1$. By Lemma 2 of [132], it follows that the carving width of $T'$ is no more than $w + 1$.

Applying Theorem 3.8 when $G$ is the structure graph of a tensor network (together with Theorem 3.6) gives us the **Line-Graph** planner, which finds contraction trees by finding tree decompositions of the corresponding line graph. Note that the **Line-Graph** planner does not modify the input tensor network and so trivially satisfies Assumption 2 in Theorem 3.2.

There are several advantages to our new analysis over the analysis of [119]: our analysis holds for tensor networks with free indices, and we analyze the max rank of contraction trees instead of the contraction complexity. Although the contraction complexity (and, for factor graphs, the width of the elimination order) is equal to one plus the width of the used tree decomposition, the max rank is smaller on some graphs.

### 3.4.3 The Factor-Tree Planner

Approaches to tensor-network contraction planning that do not modify the input tensor network (e.g., **LG**) are inherently limited by the ranks of the input tensors. If a tensor network $N$ has a rank $r$ tensor, then all contraction trees for $N$ will have max rank of at least $r$. This is a problem for tensor networks with high-rank tensors.

One example of tensor networks that may contain high-rank tensors are the networks obtained by the reduction from model counting. The tensor network produced from a formula $\varphi$ contains a tensor representing each variable $x$, where the rank of this tensor is the number of appearances of $x$ in $\varphi$ (e.g., the rank 4 tensor for $y$ in Figure 3.1). For many benchmarks, where a single variable might appear tens or even hundreds of times, this reduction will therefore produce tensor networks containing tensors of infeasibly-high rank. Reductions exist from model counting on arbitrary formulas to model counting on formulas where the number of appearances of each variables is small (e.g. [133] gives a parsimonious reduction from 3-SAT to satisfiability of formulas where every variable appears at most 3 times). However, existing reductions do not consider the carving width of the resulting incidence graph and so often do not significantly improve the max rank of available contraction trees.

We introduce here a novel planning method **Factor-Tree** that avoids this barrier by preprocessing the input tensor network. Our insight is that a tree decomposition for the incidence graph of $\varphi$ can be used as a guide to introduce new variables in a principled way, so that the resulting tensor network has good contraction trees. In the language of tensors, introducing new variables corresponds to *factoring*: replacing each high-rank tensor $A$ with a tensor network $N_A$ of low-rank tensors that contracts to $A$.

One way to see intuitively the impact of factoring is to consider running **LG** before and after factoring a rank $r$ tensor $A$ in a tensor network $N$. Before factoring, $\mathcal{I}(A)$ is a clique of $r$ vertices in Line(struct($N$)) and thus Line(struct($N$)) will have treewidth at least $r - 1$. After factoring and replacing $A$ by $N_A$, this clique is replaced with

Line(structInt($N_A$)). This new graph has a significantly sparser set of edges and so may have a lower treewidth.

The key idea of **FT**, then, is to use a tree decomposition for the structure graph to factor high-rank tensors. We state this new result as Theorem 3.11. Note that there is no need to run **LG** independently after using **FT** to factor high-rank tensors. Instead, in Theorem 3.11 we directly extract a low-rank contraction tree for the factored network as part of the factoring process of **FT**.

Since not all tensors can be factored in the ways that we require for this theorem and for **FT**, we first characterize the required property: that every tensor is factorable as an arbitrary tree of tensors:

**Definition 3.10.** *A tensor $A$ is* tree factorable *if, for every tree $T$ whose leaves are $\mathcal{I}(A)$ (called a* dimension tree *of $A$), there is a tensor network $N_A$ and a bijection $g_A : \mathcal{V}(T) \to N_A$ s.t.*

1. *$A$ is the contraction of $N_A$,*
2. *$g_A$ is an isomorphism between $T$ and structInt($N_A$),*
3. *for every index $i$ of $A$, $i$ is an index of $g_A(i)$, and*
4. *for some index $i$ of $A$, the bond dimension of $N_A$ is no bigger than $|[i]|$.*

Recall that structInt($N_A$) is the internal structure graph of $N_A$, i.e. the structure graph of $N_A$ with the free vertex (and incident edges) removed. All tensors in the reduction of Theorem 3.3 from weighted model counting to tensor networks are tree factorable. A tensor network $N_A$ that satisfies properties 1, 2, and 3 of Definition 3.10 for some tree is called a *Hierarchical Tucker representation* of $A$ [122]. Property 4 ensures the result of Theorem 3.11 has small bond dimension.

We now state the main result of this section, which allows us to use a tree decomposition for the structure graph of a tensor network (containing only tree factorable

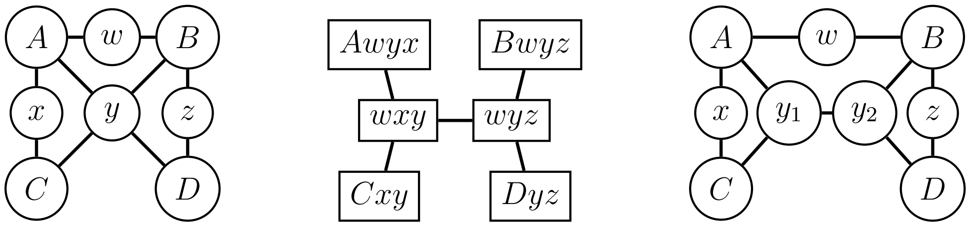


Figure 3.2 : When FT is run on the shown initial tensor network (left) using the shown tree decomposition (middle), **FT** produces a factored tensor network (right). Tensors of rank 3 or smaller are unchanged, and the tensor for $y$ is factored into two tensors, $y_1$ and $y_2$, each of rank 3. The factored tensor network has a contraction tree of max rank 3 while the initial tensor network only has contraction trees of max rank 4 or higher.

tensors) to factor each tensor in the network and find a contraction tree of low max rank for the resulting network:

**Theorem 3.11.** *Let $N$ be a tensor network of tree-factorable tensors s.t. $|\mathcal{F}(N)| \leq 3$ and the structure graph of $N$ has a tree decomposition of width $w \geq 1$.*

*Then for each $A \in N$ there is a tensor network $N_A$ whose contraction is $A$ that consists only of rank 3 or smaller tensors. Moreover, the disjoint union of these networks, $M = \cup_{A \in N} N_A$, is a tensor network that contracts to $\mathcal{T}(N)$, has the same bond dimension as $N$, and has a contraction tree of max rank no larger than $\lceil 4(w+1)/3 \rceil$.*

*Proof.* The proof proceeds in five steps: (1) compute the factored tensor network $M$, (2) construct a graph $H$ that is a simplified version of the structure graph of $M$, (3) construct a carving decomposition $S$ of $H$, (4) bound the width of $S$, and (5) use $S$ to find a contraction tree for $M$. Working with $H$ instead of directly working with the structure graph of $M$ allows us to cleanly handle tensor networks with free indices.

**Part 1: Factoring the network.** Let $G$ be the structure graph of $N$ with all degree 0 vertices removed; $G$ must also have a tree decomposition of width $w$. Moreover, the image of $\epsilon_G : \mathcal{E}(G) \to 2^{\mathcal{V}(G)}$ is an edge clique cover of $G$. Thus using Lemma 3.9 we can construct a tree decomposition $(T, \chi)$ of $G$ and a bijection

$g : \mathcal{E}(G) \to \mathcal{L}(T)$ such that $\chi \circ g = \epsilon_G$ and $width_t(T, \chi) \leq w$.

Next, for each $v \in \mathcal{V}(G)$, define $T_v$ to be the smallest connected component of $T$ containing $\{g(i) \ : \ i \in \delta_H(v)\}$. Consider each $A \in N$. If $\mathcal{F}(A) = \varnothing$, let $N_A = \{A\}$; thus $\mathcal{T}(N_A) = A$ and the tensor of $N_A$ has rank 0. Otherwise, observe that $T_A$ is a dimension tree of $A$. We can therefore factor $A$ with $T_A$ using Definition 3.10 to get a tensor network $N_A$ whose contraction is $A$ and a bijection $g_A : \mathcal{V}(T_A) \to N_A$. Moreover, every node of $T_A$ has degree 3 or smaller and so, by Definition 3.10, $N_A$ consists only of rank 3 or smaller tensors.

Define $M = \cup_{A \in N} N_A$ and let $G'$ be the structure graph of $M$ with free vertex $z'$. The remainder of the proof is devoted to bounding the carving width of $G'$.

**Part 2: Constructing a simplified structure graph of $M$.** In order to easily characterize $G'$, we define a new, closely-related graph $H$ by taking a copy of $T_v$ for each $v \in \mathcal{V}(G)$ and connecting these copies where indicated by $g$. Formally, the vertices of $H$ are $\{(v, n) : v \in \mathcal{V}(G), n \in \mathcal{V}(T_v)\}$. For every $v \in \mathcal{V}(G)$ and every arc in $T$ with endpoints $n, m \in \mathcal{V}(T_v)$, we add an edge between $(v, n)$ and $(v, m)$. Moreover, for each $e \in \mathcal{E}(G)$ incident to $v, w \in \mathcal{V}(G)$, we add an edge between $(v, g(e))$ and $(w, g(e))$.

We will prove in Part 5 that the carving width of $G'$ is bounded from above by the carving width of $H$. We therefore focus in Part 3 and Part 4 on bounding the carving width of $H$. It is helpful for this to define the two projections $\pi_G : \mathcal{V}(H) \to \mathcal{V}(G)$ and $\pi_T : \mathcal{V}(H) \to \mathcal{V}(T)$ that indicate respectively the first or second component of a vertex of $H$.

**Part 3. Constructing a carving decomposition $S$ of $H$.** The idea of the construction is, for each $n \in \mathcal{V}(T)$, to attach the elements of $\pi_T^{-1}(n)$ as leaves along arcs incident to $n$. See Figure 3.3 for an overview of the construction.

Consider an arbitrary node $n \in \mathcal{V}(T)$. $n$ is either a leaf node or and internal node of $T$. If $n$ is a leaf node, let $a \in \delta_T(n)$ be the arc incident to $n$ and define $H_{n,a} = \pi_T^{-1}(n)$. If $n$ is an internal node, let $a, b, c \in \delta_T(n)$ be the arcs incident to $n$ (note that $T$ is a

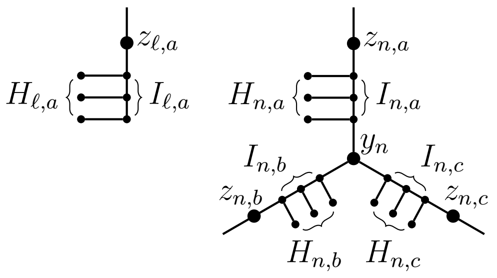


Figure 3.3 : The central construction in Part 3 of Theorem 3.11 of a carving decomposition $S$ from an input tree decomposition $(T, \chi)$. Each leaf $\ell$ of $T$ (with incident arc $a$) is replaced by the left construction in $S$. Each internal node $n$ of $T$ (with incident arcs $a$, $b$, and $c$) is replaced by the right construction in $S$. These constructions are attached together according to the arcs of $T$: if nodes $m$ and $n$ are connected by arc $a$ in $T$, then nodes $z_{m,a}$ and $z_{n,a}$ are connected in $S$.

binary tree and so $n$ has exactly three incident arcs), arbitrarily partition $\pi_T^{-1}(n)$ into three equally-sized sets $B_1$, $B_2$, and $B_3$, and finally define $H_{n,a} = B_1$, $H_{n,b} = B_2$, and $H_{n,c} = B_3$. Observe that, in either case, $\{H_{n,a} : n \in \mathcal{V}(T), a \in \delta_T(n)\}$ is a partition of $\pi_T^{-1}(n)$ and thus is a partition of $\mathcal{V}(H)$.

We use this to construct a carving decomposition $S$ from $T$ by adding each element of $H_{n,a}$ as a leaf along the arc $a$. Formally, let $x_v$ denote a fresh vertex for each $v \in \mathcal{V}(H)$, let $y_n$ denote a fresh vertex for each $n \in \mathcal{V}(T)$, and let $z_{n,a}$ denote a fresh vertex for each $n \in \mathcal{V}(T)$ and $a \in \delta_T(n)$. Define $\mathcal{V}(S)$ to be the union of $\mathcal{V}(H)$ with the set of these free vertices.

We add an arc between $v$ and $x_v$ for every $v \in \mathcal{V}(H)$. Moreover, for every $a \in \mathcal{E}(T)$ with endpoints $o, p \in \epsilon_T(a)$ add an arc between $z_{o,a}$ and $z_{p,a}$. For every $n \in \mathcal{V}(T)$ and incident arc $a \in \delta_T(n)$, construct an arbitrary sequence $I_{n,a}$ from $\{x_v : v \in H_{n,a}\}$. If $H_{n,a} = \varnothing$ then add an arc between $y_n$ and $z_{n,a}$. Otherwise, add arcs between $y_n$ and the first element of $I$, between consecutive elements of $I_{n,a}$, and between the last element of $I_{n,a}$ and $z_{n,a}$.

Finally, remove the previous leaves of $T$ from $S$. The resulting tree $S$ is a carving decomposition of $H$, since we have added all vertices of $H$ as leaves and removed the previous leaves of $T$.

**Part 4: Computing the width of $S$.** In this part, we separately bound the width of the partition induced by each of the three kinds of arcs in $S$.

First, consider an arc $d$ between some $v \in \mathcal{V}(H)$ and $x_v$. Since all vertices of $H$ are degree 3 or smaller, $d$ defines a partition of width at most $3 \leq \lceil 4(w+1)/3 \rceil$.

Next, consider an arc $e_a$ between $z_{o,a}$ and $z_{p,a}$ for some arc $a \in \mathcal{E}(T)$ with endpoints $o, p \in \epsilon_T(a)$. Observe that removing $a$ from $T$ defines a partition $\{B_o, B_p\}$ of $\mathcal{V}(T)$, denoted so that $o \in B_o$ and $p \in B_p$. Then removing $e_a$ from $S$ defines the partition $\{\pi_T^{-1}(B_o), \pi_T^{-1}(B_p)\}$ of $\mathcal{L}(S)$. By construction of $H$, all edges between $\pi_T^{-1}(B_o)$ and $\pi_T^{-1}(B_p)$ are between $\pi_T^{-1}(o)$ and $\pi_T^{-1}(p)$. Since $\pi_G(\pi_T^{-1}(o)) \subseteq \chi(o)$, $\pi_G(\pi_T^{-1}(o)) \subseteq \chi(p)$, and $\pi_G$ is an injection on $\pi_T^{-1}(n)$ for all $n \in \mathcal{V}(T)$), it follows that the partition defined by $e_a$ has width no larger than $|\chi(o) \cap \chi(p)| \leq w + 1$.

Finally, consider an arc $f$ added as one of the sequence of $|H_{n,a}|+1$ arcs between $y_n$, $I_{n,a}$, and $z_{n,a}$ for some $n \in \mathcal{V}(T)$ and $a \in \delta_T(n)$. Some elements of $H_{n,a}$ have changed blocks from the partition defined by $e_a$. Each vertex of degree 2 that changes blocks does not affect the width of the partition, but each vertex of degree 3 that changes blocks increases the width by 1. There are at most $|H_{n,a}| \leq \lceil (w+1)/3 \rceil$ elements of degree 3 added as leaves between $y_n$ and $z_{n,a}$. Thus the partition defined by $f$ has width at most $w + 1 + \lceil (w+1)/3 \rceil = \lceil 4(w+1)/3 \rceil$.

It follows that the width of $S$ is at most $\lceil 4(w+1)/3 \rceil$.

**Part 5: Bounding the max rank of $M$.** Let $z$ be the free vertex of the structure graph of $N$. We first construct a new graph $H'$ from $H$ by, if $\mathcal{F}(N) \neq \varnothing$, contracting all vertices in $\pi_G^{-1}(z)$ to a single vertex $z$. If $\mathcal{F}(N) = \varnothing$, instead add $z$ as a fresh degree 0 vertex to $H'$. Moreover, for all $A \in N$ with $\mathcal{I}(A) = \varnothing$ add $A$ as a degree 0 vertex to $H'$.

Note that adding degree 0 vertices to a graph does not affect the carving width. Moreover, since $|\mathcal{F}(N)| \leq 3$ all vertices (except at most one) of $\pi_G^{-1}(z)$ are degree 2 or smaller. It follows that contracting $\pi_G^{-1}(z)$ does not increase the carving width.

Thus the carving width of $H'$ is at most $\lceil 4(w+1)/3 \rceil$.

Moreover, $H'$ and $G'$ are isomorphic. To prove this, define an isomorphism $\varphi : \mathcal{V}(H') \to \mathcal{V}(G')$ between $H'$ and $G'$ by, for all $v \in \mathcal{V}(H')$:

$$\varphi(v) \equiv \begin{cases} v & \text{if } v \in N \text{ and } \mathcal{I}(v) = \varnothing \\ z' & v = z \\ g_{\pi_G(v)}(\pi_T(v)) & \text{if } v \in \mathcal{V}(H) \text{ and } \pi_G(v) \in N \end{cases}$$

$\varphi$ is indeed an isomorphism between $H'$ and $G'$ because the functions $g_A$ are all isomorphisms and because an edge exists between $\pi_G^{-1}(v)$ and $\pi_G^{-1}(w)$ for $v, w \in \mathcal{V}(G)$ if and only if there is an edge between $v$ and $w$ in $G$. Thus the carving width of $G'$ is at most $\lceil 4(w+1)/3 \rceil$. By Theorem 3.6, then, $M$ has a contraction tree of max rank no larger than $\lceil 4(w+1)/3 \rceil$. □

The general idea of using a tree decomposition of a graph to split high-degree nodes was previously used by Markov and Shi [134] and in the context of constraint satisfaction by Samer and Szeider [135]. Both of these works focus on minimizing the treewidth of the factored graph instead of the max rank. Translated to tensor networks (as done in Lemma 3 of [136]), their constructions produce a factored network $N'$ with structure graph $G$ that satisfies all requirements of Theorem 3.11 except with a bound of $w+1$ on the treewidth of $G$ in place of the bound on max rank. Since the treewidth of Line($G$) plus 1 is bounded by the product of the maximum degree of $G$ (namely 3) and the treewidth of $G$ plus 1 [119], we can then use **LG** to produce a contraction tree for $N'$ of max rank no larger than $3(w+2)$. Theorem 3.11 thus gives an improvement on max rank over these prior works from $3(w+2)$ to $\lceil 4(w+1)/3 \rceil$.

The construction of Theorem 3.11 gives us the **Factor-Tree** planner, which uses tree decompositions of the structure graph to preprocess the tensor network and factor high-rank tensors. See Figure 3.2 for an example of the preprocessing. Since $M$ contracts to $\mathcal{T}(N)$ in Theorem 3.11, the **Factor-Tree** planner satisfies Assumption 2 in Theorem 3.2.

We emphasize that the **Factor-Tree** planner (and Theorem 3.11) takes as input a tree decomposition of the structure graph $G$ directly, while the **Line-Graph** planner (and Theorem 3.8) takes as input a tree decomposition of the line graph of $G$. For many tensor networks the treewidth of the structure graph $G$ is much less than the treewidth of Line($G$).

For example, let $\psi = (\vee_{i=1}^{n} x_i) \wedge (\vee_{i=1}^{n} \neg x_i)$ and let $N_\psi$ be the tensor network obtained from applying Theorem 3.3 with a constant weight function. Then struct($N_\psi$) has treewidth 2 and Line(struct($N_\psi$)) has treewidth $n-1$. Intuitively, this is because $N_\varphi$ has a rank $n$ tensor corresponding to each clause of $\psi$. Each rank $n$ tensor induces a clique of $n$ vertices in Line(struct($N_\psi$)), but only a degree $n$ vertex in struct($N_\psi$).

We show in Section 3.5.3 that this difference also holds for real-world benchmarks. That is, **FT** can significantly improve the quality of the contraction tree on real-world benchmarks with high-rank tensors.

## 3.5 Implementation and Evaluation

We aim to answer the following experimental research questions:

(RQ1) Are tensor-network-based approaches competitive with existing state-of-the-art unweighted model counters?

(RQ2) Are tensor-network-based approaches competitive with existing state-of-the-art weighted model counters?

(RQ3) What are the structural properties of benchmarks for which the tensor-network-based approaches perform well?

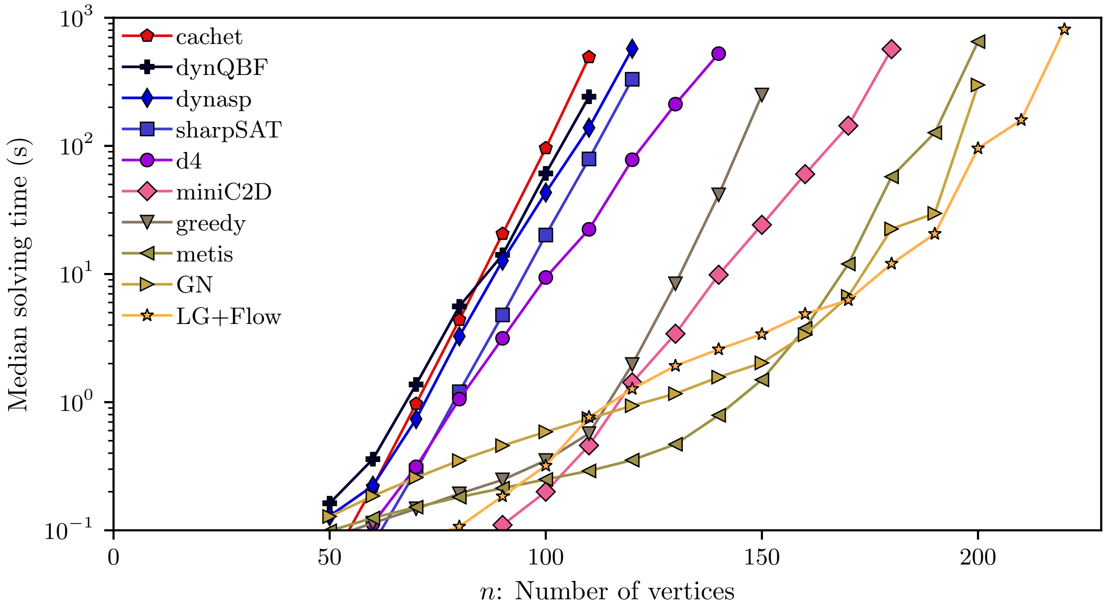
To answer these questions, we implement Algorithm 2 in `TensorOrder`, a new tool for weighted model counting using tensor networks. `TensorOrder` can be configured to find contraction trees using existing planning methods – **greedy** (using a greedy algorithm), **metis** (using graph partitioning), and **GN** (using community structure detection) [66]– or planning methods presented in this paper– **LG** (Section 3.4.2) and **FT** (Section 3.4.3). Implementation details appear in Section 3.5.1.

To answer RQ1, in Section 3.5.2 we compare `TensorOrder` with existing state-of-the-art tools for unweighted model counting (`dynQBF` [25], `dynasp` [26], `SharpSAT` [20], `cachet` [19], `miniC2D` [21] and `d4` [22]) on formulas that count the number of vertex covers of randomly-generated cubic graphs [66]. Note `dynQBF` and `dynasp` are solvers from related domains (that can be used as tools for unweighted model counting) that also use tree decompositions.

To answer RQ2, in Section 3.5.3 we compare `TensorOrder` with existing state-of-the-art tools for weighted model counting (`cachet` [19], `miniC2D` [21] and `d4` [22]) on formulas whose weighted count corresponds to exact inference on Bayesian networks [19]. Note that the other tools (`dynQBF`, `dynasp`, and `SharpSAT`) cannot perform weighted model counting.

To answer RQ3, we compute upper bounds on the treewidth and carving width of these benchmarks. We run each of three heuristic tree decomposition solvers (`Tamaki` [47], `FlowCutter` [46], and `htd` [45]) on each benchmark with a timeout of 1000 seconds: once on the structure graph $G$ corresponding to the benchmark, and once on $\text{Line}(G)$. The minimal width of all produced tree decompositions for $G$ (resp. $\text{Line}(G)$) is an upper bound for the treewidth of $G$ (resp. $\text{Line}(G)$). The minimal max rank of the contraction trees produced by running **LG** (resp. **FT**) on each tree decomposition of $\text{Line}(G)$ (resp. $G$) is an upper bound for the carving width of $G$ (resp. $G$ after FT-preprocessing).

Each experiment was run in a high-performance cluster (Linux kernel 2.6.32) using a single 2.80 GHz core of an Intel Xeon X5660 CPU and 48 GB RAM. We provide all code, benchmarks, and detailed data of benchmark runs at https://github.com/vardigroup/TensorOrder.



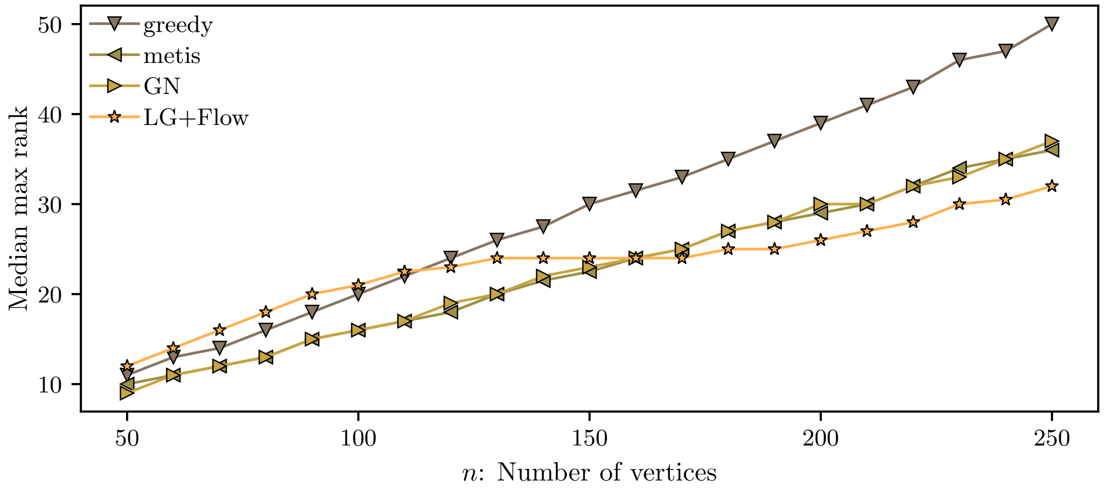


Figure 3.4 : Median solving time (top) and max rank of the computed contraction tree (bottom) of various counters and tensor-based methods, run on benchmarks counting the number of vertex covers of 100 cubic graphs with $n$ vertices. Solving time of datapoints that ran out of time (1000 seconds) or memory (48 GB) are not shown. When $n \geq 170$, our contribution **LG+Flow** is faster than all other methods and finds contraction trees of lower max rank than all other tensor-based methods.

### 3.5.1 Implementation Details of `TensorOrder`

`TensorOrder` is implemented in Python 3.6. All tensor contractions are performed using `numpy` 1.15 and 64-bit double precision floats. `TensorOrder` also supports infinite-precision integer arithmetic, but the performance is significantly degraded by limited `numpy` support. Note that `numpy` is able to leverage SIMD parallelism for tensor contraction.

Both **LG** and **FT** require first finding a tree decomposition. To do this, we leverage three heuristic tree-decomposition solvers: `Tamaki` [47], `FlowCutter` [46], and `htd` [45]. `TensorOrder` therefore has three implementations of **LG** (using **LG+Tamaki**, **LG+Flow**, and **LG+htd**) and three implementations of **FT** (using **FT+Tamaki**, **FT+Flow**, and **FT+htd**) for different choices of solver.

All the tree-decomposition solvers we consider are anytime solvers and so each implementation must decide how long to run the solver (this time is included in the measured running time). In Algorithm 3.1, this is governed by the parameter $\alpha$. `TensorOrder` estimates the time to contract each potential contraction tree (using techniques from the `einsum` package of `numpy`) and configures $\alpha$ so that it continues to look for better tree decompositions until it expects to have spent more than half of the running time on finding a tree decomposition. This strikes a balance between improving and using the contraction trees.

### 3.5.2 Unweighted Model Counting: Vertex Covers of Cubic Graphs

We first compare on benchmarks that count the number of vertex covers of randomly-generated cubic graphs [66]. For each number of vertices $n \in \{50, 60, 70, \cdots, 250\}$ we randomly sample 100 connected cubic graphs using a Monte Carlo procedure [137]. These benchmarks are monotone 2-CNF formulas in which every variable appears 3 times. We run each tool once on each benchmark with a timeout of 1000 seconds and record the wall-clock time taken.

Results on the runtime performance for these benchmarks are summarized in

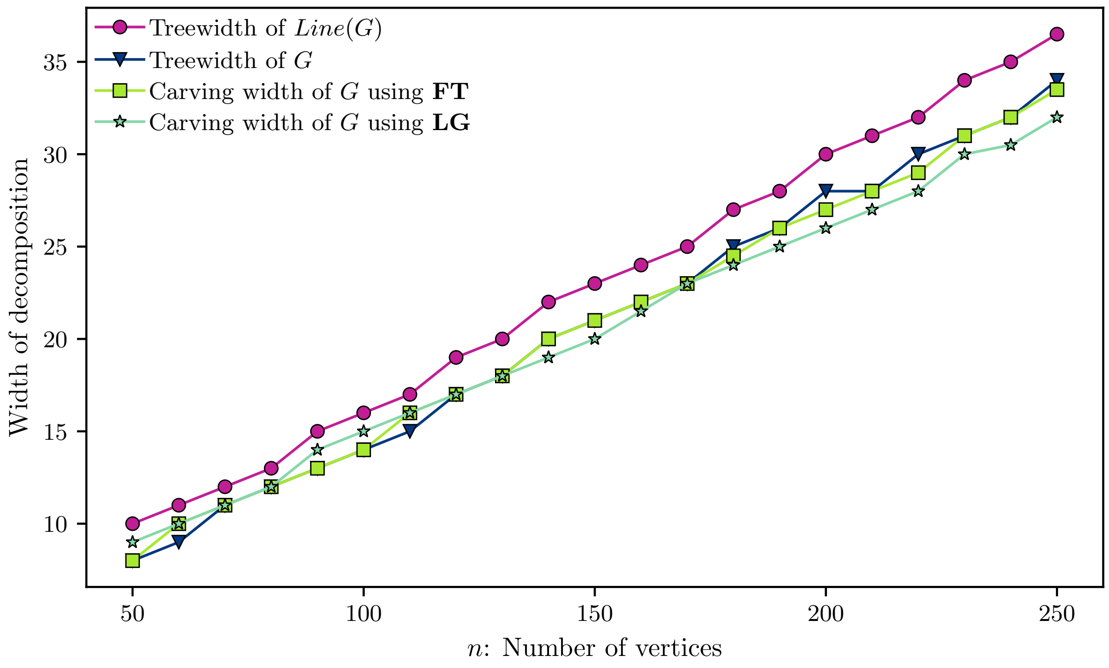


Figure 3.5 : Median of the best upper bound found for treewidth and carving width of 100 cubic graphs with $n$ vertices. For most large graphs, the carving width of $G$ is smaller than the treewidth of $G$, which is smaller that the treewidth of Line($G$).

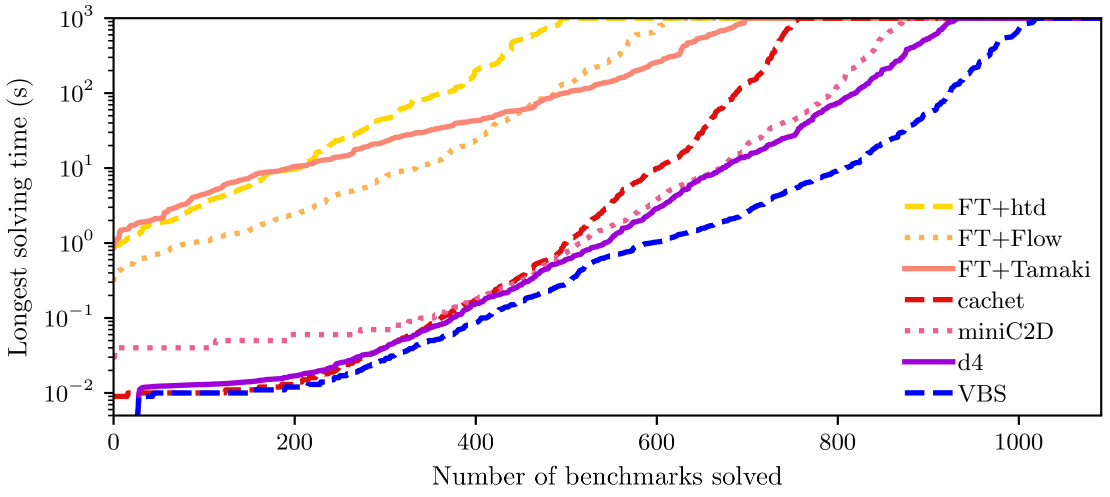


Figure 3.6 : A cactus plot of the number of benchmarks solved by various methods out of 1091 probabilistic inference benchmarks. Although our contributions **FT+*** solve fewer benchmarks than the existing weighted model counters `cachet`, `miniC2D`, and `d4`, they improve the virtual best solver on 231 benchmarks. Note that `dynQBF`, `dynasp`, and `SharpSAT` are unweighted model counters and so cannot solve these weighted benchmarks.

Figure 3.4. For ease of presentation, we display only the best-performing of the **LG** and **FT** implementations: **LG+Flow**. We observe that tensor-based methods are fastest when $n \geq 110$. On large graphs ($n \geq 170$) our contribution **LG+Flow** is fastest and able to find the lowest max rank contraction trees. **LG+Flow** is the only implementation able to solve at least 50 benchmarks within 1000 seconds when $n$ is 220. We conclude that tensor-network-based approaches outperform state-of-the-art unweighted model counters on these benchmarks.

Results on the structural properties of these benchmarks are summarized in Figure 3.5. For most large graphs $G$, we observe that the carving width of $G$ is smaller than the treewidth of $G$ which is smaller that the treewidth of Line($G$).

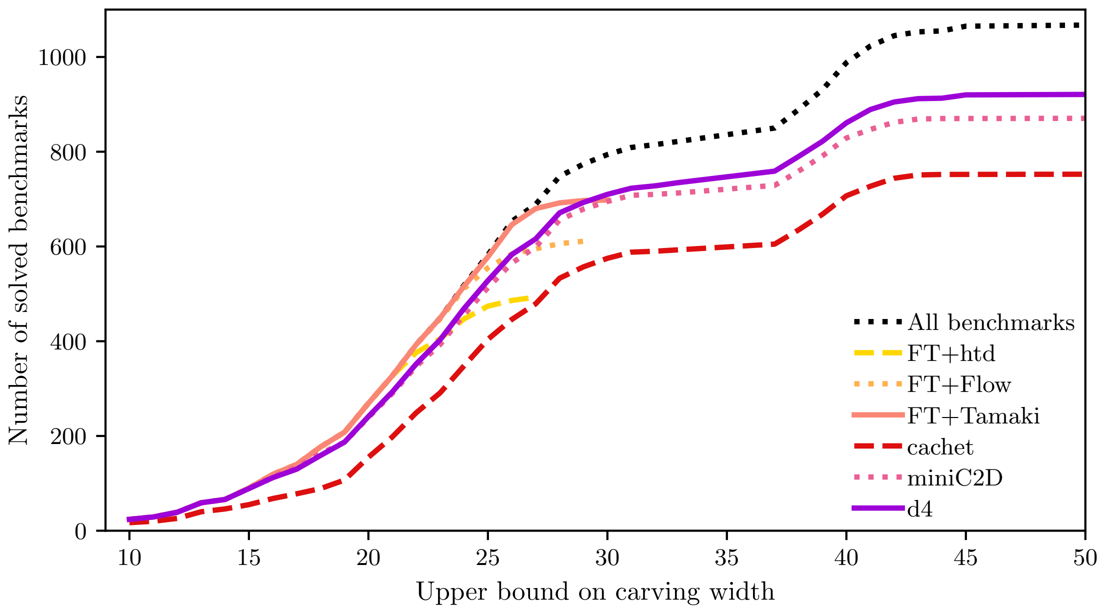


Figure 3.7 : A plot of the number of benchmarks solved by various methods organized by carving width. Each $(x, y)$ data point indicates that the corresponding tool was able to solve $y$ benchmarks whose carving width (after **FT**-preprocessing) was at most $x$. Our approach **FT+Tamaki** can solve almost all benchmarks with carving width below 27 (unlike existing model counters, which fail on many benchmarks with small carving width) and no benchmarks with carving width above 30.

### 3.5.3 Weighted Model Counting: Exact Inference

We next compare on a set of weighted model counting benchmarks from Sang, Beame, and Kautz [19]. These 1091 benchmarks are formulas whose weighted model count corresponds to exact inference on Bayesian networks. We compare against the weighted model counters `cachet` [19], `miniC2D` [21] and `d4` [22]. Since these benchmarks are weighted, we cannot compare against tools that can only perform unweighted model counting (`dynQBF` [25], `dynasp` [26] and `SharpSAT` [20]). We run each tool once on each benchmark with a timeout of 1000 seconds and record the wall-clock time taken.

We first evaluate numerical accuracy, since our approach uses 64-bit double precision floats: on all benchmarks that `miniC2D` also finishes, the weighted model count returned by our approaches agrees within $10^{-3}$.

We next evaluate runtime performance. Results on these benchmarks are summarized in Figure 3.6. **FT+Tamaki** is able to solve the most benchmarks of all tensor-based methods. Our implementations of **FT** each solve fewer benchmarks than `cachet`, `miniC2D`, and `d4`. Nevertheless, **FT+*** are together able to solve 231 benchmarks faster than existing counters (**FT+Tamaki** is fastest on 50, **FT+Flow** is fastest on 175, and **FT+htd** is fastest on 6), including 62 benchmarks on which `cachet`, `miniC2D`, and `d4` all time out. This significantly improves the virtual best solver (VBS) when **FT+*** are included. We conclude that **FT** is useful as part of a portfolio of weighted model counters.

The existing tensor-based methods (**LG**, **greedy**, **metis**, and **GN**) that do not perform factoring were only able to count a single benchmark from this set within 1000 seconds. We observe that most of these benchmarks have a variable that appears many times, which significantly hinders tensor-based methods that do not perform factoring. This justifies our motivation for **FT** in Section 3.4.3.

We next evaluate the structural properties of benchmarks for which **FT** outperforms other approaches. In Figure 3.7, we organize the number of benchmarks completed for each tool by carving width after **FT**-preprocessing. We observe that all tensor-based methods perform best on benchmarks with small carving width. In particular, **FT+Tamaki** was able to solve almost all benchmarks whose width is below 27. On the other hand, existing tools do not heavily rely on structural properties and so solve fewer low-width benchmarks than **FT+Tamaki** but significantly more high-width benchmarks. We conclude that tensor-network-based approaches perform well on benchmark instances of low carving width (after **FT**-preprocessing).

Finally, we are interested in explaining the relative performance of the tensor-based methods **FT+Tamaki**, **FT+Flow**, and **FT+htd** on these benchmarks. To

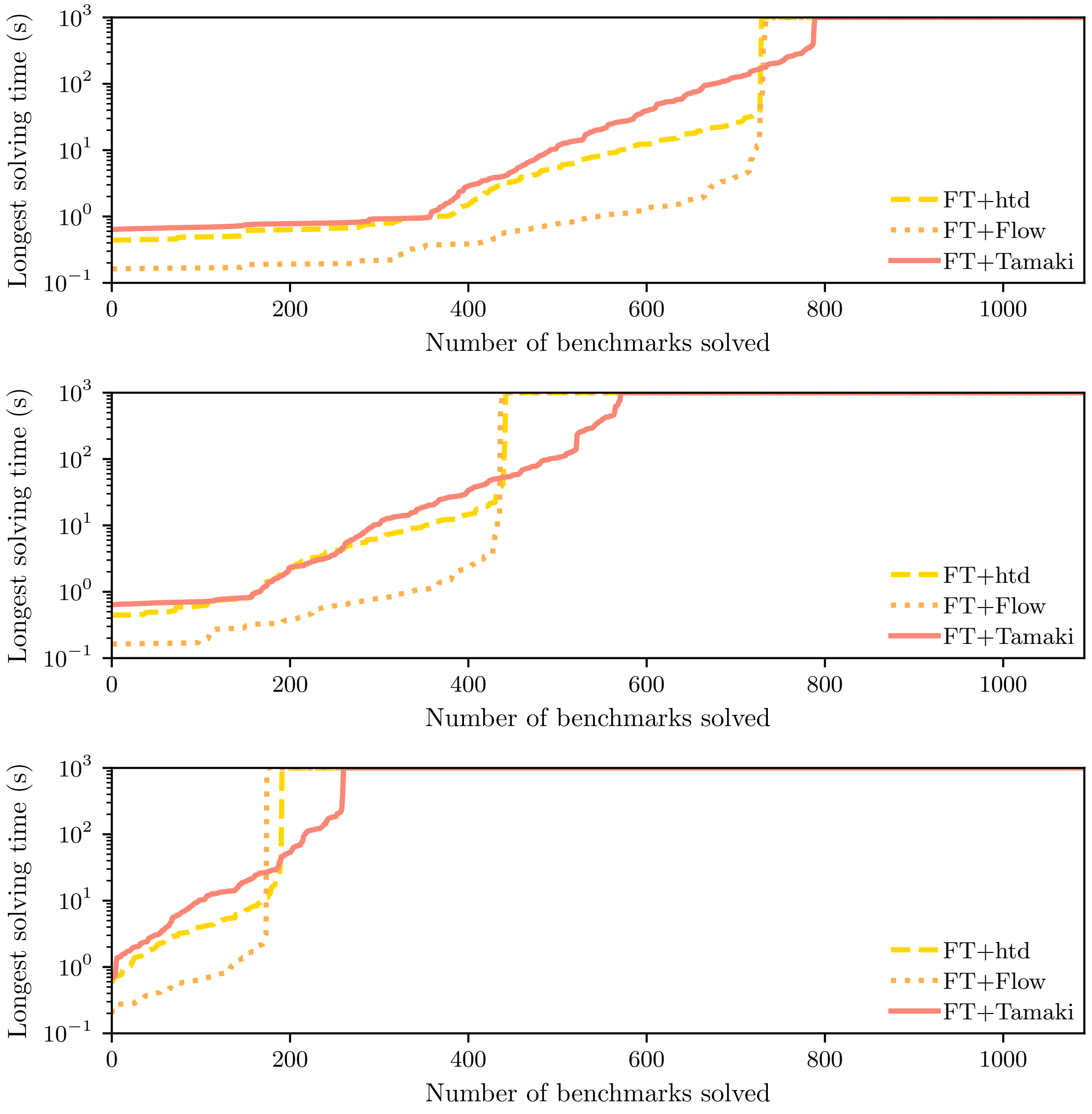


Figure 3.8 : The number of probabilistic-inference benchmarks (out of 1091) for which **FT+Tamaki**, **FT+Flow**, and **FT+htd** were able to find a contraction tree whose max rank was no larger than (top) 30, (middle) 25, or (bottom) 20 within the indicated time.

do this, we analyze the quality of the contraction trees they produce over time. Specifically, we rerun each implementation of **FT** for 1000 seconds with the contraction step disabled (i.e. remove step 3 of Algorithm 3.1). Each implementation of **FT** is an online solver and so produces a sequence of contraction trees over time. For each contraction tree produced on each benchmark, we record the max rank and time of production.

Results of this experiment are summarized in Figure 3.8. **FT+Flow** is able to find more contraction trees of small max rank within 10 seconds than the other methods, while **FT+Tamaki** is able to find more contraction trees of small max rank within 1000 seconds than the other methods. This matches our previous observations that, among the tensor-based methods, **FT+Flow** was the fastest method on the most benchmarks while **FT+Tamaki** was able to solve the most benchmarks after 1000 seconds.

We conclude from the experiments in Section 3.5.2 and Section 3.5.3 that both **LG** and **FT** are useful as part of a portfolio of model counters.

## 3.6 Chapter Summary

We presented a 3-phase algorithm to perform weighted model counting. First, in the reduction phase, a weighted model counting instance is reduced to a tensor-network problem using a novel reduction. Second, in the planning phase, an order to contract tensors in the network is determined. We presented two planning techniques, **LG** and **FT**, for using graph decompositions to find contraction trees of small max rank of tensor networks. **LG** is a general-purpose method for finding contraction orders. **FT** is a novel method tailored for constrained counting to handle high-rank, highly-structured tensors. Finally, in the execution phase, tensors in the network are contracted with state-of-the-art tensor libraries using the determined order. We implemented our algorithm in `TensorOrder`, a new tool for weighted model counting through tensor network contraction, and evaluated `TensorOrder` empirically in the context of model counting.

The primary finding of this chapter is that tensor-network approaches, including our `TensorOrder` using the **LG** planner, outperform a variety of other counters on a set of unweighted model counting benchmarks. We further found that our new **FT** planner significantly outperforms existing planning approaches on exact inference benchmarks, where variables can appear many times, by finding contraction trees that have significantly smaller max-rank and so require significantly less memory to execute. The demonstrates the value of factoring for real-world benchmarks. Overall, `TensorOrder` is able to solve many benchmarks solved by no other (exact) counter, and in particular `TensorOrder` is the best tool on large benchmarks of limited carving width. Thus `TensorOrder` is useful as part of a portfolio of weighted model counters.

# Chapter 4

# DPMC: ADDs and Graph Decompositions with Project-Join Trees

Dynamic programming has been the basis of several tools for model counting [29,31, 80,82]. Although each tool uses a different data structure–algebraic decision diagrams (ADDs) [29], tensors (see Chapter 3), or database tables [31]–the overall algorithms have similar structure. The goal of this chapter is to unify these approaches into a single conceptual framework: *project-join trees.*

Project-join trees are not an entirely new idea. Similar concepts have been used in constraint programming (as join trees [138]), probabilistic inference (as cluster trees [139]), and database-query optimization (as join-expression trees [33]). Our original contributions include the unification of these concepts into project-join trees and the application of this unifying framework to model counting.

We argue that project-join trees provide a natural formalism to describe *execution plans* for dynamic-programming algorithms for model counting. In particular, considering project-join trees as execution plans enables us to decompose dynamic-programming algorithms for model counting such as the one in `ADDMC` [29,30] into a 2-phase framework: a *planning* phase, where a project-join tree is determined, and an *execution* phase, where the determined project-join tree is used to compute the model count. This framework mirrors the 3-phase framework for model counting through tensor-network contraction in Chapter 3, although since we now attack model counting directly there is no need for a reduction phase.

This breakdown enables us to study and compare different planning algorithms, different execution libraries, and the interplay between planning and execution. Such a study is the main focus of this chapter. First, we replace the one-shot[*] planner based on constraint-satisfaction heuristics [114] used in `ADDMC` [29,30] with an anytime[†] planner based on tree-decomposition tools [33,45–47]. Second, we replace the (sparse) ADD-based executor [67] used in `ADDMC` [29, 30] with an executor based on (dense) tensors [57].

The primary contribution of this work is a dynamic-programming framework for weighted model counting based on project-join trees. In particular:

1. We show that several recent algorithms for weighted model counting [29,31] can be unified into a single framework using project-join trees.
2. We compare the planner based on constraint-satisfaction heuristics [114] used in `ADDMC` [29, 30] with a planner based on tree-decomposition tools [39, 45–47] and find that tree-decomposition tools outperform constraint-satisfaction heuristics.
3. We compare the ADD-based executor [67] used in `ADDMC` [29,30] with an executor based on tensors [57] and find that ADDs outperform tensors on single CPU cores.
4. We find that project-join-tree-based algorithms contribute to a portfolio of model counters containing `Cachet` [140], `c2d` [141], `d4` [22], and `miniC2D` [21].

This work was done in collaboration as a joint paper [81].

## 4.1 Using Project-Join Trees for Weighted Model Counting

It is well-known that weighted model counting can be performed through a sequence of projections and joins on pseudo-Boolean functions [29]. Given a CNF formula $\varphi$ and a literal-weight function $W$ over a set $X$ of variables, observe that $[\varphi] = \prod_{c \in \varphi}[c]$

[*]A *one-shot* algorithm outputs exactly one solution and then terminates immediately.

[†]An *anytime* algorithm outputs better and better solutions the longer it runs.

(since $\varphi$ is in CNF) and $W = \prod_{x \in X} W_x$ (where $\{W_x : x \in X\}$ is the set of pseudo-Boolean function guaranteed by Definition 2.5). The $W$-weighted model count of $\varphi$ can therefore be computed through $\Sigma$-projection as follows:

$$W(\varphi) = \left( \sum_X \left( \prod_{[c] \in \varphi} c \cdot \prod_{x \in X} W_x \right) \right) (\varnothing) \tag{4.1}$$

It was shown in [29] that the computational cost of Equation 4.1 can be significantly reduced using *early projection* [33]. The key idea is that, when performing a product followed by a projection, it is sometimes possible to perform the projection first. Formally:

**Theorem 4.1** (Early Projection)**.** *Let $X$ and $Y$ be sets of variables. For all functions $f : 2^X \to \mathbb{R}$ and $g : 2^Y \to \mathbb{R}$, if $x \in X \setminus Y$, then $\sum_x (f \cdot g) = (\sum_x f) \cdot g$.*

*Proof.* Expand Definition 2.2 and Definition 2.3. See **Theorem 2** of [29] for details. □

Early projection is a key technique in symbolic computation in a variety of settings, including database-query optimization [97], symbolic model checking [142], satisfiability solving [36], and model counting [29].

By taking advantage of the associative and commutative properties of multiplication as well as the commutative property of $\Sigma$-projection, [29] rearranged Equation (4.1) to apply early projection. There are a variety of possible rearrangements of Equation (4.1) of varying computational cost. Although [29] considered several heuristics for performing this rearrangement (using bucket elimination [75] and Bouquet's Method [143]), they did not attempt to analyze rearrangements.

In this chapter, we aim to analyze the quality of the rearrangement, in isolation from the underlying implementation and data structure used for Equation (4.1). This approach has been highly successful for database-query optimization [33], where the central object of theoretical reasoning is the *query plan*. The approach has also seen similar success in Bayesian network inference [144].

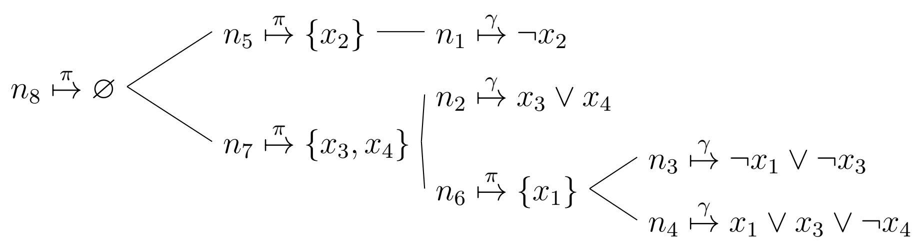

$n_8 \overset{\pi}{\mapsto} \varnothing$
$n_5 \overset{\pi}{\mapsto} \{x_2\}$
$n_1 \overset{\gamma}{\mapsto} \neg x_2$
$n_7 \overset{\pi}{\mapsto} \{x_3, x_4\}$
$n_2 \overset{\gamma}{\mapsto} x_3 \vee x_4$
$n_6 \overset{\pi}{\mapsto} \{x_1\}$
$n_3 \overset{\gamma}{\mapsto} \neg x_1 \vee \neg x_3$
$n_4 \overset{\gamma}{\mapsto} x_1 \vee x_3 \vee \neg x_4$

Figure 4.1 : A project-join tree $(T, n_8, \gamma, \pi)$ of a CNF formula $\varphi = \neg x_2 \wedge (x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_1 \vee x_3 \vee \neg x_4)$. Each leaf node is labeled by $\gamma$ with a clause of $\varphi$. Each internal node is labeled by $\pi$ with a set of variables of $\varphi$.

We model a rearrangement of Equation (4.1) as a *project-join tree*:

**Definition 4.2** (Project-Join Tree)**.** *Let $\varphi$ be a CNF formula. A* project-join tree *of $\varphi$ is a tuple $(T, r, \gamma, \pi)$ where:*

- *$T$ is a tree with root $r \in \mathcal{V}(T)$,*
- *$\gamma : \mathcal{L}(T) \to \varphi$ is a bijection between the leaves of $T$ and the clauses of $\varphi$, and*
- *$\pi : \mathcal{V}(T) \setminus \mathcal{L}(T) \to 2^{\texttt{Vars}(\varphi)}$ is a labeling function on internal nodes.*

*Moreover, $(T, r, \gamma, \pi)$ must satisfy the following two properties:*

1. *$\{\pi(n) : n \in \mathcal{V}(T) \setminus \mathcal{L}(T)\}$ is a partition of $\texttt{Vars}(\varphi)$, and*
2. *for each internal node $n \in \mathcal{V}(T) \setminus \mathcal{L}(T)$, variable $x \in \pi(n)$, and clause $c \in \varphi$ s.t. $x$ appears in $c$, the leaf node $\gamma^{-1}(c)$ must be a descendant of $n$ in $T$.*

See Figure 4.1 for a graphical example of a project-join tree. Project-join trees have previously been studied in the context of database-query optimization [33]. Project-join trees are also closely related to contraction trees in the context of tensor networks [123].

Project-join trees model a rearrangement of Equation (4.1). In detail, if $n$ is a leaf node, then $n$ corresponds to a clause $c = \gamma(n)$ in Equation (4.1). If $n$ is an internal node, then $n$'s children $\mathcal{C}_{T,r}(n)$ are to be multiplied before the projections of variables in $\pi(n)$ are performed. The two properties in Defintition 4.2 ensure that the resulting expression is equivalent to Equation (4.1) using early projection.

Given a project-join tree, we can model the computation process according to the rearrangement specified by the tree. In particular, given a literal-weight function $W = \prod_{x \in X} W_x$, we define the *$W$-valuation* of each node $n \in \mathcal{V}(T)$ as a pseudo-Boolean function associated with $n$. The $W$-valuation of a node $n \in \mathcal{V}(T)$ is denoted $f_n^W$ and defined as follows:

$$f_n^W \equiv \begin{cases} [\gamma(n)] & \text{if } n \in \mathcal{L}(T) \\ \sum_{\pi(n)} \left( \prod_{o \in \mathcal{C}_{T,r}(n)} f_o^W \cdot \prod_{x \in \pi(n)} W_x \right) & \text{if } n \notin \mathcal{L}(T) \end{cases} \tag{4.2}$$

Recall that $[\gamma(n)]$ is a pseudo-Boolean function $[\gamma(n)] : 2^{\texttt{Vars}(\gamma(n))} \to \mathbb{R}$ where $[\gamma(n)](\tau) = 1$ if and only if the assignment $\tau$ satisfies $\gamma(n)$ (and 0 otherwise).

The main idea is that the $W$-valuation at each node of $T$ is a pseudo-Boolean function computed as a subexpression of Equation (4.1). The $W$-valuation of the root is exactly the result of Equation (4.1), i.e., the $W$-weighted model count of $\varphi$:

**Theorem 4.3.** *Let $\varphi$ be a CNF formula, $(T, r, \gamma, \pi)$ be a project-join tree of $\varphi$, and let $W$ be a literal-weight function over $\texttt{Vars}(\varphi)$. Then $f_r^W(\varnothing) = W(\varphi)$.*

In order to prove this theorem, it is helpful to first state and prove a lemma that limits the appearance of variables in different subexpressions of Equation 4.2. We also refer to this lemma in Chapter 6 in the case of projected model counting.

**Lemma 4.4.** *Let $\varphi$ be a CNF formula, $(T, r, \gamma, \pi)$ be a project-join tree of $\varphi$, and let $W$ be a literal-weight function over $\texttt{Vars}(\varphi)$. For each $n \in \mathcal{V}(T)$, denote by $S(n)$ the subtree of $T$ rooted at $n$ and further define:*

$$\Phi(n) \equiv \prod_{n \in S(n) \cap \mathcal{L}(T)} [\gamma(n)]$$

$$P(n) \equiv \bigcup_{n \in S(n) \setminus \mathcal{L}(T)} \pi(n).$$

*For all $n \in \mathcal{V}(T) \setminus \mathcal{L}(T)$ and $o, o' \in \mathcal{C}_{T,r}(n)$, if $o \neq o'$ then $P(o) \cap \texttt{Vars}(\Phi(o')) = \varnothing$.*

*Proof.* For each $x \in P(o)$, $x$ is projected out at some node below $o$ in $T$. It follows by Property 2 of Definition 4.2 that $x$ appears in no descendent of $o'$, since $o'$ is not a descendent of $o$. Thus $P(o)$ and $\texttt{Vars}(\Phi(o'))$ are disjoint. □

We now use this lemma to prove Theorem 4.3.

*Proof of Theorem 4.3.* Consider each $n \in \mathcal{V}(T)$ and define $S(n)$, $\Phi(n)$, and $P(n)$ as in Lemma 4.4. The key idea is to prove for all $n \in \mathcal{V}(T)$ that

$$f_n^W = \sum_{P(n)} \left( \Phi(n) \cdot \prod_{x \in P(n)} W_x \right).$$

We proceed by induction on $n$ over the tree structure of $T$. In the base case, $n$ is a leaf and so $\Phi(n) = [\gamma(n)]$ and $P(n) = \varnothing$. So $f_n^W = \sum_{\varnothing} \left( [\gamma(n)] \cdot \prod_{x \in \varnothing} W_x \right) = [\gamma(n)]$ as desired. In the inductive case, consider an internal node $n$ of $T$. By Equation 4.2,

$$f_n^W = \sum_{\pi(n)} \left( \prod_{o \in \mathcal{C}_{T,r}(n)} f_o^W \cdot \prod_{x \in \pi(n)} W_x \right).$$

Plugging in the inductive hypothesis for each $f_o^W$, it follows that

$$f_n^W = \sum_{\pi(n)} \left( \prod_{o \in \mathcal{C}_{T,r}(n)} \left( \sum_{P(o)} \left( \Phi(o) \cdot \prod_{x \in P(o)} W_x \right) \right) \cdot \prod_{x \in \pi(n)} W_x \right).$$

If $o, o' \in \mathcal{C}_{T,r}(n)$ are distinct, by Lemma 4.4 we know $P(o) \cap \texttt{Vars}(\Phi(o')) = \varnothing$. We can therefore apply Theorem 4.1 to get that

$$f_n^W = \sum_{\pi(n)} \sum_{A} \left( \prod_{o \in \mathcal{C}_{T,r}(n)} \Phi(o) \cdot \prod_{x \in A} W_x \cdot \prod_{x \in \pi(n)} W_x \right)$$

where $A = \bigcup_{o \in \mathcal{C}_{T,r}(n)} P(o)$. Moreover, observe that $P(n) = \pi(n) \cup A$ and $\Phi(n) = \prod_{o \in \mathcal{C}_{T,r}(n)} \Phi(o)$. This completes the induction.

Finally, to complete the proof we observe that $P(r) = X$ and $\Phi(r) = [\varphi]$. It follows that $f_r^W(\varnothing) = \sum_X \left( [\varphi] \cdot \prod_{x \in X} W_x \right) = W(\varphi)$, as desired. □

This gives us a two-phase algorithm for computing the $W$-weighted model count of a formula $\varphi$. First, in the *planning* phase, we construct a project-join tree $(T, r, \gamma, \pi)$ of $\varphi$. We discuss algorithms for constructing project-join trees in Section 4.2. Second, in the *execution* phase, we compute $f_r^W$ by following Equation (4.2). We discuss data structures for computing Equation (4.2) in Section 4.3.

When computing a $W$-valuation, the number of variables that appear in each intermediate pseudo-Boolean function has a significant impact on the runtime. The set of variables that appear in the $W$-valuation of a node is actually independent of $W$. In particular, for each node $n \in \mathcal{V}(T)$, define $\texttt{Vars}(n)$ as follows:

$$\texttt{Vars}(n) \equiv \begin{cases} \texttt{Vars}(\gamma(n)) & \text{if } n \in \mathcal{L}(T) \\ \left(\bigcup_{o \in \mathcal{C}_{T,r}(n)} \texttt{Vars}(o)\right) \setminus \pi(n) & \text{if } n \notin \mathcal{L}(T) \end{cases} \tag{4.3}$$

For every literal-weight function $W$, the domain of the function $f_n^W$ is exactly $2^{\texttt{Vars}(n)}$.

To characterize the difficulty of $W$-valuation, we define the *size* of a node $n$, $\texttt{size}(n)$, to be $|\texttt{Vars}(n)|$ for leaf nodes and $|\texttt{Vars}(n) \cup \pi(n)|$ for internal nodes. The *width* of a project-join tree $(T, r, \gamma, \pi)$ is $\texttt{width}(T) \equiv \max_{n \in \mathcal{V}(T)} \texttt{size}(n)$. The width of a project-join tree is analogous to the contraction complexity of a contraction tree in the context of tensor networks (see Section 3.4.1). We see in Section 4.4 how the width impacts the computation of $W$-valuations.

## 4.2 Planning Phase: Building a Project-Join Tree

In the planning phase, we are given a CNF formula $\varphi$ over Boolean variables $X$. The goal is to construct a project-join tree of $\varphi$. In this section, we present two classes of planning techniques that have been applied to model counting: using constraint-satisfaction heuristics (as in [29]) and using tree decompositions (as in Chapter 3, and [25–28, 31]).

### 4.2.1 Planning with One-Shot Constraint-Satisfaction Heuristics

A variety of constraint-satisfaction heuristics for model counting were presented in a single algorithmic framework in `ADDMC` [29, 30]. These heuristics have a long history in constraint programming [114], database-query optimization [33], and propositional reasoning [36]. In this section, we adapt the algorithmic framework of `ADDMC` to produce project-join trees. This adaption was done by Vu H. N. Phan and is not a contribution of this dissertation. We outline it briefly here and refer the reader to [81] for the complete details.

This algorithm is presented as Algorithm 4.1, which constructs a project-join tree of a CNF formula using constraint-satisfaction heuristics. Each of the functions `ClusterVarOrder`, `ClauseRank`, and `ChosenCluster` represent heuristics for fine-tuning the specifics of the algorithm. Before discussing the various heuristics, we assert the correctness of Algorithm 4.1 in the following theorem.

**Theorem 4.5.** *Let $X$ be a set of variables and $\varphi$ be a CNF formula over $X$. Assume that* `ClusterVarOrder` *returns an injection $X \to \mathbb{N}$. Furthermore, assume that all* `ClauseRank` *and* `ChosenCluster` *calls satisfy the following conditions:*

1. $1 \leq \texttt{ClauseRank}(c, \rho) \leq m$,

2. $i < \texttt{ChosenCluster}(n_i) \leq m$, *and*

3. $X_s \cap \texttt{Vars}(n_i) = \varnothing$ *for all integers $s$ where* $i < s < \texttt{ChosenCluster}(n_i)$.

*Then Algorithm 4.1 returns a project-join tree of $\varphi$.*

*Proof.* Let $\mathcal{T} = (T, n_m, \gamma, \pi)$ be the object returned by Algorithm 4.1. To prove that $\mathcal{T}$ is a project-join tree we must verify that $T$ is a tree, that $\gamma$ is a bijection and that both properties from Definition 4.2 hold.

**Part 1: $(T, n_m)$ is a rooted tree containing every node created by Algorithm 4.1.** After the loop at Line 3, every leaf node created by Algorithm 4.1 in contained in some element of $\{\kappa_j : 1 \leq j \leq m\}$. One can prove by induction that, after iteration $i < m$ of the loop on Line 7, because of Condition 2 every node created by Algorithm 4.1 so far is the descendent of exactly one node of one element of $\{\kappa_j : i < j \leq m\}$. Thus before the final iteration (with $i = m$) every node created by Algorithm 4.1 so far is the descendent of exactly one node in $\kappa_m$. It follows that $(T, n_m)$ is indeed a rooted tree containing every node created by Algorithm 4.1.

**Part 2: $\gamma$ is a bijection.** By condition 1, $\{\Gamma_i : 1 \leq i \leq m\}$ is a partition of the clauses of $\varphi$. Moreover, observe that after Line 4 $\{\kappa_i : 1 \leq i \leq m\}$ is a partition of $\mathcal{L}(T)$. Finally, by construction $\gamma$ is a bijection between $\kappa_i$ and $\Gamma_i$ for each $1 \leq i \leq m$. Thus $\gamma$ is a bijection between $\mathcal{L}(T)$ and the clauses of $\varphi$.

**Part 3: Property 1 of Definition 4.2.** That is, we must verify that $P = \{\pi(a) : a \in \mathcal{V}(T) \setminus \mathcal{L}(T)\}$ is a partition of $X$. For each $i = 1, 2, \ldots, m$, Algorithm 4.1 constructs an internal node $n_i$ with $\pi(n_i) = X_i$. Since $\{X_i\}_{i=1}^m$ is a partition of $X$, so is $P = \{\pi(n_i)\}_{i=1}^m$.

**Part 4: Property 2 of Definition 4.2.** That is, consider an an internal node $n_i \in \mathcal{V}(T) \setminus \mathcal{L}(T)$ (created on Line 9 at iteration $i$), variable $x \in \pi(n_i)$, and clause $c \in \varphi$ s.t. $x$ appears in $c$. We must verify that $\ell = \gamma^{-1}(c)$ is a descendant of $n_i$ in $T$.

Define a sequence $p_0, p_1, \cdots, p_L$ of integers by $p_0 = p$, $p_{k+1} = \texttt{ChosenCluster}(n_{p_k})$ if $p_k < m$, and $p_L = m$. Condition 2 implies that this sequence is indeed finite and strictly increasing. The loop on Line 7 ensures that $\ell$ is a child of $n_{p_0}$ and that, for each $0 \leq k < L$, the internal node $n_{p_k}$ is a child of $n_{p_{k+1}}$. Eventually we reach $n_{p_L} = n_m$, the root of $\mathcal{T}$. So $n_{p_0}, n_{p_1}, \cdots, n_{p_L}$ are exactly the ancestors of $\ell$ in $T$.

Assume for the sake of a contradiction that $i$ is not an ancestor of $\ell$. Thus $i$ does not appear in the sequence $p_0, p_1, \cdots, p_L$. Since $x \in \pi(n_i) = X_i$, we know by Line 6 of Algorithm 4.1 that $p \leq i$. Therefore $p_0, p_1, \cdots, p_L$ is an increasing sequence of integers from $p_0 = p \leq i$ to $p_L = m \geq i$. Hence there exists some $k'$ s.t. $p_{k'} < i < p_{k'+1}$.

**Algorithm 4.1:** Using combined constraint-satisfaction heuristics to build a project-join tree

```
   Input: $X$: set of $m \geq 1$ Boolean variables
   Input: $\varphi$: CNF formula over $X$
   Output: $(T, r, \gamma, \pi)$: project-join tree of $\varphi$
1  $(T, \texttt{null}, \gamma, \pi) \leftarrow$ empty project-join tree
2  $\rho \leftarrow \texttt{ClusterVarOrder}(\varphi)$                    /* injection $\rho : X \to \mathbb{N}$ */
3  for $i = m, m-1, \ldots, 1$
4      $\Gamma_i \leftarrow \{c \in \varphi : \texttt{ClauseRank}(c, \rho) = i\}$     /* $1 \leq \texttt{ClauseRank}(c, \rho) \leq m$ */
5      $\kappa_i \leftarrow \{\texttt{LeafNode}(T, c) : c \in \Gamma_i\}$
        /* for each $c$, a leaf $l$ with $\gamma(l) = c$ is constructed and put in cluster $\kappa_i$ */
6      $X_i \leftarrow \texttt{Vars}(\Gamma_i) \setminus \bigcup_{j=i+1}^{m} \texttt{Vars}(\Gamma_j)$       /* $\{X_i\}_{i=1}^{m}$ is a partition of $X$ */
7  for $i = 1, 2, \ldots, m$
8      if $\kappa_i \neq \varnothing$
9          $n_i \leftarrow \texttt{InternalNode}(T, \kappa_i, X_i)$     /* $\mathcal{C}_{T,r}(n_i) = \kappa_i$ and $\pi(n_i) = X_i$ */
10         if $i < m$
11             $j \leftarrow \texttt{ChosenCluster}(n_i)$                 /* $i < j \leq m$ */
12             $\kappa_j \leftarrow \kappa_j \cup \{n_i\}$
13 return $(T, n_m, \gamma, \pi)$
```

Condition 3 then implies that $X_i \cap \texttt{Vars}(n_{p_{k'}}) = \varnothing$. In particular, since $x \in X_i$ this means that $x \notin \texttt{Vars}(n_{p_{k'}})$.

On the other hand, since $\{X_i\}_{i=1}^{m}$ is a partition of $X$ and $x \in \pi(n_i) = X_i$, we know that $x \notin X_{i_k}$ for all $0 \leq k \leq L$. Because $\ell$ is a child of $n_{p_0}$ and $x \in \texttt{Vars}(\ell)$, this implies that $x \in \texttt{Vars}(n_k)$ for all $0 \leq k \leq L$. In particular, $x \in \texttt{Vars}(n_{p_{k'}})$. This is a contradiction. □

There are a variety of heuristics to fine-tune Algorithm 4.1 that satisfy the conditions of Theorem 4.5. For the function `ClusterVarOrder`, we consider the heuristics **Random**, **MCS** (*maximum-cardinality search* [77]), **LexP**/**LexM** (*lexicographic search for perfect/minimal orders* [78]), and **MinFill** (*minimal fill-in* [114]) as well as their inverses (**InvMCS**, **InvLexP**, **InvLexM**, and **InvMinFill**). Heuristics for `ClauseRank` include **BE** (*bucket elimination* [75]) and **BM** (*Bouquet's Method* [143]). For `ChosenCluster`, the heuristics we use are **List** and **Tree** [29]. We combine `ClauseRank` and `ChosenCluster` as *clustering heuristics*: **BE** − **List**, **BE** − **Tree**, **BM** − **List**, and **BM** − **Tree**. See [81] for a full description of all heuristics.

### 4.2.2 Planning with Anytime Tree-Decomposition Tools

In join-query optimization, tree decompositions [40] of the *join graph* are used to find optimal join trees [33, 129]. The *join graph* of a project-join query consists of all attributes of a database as vertices and all tables as cliques. In this approach, tree decompositions of the join graph of a query are used to find optimal project-join trees; see Algorithm 3 of [33]. Similarly, tree decompositions of the *primal graph* of a factor graph, which consists of all variables as vertices and all factors as cliques, can be used to find variable elimination orders [130]. Section 3.4.2 discussed this technique in the context of tensor networks.

Translated to our project-join-tree framework, these techniques allows us to use tree decompositions of the *Gaifman graph* of a CNF formula to compute project-join trees. The Gaifman graph of a CNF formula $\varphi$, denoted $\texttt{Gaifman}(\varphi)$, has a vertex for each variable of $\varphi$, and two vertices are adjacent if the corresponding variables appear together in some clause of $\varphi$. The key idea of the technique is that each clause $c$ of $\varphi$ forms a clique in $\texttt{Gaifman}(\varphi)$ between the variables of $c$. Thus all variables of $c$ must appear together in the label of some node of the tree decomposition. We identify that node with $c$ in the resulting project-join tree.

We present this tree-decomposition-based technique as Algorithm 4.2. We first prove in the following theorem that Algorithm 4.2 is correct, i.e. that it indeed returns

**Algorithm 4.2:** Using a tree decomposition to build a project-join tree

```
   Input: $X$: set of Boolean variables
   Input: $\varphi$: CNF formula over $X$
   Input: $(S, \chi)$: tree decomposition of the Gaifman graph of $\varphi$
   Output: $(T, r, \gamma, \pi)$: project-join tree of $\varphi$
1  $(T, \texttt{null}, \gamma, \pi) \leftarrow$ empty project-join tree
2  $found \leftarrow \varnothing$                /* clauses of $\varphi$ that have been added to $T$ */
3  $s \leftarrow$ arbitrary node of $S$                /* fixing $s$ as root of $S$ */
4  function Process$(n, \ell)$:
       Input: $n \in \mathcal{V}(S)$: node of $S$ to process
       Input: $\ell \subseteq X$: variables that must not be projected out here
       Output: $N \subseteq \mathcal{V}(T)$
5      $clauses \leftarrow \{c \in \varphi : c \notin found$ and $\texttt{Vars}(c) \subseteq \chi(n)\}$
6      $found \leftarrow found \cup clauses$
7      $children \leftarrow \{\texttt{LeafNode}(T, c) : c \in clauses\} \cup \bigcup_{o \in \mathcal{C}_{S,s}(n)} \texttt{Process}(o, \chi(n))$
                       /* new leaf nodes $p \in \mathcal{V}(T)$ with $\gamma(p) = c$ */
8      if $children = \varnothing$ or $\chi(n) \subseteq \ell$
9          return $children$
10     else
11         return $\{\texttt{InternalNode}(T, children, \chi(n) \setminus \ell)\}$
               /* new internal node $o \in \mathcal{V}(T)$ with label $\pi(o) = \chi(n) \setminus \ell$ */
12 $r \leftarrow$ only element of $\texttt{Process}(s, \varnothing)$
13 return $(T, r, \gamma, \pi)$
```

a project-join tree.

**Theorem 4.6.** *Let $\varphi$ be a CNF formula over a set $X$ of variables and $(S, \chi)$ be a tree decomposition of* $\texttt{Gaifman}(\varphi)$*. Then Algorithm 4.2 returns a project-join tree of $\varphi$.*

*Proof.* Let $\mathcal{T} = (T, r, \gamma, \pi)$ be the object returned by Algorithm 4.2. For each node $a \in \mathcal{V}(T)$, $a$ was created in some `Process` call during the execution of Algorithm 4.2; let $O(a)$ denote the node in $\mathcal{V}(S)$ so that $\texttt{Process}(O(a), \ell)$ was the call that created $a$ (for some $\ell \subseteq X$). Moreover, let $s$ denote the value obtained on Line 3 of Algorithm 4.2.

We first observe that $T$ is indeed a tree with root $r$. To prove that $\mathcal{T}$ is a project-join tree we must additionally verify that $\gamma$ is a bijection and that both properties from Definition 4.2 hold.

**Part 1: $\gamma$ is a bijection.** Note that $\gamma$ is an injection since $found$ on Line 6 of Algorithm 4.2 ensures that we generate at most one leaf node for each clause. To show that $\gamma$ is a surjection, consider $c \in \varphi$. Then $\texttt{Vars}(c)$ forms a clique in the Gaifman graph of $\varphi$. It follows (since the treewidth of a complete graph on $k$ vertices is $k-1$) that $\texttt{Vars}(c) \subseteq \chi(n)$ for some $n \in \mathcal{V}(S)$. Thus $\gamma$ is a surjection as well.

**Part 2: Property 1 of Definition 4.2.** That is, we must verify that $P = \{\pi(a) : a \in \mathcal{V}(T) \setminus \mathcal{L}(T)\}$ is a partition of $X$. First, let $x \in X$. Then $x \in \texttt{Vars}(c)$ for some $c \in \varphi$. Let $n_0 = O(\gamma^{-1}(c)) \in \mathcal{V}(S)$ and observe that $x \in \chi(n_0)$. Consider the sequence of nodes $n_0, n_1, n_2, \cdots, n_k = s \in \mathcal{V}(S)$ on the unique shortest path from $n_0$ to $s$ in $S$. Observe that, by Property 3 of tree decompositions in Definition 2.9, there is some $0 \leq k' \leq k$ s.t. $x \in \chi(n_i)$ for all $0 \leq i \leq k'$ and $x \notin \chi(n_i)$ for all $k' < i \leq k$. If $k' = k$, then $x \in \chi(s)$ and so $x \in \pi(r)$. Otherwise, $x \in \chi(n_{k'})$ but $x \notin \chi(n_{k'+1})$. Consider the call on Line 7 to $\texttt{Process}(o, \chi(n))$ when $o = n_{k'}$ and $n = n_{k'+1}$. Within this call, $\chi(n_{k'} \not\subseteq \ell = \chi(n_{k'+1}))$ and so Line 11 will return an internal node $a \in \mathcal{V}(T)$ with $O(a) = n_{k'}$ and $x \in \pi(a)$.

On the other hand, consider distinct $a, b \in \mathcal{V}(T)$. Since $S$ is a tree, there is some node $p \in \mathcal{V}(S)$ on the path between $O(a)$ and $O(b)$ s.t. $p$ is the parent of either $O(a)$ or $O(b)$. By Line 7 and Line 11, this means that $\chi(p)$ is disjoint from either $\pi(a)$ or $\pi(b)$. In either case, $\chi(p)$ is disjoint from $\pi(a) \cap \pi(b)$. By Line 11 of Algorithm 4.2, we also know that $\pi(a) \cap \pi(b) \subseteq \chi(O(a)) \cap \chi(O(b))$. By Property 3 of tree decompositions in Definition 2.9, $\chi(O(a)) \cap \chi(O(b))$ (and thus $\pi(a) \cap \pi(b)$) must be contained in $\chi(p)$. It follows that $\pi(a) \cap \pi(b) = \varnothing$.

**Part 3: Property 2 of Definition 4.2.** That is, we must verify that, for each internal node $a \in \mathcal{V}(T) \setminus \mathcal{L}(T)$, variable $x \in \pi(a)$, and clause $c \in \varphi$ s.t. $x$ appears in $c$, the leaf node $\gamma^{-1}(c)$ is a descendant of $a$ in $T$. If $O(a) = s$, then $a$ is the root of $T$, so all leaf nodes are descendants. Otherwise, assume for the sake of contradiction that $\gamma^{-1}(c)$ is not a descendant of $a$ in $T$. Then $O(\gamma^{-1}(c))$ is not a descendant of $O(a)$ in $S$. This means that the parent $p \in \mathcal{V}(S)$ of $O(a)$ is on the path between $O(a)$ and $O(\gamma^{-1}(c))$. By Property 3 of tree decompositions in Definition 2.9, $x \in \chi(p)$. But Line 7 and Line 11 then implies that $x \notin \pi(a)$, a contradiction. Thus $\gamma^{-1}(c)$ is a descendant of $a$ in $T$.

It follows that $\mathcal{T}$ is a project-join tree of $\varphi$. □

The width of the resulting project-join tree is closely connected to the width of the original tree decomposition. We formalize this in the following theorem.

**Theorem 4.7.** *Let $\varphi$ be a CNF formula over a set $X$ of variables and $(S, \chi)$ be a tree decomposition of* $\texttt{Gaifman}(\varphi)$ *of width $w$. Then the project-join tree $\mathcal{T}$ of $\varphi$ returned by Algorithm 4.2 has width at most $w + 1$.*

*Proof.* The key idea is that for all $n \in \mathcal{V}(S)$ and $\ell \subseteq X$, every node $i \in \texttt{Process}(n, \ell)$ has $\texttt{Vars}(i) \subseteq \ell$. We prove this by induction on the tree structure of $S$.

Let $A$ denote the set *children* after Line 7 occurs. We begin by proving that $\texttt{Vars}(a) \subseteq \chi(n)$ for each $a \in A$. First, assume that $a$ is a leaf node of $\mathcal{T}$ corresponding to some $c \in \textit{clauses}$. In this case, $\texttt{Vars}(a) = \texttt{Vars}(c) \subseteq \chi(n)$ by Line 5. Otherwise $a \in \texttt{Process}(o, \chi(n))$ for some $o \in C(n)$. In this case, notice that $n$ is an internal node, so by the inductive hypothesis, $\texttt{Vars}(a) \subseteq \chi(n)$.

If $A = \varnothing$, then $\texttt{Process}(n, \ell)$ returns $\varnothing$, so the lemma is vacuously true. If $\chi(n) \subseteq \ell$, then $A$ is returned by $\texttt{Process}(n, \ell)$. So for every $i \in A$, we have $\texttt{Vars}(i) \subseteq \chi(n) \subseteq \ell$. Otherwise, $A \neq \varnothing$ and $\chi(n) \not\subseteq \ell$. In this case, $\texttt{Process}(n, \ell)$ returns a single node $i$ with $\texttt{Vars}(i) = \cup_{a \in A} \texttt{Vars}(a) \setminus (\chi(n) \setminus \ell) \subseteq \chi(n) \setminus (\chi(n) \setminus \ell) \subseteq \ell$.

This completes the induction. To complete the proof, notice that every internal node of $\mathcal{T}$ is returned by some call of $\texttt{Process}$ in Algorithm 4.2. Since every call Algorithm 4.2 makes to $\texttt{Process}$ has $\ell \in \{\varnothing\} \cup \{\chi(n) : n \in \mathcal{V}(S)\}$, it follows that, for every internal node $i$ of $\mathcal{T}$, $|\texttt{Vars}(i)| \leq w + 1$. Moreover, for every leaf node $i$ of $\mathcal{T}$ there is a corresponding clauses $c \in \varphi$ with $\texttt{Vars}(i) = \texttt{Vars}(c)$. Thus there is a clique in $\texttt{Gaifman}(\varphi)$ of $|\texttt{Vars}(i)|$ vertices, which implies that $w \geq |\texttt{Vars}(i)| - 1$. Thus the width of $\mathcal{T}$ is at most $w + 1$. $\square$

Theorem 4.7 allows us to leverage state-of-the-art tools for finding tree decompositions [45, 47, 145] to construct project-join trees, which we do in Section 4.4.2.

On the theoretical front, it is well-known that tree decompositions of the Gaifman graph are actually equivalent to project-join trees [33]. That is, one can go in the other direction as well: given a project-join tree of $\varphi$, one can construct a tree decomposition of $\texttt{Gaifman}(\varphi)$ of equivalent width. Formally:

**Theorem 4.8.** *Let $\varphi$ be a CNF formula and $(T, r, \gamma, \pi)$ be a project-join tree of $\varphi$ of width $w$. Then there is a tree decomposition of* $\texttt{Gaifman}(\varphi)$ *of width $w - 1$.*

*Proof.* Define $\chi : \mathcal{V}(T) \to 2^{\texttt{Vars}}(\varphi)$ by, for all $n \in \mathcal{V}(T)$, $\chi(n) \equiv \texttt{Vars}(n)$ if $n \in \mathcal{L}(T)$ and $\chi(n) \equiv \texttt{Vars}(n) \cup \pi(n)$ otherwise. Then $(T, \chi)$ is a tree decomposition of the Gaifman graph of $\varphi$. Moreover, the width of $(T, \chi)$ is $\max_{n \in \mathcal{V}(T)} |\chi(n)| - 1 =$

$\max_{n \in \mathcal{V}(T)} \texttt{size}(n) - 1 = w - 1$. $\square$

Theorem 4.8 is Lemma 1 of [33] and can be seen as the inverse of Theorem 4.7.

## 4.3 Execution Phase: Performing the Valuation

In the execution phase, we are given a CNF formula $\varphi$ over variables $X$, a project-join tree $(T, r, \gamma, \pi)$ of $\varphi$, and a literal-weight function $W$ over $X$. The goal is to compute the valuation $f_r^W$ using Equation (4.2). Several data structures can be used for the pseudo-Boolean functions that occur while using Equation (4.2). In this work, we consider two data structures that have been applied to weighted model counting: ADDs (as in [29]) and tensors (as in Chapter 3).

### 4.3.1 Executing with Algebraic Decision Diagrams

An *algebraic decision diagram (ADD)* is a compact representation of a pseudo-Boolean function in a sparse way as a directed acyclic graph [67]. ADDs with `CUDD` were used as the primary data structure for weighted model counting in `ADDMC` [29, 30]. In this work, we also use ADDs with `CUDD` to compute $W$-valuations. This was done by Vu H. N. Phan based on `ADDMC` and is not a contribution of this dissertation. We refer the reader to [81] for the complete details.

Note that all other heuristics discussed in Section 4.2.1 for cluster variable order can be used as heuristics for diagram variable order. **MCS** was the best diagram variable order on a set of standard weighted model counting benchmarks in [29]. So we use **MCS** as the diagram variable order for all ADDs in the $W$-valuation.

### 4.3.2 Executing with Tensors

A *tensor* is a multi-dimensional generalization of a matrix and can be used to represent pseudo-Boolean functions in a dense way. Tensors are particularly efficient at computing the contraction of two pseudo-Boolean functions: given two functions

$f : 2^X \to \mathbb{R}$ and $g : 2^Y \to \mathbb{R}$, their *contraction* $f \otimes g$ is the pseudo-Boolean function $\sum_{X \cap Y} f \cdot g$. The contraction of two tensors can be implemented as matrix multiplication and so leverage significant work in high-performance computing on matrix multiplication on CPUs [107] and GPUs [108]. To efficiently use tensors to compute $W$-valuations, we implement projection and product using tensor contraction.

First, we must compute the weighted projection of a function $f : 2^X \to \mathbb{R}$, i.e., we must compute $\sum_x f \cdot W_x$ for some $x \in X$. This is exactly equivalent to $f \otimes W_x$. Second, we must compute the product of two functions $f : 2^X \to \mathbb{R}$ and $g : 2^Y \to \mathbb{R}$. The central challenge is that tensor contraction implicitly projects all variables in $X \cap Y$, but we often need to maintain some shared variables in the result of $f \cdot g$. In Chapter 3, this problem was solved using a reduction from weighted model counting to tensor networks. After the reduction, all indices appear exactly twice, so one never needs to perform a product without also projecting all shared indices. Moreover, the additional overhead incurred by this reduction was lessened through the **FT** planner (see Section 3.4.3).

In order to incorporate tensors in our project-join-tree-based framework, we take a different strategy that uses copy tensors. Recall that the *copy tensor* for a set $X$ represents the pseudo-Boolean function $\mathrm{COPY}_X : 2^X \to \mathbb{R}$ s.t. $\mathrm{COPY}_X(\tau)$ is 1 if $\tau \in \{\varnothing, X\}$ and 0 otherwise. We can simulate product using contraction by including additional copy tensors. In detail, for each $z \in X \cap Y$ make two fresh variables $z'$ and $z''$. Replace each $z$ in $f$ with $z'$ to produce $f'$, and replace each $z$ in $g$ with $z''$ to produce $g'$. Then one can check that $f \cdot g = f' \otimes g' \otimes \bigotimes_{z \in X \cap Y} \mathrm{COPY}_{\{z, z', z''\}}$.

When a product is immediately followed by the projection of shared variables (i.e., we are computing $\sum_Z f \cdot g$ for some $Z \subseteq X \cap Y$), we can optimize this procedure. In particular, we skip creating copy tensors for the variables in $Z$ and instead eliminate them directly as we perform $f' \otimes g'$. In this case, we do not ever fully compute $f \cdot g$, so the maximum number of variables needed in each intermediate tensor may be lower than the width of the project-join tree. In the context of tensor networks and contraction trees, the maximum number of variables needed after accounting for this optimization is exactly the *max rank* of the contraction tree [66]. The max rank is often lower than the width of the corresponding project-joint tree. On the other hand, the intermediate terms in the computation of $f \cdot g$ with contractions may have more variables than either $f$, $g$, or $f \cdot g$. Thus the number of variables in each intermediate tensor may be higher than the width of the project-join tree.

## 4.4 Implementation and Evaluation

We aim to answer the following experimental research questions:

(RQ1) In the planning phase, how do constraint-satisfaction heuristics compare to tree-decomposition solvers?

(RQ2) In the execution phase, how do ADDs compare to tensors as the underlying data structure?

(RQ3) Are project-join-tree-based weighted model counters competitive with state-of-the-art tools?

To answer RQ1, we build two implementations of the planning phase: `HTB`, based on constraint satisfaction heuristics, and `LG`, based on tree-decomposition solvers. We compare these implementations on the planning phase in Section 4.4.2.

To answer RQ2, we build two implementations of the execution phase: `DMC`, which uses ADDs as the underlying data structure, and `tensor`, which uses tensors as the underlying data structure. We compare these implementations on the execution phase in Section 4.4.3.

To answer RQ3, we combine each implementation of the planning phase and each implementation of the execution phase to produce model counters that use project-join trees. We then compare these model counters with the state-of-the-art tools `Cachet` [140], `c2d` [141], `d4` [22], and `miniC2D` [21] in Section 4.4.4.

We use a set of 1976 literal-weighted model counting benchmarks from [29]. These benchmarks were gathered from two sources. First, the **Bayes** class[‡] consists of 1080 CNF benchmarks[§] that encode Bayesian inference problems [146]. All literal weights in this class are between 0 and 1. Second, the **Non-Bayes** class[¶] consists of 896 CNF benchmarks[‖] that are divided into eight families: *Bounded Model Checking (BMC)*, *Circuit*, *Configuration*, *Handmade*, *Planning*, *Quantitative Information Flow (QIF)*, *Random*, and *Scheduling* [86, 147–149]. All **Non-Bayes** benchmarks are originally unweighted. As we focus in this work on weighted model counting, we generate weights for these benchmarks. Each variable $x$ is randomly assigned literal weights: either $W_x(\{x\}) = 0.5$ and $W_x(\varnothing) = 1.5$, or $W_x(\{x\}) = 1.5$ and $W_x(\varnothing) = 0.5$. Generating weights in this particular fashion results in a reasonably low amount of floating-point underflow and overflow for all model counters. Note that we count overflows as errors for all counters, and underflows as errors for `DMC`.

We ran all experiments on single CPU cores of a Linux cluster with Xeon E5-2650v2 processors (2.60-GHz) and 30 GB of memory[**]. All code, benchmarks, and experimental data are available at https://github.com/vardigroup/DPMC.

### 4.4.1 Implementation Details of `DPMC`

**Planning.** `DPMC` contains two implementations of the planning phase: `HTB` (Heuristic Tree Builder, based on `ADDMC` [29, 30]) and `LG` (Line Graph, based on Chapter 3). `HTB` implements Algorithm 4.1 and so is representative of the constraint-satisfaction approach. `HTB` contains four clustering heuristics (**BE-List**, **BE-Tree**, **BM-List**, and **BM-Tree**) and nine cluster-variable-order heuristics (**Random**, **MCS**, **InvMCS**, **LexP**, **InvLexP**, **LexM**, **InvLexM**, **MinFill**, and **InvMinFill**). `LG` implements

[‡] https://www.cs.rochester.edu/u/kautz/Cachet/Model_Counting_Benchmarks

[§] excluding 11 benchmarks double-counted by [29]

[¶] http://www.cril.univ-artois.fr/KC/benchmarks.html

[‖] including 73 benchmarks missed by [29]

[**] Vu H. N. Phan implemented `HTB` and `DMC`, and ran experiments for the counter `c2d`.

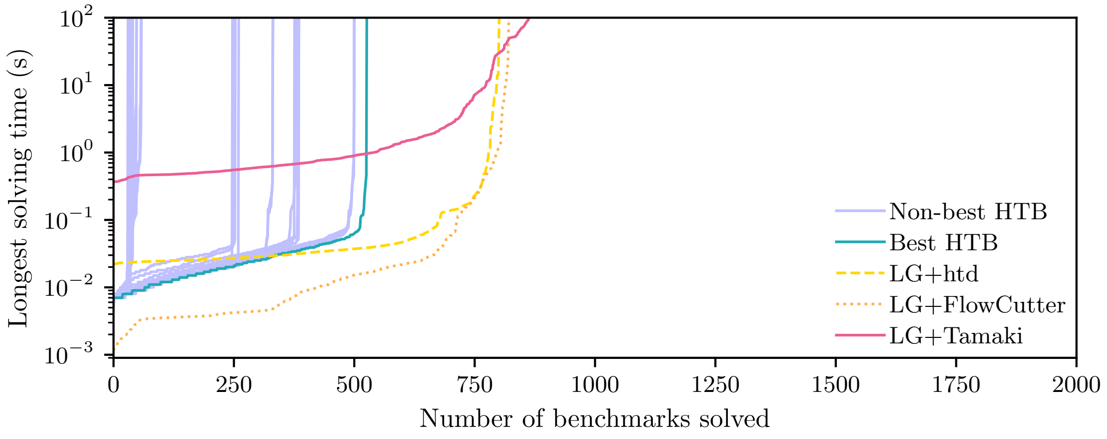


Figure 4.2 : A cactus plot of the performance of various planners. A planner "solves" a benchmark when it finds a project-join tree of width 30 or lower.

Algorithm 4.2 and so is representative of the tree-decomposition approach. In order to find tree decompositions, `LG` leverages three state-of-the-art heuristic tree-decomposition solvers: `FlowCutter` [145], `htd` [45], and `Tamaki` [47]. These solvers are all *anytime*, meaning that `LG` never halts but continues to produce better and better project-join trees when given additional time. On the other hand, `HTB` produces a single project-join tree.

**Execution.** `DPMC` contains two implementations of the execution phase: `DMC` (Diagram Model Counter, based on `ADDMC` [29, 30]) and `tensor`. `DMC` uses ADDs as the underlying data structure with `CUDD` [72]. `tensor` uses tensors as the underlying data structure with `NumPy` [59]. Since `LG` is an anytime tool, each execution tool must additionally determine the best time to terminate `LG` and begin performing the valuation. We explore options for this in Section 4.4.3.

### 4.4.2 Experiment 1: Comparing Project-Join Planners

We first compare constraint-satisfaction heuristics (`HTB`) and tree-decomposition tools (`LG`) at building project-join trees of CNF formulas. To do this, we ran all 36 configurations of HTB (combining four clustering heuristics with nine cluster-variable-order heuristics) and all three configurations of LG (choosing a tree-decomposition solver) once on each benchmark with a 100-second timeout. In Figure 4.2, we compare how long it takes various methods to find a high-quality (meaning width at most 30) project-join tree of each benchmark. We chose 30 for Figure 4.2 since we observed in Chapter 3 that tensor-based approaches were unable to handle trees whose widths are above 30, but Figure 4.2 is qualitatively similar for other choices of widths. We observe that LG is generally able to find project-join trees of lower widths than those HTB is able to find. We therefore conclude that tree-decomposition solvers outperform constraint-satisfaction heuristics in this case. We observe that **BE-Tree** as the clustering heuristic and **InvLexP** as the cluster-variable-order heuristic make up the best-performing heuristic configuration from HTB. This was previously observed to be the second-best heuristic configuration for weighted model counting in [29]. We therefore choose **BE-Tree** with **InvLexP** as the representative heuristic configuration for HTB in the remaining experiments. For LG, we choose FlowCutter as the representative tree-decomposition tool in the remaining experiments.

### 4.4.3 Experiment 2: Comparing Execution Environments

Next, we compare ADDs (DMC) and tensors (tensor) as a data structure for valuating project-join trees. To do this, we ran both DMC and tensor on all project-join trees generated by HTB and LG (with their representative configurations) in Experiment 1, each with a 100-second timeout. The total times recorded include both the planning phase and the execution phase.

Since LG is an anytime tool, it may have produced more than one project-join tree of each benchmark in Experiment 1. We follow Chapter 3 by allowing tensor and DMC to stop LG at a time proportional to the estimated cost to valuate the best-seen project-join tree. The constant of proportionality is chosen to minimize the PAR-2 score (i.e., the sum of the running times of all completed benchmarks plus twice the

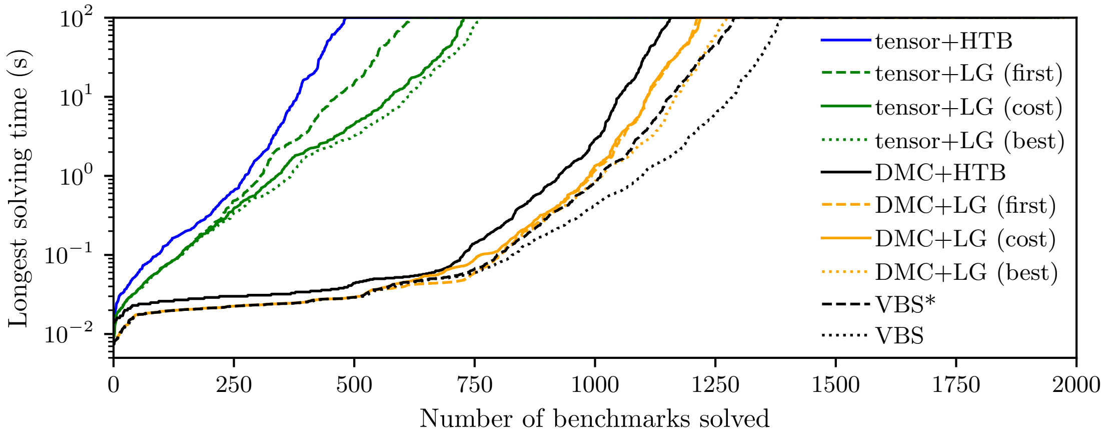


Figure 4.3 : A cactus plot of the performance of various planners and executors for weighted model counting. Different strategies for stopping `LG` are considered. "(first)" indicates that `LG` was stopped after it produced the first project-join tree. "(cost)" indicates that the executor attempted to predict the cost of computing each project-join tree. "(best)" indicates a simulated case where the executor has perfect information on all project-join trees generated by `LG` and valuates the tree with the shortest total time. `VBS*` is the virtual best solver of `DMC`+`HTB` and `DMC`+`LG` (cost). `VBS` is the virtual best solver of `DMC`+`HTB`, `DMC`+`LG` (cost), `tensor`+`HTB`, and `tensor`+`LG` (cost).

timeout for every uncompleted benchmark) of each executor. `tensor` and `DMC` use different methods for estimating cost. Tensors are a dense data structure, so the number of floating-point operations to valuate a project-join tree can be computed exactly as in Chapter 3. We use this as the cost estimator for `tensor`. ADDs are a sparse data structure, and estimating the amount of sparsity is difficult. It is thus hard to find a good cost estimator for `DMC`. As a first step, we use $2^w$ as an estimate of the cost for `DMC` to valuate a project-join tree of width $w$.

We present results from this experiment in Figure 4.3. We observe that the benefit of `LG` over `HTB` seen in Experiment 1 is maintained once the full weighted model count is computed. We also observe that `DMC` is able to solve significantly more benchmarks than `tensor`, even when using identical project-join trees. We attribute this difference to the sparsity of ADDs over tensors. Nevertheless, we observe that `tensor` still

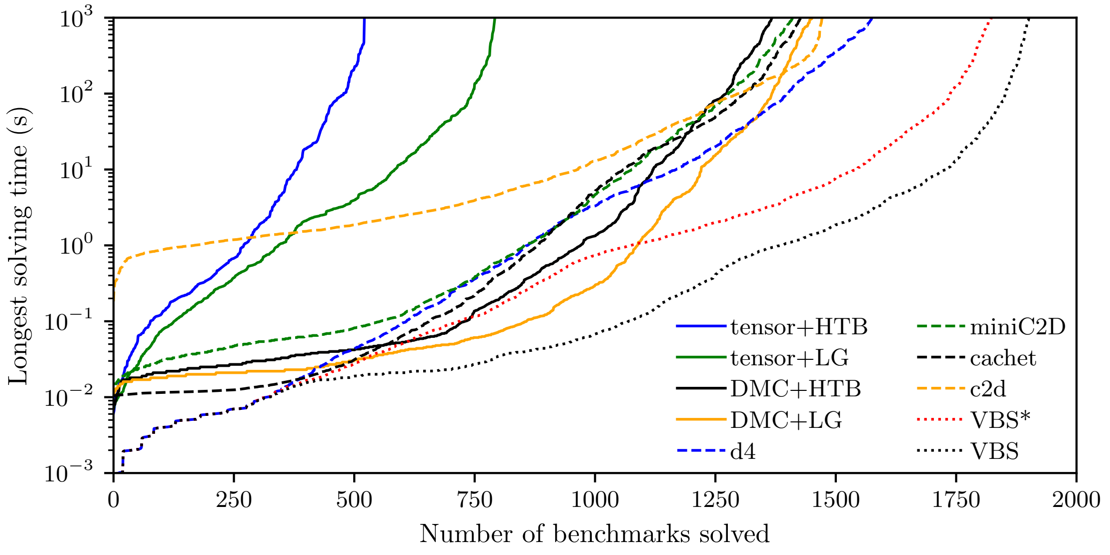


Figure 4.4 : A cactus plot of the performance of four project-join-tree-based model counters, two state-of-the-art model counters, and two virtual best solvers: `VBS*` (without project-join-tree-based counters) and `VBS` (with project-join-tree-based counters).

outperforms `DMC` on some benchmarks; compare `VBS*` (which excludes `tensor`) with `VBS` (which includes `tensor`).

Moreover, we observe significant differences based on the strategy used to stop `LG`. The executor `tensor` performs significantly better when cost estimation is used than when only the first project-join tree of `LG` is used. In fact, the performance of `tensor` is almost as good as the hypothetical performance if `tensor` is able to predict the planning and valuation times of all trees produced by `LG`. On the other hand, `DMC` is not significantly improved by the cost estimation we considered. It would be interesting in the future to find better cost estimators for `DMC`, possibly by estimating the sparsity of the ADDs that occur through the execution phase.

### 4.4.4 Experiment 3: Comparing Exact Weighted Model Counters

Finally, we compare project-join-tree-based model counters with state-of-the-art tools for weighted model counting. We construct four project-join-tree-based model counters by combining HTB and LG (using the representative configurations from Experiment 1) with DMC and tensor (using the cost estimators for LG from Experiment 2). Note that DMC+HTB is equivalent to ADDMC [29]. We compare against the state-of-the-art model counters Cachet [140], c2d [141], d4 [22], and miniC2D [21]. We ran each benchmark once with each model counter with a 1000-second timeout and recorded the total time taken. For the project-join-tree-based model counters, time taken includes both the planning phase and the execution phase.

We present results from this experiment in Figure 4.4. For each benchmark, the solving time of VBS* is the shortest solving time among all pre-existing model counters (Cachet, c2d, d4, and miniC2D). Similarly, the time of VBS is the shortest time among all model counters, including those based on project-join trees. We observe that VBS performs significantly better than VBS*. In fact, DMC+LG is the fastest model counter on 584 of 1976 benchmarks. Thus project-join-tree-based tools are valuable for portfolios of weighted model counters.

## 4.5 Chapter Summary

In this chapter, we introduced the concept of project-join trees for weighted model counting. These trees are at the center of a dynamic-programming framework that unifies and generalizes several model counting algorithms, including those based on ADDs [29] and database management systems [31]. This framework performs model counting in two phases. First, the planning phase produces a project-join tree from a CNF formula. We implemented a planner HTB based on constraint-satisfaction heuristics [75, 77, 78, 114, 143] and a planner LG based on tree-decomposition tools [45, 47, 145]. Second, the execution phase uses the project-join tree to guide the dynamic-programming computation of the model count of the formula w.r.t. a literal-weight function. We implemented an executor `DMC` based on ADDs [29, 72] and an executor `tensor` based on tensors [59]. We evaluated the resulting tool `DPMC` empirically in the context of model counting.

The primary findings of this chapter are the comparisons between the planning and execution approaches enabled by our framework. We found in the planning phase that tree-decomposition tools tend to produce project-join trees of lower widths in shorter times than constraint-satisfaction heuristics. We found in the execution phase that sparse ADDs outperform dense tensors on single CPU cores. Overall, although no single model counter dominates, `DPMC` considerably improves a portfolio of existing state-of-the-art counters (`Cachet` [140], `c2d` [141], `d4` [22], and `miniC2D` [21]) and so is a valuable addition to the portfolio.

# Chapter 5

# Parallelizing TensorOrder

Most tools for discrete integration focus on single-core performance. There have been only a few parallel counters, notably the multi-core unweighted model counter `countAntom` [150] and the GPU-based weighted model counter `gpuSAT2` [27, 28]. Developing tools for discrete integration that can leverage available parallelism is increasingly important.

Neural networks are a recent success story of parallelization. The parallelization of neural network training and inference has seen massive research across the machine learning and high-performance computing communities [60, 61, 64]. Consequently, GPUs give orders of magnitude of speedup over a single core for neural-network algorithms [62, 63]. Another line of recent work has focused on specialized hardware (e.g., using TPUs–Tensor Processing Units [64]) to provide further speedup. In this work, we aim to directly leverage advances in parallelization designed for neural-network algorithms in the service of weighted model counting.

Chapter 3 demonstrated that *tensor networks* are a natural bridge between high-performance computing and weighted model counting. In this chapter, we further exploit this bridge to bring parallelization techniques from high-performance computing to weighted model counting. Specifically, we explore the impact of multiple-core, GPU, and TPU use on tensor network contraction for weighted model counting.

In detail, Chapter 3 presented a 3-phase algorithm for weighted model counting. First, in the *reduction* phase, a counting instance is reduced to a tensor-network problem. Second, in the *planning* phase, an order to contract tensors in the network is determined. Finally, in the *execution* phase, tensors in the network are contracted using the determined order. The algorithmic and experimental results of Chapter 3 focused on a single-core performance. Since the reduction phase was not a significant source of runtime, we focus on opportunities for parallelism in the planning and execution phases.

The planning phase in Chapter 3 was done using a choice of several single-core heuristic tree-decomposition solvers [45–47]. There is little recent work on parallelizing heuristic tree-decomposition solvers. Instead, we implement a parallel portfolio of single-core tree-decomposition solvers and find that this portfolio significantly improves planning on multiple cores. Similar portfolio approaches have been well-studied and shown to be beneficial in the context of SAT solvers [79, 151]. As a theoretical contribution, we prove that branch-decomposition solvers can also be included in this portfolio. Unfortunately, we find that a current branch-decomposition solver only marginally improves the portfolio.

The execution phase in Chapter 3 was done using `numpy` [59] and evaluated on a single core. We add an implementation of the execution phase that uses `TensorFlow` [60] to leverage a GPU for large contractions. Since GPU memory is significantly more limited than CPU memory, we add an implementation of *index slicing*. Index slicing is a recent technique from the tensor-network community [152–154], analogous to the classic technique of conditioning in Bayesian Network reasoning [75, 124, 139, 155], that allows memory to be significantly reduced at the cost of additional time. We find that, while multiple cores do not significantly improve the contraction phase, a GPU provides significant speedup, especially when augmented with index slicing.

We also add an implementation of the execution phase that uses index slicing with `jax` [156] in order to efficiently leverage a TPU. Unfortunately, we observe that current tensor libraries fail to compile tensor computations with high-dimensional tensors to a TPU. Thus our approach does not scale well on a TPU for practical model counting benchmarks.

We implement our techniques in `TensorOrder2`, a new parallel weighted model counter. We compare `TensorOrder2` to a variety of state-of-the-art counters. We show that the improved `TensorOrder2` is the fastest counter on 11% of benchmarks after preprocessing [157], outperforming the GPU-based counter `gpuSAT2` [28]. Thus `TensorOrder2` is useful as part of a portfolio of counters. All code and data are available at https://github.com/vardigroup/TensorOrder.

The rest of the chapter is organized as follows. In Section 5.1, we describe parallelization of the planning phase with an algorithmic portfolio and planning with branch decompositions. In Section 5.2, we describe parallelization of the execution phase with index slicing on a GPU and on a TPU. In Section 5.3, we implement this parallelization in `TensorOrder2` and analyze its performance experimentally.

## 5.1 Planning Phase: Parallelizing with a Portfolio

As discussed in Section 3.4.3, `FactorTree`$(N)$ first computes a tree decomposition of struct$(N)$ and then applies Theorem 3.11. In [80], most time in the planning phase was spent finding low-width tree decompositions with a choice of single-core heuristic tree-decomposition solvers [45–47]. Finding low-width tree decompositions is thus a valuable target for parallelization.

We were unable to find state-of-the-art parallel heuristic tools for tree decompositions. There are several classic algorithms for parallel tree-decomposition construction [158, 159], but no implementations that are competitive with state-of-the-art single-core tools. There is also an exact tree-decomposition solver that leverages a GPU [160], but exact tools are not able to scale to handle our benchmarks. Existing single-core heuristic tree-decomposition solvers are highly optimized and so parallelizing them is nontrivial.

Instead, we take inspiration from the SAT solving community. One competitive approach to building a parallel SAT solver is to run a variety of SAT solvers in parallel across multiple cores [79, 151, 161]. The portfolio SAT solver `CSHCpar` [161] won the open parallel track in the 2013 SAT Competition with this technique, and such portfolio solvers are now banned from most tracks of the SAT Competition due to their effectiveness relative to their ease of implementation. Similar parallel portfolios are thus promising for integration into the planning phase.

We apply this technique to heuristic tree-decomposition solvers by running a variety of single-core heuristic tree-decomposition solvers in parallel on multiple cores and collating their outputs into a single stream. We analyze the performance of this technique in Section 5.3.

While Theorem 3.11 lets us integrate tree-decomposition solvers into the portfolio, the portfolio might also be improved by integrating solvers of other graph decompositions. The following theorem shows that branch-decomposition solvers may also be used for `FactorTree`$(N)$:

**Theorem 5.1.** *Let $N$ be a tensor network of tree-factorable tensors s.t. $|\mathcal{F}(N)| \leq 3$ and $G = struct(N)$ has a branch decomposition $T$ of width $w \geq 1$. Then we can construct in polynomial time a tensor network $M$ and a contraction tree $S$ for $M$ s.t. $S$ has max rank no larger than $\lceil 4w/3 \rceil$ and $N$ is a partial contraction of $M$.*

*Proof.* This proof differs from the proof of Theorem 3.11 in Chapter 3 only in Part 1 and Part 4 (and in the definition of $\rho$ in Part 3).

The proof proceeds in five steps: (1) compute the factored tensor network $M$, (2) construct a graph $H$ that is a simplified version of the structure graph of $M$, (3) construct a carving decomposition $S$ of $H$, (4) bound the width of $S$, and (5) use $S$ to find a contraction tree for $M$. Working with $H$ instead of directly working with the structure graph of $M$ allows us to cleanly handle tensor networks with free indices.

**Part 1: Factoring the network.** Next, for each $v \in \mathcal{V}(G)$, define $T_v$ to be the smallest connected component of $T$ containing $\delta_G(v) \subseteq \mathcal{L}(T)$. Consider each $A \in N$. If $\mathcal{F}(A) = \varnothing$, let $N_A = \{N_A\}$. Otherwise, observe that $T_A$ is a dimension tree of $A$ and so we can factor $A$ with $T_A$ using Definition 3.10 to get a tensor network $N_A$ and a bijection $g_A : \mathcal{V}(T_A) \to N_A$. Define $M = \cup_{A \in N} N_A$ and let $G'$ be the structure graph of $M$ with free vertex $z'$. The remainder of the proof is devoted to bounding the carving width of $G'$. To do this, it is helpful to define $\rho : \mathcal{V}(T) \to \mathcal{V}(G)$ by $\rho(n) = \{v \in \mathcal{V}(G) : n \in \mathcal{V}(T_v), |\delta_{T_v}(n)| = 3\}$. Note that $|\rho(n)| \leq w$ for all $n \in \mathcal{V}(T)$.

**Part 2: Constructing a simplified structure graph of $M$.** In order to easily characterize $G'$, we define a new, closely-related graph $H$ by taking a copy of $T_v$ for each $v \in \mathcal{V}(G)$ and connecting these copies where indicated by $g$. Formally, the vertices of $H$ are $\{(v, n) : v \in \mathcal{V}(G), n \in \mathcal{V}(T_v)\}$. For every $v \in \mathcal{V}(G)$ and every arc in $T$ with endpoints $n, m \in \mathcal{V}(T_v)$, we add an edge between $(v, n)$ and $(v, m)$. Moreover, for each $e \in \mathcal{E}(G)$ incident to $v, w \in \mathcal{V}(G)$, we add an edge between $(v, g(e))$ and $(w, g(e))$.

We will prove in Part 5 that the carving width of $G'$ is bounded from above by the carving width of $H$. We therefore focus in Part 3 and Part 4 on bounding the carving width of $H$. It is helpful for this to define the two projections $\pi_G : \mathcal{V}(H) \to \mathcal{V}(G)$ and $\pi_T : \mathcal{V}(H) \to \mathcal{V}(T)$ that indicate respectively the first or second component of a vertex of $H$.

**Part 3. Constructing a carving decomposition $S$ of $H$.** The idea of the construction is, for each $n \in \mathcal{V}(T)$, to attach the elements of $\pi_T^{-1}(n)$ as leaves along the arcs incident to $n$. To do this, for every leaf node $\ell \in \mathcal{L}(T)$ with incident arc $a \in \delta_T(\ell)$ define $H_{\ell,a} = \pi_T^{-1}(\ell)$. For every non-leaf node $n \in \mathcal{V}(T) \setminus \mathcal{L}(T)$ partition $\pi_T^{-1}(n)$ into three sets $\{H_{n,a} : a \in \delta_T(n)\}$, ensuring that the degree 3 vertices are divided evenly (the degree 1 and 2 vertices can be placed arbitrarily). Observe that $\{H_{n,a} : n \in \mathcal{V}(T), a \in \delta_T(n)\}$ is a partition of $\mathcal{V}(H)$, and there are at most $\lceil |\rho(n)|/3 \rceil$ vertices of degree 3 in each $H_{n,a}$, since there are exactly $|\rho(n)|$ vertices of degree 3 in $\pi_T^{-1}(n)$.

We use this to construct a carving decomposition $S$ from $T$ by adding each element of $H_{n,a}$ as a leaf along the arc $a$. Formally, let $x_v$ denote a fresh vertex for each $v \in \mathcal{V}(H)$, let $y_n$ denote a fresh vertex for each $n \in \mathcal{V}(T)$, and let $z_{n,a}$ denote a fresh vertex for each $n \in \mathcal{V}(T)$ and $a \in \delta_T(n)$. Define $\mathcal{V}(S)$ to be the union of $\mathcal{V}(H)$ with the set of these free vertices.

We add an arc between $v$ and $x_v$ for every $v \in \mathcal{V}(H)$. Moreover, for every $a \in \mathcal{E}(T)$ with endpoints $o, p \in \epsilon_T(a)$ add an arc between $y_{o,a}$ and $y_{p,a}$. For every $n \in \mathcal{V}(T)$ and incident arc $a \in \delta_T(n)$, construct an arbitrary sequence $I_{n,a}$ from $\{x_v : v \in H_{n,a}\}$. If $H_{n,a} = \varnothing$ then add an arc between $y_n$ and $z_{n,a}$. Otherwise, add arcs between $y_n$ and the first element of $I$, between consecutive elements of $I_{n,a}$, and between the last element of $I_{n,a}$ and $z_{n,a}$.

Finally, remove the previous leaves of $T$ from $S$. The resulting tree $S$ is a carving decomposition of $H$, since we have added all vertices of $H$ as leaves and removed the previous leaves of $T$.

**Part 4: Computing the width of $S$.** In this part, we separately bound the width of the partition induced by each of the three kinds of arcs in $S$.

First, consider an arc $b$ between some $v \in \mathcal{V}(H)$ and $x_v$. Since all vertices of $H$ are degree 3 or smaller, $b$ defines a partition of width at most $3 \leq \lceil 4w/3 \rceil$.

Next, consider an arc $c_a$ between $z_{o,a}$ and $z_{p,a}$ for some arc $a \in \mathcal{E}(T)$ with endpoints $o, p \in \epsilon_T(a)$. Observe that removing $a$ from $T$ defines a partition $\{B_o, B_p\}$ of $\mathcal{V}(T)$, denoted so that $o \in B_o$ and $p \in B_p$.

Then removing $c_a$ from $S$ defines the partition $\{\pi_T^{-1}(B_o), \pi_T^{-1}(B_p)\}$ of $\mathcal{L}(S)$. By construction of $H$, all edges between $\pi_T^{-1}(B_o)$ and $\pi_T^{-1}(B_p)$ are between $\pi_T^{-1}(o)$ and $\pi_T^{-1}(p)$. Observe that every edge $e \in \mathcal{E}(H)$ between $\pi_T^{-1}(o)$ and $\pi_T^{-1}(p)$ corresponds under $g_v$ to $a$ in $T_v$ for some $v$. It follows that the number of edges between $\pi_T^{-1}(o)$ and $\pi_T^{-1}(o)$ is exactly the number of vertices in $G$ that are endpoints of edges in both $C_a$ and $\mathcal{E}(G) \setminus C_a$, which is bounded by $w$. Thus the partition defined by $c_a$ has width no larger than $w$.

Finally, consider an arc $d$ added as one of the sequence of $|H_{n,a}|+1$ arcs between $y_n$, $I_{n,a}$, and $z_{n,a}$ for some $n \in \mathcal{V}(T)$ and $a \in \delta_T(n)$. Some elements of $H_{n,a}$ have changed blocks from the partition defined by $c_a$. Each vertex of degree 2 that changes blocks does not affect the width of the partition, but each vertex of degree 3 that changes blocks increases the width by 1. There are at most $\lceil w/3 \rceil$ elements of degree 3 added as leaves between $y_n$ and $z_{n,a}$. Thus the partition defined by $d$ has width at most $w + \lceil w/3 \rceil = \lceil 4w/3 \rceil$.

It follows that the width of $S$ is at most $\lceil 4w/3 \rceil$.

**Part 5: Bounding the max rank of $M$.** Let $z$ be the free vertex of the structure graph of $N$. We first construct a new graph $H'$ from $H$ by, if $\mathcal{F}(N) \neq \varnothing$, contracting all vertices in $\pi_G^{-1}(z)$ to a single vertex $z$. If $\mathcal{F}(N) = \varnothing$, instead add $z$ as a fresh degree 0 vertex to $H'$. Moreover, for all $A \in N$ with $\mathcal{I}(A) = \varnothing$ add $A$ as a degree 0 vertex to $H'$.

Note that adding degree 0 vertices to a graph does not affect the carving width. Moreover, since $|\mathcal{F}(N)| \leq 3$ all vertices (except at most one) of $\pi_G^{-1}(z)$ are degree 2 or smaller. It follows that contracting $\pi_G^{-1}(z)$ does not increase the carving width. Thus the carving width of $H'$ is at most $\lceil 4w/3 \rceil$.

Moreover, $H'$ and $G'$ are isomorphic. To prove this, define an isomorphism $\varphi : \mathcal{V}(H') \to \mathcal{V}(G')$ between $H'$ and $G'$ by, for all $v \in \mathcal{V}(H')$:

$$\varphi(v) \equiv \begin{cases} v & \text{if } v \in N \text{ and } \mathcal{I}(v) = \varnothing \\ z' & v = z \\ g_{\pi_G(v)}(\pi_T(v)) & \text{if } v \in \mathcal{V}(H) \text{ and } \pi_G(v) \in N \end{cases}$$

$\varphi$ is indeed an isomorphism between $H'$ and $G'$ because the functions $g_A$ are all isomorphisms and because an edge exists between $\pi_G^{-1}(v)$ and $\pi_G^{-1}(w)$ for $v, w \in \mathcal{V}(G)$ if and only if there is an edge between $v$ and $w$ in $G$. Thus the carving width of $G'$ is at most $\lceil 4w/3 \rceil$. By Theorem 3.6, then, $M$ has a contraction tree of max rank no larger than $\lceil 4w/3 \rceil$. □

Theorem 5.1 subsumes Theorem 3.11, since given a graph $G$ and a tree decomposition for $G$ of width $w+1$ one can construct in polynomial time a branch decomposition for $G$ of width $w$ [40]. Moreover, on many graphs there are branch decompositions whose width is smaller than the width of all tree decompositions. We explore this potential in practice in Section 5.3.

It was previously known that branch decompositions can also be used in variable elimination [106], but Theorem 5.1 considers branch decompositions of a different graph. For example, consider again computing the model count of $\psi = (\vee_{i=1}^{n} x_i) \wedge (\vee_{i=1}^{n} \neg x_i)$. Theorem 5.1 uses branch decompositions of the incidence graph of $\psi$, which has branchwidth 2. Variable elimination uses branch decompositions of the primal graph of $\psi$, which has branchwidth $n-1$.

## 5.2 Execution Phase: Parallelizing with Index Slicing

As discussed in Section 2.4, each contraction in Algorithm 3.2 can be implemented as a matrix multiplication. This was done in Chapter 3 using `numpy` [59], and it is straightforward to adjust the implementation to leverage multiple cores with `numpy` or leverage a GPU with `TensorFlow` [60]. In Section 5.2.1, we discuss challenges in dealing with GPU on-board memory and our solution of index slicing. In Section 5.2.2, we then discuss how index slicing allows us to efficiently run the execution phase on a TPU using `jax` [156].

### 5.2.1 Contraction on a GPU

The primary challenge that emerges when using a GPU is dealing with the small onboard memory. For example, the `NVIDIA Tesla-V100` (which we use in Section 5.3) has just 16GB of onboard memory. This limits the size of tensors that can be easily manipulated. One solution is to perform a single contraction of two tensors cross multiple GPU kernel calls [162]. This is similar to the technique implemented in `gpusat2` [28]. This solution, however, require parts of the input tensors to be copied into and out of GPU memory multiple times for each contraction, which incurs significant slowdown.

Instead, we use *index slicing* [152–154] of a tensor network $N$. This technique is analogous to *conditioning* on Bayesian networks [75, 124, 139, 155]. The idea is to represent $\mathcal{T}(N)$ as the sum of contractions of smaller tensor networks. Each smaller

**Algorithm 5.1:** Sliced contraction of a tensor network

```
Input: $N$: a tensor network
Input: $T$: a contraction tree for $N$
Input: $m$: a memory bound
Output: $\mathcal{T}(N)$, the contraction of $N$, performed using at most $m$ memory.
1 function ExecuteSliced($N, T, m$):
2     $I \leftarrow \varnothing$
3     while MemCost($N, T, I$) $> m$
4         $I \leftarrow I \cup \{$ChooseSliceIndex($N, T, I$)$\}$
5     return $\sum_{\eta \in [I]}$ Execute($N[\eta], T[\eta]$)
```

network contains tensor slices:

**Definition 5.2.** *Let $A$ be a tensor, $I \subseteq$ **Ind**, and $\eta \in [I]$. Then the $\eta$-*slice *of $A$ is the tensor $A[\eta]$ with indices $\mathcal{I}(A) \setminus I$ defined for all $\tau \in [\mathcal{I}(A) \setminus I]$ by $A[\eta](\tau) \equiv A((\tau \cup \eta)\big|_{\mathcal{I}(A)})$.*

These are called slices because every value in $A$ appears in exactly one tensor in $\{A[\eta] : \eta \in [I]\}$. We now define and prove the correctness of index slicing:

**Theorem 5.3.** *Let $N$ be a tensor network and let $I \subseteq \mathcal{B}(N)$. For each $\eta \in [I]$, let $N[\eta] = \{A[\eta] : A \in N\}$. Then $\mathcal{T}(N) = \sum_{\eta \in [I]} \mathcal{T}(N[\eta])$.*

*Proof.* Move the summation over $\eta \in [I]$ to the outside of Equation 3.1, then apply the definition of tensor slices and recombine terms. □

By choosing $I$ carefully, computing each $\mathcal{T}(N[\eta])$ uses less intermediate memory (compared to computing $\mathcal{T}(N)$) while using the same contraction tree. In exchange, the number of floating point operations to compute all $\mathcal{T}(N[\eta])$ terms increases.

Choosing $I$ is itself a difficult problem. Our goal is to choose the smallest $I$ so that contracting each network slice $N[\eta]$ can be done in onboard memory. We first consider adapting Bayesian network conditioning heuristics to the context of tensor networks. Two popular conditioning heuristics are (1) *cutset conditioning* [155], which chooses $I$ so that each network slice is a tree, and (2) *recursive conditioning* [124], which chooses $I$ so that each network slice has a disconnected structure graph*. Both of these heuristics result in a choice of $I$ far larger than our goal requires.

Instead, in this work as a first step we use a heuristic from [152, 153]: choose $I$ incrementally, greedily minimizing the memory cost of contracting $N[\eta]$ until the memory usage fits in onboard memory. Unlike cutset and recursive conditioning, the resulting networks $N[\eta]$ typically still have highly connected structure graphs. One direction for future work is to compare other heuristics for choosing $I$ (e.g., see the discussion in Section 10 of [75]).

This gives us Algorithm 5.1, which performs the execution phase with limited memory at the cost of additional time. $T[\eta]$ is the contraction tree obtained by computing the $\eta$-slice of every tensor in $T$. `MemCost`$(N, T, I)$ computes the memory for one `Execute`$(N[\eta], T[\eta])$ call. `ChooseSliceIndex`$(N, T, I)$ chooses the next slice index greedily to minimize memory cost.

### 5.2.2 Contraction on a TPU

A TPU (tensor processing unit) [64] is a specialized computing hardware designed for neural network training and inference. In particular, a TPU consists of multiple cores that each contain application-specific integrated circuits (ASICs) for matrix multiplication. Several tensor libraries (e.g. `TensorFlow` [60] and `jax` [156]) support tensor contraction on a TPU. Thus tensor networks provide a natural framework to leverage TPUs for model counting.

*The full recursive conditioning procedure then recurses on each connected component. While recursive conditioning is an any-space algorithm, the partial caching required for this is difficult to implement on a GPU.

The primary challenge that emerges is that computing on a TPU requires an additional compilation step. So far, we have focused on using tensor libraries (`numpy` [59] and `TensorFlow` [60]) in *eager execution* mode. In eager execution mode, a tensor library performs each tensor operation directly on a CPU or GPU using precompiled and preoptimized code. Thus no extra compilation is needed at runtime. Unfortunately, no current tensor libraries can leverage a TPU while in eager execution mode.

Instead, tensor libraries like `TensorFlow` [60] and `jax` [156] support using a TPU only in *graph execution* mode. In graph execution mode, a sequence of tensor operations on unspecified input is translated by the tensor library into an XLA graph program [163], which is a representation of the computation as a directed acyclic graph. This XLA graph program is then compiled and optimized (e.g. by modifying tensor layouts, or by fusing operations) for various hardware targets, including CPUs, GPUs, or TPUs.

In general, the compilation step required by graph execution may take orders of magnitude longer than a single eager execution of the same computation, but a single execution of the compiled program is faster than a single eager execution. This trade-off is beneficial in the context of neural network training and inference, where the same computation (say, training a neural network once on a single batch of examples) is repeated many times with different inputs. The challenge, then, is to identify repeated, independent computations in the contraction of a tensor network that may be profitably compiled.

Our key observation is that Algorithm 5.1 computes the contraction of a tensor network through a sequence of similar, independent computations: the contraction of the sliced networks $N[\eta]$. In particular, each call to `Execute` on Line 5 of Algorithm 5.1 has the same computational structure when following Algorithm 3.2. This occurs because, for each pair $\eta, \eta' \in [I]$, there is a natural correspondence between the tensors of $N[\eta]$ and $N[\eta']$, where two tensors $A[\eta] \in N[\eta]$ and $A[\eta'] \in N[\eta']$ correspond if they are each slices of the same tensor $A \in N$. Corresponding tensors $A[\eta]$ and $A[\eta']$ have the same set of indices and differ only in their specific entries. It follows that the computations of $\texttt{Execute}(N[\eta], T[\eta])$ and $\texttt{Execute}(N[\eta'], T[\eta'])$ proceed in lockstep in Algorithm 3.2, differing only in the entries of the relevant tensors.

This allows us to run Line 5 of Algorithm 5.1 in graph execution mode on a TPU. For a given tensor network $N$, contraction tree $T$ for $N$, and $I \subseteq \mathcal{B}(N)$, consider the function $f_{N,T,I}$ that takes as input $\eta \in [I]$ and outputs $\texttt{Execute}(N[\eta], T[\eta])$. Our approach is to represent $f_{N,T,I}$ as an XLA graph program. We compile $f_{N,T,I}$ once, targeting a TPU, and then run the compiled program on each $\eta \in [I]$. Note that since a TPU contains multiple cores we run the compiled program multiple times in parallel, once for each core. We evaluate the performance of this approach in Section 5.3.5.

## 5.3 Implementation and Evaluation

We aim to answer the following experimental research questions:

(RQ1) Is the planning phase improved by a parallel portfolio of decomposition tools?

(RQ2) Is the planning phase improved by adding a branch-decomposition tool?

(RQ3) When should Algorithm 3.1 transition from the planning phase to the execution phase (i.e., what should be the value of the performance factor $\alpha$)?

(RQ4) Is the execution phase improved by leveraging multiple cores and a GPU?

(RQ5) Is the execution phase improved by leveraging a TPU in graph execution mode?

(RQ6) Do parallel tensor-network approaches improve a portfolio of state-of-the-art weighted model counters (`cachet`, `miniC2D`, `d4`, `ADDMC`, and `gpuSAT2`)?

We implement our changes on top of `TensorOrder` (which implements Algorithm 3.1; see Section 3.5.1) to produce `TensorOrder2`, a new parallel weighted model counter. Implementation details are described in Section 5.3.1. All code is available at https://github.com/vardigroup/TensorOrder.

We use a standard set[†] of 1914 weighted model counting benchmarks [29]. Of these, 1091 benchmarks[‡] are from Bayesian inference problems [19] and 823 benchmarks[§] are unweighted benchmarks (from various domains) that were artificially weighted by [29]. For weighted model counters that cannot handle real weights larger than 1 (`cachet` and `gpusat2`), we rescale the weights of benchmarks with larger weights. In Experiment 3, we also consider preprocessing these 1914 benchmarks by applying `pmc-eq` [157] (which preserves weighted model count). We evaluate the performance of each tool using the PAR-2 score, which is the sum of the wall-clock times for each completed benchmark, plus twice the timeout for each uncompleted benchmark.

All counters are run in Docker images (one for each counter) with Docker 19.03.5. All experiments are run on Google Cloud `n1-standard-8` machines with 8 cores (Intel Haswell, 2.3 GHz) and 30 GB RAM. GPU-based counters are provided an `NVIDIA Tesla V100` GPU (16 GB of onboard RAM) using NVIDIA driver 418.67 and CUDA 10.1.243. TPU-based counters are provided a v2-8 TPU, which contains 8 TPU cores and 8 GB of onboard RAM per core.

### 5.3.1 Implementation Details of `TensorOrder2`

`TensorOrder2` is primarily implemented in Python 3 as a modified version of the tool `TensorOrder` from Chapter 3. We replace portions of the Python code with C++ (`g++` v7.4.0) using Cython 0.29.15 for general speedup, especially in the implementation of the **FT** planner.

[†]https://github.com/vardigroup/ADDMC/releases/tag/v1.0.0

[‡]https://www.cs.rochester.edu/u/kautz/Cachet/

[§]http://www.cril.univ-artois.fr/KC/benchmarks.html

**Planning.** `TensorOrder` contains an implementation of the planning phase using the **FT** planner together with a choice of three single-core tree-decomposition solvers: `Tamaki` [47], `FlowCutter` [46], and `htd` [45]. We add to `TensorOrder2` an implementation of Theorem 5.1 and use it to add a branch-decomposition solver `Hicks` [48]. We implement a parallel portfolio of graph-decomposition solvers in C++ and give `TensorOrder2` access to two portfolios, each with access to all cores: `P3` (which combines `Tamaki`, `FlowCutter`, and `htd`) and `P4` (which includes `Hicks` as well).

**Execution.** `TensorOrder` is able to perform the execution phase on a single core and on multiple cores using `numpy` v1.18.1 and `OpenBLAS` v0.2.20.

We add to `TensorOrder2` a way to contract tensors on a GPU with `TensorFlow` v2.1.0 [60]. To avoid GPU kernel calls for small contractions, `TensorOrder2` uses a GPU only for contractions where one of the tensors involved has rank $\geq 20$, and reverts back to using multi-core `numpy` otherwise. We also add an implementation of Algorithm 5.1.

We also add a way to contract tensors on a TPU with `jax` v0.2.13 [156] in graph execution mode. We execute each slice (i.e., each call to `Execute` on Line 5 of Algorithm 5.1) on a separate TPU core. Thus on a v2-8 TPU we execute 8 slices in parallel. For comparison, we also add support for contracting tensors on multiple-CPUs with `jax` in graph execution mode.

Overall, `TensorOrder2` runs the execution phase in three possible configurations in eager execution mode– `CPU1` (restricted to a single CPU core), `CPU8` (allowed to use all 8 CPU cores), `GPU` (allowed to use all 8 CPU cores and use a GPU)– and two possible configurations in graph execution mode– `TPU-graph` (allowed to use all 8 CPU cores, and a TPU), and `CPU8-graph` (allowed to use all 8 CPU cores).

Finally, note that `TensorOrder2` supports 64-bit floats for all execution configurations except for `TPU-graph`, which supports only 32-bit floats due to TPU hardware limitations. We therefore output 64-bit floats in Experiments 1, 2, and 3, and output

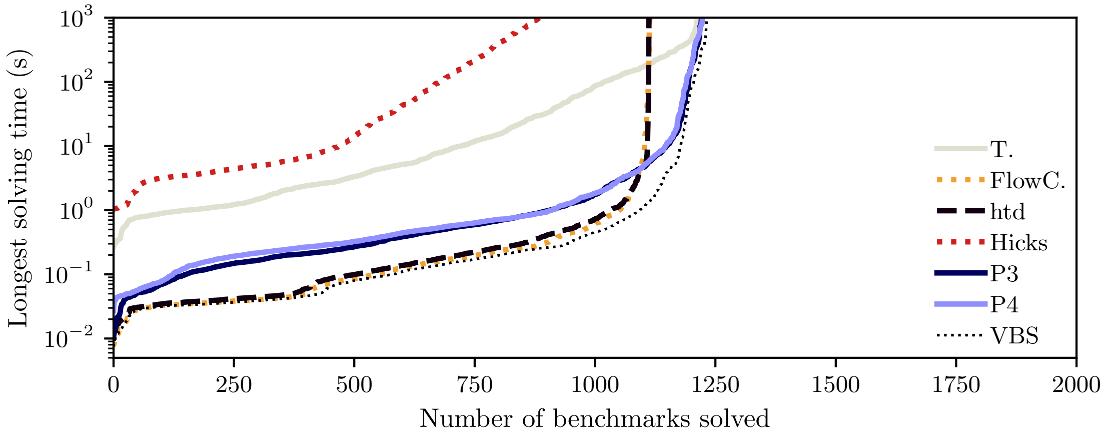


Figure 5.1 : A cactus plot of the performance of various planners. A planner "solves" a benchmark when it finds a contraction tree of max rank 30 or smaller.

32-bit floats for Experiment 4.

## 5.3.2 Experiment 1: The Planning Phase (RQ1 and RQ2)

We run each planning implementation (`FlowCutter`, `htd`, `Tamaki`, `Hicks`, `P3`, and `P4`) once on each of our 1914 benchmarks and save all contraction trees found within 1000 seconds (without executing the contractions). Results are summarized in Figure 5.1.

We observe that the parallel portfolio planners outperform all four single-core planners after 5 seconds. In fact, after 20 seconds both portfolios perform almost as well as the virtual best solver. We conclude that portfolio solvers significantly speed up the planning phase.

We also observe that `P3` and `P4` perform almost identically in Figure 5.1. Although after 1000 seconds `P4` has found better contraction trees than `P3` on 407 benchmarks, most improvements are small (reducing the max rank by 1 or 2) or still do not result in good-enough contraction trees. We conclude that adding `Hicks` improves the portfolio slightly, but not significantly. There is opportunity here to evaluate adding other graph decomposition tools to the portfolio in future work.

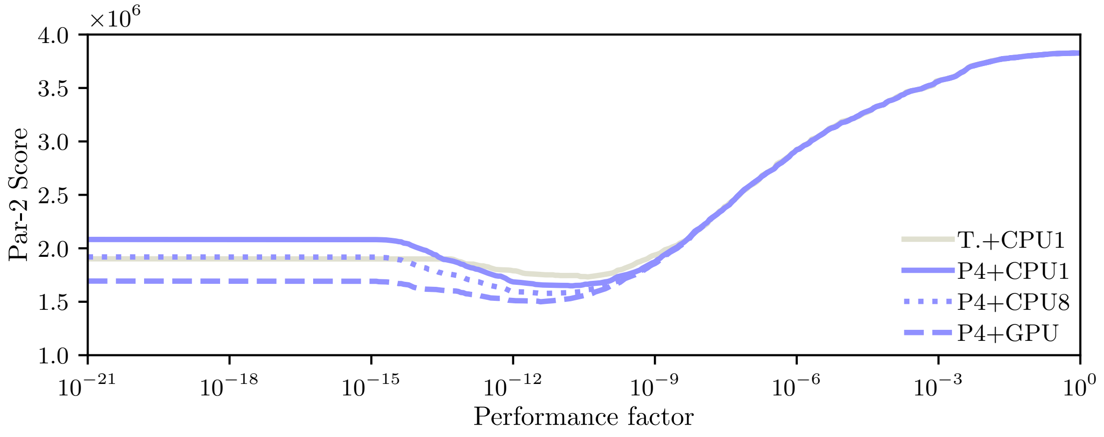


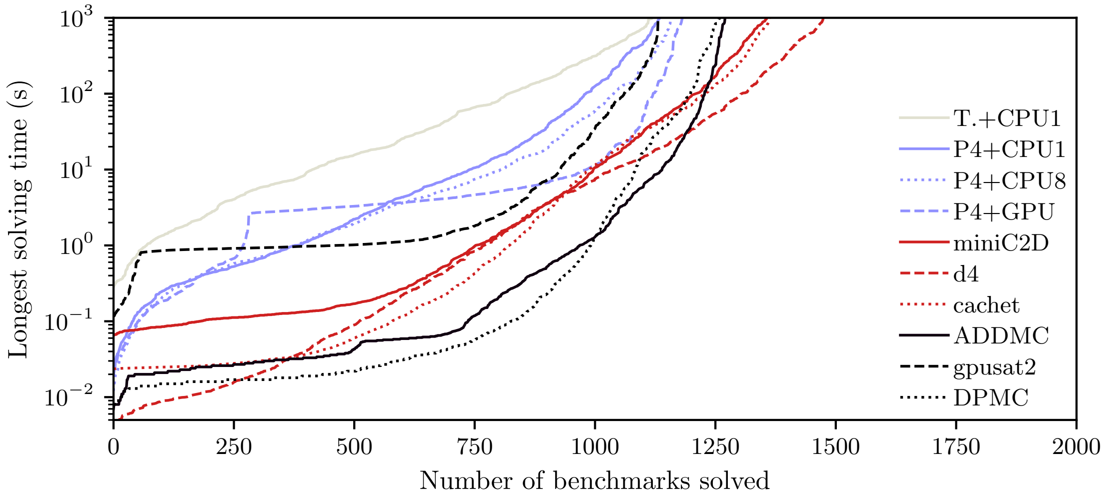
Figure 5.2 : A graph of the simulated PAR-2 score for various combinations of planners and hardware as the performance factor varies.

### 5.3.3 Experiment 2: Determining the Performance Factor (RQ3)

We take each contraction tree discovered in Experiment 1 (with max rank below 36) and use `TensorOrder2` to execute the tree with a timeout of 1000 seconds on each of the three hardware configurations in eager execution mode (`CPU1`, `CPU8`, and `GPU`). We observe that the max rank of almost all solved contraction trees is 30 or smaller.

Given a performance factor, a benchmark, and a planner, we use the planning times from Experiment 1 to determine which contraction tree would have been chosen in step 4 of Algorithm 3.1. We then add the execution time of the relevant contraction tree on each hardware. In this way, we simulate Algorithm 3.1 for a given planner and hardware with many performance factors.

Figure 5.2 indicates how varying the performance factor affects the simulated PAR-2 score for various combinations of planners and hardware. For each planner and hardware, Table 2 shows the performance factor $\alpha$ that minimizes the simulated PAR-2 score. We observe that the performance factor for `CPU8` is lower than for `CPU1`, but not necessarily higher or lower than for `GPU`. We conclude that different combinations of planners and hardware are optimized by different performance factors.



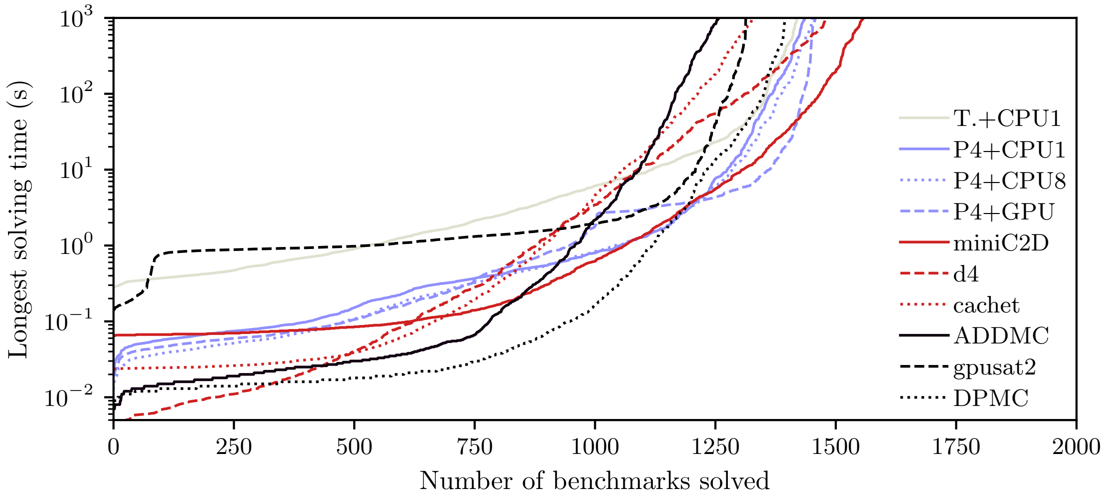


Figure 5.3 : A cactus plot of the number of benchmarks solved by various counters, without (above) and with (below) the `pmc-eq` preprocessor.

Table 5.1 : The performance factor for each combination of planner and hardware that minimizes the simulated PAR-2 score.

| | `Tamaki` | `FlowCutter` | `htd` | `Hicks` | `P3` | `P4` |
|---|---|---|---|---|---|---|
| `CPU1` | $3.8 \cdot 10^{-11}$ | $4.8 \cdot 10^{-12}$ | $1.6 \cdot 10^{-12}$ | $1.0 \cdot 10^{-21}$ | $1.4 \cdot 10^{-11}$ | $1.6 \cdot 10^{-11}$ |
| `CPU8` | $7.8 \cdot 10^{-12}$ | $1.8 \cdot 10^{-12}$ | $1.3 \cdot 10^{-12}$ | $1.0 \cdot 10^{-21}$ | $5.5 \cdot 10^{-12}$ | $6.2 \cdot 10^{-12}$ |
| `GPU` | $2.1 \cdot 10^{-12}$ | $5.5 \cdot 10^{-13}$ | $1.3 \cdot 10^{-12}$ | $1.0 \cdot 10^{-21}$ | $3.0 \cdot 10^{-12}$ | $3.8 \cdot 10^{-12}$ |

### 5.3.4 Experiment 3: End-to-End Performance (RQ4 and RQ6)

Next, we compare `TensorOrder2` with existing state-of-the-art weighted model counters `cachet`, `miniC2D`, `d4`, `ADDMC`, and `gpuSAT2`. We consider `TensorOrder2` using `P4` as the planner combined with each the three hardware configurations in eager execution mode (`CPU1`, `CPU8`, and `GPU`), along with `Tamaki`+`CPU1` as the best non-parallel configuration from [80]. Note that `P4`+`CPU1` still leverages multiple cores in the planning phase. The performance factor from Experiment 2 is used for each `TensorOrder2` configuration.

We run each counter once on each benchmark (both with and without `pmc-eq` preprocessing) with a timeout of 1000 seconds and record the wall-clock time taken. When preprocessing is used, both the timeout and the recorded time include preprocessing time. For `TensorOrder2`, recorded times include all of Algorithm 3.1. Results are summarized in Figure 5.3 and Table 5.2.

We observe that `TensorOrder2` is improved by the portfolio planner and, on hard benchmarks, by executing on a multi-core CPU and on a GPU. The flat line at 3 seconds for `P4` + `GPU` is caused by overhead from initializing the GPU.

Comparing `TensorOrder2` with the other counters, `TensorOrder2` is competitive without preprocessing but solves fewer benchmarks than all other counters, although `TensorOrder2` (with some configuration) is faster than all other counters on 168 benchmarks before preprocessing. We observe that preprocessing significantly boosts

Table 5.2 : The numbers of benchmarks solved by each counter fastest and in total after 1000 seconds, and the PAR-2 score.

| | Without preprocessing | | | With `pmc-eq` preprocessing | | |
|---|---|---|---|---|---|---|
| | # Fastest | # Solved | PAR-2 | # Fastest | # Solved | PAR-2 |
| `T.+CPU1` | 0 | 1119 | 1704469. | 0 | 1421 | 874869. |
| `P4+CPU1` | 47 | 1135 | 1620313. | 85 | 1436 | 840334. |
| `P4+CPU8` | 49 | 1157 | 1556828. | 60 | 1452 | 806600. |
| `P4+GPU` | 72 | 1182 | 1492803. | 47 | 1459 | 780434. |
| `miniC2D` | 47 | 1357 | 1179412. | 127 | 1558 | 610212. |
| `d4` | 542 | 1474 | 951718. | 436 | 1480 | 791105. |
| `cachet` | 230 | 1363 | 1156309. | 207 | 1330 | 1075215. |
| `ADDMC` | 184 | 1270 | 1306260. | 33 | 1258 | 1207856. |
| `gpusat2` | 8 | 1131 | 1589061. | 5 | 1313 | 1071441. |
| `DPMC` | 588 | 1260 | 1339263. | 654 | 1395 | 912741. |

`TensorOrder2` relative to other counters, similar to prior observations with `gpusat2` [28]. `TensorOrder2` solves the third-most preprocessed benchmarks of any solver and has the second-lowest PAR-2 score (notably, outperforming `gpusat2` in both measures). `TensorOrder2` (with some configuration) is faster than all other counters on 192 benchmarks with preprocessing. `TensorOrder2` performs especially well on benchmarks whose incidence graph has low treewidth. Since `TensorOrder2` is the fastest solver on 168 benchmarks without preprocessing and on 192 benchmarks with preprocessing, we conclude that `TensorOrder2` is useful as part of a portfolio of counters.

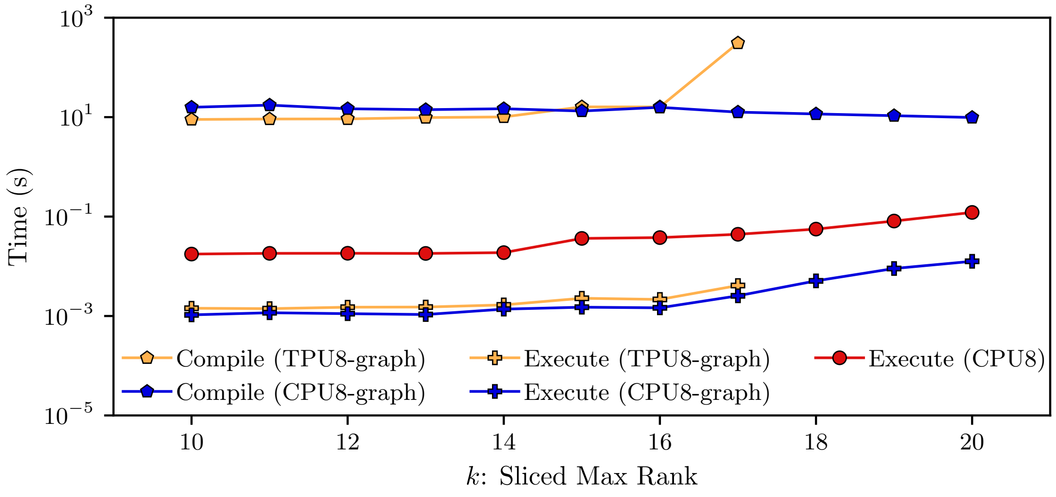


Figure 5.4 : The XLA compilation and average execution time of contracting a single tensor-network slice with a contraction tree of max rank $k$ on a TPU (`TPU-graph`), on a CPU in graph-execution mode (`CPU8-graph`), and on a CPU in eager execution mode (`CPU8`). `TPU-graph` took more than 1000 seconds in the compilation step when $k > 17$.

### 5.3.5 Experiment 4: Executing on a TPU (RQ5)

Finally, we examine the feasibility of leveraging a TPU in the execution phase. We first run the `TPU-graph` executor manually on a subset of benchmarks from Experiment 3. Unfortunately, we were unable to find nontrivial benchmarks (i.e., benchmarks that took more than 1 second to solve in `TensorOrder2` using a single CPU core) that were solvable by `TPU-graph` within 1000 seconds.

To investigate this failure, we consider a tensor network $N$ whose contraction is the number of vertex covers of a randomly-generated cubic graph with 200 vertices [66]. Using the `FlowCutter` planner, we construct a contraction tree $T$ for $N$ of max rank 27 within 10 seconds. For each $k \in \{10, 11, \cdots, 20\}$, then, we slice $N$ greedily to get slice variables $I_k \subseteq \mathcal{B}(N)$ so that $T[\eta]$ has max rank $k$ for each $\eta \in [I_k]$. Using each of the execution configurations `CPU8`, `CPU8-graph`, and `TPU-graph`, we compute `Execute`$(N[\eta], T[\eta])$ for the first 80 slices $\eta \in [I_k]$. We then compute the average time to contract each slice and (for configurations in graph execution mode) the XLA compilation time.

Results are summarized in Figure 5.4. We observe that, as expected, the compilation time in graph execution mode (`CPU8-graph` and `TPU-graph`) is significantly longer than the time for a single slice in eager execution mode (`CPU8`). Moreover, the execution time in graph execution mode is faster than the execution time in eager execution mode.

We also observe that the compilation time of `TPU-graph` scales dramatically with the sliced max rank $k$, which is the maximum number of indices (i.e., dimensions) of the tensors produced as intermediate results throughout the computation. On tensors with more than 17 indices the compilation time of `TPU-graph` is above 1000 seconds and so the compilation times out. For comparison, the max rank of nontrivial benchmarks in Experiment 3 is always more than 25, even after significant slicing. On the other hand, `CPU8-graph` does not suffer from long XLA compilation times even on high-dimensional tensors and so is promising for future analysis.

Finally, we observe that the execution time of `CPU8-graph` is less than the execution time of `TPU-graph`. We hypothesize that this is an artifact of the small tensors involved in these experiments. Since every index in the tensors we consider has size 2, tensors with no more than 17 indices are no larger than 1MB. Once the XLA compilation bottleneck is improved and tensor networks with larger tensors may be used, the execution time of `TPU-graph` may outperform `CPU8-graph`.

We conclude that `TPU-graph` is currently unsuitable for nontrivial model counting benchmarks. The main problem is the long XLA compilation time for computations that involve high-dimensional tensors, i.e. tensors with more than 17 indices. We hypothesize that XLA compilation times are long for high-dimensional tensors because of optimizations for neural network training and inference. Unlike our setting, where tensors often have many indices (more than 25) but each index has a domain of size 2, tensors in neural network training and inference typically have relatively few indices (less than 5) but each index has a much larger domain (thousands) [64, 164].

We emphasize that these results should be taken only as initial observations. Further analysis of XLA compilation with high-dimensional tensors may help to understand and improve the poor performance of tensor-network-based methods for model counting on a TPU. Unfortunately, the implementation of the XLA compiler for a TPU is proprietary and available for use only as a black-box tool. We thus leave an in-depth analysis for future work that has access to internals of the XLA compiler.

## 5.4 Chapter Summary

In this work, we explored the impact of multiple-core, GPU, and TPU use on tensor network contraction for weighted model counting. We implemented our techniques in `TensorOrder2`, a new parallel counter.

The primary finding of this chapter is that tensor networks enable parallel model counting in a variety of hardware contexts.

In the planning phase, we found that a parallel portfolio of graph-decomposition solvers is significantly faster than single-core approaches. We proved that branch decomposition solvers can also be included in this portfolio, but found that a state-of-the-art branch decomposition solver only slightly improves the portfolio.

In the execution phase, we proved that tensor contractions can be performed on a GPU or a TPU. When combined with index slicing, we found that a GPU speeds up the execution phase for many hard benchmarks. For easier benchmarks, the overhead of a GPU may outweigh any contraction speedups. Moreover, we found that current tensor libraries fail to compile tensor computations with high-dimensional tensors to a TPU and so our approach does not scale well on a TPU.

Overall, we found that `TensorOrder2` was the fastest counter on a significant number of benchmarks in comparison to other state-of-the-art counters, especially after preprocessing. Tensor-network techniques are thus useful as part of a portfolio of counters.

# Chapter 6

# Adapting DPMC for Projected Counting

Chapter 4 proposed a unifying framework based on *project-join trees* for dynamic-programming algorithms. The key idea is to consider project-join trees as *execution plans* and decompose dynamic-programming algorithms into two phases: a *planning phase*, where a project-join tree is constructed from an input problem instance, and an *execution phase*, where the project-join tree is used to compute the result.

In this chapter, we adapt this framework for weighted projected model counting. The input in weighted projected model counting is a set of constraints, whose variables are divided into *relevant variables* $X$ and *irrelevant variables* $Y$. The goal is to compute the weighted number of assignments to $X$ that, together with some assignment to $Y$, satisfy the constraints. This enables us to handle applications of discrete integration that require projected counting, e.g. planning [85], formal verification [86], and reliability estimation [87].

The central challenge is that for projected model counting there are two kinds of variables: relevant and irrelevant. This contrasts with model counting, where all variables are relevant and so can be treated similarly. This challenge also occurs for other problems. For example, in Boolean functional synthesis [165], some variables are *free* and must not be projected out. Our solution is to model multiple types of variables by requiring the project-join tree to be *graded*, meaning that irrelevant variables must be projected before relevant variables. The main theoretical contribution of this chapter is a novel algorithm to construct graded project-join trees from standard project-join trees. This has two primary advantages.

The first advantage is that graded project-join trees can be constructed by using existing tools for standard project-join trees (see Chapter 4) in a black-box way. Tools exist to construct standard project-join trees with tree decompositions [40] or with constraint-satisfaction heuristics [75, 77, 78, 114, 143]. We can thus easily leverage all current and future work in tree-decomposition solvers [45, 47, 145] and constraint-satisfaction heuristics to produce graded project-join trees. This is crucial for the practical success of our tool.

The second advantage to our approach is in the simplicity of the algorithm. Given a project-join tree, its gradedness can be easily verified before starting the execution stage. Moreover, the execution algorithm to compute the projected model count from a graded project-join tree is straightforward. This gives us confidence in the correctness of our implementation. During our experimental evaluation, we found correctness errors in `D4`$_\texttt{P}$ [88], `projMC` [88], and `reSSAT` [89]. We reported these issues to the authors, who then fixed the tools. We believe that this work is a step towards a certificate for the verification of projected model counters, similar to SAT solver certificates [166].

The primary contribution of this work is a dynamic-programming framework for weighted projected model counting based on project-join trees. In particular:

1. We show that graded project-join trees can be used to compute weighted projected model counts.

2. We prove that building graded project-join trees and project-join trees with free variables can be reduced to building standard project-join trees.

3. We find that project-join-tree-based algorithms make a significant contribution to the portfolio of exact weighted projected model counters (`D4`$_\texttt{P}$, `projMC`, and `reSSAT`). Our tool, `ProCount`, achieves the shortest solving time on 131 benchmarks of 390 benchmarks solved by at least one tool, from 849 benchmarks in total.

This work was done in collaboration as a joint paper [83].

## 6.1 Using Project-Join Trees for Projected Model Counting

In this section, we adapt the framework from Chapter 4 in order to perform projected model counting. Recall that Chapter 4 describes a two-phase algorithm for computing the weighted model count of a CNF formula $\varphi$. First, the *planning phase* in Section 4.2 builds a project-join tree $(T, r, \gamma, \pi)$ of $\varphi$. Second, the *execution phase* in Section 4.3 computes the $W$-valuation of the project-join tree to obtain the weighted model count.

In order to adapt this framework to perform weighted projected model counting, we aim to modify the valuation of project-join trees to incorporate both disjunctive variables and additive variables. In particular, we must perform $\exists$-projection with all disjunctive variables and $\Sigma$-projection with all additive variables.

The challenge is that not all projections commute: $\Sigma$-projections do not commute with $\exists$-projections in general. Since the $\exists$-projections appear on the inside of the expression for projected model counting, we must ensure that all $\exists$-projections occur before all $\Sigma$-projections while traversing the project-join tree. We formalize this by requiring the project-join tree to be *graded*:

**Definition 6.1** (Graded Project-Join Tree)**.** *Let $\varphi$ be a CNF formula with project-join tree $\mathcal{T} = (T, r, \gamma, \pi)$, and let $\{X, Y\}$ be a partition of* $\texttt{Vars}(\varphi)$*. We say that $\mathcal{T}$ is* $(X, Y)$-graded *if there exist $\mathcal{I}_X, \mathcal{I}_Y \subseteq \mathcal{V}(T)$, called* grades*, where:*

1. *$\{\mathcal{I}_X, \mathcal{I}_Y\}$ is a partition of $\mathcal{V}(T) \setminus \mathcal{L}(T)$,*
2. *if $n_X \in \mathcal{I}_X$, then $\pi(n_X) \subseteq X$,*
3. *if $n_Y \in \mathcal{I}_Y$, then $\pi(n_Y) \subseteq Y$, and*
4. *if $n_X \in \mathcal{I}_X$ and $n_Y \in \mathcal{I}_Y$, then $n_X$ is not a descendant of $n_Y$ in the rooted tree $(T, r)$.*

Intuitively, a project-join tree is $(X, Y)$-graded if all $X$ variables are projected (according to $\pi$) closer to the root than all $Y$ variables in the tree. Figure 6.1 illustrates

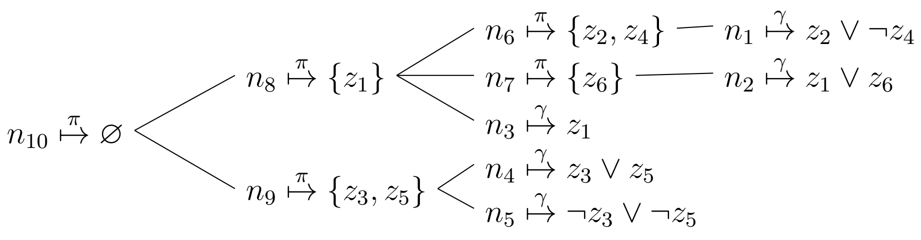

$n_{10} \overset{\pi}{\mapsto} \varnothing$
$n_8 \overset{\pi}{\mapsto} \{z_1\}$
$n_9 \overset{\pi}{\mapsto} \{z_3, z_5\}$
$n_6 \overset{\pi}{\mapsto} \{z_2, z_4\}$
$n_1 \overset{\gamma}{\mapsto} z_2 \vee \neg z_4$
$n_7 \overset{\pi}{\mapsto} \{z_6\}$
$n_2 \overset{\gamma}{\mapsto} z_1 \vee z_6$
$n_3 \overset{\gamma}{\mapsto} z_1$
$n_4 \overset{\gamma}{\mapsto} z_3 \vee z_5$
$n_5 \overset{\gamma}{\mapsto} \neg z_3 \vee \neg z_5$

Figure 6.1 : A graded project-join tree $\mathcal{T} = (T, n_{10}, \gamma, \pi)$ of a CNF formula $\varphi$ with relevant variables $X = \{z_1, z_3, z_5\}$ and irrelevant variables $Y = \{z_2, z_4, z_6\}$. Each leaf node corresponds to a clause of $\varphi$ under $\gamma$. Each internal node is labeled by $\pi$ with a set of variables of $\varphi$. Note that $\mathcal{T}$ is graded with grades $\mathcal{I}_X = \{n_8, n_9, n_{10}\}$ and $\mathcal{I}_Y = \{n_6, n_7\}$.

an exemplary graded project-join tree.

We now define a new valuation on graded project-join trees, which performs $\Sigma$-projections at nodes in $\mathcal{I}_X$ and $\exists$-projections at nodes in $\mathcal{I}_Y$:

**Definition 6.2** (Projected Valuation)**.** *Let $(X, Y, \varphi, W)$ be a weighted projected model counting instance, and let $\mathcal{T} = (T, r, \gamma, \pi)$ be an $(X, Y)$-graded project-join tree of $\varphi$ with grades $\mathcal{I}_X$ and $\mathcal{I}_Y$. The $W$-*projected-valuation* of each node $n \in \mathcal{V}(T)$, denoted $g^W_n$, is defined by*

$$g^W_n \equiv \begin{cases} [\gamma(n)] & \text{if } n \in \mathcal{L}(T) \\ \sum\limits_{\pi(n)} \left( \prod\limits_{o \in \mathcal{C}_{T,r}(n)} g^W_o \cdot \prod\limits_{x \in \pi(n)} W_x \right) & \text{if } n \in \mathcal{I}_X \\ \exists\limits_{\pi(n)} \left( \prod\limits_{o \in \mathcal{C}_{T,r}(n)} g^W_o \right) & \text{if } n \in \mathcal{I}_Y \end{cases}$$

*where $[\gamma(n)]$ is the pseudo-Boolean function corresponding to the clause $\gamma(n) \in \varphi$.*

If the project-join tree is graded, then the projected valuation of the root node is the weighted projected model count.

**Theorem 6.3.** *Let $(X, Y, \varphi, W)$ be an instance of weighted projected model counting, and let $\mathcal{T}$ be a project-join tree of $\varphi$ with root $r$. If $\mathcal{T}$ is $(X, Y)$-graded, then $g^W_r(\varnothing) =$ $\texttt{WPMC}(\varphi, W, Y)$.*

*Proof.* This proof is an extension of the proof of Theorem 4.3 to include $\exists$-projection. Consider each $n \in \mathcal{V}(T)$ and define $S(n)$, $\Phi(n)$, and $P(n)$ as in Lemma 4.4. For ease of notation, further define

$$h_n^W \equiv \Phi(n) \cdot \left( \prod_{x \in P(n) \cap X} W_x \right).$$

The key idea is to prove for all $n \in \mathcal{V}(T)$ that

$$g_n^W = \sum_{P(n) \cap X} \exists_{P(n) \cap Y} h_n^W.$$

We proceed by induction on $n$ over the tree structure of $T$. In the base case, $n$ is a leaf and so $\Phi(n) = [\gamma(n)]$ and $P(n) = \varnothing$. So $g_n^W = \sum_\varnothing \left([\gamma(n)] \cdot \prod_{x \in \varnothing} W_x\right) = [\gamma(n)]$ as desired. In the inductive case, consider an internal node $n$ of $T$ and apply the inductive hypothesis to the following product:

$$\prod_{o \in \mathcal{C}_{T,r}(n)} g_o^W = \prod_{o \in \mathcal{C}_{T,r}(n)} \sum_{P(o) \cap X} \exists_{P(o) \cap Y} h_o^W. \tag{6.1}$$

If $o, o' \in \mathcal{C}_{T,r}(n)$ are distinct, by Lemma 4.4 we know $P(o) \cap \texttt{Vars}(\Phi(o')) = \varnothing$. We can therefore apply Theorem 4.1 repeatedly to Equation 6.1 to get that

$$\prod_{o \in \mathcal{C}_{T,r}(n)} g_o^W = \sum_{A \cap X} \prod_{o \in \mathcal{C}_{T,r}(n)} \exists_{P(o) \cap Y} h_o^W = \sum_{A \cap X} \exists_{A \cap Y} \prod_{o \in \mathcal{C}_{T,r}(n)} h_o^W \tag{6.2}$$

where $A = \bigcup_{o \in \mathcal{C}_{T,r}(n)} P(o)$.

Let $\mathcal{I}_X$ and $\mathcal{I}_Y$ be the grades of $\mathcal{T}$. By Definition 6.1, either $n \in \mathcal{I}_X$ or $n \in \mathcal{I}_Y$. We divide the inductive case further into these two cases.

**Case 1:** $n \in \mathcal{I}_Y$**.** Then for each $p \in S(n)$, by Definition 6.1, we have $p \in \mathcal{I}_Y$, so $\pi(p) \subseteq Y$. Moreover, $\mathcal{V}(S(N)) \subseteq \mathcal{I}_Y$ by Definition 6.1. Thus $A \subseteq Y$. By Definition 6.2 and Equation (6.2), we have

$$g_n^W = \exists_{\pi(n)} \prod_{o \in \mathcal{C}_{T,r}(n)} g_o^W = \exists_{\pi(n)} \exists_{A} \prod_{o \in \mathcal{C}_{T,r}(n)} h_o^W = \exists_{P(n)} \prod_{o \in \mathcal{C}_{T,r}(n)} h_o^W.$$

After plugging in each $h_o^W$, we conclude that

$$g_n^W = \exists_{P(n)} \prod_{o \in \mathcal{C}_{T,r}(n)} \prod_{C \in \Phi(o)} [C] = \exists_{P(n)} \prod_{C \in \Phi(n)} [C] = \exists_{P(n)} h_n^W.$$

Since $P(n) \subseteq Y$, this completes the induction for this case.

**Case 2:** $n \in \mathcal{I}_X$**.** Thus $\pi(n) \subseteq X$. By Definition 6.2 and Equation (6.2), we have

$$\begin{aligned} g_n^W &= \sum_{\pi(n)} \left( \prod_{o \in \mathcal{C}_{T,r}(n)} g_o^W \cdot \prod_{x \in \pi(n)} W_x \right) \\ &= \sum_{\pi(n)} \left( \left( \sum_{A \cap X} \exists_{A \cap Y} \prod_{o \in \mathcal{C}_{T,r}(n)} h_o^W \right) \cdot \prod_{x \in \pi(n)} W_x \right). \end{aligned}$$

Since $\pi(n) \cap A = \varnothing$, we can apply Theorem 4.1 to get that

$$g_n^W = \sum_{\pi(n)} \left( \sum_{A \cap X} \exists_{A \cap Y} \left( \prod_{o \in \mathcal{C}_{T,r}(n)} h_o^W \cdot \prod_{x \in \pi(n)} W_x \right) \right).$$

Finally, observe that $\pi(n) \cup A = P(n)$ and that $h_n^W = \prod_{o \in \mathcal{C}_{T,r}(n)} h_o^W \cdot \prod_{x \in \pi(n)} W_x$. We therefore conclude that

$$g_n^W = \sum_{P(n) \cap X} \exists_{P(n) \cap Y} \left( \prod_{o \in \mathcal{C}_{T,r}(n)} h_o^W \cdot \prod_{x \in \pi(n)} W_x \right) = \sum_{P(n) \cap X} \exists_{P(n) \cap Y} h_n^W.$$

This completes the induction.

Finally, to complete the proof we observe that $P(r) = X \cup Y$ and $\Phi(r) = [\varphi]$. It follows that $g_r^W(\varnothing) = \sum_X \left( [\varphi] \cdot \prod_{x \in X} W_x \right) = \texttt{WPMC}(\varphi, W, Y)$, as desired. □

This gives us a two-phase algorithm for computing the $W$-weighted $Y$-projected model count of a formula $\varphi$ over Boolean variables $X \cup Y$. First, in the *planning* phase, we construct a $\{X, Y\}$-graded project-join tree $(T, r, \gamma, \pi)$ of $\varphi$. We discuss algorithms for constructing graded project-join trees in Section 6.2. Second, in the *execution* phase, we compute $g_r^W$ by following Definition 6.2. We discuss data structures for computing Definition 6.2 in Section 6.3.

## 6.2 Planning Phase: Building Graded Project-Join Trees

In the planning phase, we are given a CNF formula $\varphi$ over Boolean variables $X \cup Y$. The goal is to construct an $\{X, Y\}$-graded project-join tree of $\varphi$.

We now show how building graded project-join trees can be reduced to building ungraded project-join trees. This allows us our techniques for ungraded project-join trees (see Section 4.2) to directly compute graded project-join trees. As a building block, we first show how constructing project-join trees with free variables can be reduced to constructing ungraded project-join trees. This both illustrates the key ideas of our approach and appears as a subroutine in the larger graded reduction.

### 6.2.1 From Free Project-Join Trees to Ungraded Project-Join Trees

Project-join trees project out every variable in the set of corresponding clauses. This is desirable for applications where all variables are processed in the same way, e.g., model counting. In many other applications, however, it is desirable to process a set of clauses while leaving specified *free variables* untouched. This is analogous to the set of free indices of a tensor network (see Definition 2.12).

We model free variables by ensuring that they are projected in the project-join tree as late as possible, at the root node. Thus free variables must be "kept alive" throughout the entire tree.

**Definition 6.4.** *Let $F$ be a set of variables, and let $\mathcal{T} = (T, r, \gamma, \pi)$ be a project-join tree. We say that $\mathcal{T}$ is* $F$-free *if $F = \pi(r)$.*

Note that Definition 6.4 is a much stronger restriction than Definition 6.1. In particular, if a project-join tree $\mathcal{T}$ of a CNF formula $\varphi$ is $F$-free, then $\mathcal{T}$ is also $(F, \texttt{Vars}(\varphi) \setminus F)$-graded.

We now reduce the problem of building $F$-free project-join trees to building ungraded project-join trees. One approach is to build a project-join tree while ignoring all variables in $F$, then insert the variables in $F$ as projections at the root. However, building minimal-width project-join trees while ignoring variables may not produce minimal-width $F$-free project-join trees for the full formula.

Instead, we adapt a similar reduction in the context of tensor networks (see Theorem 3.6, which includes tensor networks with free indices) for the context of project-join trees. The key idea is to add to $\varphi$ a *virtual clause* that contains all variables in $F$. For a set $Z$ of variables, let $\texttt{virtual}(Z)$ denote a fresh clause with variables $Z$. Project-join trees of $\varphi \cup \{\texttt{virtual}(F)\}$ can then be used to find $F$-free project-join trees of $\varphi$. This virtual clause can be viewed as a goal atom in `DataLog` [167], and is analogous to the free vertex in Definition 3.4.

We present this reduction as Algorithm 6.1. The input is a project-join tree $\mathcal{T}$ of $\varphi \cup \{C_F\}$, where $C_F$ is a fresh clause with variables $\texttt{Vars}(C_F) = F$. On lines 2-6, we rotate $\mathcal{T}$. This rotation does not increase the width. We then remove $s$ and obtain a project-join tree of $\varphi$. Projecting $F$ at the new root still does not increase the width.

We state the correctness of Algorithm 6.1 in the following theorem. In particular, the width of the output $F$-free project-join tree is no worse than the width of the unrestricted input tree.

**Theorem 6.5.** *Let $\varphi$ be a CNF formula and let $F \subseteq \texttt{Vars}(\varphi)$. If $\mathcal{T}$ is a project-join tree of $\varphi \cup \{\texttt{virtual}(F)\}$, then Algorithm 6.1 returns an $F$-free project-join tree of $\varphi$ of width at most $\texttt{width}(\mathcal{T})$.*

*Proof.* Let $C_F = \texttt{virtual}(F)$. Let $\mathcal{T} = (T, r, \gamma, \pi)$ be the input project-join tree for $\varphi \cup \{C_F\}$, and let $\mathcal{T}' = (T, s, \gamma', \pi')$ be the output of Algorithm 6.1. Moreover, let $\texttt{Vars}_{\mathcal{T}}(n)$ and $\texttt{Vars}_{\mathcal{T}'}(n)$ denote the sets of variables at a node $n$ of $\mathcal{T}$ and $\mathcal{T}'$ respectively.

First, we prove that $\mathcal{T}'$ is a project-join tree. Since $\gamma$ is a bijection onto $\varphi \cup \{C_F\}$ and we have removed both $C_F$ and the leaf corresponding to $C_F$, $\gamma'$ is indeed a bijection onto $\varphi$. Moreover, since every variable appears in exactly one set in the image of $\pi'$, the first condition of Definition 4.2 is satisfied. Finally, each variable $y$

**Algorithm 6.1:** Building an $F$-free project-join tree of a CNF formula

```
    Input: $\varphi$: a CNF formula
    Input: $F$: a subset of $\texttt{Vars}(\varphi)$
    Input: $\mathcal{T} = (T, r, \gamma, \pi)$: a project-join tree of $\varphi \cup \{C_F\}$, where
             $C_F = \texttt{virtual}(F)$ is a virtual clause with variables $F$
    Output: an $F$-free project-join tree of $\varphi$
1   $s \leftarrow \gamma^{-1}(C_F)$ // $s$ will be the root node of the returned project-join tree
2   $\pi' \leftarrow$ a mapping where $\pi'(n) = \varnothing$ for all $n \in \mathcal{V}(T) \setminus \mathcal{L}(T)$ // $\pi'$ will be the
      labeling function of the returned project-join tree
3   for $y \in \texttt{Vars}(\varphi) \setminus F$
4       $\varphi_y = \{C \in \varphi : y \in \texttt{Vars}(C)\}$
5       $i \leftarrow$ lowest common ancestor of $\{\gamma^{-1}(C) : C \in \varphi_y\}$ in the rooted tree
          $(T, s)$
6       $\pi'(i) \leftarrow \pi'(i) \cup \{y\}$ // project out $y$ at the lowest allowable node
7   $\pi'(s) \leftarrow F$ // project out variables in $F$ at the new root $s$
8   $\gamma' \leftarrow \gamma \setminus \{s \mapsto C_F\}$ // $\gamma'$ is the bijection $\gamma$ without the pair $(s, C_F)$
9   return $(T, s, \gamma', \pi')$
```

is projected out at the lowest common ancestor of all leaves corresponding to clauses that contain $y$; thus the second condition of Definition 4.2 is satisfied. It follows that $\mathcal{T}'$ is a project-join tree.

Second, we prove that $\mathcal{T}'$ is $F$-free. Since $s$ has degree 1, $s$ is never the lowest common ancestor of a set of leaves of $\mathcal{T}'$. Thus $\pi'(s) \setminus F = \varnothing$. By line 7, it follows that $\pi'(s) = F$, so $\mathcal{T}'$ is $F$-free.

Finally, we prove that the width of $\mathcal{T}'$ is at most $\texttt{width}(\mathcal{T})$. To do this, if $S$ is a project-join tree and $n$ is a node of $S$, define $\texttt{rel}_S(n) \equiv \texttt{Vars}_S(n)$ for leaf nodes and $\texttt{rel}_S(n) \equiv \texttt{Vars}_S(n) \cup \pi(n)$ for internal nodes. Notice the size of $n$ in $S$ is exactly $|\texttt{rel}_S(n)|$, so the width of $S$ is exactly the maximum size of $\texttt{rel}_S(n)$ across all nodes

$n$.

Consider an arbitrary node $n \in \mathcal{V}(T) \setminus \{s\}$. Define:

$$\begin{aligned}
A(n) = \{y : \exists \ell \in \mathcal{L}(T) \text{ s.t. } &\ell \text{ is a descendant of } n \text{ in the rooted tree } (T, s) \text{ and} \\
&y \in \texttt{rel}_{\mathcal{T}'}(\gamma'(\ell))\} \\
B(n) = \{x \in \texttt{Vars}(\varphi) : \exists \ell, \ell' \in \mathcal{L}(T) \text{ s.t. } &n \text{ is between } \ell, \ell' \text{ in } (T, r) \text{ and} \\
&x \in \texttt{rel}_{\mathcal{T}}(\gamma(\ell)) \cap \texttt{rel}_{\mathcal{T}}(\gamma(\ell'))\} \\
B'(n) = \{x \in \texttt{Vars}(\varphi) : \exists \ell, \ell' \in \mathcal{L}(T) \text{ s.t. } &n \text{ is between } \ell, \ell' \text{ in } (T, s) \text{ and} \\
&x \in \texttt{rel}_{\mathcal{T}'}(\gamma'(\ell)) \cap \texttt{rel}_{\mathcal{T}'}(\gamma'(\ell'))\}
\end{aligned}$$

Note that a node $n$ is *between* $\ell, \ell' \in \mathcal{V}(T)$ if $n$ is on the unique shortest path between $\ell$ and $\ell'$. There are several key relationships among $A(n)$, $B(n)$, $B'(n)$, $\texttt{rel}_{\mathcal{T}}(n)$, and $\texttt{rel}_{\mathcal{T}'}(n)$:

1. By the construction in Algorithm 6.1, we know $\texttt{rel}_{\mathcal{T}'}(n) \setminus F = B'(n) \setminus F$ and $\texttt{rel}_{\mathcal{T}'}(n) \cap F = A(n) \cap F$.
2. Since $\gamma$ and $\gamma'$ agree on all nodes of $T$ except for $s$, we know $B(n) \setminus F = B'(n) \setminus F$.
3. Since $\texttt{Vars}(C_F) = F$, we know $B(n) \cap F = A(n) \cap F$.
4. By Property 2 of Definition 4.2, $B(n) \subseteq \texttt{rel}_{\mathcal{T}}(n)$.

Putting these relationships together, we observe that:

$$\begin{aligned}
\texttt{rel}_{\mathcal{T}'}(n) &= (\texttt{rel}_{\mathcal{T}'}(n) \setminus F) \cup (\texttt{rel}_{\mathcal{T}'}(n) \cap F) \\
&= (B'(n) \setminus F) \cup (A(n) \cap F) \\
&= (B(n) \setminus F) \cup (B(n) \cap F) \\
&= B(n) \\
&\subseteq \texttt{rel}_{\mathcal{T}}(n)
\end{aligned}$$

Finally, observe that $\texttt{rel}_{\mathcal{T}'}(s) = \texttt{rel}_{\mathcal{T}}(s) = F$. Hence the width of $\mathcal{T}'$ is indeed no larger than the width of $\mathcal{T}$, as desired. $\square$

We also prove that Algorithm 6.1 is optimal. That is, a minimal-width project-join tree for $\varphi \cup \{C_F\}$ produces a minimal-width $F$-free project-join tree for $\varphi$.

**Theorem 6.6.** *Let $\varphi$ be a CNF formula, let $F \subseteq \texttt{Vars}(\varphi)$, and let $w$ be a positive integer. If there is an $F$-free project-join tree of $\varphi$ of width $w$, then there is a project-join tree of $\varphi \cup \{\texttt{virtual}(F)\}$ of width $w$.*

*Proof.* Let $\mathcal{T}$ be an $F$-free project-join tree for $\varphi$ of width at most $w$. Produce $\mathcal{T}'$ by attaching to the root of $\mathcal{T}$ a new leaf node corresponding to $C_F$. Then $\mathcal{T}'$ is a project-join tree for $\varphi \cup \{C_F\}$, and its width is identical to $\mathcal{T}$. □

### 6.2.2 From Graded Project-Join Trees to Ungraded Project-Join Trees

In this section, we use free project-join trees as a building block to construct graded project-join trees. We present this framework as Algorithm 6.2. The key idea is to create a graded project-join tree by combining many free project-join trees for sub-formulas. We first combine clauses to remove $Y$ variables, then we combine project-join-tree components to remove $X$ variables.

In detail, on line 1, we partition the clauses of $\varphi$ into blocks that share $Y$ variables. On line 3, we find a project-join tree $\mathcal{T}_N$ for each block $N$. This tree must keep all $X$ variables free, i.e., must be $(\texttt{Vars}(N) \cap X)$-free. The trees $\{\mathcal{T}_N\}$ collectively project out all $Y$ variables. On line 6, we construct a project-join tree $\mathcal{T}$ that will guide the combination of all trees in $\{\mathcal{T}_N\}$ while projecting out all $X$ variables, where each $\mathcal{T}_N$ is represented by the corresponding virtual clause $C_N$. On lines 7-9, we hook the trees in $\{\mathcal{T}_N\}$ together as indicated by $\mathcal{T}$.

The function $\texttt{GroupBy}(\varphi, Y)$ in Algorithm 6.2 partitions the clauses of $\varphi$ so that every pair of clauses that share a variable from $Y$ appear together in the same block of the partition. Formally:

**Definition 6.7.** *Let $\varphi$ be a set of clauses, and let $Y$ be a subset of $\texttt{Vars}(\varphi)$. Define $\sim_Y \subseteq \varphi \times \varphi$ to be the relation such that, for clauses $c, c' \in \varphi$, we have $c \sim_Y c'$ if and*

**Algorithm 6.2:** Building a graded project-join tree of a CNF formula

```
   Input: $X$: a set of $\Sigma$-variables
   Input: $Y$: a set of $\exists$-variables, where $X \cap Y = \varnothing$
   Input: $\varphi$: a CNF formula where $\texttt{Vars}(\varphi) = X \cup Y$
   Output: $\mathcal{T}$: an $(X, Y)$-graded project-join tree of $\varphi$
1  $\textit{partition} \leftarrow \texttt{GroupBy}(\varphi, Y)$ // group clauses that share $Y$ variables
2  for $N \in \textit{partition}$
3      $\mathcal{T}_N \leftarrow \texttt{BuildComponent}(N, \texttt{Vars}(N) \cap X)$ // build a $(\texttt{Vars}(N) \cap X)$-free
          project-join tree of $N$
4      $\mathcal{T}_N \leftarrow \mathcal{T}_N$ with all projections at the root of $\mathcal{T}_N$ removed
5      $C_N \leftarrow \texttt{virtual}(\texttt{Vars}(N) \cap X)$
6  $\mathcal{T} \leftarrow \texttt{BuildComponent}(\{C_N : N \in \textit{partition}\}, \varnothing)$ // build a project-join tree
      from virtual clauses $C_N$
7  for $N \in \textit{partition}$
8      $\ell_N \leftarrow$ leaf of $\mathcal{T}$ corresponding to $C_N$
9      $\mathcal{T} \leftarrow \mathcal{T}$ with $\ell_N$ replaced by $\mathcal{T}_N$
10 return $\mathcal{T}$
```

*only if* $\texttt{Vars}(c) \cap \texttt{Vars}(c') \cap Y \neq \varnothing$. *Then* $\texttt{GroupBy}(\varphi, Y)$ *is the set of equivalence classes of the reflexive transitive closure of* $\sim_Y$.

The intuition is that two clauses in the same block in $\texttt{GroupBy}(\varphi, Y)$ must be combined to project out all variables in $Y$. Conversely, clauses that appear in separate blocks need not be combined in order to project out all variables in $Y$.

In Algorithm 6.2, each call to $\texttt{BuildComponent}(\alpha, F)$ returns an $F$-free project-join tree of $\alpha$, where $\alpha$ is a set of clauses and $F \subseteq \texttt{Vars}(\alpha)$. `BuildComponent` can be implemented by implementing Algorithm 6.1 on top of an algorithm for building ungraded project-join trees. For example, in Section 6.4, we consider two implementations of Algorithm 6.2 built on top of the two algorithms to construct standard project-join trees [81] discussed in Chapter 4.

We next state the correctness of Algorithm 6.2 and show that the width of the output graded project-join tree is no worse than the widths of the trees used for the components. Formally:

**Theorem 6.8.** *Let $\varphi$ be a CNF formula, let $\{X, Y\}$ be a partition of* `Vars`$(\varphi)$*, and let $w$ be a positive integer. Assume each call to* `BuildComponent`$(\alpha, F)$ *returns an $F$-free project-join tree for $\alpha$ of width at most $w$. Then Algorithm 6.2 returns an $(X, Y)$-graded project-join tree for $\varphi$ of width at most $w$.*

*Proof.* Let $\mathcal{T}$ be the project-join tree produced on line 6, and let $\mathcal{T}' = (T, r, \gamma, \pi)$ be the project-join tree returned by Algorithm 6.2. By Definition 6.7, every $y \in Y$ is a variable of exactly one $N_y \in$ `GroupBy`$(\varphi, Y)$. It follows that every $y \in Y$ is projected out at exactly one node of $\mathcal{T}'$, namely the node at which $y$ is projected out in $\mathcal{T}_{N_y}$. Similarly, after the loop on line 2 completes, no variable from $X$ is projected out across all of $\{\mathcal{T}_N : N \in$ `GroupBy`$(\varphi, Y)\}$, since all $X$ projections are removed on line 4. Thus every $x \in X$ is also projected out at exactly one node of $\mathcal{T}'$, namely the node at which $x$ is projected out in $\mathcal{T}$. Thus $\mathcal{T}'$ satisfies the first property of Definition 4.2.

We prove the second property of Definition 4.2 by contrapositive. That is, assume that there is some $n \in \mathcal{V}(T) \setminus \mathcal{L}(T)$, variable $z \in \pi(n)$, and $c \in \varphi$ s.t. $z \in$ `Vars`$(c)$ but $\gamma^{-1}(c)$ is not a descendant of $n$ in $\mathcal{T}'$. Let $N$ be the block of `GroupBy`$(\varphi, Y)$ that contains $c$. We split into two cases:

- *Case: $z \in Y$.* Let $n'$ be the node in $\mathcal{T}_N$ that corresponds to $n$, where $z$ is projected. Then $\gamma^{-1}(c)$ is not a descendant of $n'$ in $\mathcal{T}_N$. It follows that $\mathcal{T}_N$ is not a project-join tree.

- *Case: $z \in X$.* Let $n'$ be the node in $\mathcal{T}$ that corresponds to $n$, where $z$ is projected. Since $\gamma^{-1}(c)$ is not a descendant of $n$ in $\mathcal{T}$, the leaf corresponding to

$C_N$ is not a descendant of $n'$ in $\mathcal{T}$. But $z \in \texttt{Vars}(N) \cap X$, so $z \in \texttt{Vars}(C_N)$. It follows that $\mathcal{T}$ is not a project-join tree.

We conclude that $\mathcal{T}'$ satisfies the second property of Definition 4.2 provided that `BuildComponent` always returns project-join trees.

Finally, we prove that the width of $\mathcal{T}'$ is at most $w$. To see this, we observe that the set of variables at each node of $\mathcal{T}'$ is exactly the set of variables appearing at the node of the corresponding component project-join tree. The width of $\mathcal{T}'$ is thus the maximum size that appears across all component project-join trees returned by `BuildComponent`. □

Although Algorithm 6.2 constructs a sequence of small ungraded project-join trees, it is sufficient to compute a single ungraded project-join tree from which all smaller trees can be extracted. This is demonstrated by the following theorem.

**Theorem 6.9.** *Let $\varphi$ be a CNF formula, let $\{X, Y\}$ be a partition of* $\texttt{Vars}(\varphi)$*, and let $w$ be a positive integer. Let $\psi$ be the CNF formula $\varphi \cup \{\texttt{virtual}(\texttt{Vars}(N) \cap X) : N \in \texttt{GroupBy}(\varphi, Y)\}$. If there is a project-join tree $\mathcal{T}'$ for $\psi$ of width $w$, then there is an $(X, Y)$-graded project-join tree for $\varphi$ of width at most $w$.*

*Proof.* The key idea of the proof is to answer every `BuildComponent` call in Algorithm 6.2 by extracting a subtree of $\mathcal{T}'$ and applying Theorem 6.5.

We first show that, for each call to $\texttt{BuildComponent}(\alpha, F)$ in Algorithm 6.2, there is an $F$-free project-join tree for $\alpha$ of width at most $w$. There are two cases to consider: the calls on line 3 and the call on line 6.

- *Case: line 3.* Consider some $N \in \texttt{GroupBy}(\varphi, Y)$. Our goal is to find a $(\texttt{Vars}(N) \cap X)$-free project-join tree for $N$. Observe that $N \cup \{C_N\}$ is a subset of $\psi$. Thus let $S_N$ be the smallest subtree of $\mathcal{T}'$ containing all leaves labeled by some element of $N \cup \{C_N\}$. $S_N$ is a project-join tree for $N \cup \{C_N\}$ whose width is no more than $w$. By Theorem 6.5, there is a $(\texttt{Vars}(N) \cap X)$-free project-join tree for $N$ of width no more than $w$.

- *Case: line 6.* Similarly, let $N' = \{C_N : N \in \texttt{GroupBy}(\varphi, Y)\}$ and observe that $N'$ is a subset of $\psi$. Let $S$ be the smallest subtree of $\mathcal{T}'$ containing all leaves labeled by some element of $N'$. Then $S$ is a project-join tree for $N'$ whose width is no more than $w$.

It then follows from Theorem 6.8 that there is an $(X, Y)$-graded project-join tree for $\varphi$ of width at most $w$. □

We show in the following theorem that this approach is optimal. Thus $(X, Y)$-graded project-join trees of $\varphi$ are equivalent to project-join trees of $\psi$.

**Theorem 6.10.** *Let $\varphi$ be a CNF formula, let $\{X, Y\}$ be a partition of $\texttt{Vars}(\varphi)$, and let $w$ be a positive integer. Let $\psi$ be the CNF formula $\varphi \cup \{\texttt{virtual}(\texttt{Vars}(N) \cap X) : N \in \texttt{GroupBy}(\varphi, Y)\}$. If there is an $(X, Y)$-graded project-join tree for $\varphi$ of width $w$, then there is a project-join tree for $\psi$ of width $w$.*

*Proof.* Let $\mathcal{T}$ be an $(X, Y)$-graded project-join tree for $\varphi$ of width $w$, and let $\mathcal{I}_X, \mathcal{I}_Y$ be the grades of $\mathcal{T}$. We first aim to show that, for every $N \in \texttt{GroupBy}(\varphi, Y)$, there is some node $n_N$ of $\mathcal{T}$ with $N \cap X \subseteq \texttt{Vars}(n_N)$.

Consider an arbitrary $N \in \texttt{GroupBy}(\varphi, Y)$. If $|N| = 1$, define $n_N$ to be the node of $\mathcal{T}$ corresponding to the only element of $N$; thus $\texttt{Vars}(n_N) = \texttt{Vars}(N)$, so indeed $N \cap X \subseteq \texttt{Vars}(n_N)$. Otherwise, define $n_N \in \mathcal{V}(T)$ to be the lowest common ancestor of $N$ in $\mathcal{T}$. Then there exist distinct clauses $A, B \in N$ s.t. $n_N$ is also the lowest common ancestor of the two leaves labeled by $A$ and $B$. By Definition 6.7, since $A, B \in N$, there must be a sequence $C_1, C_2, \cdots, C_k \in N$ s.t. $C_1 = A$, $C_k = B$, and for each $1 \leq i \leq k$, we have $\texttt{Vars}(C_i) \cap \texttt{Vars}(C_{i+1}) \cap Y \neq \varnothing$. Since $n_N$ is the lowest common ancestor of $A$ and $B$, and because $\mathcal{T}$ is a tree, there must be some $1 \leq i \leq k$ s.t. $n_N$ is also the lowest common ancestor of $C_i$ and $C_{i+1}$. Thus $\texttt{Vars}(C_i) \cap \texttt{Vars}(C_{i+1}) \subseteq \texttt{Vars}(n_N)$. Since $\texttt{Vars}(C_i) \cap \texttt{Vars}(C_{i+1}) \cap Y \neq \varnothing$, it follows that $Y \cap \texttt{Vars}(n_N) \neq \varnothing$ as well. Thus $n_N \in \mathcal{I}_Y$. By Definition 6.1, this means that for all descendants $o$ of $n_N$, $\pi(o) \cap X = \varnothing$. It follows that $\texttt{Vars}(n_N)$ must still contain all variables in $N$ from $X$; that is, $\texttt{Vars}(N) \cap X \subseteq \texttt{Vars}(n_N)$.

Construct $\mathcal{T}'$ from $\mathcal{T}$ by, for each $N \in \texttt{GroupBy}(\varphi, Y)$, attaching a new leaf labeled by $\texttt{virtual}(\texttt{Vars}(N) \cap X)$ as a child of $n_N$. Since $N \cap X \subseteq \texttt{Vars}(n_N)$ in the initial tree, the width of $\mathcal{T}'$ is equal to the width of $\mathcal{T}$. Moreover, $\mathcal{T}'$ is now a project-join tree for $\psi$. $\square$

We emphasize that our theoretical results do not imply that gradedness can be obtained for a project-join free with no increase in width. Requiring the project-join tree to be graded may significantly increase the width of available project-join trees. Rather, Theorems 6.8 and 6.10 indicate that our algorithm for constructing a graded project-join tree pays no additional cost in width beyond what is required by gradedness.

## 6.3 Execution Phase: Performing the Projected Valuation

In the execution phase, we are given a CNF formula $\varphi$ over variables $X \cup Y$, an $\{X, Y\}$-graded project-join tree $(T, r, \gamma, \pi)$ of $\varphi$, and a literal-weight function $W$ over $X$. The goal is to compute the $W$-projected-valuation $g_r^W$ using Definition 6.2. Several data structures can be used for the pseudo-Boolean functions that occur while using Definition 6.2.

In Chapter 4, *algebraic decision diagrams (ADDs)* were used for computing $W$-valuations. An ADD is a compact representation of a pseudo-Boolean function as a directed acyclic graph [67]. ADDs were found to outperform tensors in the execution phase for model counting on a single-core, especially on benchmarks where the width of the project-join tree was above 30. We observe in Section 6.4.2 that for projected counting most benchmarks have large width, since requiring the project-join trees to be graded often leads to an increase in minimum width. In this work, we therefore use ADDs to compute $W$-projected-valuations.

This work was done by Vu H. N. Phan and is not a contribution of this dissertation. We refer the reader to [168] for the complete details.

## 6.4 Implementation and Evaluation

To implement our projected model counter `ProCount`, we modify the unprojected model counter `DPMC` (see Chapter 4), which is based on ungraded project-join trees. The `DPMC` framework includes: (1) the `LG` planner that uses tree-decomposition techniques, (2) the `HTB` planner that uses constraint-satisfaction heuristics, and (3) the `DMC` executor that uses algebraic decision diagrams (ADDs). We generalize these three components to support graded project-join trees and projected model counting.

We aim to answer the following experimental research questions:

(RQ1) In the planning phase, how do tree-decomposition techniques compare with constraint-satisfaction heuristics?

(RQ2) In the execution phase, how do different ADD variable orders compare?

(RQ3) How does `ProCount` compare with other exact weighted projected counters?

To answer RQ1, in Experiment 1 we compare the planner `LG` (which uses tree decompositions) and the planner `HTB` (which uses constraint-satisfaction heuristics). To answer RQ2, in Experiment 2 we compare variable-ordering heuristics for the ADD-based executor `DMC`. To answer RQ3, in Experiment 3 we compare `ProCount` to state-of-the-art exact weighted projected model counters $\texttt{D4}_{\texttt{P}}$ [88], `projMC` [88], and `reSSAT` [89].

We use 849 CNF benchmarks gathered from two families. The first family contains 90 formulas and was used for weighted projected sampling [169]. For each benchmark in this family, a positive literal $x$ has weight $0 < W_x(\{x\}) < 1$, and a negative literal $\neg x$ has weight $W(\varnothing) = 1 - W_x(\{x\})$. The second family contains 759 formulas and was used for unweighted projected model counting [93]. We add weights to this family by randomly assigning $W_x(\{x\}) = 0.4$ and $W_x(\varnothing) = 0.6$ or vice versa to each variable $x$. All 849 benchmarks are satisfiable, as verified by the SAT solver `CryptoMiniSat` [170].

We run all experiments on single CPU cores of a Linux cluster with Intel Xeon E5-2650v2 processors (2.60-GHz) and 30 GB of RAM*. All code and data are available (https://github.com/vardigroup/DPMC).

### 6.4.1 Implementation Details of `ProCount`

**Planning.** `ProCount` contains two implementations of the planning phase: `LG` (Line Graph) and `HTB` (Heuristic Tree Builder). `LG` uses the tree decomposers `FlowCutter` [145], `htd` [45], and `Tamaki` [47]. While `LG` is an *anytime* tool that produces several graded project-join trees (of decreasing widths) for each benchmark, we use only the first tree produced on each benchmark. We find that this does not significantly affect the performance of `ProCount`.

`HTB` implements four heuristics for variable ordering: maximal-cardinality search (**MCS**) [77], lexicographic search for perfect/minimal orders (**LP**/**LM**) [78], and min-fill (**MF**) [114]. `HTB` also implements two clause-ordering heuristics: bucket elimination (**BE**) [75] and Bouquet's Method (**BM**) [143]

**Execution.** `ProCount` contains a single implementation of the execution phase: `DMC` (Diagram Model Counter). `DMC` uses ADDs as the underlying data structure with `CUDD` [72]. Note that we use ADDs throughout the entire execution for consistency, although binary decision diagrams [115] or SAT solvers would suffice to valuate existential nodes. The variable order of an ADD strongly influences its compactness. `DMC` implements four variable-ordering heuristics (see above): **MCS**, **LP**, **LM**, and **MF**.

* Vu H. N. Phan implemented the modifications in `HTB` and `DMC`, and ran the experiments.

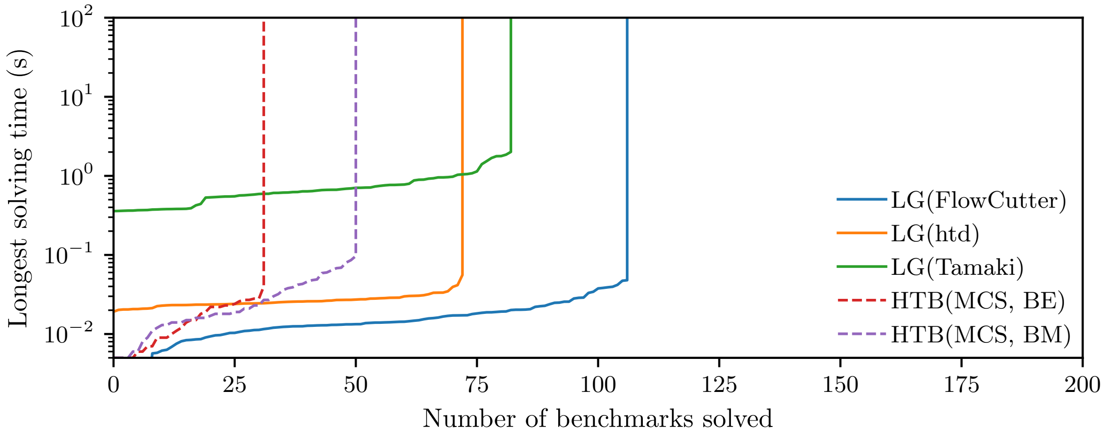


Figure 6.2 : Experiment 1 compares the tree-decomposition-based planner `LG` to the constraint-satisfaction-based planner `HTB`. A planner "solves" a benchmark when it finds a project-join tree of width 30 or lower. For `HTB`, we only show the variable-ordering heuristic **MCS**; the **LP**, **LM**, and **MF** curves are qualitatively similar.

### 6.4.2 Experiment 1: Comparing Planners

In this experiment, we run all configurations of the planners `LG` and `HTB` once on each CNF benchmark with a timeout of 100 seconds. We present results in Figure 6.2. Each point $(x, y)$ on a plotted curve indicates that, within $x$ seconds on each of $y$ benchmarks, the first graded project-join tree produced by the corresponding planner has width at most 30. We choose 30 because previous work shows that executors do not handle larger project-join trees well [80, 81].

The tree-decomposition-based planner `LG` outputs more low-width trees than the constraint-satisfaction-based planner `HTB`. Moreover, for `LG`, the tree decomposer `FlowCutter` is faster than `htd` and `Tamaki`. Thus we use `LG` with `FlowCutter` in `ProCount` for later experiments.

### 6.4.3 Experiment 2: Comparing Execution Heuristics

In this experiment, we take all 346 graded project-join trees produced by `LG` with `FlowCutter` in Experiment 1 and run `DMC` once for 100 seconds with each of four

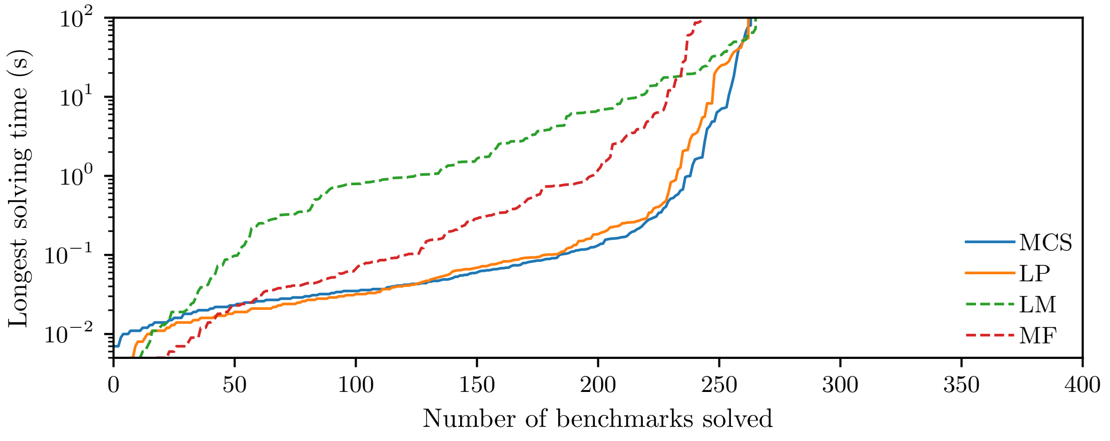


Figure 6.3 : Experiment 2 compares variable-ordering heuristics (**MCS**, **LP**, **LM**, and **MF**) for the ADD-based executor `DMC`. **MCS** and **LP** are significantly faster than **LM** and **MF**.

ADD variable-ordering heuristics. We present the execution time of each heuristic (excluding planning time) in Figure 6.3. We observe that **MCS** and **LP** outperform **LM** and **MF**. We use `DMC` with **MCS** in `ProCount` for Experiment 3.

### 6.4.4 Experiment 3: Comparing Weighted Projected Model Counters

Informed by Experiments 1 and 2, we choose `LG` with `FlowCutter` as the planner and `DMC` with **MCS** as the executor for our framework `ProCount`. We compare `ProCount` with the weighted projected model counters $\texttt{D4}_{\texttt{P}}$, `projMC`, and `reSSAT`. Since all benchmarks are satisfiable with positive literal weights, the model counts must be positive. Thus, for all tools, we exclude outputs that are zero (possible floating-point underflow). We are confident that the remaining results are correct. Differences in model counts among tools are less than $10^{-6}$.

Figure 6.4 shows the performance of `ProCount`, $\texttt{D4}_{\texttt{P}}$, `projMC`, and `reSSAT` with a 1000-second timeout. Additional statistics are given in Table 6.1. Of 849 benchmarks, 390 are solved by at least one of four tools. `ProCount` achieves the shortest solving time on 131 benchmarks, including 44 solved by none of the other three tools. Between

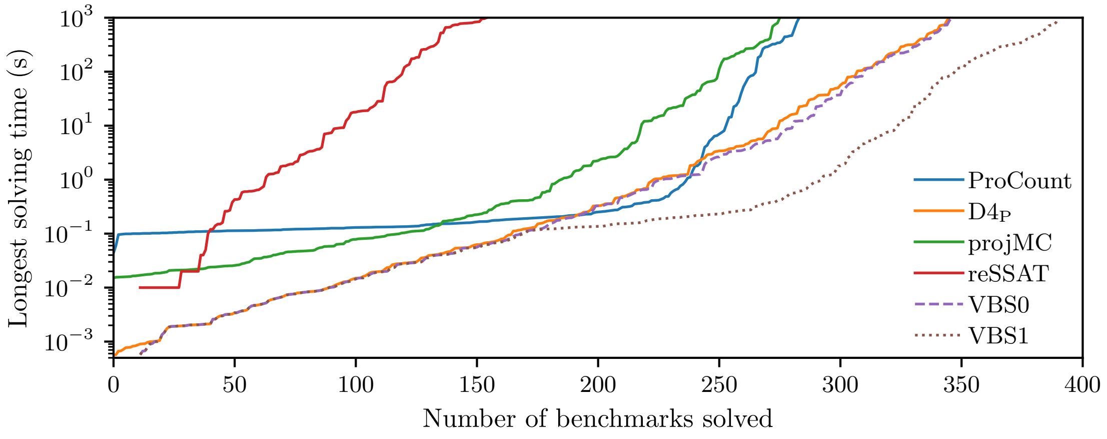


Figure 6.4 : Experiment 3 compares our framework `ProCount` to the state-of-the-art exact weighted projected model counters `D4`$_{\text{P}}$, `projMC`, and `reSSAT`. `VBS0` is the virtual best solver of the three existing tools, excluding `ProCount`. `VBS1` includes all four tools. Adding `ProCount` significantly improves the portfolio of projected model counters.

the two *virtual best solvers* in Figure 6.4, `VBS1` (all four tools) is significantly faster than `VBS0` (three existing tools, without `ProCount`).

### Project-Join Tree Width and Computation Time

To identify which type of benchmarks can be solved efficiently by `ProCount`, we study how the performance of each projected model counter varies with the widths of graded project-join trees. In particular, for each benchmark, we consider the width of the first graded project-join tree produced by the planner `LG` in Experiment 1. Figure 6.5 shows how these widths relate to PAR-2 scores of projected model counters. `ProCount` seems to be the fastest solver on instances for which there exist graded project-join trees of widths between 50 and 100.

Table 6.1 : Experiment 3 compares our framework `ProCount` to the state-of-the-art exact weighted projected model counters `D4`$_{\texttt{P}}$, `projMC`, and `reSSAT`. For each solver, the PAR-2 score is the cumulative solving time of completed benchmarks plus twice the 1000-second timeout for each unsolved benchmark. There are 390 benchmarks solved by at least one of four tools. By including `ProCount`, the portfolio of tools solves 44 more benchmarks and achieves shorter solving time on 87 other benchmarks.

| Tool | Number of benchmarks solved (of 849) | | | PAR-2 score |
|---|---|---|---|---|
| | By no other | In shortest time | In total | |
| `ProCount` | 44 | 131 | 283 | 1139215 |
| `D4`$_{\texttt{P}}$ | 50 | 235 | 345 | 1021809 |
| `projMC` | 0 | 8 | 275 | 1157018 |
| `reSSAT` | 1 | 16 | 154 | 1408853 |
| `VBS0` | - | - | 346 | 1018784 |
| `VBS1` | - | - | 390 | 933494 |

## 6.5 Related Work

It is worth comparing our theoretical results to a different algorithm for (unweighted) projected counting [171], which runs on a formula $\varphi$ in time $2^{2^{O(k)}} \cdot p(\varphi)$, where $k$ is the *primal treewidth* [51] of $\varphi$, and $p$ scales polynomially in the size of $\varphi$. Assuming the Exponential-Time Hypothesis [172], all FPT algorithms parameterized by primal treewidth must be double-exponential [171]. On the other hand, by Theorem 4.7 and Theorem 6.9 our algorithm, based on graded project-join trees, runs in time $2^{O(k')}$, where $k'$ is the primal treewidth of $\psi$ (which we call the $\{X, Y\}$*-graded primal treewidth* of $\varphi$). While $k'$ is larger than $k$, on many benchmarks $k'$ is significantly smaller than $2^k$.

Our proposed graded project-join trees can be seen as a specialization of *structure trees* [6] to the case of projected model counting. Sterns and Hunt [6] suggest constructing structure trees by manually modifying tree decomposers to consider only structure trees respecting the variable quantification order (i.e., to consider graded-

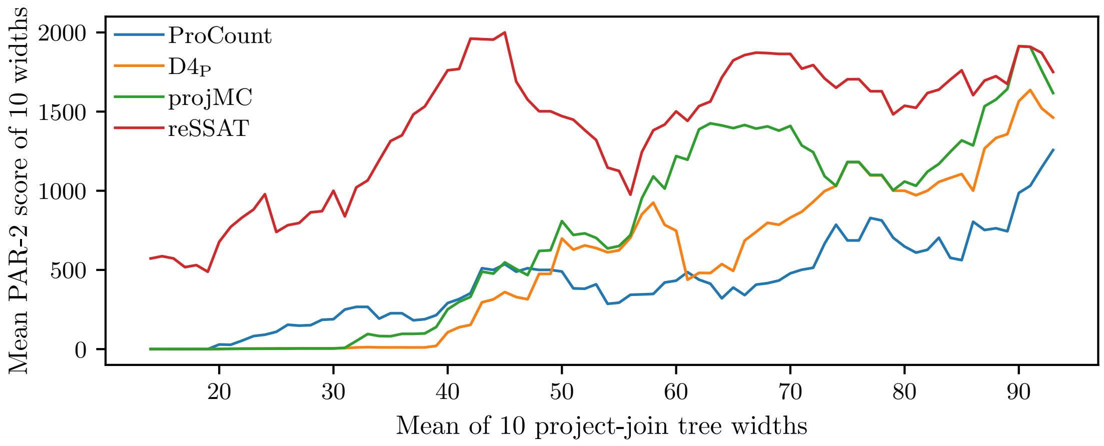


Figure 6.5 : A plot of mean PAR-2 scores (in seconds) against mean project-join tree widths. On this plot, each projected model counter (`ProCount`, `D4`$_\text{P}$, `projMC`, or `reSSAT`) is represented by a curve, on which a point $(x, y)$ indicates that $x$ is the central moving average of 10 consecutive project-join tree widths ($1 \leq w_1 < w_2 < \ldots < w_{10} \leq 99$) and $y$ is the average PAR-2 score of the benchmarks whose project-join trees have widths $w$ s.t. $w_1 \leq w \leq w_{10}$. We observe that the performance of `ProCount` degrades as the project-join tree width increases. However, `ProCount` tends to be the fastest solver on benchmarks whose graded project-join trees have widths roughly between 50 and 100.

ness directly). In this work, we take a different approach by using existing tools for standard project-join trees (in particular, tree decomposers) in a black-box way. This is crucial for the practical success of our tool, as we can leverage continual progress in tree decomposition.

Projected model counting is also a special case of the *functional aggregate query (FAQ)* problem [7]. Our graded project-join trees can be seen as a specialization of FAQ variable orders. **Theorem 7.5** of [7] gives an algorithm for constructing an FAQ variable orders from a sequence of tree decompositions, which, in the context of projected model counting, is equivalent to the technique we discussed in Section 6.2.1 of ignoring relevant variables while planning to project irrelevant variables. In contrast, our approach can potentially find lower-width graded project-join trees by incorporating relevant variables even when planning to project irrelevant variables.

It may be possible to lift this improvement to the FAQ framework in future work.

A recent framework for projected counting is `nestHDB` [95], a hybrid solver. Similar to our framework, `nestHDB` includes a planning phase (using the tree-decomposition tool `htd` [45]) and an execution phase (using the database engine `Postgres` [173] and the projected counter `projMC` [88], alongside other tools). We predict that `nestHDB` may benefit from switching the projected-counting component to `ProCount`, which was often faster than `projMC` in Experiment 3. While we were unable to run a full experimental comparison[†] with `nestHDB`, we evaluated `nestHDB` against `ProCount` on 90 benchmarks [169] (with weights removed) using a single CPU core of an Intel i7-7700HQ processor (2.80-GHz) with 30 GB of RAM. `ProCount` and `nestHDB` respectively solved 69 and 59 benchmarks, with a 100-second timeout. The mean PAR-2 scores for `ProCount` and `nestHDB` were 47 and 87. Further comparison is needed in future work.

In some sense, projected model counting on Boolean formulas is a dual problem of *maximum a posteriori hypothesis (MAP)* inference [174–176] on Bayesian networks [177]: a projected model count has the form $\sum_X \max_Y f(X, Y)$, while a MAP probability has the form $\max_Y \sum_X f(X, Y)$. Both problems can be solved using variable elimination, but an elimination order may not freely interleave $X$ variables with $Y$ variables. A valid variable order induces an *evaluation tree* (similar to a project-join tree) [178]. As mentioned in [178], exact MAP algorithms construct evaluation trees using constraint-satisfaction heuristics (similar to our planner `HTB`). Our work goes further by constructing low-width graded project-join trees using tree-decomposition techniques (with our planner `LG`) and by performing efficient computations using compact ADDs (with our executor `DMC`).

[†]`nestHDB` is an unweighted tool, but the benchmarks in Section 6.4 are weighted. Moreover, the cluster used in Section 6.4 does not support database management systems.

## 6.6 Chapter Summary

We adapted the dynamic-programming framework from Chapter 4 to perform projected model counting by requiring project-join trees to be graded. This framework decomposes projected model counting into two phases. First, the planning phase produces a graded project-join tree from a CNF formula. Second, the execution phase uses the this tree to guide the computation of the projected model count of the formula w.r.t. a literal-weight function. We proved that algorithms for building project-join trees can be used in a black-box way to build graded project-join trees.

The primary finding of this chapter is that our dynamic-programming framework of planning and execution is a competitive approach to projected model counting. We found in the planning phase that planning based on tree-decomposition tools continues to outperform planning based on constraint-satisfaction heuristics, even when considering graded project-join trees. Although requiring the project-join trees to be graded increased their width, our planning techniques were able to find low-width graded project-join trees. Overall, our planning techniques were able to find low-width graded project-join trees for a significant number of benchmarks. The resulting tool `ProCount` was competitive with the exact weighted projected model counters $\texttt{D4}_{\texttt{P}}$ [88], `projMC` [88], and `reSSAT` [89]. `ProCount` considerably improves the virtual best solver on low-width benchmarks and thus is a valuable addition to the portfolio.

# Chapter 7

# Conclusion and Future Directions

Discrete integration is a fundamental problem in artificial intelligence, with applications in probabilistic reasoning, planning, inexact computing, engineering reliability, and statistical physics [1–4]. In discrete integration, the task is to count the total weight, subject to a given weight function, of the set of solutions of input constraints [3]. Although discrete integration is theoretically difficult [10], the development of tools to compute the total weight on large industrial formulas is an area of active research.

In this dissertation, we presented a series of dynamic-programming algorithms and tools for discrete integration based on a single conceptual framework. Our framework consisted of two cleanly separated phases: a *planning phase* of high-level reasoning, followed by an *execution phase* of low-level computation. We showed how to leverage existing graph decomposition tools for the planning phase. Similarly, we showed how to leverage existing tensor libraries and ADD libraries for the execution phase. The resulting tools scale to large discrete-integration queries and run flexibly in a variety of hardware environments.

We introduced this framework in Chapter 3 with a new counter, `TensorOrder`, which used graph decompositions for planning and tensor libraries for execution. To enable the use of tensor libraries, we presented a new reduction from weighted model counting to tensor network contraction. We analyzed two planning techniques– an existing approach **LG** and a new approach **FT**– that use graph decompositions to find contraction trees of small max rank of tensor networks. `TensorOrder`, when equipped with the **FT** planner, was the best tool on discrete-integration queries with incidence graphs of small ($< 30$) treewidth.

We then applied this framework in Chapter 4 to an existing model counter, `ADDMC` [29, 30]. We unified a variety of discrete-integration approaches, including `ADDMC`, into a single conceptual framework the uses project-join trees to separate planning and execution. We showed that replacing the existing constraint-satisfaction planner in `ADDMC` with **LG** (a planner based on graph decompositions) led to a faster model counter. Moreover, we compared (dense) tensors with (sparse) algebraic decision diagrams in the execution phase and found that algebraic decision diagrams outperform tensors on single CPU cores. The resulting tool `DPMC` was the fastest solver on a significant number of benchmarks, especially on those whose primal graphs had small to medium ($< 100$) treewidth, and thus is valuable as part of a portfolio of counters.

Next, in Chapter 5 we exploited our framework to develop a parallel model counter by separately parallelizing the planning and execution phases in `TensorOrder`. We showed that the planning phase may be parallelized by using an algorithmic portfolio [79] of planners. Further, we showed that the execution phase may be parallelized through parallel tensor libraries. In order to handle limited-memory environments (e.g., on a GPU), we introduced index slicing to divide the computation of a single tensor network contraction. Although we showed that index slicing also allows for efficient tensor network contraction on a TPU theoretically, in practice we found that current tensor libraries fail to compile tensor computations with high-dimensional tensors to a TPU. The resulting tool `TensorOrder2` was the fastest parallel model counter on a significant number of benchmarks, especially those with incidence graphs of small ($< 30$) treewidth.

Finally, in Chapter 6 we applied our framework to projected model counting by extending the approach in Chapter 4. We presented a novel algorithm for performing projected model counting through a planning and execution phase using graded project-join trees. Our key theoretical contribution was a novel algorithm to construct graded project-join trees from standard project-join trees. The resulting tool `ProCount` made a significant contribution to a portfolio of exact weighted projected model counters, especially on benchmarks with small to medium ($< 100$) graded primal treewidth.

Overall, we demonstrated that a clean separation of dynamic-programming algorithms for discrete integration into planning and execution phases is fruitful. The resulting algorithms and tools for discrete integration scale to large instances and run flexibly in a variety of configurations and hardware environments.

## 7.1 Future Work

The framework of planning and execution introduced in this dissertation is just the next step towards developing practical tools for automated reasoning. We end with a list of several open directions to improve this framework and to apply this framework in broader contexts.

### 7.1.1 Improving Planning

The planning approaches in this dissertation were explicitly designed to take advantage of existing graph decomposition tools in an unmodified, black-box way. Future improvements to graph decomposition tools, itself an active area of research, may allow us to scale to larger benchmarks. Moreover, developing customized graph decomposition tools for planning in discrete integration is a promising avenue for future work.

While this dissertation evaluated a subset of graph decomposition tools for planning, one avenue for improvement of our tools is to consider other graph decomposition tools. Integrating a diverse set of graph decomposition tools would especially benefit a portfolio of planners as in Chapter 5. For example, there has been work on parallel branch decomposition tools that would be possible to integrate with our approach [179]. It may also be possible to exploit other types of graph decompositions (e.g. hypertree decompositions [180]) for planning.

The parallel portfolio of planners that we developed in Chapter 5 was functional but simplistic. It may be possible to improve the portfolio through more advanced algorithm-selection techniques [181, 182]. This portfolio planner may be useful to apply in other planning contexts, for example to projected model counting in Chapter 6. More broadly, this portfolio strategy could be useful to parallelize a variety of applications that use graph decomposition tools [33, 42–44].

### 7.1.2 Improving Execution

Similar to planning, the execution approaches in this dissertation were explicitly designed to take advantage of existing tensor and ADD libraries in an unmodified, black-box way. Future improvements to these computational libraries, which is itself an extremely active area of independent research in high-performance computing, may allow us to scale to larger benchmarks in the future. Moreover, developing computational libraries that are specifically optimized for execution in discrete integration is a promising avenue for future work. A problem of particular interest is to re-engineer the XLA compiler when targeting a TPU in order to handle the high-dimensional tensors seen in model counting. One could also consider implementing our execution algorithms using other computational libraries, e.g. using databases as in [31], using the parallel ADD library `Sylvan` [73], or using other sparse data structures [69, 183, 184].

In Chapter 5, we evaluated index slicing in the context of tensor-based execution. One avenue for future development of parallel counters is to integrate slicing with the ADD-based execution from Chapter 4 or Chapter 6. This may require more complex techniques for choosing which indices (i.e., variables) to slice, since it is difficult to estimate the memory usage of (sparse) ADDs a priori.

Moreover, we focused on contracting all tensor network slices in sequence on a single GPU. There are opportunities in future work to apply index slicing in more complex ways to leverage parallelism at a larger scale. One potential approach is to run each slice computation in Algorithm 5.1 in parallel on a separate CPU core, on a separate GPU, or even on entirely separate nodes in a computing cluster. For example, recent work in the tensor network community used index slicing to divide a single tensor network contraction across a computing cluster containing over 100k CPU cores running in parallel [152]. Such techniques would allow our framework to scale to leverage huge computational resources for difficult discrete-integration queries.

### 7.1.3 Beyond CNF Formulas

While we focused on discrete integration over CNF formulas, many of the techniques in this dissertation easily extend to discrete integration over more general classes of constraints. For example, all techniques we introduced in this work would have similar performance when applied to formulas that mix CNF clauses (i.e., OR clauses) with XOR clauses and Exactly-One clauses. The techniques from Chapter 4 and 6 require only that the discrete-integration query is presented as the conjunction of constraints, but need no restriction on the type of each constraint. In contrast, techniques for discrete integration based on search or knowledge compilation would require different reasoning techniques for each type of constraint (or requires the constraints to be encoded as a CNF formula, which may dramatically increase the size of the constraints). Evaluating our techniques on a wider collection of constraints is an exciting direction for future work that may enable new applications of discrete integration.

### 7.1.4 Beyond Discrete Integration

We focused in this dissertation on processing a single discrete-integration query in isolation. In many applications, however, several discrete-integration queries are made with an identical or similar set of constraints. Such repeated queries are handled well by counters based on knowledge compilation, where the same set of compiled constraints can be queried multiple times [21, 22, 141, 185, 186]. Exploring how our framework can be optimized to handle several queries is another promising research direction. For example, in some contexts one could reuse the same execution plan from the planning stage across multiple discrete-integration queries.

Our framework of planning and execution can be used as a guide and applied to other problems beyond discrete integration. For example, Tabajara and Vardi [165] described a dynamic-programming, decision-diagram-based framework for functional Boolean synthesis. Refactoring the algorithm into a planning phase followed by an execution phase is also of interest in that context to produce more scalable and flexible tools. Our framework could also be generalized for maximum model counting [92] and other types of functional aggregate queries (FAQs) [7], including MAP [174–176].

# Bibliography

[1] F. Bacchus, S. Dalmao, and T. Pitassi, "Algorithms and complexity results for #SAT and Bayesian inference," in *FOCS*, pp. 340–351, 2003.

[2] C. Domshlak and J. Hoffmann, "Probabilistic planning via heuristic forward search and weighted model counting," *Journal of Artificial Intelligence Research*, vol. 30, no. 1, pp. 565–620, 2007.

[3] C. P. Gomes, A. Sabharwal, and B. Selman, "Model counting," in *Handbook of Satisfiability*, pp. 633–654, IOS Press, 2009.

[4] Y. Naveh, M. Rimon, I. Jaeger, Y. Katz, M. Vinov, E. s Marcu, and G. Shurek, "Constraint-based random stimuli generation for hardware verification," *AI Magazine*, vol. 28, no. 3, pp. 13–13, 2007.

[5] S. Chakraborty, K. S. Meel, and M. Y. Vardi, "Approximate model counting," in *Handbook of Satisfiability, 2nd Edition*, IOS Press, 2021.

[6] R. E. Stearns and H. B. Hunt III, "Exploiting structure in quantified formulas," *Journal of Algorithms*, vol. 43, no. 2, pp. 220–263, 2002.

[7] M. Abo Khamis, H. Q. Ngo, and A. Rudra, "FAQ: questions asked frequently," in *PODS*, pp. 13–28, ACM, 2016.

[8] V. Belle, A. Passerini, and G. Van den Broeck, "Probabilistic inference in hybrid domains by weighted model integration," in *IJCAI*, vol. 2015, pp. 2770–2776, 2015.

[9] D. Koller and N. Friedman, *Probabilistic graphical models: principles and techniques.* MIT press, 2009.

[10] L. G. Valiant, “The complexity of enumeration and reliability problems,” *SICOMP*, vol. 8, no. 3, pp. 410–421, 1979.

[11] S. Toda, “On the computational power of PP and (+) P,” in *FOCS*, pp. 514–519, IEEE Computer Society, 1989.

[12] S. Chakraborty, D. Fried, K. S. Meel, and M. Y. Vardi, “From weighted to unweighted model counting,” in *AAAI*, pp. 689–695, 2015.

[13] J. M. Dudek, D. Fried, and K. S. Meel, “Taming discrete integration via the boon of dimensionality,” in *NeurIPS*, pp. 1071–1082, 2020.

[14] E. P. Zawadzki, A. Platzer, and G. J. Gordon, “A generalization of SAT and #SAT for robust policy evaluation,” in *IJCAI*, pp. 2583–2589, 2013.

[15] E. Birnbaum and E. L. Lozinskii, “The good old davis-putnam procedure helps counting models,” *Journal of Artificial Intelligence Research*, vol. 10, pp. 457–477, 1999.

[16] M. Davis and H. Putnam, “A computing procedure for quantification theory,” *JACM*, vol. 7, no. 3, pp. 201–215, 1960.

[17] M. Davis, G. Logemann, and D. Loveland, “A machine program for theorem-proving,” *Communications of the ACM*, vol. 5, no. 7, pp. 394–397, 1962.

[18] A. Biere, M. Heule, H. van Maaren, and T. Walsh, “Conflict-driven clause learning sat solvers,” in *Handbook of Satisfiability*, pp. 131–153, IOS Press, 2009.

[19] T. Sang, P. Beame, and H. Kautz, “Heuristics for fast exact model counting,” in *SAT*, pp. 226–240, 2005.

[20] M. Thurley, “SharpSAT: counting models with advanced component caching and implicit BCP,” in *SAT*, pp. 424–429, 2006.

[21] U. Oztok and A. Darwiche, “A top-down compiler for sentential decision diagrams.,” in *IJCAI*, pp. 3141–3148, 2015.

[22] J.-M. Lagniez and P. Marquis, “An improved decision-DNNF compiler,” in *IJCAI*, pp. 667–673, 2017.

[23] S. B. Akers, “Binary decision diagrams,” *IEEE Computer Architecture Letters*, vol. 27, no. 06, pp. 509–516, 1978.

[24] A. Darwiche, “Sdd: A new canonical representation of propositional knowledge bases,” in *IJCAI*, 2011.

[25] G. Charwat and S. Woltran, “Dynamic programming-based QBF solving.,” in *SAT*, pp. 27–40, 2016.

[26] J. K. Fichte, M. Hecher, M. Morak, and S. Woltran, “Answer set solving with bounded treewidth revisited,” in *LPNMR*, pp. 132–145, Springer, 2017.

[27] J. K. Fichte, M. Hecher, S. Woltran, and M. Zisser, “Weighted model counting on the GPU by exploiting small treewidth,” in *ESA*, pp. 28:1–28:16, 2018.

[28] J. K. Fichte, M. Hecher, and M. Zisser, “gpusat2–an improved GPU model counter,” in *CP*, pp. 491–509, 2019.

[29] J. M. Dudek, V. H. N. Phan, and M. Y. Vardi, “ADDMC: Weighted model counting with algebraic decision diagrams,” in *AAAI*, pp. 1468–1476, 2020.

[30] V. H. N. Phan, “Weighted Model Counting with Algebraic Decision Diagrams,” Master’s thesis, Rice University, 2019.

[31] J. K. Fichte, M. Hecher, P. Thier, and S. Woltran, "Exploiting database management systems and treewidth for counting," in *PADL*, pp. 151–167, Springer, 2020.

[32] R. Bellman, "Dynamic programming," *Science*, vol. 153, no. 3731, pp. 34–37, 1966.

[33] B. J. McMahan, G. Pan, P. Porter, and M. Y. Vardi, "Projection pushing revisited," in *EDBT*, pp. 441–458, 2004.

[34] T. E. Uribe and M. E. Stickel, "Ordered binary decision diagrams and the Davis-Putnam procedure," in *CCL*, pp. 34–49, Springer, 1994.

[35] A. S. M. Aguirre and M. Vardi, "Random 3-SAT and BDDs: the plot thickens further," in *CP*, pp. 121–136, Springer, 2001.

[36] G. Pan and M. Y. Vardi, "Symbolic techniques in satisfiability solving," *Journal of Automated Reasoning*, vol. 35, no. 1-3, pp. 25–50, 2005.

[37] G. Charwat and S. Woltran, "BDD-based dynamic programming on tree decompositions," tech. rep., Technische Universität Wien, Institut für Informationssysteme, 2016.

[38] R. Halin, "S-functions for graphs," *Journal of geometry*, vol. 8, no. 1-2, pp. 171–186, 1976.

[39] N. Robertson and P. D. Seymour, "Graph minors. iii. planar tree-width," *Journal of Combinatorial Theory, Series B*, vol. 36, no. 1, pp. 49–64, 1984.

[40] N. Robertson and P. D. Seymour, "Graph minors. X. Obstructions to tree-decomposition," *Journal of Combinatorial Theory, Series B*, vol. 52, no. 2, pp. 153–190, 1991.

[41] P. D. Seymour and R. Thomas, "Call routing and the ratcatcher," *Combinatorica*, vol. 14, no. 2, pp. 217–241, 1994.

[42] D. L. Applegate, R. E. Bixby, V. Chvátal, and W. J. Cook, *The traveling salesman problem*. Princeton university press, 2011.

[43] M. Cygan, F. V. Fomin, Ł. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk, and S. Saurabh, *Parameterized algorithms*. Springer, 2015.

[44] G. Greco, N. Leone, F. Scarcello, and G. Terracina, "Structural decomposition methods: Key notions and database applications," in *A Comprehensive Guide Through the Italian Database Research Over the Last 25 Years*, pp. 253–267, Springer, 2017.

[45] M. Abseher, N. Musliu, and S. Woltran, "htd- a free, open-source framework for (customized) tree decompositions and beyond," in *CPAIOR*, pp. 376–386, 2017.

[46] M. Hamann and B. Strasser, "Graph bisection with pareto optimization," *Journal of Experimental Algorithmics (JEA)*, vol. 23, no. 1, pp. 1–2, 2018.

[47] H. Tamaki, "Positive-instance driven dynamic programming for treewidth," in *ESA*, pp. 68:1–68:13, 2017.

[48] I. V. Hicks, "Branchwidth heuristics," *Congressus Numerantium*, pp. 31–50, 2002.

[49] H. Dell, C. Komusiewicz, N. Talmon, and M. Weller, "The PACE 2017 parameterized algorithms and computational experiments challenge: The second iteration," in *IPEC*, pp. 30:1–30:12, 2017.

[50] E. Fischer, J. A. Makowsky, and E. V. Ravve, "Counting truth assignments of formulas of bounded tree-width or clique-width," *Discrete Applied Mathematics*, vol. 156, no. 4, pp. 511–529, 2008.

[51] M. Samer and S. Szeider, "Algorithms for propositional model counting," *Journal of Discrete Algorithms*, vol. 8, no. 1, pp. 50–64, 2010.

[52] J. Biamonte and V. Bergholm, "Tensor networks in a nutshell," *arXiv preprint arXiv:1708.00006*, 2017.

[53] A. Cichocki, "Era of big data processing: A new approach via tensor networks and tensor decompositions," *arXiv preprint arXiv:1403.2048*, 2014.

[54] R. Orús, "Tensor networks for complex quantum systems," *Nature Reviews Physics*, vol. 1, no. 9, pp. 538–550, 2019.

[55] B. W. Bader and T. G. Kolda, "Efficient MATLAB computations with sparse and factored tensors," *SISC*, vol. 30, no. 1, pp. 205–231, 2007.

[56] S. Hirata, "Tensor contraction engine: Abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories," *The Journal of Physical Chemistry A*, vol. 107, no. 46, pp. 9887–9897, 2003.

[57] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *PACMPL*, pp. 1–29, 2017.

[58] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, "Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions," tech. rep., Facebook AI Research, 2 2018.

[59] T. E. Oliphant, *A guide to NumPy*, vol. 1. Trelgol Publishing USA, 2006.

[60] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "Tensorflow: A system for large-scale machine learning," in *OSDI*, pp. 265–283, 2016.

[61] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *NeurIPS*, pp. 8024–8035, 2019.

[62] J. Kim, A. Sukumaran-Rajam, V. Thumma, S. Krishnamoorthy, A. Panyala, L.-N. Pouchet, A. Rountev, and P. Sadayappan, "A code generator for high-performance tensor contractions on GPUs," in *CGO*, pp. 85–95, IEEE Press, 2019.

[63] T. Nelson, A. Rivera, P. Balaprakash, M. Hall, P. D. Hovland, E. Jessup, and B. Norris, "Generating efficient tensor contractions for GPUs," in *ICPP*, pp. 969–978, IEEE, 2015.

[64] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *ISCA*, pp. 1–12, 2017.

[65] J. D. Biamonte, J. Morton, and J. Turner, "Tensor network contractions for #SAT," *Journal of Statistical Physics*, vol. 160, no. 5, pp. 1389–1404, 2015.

[66] S. Kourtis, C. Chamon, E. Mucciolo, and A. Ruckenstein, "Fast counting with tensor networks," *SciPost Physics*, vol. 7, Nov 2019.

[67] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi, "Algebraic decision diagrams and their applications," *Formal methods in system design*, vol. 10, no. 2-3, pp. 171–206, 1997.

[68] S.-i. Minato, "Zero-suppressed BDDs for set manipulation in combinatorial problems," in *DAC*, pp. 272–277, 1993.

[69] S. Sanner and D. McAllester, "Affine algebraic decision diagrams and their application to structured probabilistic inference," in *IJCAI*, pp. 1384–1390, 2005.

[70] M. Kwiatkowska, G. Norman, and D. Parker, "Stochastic model checking," in *SFM*, pp. 220–270, 2007.

[71] J. Hoey, R. St-Aubin, A. Hu, and C. Boutilier, "SPUDD: stochastic planning using decision diagrams," in *UAI*, pp. 279–288, 1999.

[72] F. Somenzi, *CUDD: CU decision diagram package–release 3.0.0.* University of Colorado at Boulder, 2015.

[73] T. van Dijk and J. van de Pol, "Sylvan: multi-core decision diagrams," in *TACAS*, pp. 677–691, Springer, 2015.

[74] J. K. Fichte, M. Hecher, and F. Hamiti, "The Model Counting Competition 2020," *arXiv preprint arXiv:2012.01323*, 2020.

[75] R. Dechter, "Bucket elimination: A unifying framework for reasoning," *Artificial Intelligence*, vol. 113, no. 1-2, pp. 41–85, 1999.

[76] F. Bouquet, *Gestion de la dynamicité et énumération d'impliquants premiers: une approche fondée sur les diagrammes de décision binaire.* PhD thesis, Aix-Marseille 1, 1999.

[77] R. E. Tarjan and M. Yannakakis, "Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs," *SICOMP*, vol. 13, no. 3, pp. 566–579, 1984.

[78] A. M. Koster, H. L. Bodlaender, and S. P. Van Hoesel, "Treewidth: computational experiments," *Electron Notes Discrete Math*, vol. 8, pp. 54–57, 2001.

[79] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "SATzilla: portfolio-based algorithm selection for SAT," *JAIR*, vol. 32, pp. 565–606, 2008.

[80] J. M. Dudek, L. Dueñas-Osorio, and M. Y. Vardi, "Efficient contraction of large tensor networks for weighted model counting through graph decompositions," *arXiv preprint arXiv:1908.04381*, 2019.

[81] J. M. Dudek, V. H. N. Phan, and M. Y. Vardi, "DPMC: weighted model counting by dynamic programming on project-join trees," in *CP*, pp. 211–230, 2020.

[82] J. M. Dudek and M. Y. Vardi, "Parallel weighted model counting with tensor networks," in *MCW*, 2020.

[83] J. M. Dudek, V. H. N. Phan, and M. Y. Vardi, "ProCount: Weighted Projected Model Counting with Graded Project-Join Trees," in *SAT*, 2021.

[84] M. Jerrum and A. Sinclair, "Polynomial-time approximation algorithms for the ising model," *SICOMP*, vol. 22, no. 5, pp. 1087–1116, 1993.

[85] R. A. Aziz, G. Chu, C. Muise, and P. Stuckey, "Projected model counting," in *SAT*, pp. 121–137, 2015.

[86] V. Klebanov, N. Manthey, and C. Muise, "SAT-based analysis and quantification of information flow in programs," in *QEST*, pp. 177–192, 2013.

[87] L. Duenas-Osorio, K. S. Meel, R. Paredes, and M. Y. Vardi, "Counting-based reliability estimation for power-transmission grids," in *AAAI*, pp. 4488–4494, 2017.

[88] J.-M. Lagniez and P. Marquis, "A recursive algorithm for projected model counting," in *AAAI*, vol. 33, pp. 1536–1543, 2019.

[89] N.-Z. Lee, Y.-S. Wang, and J.-H. R. Jiang, "Solving stochastic Boolean satisfiability under random-exist quantification," in *IJCAI*, pp. 688–694, 2017.

[90] S. Sharma, S. Roy, M. Soos, and K. S. Meel, "GANAK: a scalable probabilistic exact model counter," in *IJCAI*, pp. 1169–1176, 2019.

[91] S. Ermon, C. Gomes, A. Sabharwal, and B. Selman, "Taming the curse of dimensionality: discrete integration by hashing and optimization," in *ICML*, pp. 334–342, 2013.

[92] D. J. Fremont, M. N. Rabe, and S. A. Seshia, “Maximum model counting,” in *AAAI*, pp. 3885–3892, 2017.

[93] M. Soos and K. S. Meel, “BIRD: engineering an efficient CNF-XOR SAT solver and its applications to approximate model counting,” in *AAAI*, vol. 33, pp. 1592–1599, 2019.

[94] S. Möhle and A. Biere, “Dualizing projected model counting,” in *ICTAI*, pp. 702–709, 2018.

[95] M. Hecher, P. Thier, and S. Woltran, “Taming high treewidth with abstraction, nested dynamic programming, and database technology,” in *SAT*, pp. 343–360, 2020.

[96] R. Sasak, “Comparing 17 graph parameters,” Master’s thesis, The University of Bergen, 2010.

[97] P. G. Kolaitis and M. Y. Vardi, “Conjunctive-query containment and constraint satisfaction,” *Journal of Computer and System Sciences*, vol. 61, no. 2, pp. 302–332, 2000.

[98] A. Swami and A. Gupta, “Optimization of large join queries,” in *SIGMOD*, pp. 8–17, 1988.

[99] F. R. Kschischang, B. J. Frey, and H.-A. Loeliger, “Factor graphs and the sum-product algorithm,” *IEEE Transactions on information theory*, vol. 47, no. 2, pp. 498–519, 2001.

[100] A. Darwiche and M. Hopkins, “Using recursive decomposition to construct elimination orders, jointrees, and dtrees,” in *ECSQARU*, pp. 180–191, Springer, 2001.

[101] J. D. Biamonte, S. R. Clark, and D. Jaksch, “Categorical tensor network states,” *AIP Advances*, vol. 1, no. 4, p. 042172, 2011.

[102] S. Chaudhuri and M. Y. Vardi, “Optimization of real conjunctive queries,” in *PODS*, pp. 59–70, 1993.

[103] R. Cavallo and M. Pittarelli, “The theory of probabilistic databases.,” in *VLDB*, pp. 1–4, 1987.

[104] L. Dueñas-Osorio, M. Vardi, and J. Rojo, “Quantum-inspired boolean states for bounding engineering network reliability assessment,” *Structural Safety*, vol. 75, pp. 110–118, 2018.

[105] E. Robeva and A. Seigal, “Duality of graphical models and tensor networks,” *Information and Inference: A Journal of the IMA*, vol. 8, no. 2, pp. 273–288, 2019.

[106] F. Bacchus, S. Dalmao, and T. Pitassi, “Solving #SAT and bayesian inference with backtracking search,” *Journal of Artificial Intelligence Research*, vol. 34, pp. 391–442, 2009.

[107] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, “Basic linear algebra subprograms for fortran usage,” *TOMS*, pp. 308–323, 1977.

[108] K. Fatahalian, J. Sugerman, and P. Hanrahan, “Understanding the efficiency of GPU algorithms for matrix-matrix multiplication,” in *EUROGRAPHICS*, pp. 133–137, ACM, 2004.

[109] M. Chavira and A. Darwiche, “Compiling Bayesian networks using variable elimination,” in *IJCAI*, pp. 2443–2449, 2007.

[110] V. Gogate and P. Domingos, “Approximation by Quantization,” in *UAI*, pp. 247–255, 2011.

[111] J. Hoey, R. St-Aubin, A. Hu, and C. Boutilier, “SPUDD: stochastic planning using decision diagrams,” in *UAI*, pp. 279–288, 1999.

[112] M. Kwiatkowska, G. Norman, and D. Parker, "Stochastic model checking," in *SFM*, pp. 220–270, Springer, 2007.

[113] H. Fargier, P. Marquis, A. Niveau, and N. Schmidt, "A knowledge compilation map for ordered real-valued decision diagrams," in *AAAI*, pp. 1049–1055, 2014.

[114] R. Dechter, *Constraint processing*. Morgan Kaufmann, 2003.

[115] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE TC*, vol. 100, no. 8, pp. 677–691, 1986.

[116] C. Roberts, A. Milsted, M. Ganahl, A. Zalcman, B. Fontaine, Y. Zou, J. Hidary, G. Vidal, and S. Leichenauer, "Tensornetwork: A library for physics and machine learning," *arXiv preprint arXiv:1905.01330*, 2019.

[117] R. Pfeifer, J. Haegeman, and F. Verstraete, "Faster identification of optimal contraction sequences for tensor networks," *Physical Review E*, vol. 90, no. 3, p. 033315, 2014.

[118] S.-J. Ran, E. Tirrito, C. Peng, X. Chen, L. Tagliacozzo, G. Su, and M. Lewenstein, "Lecture notes of tensor network contractions," *arXiv preprint arXiv:1708.09213*, 2019.

[119] I. L. Markov and Y. Shi, "Simulating quantum computation by contracting tensor networks," *SICOMP*, vol. 38, no. 3, pp. 963–981, 2008.

[120] M. de Oliveira Oliveira, "On the satisfiability of quantum circuits of small treewidth," in *CSR*, pp. 157–172, Springer, 2015.

[121] E. F. Dumitrescu, A. L. Fisher, T. D. Goodrich, T. S. Humble, B. D. Sullivan, and A. L. Wright, "Benchmarking treewidth as a practical component of tensor network simulations," *PloS one*, vol. 13, no. 12, p. e0207827, 2018.

[122] L. Grasedyck, “Hierarchical singular value decomposition of tensors,” *SIMAX*, vol. 31, no. 4, pp. 2029–2054, 2010.

[123] G. Evenbly and R. N. Pfeifer, “Improving the efficiency of variational tensor network algorithms,” *Physical Review B*, vol. 89, no. 24, p. 245118, 2014.

[124] A. Darwiche, “Recursive conditioning,” *Artificial Intelligence*, vol. 126, no. 1-2, pp. 5–41, 2001.

[125] P. P. Shenoy, “Binary join trees for computing marginals in the shenoy-shafer architecture,” *International Journal of approximate reasoning*, vol. 17, no. 2-3, pp. 239–263, 1997.

[126] P. P. Shenoy and G. Shafer, “Axioms for probability and belief-function propagation,” in *Classic works of the dempster-shafer theory of belief functions*, pp. 499–528, Springer, 2008.

[127] L. Ying, “Tensor network skeletonization,” *Multiscale Modeling & Simulation*, vol. 15, no. 4, pp. 1423–1447, 2017.

[128] Q.-P. Gu and H. Tamaki, “Optimal branch-decomposition of planar graphs in $O(n^3)$ time,” *TALG*, vol. 4, no. 3, p. 30, 2008.

[129] V. Dalmau, P. G. Kolaitis, and M. Y. Vardi, “Constraint satisfaction, bounded treewidth, and finite-variable logics,” in *CP*, pp. 310–326, 2002.

[130] K. Kask, R. Dechter, J. Larrosa, and A. Dechter, “Unifying tree decompositions for reasoning in graphical models,” *Artificial Intelligence*, vol. 166, no. 1-2, pp. 165–193, 2005.

[131] D. J. Harvey and D. R. Wood, “The treewidth of line graphs,” *Journal of Combinatorial Theory, Series B*, vol. 132, pp. 157–179, 2018.

[132] C. Àlvarez, R. Cases, J. Díaz, J. Petit, and M. Serna, “Communication tree problems,” *Theoretical computer science*, vol. 381, no. 1-3, pp. 197–217, 2007.

[133] C. A. Tovey, “A simplified np-complete satisfiability problem,” *Discrete applied mathematics*, vol. 8, no. 1, pp. 85–89, 1984.

[134] I. L. Markov and Y. Shi, “Constant-degree graph expansions that preserve treewidth,” *Algorithmica*, vol. 59, no. 4, pp. 461–470, 2011.

[135] M. Samer and S. Szeider, “Constraint satisfaction with bounded treewidth revisited,” *Journal of Computer and System Sciences*, vol. 76, no. 2, pp. 103–114, 2010.

[136] M. de Oliveira Oliveira, “Size-treewidth tradeoffs for circuits computing the element distinctness function,” *Theory of Computing Systems*, vol. 62, no. 1, pp. 136–161, 2018.

[137] F. Viger and M. Latapy, “Efficient and simple generation of random simple connected graphs with prescribed degree sequence,” in *COCOON*, pp. 440–449, 2005.

[138] R. Dechter and J. Pearl, “Tree clustering for constraint networks,” *AIJ*, vol. 38, no. 3, pp. 353–366, 1989.

[139] R. D. Shachter, S. K. Andersen, and P. Szolovits, “Global conditioning for probabilistic inference in belief networks,” in *UAI*, pp. 514–522, Elsevier, 1994.

[140] T. Sang, F. Bacchus, P. Beame, H. A. Kautz, and T. Pitassi, “Combining component caching and clause learning for effective model counting,” in *SAT*, vol. 4, pp. 20–28, 2004.

[141] A. Darwiche, “New advances in compiling CNF to decomposable negation normal form,” in *ECAI*, pp. 318–322, 2004.

[142] J. R. Burch, E. M. Clarke, and D. E. Long, "Symbolic model checking with partitioned transition relations," in *VLSI*, pp. 49–58, 1991.

[143] F. Bouquet, *Gestion de la dynamicité et énumération d'impliquants premiers: une approche fondée sur les Diagrammes de Décision Binaire*. PhD thesis, Aix-Marseille 1, 1999.

[144] A. Darwiche, "Dynamic jointrees," in *UAI*, pp. 97–104, 1998.

[145] M. Hamann and B. Strasser, "Graph bisection with pareto optimization," *JEA*, vol. 23, pp. 1–34, 2018.

[146] T. Sang, P. Beame, and H. A. Kautz, "Performing Bayesian inference by weighted model counting," in *AAAI*, vol. 1, pp. 475–482, AAAI Press, 2005.

[147] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded model checking using satisfiability solving," *Formal methods in system design*, vol. 19, no. 1, pp. 7–34, 2001.

[148] C. Sinz, A. Kaiser, and W. Küchlin, "Formal methods for the validation of automotive product configuration data," *AI EDAM*, vol. 17, no. 1, pp. 75–97, 2003.

[149] H. Palacios and H. Geffner, "Compiling uncertainty away in conformant planning problems with bounded width," *JAIR*, vol. 35, pp. 623–675, 2009.

[150] J. Burchard, T. Schubert, and B. Becker, "Laissez-faire caching for parallel #SAT solving," in *SAT*, pp. 46–61, Springer, 2015.

[151] T. Balyo, P. Sanders, and C. Sinz, "HordeSat: A massively parallel portfolio SAT solver," in *SAT*, pp. 156–172, Springer, 2015.

[152] J. Chen, F. Zhang, C. Huang, M. Newman, and Y. Shi, "Classical simulation of intermediate-size quantum circuits," *arXiv preprint arXiv:1805.01450*, 2018.

[153] J. Gray and S. Kourtis, "Hyper-optimized tensor network contraction," *Quantum*, vol. 5, p. 410, Mar 2021.

[154] B. Villalonga, S. Boixo, B. Nelson, C. Henze, E. Rieffel, R. Biswas, and S. Mandrà, "A flexible high-performance simulator for verifying and benchmarking quantum circuits implemented on real hardware," *NPJ Quantum Information*, vol. 5, pp. 1–16, 2019.

[155] J. Pearl, "Fusion, propagation, and structuring in belief networks," *Artificial intelligence*, vol. 29, no. 3, pp. 241–288, 1986.

[156] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, *JAX: composable transformations of Python+NumPy programs*, 2018.

[157] J.-M. Lagniez and P. Marquis, "Preprocessing for propositional model counting," in *AAAI*, pp. 2688–2694, 2014.

[158] J. Lagergren, "Efficient parallel algorithms for tree-decomposition and related problems," in *FOCS*, pp. 173–182, IEEE, 1990.

[159] B. D. Sullivan, D. Weerapurage, and C. Groër, "Parallel algorithms for graph optimization using tree decompositions," in *IPDPSW*, pp. 1838–1847, IEEE, 2013.

[160] T. C. van der Zanden and H. L. Bodlaender, "Computing Treewidth on the GPU," in *IPEC*, vol. 89, pp. 29:1–29:13, 2017.

[161] Y. Malitsky, A. Sabharwal, H. Samulowitz, and M. Sellmann, "Parallel lingeling, CCASat, and CSCH-based portfolios," *SAT Competition*, pp. 26–27, 2013.

[162] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," in *PPoPP*, pp. 73–82, 2008.

[163] "XLA: Optimizing Compiler for Machine Learning.." https://www.tensorflow.org/xla. Accessed: 2020-02-20.

[164] U. Kang, E. Papalexakis, A. Harpale, and C. Faloutsos, "Gigatensor: scaling tensor analysis up by 100 times-algorithms and discoveries," in *KDD*, pp. 316–324, 2012.

[165] L. M. Tabajara and M. Y. Vardi, "Factored Boolean functional synthesis," in *FMCAD*, pp. 124–131, IEEE, 2017.

[166] N. Wetzler, M. J. Heule, and W. A. Hunt, "DRAT-trim: Efficient checking and trimming using expressive clausal proofs," in *SAT*, pp. 422–429, Springer, 2014.

[167] J. W. Lloyd, *Foundations of logic programming*. Springer Science & Business Media, 2012.

[168] V. H. N. Phan, *Combining knowledge compilation and dynamic programming for stochastic satisfiability*. PhD thesis, Rice University, 2022. To appear.

[169] R. Gupta, S. Sharma, S. Roy, and K. S. Meel, "WAPS: weighted and projected sampling," in *TACAS*, pp. 59–76, 2019.

[170] M. Soos, K. Nohl, and C. Castelluccia, "Extending SAT solvers to cryptographic problems," in *SAT*, pp. 244–257, 2009.

[171] J. K. Fichte, M. Hecher, M. Morak, and S. Woltran, "Exploiting treewidth for projected model counting and its limits," in *SAT*, pp. 165–184, 2018.

[172] R. Impagliazzo, R. Paturi, and F. Zane, "Which problems have strongly exponential complexity?," *JCSS*, vol. 63, no. 4, pp. 512–530, 2001.

[173] M. Stonebraker and L. A. Rowe, "The design of Postgres," *ACM Sigmod Record*, vol. 15, no. 2, pp. 340–355, 1986.

[174] K. P. Murphy, *Machine learning: a probabilistic perspective*. MIT press, 2012.

[175] D. D. Maua, C. P. de Campos, and F. G. Cozman, “The complexity of MAP inference in Bayesian networks specified through logical languages,” in *IJCAI*, pp. 889–895, 2015.

[176] Y. Xue, Z. Li, S. Ermon, C. P. Gomes, and B. Selman, “Solving marginal MAP problems with NP oracles and parity constraints,” in *NeurIPS*, pp. 1127–1135, 2016.

[177] J. Pearl, “Bayesian networks: A model cf self-activated memory for evidential reasoning,” in *CogSci*, pp. 15–17, 1985.

[178] J. D. Park and A. Darwiche, “Complexity results and approximation strategies for MAP explanations,” *JAIR*, vol. 21, pp. 101–133, 2004.

[179] I. V. Hicks, *Branch decompositions and their applications.* PhD thesis, Rice University, 2000.

[180] I. Adler, G. Gottlob, and M. Grohe, “Hypertree width and related hypergraph invariants,” *European Journal of Combinatorics*, vol. 28, no. 8, pp. 2167–2181, 2007.

[181] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle, “Paramils: an automatic algorithm configuration framework,” *Journal of Artificial Intelligence Research*, vol. 36, pp. 267–306, 2009.

[182] L. Xu, F. Hutter, H. Hoos, and K. Leyton-Brown, “Evaluating component solver contributions to portfolio-based algorithm selectors,” in *SAT*, pp. 228–241, Springer, 2012.

[183] M. Baskaran, B. Meister, N. Vasilache, and R. Lethin, “Efficient and scalable computations with sparse tensors,” in *HPEC*, pp. 1–6, IEEE, 2012.

[184] J. Li, J. Sun, and R. Vuduc, “Hicoo: Hierarchical storage of sparse tensors,” in *SC*, pp. 238–252, IEEE, 2018.

[185] A. Darwiche and P. Marquis, “A knowledge compilation map,” *Journal of Artificial Intelligence Research*, vol. 17, pp. 229–264, 2002.

[186] F. Koriche, J.-M. Lagniez, P. Marquis, and S. Thomas, “Knowledge compilation for model counting: affine decision trees,” in *IJCAI*, pp. 947–953, 2013.