ON SCALABILITY OF KNOWLEDGE COMPILATION APPLICATIONS

by

SUWEI YANG
(B.Comp., National University of Singapore)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2025

Thesis Advisors:
Associate Professor Kuldeep S. Meel, Main Advisor
Professor Wee Sun Lee, Co-Advisor

Examiners:
Assistant Professor Umang Mathur
Associate Professor Roland Yap Hock Chuan

To my family, wife, friends, and myself

Acknowledgments

It has been a long PhD, much longer and tedious than I had anticipated, and I would like to thank and acknowledge the people and entities that provided support, enabling me to complete my PhD journey. I would like that thank EDB, Grab, and NUS for providing me the opportunity and funding to pursue PhD. I would also like to thank Dr. Jagannadan Varadarajan, Dr. Victor Liang Chen, Dr. Hannes Kruppa from Grab and Prof. See-Kiong Ng from Grab-NUS AI Lab for their support. During my PhD journey, there have been lots of periods of setbacks, self-doubts, challenges and learnings, throughout which I received support from family, friends, colleagues and advisor. I am grateful to everyone for the support.

I would like express sincere gratitude to Assoc. Prof. Kuldeep S. Meel, my advisor, for the guidance and teachings provided over all the years of the PhD journey. I am thankful for the consistent effort that he puts in as my advisor – spending time to meet weekly, guiding me on how to write papers and present ideas, constant feedbacks on how to improve, providing sufficient resources and help with administrative tasks so that I can focus on research. I also express sincere gratitude to Prof. Wee Sun Lee for stepping in as my school advisor to help me with the administrative tasks during my graduation process. I also thank Assistant Prof. Umang Mathur and Assoc. Prof Roland Yap Hock Chuan for their time and detailed feedbacks.

I would like to thank my labmates and research colleagues for the helpful discussions on technical topics and the company provided over the years, especially Arijit Shaw, Jiong Yang, Yong Lai, Priyanka Golia, Tim van Bremen, Anna Latour, Yash Pote, Teodora Baluta, Pang Zhanzhong, Eugene Lim and many others. I am also very thankful for the support provided by my family and friends during this PhD journey. Finally, I am would also like to thank the administrative personnels who helped me navigate through the various administrative process encountered in my PhD journey.

Contents

Summary

Knowledge compilation, a fundamental technique in computer science, involves the representation of logical formulas using directed acyclic graphs (DAGs) to allow for efficient handling of downstream computation queries. Knowledge compilation is closely related to probabilistic circuits through the addition of literal weights when handling queries, to account for probability distributions. Knowledge compilation has numerous applications spanning different areas of research, from sampling test configurations in combinatorial testing, to counting models of logical formulas, and tackling optimization problems in operational research. While the applications of knowledge compilation are typically on problems known to be theoretically hard, such as model counting for Boolean formula in conjunctive normal form being #P-hard, this has not stopped the research community from attempting to scale up approaches involving knowledge compilation in practice. Existing attempts study knowledge compilation techniques in isolation, and focus on developing more succinct knowledge compilation forms as well as better compilation heuristics. This thesis takes an orthogonal view on the scalability of knowledge compilation applications by looking at all aspects of the task, and provides a comprehensive study of various ways to more efficiently use knowledge compilation techniques in practice.

The core focus of this thesis lies in presenting a framework, termed ERA, that encompasses novel approaches to enhance the scalability of applications involving knowledge compilation techniques. The approaches range from developing relaxed encodings for problems, enabling knowledge compilation to work with more succinct logical formulas, and using the compiled representations in a more efficient manner. In particular, this thesis covers applications that employ knowledge compilation techniques to perform incremental constrained sampling for path planning and combinatorial testing, as well as model counting for pseudo-boolean formulas. The three key pillars of the ERA framework for scaling up knowledge compilation applications are a) Efficient usage of knowledge compilation forms b) Recoverable approximations whenever appropriate and c) Alternate logic forms for knowledge compilation applications. Through the demonstration of the key ideas, this thesis introduces a framework for scaling up applications involving knowledge compilation techniques.

The thesis illustrates the ERA framework via prototype implementations – INC and PBCount. INC is an incremental constrained sampler for Boolean formulas in conjunctive normal form (CNF) that reduces the number of traversals required for incremental weight updates. Typical weighted constrained samplers require two passes for each set of new weights – one annotation pass to compute joint probabilities and another pass to perform actual sampling according to joint probabilities. In contrast, INC's sampling routine combines both passes into a single pass, reducing the traversal of the compiled representation for more efficiency. PBCount makes use of knowledge compilation techniques to perform pseudo-boolean model counting, changing the typical CNF formulas that existing model counters handle for a more succinct form. In addition, the thesis also demonstrates the top-down PB counter design, via the prototype counter PBMC, and show that the design could also benefit from alternative input logic forms. In both cases, the change in input formats can reduce the number of variables involved, enabling the counters to scale to instances that time out if expressed in CNF.

List of Figures

List of Tables

List of Algorithms

Publications and Tools

List of publications that contributed to this thesis:

- [YLM22] S. Yang, V. Liang, and K. S. Meel. "INC: A Scalable Incremental Weighted Sampler". In: *Proceedings of the 22nd Conference on Formal Methods in Computer-Aided Design*. 2022
- [YLM23] S. Yang, C. Liang, and K. S. Meel. "Scalable Probabilistic Routes". In: *International Conference on Logic for Programming, Artificial Intelligence and Reasoning*. 2023
- [YM24] S. Yang and K. S. Meel. "Engineering an Exact Pseudo-Boolean Model Counter". In: *Proceedings of the 38th Annual AAAI Conference on Artificial Intelligence*. 2024
- [YM25] S. Yang and K. S. Meel. "Towards Projected and Incremental Pseudo-Boolean Model Counting". In: *Proceedings of the 39th Annual AAAI Conference on Artificial Intelligence*. 2025
- [YLM25] S. Yang, Y. Lai, and K. S. Meel. "On Top-Down Pseudo-Boolean Model Counting". In: *nternational Conference on Theory and Applications of Satisfiability Testing*. 2025

List of tools from this thesis:

- [YLM22] INC Weighted Sampler
- [YM24; YM25] PBCount bottom-up pseudo-boolean model counter
- [YLM25] PBMC top-down pseudo-boolean model counter

Chapter 1

Introduction

Knowledge compilation and decision diagrams are fundamental tools in numerous areas within the computer science community. They started as techniques for individual communities in the early days and became intertwined in part due to their complementary nature and heavily overlapping methodologies. Decision diagrams emerged from hardware verification and computer-aided design as a tool to perform numerous verification tasks [Bry86]. On the other hand, knowledge compilation techniques developed from the field of artificial intelligence as tools to address tractability in propositional reasoning, in particular, to perform tasks such as probabilistic reasoning and model counting [DM02; BDP03]. In recent years, the two fields have converged, with numerous types of decision diagrams becoming the target compilation forms for applications involving knowledge compilation techniques.

Knowledge compilation is a core technique in propositional reasoning, especially for tasks that are beyond NP-Complete - such as constrained sampling and model counting which are #P-Complete [Val79]. Knowledge compilation techniques allowed for the handling of aforementioned queries in runtime that is polynomial to the size of the target representation form, typically a type of decision diagram or negated normal form [Dar01; DM02].

However, as knowledge compilation techniques are increasingly popular and applied in a myriad of different application settings, their limitations also emerge. Whilst knowledge compilation and decision diagrams are designed to be compact representations of the assignment spaces of logical formulas and theories, the challenge is still in scalability especially when handling real-world problems which are increasingly more complex with advancements in various fields. As such, there is

continuous demand for more scalable knowledge compilation techniques and decision diagram forms, so that the increasingly complex input formulas can be handled with tractability. In this thesis, we take a journey to explore various alternative ways to scale knowledge compilation and decision diagram techniques under different application settings. In particular, we introduce the ERA framework, orthogonal to the majority of existing techniques to address scalability, for the applications involving the aforementioned techniques. Existing works mainly focus, in isolation, on either a) developing more succinct decision diagram forms as targets for knowledge compilation or b) better encoding of problem formulas and reducing the number of query calls to knowledge compilation tools. This thesis mainly explores different ways of increasing scalability by looking at both the application setup as well as the particular underlying knowledge compilation technique used and optimizing both in tandem, combining them into a comprehensive framework. To this end, the ERA framework introduces three pillars of scaling knowledge compilation techniques 1) effective usage of knowledge compilation diagrams 2) recoverable approximations, and 3) alternative logic inputs for knowledge compilation.

1.1 Efficient Usage of Knowledge Compilation Diagrams

In this pillar, this thesis explores how to efficiently use knowledge compilation diagrams. Although knowledge compilation forms enable downstream tasks and queries to be handled in polynomial runtime with respect to the size of the representation i.e. number of nodes of the target decision diagram, the compiled diagrams themselves tend to be large. As such it would be of interest to use the compiled diagrams as efficiently as possible. This thesis demonstrates efficient usage of compiled diagrams with two main ideas – a) reducing the number of traversal of diagrams when performing incremental sampling and b) reusing parts of the compilation process to handle incremental counting settings. In both settings, the aim is to reduce the number of diagram compilations required and traversing the diagrams as little as possible.

The thesis demonstrates the idea with the application of incremental weighted sampling in the context of test case generation for configurable systems [BLM20;

YLM22]. The goal would be to generate a predetermined number of test cases via sampling, for a given configurable system expressed as a propositional formula. This application scenario comprises two main components, a controller that determines the sampling distribution and updates it throughout the process, and a sampler that can handle distribution updates and performs weighted sampling accordingly. The existing work [BLM20] made use of an off-the-shelf sampler and was shown to be less able to amortize the knowledge compilation cost over multiple incremental sampling rounds. To this end, the thesis introduces the INC sampler [YLM22], designed for incremental weighted sampling. INC is able to complete a sampling round with a single pass of the underlying knowledge compilation form instead of the two passes that the existing sampler requires. Additionally, INC uses an empirically smaller knowledge compilation form which will be detailed in later chapters. The two improvements led to much better amortization of compilation cost and overall a substantially faster incremental sampling routine.

1.2 Recoverable Approximations

In this pillar, the thesis takes the perspective of jointly optimizing the computation process by looking at both the underlying knowledge compilation technique involved and the required application output. The thesis demonstrates the idea via the application of probabilistic routes, whereby the goal would be to sample a probable path, based on historical data, given start and destination. There are two main parts to the application, one being the encoding of all possible paths without loops and the other being the sampling process using knowledge compilation forms.

Existing works on propositional formula encodings of paths in a graph, more specifically in conjunctive normal form, use $O(n \log n)$ and n^2 number of variables with respect to the number of vertices in graph [Pre03]. The thesis makes the observation that it is not necessary to encode ordering information of vertices in the path, as it can be recovered given the start and destination vertices. As such, the thesis introduces an encoding that uses a linear number of variables with respect to the number of vertices in the graph, enabling better scalability of the subsequent knowledge compilation technique. More specifically, the encoding relaxes the requirement and also includes sets of vertices representing paths along

with disjoint loop components. The insight is that the disjoint loop components can also be easily removed as a post-sampling refinement step. In regards to the sampling process, existing works sample a path iteratively, according to probabilities conditioned on steps already taken [CSD17]. In contrast, this thesis introduces an approach that is able to sample an entire path at once [YLM23].

1.3 Knowledge Compilation for Alternative Logic

In this pillar, the thesis studies the impact on scalability when using alternative logic input formulas for knowledge compilation techniques. In particular, the thesis demonstrates the scalability of using knowledge compilation techniques for pseudo-boolean formulas as input instead of the typical propositional formula in conjunctive normal form. The main motivation is that pseudo-boolean (PB) formulas can express the same constraints with fewer variables compared to CNF formulas, in particular, a CNF clause can always be represented with one PB constraint while the converse is not true [Le +18]. Having fewer variables improves the overall scalability of using the knowledge compilation techniques for model counting tasks.

Under this pillar, the thesis consists of a line of work on PB model counting that introduces the PB model counter PBCount [YM24] and its extension PBCount2. The empirical evaluations demonstrate the potential of alternative logic inputs, namely that PBCount is able to outperform state-of-the-art propositional model counters by handling PB input which has substantially fewer variables. In addition, PBCount2 is introduced to support projected PB model counting, analogous to projected CNF model counting. PBCount2 also shows performance advantages over propositional model counters, lending credence to the pillar about considering alternative logic inputs. Furthermore, PBCount2 also demonstrates the combination of alternative logic and efficient diagram usage through the support for incremental PB model counting, as it reuses intermediate KC diagrams to avoid complete recompilations wherever possible.

Chapter 2

Preliminaries and Related Work

2.1 Boolean Functions and Associated Tasks

Boolean Formula A Boolean variable can take values *true* or *false*. A literal is either a Boolean variable or its negation. A Boolean formula consists of a series of Boolean literals connected by binary operators, typically $\wedge$ (conjunction) and $\vee$ (disjunction). Let F be a Boolean formula. F is in conjunctive normal form (CNF) if F is a conjunction of clauses, where each clause is a disjunction of literals.

Satisfiability and Model Counting F is satisfiable if there exists an assignment τ of variables of F such that F evaluates to *true*. We refer to τ as a satisfying assignment of F and denote the set of all τ as $\mathsf{Sol}(F)$. Model counting for Boolean formula F refers to the task of determining $|\mathsf{Sol}(F)|$, and has been shown to be #P-Complete [Val79].

Projected Model Counting Let F be a formula defined over the set of variables $\mathsf{Var}(F)$. Let X, Y be subsets of $\mathsf{Var}(F)$ such that $X \cap Y = \emptyset$ and $X \cup Y = \mathsf{Var}(F)$. The projected model count of F on X refers to the number of assignments of all variables in X such that there exists an assignment of variables in Y that makes F evaluate to *true* [Azi+15].

Constrained Sampling Constrained Sampling refers to the task of sampling elements from a distribution, typically described using a Boolean formula and a weight

function. More specifically, constrained sampling refers to the process of sampling from the space of satisfying assignments of a Boolean formula F i.e. sampling from $\mathsf{Sol}(F)$. If each satisfying assignment is sampled with uniform probability ($\frac{1}{|\mathsf{Sol}(F)|}$), we refer to the process as uniform sampling. Otherwise, we refer to constrained sampling in the presence of a weight function as weighted sampling. Jerrum, Valiant, and Vazirani showed that one could design a uniform sampler which requires only a polynomial number of queries to an exact model counter [JVV86]. In weighted sampling, the weight function W assigns a non-negative weight to each literal l of F. The weight of an assignment τ is defined as the product of the weight of its literals.

Incremental SAT Solving Another popular task in the Boolean satisfiability community is Incremental SAT solving [NR12; NRS14; FBS19; Nad22]. Incremental SAT solving involves finding satisfying assignments to a user-provided Boolean formula, typically in CNF, under certain assumptions that users can incrementally specify. Specifically, users can add and remove CNF clauses to the initially specified CNF formula. Incremental SAT solving is also useful for cases where users have to solve a large number of similar CNF formulas and has led to a wide range of applications such as combinatorial testing, circuit design, and solving string constraints [Yam+15; Yu+17; Lot+23].

Boolean Formula Preprocessing Boolean formula preprocessing involves simplifying a given formula to reduce runtimes of downstream tasks such as determining the satisfiability of the formula (SAT-solving) and model counting. Preprocessing is crucial to modern SAT solvers and model counters' performance improvements in recent decades. There are numerous preprocessing techniques introduced over the years by the research community, some of which are *unit propagation*, *bounded variable elimination*, *failed literal probing*, and *vivification* [DG84; Le 01; EB05; PHS08].

vivification refers to the technique of simplifying the clauses in a Boolean formula by removing redundant literals. *Failed literal probing* involves testing if a literal l leads to unsatisfiability, and adding $\bar{l}$ if literal l led to unsatisfiability. *Unit propagation* refers to the process of inferring variable assignments from unit clauses and propagating the assignments throughout the formula. *Bounded variable elimination*

involves removing variables in the formula using resolution and adding the resolvent clauses to the formula, with the additional restriction that the total number of clauses does not increase.

2.2 Pseudo Boolean Functions

Pseudo Boolean Formula A PB formula F consists of the conjunction of a set of one or more PB constraints, each of which is either an equality or inequality. A PB constraint takes the form $\sum_{i=1}^{n} a_i x_i \square k$ where $x_1, ..., x_n$ are Boolean literals, $a_1, ..., a_n$, and k are integers, and $\square$ is one of $\{\geq, =, \leq\}$. We refer to $a_1, ..., a_n$ as term coefficients in the PB constraint, where each term is of the form $a_i x_i$. k is known as the degree of the PB constraint. F is satisfiable if there exists an assignment τ of all variables of F such that all its PB constraints evaluate to *true*. PB model counting refers to the computation of the number of satisfying assignments of F or in other words determining $|\mathsf{Sol}(F)|$. It is worth noting that PB model counting is #P-hard, even if there is only a single constraint of a PB formula in the form of a #Knapsack problem [Dye+93].

Without loss of generality, it suffices to focus on PB constraints of the '$\geq$' form with only positive term coefficients. This is because '$\leq$' type constraints can be converted to '$\geq$' type constraints by multiplying both sides of the constraint by -1. In addition, '=' type constraints can be converted to a pair of '$\geq$' and '$\leq$' constraints. Similarly, negative term coefficients can also be manipulated to positive coefficients by negating the term literal using $\bar{x} = 1 - x$. In the context of '$\geq$' type constraints, we use the term *gap* s to denote the remaining value of degree under some variable assignment τ, that is after substituting the variable assignment values into the constraint and adjusting the value of degree accordingly. More specifically for a partial assignment τ and an arbitrary PB constraint, the gap of the constraint is given by $k - (\sum_{j:\tau(\ell_j)=1} a_j)$, where $j : \tau(\ell_j) = 1$ indicates all literals ℓ of the constraint that evaluates to true under τ. A gap value of 0 or less indicates that a PB constraint is always satisfied under τ, otherwise the constraint is yet to be satisfied. Another commonly used term for '$\geq$' PB constraints is *slack*, which indicates how close the PB constraint is to being falsified. The *slack* of a constraint

under assignment τ is defined as $(\sum_{j:\tau(\ell_j)\neq 0} a_j) - k$, where $\sum_{j:\tau(\ell_j)\neq 0} a_j$ is the sum of coefficients of all literals not falsified by τ, including unassigned literals.

Relation of Pseudo-Boolean Constraint to CNF Clause Given an arbitrary CNF clause D, one could always convert D to a PB constraint. Given that D is of the form $\bigvee_{i=1}^{m} l_i$, where $l_1, ..., l_m$ are Boolean literals, D can be represented by a single PB constraint $\sum_{i=1}^{m} a_i l_i \geq 1$ where all coefficients $a_1, ..., a_i, ...a_m$ are 1. However, there are PB constraints that require many CNF clauses to represent. An example would be $\sum_{i=1}^{m} l_i \geq k$ which requires at least k of m literals to be *true*. Existing literature established that PB constraints are exponentially more succinct than CNF clauses [Le +18]. Le Berre et al. showed that PB constraint is strictly more succinct than cardinality constraint using the example PB constraint $kx + \sum_{i=1}^{2k} x_i \geq k$, and showing that it requires cardinality constraints that are clausal, and subsequently used a result from prior work [Dix04] that showed the PB constraint required exponentially many CNF clauses to represent.

In practice, there are existing techniques to convert PB formulas to CNF formulas. One notable tool for the conversion of PB to CNF is PBLib [PS15]. PBLib implements various encodings to convert PB formulas into CNF form, some of which include cardinality networks, sorting networks, and BDD-based encodings [ES06; Abí+11; Abí+13]. In the absence of tools that natively handle tasks involving PB formulas, one could potentially perform PB to CNF conversion and employ one of the many existing CNF tools for the task.

2.3 Knowledge Compilation and Related Applications

Knowledge Compilation Knowledge compilation (KC) involves representing logical formulas as directed acyclic graphs (DAG), which are commonly referred to as knowledge compilation diagrams [DM02] or knowledge compilation forms. The goal of knowledge compilation is to allow for tractable computation of certain queries such as model counting and sampling, typically in polynomial runtime with respect to the size or number of nodes in the knowledge compilation diagram. Prior studies

also showed that knowledge compilation forms can be exponential in the number of variables of the given logical formula [CD97]. Despite the limitation, there are many well-studied forms of knowledge compilation diagrams such as d-DNNF, SDD, BDD, ZDD, OBDD, AOBDD, and the likes [Lee59; Bry86; Min93; Dar01; Dar02; MDM08; Dar11]. In this thesis, we study applications involving Reduced Ordered Binary Decision Diagram [∧] (OBDD[∧]) [LLY17] and Algebraic Decision Diagram (ADD) [Bah+93].

OBDD[∧] Lee [Lee59] introduced Binary Decision Diagram (BDD) as a way to represent Shannon expansion [Boo54]. Bryant introduced fixed variable orderings to BDDs (known as OBDD) [Bry86] for canonical representation and compression of BDDs via shared sub-graphs. Lai et al. [LLY17] introduced conjunction nodes to OBDDs (known as OBDD[∧]) [LLY17] to further reduce the size of the resultant DAG to represent a given Boolean formula.

PROB In the subsequent chapters of this thesis, we introduce a probabilistic variant of OBDD[∧] to perform sampling. We term the variant Probabilistic OBDD[∧], or PROB for simplicity. PROB is a DAG composed of four types of nodes - *conjunction*, *decision*, *true* and *false* nodes. The internal nodes of a PROB consist of conjunction and decision nodes whereas the leaf nodes of the PROB consist of true and false nodes. A PROB is recursively made up of sub-PROBs that represent sub-formulas of Boolean formula F. We use $\mathsf{VarSet}(n)$ to refer to the set of variables of F represented by a PROB with n as the root node. $\mathsf{Subdiagram}(n)$ refers to the sub-PROB starting at node n and $\mathsf{Parent}(n)$ refers to the immediate parent of node n in PROB. The PROB structure differs from OBDD[∧] by including additional parameters on the edges of decision nodes, allowing for the representation of non-uniform probability distributions of the represented assignment space. We introduce the PROB structure in greater detail in Chapter 3.2.

Algebraic Decision Diagram An algebraic decision diagram (ADD), also known as Multi-Terminal Binary Decision Diagram (MTBDD), is a directed acyclic graph representation of a function $f : 2^X \to S$ where X is the set of Boolean variables that f is defined over, and S is an arbitrary set known as the carrier set. We denote

the function represented by an ADD ψ as $\mathsf{Func}(\psi)$. The internal nodes of ADD represent decisions on variables $x \in X$ and the leaf nodes represent $s \in S$. As an example, an ADD representing $3x_1 + 4x_2$ is shown in Figure 2.1. In the figure, a dotted arrow from an internal node represents when the corresponding variable is set to *false* and a solid arrow represents when it is set to *true*.

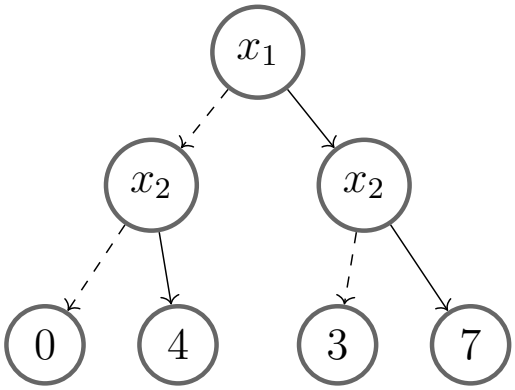


Figure 2.1: An ADD representing $3x_1 + 4x_2$ with variable ordering x_1, x_2

d-DNNF The deterministic decomposable negated normal form (d-DNNF) belongs to the negated normal form (NNF) family of representations [DM02], with determinism and decomposable properties. A negated normal form is a rooted directed acyclic graph (DAG) of a Boolean formula F, where the leaf nodes are $\top$, $\bot$, x or $\bar{x}$, where $x \in \mathsf{Var}(F)$. The internal nodes of NNF represent either $\wedge$ or $\vee$, overall forming the representation of F.

Knowledge Compilation Properties Knowledge compilation diagrams are commonly employed in tasks such as model counting and constrained sampling, as these task queries can be handled with polynomial runtime with respect to the size of the knowledge compilation diagram used. Some of the key properties of knowledge compilation diagrams that are responsible for the efficient handling of aforementioned queries are *determinism*, *decomposability*, and *smoothness*.

- **Determinism** A knowledge compilation diagram has the determinism property if its internal nodes representing disjunctions are logically disjoint. That is to say for any internal disjunction nodes, the assignment spaces represented by each of its child branches are non-overlapping.

- **Decomposability** A knowledge compilation diagram has the decomposability property if its internal nodes representing conjunctions are decomposable. That is, the variables appearing in each of the child branches of an internal conjunction node are mutually disjoint.

- **Smoothness** A knowledge compilation diagram is smooth if the variables appearing in child branches of internal disjunction nodes are exactly the same.

2.3.1 Existing Model Counting Approaches

Existing approaches to model counting can typically be classified under one of two paradigms – a) search-based or b) decision-diagram based.

Search-Based Model Counters Among the numerous existing CNF model counters, we can classify them into two main categories – search-based model counters and decision diagram-based model counters. Notable existing search-based model counters include GPMC, Ganak, and SharpSAT-TD [RS17; Sha+19; KJ21]. Search-based model counters work by setting values to variables in a given formula in an iterative manner, which is equivalent to implicitly exploring a search tree. In addition, search-based model counters adapt techniques such as sub-component caching from SAT solving for more efficient computation. Search-based model counters are often related to knowledge compilation in that the implicit search tree explored can be output as a target language in knowledge compilation, thus the implication would be that search-based model counters can be top-down knowledge compilers. Examples of such counters include GPMC and D4 [RS17; LM17]. In the subsequent chapter of this work, we introduce the first search-based exact PB model counter PBMC [YLM25].

Decision Diagram-Based Model Counter Decision diagram-based model counters employ knowledge compilation techniques to compile a given formula into directed acyclic graphs (DAGs) and perform model counting with these DAGs. Some of the recent decision diagram-based model counters are D4, ExactMC, ADDMC, and its related variant DPMC [LM17; DPV20a; DPV20b; LMY21]. D4 and ExactMC are decision diagram-based as well as search-based counters. They compile the

formula in a top-down manner into the respective decision diagram forms, making use of techniques adapted from search-based CDCL solvers. In contrast, ADDMC and DPMC (decision diagram mode) perform bottom-up compilations of algebraic decision diagrams (ADDs). In subsequent chapters of this thesis, we introduce decision diagram-based PB model counters PBCount and PBCount2.

2.3.2 Existing Sampling Approaches

KUS [Sha+18] utilizes knowledge compilation techniques, specifically Deterministic Decomposable Negation Normal Form (d-DNNF) [Dar02], to perform uniform sampling in 2 passes of the d-DNNF. *Annotation* is performed in the first pass, followed by sampling. WAPS [Gup+19] improves upon KUS by enabling weighted sampling via parameterization of the d-DNNF. WAPS performs sampling in a similar manner to KUS, the main difference being that the *annotation* step in WAPS takes into account the provided weight function.

Existing work such as the Baital framework [BLM20] for test case generation employed WAPS in settings where the weight function required updates as follows. The sampling space is first expressed as satisfying assignments of a Boolean formula, which is then compiled into the respective knowledge compilation form. In the following step, samples are drawn according to the given weight function W. Subsequently, the weights are updated depending on application logic, and weighted sampling is performed again. The process is repeated until an application-specific stopping criterion is met.

Chapter 3

Effective Usage of Knowledge Compilation Diagrams

3.1 Motivation

In this chapter, we explore the aspect of using knowledge compilation diagrams effectively through the classic knowledge compilation application of weighted sampling, with incremental weight updates. Given a Boolean formula F and weight function W, weighted sampling involves sampling from the set of satisfying assignments of F according to the distribution defined by W. Weighted sampling is a fundamental problem in many fields such as computer science, mathematics, and physics, with numerous applications. In particular, constrained-random simulation forms the bedrock of modern hardware and software verification efforts [KK07].

Sampling techniques are fundamental building blocks, and there has been sustained interest in the development of sampling tools and techniques. Recent years witnessed the introduction of numerous sampling tools and techniques, from approximate sampling techniques to uniform samplers SPUR and KUS, and weighted sampler WAPS [JS96; SSL15; AHT18; Sha+18; Gup+19]. Sampling tools and techniques have seen continuous adoption in many applications and settings [Nav+07; Goo+14; KW19; BLM20; Peh+20; Bal+21]. The scalability of a sampler is a consideration that directly affects its adoption rate. Therefore, improving scalability continues to be a key objective for the community focused on developing samplers.

The tight integration of sampling routines in various applications has highlighted the importance for samplers to handle incremental weight updates over multiple

sampling rounds, also known as incremental weighted sampling. Existing efforts on improving scalability typically focus on single round weighted sampling, and might have overlooked the incremental setting. In particular, existing approaches involving incremental weighted sampling typically employ off-the-shelf weighted samplers which could lead to less than ideal incremental sampling performance.

In this chapter, we demonstrate the effective usage of knowledge compilation diagrams via a prototype scalable weighted sampler INC. INC is designed from the ground up to address scalability issues in incremental weighted sampling settings, in particular drop-in replacement sampler for the Baital framework mentioned in Chapter 2. The core architecture of INC is based on the knowledge compilation (KC) paradigm, which seeks to succinctly represent all satisfying assignments of a Boolean formula with a directed acyclic graph (DAG) [DM02]. In the design of INC, we make two core decisions that are responsible for outperforming the current state-of-the-art weighted sampler. Firstly, INC employs PROB (Probabilistic OBDD[$\wedge$]) knowledge compilation form, based on OBDD[$\wedge$] [LLY17], that is substantially smaller in practice than the KC diagram, deterministic decomposable negation normal form (d-DNNF), used in the prior state-of-the-art approaches. Secondly, INC is designed to perform *annotation*, which refers to the computation of joint probabilities, in log-space to avoid the slower alternative of using arbitrary precision math computations.

Given a Boolean formula F and weight function W, INC compiles and stores the compiled PROB in the first round of sampling. The weight updates for subsequent incremental sampling rounds are processed without recompilation, amortizing the compilation cost. Furthermore, for each sampling round, INC simultaneously performs *annotation* and sampling in a single bottom-up pass of the PROB, achieving speedup over existing approaches. We observed that INC is significantly faster than the existing state-of-the-art in the incremental sampling routine. In our empirical evaluations, INC achieved a median of 1.69$\times$ runtime improvement over the state-of-the-art weighted sampler, WAPS [Gup+19]. Additional performance breakdown analysis supports our design choices in the development of INC. In particular, PROB is on median 4.64$\times$ smaller than the KC diagram used by the competing approach, and log-space *annotation* computations are on median 1.12$\times$ faster than arbitrary precision computations. Furthermore, INC demonstrated significantly better han-

dling of incremental sampling rounds, with incremental sampling rounds to be on median 5.9% of the initial round, compared to 67.6% for WAPS.

3.2 PROB: - Probabilistic OBDD[∧]

In this section, we introduce the structure of PROB, a probabilistic variant of OBDD[∧], in greater detail as mentioned in Chapter 2. To recap, PROB is a DAG composed of four types of nodes - *conjunction*, *decision*, *true* and *false* nodes. A PROB is recursively made up of sub-PROBs that represent sub-formulas of Boolean formula F. We use $\mathsf{VarSet}(n)$ to refer to the set of variables of F represented by a PROB with n as the root node. $\mathsf{Subdiagram}(n)$ refers to the sub-PROB starting at node n and $\mathsf{Parent}(n)$ refers to the immediate parent of node n in PROB. A PROB representation can be created from an OBDD[∧] by augmenting the latter with additional parameters which we detailed in Section 3.2.2. Additionally, the compilation of Boolean formula to OBDD[∧] can be achieved using existing off-the-shelf compiler in KCBox [LMY25].

3.2.1 PROB Structure

Conjunction node (∧-node) A ∧-node n_c represents conjunctions in the assignment space. There are no limits to the number of child nodes that n_c can have. However, the set of variables ($\mathsf{VarSet}(\cdot)$) of each child node of n_c must be disjoint. An example of a ∧-node would be $n2$ in Figure 3.1. Notice that $\mathsf{VarSet}(n4) = \{z\}$ and $\mathsf{VarSet}(n5) = \{y\}$ are disjoint.

Decision node A decision node n_d represents decisions on the associated Boolean variable $\mathsf{Var}(n_d)$ in Boolean formula F that the PROB represents. A decision node can have exactly two children - *lo-child* ($\mathsf{Lo}(n_d)$) and *hi-child* ($\mathsf{Hi}(n_d)$). $\mathsf{Lo}(n_d)$ represents the assignment space when $\mathsf{Var}(n_d)$ is set to *false* and $\mathsf{Hi}(n_d)$ represents otherwise. $\theta_{n_{d_{hi}}}$ and $\theta_{n_{d_{lo}}}$ refer to the parameters associated with the edge connecting decision node n_d with $\mathsf{Hi}(n_d)$ and $\mathsf{Lo}(n_d)$ respectively in a PROB. Node $n1$ in Figure 3.1 is a decision node with $\mathsf{Var}(n1) = x$, $\mathsf{Hi}(n1) = n3$ and $\mathsf{Lo}(n1) = n2$.

True and False nodes True ($\top$) and false ($\bot$) nodes are leaf nodes in a PROB. Let τ be an assignment of all variables of Boolean formula F and let PROB ψ represent F. τ corresponds to a traversal of ψ from the root node to leaf nodes. The traversal follows τ at every decision node and visits all child nodes of every conjunction node encountered along the way. τ is a satisfying assignment if all parts of the traversal eventually lead to the true node. τ is not a satisfying assignment if any part of the traversal leads to the false node. With reference to Figure 3.1, let $\tau_1 = \{x, y, \neg z\}$ and $\tau_2 = \{x, y, z\}$. For τ_1, the traversal would visit $n1, n3, n6, n7, n9$, and τ_1 is a satisfying assignment since the traversal always leads to $\top$ node ($n9$). As a counter-example, τ_2 is not a satisfying assignment with its corresponding traversal visiting $n1, n3, n6, n7, n8, n9$. τ_2 traversal visits $\bot$ node ($n8$) because variable $z \mapsto \mathit{true}$ in τ_2 and $\mathsf{Hi}(n6)$ is node $n8$.

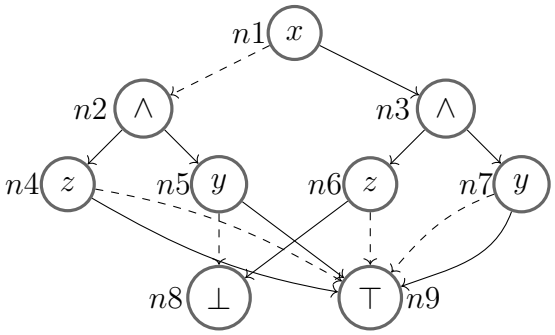


Figure 3.1: A smooth PROB ψ_1 with 9 nodes, $n1, ..., n9$, representing $F = (x \vee y) \wedge (\neg x \vee \neg z)$. Branch parameters are omitted

3.2.2 PROB Parameters

In the PROB structure, each decision node n_d has two parameters $\theta_{\mathsf{Lo}(n_d)}$ and $\theta_{\mathsf{Hi}(n_d)}$, associated with the two branches of n_d, which sums up to 1. $\theta_{\mathsf{Lo}(n_d)}$ is the normalized weight of the literal $\neg\mathsf{Var}(n_d)$ and similarly, $\theta_{\mathsf{Hi}(n_d)}$ is that of the literal $\mathsf{Var}(n_d)$. One can view $\theta_{\mathsf{Lo}(n_d)}$ to be the probability of picking $\neg\mathsf{Var}(n_d)$ and $\theta_{\mathsf{Hi}(n_d)}$ to be that of picking $\mathsf{Var}(n_d)$ by the *determinism* property introduced later. Let x_i be $\mathsf{Var}(n_d)$. Given a weight function W:

$$\theta_{\mathsf{Lo}(n_d)} = \frac{W(\neg x_i)}{W(\neg x_i) + W(x_i)} \qquad \theta_{\mathsf{Hi}(n_d)} = \frac{W(x_i)}{W(\neg x_i) + W(x_i)}$$

3.2.3 PROB Properties

The PROB structure has important properties such as *determinism* and *decomposability*. In addition to the *determinism* and *decomposability* properties, we ensure that PROBs used in this work have the *smoothness* property through a smoothing process (Algorithm 3.1).

Property 1 (Determinism)**.** *For every decision node n_d, the set of satisfying assignments represented by* $\mathsf{Hi}(n_d)$ *and* $\mathsf{Lo}(n_d)$ *are logically disjoint.*

Property 2 (Decomposability)**.** *For every conjunction node n_c,* $\mathsf{VarSet}(c_i) \cap \mathsf{VarSet}(c_j) = \emptyset$ *for all c_i and c_j where $c_i, c_j \in \mathsf{Child}(n_c)$ and $c_i \neq c_j$.*

Property 3 (Smoothness)**.** *For every decision node n_d,* $\mathsf{VarSet}(\mathsf{Hi}(n_d)) = \mathsf{VarSet}(\mathsf{Lo}(n_d))$.

3.2.4 Joint Probability Calculation with PROB

In Section 3.2.2, we mention that one can view the branch parameters as the probability of choosing between the positive and negative literal of a decision node. Notice that because of the *decomposability* and *determinism* properties of PROB, it is straightforward to calculate the joint probabilities at every given node. At each conjunction node n_c, since the variable sets of the child nodes of n_c are disjoint by *decomposability*, the joint probability of n_c is simply the product of joint probabilities of each child node. At each decision node n_d, there are only two possible outcomes on $\mathsf{Var}(n_d)$ - positive literal $\mathsf{Var}(n_d)$ or negative literal $\neg\mathsf{Var}(n_d)$. By *determinism* property, the joint probability is the sum of the two possible scenarios. Formally, the calculations for joint probabilities P' at each node in PROB are as follows:

$$P' \text{ of } \wedge\text{-node } n_c = \prod_{c \in \mathsf{Child}(n_c)} P'(c) \tag{EQ1}$$

$$\begin{aligned} P' \text{ of decision-node } n_d = {} & \theta_{\mathsf{Lo}(n_d)} \times P'(\mathsf{Lo}(n_d)) \\ & + \theta_{\mathsf{Hi}(n_d)} \times P'(\mathsf{Hi}(n_d)) \end{aligned} \tag{EQ2}$$

For true node n, $P'(n) = 1$ because it represents satisfying assignments when reached. In contrast $P'(n) = 0$ when n is a *false* node as it represents non-satisfying

assignments. In Proposition 2, we show that weighted sampling is equivalent to sampling according to joint probabilities of satisfying assignments of a PROB.

3.3 INC - Sampling from PROB

In this section, we introduce INC - a bottom-up algorithm for weighted sampling on PROB. We first describe INC for drawing one sample and subsequently describe how to extend INC to draw k samples at once. We also provide proof of correctness that INC is indeed performing weighted sampling. As a side note, samples are drawn with replacement, in line with the existing state-of-the-art weighted sampler [Gup+19].

3.3.1 Preprocessing PROB

In the main sampling algorithm (Algorithm 3.2) to be introduced later in this section, the input is a smooth PROB. As a preprocessing step, we introduce Smooth algorithm that takes in a PROB ψ and performs smoothing.

The Smooth algorithm processes the nodes in the input PROB ψ in a bottom-up manner while keeping track of VarSet(n) for every node n in ψ using a map κ. *True* and *false* nodes have $\emptyset$ as they are leaf nodes and do not represent any variables. At each conjunction node, its variable set is the union of variable sets of its child nodes.

The smoothing happens at decision node n in ψ when VarSet(Lo(n)) and VarSet(Hi(n)) do not contain the same set of variables as shown by lines 8 and 16 of Algorithm 3.1. In the smoothing process, a new conjunction node (*lcNode* for Lo(n) and *rcNode* for Hi(n)) is created to replace the corresponding child of n, with the original child node now set as a child of the conjunction node. Additionally, for each of the missing variables v, a decision node representing v is created and added as a child of the respective conjunction node. The decision nodes created during smoothing have both their lo-child and hi-child set to the *true* node. To reduce memory footprint, we check if there exists the same decision node before creating it in the checkMakeTrueDecisionNode function.

As an example, we refer to ψ_2 in Figure 3.2. It is obvious that ψ_2 is not smooth, because VarSet(Lo($n1$)) = $\{y\}$ and VarSet(Hi($n1$)) = $\{z\}$. In the smoothing process, we replace Lo($n1$) with a new conjunction node $n2$ and add a decision node $n4$

Algorithm 3.1: Smooth - returns a smoothed PROB

Input: PROB ψ
Output: smooth PROB

```
1:  κ ← initMap();
2:  for node n of ψ in bottom-up order do
3:      if n is true node or false node then
4:          κ[n] ← ∅;
5:      else if n is ∧-node then
6:          κ[n] ← unionVarSet(Child(n), κ);
7:      else
8:          if κ[Hi(n)] − κ[Lo(n)] ≠ ∅ then
9:              lset ← κ[Hi(n)] − κ[Lo(n)];
10:             lcNode ← new∧-node();
11:             lcNode.addChild(Lo(n));
12:             for var v in lset do
13:                 dNode ← checkMakeTrueDecisionNode(v) ;
14:                 lcNode.addChild(dNode);
15:             Lo(n) ←lcNode;
16:         if κ[Lo(n)] − κ[Hi(n)] ≠ ∅ then
17:             rset ← κ[Lo(n)] − κ[Hi(n)];
18:             rcNode ← new∧-node();
19:             rcNode.addChild(Hi(n));
20:             for var v in rset do
21:                 dNode ← checkMakeTrueDecisionNode(v) ;
22:                 rcNode.addChild(dNode);
23:             Hi(n) ←rcNode;
24:         κ[n] ← Var(n) ∪ unionVarSet({Hi(n), Lo(n)});
25: return ψ
```

representing missing variable z, with both child set to *true* node $n9$. We repeat the steps for Hi($n1$) to arrive at PROB ψ_1 in Figure 3.1.

3.3.2 Sampling Algorithm

INC takes a PROB ψ representing Boolean formula F and draws a sample from the space of satisfying assignments of F, the process is illustrated by Algorithm 3.2. INC performs sampling in a bottom-up manner while integrating the *annotation* process in the same bottom-up pass. Since we want to sample from the space of satisfying assignments we can ignore *false* nodes in ψ entirely by considering a sub-DAG that excludes *false* nodes and edges leading to them, as shown by line 3. As

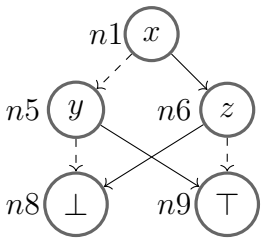


Figure 3.2: A PROB ψ_2 representing Boolean formula $F = (x \vee y) \wedge (\neg x \vee \neg z)$, branch parameters are omitted

Algorithm 3.2: INC - returns a satisfying assignment based on PROB ψ parameters

Input: smooth PROB ψ
Output: a sampled satisfying assignment

1: cache $\zeta \leftarrow$ initCache() ; ▷ Algorithm uses 2 caches, ζ and φ
2: joint prob cache $\varphi \leftarrow$ initCache();
3: $\psi' \leftarrow$ hideFalseNode(ψ);
4: **for** node n of ψ' in bottom-up order **do**
5: | **if** n is *true* node **then**
6: | | $\zeta[n] \leftarrow \emptyset$;
7: | | $\varphi[n] \leftarrow 1$;
8: | **else if** n is $\wedge$-node **then**
9: | | $\zeta[n] \leftarrow$ unionChild(Child(n), ζ);
10: | | $\varphi[n] \leftarrow \prod_{c \in \mathsf{Child}(n)} \varphi[c]$;
11: | **else**
12: | | $p_{lo} \leftarrow \theta_{\mathsf{Lo}(n)} \times \varphi[\mathsf{Lo}(n)]$;
13: | | $p_{hi} \leftarrow \theta_{\mathsf{Hi}(n)} \times \varphi[\mathsf{Hi}(n)]$;
14: | | $p_{joint} \leftarrow p_{lo} + p_{hi}$;
15: | | $\varphi[n] \leftarrow p_{joint}$;
16: | | $r \leftarrow x \sim \mathsf{binomial}(1, \frac{p_{hi}}{p_{joint}})$;
17: | | **if** r is 1 **then**
18: | | | $\zeta[n] \leftarrow \zeta[\mathsf{Hi}(n)] \cup \mathsf{Var}(n)$;
19: | | **else**
20: | | | $\zeta[n] \leftarrow \zeta[\mathsf{Lo}(n)] \cup \neg\mathsf{Var}(n)$;
21: **return** ζ[rootnode(ψ)]

an example, hideFalseNode when applied to ψ_1 in Figure 3.1 would remove node $n8$ and the edges immediately leading to it. Next, INC processes each of the remaining nodes in bottom-up order while keeping two caches - ζ to store the partial samples from each node, φ to store the joint probability at each node. INC starts with $\emptyset$ at the *true* node since there is no associated variable.

At each conjunction node, INC takes the union of the child nodes in line 9. Using

$n2$ in Figure 3.1 as an example, if sample drawn at $n4$ is $\zeta[n4] = \{\neg z\}$ and at $n5$ is $\zeta[n5] = \{y\}$, then $\mathsf{unionChild}(\mathsf{Child}(n2), \zeta) = \{y, \neg z\}$. At each decision node n, a decision on $\mathsf{Var}(n)$ is sampled from lines 16 to 20. We first calculate the joint probabilities, p_{lo} and p_{hi} of choosing $\neg\mathsf{Var}(n)$ and choosing $\mathsf{Var}(n)$. Subsequently, we sample decision on $\mathsf{Var}(n)$ using a binomial distribution in line 16 with the probability of success being the joint probability of choosing $\mathsf{Var}(n)$. After processing all nodes, the sampled assignment is the output at root node of ψ. In this work, we implemented the computations in lines 10 and 12 to 14 of Algorithm 3.2 in log space, which we detail in Section 3.3.3.

Extending INC to k samples It is straightforward to extend the single sample INC shown in Algorithm 3.2 to draw k samples in a single pass, where k is a user-specified number. At each node, we have to store a list of k independent copies of partial assignments drawn in ζ. At each conjunction node n_c, we perform the same union process in line 9 of Algorithm 3.2 for child outputs in the same indices of the respective lists in ζ. More specifically, if n_c has child nodes c_x and c_y, the outputs of index i are combined to get the output of n_c at index i. This process is performed for all indices from 1 to k. At each decision node n_d, we now draw k independent samples instead of a single sample from the binomial distribution as shown in line 16. The sampling step in lines 16 to 20 are performed independently for the k random numbers. There is no change necessary for the calculation of joint probabilities in Algorithm 3.2 as there is no change in literal weights.

Incremental sampling Given a Boolean formula F and weight function W, INC performs incremental sampling with the sampling process shown in Figure 3.3. In the initial round, INC compiles F and W into a PROB ψ and performs sampling. Subsequent rounds involve applying a new set of weights W to ψ, typically generated based on existing samples by the controller [BLM20], and performing weighted sampling according to the updated weights. The number of sampling rounds is determined by the controller component, whose logic varies according to application.

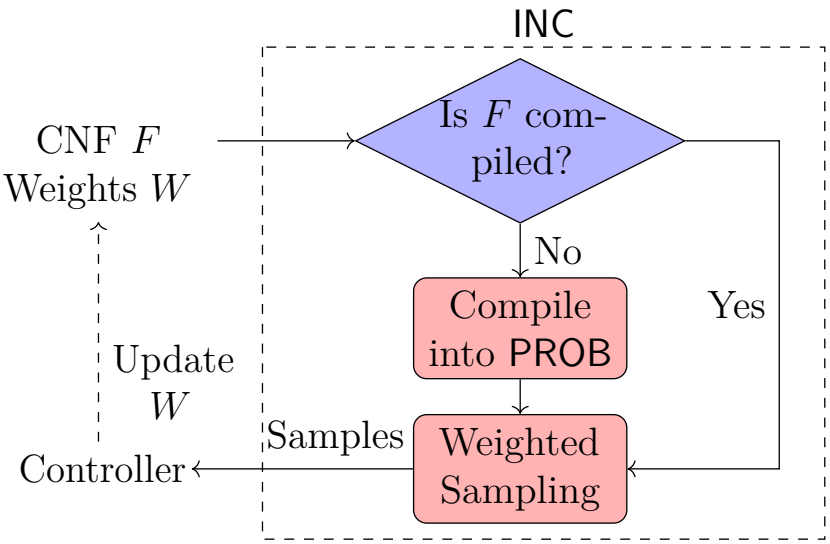


Figure 3.3: INC's incremental sampling flow

3.3.3 Implementation Decisions

Log-Space Calculations INC performs *annotation* process - computation of joint probabilities in log space. This design choice is made to avoid the usage of arbitrary precision math libraries, which WAPS utilized to prevent numerical underflow after many successive multiplications of probability values. Using the LogSumExp trick below, it is possible to avoid numerical underflow.

$$\begin{aligned}\log(a+b) &= \log(a) + \log(1+\frac{b}{a}) \\ &= \log(a) + \log(1+\exp(\log(b)-\log(a)))\end{aligned}$$

The joint probability at a decision node n_d is given by $\theta_{\mathsf{Lo}(n_d)} \times$ joint probability of $\mathsf{Lo}(n_d) + \theta_{\mathsf{Hi}(n_d)} \times$ joint probability of $\mathsf{Hi}(n_d)$. Notice that if we were to perform the calculation in log space, we would have to add the two weighted log joint probabilities, termed p_{lo} and p_{hi} in Algorithm 3.2. Using the LogSumExp trick, we do not need to exponentiate p_{lo} and p_{hi} independently which risks running into numerical underflow. Instead, we only need to exponentiate the difference of p_{lo} and p_{hi} which is more numerically stable. Equations EQ1 and EQ2 can be implemented in log space as

follows:

$$Q \text{ of } \wedge\text{-node } n_c = \sum_{c \in \mathsf{Child}(n_c)} Q(c)$$

$$\begin{aligned} Q \text{ of decision-node } n_d = \mathsf{LogSumExp}[& \\ & \log(\theta_{\mathsf{Lo}(n_d)}) + Q(\mathsf{Lo}(n_d)), \\ & \log(\theta_{\mathsf{Hi}(n_d)}) + Q(\mathsf{Hi}(n_d))] \end{aligned}$$

In the equations above, Q refers to the corresponding log joint probabilities in EQ1 and EQ2. In the experiments section, we detail the runtime advantages of using log computations compared to arbitrary precision math computations.

Dynamic Annotation In existing state-of-the-art weighted sampler WAPS, sampling is performed in two passes - the first pass performs *annotation* and the second pass samples assignments according to the joint probabilities. In INC, we combine the two passes into a single bottom-up pass performing *annotation* dynamically while sampling at each node.

3.3.4 Theoretical Analysis

Proposition 1. *Branch parameters of any decision node n_d are correct sampling probabilities, i.e. $W(x_i) : W(\neg x_i) = \theta_{\mathsf{Hi}(x_i)} : \theta_{\mathsf{Lo}(x_i)}$ where $\mathsf{Var}(n_d) = x_i$.*

Proof.

$$\frac{W(x_i)}{W(\neg x_i)} = \frac{\frac{W(x_i)}{W(x_i)+W(\neg x_i)}}{\frac{W(\neg x_i)}{W(x_i)+W(\neg x_i)}} = \frac{\theta_{\mathsf{Hi}(x_i)}}{\theta_{\mathsf{Lo}(x_i)}}$$

We start with the ratio of literal weights of x, multiply both numerator and denominator by $W(x_i) + W(\neg x_i)$ and arrive at the ratio of branch parameters of n_d. Notice that only the ratio matters for sampling correctness and not the absolute value of weights. □

Remark 1. *Let n_d be an arbitrary decision node in* PROB ψ. *When performing sampling according to a weight function W, $\theta_{\mathsf{Lo}(n_d)}$ is the probability of picking $\neg\mathsf{Var}(n_d)$ and $\theta_{\mathsf{Hi}(n_d)}$ is that of $\mathsf{Var}(n_d)$. The determinism property states that the choice of either literal is disjoint at each decision node.*

Proposition 2. INC *samples an assignment* τ *from* PROB ψ *with probability* $\frac{1}{N}\prod_{l\in\tau} W(l)$*, where* N *is a normalization factor.*

Proof. The proof consists of two parts, one for $\wedge$-node and another for decision node.

$\wedge$-node Let n_c be an arbitrary conjunction node in PROB ψ. Recall that by decomposability property, $\forall c_i, c_j \in \mathsf{Child}(n_c)$ and $c_i \neq c_j$, $\mathsf{VarSet}(c_i) \cap \mathsf{VarSet}(c_j) = \emptyset$. As such an arbitrary variable $x_i \in \mathsf{VarSet}(n_c)$ only belongs to the variable set of one child node $c_i \in \mathsf{Child}(n_c)$. Therefore, assignment of x_i can be sampled independent of x_j where $x_j \in \mathsf{VarSet}(c_j), \forall c_j \neq c_i$. Let τ'_{c_i} be partial assignment for child node $c_i \in \mathsf{Child}(n_c)$. Notice that each partial assignment τ'_{c_i} is sampled independently of others as there are no overlapping variables, hence their joint probability is simply the product of their individual probabilities. This agrees with the weight of an assignment being the product of its components, up to a normalization factor.

Decision node Let n_d be an arbitrary decision node in PROB ψ and x_d be $\mathsf{Var}(n_d)$. At n_d, we sample an assignment of x_d based on the parameters $\theta_{\mathsf{Lo}(x_d)}$ and $\theta_{\mathsf{Hi}(x_d)}$, which are probabilities of literal assignment by Proposition 1. By Proposition 1, one can see that the assignment of x_d is sampled correctly according to W. As the sampling process at n_d is independent of its child nodes by the determinism property, the joint probability of sampled assignment of x_d and the output partial assignment from the corresponding child node would be the product of their probabilities. Notice that the joint probability aligns with the definition of weight of an assignment being the product of the weight of its literals, up to a normalization factor.

Since we do not consider the *false* node and treat it as having 0 probability, we always sample from satisfying assignments by starting at the *true* node in bottom-up ordering. Reconciling the sampling process at the two types of nodes, it is obvious that any combination of decision and $\wedge$-nodes encountered in the sampling process would agree with a given weight function W up to a normalization factor $1/N$. In fact, $N = \sum_{\tau_i \in S} W(\tau_i)$ where S is the set of satisfying assignments of Boolean formula F that ψ represents. As mentioned in Proposition 1 proof, normalization factors do not affect the correctness of sampling according to W, and we have

shown that INC performs weighted sampling correctly under multiplicative weight functions. □

Remark 2. *From the proof of Proposition 2, the determinism and decomposability property is important to ensure the correctness of* INC. *The smoothness property is important to ensure that the sampled assignment by* INC *is complete. For formula* $F = (x \vee y) \wedge (\neg x \vee \neg z)$, *an assignment* τ_1 *sampled from a non-smooth* PROB *could be* $\{x, \neg z\}$. *Notice that* τ_1 *is missing assignment for variable* y. *By performing smoothing, we will be able to sample a complete assignment of all variables in the Boolean formula as both child nodes of each decision node* n *have the same* $\mathsf{VarSet}(\cdot)$.

3.4 Experiments

We implement INC in Python 3.7.10, using NumPy 1.15 and Toposort package. In our experiments, we make use of an off-the-shelf KC diagram compiler, KCBox [LMY25]. We compared against WAPS [Gup+19] using its default bundled d-DNNF compilers, D4 [LM17] for non-projected CNF and DSHARP [Mui+12] for projected CNF. In the later parts of this section, we performed additional comparisons against an implementation of INC using the Gmpy2 arbitrary precision math package ($\mathsf{INC}_{\mathsf{AP}}$) to determine the impact of log-space *annotation* computations.

Our benchmark suite consists of instances arising from a wide range of real-world applications such as DQMR networks, bit-blasted versions of SMT-LIB (SMT) benchmarks, ISCAS89 circuits, and configurable systems [Gup+19; BLM20]. For incremental updates, we rely on the weight generation mechanism proposed in the context of prior applications of incremental sampling [BLM20]. In particular, new weights are generated based on the samples from the previous rounds, resulting in the need to recompute joint probabilities in each round. Keeping in line with prior work, we perform 10 rounds (R1-R10) of incremental weighted sampling and 100 samples drawn in each round. The experiments were conducted with a timeout of 3600 seconds on clusters with Intel Xeon Platinum 8272CL processors.

In this section, we detail the extensive experiments conducted to understand INC's runtime behavior and to compare it with the existing state-of-the-art weighted sampler WAPS [Gup+19] in incremental weighted sampling tasks. We chose WAPS as it

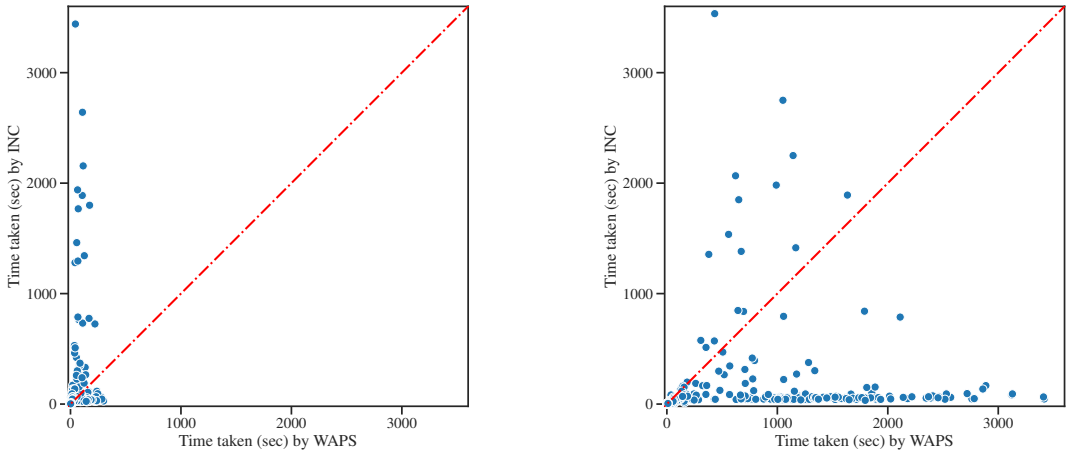
has been shown to achieve significant runtime improvement over other samplers, and accordingly has emerged as a sampler of the choice for practical applications [BLM20]. In particular, our empirical evaluation sought to answer the following questions:

RQ 1 How does INC's incremental weighted sampling runtime performance compare to current state-of-the-art?

RQ 2 How does using PROB affect runtime performance?

RQ 3 How does log-space calculation impact runtime performance?

RQ 4 Does INC correctly perform weighted sampling?



(a) Single Round (R1) Runtime Scatter Plot

(b) Incremental Runtime Scatter Plot

Figure 3.4: Runtime comparisons between INC and state-of-the-art weighted sampler WAPS

3.4.1 RQ 1: Incremental Sampling Performance

The scatter plot of incremental sampling runtime comparison is shown in Figure 3.4, with Figure 3.4a showing runtime comparison for the first round (R1) and Figure 3.4b showing runtime comparison over 10 rounds. The vertical axes represent the runtime of INC and the horizontal axes represent that of WAPS. In the experiments, INC completed 650 out of 896 benchmarks whereas WAPS completed

674. INC completed 21 benchmarks that WAPS timed out and similarly, WAPS completed 45 benchmarks that INC timed out. In the experiments, INC achieved a median speedup of 1.69× over WAPS.

Statistic	WAPS AVG(R2-10) / WAPS R1	INC AVG(R2-10) / INC R1	WAPS R1 / INC R1	WAPS SUM(R2-10) / INC SUM(R2-10)	WAPS Total / INC Total
Mean	0.74	0.064	1.03	15.66	6.12
Std	0.24	0.040	1.47	26.42	10.73
Min	0.22	0.003	0.01	1.37	0.12
25%	0.55	0.031	0.22	3.43	1.01
50%	0.67	0.059	0.44	4.48	1.69
75%	0.92	0.086	0.92	9.38	3.61
Max	1.25	0.188	10.65	172.66	73.96

Table 3.1: Incremental weighted sampling runtime ratio statistics for WAPS and INC (25% stands for the 25th percentile and all numerators and denominators refer to the corresponding runtimes)

Further results are shown in Table 3.1. Observe that for runtime taken for R1 (column 3), WAPS is faster and takes around 0.44× of INC's runtime in the median case. However, INC takes the lead in runtime performance when we examine the total time taken for the incremental rounds R2 to R10 (column 4). For incremental rounds, WAPS always took longer than INC, in the median case WAPS took 4.48× longer than INC. We compare the average incremental round runtime with the first round runtime for both samplers in columns 1 and 2. In the median case, an incremental round for WAPS takes 67% of the time for R1 whereas an incremental

Benchmark	Tool	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	Total	Speed
or-50-5-5-UC-10	WAPS	**56.6**	56.3	52.5	59.4	52.5	53.6	59.4	53.2	53.4	61.7	**558.6**	1.0×
(100, 253)	INC	1461.3	**7.6**	**8.4**	**8.4**	**8.4**	**8.4**	**8.5**	**8.5**	**8.4**	**8.5**	1536.3	0.4×
or-100-20-9-UC-30	WAPS	**73.0**	69.1	66.7	76.0	66.5	66.9	76.6	66.0	66.9	78.6	706.1	1.0×
(200, 528)	INC	269.5	**4.7**	**4.8**	**4.8**	**4.9**	**5.1**	**4.8**	**4.8**	**4.8**	**5.1**	**313.4**	2.3×
s953a_15_7	WAPS	**1.7**	1.1	1.1	1.2	1.0	1.1	1.2	1.1	1.1	1.3	11.9	1.0×
(602, 1657)	INC	4.9	**0.7**	**0.7**	**0.7**	**0.7**	**0.7**	**0.7**	**0.7**	**0.7**	**0.7**	**11.5**	1.0×
h8max	WAPS	90.3	104.2	92.4	116.0	94.3	94.1	112.9	92.9	94.4	120.4	1011.9	1.0×
(1202, 3072)	INC	**34.1**	**2.1**	**2.2**	**2.4**	**2.3**	**2.4**	**2.2**	**2.4**	**2.4**	**2.3**	**55.7**	18.2×
innovator	WAPS	195.5	221.9	201.3	244.4	200.1	206.7	247.2	202.0	202.9	257.4	2179.3	1.0×
(1256, 50452)	INC	**32.8**	**1.6**	**1.8**	**1.9**	**1.9**	**1.9**	**1.8**	**1.9**	**1.9**	**1.9**	**49.4**	44.1×

Table 3.2: Runtime (seconds) breakdowns for each of ten rounds (R1-R10) between WAPS and INC for benchmarks of different sizes e.g. 'h8max' benchmark consists of 1202 variables and 3072 clauses.

Number of rounds	WAPS	INC
10	674	650
20	596	649

Table 3.3: Number of completed benchmarks within 3600s, for 10 and 20 round settings

round for INC only requires 5.9% of the time R1 takes. We show the per round runtime for 5 benchmarks in Table 3.2 to further illustrate INC's runtime advantage over WAPS for incremental sampling rounds, even though both tools reuse the respective KC diagram compiled in R1. This set of results highlights INC's superior performance over WAPS in the handling of incremental sampling settings. INC's advantage in incremental sampling rounds led to better overall runtime performance than WAPS in 75% of evaluations. The runtime advantage of INC would be more obvious in applications requiring more than 10 rounds of samples.

Therefore, we conducted sampling experiments for 20 rounds to substantiate our claims that INC will have a larger runtime lead over WAPS with more rounds. Both samplers are given the same 3600s timeout as before and are to draw 100 samples per round, for 20 rounds. The number of completed benchmarks is shown in Table 3.3 In the 20 sampling round setting, INC completed 649 out of 896 benchmarks, timing out on 1 additional benchmark compared to 10 sampling round setting. In comparison, WAPS completed 596 of 896 benchmarks, timing out on 78 additional benchmarks than in the 10 sampling round setting. In addition, WAPS takes on median 2.17× longer than INC under the 20 sampling round setting, an increase over the 1.69× under the 10 sampling round setting.

The runtime results clearly highlight the advantage of INC for incremental weighted sampling applications and that INC is noticeably better at incremental sampling than the current state-of-the-art.

3.4.2 RQ 2: PROB Performance Impacts

We now focus on the analysis of the impact of using PROB compared to d-DNNF in the design of a weighted sampler. We analyzed the size of both PROB and d-DNNF across the benchmarks that both tools managed to compile and show the results in Table 3.4. From Table 3.4, PROB is always smaller than the corresponding

Statistic	$\frac{\text{WAPS KC size}}{\text{INC KC size}}$
Mean	18.92
Std	81.19
Min	1.10
25%	2.94
50%	4.64
75%	12.56
Max	1734.08

Table 3.4: Statistics for number of nodes in d-DNNF (WAPS KC diagram) over that of smoothed PROB (INC KC diagram). (25% stands for the 25th percentile)

d-DNNF. Additionally, PROB is at median 4.64× smaller than the corresponding d-DNNF, and that for PROB is an order of magnitude smaller for at least 25% of the benchmarks. As such, PROB emerges as the clear choice of knowledge compilation diagram used in INC, owing to its compactness in practice which leads to fast incremental sampling runtimes.

3.4.3 RQ 3: Log-space Computation Performance Impacts

Statistic	$\frac{\text{INC}_{\text{AP}}\text{ runtime}}{\text{INC runtime}}$
Mean	1.14
Std	0.16
Min	0.70
25%	1.02
50%	1.12
75%	1.25
Max	1.89

Table 3.5: Runtime comparison of INC and INC_{AP}

In the design of INC, we utilized log-space computations to perform *annotation* computations as opposed to naively using arbitrary precision math libraries. In order to analyze the impact of this design choice, we implemented a version of INC where the dynamic *annotation* computations are performed using arbitrary precision math in a similar manner as WAPS. We refer to the arbitrary precision math version of INC as INC_{AP}. As an ablation study, we compare the runtime of both implementations across all the benchmarks and show the comparison in Table 3.5. The statistics shown is for the ratio of INC_{AP} runtime to INC runtime, a value of

1.12 means that $\mathsf{INC}_{\mathsf{AP}}$ takes 1.12× that of INC for the corresponding statistics.

The results in Table 3.5 highlight the runtime advantages of our decision to use log-space computations over arbitrary precision computations. INC has faster runtime than $\mathsf{INC}_{\mathsf{AP}}$ in majority of the benchmarks. INC displayed a minimum of 0.70×, a median of 1.12×, and a max of 1.89× speedup over $\mathsf{INC}_{\mathsf{AP}}$. Furthermore, $\mathsf{INC}_{\mathsf{AP}}$ timed out on 2 more benchmarks compared to INC. It is worth emphasizing that log-space computations do not introduce any error, and our usage of them sought to improve on the naive usage of arbitrary precision math libraries.

3.4.4 RQ 4: INC Sampling Quality

We conducted additional evaluation to further substantiate evidence of INC's sampling correctness, apart from theoretical analysis in Section 3.3.4. Specifically, we compared the samples from INC and WAPS, which has proven theoretical guarantees [Gup+19], on the 'case110' benchmark that is extensively used by prior works [Sha+18; AHT18; Gup+19]. We gave each positive literal weight of 0.75 and each negative literal 0.25, and subsequently drew one million samples using both INC and WAPS and compare them in Figure 3.5.

Figure 3.5 shows the distributions of samples drawn by INC and WAPS for 'case110' benchmark. A point (x, y) on the plot represents y number of unique solutions that were sampled x times in the sampling process by the respective samplers. The almost perfect match between the weighted samples drawn by INC and WAPS, coupled with our theoretical analysis in Section 3.3.4, substantiates our claim of INC's correctness in performing weighted sampling. Additionally, it also shows that INC can be a functional replacement for existing state-of-the-art sampler WAPS, given that both have theoretical guarantees.

3.5 Summary

In this chapter, we introduced a bottom-up weighted sampler, INC, optimized for incremental weighted sampling, in line with one of the three pillar of our framework, specifically on the efficient usage of knowledge compilation diagrams. By exploiting the succinct structure of PROB and log-space computations, INC demonstrated superior runtime performance in a series of extensive benchmarks when compared to

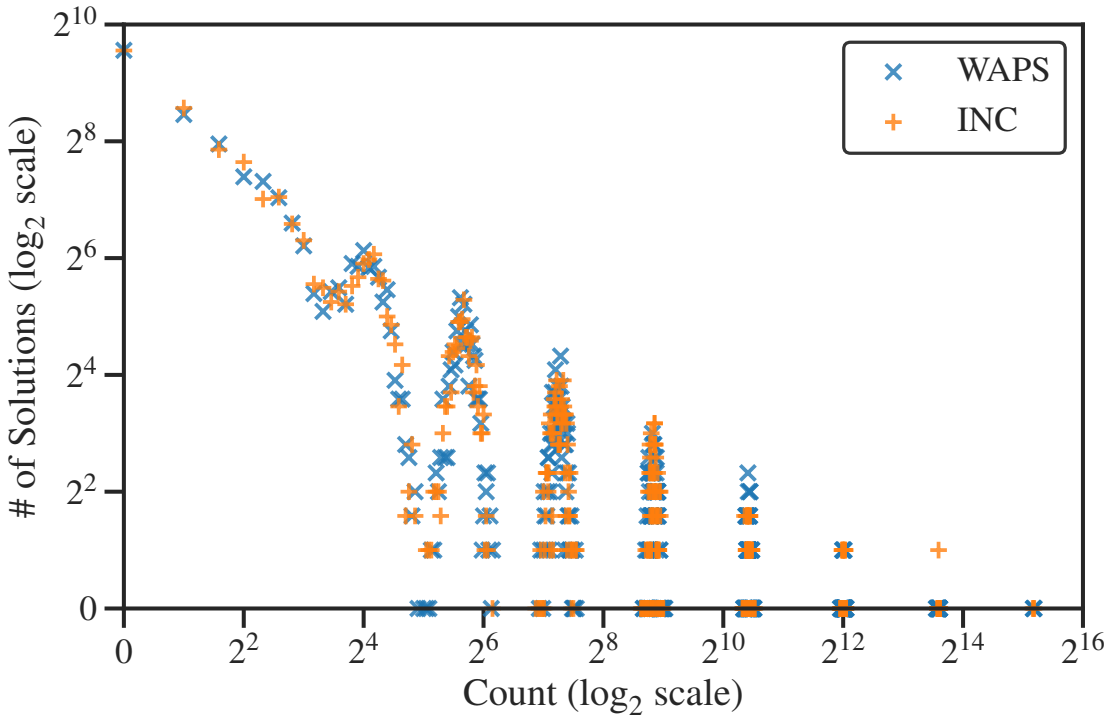


Figure 3.5: Distribution comparison for Case110, with log scale for both axes

the current state-of-the-art weighted sampler WAPS. In addition, INC has a single pass sampling routing as opposed to the existing state-of-the-art which performs *annotation* and sampling in separate passes, contributing to INC's performance lead in our evaluations. In the incremental sampling setting, the compilation costs of KC diagrams are amortized, and since INC is substantially better at handling incremental updates, it thus took the overall runtime lead from WAPS in the majority of the benchmarks. Extrapolating the trend, it is most likely that INC would have a larger runtime lead over WAPS for applications requiring more than 10 sampling rounds. The runtime breakdown demonstrates that INC is able to amortize the compilation time over the incremental sampling rounds, with subsequent rounds being much faster than WAPS. In summary, we demonstrate in this chapter the aspect of efficiently using knowledge compilation diagrams through picking a more succinct diagram form and a more optimized incremental sampling routine.

Chapter 4

Recoverable Approximations With Knowledge Compilation Diagrams

4.1 Motivation

This chapter illustrates the perspective of applying knowledge compilation techniques in a more scalable manner, by looking at the application scenario and relaxing the information that the underlying knowledge compilation diagram needs to represent. In particular, the perspective is demonstrated through the task of constrained sampling of valid routes from historical data. The core idea lies in the observation that ordering information does not need to be represented in the knowledge compilation diagram, but can instead be recovered via a refinement step with knowledge of start and destination.

The past decade has witnessed an unprecedented rise of the service economy, best highlighted by the prevalence of delivery and ride-sharing services [CM17; Col20]. For operational and financial efficiency, a fundamental problem for such companies is the inference and prediction of routes taken by the drivers. When a driver receives a job to navigate from location A to B, the ride-sharing app needs to predict the route in order to determine: (1) the trip time, which is an important consideration for the customer, (2) the fare, an important consideration for both the driver and the customer, and (3) the trip experience since customers feel safe when the driver takes the route described in their app [BRJ15; WFY18]. However, the reality is that drivers and customers have preferences, as such the trips taken are not always the shortest possible by distance or time [LKH06]. To this end, delivery and ride-sharing

service companies have a need for techniques that can infer the distribution of routes and efficiently predict the likely route a driver takes for a given start and end location.

Routing, a classic problem in computer science, has traditionally been approached without considering the learning of distributions [Ahu+90; RSL74]. However, Choi, Shen, and Darwiche demonstrated through a series of papers that the distribution of routes can be conceptualized as a structured probability distribution (SPD) given the underlying combinatorial structure [CSD17; SCD18; She+19]. Decision diagrams, which are particularly well-suited for representing SPDs, have emerged as the state-of-the-art approach for probability-guided routing. The decision diagram-based approach allows for learning of SPDs through the use of decision diagrams augmented with probability values, followed by a stepwise process for uncovering the route.

However, scalability remains a challenge when using decision diagrams to reason about route distributions, particularly for large road networks. Existing works address this concern in various ways, such as through the use of hierarchical diagrams [CSD17] and Structured Bayesian Networks [SCD18]. Choi et al. [CSD17] partition the structured space into smaller subspaces, with each subspace's SPD being represented by a decision diagram. Shen et al. used Structured Bayesian Networks to represent conditional dependencies between sets of random variables, with the distribution within each set of variables represented by a conditional Probabilistic Sentential Decision Diagram (PSDD) [SCD18; She+19]. Despite these efforts, the scalability of decision diagrams for routing, in terms of space complexity, remains an open question [CVD15].

This chapter illustrates how to address scalability challenges faced by current state-of-the-art approaches via the introduction of an approach termed ProbRoute. The contributions are two-fold: first, ProbRoute focuses on minimizing the size of the compiled diagram by *relaxation and refinement*. In particular, instead of learning distributions over the set of all valid routes, ProbRoute learns distributions over an over-approximation, and performs sampling followed by refinement to output a valid route. Secondly, instead of a stepwise sampling procedure, ProbRoute performs one-pass sampling by adapting the existing sampling algorithm [YLM22] to perform conditional sampling. Empirical evaluations over benchmarks arising from real-world road network data demonstrate that ProbRoute can handle real-world instances that

were clearly beyond the reach of the state-of-the-art. Furthermore, on instances that can be handled by prior state-of-the-art, ProbRoute achieves a median of 10× runtime performance improvements.

4.2 Background

In the remaining parts of this chapter, we will discuss how to encode simple, more specifically simple trips, in a graph using Boolean formulas. In addition, we will also discuss decision diagrams and probabilistic reasoning with them. In this chapter, we use the Probabilistic OBDD[∧] (PROB) [YLM22] decision diagram, for which there are existing efficient sampling algorithm. We will discuss the PROB decision diagram in detail in the subsequent section. To avoid ambiguity, we use *vertices* to refer to nodes of road network graphs and *nodes* to refer to nodes of decision diagrams.

4.2.1 Preliminaries

Simple Trip Let G be an arbitrary undirected graph, a path on G is a sequence of connected vertices $v_1, v_2, ..., v_m$ of G where $\forall_{i=1}^{m-1} v_{i+1} \in N(v_i)$, with $N(v_i)$ referring to neighbours of v_i. A path π does not contain loops if $\forall_{v_i,v_j \in \pi} v_i \neq v_j$. π does not contain detour if $\forall_{v_i,v_j,v_k,v_l \in \pi} v_j \notin N(v_i) \vee v_k \notin N(v_i) \vee v_l \notin N(v_i)$ i.e. there is no vertex in π that has three of its neighbour also included in π. Path π is a simple path if it does not contain loops. A simple path π is a simple trip if it does not contain detours. We denote the set of all simple trips in G as $\mathsf{SimpleTrip}(G)$. In Figure 4.1, d-e-h is a simple trip whereas d-e-f-c-b-e-h and d-e-f-i-h are not because they contain a loop and a detour respectively. We use $\mathsf{Term}(\pi)$ to refer to the terminal vertices of path π i.e. both the start and destination vertex of a path.

Probabilistic Routing Problem In this chapter, we tackle the probabilistic routing problem which we define as the following. Given a graph G of an underlying road network, training and testing data D_{train}, D_{test}, start and end vertices s, t, sample path π from s to t such that ε-match rate with ground truth path $\pi' \in D_{test}$ is maximized. We define ε-match rate between π and π' as $|U_{close(\pi)}| \div |U|$ where

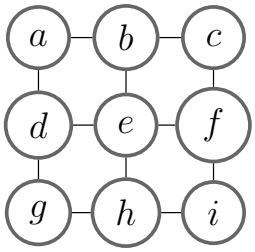


Figure 4.1: A 3 × 3 grid graph

U is the set of vertices of π' and $U_{close(\pi)}$ is the set of vertices of π' that are within ε euclidean distance away from the nearest vertex in π. More details on ε will be discussed in Section 4.4.

4.2.2 Related Work

The continuous pursuit of compact representations by the research community has resulted in several decision diagram forms over the years. Some of the decision diagram forms include AOMDD for multi-valued variables, OBDD and SDD for binary variables [Bry86; Dar11; MDM08]. Both OBDD and SDD are canonical representations of Boolean formulas given variable ordering for OBDD and *Vtree* for SDD respectively. OBDD [Bry86] is comprised of internal nodes that correspond to variables and leaf nodes that correspond to $\top$ or $\bot$. Each internal node of OBDD has exactly two child and represents the Shannon decomposition [Boo54] on the variable represented by that internal node. SDDs are comprised of elements and nodes [Dar11]. Elements represent the conjunction of a *prime* and a *sub*, each of which can either be a terminal SDD or a decomposition SDD. A decomposition SDD is represented by a node with child elements representing the decomposition. A terminal SDD can be a literal, $\top$ or $\bot$. The decompositions of SDDs follow that of the respective *Vtree*, which is a full binary decision tree of Boolean variables in the formula. In this chapter, we use the OBDD[$\wedge$] [LLY17] variant of OBDD, which is shown to be theoretically incomparable but empirically more succinct than SDDs [LLY17].

A related development formulates probabilistic circuits [CVB20], based on sum-product networks [PD11] and closely related to decision diagrams, as a special class of neural networks known as Einsum networks [Peh+20]. In the Einsum network structure, leaf nodes represent different gaussian distributions. By learning from

data, Einsum networks are able to represent SPDs as weighted sums and mixtures of gaussian distributions. Einsum networks address scalability by utilizing tensor operations implemented in existing deep learning frameworks such as PyTorch [Pas+19]. Our work differs from the Einsum network structure, we require the determinism property for decision diagrams whereas the underlying structure for Einsum network lacks this property. We will introduce the determinism property in the following section.

Various Boolean encodings have been proposed for representing paths within a graph, including Absolute, Compact, and Relative encodings [Pre03]. These encodings capture both the vertices comprising the path and the ordering information of said vertices. However, these encodings necessitate the use of polynomial number of Boolean variables, specifically $|V|^2$, $|V|log_2|V|$, and $2|V|^2$ variables for Absolute, Compact, and Relative encoding respectively. While these encodings accurately represent the space of paths within a graph, they are not efficient and lead to high space and time complexity for downstream routing tasks.

Choi, Shen, and Darwiche demonstrated over a series of papers that the distribution of routes can be conceptualized as a structured probability distribution (SPD) given the underlying combinatorial structure [CSD17; SCD18; She+19]. This approach, referred to as the 'CSD' approach in the rest of this paper, builds on top of existing approaches that represent paths using zero-suppressed decision diagrams [Knu05; Ino+16; Kaw+17]. The CSD approach utilizes sentential decision diagrams to represent the SPD of paths and employs a stepwise methodology for handling path queries. Specifically, at each step, the next vertex to visit is determined to be the one with the highest probability, given the vertices already visited and the start and destination vertices. While the CSD approach has been influential in its incorporation of probabilistic elements in addressing the routing problem, it is not without limitations. In particular, there are two main limitations (1) there are no guarantees of completion, meaning that even if a path exists between a given start and destination vertex, it may not be returned using the CSD approach [CSD17]. (2) the stepwise routing process necessitates the repeated computation of conditional probabilities, resulting in runtime inefficiencies.

In summary, the limitations of prior works are Boolean encodings that require a high number of variables, lack of routing task completion guarantees, and numerous

conditional probability computations.

4.2.3 PROB: Probabilistic OBDD[∧]

We use the PROB (Probabilistic OBDD[∧]) decision diagrams introduced in Chapter 3.2, and adopt the same notations for consistency.

Notations We use *nodes* to refer to nodes in PROB ψ and *vertices* to refer to nodes in graph $G(V, E)$ to avoid ambiguity. $\mathsf{Child}(n)$ refers to the children of node n. $\mathsf{Hi}(n)$ refers to the hi child of decision node n and $\mathsf{Lo}(n)$ refers to the lo child of n. $\theta_{\mathsf{Hi}}(n)$ and $\theta_{\mathsf{Lo}}(n)$ refer to the parameters associated with the edge connecting decision node n with $\mathsf{Hi}(n)$ and $\mathsf{Lo}(n)$ respectively in a PROB. $\mathsf{Var}(n)$ refers to the associated variable of decision node n. $\mathsf{VarSet}(n)$ refers to the set of variables of F represented by a PROB with n as the root node. $\mathsf{Subdiagram}(n)$ refers to the PROB starting at node n. $\mathsf{Parent}(n)$ refers to the immediate parent nodes of n in PROB.

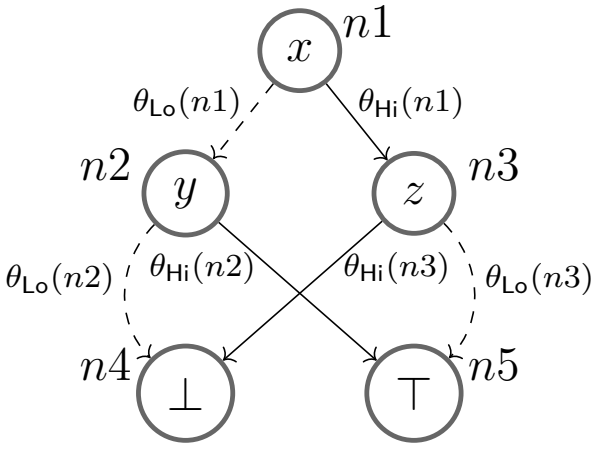


Figure 4.2: A PROB ψ_1 representing $F = (x \vee y) \wedge (\neg x \vee \neg z)$

PROB Structure An assignment of Boolean formula F is represented by a top-down traversal of a PROB compiled from F. For example, we have a Boolean formula $F = (x \vee y) \wedge (\neg x \vee \neg z)$, represented by the PROB ψ_1 in Figure 4.2. When x is assigned *true* and z is assigned *false*, F will evaluate to *true*. If we have a partial assignment τ, we can perform inference conditioned on τ if we visit only the branches of decision nodes in ψ that agree with τ. This allows for conditional sampling, which we discuss in Section 4.3.

PROB inherits important properties of OBDD[∧] that are useful to our algorithms in later sections. The properties are - *determinism*, *decomposability*, and *smoothness*.

In the rest of this chapter, all mentions of PROB refer to smooth PROB. *Smoothness* can be achieved via the smooth algorithm, Algorithm 3.1, introduced in Chapter 3 [YLM22]. It is worth noting that we always create true decision nodes instead of checking via checkMakeTrueDecisionNode at lines 13 and 21 of Algorithm 3.1.

4.3 Approach

In this section, we introduce our approach, ProbRoute, which addresses the aforementioned limitations of existing methods using (1) a novel relaxed encoding that requires a linear number of Boolean variables and (2) a single-pass sampling and refinement approach which provides completeness guarantees. The flow of ProbRoute is shown in Figure 4.3.

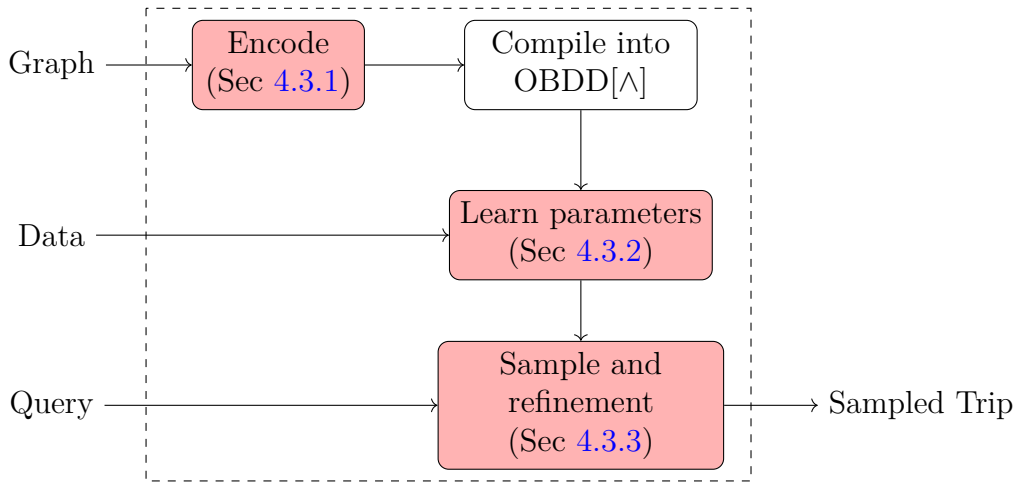


Figure 4.3: Flow of ProbRoute, with red rectangles indicating this work. For compilation, we use existing off-the-shelf techniques.

In our approach, we first use our relaxed encoding to encode SimpleTrip(G) of graph G into a Boolean formula. Next, we compile the Boolean formula into OBDD[∧]. In order to learn from historical trip data, we convert the data into assignments. Subsequently, the OBDD[∧] is parameterized into PROB ψ and the parameters are learned from data. Finally to sample trips from start vertex v_s to destination vertex v_t, we create a partial assignment τ' with the variables that indicate v_s and v_t are terminal vertices set to *true*. The ProbSample algorithm,

algorithm 4.4, takes τ' as input and samples a set of satisfying assignments. Finally, in the refinement step, a simple trip π is extracted from each satisfying assignment τ using depth-first search to remove disjoint loop components.

4.3.1 Relaxed Encoding

In this work, we present a novel relaxed encoding that only includes vertex membership and terminal information. Our encoding only requires a linear $(2|V|)$ number of Boolean variables, resulting in more succinct decision diagrams and improved runtime performance for downstream tasks. In relation to prior encodings, we observed that the ordering information of vertices can be inferred from the graph given a set of vertices and the terminal vertices, thus enabling us to exclude ordering information in our relaxed encoding. Our relaxed encoding represents an over-approximation of trips in $\mathsf{SimpleTrip}(G)$ for graph $G(V, E)$ using a linear number of Boolean variables with respect to $|V|$. We discuss the over-approximation in later parts of this section.

Our encoding uses two types of Boolean variables, n-type and s-type variables. Each vertex $v \in V$ in graph $G(V, E)$ has a corresponding n-type and s-type variable. The n-type variable indicates if vertex v is part of a trip and s-type variable indicates if v is a terminal vertex of the trip. Our encoding is the conjunction of the five types of clauses over all vertices in graph G as follows.

$$\bigvee_{i \in V} s_i \tag{H1}$$

$$\bigwedge_{i \in V} [n_i \longrightarrow \bigvee_{j \in adj(i)} n_j] \tag{H2}$$

$$\bigwedge_{\substack{i,j,k \in V, \\ i \neq j \neq k}} (\neg s_i \vee \neg s_j \vee \neg s_k) \tag{H3}$$

$$\bigwedge_{i \in V} s_i \longrightarrow n_i \wedge \bigwedge_{j,k \in adj(i), j \neq k} (\neg n_j \vee \neg n_k) \tag{H4}$$

$$\bigwedge_{i,j \in V, j \in adj(i)} [n_i \wedge n_j \longrightarrow s_i \vee [(\bigvee_{\substack{k \in adj(i), \\ k \neq j \neq i}} n_k) \wedge \bigwedge_{\substack{l,m \in adj(i), \\ l,m \neq j}} (\neg n_l \vee \neg n_m)]] \tag{H5}$$

A simple trip π in graph G has at least one terminal vertex and at most two terminal vertices, described by encoding components H1 and H3 respectively. At

each terminal vertex v_i of π, there can only be at most 1 adjacent vertex of v_i that is also part of π and this is encoded by H4. For each vertex v_i in π, at least one of their adjacent vertices is in π regardless if v_i is a terminal vertex or otherwise, this is captured by H2. Finally, H5 encodes that if a given vertex v_i and one of its adjacent vertices are part of π, then either another neighbour vertex of v_i is part of π or v_i is a terminal vertex.

Definition 1. *Let* $\mathcal{M} : \mathsf{SimpleTrip}(G) \mapsto \mathsf{Sol}(F)$ *such that for a given trip* $\pi \in \mathsf{SimpleTrip}(G)$, $\tau = \mathcal{M}(\pi)$ *is the assignment whereby the* n*-type variables of all vertices* $v \in \pi$ *and the* s*-type variables of* $v \in \mathsf{Term}(\pi)$ *are set to true. All other variables are set to false in* τ*.*

We refer to our encoding as relaxed encoding because the solution space of constraints over-approximates the space of simple trips in the graph. Notice that while all simple trips correspond to a satisfying assignment of the encoding, they are not the only satisfying assignments. Assignments corresponding to a simple trip π with disjoint loop component β also satisfy the constraints. The intuition is that β introduces no additional terminal vertices, hence H1, H3, and H4 remain satisfied. Since the vertices in β always have n-type variables of exactly two of its neighbours set to *true*, H5 and H2 remain satisfied. Thus, a simple trip with a disjoint loop component also corresponds to a satisfying assignment of our encoding. It is worth noting that a non-simple trip, with loops or detours, will not correspond to a satisfying assignment as H5 will be violated. As a sidenote, while the relaxed encoding handles undirected graphs, it could potentially be possible to extend the encoding using more variables to support directed graphs or enforce direction adherence as an additional refinement step.

4.3.2 Learning Parameters from Data

We introduce algorithm 4.3, ProbLearn, for updating branch counters of PROB ψ from assignments. In order to learn branch parameters $\theta_{\mathsf{Hi}}(n)$ and $\theta_{\mathsf{Lo}}(n)$ of decision node n, we require a counter for each of its branches, $\mathsf{Hi}_{\#}(n)$ and $\mathsf{Lo}_{\#}(n)$ respectively. In the learning process, we have a dataset of assignments for Boolean variables in the Boolean formula represented by PROB ψ. For each assignment τ in the dataset, we perform a top-down traversal of ψ following Algorithm 4.3. In the traversal, we

Algorithm 4.3: ProbLearn - updates counters of ψ from data

Input: PROB ψ, τ - complete assignment of data instance

```
1:  n ← rootNode(ψ);
2:  if n is ∧-node then
3:      for c in Child(n) do
4:          ProbLearn(c, τ);
5:  if n is decision node then
6:      l ← getLiteral(τ, Var(n));
7:      if l is positive literal then
8:          Hi#(n) += 1;
9:          ProbLearn(Hi(n), τ);
10:     else
11:         Lo#(n) += 1;
12:         ProbLearn(Lo(n), τ);
```

visit all child branches of conjunction nodes (line 4) and the child branch of decision node n corresponding to the assignment of $\mathsf{Var}(n)$ in τ (lines 6 to 12), and increment the corresponding counters in the process. Subsequently, the branch parameters for node n are updated according to the following formulas.

$$\theta_{\mathsf{Hi}}(n) = \frac{\mathsf{Hi}_{\#}(n) + 1}{\mathsf{Hi}_{\#}(n) + \mathsf{Lo}_{\#}(n) + 2} \qquad \theta_{\mathsf{Lo}}(n) = \frac{\mathsf{Lo}_{\#}(n) + 1}{\mathsf{Hi}_{\#}(n) + \mathsf{Lo}_{\#}(n) + 2}$$

While we add 1 to numerator and 2 to denominator as a form of Laplace smoothing [MRS08], other forms of smoothing to account for division by zero is possible. Notice that the learnt branch parameters of node n are in fact approximations of conditional probabilities according to Proposition 3 and Remark 3 as follows.

Proposition 3. *Let* $n1$ *and* $n2$ *be decision nodes where* $n1 = \mathsf{Parent}(n2)$ *and* $\mathsf{Lo}(n1) = n2$, $\theta_{\mathsf{Lo}}(n2) = \frac{\mathsf{Lo}_{\#}(n2)+1}{\mathsf{Lo}_{\#}(n1)+2}$ *and* $\theta_{\mathsf{Hi}}(n2) = \frac{\mathsf{Hi}_{\#}(n2)+1}{\mathsf{Lo}_{\#}(n1)+2}$.

Proof. Recall that the Lo branch parameter of $n2$ is:

$$\theta_{\mathsf{Lo}}(n2) = \frac{\mathsf{Lo}_{\#}(n2) + 1}{\mathsf{Hi}_{\#}(n2) + \mathsf{Lo}_{\#}(n2) + 2}$$

Notice that $\mathsf{Hi}_{\#}(n2) + \mathsf{Lo}_{\#}(n2) = \mathsf{Lo}_{\#}(n1)$ as all top-down traversals of ψ that pass through $n2$ will have to pass through the Lo branch of $n1$.

$$\theta_{\mathsf{Lo}}(n2) = \frac{\mathsf{Lo}_{\#}(n2) + 1}{\mathsf{Lo}_{\#}(n1) + 2}$$

A similar argument can be made for $\theta_{\mathsf{Hi}}(n2)$ by symmetry. In the general case if $n2$ has more than one parent, then the term $\mathsf{Hi}_{\#}(n2) + \mathsf{Lo}_{\#}(n2)$ is the sum of counts of branch traversals of all parent nodes of $n2$ that leads to $n2$. Additionally, any conjunction node c between $n1$ and $n2$ will not affect the proof because all children of c will be traversed. For understanding, one can refer to the example in Figure 4.2 where $n1$ corresponds to the root node.

□

Remark 3. *Recall that* $\mathsf{Var}(n1) = x$ *and* $\mathsf{Var}(n2) = y$ *in* PROB ψ_1 *in Figure 4.2. Observe that* $\frac{\mathsf{Lo}_{\#}(n2)}{\mathsf{Lo}_{\#}(n1)}$ *for* PROB ψ_1 *in Figure 4.2 is the conditional probability* $Pr(\neg y|\neg x)$ *as it represents the count of traversals that passed through Lo branch of* $n2$ *out of total count of traversals that passed through Lo branch of* $n1$*. A similar observation can be made for* $\mathsf{Hi}(n2)$*.*

Notice that as the $\mathsf{Lo}_{\#}(n2)$ *and* $\mathsf{Lo}_{\#}(n1)$ *becomes significantly large, that is* $\mathsf{Lo}_{\#}(n2) >> 1$ *and* $\mathsf{Lo}_{\#}(n1) >> 2$*:*

$$\theta_{\mathsf{Lo}}(n2) = \frac{\mathsf{Lo}_{\#}(n2) + 1}{\mathsf{Lo}_{\#}(n1) + 2} \approx \frac{\mathsf{Lo}_{\#}(n2)}{\mathsf{Lo}_{\#}(n1)} = Pr(\neg y|\neg x)$$

As such, the learnt branch parameters are approximate conditional probabilities.

An important property to enable a PROB to learn the correct distribution is smoothness. A non-smooth PROB could be missing certain parameters. An example would be if we have an assignment $\tau_1 = [\neg x, y, -z]$, ψ_1 in Figure 4.2 will not have a counter for $\neg z$ as the traversal ends after reaching the true node from the decision node representing variable y. Observe that the above-mentioned issue would not occur in a smooth PROB ψ_2 in Figure 4.4. It is worth noting that non-zero branch parameters ensure that satisfying assignments not in the parameter learning dataset are sampled with non-zero probability. Additionally, the shared sub-diagram structure of PROB and compositional nature of the sampling process, which we detail in the subsequent section, allow for longer trips to be sampled with higher probability if overlapping trips or sub-trips appeared frequently in parameter learning data.

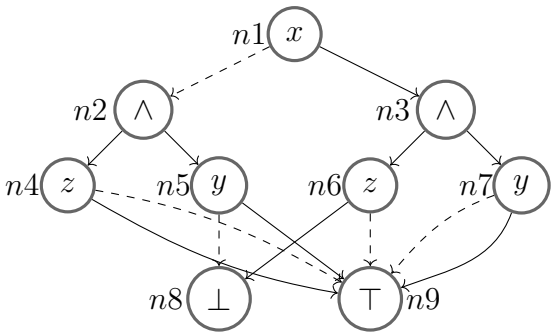


Figure 4.4: A smooth PROB ψ_2 with 9 nodes, $n1, ..., n9$, representing $F = (x \vee y) \wedge (\neg x \vee \neg z)$. Branch parameters are omitted

4.3.3 Sampling Trip Query Answers

The ability to conditionally sample trips is critical to handling trip queries for arbitrary start-end vertices, for which a trip is to be sampled conditioned on the given start and end vertices. In this work, we adapted the weighted sampling algorithm using PROB, which was introduced by prior work [YLM22], to support conditional sampling and denote it as ProbSample.

Algorithm 4.4, ProbSample, performs sampling of satisfying assignments from a PROB ψ in a bottom-up manner. ProbSample takes an input PROB ψ and partial assignment τ' and returns a sampled complete assignment that agrees with τ'. The input τ' specifies the terminal vertices for a given trip query by assigning the s-type variables. ProbSample employs two caches ζ and γ, for partially sampled assignment at each node and joint probabilities during the sampling process. In the process, ProbSample performs calculations of joint probabilities at each node. In addition, ProbSample stores the partial samples at each node in ζ. The partial sample for a false node would be Invalid as it means that an assignment is unsatisfiable. On the other hand, the partial sample for a true node is $\emptyset$ which will be incremented with variable assignments during the processing of internal nodes of ψ. The partially sampled assignment at every $\wedge$-node c is the union of the samples of all its child nodes, as the child nodes have mutually disjoint variable sets due to *decomposability* property. For a decision node d, if $\mathsf{Var}(d)$ is in τ', the partial sample at d will be the union of the literal in τ' and the partial sample at the corresponding child node (lines 11 to 19) to condition on τ'. Otherwise, the partial assignment at d is sampled according to the weighted joint probabilities l and h (lines 21 to 28). Finally, the

Algorithm 4.4: ProbSample - returns sampled assignment

Input: PROB ψ, partial assignment τ'
Output: complete assignment that agrees with τ'

caches $\zeta, \gamma \longleftarrow$ initCache();
for node n in bottom-up ordering of ψ **do**
  if n is $\top$-node **then**
    $\zeta[n] \longleftarrow \emptyset$, $\gamma[n] \longleftarrow 1$;
  else if n is $\bot$-node **then**
    $\zeta[n] \longleftarrow \mathsf{Invalid}$, $\gamma[n] \longleftarrow 0$;
  else if n is $\wedge$-node **then**
    $\zeta[n] \longleftarrow \mathsf{unionChild}(\mathsf{Child}(n), \zeta)$;
    $\gamma[n] \longleftarrow \prod_{c \in \mathsf{Child}(n)} \gamma[c]$;
  else
    if $\mathsf{Var}(n)$ in τ' **then**
      if $\zeta[\tau'[\mathsf{Var}(n)]]$ is $\mathsf{Invalid}$ **then**
        $\zeta[n] \longleftarrow \mathsf{Invalid}$, $\gamma[n] \longleftarrow 0$;
      else
        $\zeta[n] \longleftarrow \mathsf{followAssign}(\tau)$;
        if $\tau'[\mathsf{Var}(n)]$ is $\neg\mathsf{Var}(n)$ **then**
          $\gamma[n] \longleftarrow \theta_{\mathsf{Lo}}(n) \times \gamma[\mathsf{Lo}(n)]$;
        else
          $\gamma[n] \longleftarrow \theta_{\mathsf{Hi}}(n) \times \gamma[\mathsf{Hi}(n)]$;
    else
      $l \longleftarrow \theta_{\mathsf{Lo}}(n) \times \gamma[\mathsf{Lo}(n)]$;
      $h \longleftarrow \theta_{\mathsf{Hi}}(n) \times \gamma[\mathsf{Hi}(n)]$;
      $\gamma[n] \longleftarrow l + h$;
      $\alpha \longleftarrow \mathsf{Binomial}(\frac{h}{l+h})$;
      if α is 1 **then**
        $\zeta[n] \longleftarrow \zeta[\mathsf{Hi}(n)] \cup \mathsf{Var}(n)$;
      else
        $\zeta[n] \longleftarrow \zeta[\mathsf{Lo}(n)] \cup \neg\mathsf{Var}(n)$;
return $\zeta[\text{rootnode}(\psi)]$

output of ProbSample would be the sampled assignment at the root node of ψ. To extend ProbSample to sample k complete assignments, one has to keep k partial assignments in ζ at each node during the sampling process and sample k independent partial assignments at each decision node.

Proposition 4. *Let* PROB ψ *represent Boolean formula* F*,* ProbSample *samples* $\tau \in \mathsf{Sol}(F)$ *according to the joint branch parameters, that is* $\prod_{n \in \mathsf{Rep}_\psi(\tau)}[(1-I_n)\theta_{\mathsf{Lo}}(n) + I_n\theta_{\mathsf{Hi}}(n)]$ *where* I_n *is 1 if* $\mathsf{Hi}(n) \in \mathsf{Rep}_\psi(\tau)$ *and 0 otherwise.*

Proof. Let ψ be a PROB that only consists of decision nodes as internal nodes. At each decision node d in the bottom-up sampling pass, assignment of $\mathsf{Var}(d)$ is sampled proportional to $\theta_{\mathsf{Lo}}(d) \times \gamma[\mathsf{Lo}(d)]$ and $\theta_{\mathsf{Hi}}(d) \times \gamma[\mathsf{Hi}(d)]$ to be *false* and *true* respectively. Without loss of generality, we focus on the term $\theta_{\mathsf{Lo}}(d) \times \gamma[\mathsf{Lo}(d)]$, a similar argument would follow for the other branch by symmetry.

Let $d2$ denote $\mathsf{Lo}(d)$. Notice that $\gamma[d2]$ is $\theta_{\mathsf{Lo}}(d2) \times \gamma[\mathsf{Lo}(d2)] + \theta_{\mathsf{Hi}}(d2) \times \gamma[\mathsf{Hi}(d2)]$. Expanding the equation, the probability of sampling $\neg\mathsf{Var}(d)$ is $\theta_{\mathsf{Lo}}(d) \times \theta_{\mathsf{Lo}}(d2) \times \gamma[\mathsf{Lo}(d2)] + \theta_{\mathsf{Lo}}(d) \times \theta_{\mathsf{Hi}}(d2) \times \gamma[\mathsf{Hi}(d2)]$. If we expand $\gamma[\mathsf{Lo}(d)]$ recursively, we are considering all possible satisfying assignments of $\mathsf{VarSet}(\mathsf{Lo}(d))$, more specifically we would be taking the sum of the product of corresponding branch parameters of each possible satisfying assignment of $\mathsf{VarSet}(\mathsf{Lo}(d))$.

Observe that $\mathsf{Var}(d)$ is sampled to be assigned *false* with probability $\theta_{\mathsf{Lo}}(d) \times \theta_{\mathsf{Lo}}(d2) \times \gamma[\mathsf{Lo}(d2)] + \theta_{\mathsf{Lo}}(d) \times \theta_{\mathsf{Hi}}(d2) \times \gamma[\mathsf{Hi}(d2)]$. Similarly, $\mathsf{Var}(d2)$ is sampled to be assigned *false* with probability $\theta_{\mathsf{Lo}}(d2) \times \gamma[\mathsf{Lo}(d2)]$. Notice that if we view the bottom-up process in reverse, the probability of sampling $\neg\mathsf{Var}(d)$ and $\neg\mathsf{Var}(d2)$ is $\theta_{\mathsf{Lo}}(d) \times \theta_{\mathsf{Lo}}(d2) \times \gamma[\mathsf{Lo}(d2)]$. In the general case, it then follows that a satisfying assignment would reach the *true* node which has γ value set to 1. It then follows that for each $\tau \in \mathsf{Sol}(F)$, τ is sampled with probability $P = \prod_{n \in \mathsf{Rep}_\psi(\tau)}[(1 - I_n)\theta_{\mathsf{Lo}}(n) + I_n\theta_{\mathsf{Hi}}(n)]$. Notice that $\wedge$-nodes have no impact on the sampling probability as no additional terms are introduced in the product of branch parameters. □

Remark 4. *Recall in Remark 3 that $\theta_{\mathsf{Hi}}(n)$ and $\theta_{\mathsf{Lo}}(n)$ are approximately conditional probabilities. By Proposition 4, assignment $\tau \in \mathsf{Sol}(F)$ is sampled with probability proportional to $\prod_{n \in \mathsf{Rep}_\psi(\tau)}[(1 - I_n)\theta_{\mathsf{Lo}}(n) + I_n\theta_{\mathsf{Hi}}(n)]$. Notice that if we rewrite the product of branch parameters as the product of approximate conditional probability, it is approximately the joint probability of sampling τ.*

Refinement In the refinement step, we extract a trip from sampled assignment τ by removing spurious disjoint loop components using depth-first search.

Definition 2. *Let $\mathcal{M}' : \mathsf{Sol}(F) \mapsto \mathsf{SimpleTrip}(G)$ be the mapping function of the refinement process, for a given graph G and its relaxed encoding F. For an assignment $\tau \in \mathsf{Sol}(F)$, let V_τ be the set of vertices in G that have their n-type variables set to*

true in τ. A depth-first search is performed from the starting vertex on V_τ, removing disjoint components. The resultant simple path is $\pi = \mathcal{M}'(G)$.

Although $\mathcal{M}'(\cdot)$ is a many-to-one (i.e. surjective) mapping function, it is not a concern in practice as trips with disjoint loop components are unlikely to occur in real-world or synthetic trip data from which probabilities can be learned.

Theorem 1. *Given $v_s, v_t \in G$, let $\pi_{s,t} \in \mathsf{SimpleTrip}(G)$ be a trip between v_s and v_t. Let $R_{\pi_{s,t}} = \{\tau \mid (\tau \in \mathsf{Sol}(F)) \wedge (\mathcal{M}'(\tau) = \pi_{s,t})\}$. Then,*

$$\Pr[\pi_{s,t} \textit{ is sampled}] \propto \sum_{\tau \in R_{\pi_{s,t}}} \prod_{n \in \mathsf{Rep}_\psi(\tau)} [(1 - I_n)\theta_{\mathsf{Lo}}(n) + I_n \theta_{\mathsf{Hi}}(n)]$$

Proof. From Definition 1 and 2, one can say that given a graph G and its relaxed encoding F, $\forall \pi \in \mathsf{SimpleTrip}(G), \exists \tau \in \mathsf{Sol}(F)$ such that $\mathcal{M}'(\tau) = \pi$.

Notice that sampling $\pi_{s,t}$ amounts to sampling $\tau \in R_{\pi_{s,t}}$. As such, the probability of sampling $\pi_{s,t}$ would be the sum over probability of sampling each member of $R_{\pi_{s,t}}$. Recall that the probability of sampling a single assignment τ is proportional to $\prod_{n \in \mathsf{Rep}_\psi(\tau)}[(1 - I_n)\theta_{\mathsf{Lo}}(n) + I_n \theta_{\mathsf{Hi}}(n)]$ by Proposition 4. As such the probability $Pr[\pi_{s,t}$ is sampled] is proportional to $\sum_{\tau \in R_{\pi_{s,t}}} \prod_{n \in \mathsf{Rep}_\psi(\tau)}[(1 - I_n)\theta_{\mathsf{Lo}}(n) + I_n \theta_{\mathsf{Hi}}(n)]$. □

Remark 5. *It is worth noting that $Pr[\pi_{s,t}$ is sampled$] > 0$, as all branch parameters are greater than 0 by definition. Recall that branch parameters are computed with a '+1' in numerator and '+2' in denominator, and given that branch counters are 0 or larger, branch parameters are strictly positive.*

We also introduce an algorithm ComputeProb to compute probability of an assignment τ. ComputeProb is similar to ProbSample, less the sampling aspect. ComputeProb is the joint probability computation component of ProbSample. In line 11, the γ cache value is the product of branch parameter and child γ cache value of the corresponding assignment of $\mathsf{Var}(n)$ instead of both possible assignments in line 13.

Algorithm 4.5: ComputeProb - returns probability of τ

Input: PROB ψ, Assignment τ
Output: probability of τ

Input: PROB ψ, Assignment τ
Output: probability of τ

```
1:  cache γ ⟵ initCache();
2:  for node n in bottom-up pass of ψ do
3:      if n is ⊤-node then
4:          γ[n] ⟵ 1;
5:      else if n is ⊥-node then
6:          γ[n] ⟵ 0;
7:      else if n is ∧-node then
8:          γ[n] ⟵ ∏_{c∈Child(n)} γ[c];
9:      else
10:         if Var(n) in τ then
11:             γ[n] ⟵ assignProb(θ_Hi(n), θ_Lo(n), γ);
12:         else
13:             γ[n] ⟵ θ_Lo(n) × γ[Lo(n)] + θ_Hi(n) × γ[Hi(n)];
14: return γ[rootnode(ψ)]
```

4.4 Experiments

In order to evaluate the efficacy of ProbRoute, we built a prototype in Python 3.8 with NumPy [Har+20], toposort [Smi22], OSMnx[Boe17], and NetworkX [HSS08] packages. We employ KCBox tool[LMY25][1] for OBDD[$\wedge$] compilation [LLY17]. The experiments were conducted on a cluster of machines with Intel Xeon Platinum 8272CL processors and 64GB of memory. In the experiments, we evaluated ProbRoute against an adaptation of the state-of-the-art probabilistic routing approach [CSD17] and an off-the-shelf non-probabilistic routing library, Pyroutelib3 [WK17], in terms of quality of trip suggestions and runtime performance. In particular, we adapted the state-of-the-art approach by Choi et al [CSD17] to sample for trips instead of computing the most probable trip and refer to the adapted approach as 'CSD' in the rest of the section. In addition, we compared our relaxed encoding to existing path encodings across various graphs, specifically to absolute encoding and compact encoding [Pre03].

Through the experiments, we investigate the following:

[1] https://github.com/meelgroup/KCBox

R1 Can ProbRoute effectively learn from data and sample quality trips?

R2 How efficient is our relaxed encoding technique?

R3 What is the runtime performance of the ProbRoute?

Data Generation In this study, we use the real-world road network of Singapore obtained from OpenStreetMap [Ope17] using OSMnx. The road network graph G_r consisted of 23522 vertices and 45146 edges along with their lengths. In addition, we also use an abstracted graph in the form of geohash level 5 partitions[2] of G_r which we denote as G_a for the remaining of this section. A vertex in G_a corresponds to a unique subgraph, in this case a geohash level 5 region, of G_r.

Synthetic trips were generated by deviating from shortest path given start and end vertices. A random pair of start and end vertices were selected in G_r and the shortest path π was found. Next, the corresponding intermediate regions of π in G_a are blocked in G_r, and a new shortest path π' was found and deemed to be the synthetic trip generated. We generated synthetic trips for 11000 random pairs of start and end vertices, 10000 trips for training and 1000 trips for evaluation. While we used G_a to keep the trip sampling time reasonable, it is possible to use more fine-grained regions for offline applications. In addition, we do not prevent the generation of overlapping or intersecting trips i.e. the trips might not be disjoint. Thus, the probability of longer trips are allowed to be influenced by smaller overlapping trips.

4.4.1 R1: ProbRoute's Ability to Learn Probabilities

To understand ProbRoute's ability to learn probabilities from data, we evaluate its ability to produce trips that closely resembles the ground truth. Both ProbRoute and CSD, which are sampling-based approaches, were evaluated on the 1000 start-destination instances. For each start-destination instance, the respective approaches are tasked to each sample 20 trips and we evaluate on the median match rate for the 20 trips. Recall that the ε-match rate is defined as the proportion of vertices in the ground truth trip that were within ε meters of euclidean distance from the closest vertex in the proposed trip. In the evaluation, we set the ε tolerance to be

[2]For more information on the format https://en.wikipedia.org/wiki/Geohash

the median edge length of G_r, which is 64.359 meters, to account for parallel trips. To further emphasize the advantages of probabilistic approaches, we included an off-the-shelf routing library, Pyroutelib3 [WK17], in the comparison. Pyroutelib3 is a routing library which loads OpenStreetMap data, and finds routes given start and destination vertex coordinates, in the process respecting rules such as one-way paths and turn restriction attributes. Additionally, Pyroutelib3's routing algorithm is not probabilistic i.e. it does not take historical trip data into consideration.

In order to conduct a fair comparison, we have adapted the CSD approach to utilize PROB derived from our relaxed encoding. Our evaluation utilizes this adapted approach to sample a trip on G_a in a stepwise manner, where the probability of the next step is conditioned on the previous step and destination. The conditional probabilities are computed in a similar manner to the computations of joint probabilities, which are the γ cache values, in the ProbSample. The predicted trip on the road network G_r is determined by the shortest trip on the subgraph formed by the sequence of sampled regions. In contrast, ProbRoute samples a trip on G_a in a single pass, and subsequently retrieves the shortest trip on the subgraph of the sampled regions as the predicted trip on G_r. It is worth noting that for sampling-based approaches, there may be instances where a trip cannot be found on G_r due to factors such as a region in G_a containing disconnected components. We incorporated a rejection sampling process with a maximum of 400 attempts and 5 minutes to account for such scenarios.

Stats	Exact Match			ε-Match		
	Pyroutelib	CSD	ProbRoute	Pyroutelib	CSD	ProbRoute
25%	0.045	0.049	**0.082**	0.061	0.066	**0.102**
50%	0.088	0.160	**0.310**	0.107	0.172	**0.316**
75%	0.185	0.660	**1.000**	0.208	0.663	**1.000**
Mean	0.151	0.359	**0.445**	0.171	0.372	**0.456**

Table 4.1: Match rate statistics for completed benchmark instances by respective methods. The percentages under 'Stats' column refer to the corresponding percentiles. 'Exact Match' refers to match rate when $\varepsilon = 0$, and 'ε-Match' refers to match rate when ε is set to median edge length of G_r.

Table 4.1 shows the match rate statistics of the respective methods. Under ε-Match setting, where ε is set as the median edge length of G_r to account for

parallel trips, ProbRoute has the highest match rate among the three approaches. In addition, ProbRoute produced perfect matches for more than 25% of instances. ProbRoute has 0.316 ε-match rate on median, significantly more than 0.172 for CSD and 0.107 for Pyroutelib. The trend is similar for exact matches, where ε is set to 0 as shown under the 'Exact Match' columns in Table 4.1. In the exact match setting, ProbRoute achieved a median of 0.310 match rate, almost double that of CSD's 0.160 median match rate. The evaluation results also demonstrate the usefulness of probabilistic approaches such as ProbRoute, especially in scenarios where experienced drivers navigate according to their own heuristics which may be independent of the shortest trip. In particular, ProbRoute would be able to learn and suggest trips that align with the unknown heuristics of driver preferences given start and destination locations. Thus, the results provide an affirmative answer to **R1**.

We show in Table 4.2 additional ε-match rate statistics on how well ProbRoute performs when provided different amount of data to learn probabilities. As we increase the amount of data provided for learning, in increments of 2000 instances (20% of 10000 total), there is a general improvement in the match rate of the trips produced by ProbRoute. A similar trend is observed when $\varepsilon = 0$, with corresponding stats shown in Table 4.3.

Stats	ProbRoute					
	0%	20%	40%	60%	80%	100%
Mean	0.210	0.416	0.434	0.451	0.466	0.456
Std	0.192	0.360	0.373	0.376	0.383	0.386
25%	0.081	0.102	0.095	0.098	0.098	0.102
50%	0.149	0.286	0.297	0.318	0.349	0.316
75%	0.257	0.715	0.854	0.964	1.000	1.000

Table 4.2: ε-match rate statistics for ProbRoute where ε is set as median edge length of road network graph G_r. The percentages under 'Stats' column refer to the percentiles, for example '25%' row refer to the 25th percentile match rate for various methods. The percentages under ProbRoute header indicates the percentage of data that ProbRoute has learned from, out of the 10000 learning instances in total.

Stats	ProbRoute					
	0%	20%	40%	60%	80%	100%
Mean	0.192	0.404	0.422	0.440	0.455	0.445
Std	0.192	0.365	0.379	0.382	0.389	0.391
25%	0.063	0.080	0.075	0.078	0.077	0.082
50%	0.132	0.275	0.282	0.308	0.334	0.305
75%	0.243	0.702	0.848	0.963	1.000	1.000

Table 4.3: Exact match rate statistics for ProbRoute where $\varepsilon = 0$. The percentages under 'Stats' column refer to the percentiles, for example '25%' row refer to the 25th percentile match rate for various methods. The percentages under ProbRoute header indicates the percentage of data that ProbRoute has learned from, out of the 10000 learning instances in total.

Encoding	Grid				SGP
	2	3	4	5	
Absolute	99	1500	31768	1824769	TO
Compact	771	TO	TO	TO	TO
Relaxed(Ours)	**31**	**146**	**2368**	**20030**	**38318**

Table 4.4: Comparison of OBDD[$\wedge$] size for different graphs, with 3600s timeout. Grid 2 refers to a 2x2 grid graph. SGP refers to abstract graph (G_a) of Singapore road network.

4.4.2 R2: Efficiency of Relaxed Encoding

We compared our relaxed encoding to existing path encodings across various graphs, specifically to absolute encoding and compact encoding [Pre03]. In the experiment, we had to adapt compact encoding to CNF form with Tseitin transformation [Tse83], as CNF is the standard input for compilation tools. We compiled the CNFs of the encodings into OBDD[$\wedge$] form with 3600s compilation timeout and compared the size of resultant diagrams. The results are shown in Table 4.4, with rows corresponding to the different encodings used and columns corresponding to different graphs being encoded. Entries with *TO* indicate that the compilation has timed out. Table 4.4 shows that our relaxed encoding consistently results in smaller decision diagrams, up to 91$\times$ smaller. It is also worth noting that relaxed encoding is the only encoding that leads to compilation times under 3600s for the abstracted *Singapore* graph. The results strongly support our claims about the significant improvements that our relaxed encoding brings.

4.4.3 R3: ProbRoute's Runtime Performance

Stats	$\frac{\text{CSD}}{\text{Pyroutelib}} \times 10^3$	$\frac{\text{ProbRoute}}{\text{Pyroutelib}} \times 10^3$
25%	6.33	**1.40**
50%	21.64	**2.00**
75%	47.90	**3.03**
Mean	36.16	**2.62**

Table 4.5: Relative runtime statistics (lower is better) for completed instances by CSD and ProbRoute. Under column '$\frac{\text{CSD}}{\text{Pyroutelib}}$' and row '50%', CSD approach takes a median of 21.64×10^3 times the runtime of Pyroutelib.

For wide adoption of new routing approaches, it is crucial to be able to handle the runtime demands of existing applications. As such, we measured the relative runtimes of probabilistic approaches, that is ProbRoute and CSD, with respect to existing routing system Pyroutelib and show the relative runtimes in Table 4.5. From the result, ProbRoute is more than one order of magnitude faster on median than the existing probabilistic approach CSD. The result also shows that ProbRoute is also on median more than a magnitude closer to Pyroutelib's runtime using the same PROB as compared to CSD approach. In addition, CSD approach timed out on 650 of the 1000 test instances, while ProbRoute did not time out. Additionally, as mentioned in [CSD17], CSD does not guarantee being able to produce a complete trip from start to destination. The results in Table 4.5 highlight the progress made by ProbRoute in closing the gap between probabilistic routing approaches and existing routing systems.

4.5 Summary

In this chapter, we focused on addressing the scalability barrier for reasoning over route distributions as the application, demonstrating the aspect of recoverable relaxations of our framework. In particular, we contribute two techniques: a relaxation and refinement approach that allows us to efficiently and compactly compile routes corresponding to real-world road networks, and a one-pass route sampling technique adapted from INC in Chapter 3. We demonstrated the effectiveness of our approach on a real-world road network and observed around 91× smaller PROB, 10× faster trip sampling runtime, and almost 2× the match rate of state-of-the-art probabilistic

approach, bringing probabilistic approaches closer to achieving comparable runtime to traditional routing tools.

Chapter 5

Alternative Logic Forms for Knowledge Compilation Applications

5.1 Motivation

In this chapter, we highlight our framework pillar on alternative logic forms for knowledge compilation applications. In particular, we demonstrate the idea of considering alternative logic forms for knowledge compilation via the classic application of model counting. The core idea is that certain problems and constraints are more suitable for logic forms other than the typically used propositional boolean formula in conjunctive normal form. Considering a more suitable logic form could increase the efficacy of knowledge compilation techniques in the particular application, in this case, model counting.

Propositional model counting involves computing the number of satisfying assignments to a Boolean formula. Model counting is closely related to the Boolean satisfiability problem where the task is to determine if there exists an assignment of variables such that the Boolean formula evaluates to *true*. Boolean satisfiability and model counting have been extensively studied in the past decades and are the cornerstone of an extensive range of real-life applications such as software design, explainable machine learning, planning, and probabilistic reasoning [BDP03; Nar+19; Jac19; FMM20]. Owing to decades of research, there are numerous tools and techniques developed for various aspects of Boolean satisfiability and model counting, from Boolean formula preprocessors to SAT solvers and model counters.

The dominant representation format of Boolean formulas is Conjunctive Normal

Form (CNF), and accordingly, the tools in the early days focused on CNF as the input format. Over the past decade and a half, there has been considerable effort in exploring other representation formats: one such format that has gained significant interest from the community is Pseudo-Boolean (PB) formulas, which are expressed as the conjunction of linear inequalities. PB formulas are shown to be more succinct than CNF formulas and natural for problems such as Knapsack, sensor placement, binarized neural networks, and the like. Furthermore, PB formulas are able to express constraints more succinctly compared to Boolean formulas in CNF [Le +18]. As an example, a single PB constraint is sufficient to express at-most-k and at-least-k types of cardinal constraints whereas the equivalent in CNF would require a polynomial number of clauses [Sin05]. On a higher level, an arbitrary CNF clause can be expressed with a single PB constraint but the converse is not true [Le +18]. The past decade has witnessed the development of satisfiability solving techniques based on the underlying proof systems naturally suited to PB constraints, and accordingly, the state-of-the-art PB solvers, such as RoundingSat significantly outperform CNF solvers on problems that are naturally encoded in PB [EN18; Dev20; Dev+21].

In contrast to satisfiability, almost all the work in the context of model counting has focused on the representation of Boolean formulas in Conjunctive Normal Form (CNF), with the sole exception of the development of an approximate model counter for PB formulas [YM21].

This chapter explores knowledge compilation diagrams for alternative logic forms, in particular for PB formulas, by introducing a native scalable exact model counter PBCount and its extended variant PBCount2. PBCount is based on the knowledge compilation paradigm, and in particular, compiles a given PB formula into algebraic decision diagrams (ADDs) [Bah+93], which allows us to perform model counting for PB formulas natively. Furthermore, we extend PBCount with projected PB model counting and incremental model counting capabilities and term the extended version PBCount2. PBCount is a bottom-up PB counter, in that the computation process of model count builds up the count from small component ADDs that represent individual constraints of the PB formula. In the name of completeness, this chapter also explores the top-down model counter design in the context of PB model counting, by introducing an exact PB counter PBMC. PBMC uses a top-down search-based approach along with techniques similar to conflict-driven clause learning (CDCL),

adapted for PB formulas, to perform model counting. In particular, PBMC builds on top of methodologies adapted from RoundingSat PB solver [EN18], GPMC CNF model counter [RS17], and PBCount PB counter [YM24]. In addition, PBMC is related to knowledge compilation as the implicit search tree of its computation process can be framed as a top-down compilation process for decision-dnnf or even ddnnf knowledge compilation form, similar to top-down knowledge compilers such as D4.

We perform extensive empirical evaluations on benchmark instances arising from different applications, such as sensor placement, multi-dimension knapsack, and combinatorial auction benchmarks [GL80; BN07; LSM23]. Our evaluations highlighted the efficacy of PBCount and PBCount2 against existing state-of-the-art CNF model counters. Firstly, we replace the ADD compilation mechanism of existing work [DPV20a] to natively handle PB constraints, resulting in PBCount. PBCount is able to successfully count 1513 instances while the prior state of the art could only count 1013 instances, thereby demonstrating significant runtime improvements. It is worth remarking that PBCount achieves superior performance with substantially weaker preprocessing techniques in comparison to techniques employed in CNF model counters, making a strong case for the advantages of considering alternative logic formats for knowledge compilation applications.

Subsequently with PBCount2, we introduce the *Least Occurrence Weighted Min Degree* computation order heuristic (LOW-MD in short) to support projected PB model counting, and caching mechanism to support incremental model counting. In our evaluations, PBCount2 is able to successfully count 1957 projected model counting instances while the state-of-the-art CNF-based projected model counter could only count 1398 instances. Our evaluations also showed that PBCount2 is able to complete 1618 instances for incremental benchmarks involving 5 counts with 1 count for the initial PB formula and 4 counts for modification steps, whereas state-of-the-art CNF model counters completed less than 1000 incremental benchmarks.

The prototype implementations of PBCount and PBCount2 demonstrate the usefulness of considering alternative forms to encode problems when pursuing knowledge compilation-based applications, as opposed to the convention of always using CNF formulas. The evaluations show that potentially huge performance improvements can be gained by switching to more suitable logic forms for certain

problems.

5.2 Preliminaries

Algebraic Decision Diagram As introduced in Chapter 2, an algebraic decision diagram (ADD) is a directed acyclic graph representation of a function $f : 2^{\mathsf{Var}(f)} \to S$ where $\mathsf{Var}(f)$ is the set of Boolean variables that f is defined over, and S is an arbitrary set known as the carrier set. In the computations involving ADDs in this chapter, we make use of the Apply and ITE operations on ADDs [Bry86; Bah+93]. The Apply operation takes as input a binary operator $\bowtie$, two ADDs ψ_1, ψ_2, and outputs an ADD ψ_3 such that the $\mathsf{Func}(\psi_3) = \mathsf{Func}(\psi_1) \bowtie \mathsf{Func}(\psi_2)$. We use the term *merge* to refer to the usage of Apply with $\times$ operator on two ADDs. Referring to the explanation of Apply, we say ψ_3 represents PB constraints $\{c_1, c_2\}$ if ψ_1 represents $\{c_1\}$ and ψ_2 represents $\{c_2\}$. The ITE operation (*if-then-else*) involves 3 ADDs ψ_1, ψ_2, ψ_3, where carrier set of ψ_1 is restricted to $\{0, 1\}$. ITE outputs an ADD that is equivalent to having 1 valued leaf nodes in ψ_1 replaced with ψ_2 and 0 valued leaf nodes with ψ_3.

5.2.1 Model Counting with ADDs

In this work, we adapt the existing dynamic programming counting algorithm of ADDMC [DPV20a], shown in Algorithm 5.6, as the building block of PBCount to perform PB model counting with ADDs. This includes using the default ADDMC configurations for ADD variable ordering (MCS) and cluster ordering ρ (BOUQUET_TREE). The ADDMC approach involves partitioning the input formula into clusters, which can be viewed as subtasks in the dynamic programming paradigm. The algorithm takes in a list φ of ADDs, representing all constraints, and a cluster ordering ρ which specifies the order to process the ADDs. ADD ψ is initialized with value 1. According to cluster ordering ρ, cluster ADDs ψ_j are formed using the Apply operation with $\times$ operator on each of the individual constraint ADDs of constraints in the cluster. The cluster ADD ψ_j is combined with ψ using the same Apply operation. If variable x does not appear in later clusters in ρ, it is projected out from ψ (early projection process in ADDMC) using $\psi \leftarrow \psi[x \mapsto 0] + \psi[x \mapsto 1]$

in line 8. Once all clusters have been processed, the unprocessed variables x of the formula F are projected out using the same operation as before (line 10). After all variables are projected out, ψ is a constant ADD that represents the final count.

Algorithm 5.6: computeCount(φ, ρ)

Input: φ - list of ADD, ρ - cluster merge ordering
Output: model count

```
1:  ψ ← constantADD(1);
2:  for cluster A_j ∈ ρ do
3:  |   ψ_j ← constantADD(1);
4:  |   for constraint C_i ∈ A_j do
5:  |   |   ψ_j ← ψ_j × φ[C_i];
6:  |   ψ ← ψ × ψ_j;
7:  |   foreach x ∈ ψ where x not in later clusters in ρ do
8:  |   |   ψ ← ψ[x ↦ 0] + ψ[x ↦ 1];
9:  forall unprocessed variable x do
10: |   ψ ← ψ[x ↦ 0] + ψ[x ↦ 1];
11: return getValue(ψ)
```

Projected Model Counting Let F be a formula and $\mathsf{Var}(F)$ be the set of variables of F. Let X and Y be disjoint subsets of $\mathsf{Var}(F)$ such that $X \cap Y = \emptyset$ and $X \cup Y = \mathsf{Var}(F)$. (F, X, Y) is a projected model counting instance. As previously introduced in Chapter 2, the projected model count of F on X is the number of assignments of all variables in X such that there exists an assignment of variables in Y that makes F evaluate to *true* [Azi+15]. Non-projected model count is a special case of projected model counting where all variables are in the projection set, i.e. $X = \mathsf{Var}(F)$, and $Y = \emptyset$. More formally, the projected model count is $\sum_{\beta \in 2^X} (\max_{\alpha \in 2^Y} [F](\alpha \cup \beta))$ [DPV21]. Where $[F](\alpha \cup \beta)$ is the evaluation of F with the variable assignment $\alpha \cup \beta$, and returns 1 if F is satisfied and 0 otherwise. With reference to Figure 5.1, suppose the PB formula has only the single PB constraint $3x_1 + 4x_2 \geq 3$, then the 3 satisfying assignments are $(x_1, x_2), (x_1, \bar{x_2}), (\bar{x_1}, x_2)$. If the projection set is x_1 then the corresponding projected model count is 2, because both partial assignments involving x_1 can be extended to a satisfying assignment.

Project-Join Tree Let F be a formula. A project-join tree [DPV20b] of F is a tuple $\mathcal{T} = (T, r, \gamma, \pi)$ where T is a tree with nodes $\mathcal{V}(T)$ and rooted at node r. γ is

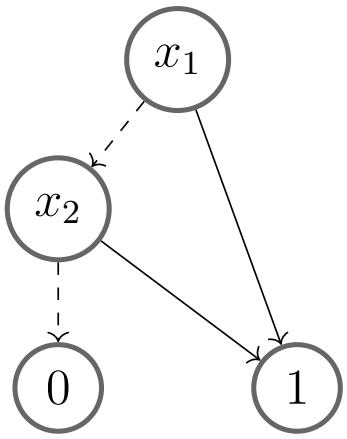


Figure 5.1: An ADD representing $3x_1 + 4x_2 \geq 3$ with ordering x_1, x_2

a bijection from leaves $\mathcal{L}(T)$ of T to constraints of F and π is a labelling function $\mathcal{V}(T) \setminus \mathcal{L}(T) \to 2^{\mathsf{Var}(F)}$ on internal nodes $\mathcal{V}(T) \setminus \mathcal{L}(T)$ of T. Additionally, $\mathcal{T}$ satisfies the following:

1. The set $\{\pi(n) : n \in \mathcal{V}(T) \setminus \mathcal{L}(T)\}$ partitions $\mathsf{Var}(F)$
2. For an arbitrary internal node n. Let x be a variable in πn, and c be a PB constraint of F. If $x \in \mathsf{Var}(c)$, then leaf node $\gamma^{-1}(c)$ is a descendant of n.

X,Y-Graded Project-Join Tree Let F be a formula with project join tree $\mathcal{T}$, and variable sets X, Y being partitions of $\mathsf{Var}(F)$. The project join tree $\mathcal{T}$ is an X, Y graded project-join tree [DPV21] if there exist grades $\mathcal{I}_X$ and $\mathcal{I}_Y$ such that:

1. The set $\{\mathcal{I}_X, \mathcal{I}_Y\}$ is a partition of internal nodes of $\mathcal{T}$
2. If node $n_X \in \mathcal{I}_X$, $\pi(n_X) \subseteq X$
3. If node $n_Y \in \mathcal{I}_Y$, $\pi(n_Y) \subseteq Y$
4. If $n_X \in \mathcal{I}_X$ and $n_Y \in \mathcal{I}_Y$, n_X is not a descendant of n_Y in tree T rooted at r.

Projected Model Counting with Project Join Trees Over a series of works [DPV20a; DPV20b; DPV21] Dudek, Phan, and Vardi established a CNF model counting framework with project join trees and extended the framework to projected CNF model counting by employing a specific type of project join tree,

namely an X, Y-graded project join tree. Dudek, Phan, and Vardi showed that performing $\exists$-projection at $\mathcal{I}_Y$ nodes and Σ-projection at $\mathcal{I}_X$ according to $\mathcal{T}$ produces the correct projected CNF model count [DPV21]. We show that our projected PB model counting approach in PBCount2 shares similarities with the project join tree framework in Section 5.4.2, providing the intuition for algorithm correctness.

Search-Based Model Counting Existing search-based model counters, typically for CNF formulas, adopt similar search methodologies in solvers, which tend to be the Conflict-Driven Clause Learning (CDCL) algorithm, to compute the model count. Notable counters with the search-based component caching design include D4, GPMC, Ganak, and SharpSAT-TD [LM17; RS17; Sha+19; KJ21] which demonstrated superior performance as winners of recent model counting competitions. On a high level, search-based counters iteratively assign values to variables until all variables are assigned a value or a conflict is encountered. In the process, sub-components are created by either branching on a variable or splitting a component into smaller variable-disjoint components. The counters cache the components and their respective counts to avoid duplicate computation. When all variables are assigned, the counter reaches a satisfying assignment and will assign the current component the count 1 and backtrack to account for other branches. When the counter encounters a conflict i.e. the existing assigned variable already falsifies the formula, it learns the reason for the conflict and prunes the search space such that this sub-branch of the search will be avoided in the remaining computation process. A leaf component in the implicit search tree either has counts 0 and 1 for conflict and all variables assigned states respectively. The count of a component that has variable-disjoint sub-components is the product of the counts of its sub-components. The count of a component that branches on a variable is the sum of the counts from the two branches. After accounting for all search branches, the counter returns the final count of satisfying assignments.

5.3 Preprocessing and Individual PB Constraint ADDs

We show the overall flow of PBCount in Figure 5.2. We first preprocess the PB formula using *propagation* and *assumption probing*. Subsequently, we compile each of the PB constraints into an algebraic decision diagram (ADD). Next, we merge constraint ADDs using Apply operation and perform model counting by projecting out variables (Section 5.2.1). The model count would be the value after all variables are projected out. Without loss of generality, the algorithms described in this chapter handle PB constraints involving '=' and '$\geq$' operators, as '$\leq$' type constraints can be manipulated into '$\geq$' type constraints.

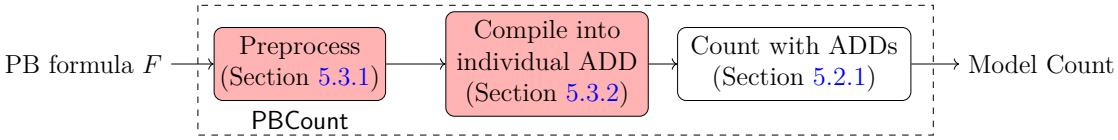


Figure 5.2: Overall flow of our PB model counter PBCount. Shaded boxes indicate our contributions in PBCount and non-shaded boxes indicate adaptations from prior works.

5.3.1 Preprocessing

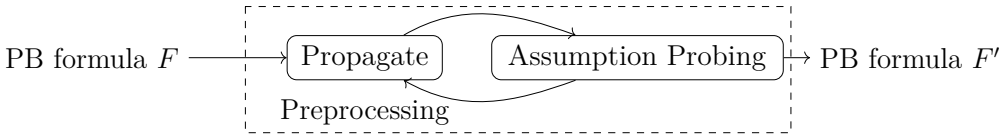


Figure 5.3: Preprocessing of PB formula

The preprocessing phase of PBCount performs assumption probing and unit propagation [BJK21]. PBCount repeatedly performs unit propagation and assumption probing until no change is detected, as shown in Algorithm 5.7.

Sign Manipulation Let C be the PB constraint $-3x_1 - 4x_2 \leq -3$. One can multiply both sides of the constraint by -1 to form $3x_1 + 4x_2 \geq 3$. In addition, one

Algorithm 5.7: Preprocess(F)

Input: F - PB formula
Output: F' - preprocessed PB formula

```
mapping ← [ ]; F' ← F;
repeat
    forall single variable constraint C ∈ F' do
        mapping ← mapping ∪ InferDecision(C);
    F' ← propagate(F', mapping);
    forall variable x ∈ F' do
        mapping ← mapping ∪ AssumProbe(F', x);
    F' ← propagate(F', mapping);
until F' does not change;
return F'
```

would be able to switch the sign of the coefficient of x_2 as follows.

$$3x_1 + 4x_2 \geq 3$$
$$3x_1 + 4(1 - \bar{x}_2) \geq 3$$
$$3x_1 - 4\bar{x}_2 \geq -1$$

In general, one is able to manipulate the sign of any term coefficient as shown in the example above. We use the above technique to optimize PB constraint compilation approaches in Algorithms 5.13 and 5.14,which we discuss in later sections.

Propagation Propagation in the Pseudo-Boolean context refers to the simplification of the PB constraints if decisions on some PB variables can be inferred. In particular, one might be able to infer decisions on PB variable x_i from PB constraint C_j when the constraint is of either 1) $a_i x_i \geq k$ or 2) $a_i x_i = k$ forms using the InferDecision algorithm. We show the pseudocode of the InferDecision in Algorithm 5.8.

Assumption Probing Assumption probing can be viewed as a weaker form of failed literal probing [BJK21] as well as single step look ahead propagation process. For an arbitrary variable $x_i \in F$, where F is the PB formula, assumption probing involves performing propagation and decision inference independently for when $x_i = 0$ and $x_i = 1$. If another variable x_j is inferred to have the same value assignment $\tau[x_j]$ in both cases, then it can be inferred that x_j should be set to $\tau[x_j]$

Algorithm 5.8: InferDecision(C)

Input: C - PB constraint with single term $a_i l_i$ where l_i is a literal of variable x_i
Output: assignment mapping of x_i if inferred

```
mapping ← [];
if C is equality then
    if a_i = k and k ≠ 0 then
        mapping ← mapLitTrue(C);
    else if k = 0 then
        mapping ← mapLitFalse(C);
else
    if (0 < k ≤ a_i and isPosLit(l_i)) or (a_i ≤ k < 0 and isNegLit(l_i)) then
        mapping ← mapLitTrue(C);
return mapping
```

Algorithm 5.9: AssumProbe(F, x_i)

Input: F - PB formula, x_i - assumption variable
Output: mapping of variable values

```
temp, mapping ← [];
forall constraint C ∈ F[x_i ↦ 1] do
    temp ← temp ∪ InferDecision(C);
forall constraint C ∈ F[x_i ↦ 0] do
    temp ← temp ∪ InferDecision(C);
forall variable x_j, where j ≠ i do
    if exactly one literal of x_j in temp then
        mapping ← mapping ∪ temp[x_j];
return mapping
```

in all satisfying assignments of F. Algorithm 5.9 illustrates the process for a single variable x_i, and in the preprocessing stage, we perform assumption probing on all variables in F.

5.3.2 Pseudo-Boolean Constraint Compilation

In this work, we introduce two approaches, namely top-down and bottom-up, to compile each constraint of a PB formula into an ADD. We use T, k, and eq in place of PB constraint C when describing the compilation algorithms. T refers to the term list, which is a list of $a_i x_i$ terms of C. k is the constraint constant and eq indicates if C is '=' constraint.

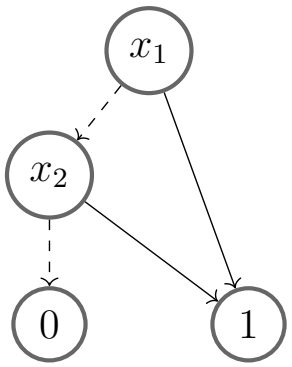


Figure 5.4: An ADD ψ_1 representing $3x_1 + 4x_2 \geq 3$

Bottom-up ADD Constraint Compilation In order to compile an ADD which represents a PB constraint of the following form $\sum_{i=1}^{n} a_i x_i [\geq, =, \leq] k$, we first start compiling the expression $\sum_{i=1}^{n} a_i x_i$ from literal and constant ADDs as shown by line 3 of Algorithm 5.10. A constant ADD which represents integer a_i is a single leaf node that has value a_i. A literal ADD comprises of an internal node, which represents variable x, and true and false leaf nodes, which represent the evaluated values of the literal if x is set to true and false. With the literal and constant ADDs, we use Apply with $\times$ operator to form ADDs for each term $a_i x_i$. We use Apply with $+$ operator on term ADDs to form the ADD representing expression $\sum_{i=1}^{n} a_i x_i$. As an example, the ADD ψ for the expression $3x_1 + 4x_2$ is shown in Figure 2.1. To account for the inequality or equality, we look at the value of leaf nodes in expression ADD ψ and determine if they satisfy the constraint (lines 4 to 10). We replace the leaf nodes with 1 node if the constraint is satisfied and 0 node otherwise, the resultant ADD is illustrated in Figure 5.4.

Top-down ADD Constraint Compilation In contrast to the bottom-up ADD compilation approach, the top-down ADD compilation for a given PB constraint involves the if-then-else (ITE) operation for decision diagrams. We only consider PB constraints that involve $=$ or $\geq$ as mentioned previously. The top-down compilation algorithm (Algorithm 5.11) makes use of recursive calls of Algorithm 5.12 to construct an ADD that represents a given PB clause. In particular, Algorithms 5.11 and 5.12 work by iterating through the terms of the PB constraint using *idx*. The algorithms build the sub-ADDs when the literal at position *idx* evaluates to *true* for the if-then case and otherwise for the else case of the ITE operation while updating the constraint

Algorithm 5.10: compileConstraintBottomUp(T, k, eq)

Input: T - term list, k - constraint value, eq - indicator if constraint is '='
Output: ψ - constraint ADD

```
1:  ψ ← constantADD(0);
2:  for term t in T do
3:  |  ψ += constantADD(t.coeff) × literalADD(t.literal);
4:  for node n in LeafNode(ψ) do
5:  |  if eq is true and n.value = k then
6:  |  |  n.value ← 1;
7:  |  else if eq is false and n.value ≥ k then
8:  |  |  n.value ← 1;
9:  |  else
10: |  |  n.value ← 0;
11: return ψ
```

constant k (lines 2-3 of Algorithm 5.11 and lines 9-10 of Algorithm 5.12). Notice that the top-down compilation approach allows for early termination when the current k value is negative for $\geq k$ case. However, early termination is possible only if all unprocessed coefficients are positive, implying that k in subsequent recursive calls cannot increase. One way would be to sort the term list T in ascending order of term coefficients, processing terms with negative coefficients before positive coefficients.

Algorithm 5.11: compileConstraintTopDown(T, k, eq)

Assumption: T is in ascending order of term coefficients or all coefficients are non-negative
Input: T - term list, k - constraint value, eq - indicator if constraint is '='
Output: ψ - constraint ADD

```
1: ψ ← literalADD(T[0].literal);
2: ψ_lo ← compileTDRecur(T, k, eq, 1);
3: ψ_hi ← compileTDRecur(T, k − T[0].coeff, eq, 1);
4: ψ.ITE(ψ_hi, ψ_lo);
5: return ψ
```

Optimizations for Bottom-up Compilation In the bottom-up compilation approach, an ADD is built from the individual literal and constant ADDs to represent the expression, before subsequently having leaf node values converted to 1 and 0 depending on if the PB constraint is satisfied. In the process, an ADD could be exponential in size with respect to the number of variables processed. In order to

Algorithm 5.12: compileTDRecur(T, k, eq, idx)

Input: T - term list, k - current constraint value, eq-input constraint equality, idx-index of current term in T
Output: ψ - constraint ADD from idx to end of T

```
1:  if T[idx].coeff ≥ 0 then
2:  |  isPos ← true;
3:  if eq and isPos and k < 0 then
4:  |  constantADD(0);
5:  else if !eq and isPos and k ≤ 0 then
6:  |  constantADD(1);
7:  else if idx < T.length then
8:  |  ψ ← literalADD(T[idx].literal);
9:  |  ψ_lo ← compileTDRecur(T, k, eq, idx + 1);
10: |  ψ_hi ← compileTDRecur(T, k − T[idx].coeff, eq, idx + 1);
11: |  return ψ.ITE(ψ_hi, ψ_lo)
12: else
13: |  if eq and k = 0 then
14: |  |  return constantADD(1)
15: |  else
16: |  |  return constantADD(0)
```

minimize the intermediate ADD during the compilation process, we introduce an optimization for bottom-up compilation. The key idea is to increase the number of shared sub-components of the intermediate ADD, and this amounts to processing the PB constraint terms in a manner that results in fewer distinct subset sums of term coefficients as every distinct subset sum requires a separate leaf node. To this end, we optimize the compilation process by sorting the terms according to the absolute values of their coefficients in ascending order. Subsequently, we manipulate the coefficients, using $x = (1 - \bar{x})$, of the terms such that alternate terms have coefficients of different signs. We show the pseudocode in Algorithm 5.13.

Algorithm 5.13: optimizeCompileBottomUp(T, k, eq)

Input: T - term list, k - constraint value, eq - input constraint equality
Output: ψ - constraint ADD

```
1: T ← sortAscendingAbsoluteCoeff(T);
2: T', k' ← makeEveryAltCoeffPos(T, k) ;          ▷ sign manipulation
3: return compileConstraintBottomUp(T', k', eq)
```

Optimizations for Top-down Compilation Similarly, we also introduce optimizations for the top-down compilation approach. Recall that one would only be able to perform early termination for PB constraints of the form $\sum a_i x_i \geq k$ after all negative coefficient terms have been processed. To this end, we manipulate all coefficients to be positive and adjust k accordingly so that early termination is possible. Furthermore, we sort the terms in descending value of the term coefficients as larger coefficients are more likely to satisfy the constraint. We show the pseudocode in Algorithm 5.14.

Algorithm 5.14: optimizeCompileTopDown(T, k, eq)

Input: T - term list, k - constraint value, eq - input constraint equality
Output: ψ - constraint ADD

```
1: T', k' ← makeAllCoeffPos(T, k) ;                    ▷ sign manipulation
2: T' ← sortDescendingCoeff(T);
3: return compileConstraintTopDown(T', k', eq)
```

Algorithm 5.15: compileConstraintDynamic(T, k, eq)

Input: T - term list, k - constraint value, eq-input constraint equality
Output: ψ - constraint ADD
Small k: T.length ≤ 25 **and** $k < 25^{th}$ percentile of T.coeff
Uniq Coeff: $k < 25^{th}$ percentile of T.coeff **and** unique coefficient rate $\geq$ 0.9 **and** unique adjacent difference rate ≥ 0.85

```
1: if Small k or Uniq Coeff then
2:     bottomUp ← false;
3: else
4:     bottomUp ← true;
5: if bottomUp then
6:     return optimizeCompileBottomUp(T, k, eq)
7: else
8:     return optimizeCompileTopDown(T, k, eq)
```

Dynamic Compilation A PB formula can include more than one PB constraint. As we will show in a case study in the experiments section, the choice of compilation approach has a substantial impact on overall runtime. To this end, we introduce a dynamic heuristic (Algorithm 5.15) to select the appropriate compilation approach and perform optimization of the compilation process as previously discussed. In Algorithm 5.15, we choose top-down compilation if either *Small k* condition or *Uniq*

Coeff is met. The two conditions are designed to be in favor of the botttom-up compilation approach, we provide performance analysis in the experiments section. The intuition for *Small k* condition is that top-down compilation might perform better when the constraint is relatively short, and that k is small relative to the coefficients of the constraint i.e. the iterative assignment of variables in Algorithms 5.11 and 5.12 will be able to terminate earlier. The intuition for *Uniq Coeff* condition is to use top-down compilation when bottom-up compilation might not be efficient. More specifically, bottom-up compilation might not be efficient when the ADDs have a lot of unique leaf nodes, indicating less shared sub-diagrams and potentially more expensive Apply operations in practice.

5.3.3 Pseudo-Boolean Model Counting with ADDs

We detailed the building blocks of PBCount in the prior parts of this chapter, and show the overall flow of PBCount in Figure 5.2. We also show the overall PB model counting algorithm of PBCount in Algorithm 5.16. PBCount starts by preprocessing the input PB formula F using the techniques introduced previously. Next, as mentioned in Section 5.2.1, PBCount follows the existing cluster ordering heuristics in ADDMC [DPV20a], (BOUQUET_TREE), which is based on Bouquet's method [Bou99]. The clusters, broadly speaking, are partitions of the constraints of F and can be viewed as subtasks in the dynamic programming paradigm. The computed cluster ordering can be viewed as a plan that determines how the constraint ADDs are merged and when variables can be projected out. Finally, PBCount computes the model counting using the ADD counting algorithm, Algorithm 5.6.

Algorithm 5.16: PBCount main algorithm

Input: F - PB formula
Output: Model Count of F

1: $F \leftarrow \mathsf{preprocess}(F)$;
2: $\rho \leftarrow \mathsf{getClusterOrdering}(F)$; ▷ Using bouquet tree heuristic
3: $\varphi \leftarrow [\,]$;
4: **for** all constraints c of F **do**
5: $\quad T, k, eq \leftarrow \mathsf{extract}(c)$; ▷ term list, constraint value, equality indicator
6: $\quad \psi \leftarrow \mathsf{compileConstraintDynamic}(T, k, eq)$;
7: $\quad \varphi.insert(\psi)$;
8: **return** $\mathsf{computeCount}(\varphi, \rho)$;

5.4 Supporting Projected and Incremental Model Counting

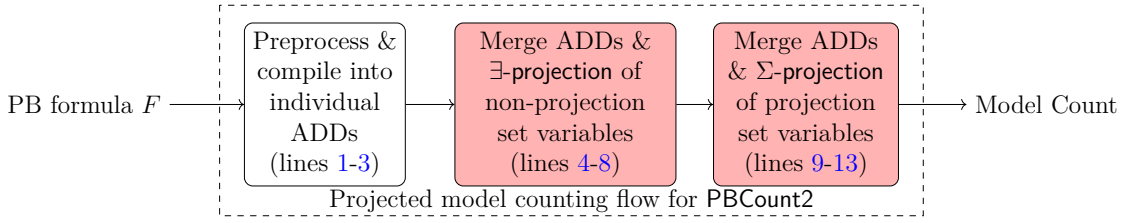


Figure 5.5: Overall flow of our projected model counter PBCount2. Shaded boxes indicate our contributions and white box indicates techniques adapted from existing works. Line numbers correspond to lines in Algorithm 5.17.

In this section, we detail our Least Occurrence Weighted Min Degree (LOW-MD) computation ordering heuristic as well as the implementations of PBCount2 to support a) projected model counting and b) incremental model counting.

5.4.1 Least Occurrence Weighted Min Degree

The general methodology of performing PB model counting tasks with ADDs involves representing each constraint with its individual ADD and subsequently merging the ADDs and projecting away variables to produce the final model count. As implemented in existing work [YM24], the ordering of merging ADDs and projecting away variables is determined by heuristics. However, the existing heuristics did not support projected model counting. To this end, we introduce a new ordering heuristic in PBCount2 in this work, which we term the *Least Occurrence Weighted Min Degree* heuristic (LOW-MD in short).

Our *Least Occurrence Weighted Min Degree* heuristic is as follows. Let G be an undirected bipartite graph, where the vertices either represent variables in PB formula F or a single PB constraint of F. A variable vertex v_x is connected to a constraint vertex v_c if the variable appears in that PB constraint. The LOW-MD heuristic entails picking the variable that has the corresponding variable vertex in G with the minimum degree. The heuristic is equivalent to a weighted version of the min-degree heuristic on Gaifman graphs, where weights correspond to the number of PB constraints in which two variables appear together.

Algorithm 5.17: PBCount2 model counting with LOW-MD heuristic

Input : PB formula F, Projection set X, Non-projection set Y
Output : Model count

$F \leftarrow \mathsf{preprocess}(F)$;
for all constraints c of F **do**
  Compile ADD for c;
while $Y \neq \emptyset$ **do**
  $\psi \leftarrow \mathsf{ADD}(1)$; ▷ ADD(1) returns ADD representing 1
  $y \leftarrow \mathsf{popNextVar}(Y)$;
  foreach ADD φ containing variable y **do** $\psi \leftarrow \psi \times \varphi$;
  $\psi \leftarrow \psi[y \mapsto 1] \vee \psi[y \mapsto 0]$; ▷ ∃-projection
while $X \neq \emptyset$ **do**
  $\psi \leftarrow \mathsf{ADD}(1)$;
  $x \leftarrow \mathsf{popNextVar}(X)$;
  foreach ADD φ containing variable x **do** $\psi \leftarrow \psi \times \varphi$;
  $\psi \leftarrow \psi[x \mapsto 1] + \psi[x \mapsto 0]$; ▷ Σ-projection
$\psi \leftarrow \mathsf{ADD}(1)$;
foreach remaining intermediate ADD φ **do** $\psi \leftarrow \psi \times \varphi$;
return ψ.value;

The intuition behind LOW-MD stems from the observation that the algorithmic complexity of merging two ADDs of size m, n is on the order of $\mathcal{O}(mn)$. As such, we would like to reduce the size of operand ADDs as much as possible, especially when the overall model counting algorithm involves many such ADD merging operations. In the computation process, the size of an ADD reduces when a variable is projected away. To ensure correctness, a variable can only be projected away when all ADDs involving it have been merged [DPV20a]. Hence, we designed our LOW-MD heuristic to pick the least frequently occurring variable to project away, as it involves merging the fewest number of ADDs before projecting away the variable.

5.4.2 Projected Model Counting

Recall that in projected model counting, there are two non-overlapping sets of variables X, Y where X is the projection set and Y is the non-projection set. The key idea to support projected model counting is the different way variables in X and Y are projected away. For all variables $x \in X$, we project x away from an ADD ψ using $\psi \leftarrow \psi[x \mapsto 1] + \psi[x \mapsto 0]$, also referred to as Σ-projection [DPV21]. In contrast, for all variables $y \in Y$, we project away y from ψ using $\psi \leftarrow \psi[y \mapsto 1] \vee \psi[y \mapsto 0]$, also

referred to as ∃-projection [DPV21]. In addition, all variables in Y must be projected away before any variable in X because the different projection operations are not commutative across variables in X and Y [DPV21]. To this end, we introduce the LOW-MD ordering which is compatible with the projected model counting ordering requirements. The overall flow of PBCount2 is shown in Figure 5.5, and the pseudocode is shown in Algorithm 5.17.

In Algorithm 5.17, we employ the same preprocessing (line 1) and individual constraint compilation techniques (line 3) as the original PBCount. Next, we process each variable y in non-projection set Y by merging all ADDs containing y and projecting away y from the merged ADD (lines 4-8). In lines 9-13, we do the same for variables in projection set X. In each iteration of the merge and project process, we select a variable using our LOW-MD heuristic, indicated by popNextVar$(\cdot)$ on lines 6 and 11, and remove it from X or Y respectively. As discussed previously, the LOW-MD ordering heuristic entails picking the variable that has the least occurrence in the ADDs at that moment of the computation.

Algorithm Correctness The algorithm correctness of projected model counting of PBCount2 follows prior work on projected CNF model counting with ADDs [DPV21]. Dudek, Phan, and Vardi showed that for projected CNF model counting correctness, the computations should be performed according to an X, Y-graded project join tree. In particular, performing ∃-projection at $\mathcal{I}_Y$ nodes and Σ-projection at $\mathcal{I}_X$ nodes of an X, Y-graded project join tree $\mathcal{T}$ produces the correct projected model count. PBCount2's algorithm correctness for projected PB model counting comes from the fact that the computation in Algorithm 5.17 with LOW-MD heuristic implicitly follows an X, Y-graded project join tree, and therefore produces the correct count.

Theorem 2. *Let F be a formula defined over $X \cup Y$ such that X is the projection set, and Y is the set of variables not in projection set, then given an instance (F, X, Y), Algorithm 5.17 returns c such that $c = \sum_{\beta \in 2^X} (\max_{\alpha \in 2^Y} [F](\alpha \cup \beta))$*

We show an adapted proof based on prior work [DPV21] for completeness. We introduce additional notations and definitions that are relevant to the proofs that follow. Let $\mathcal{T} = (T, r, \gamma, \pi)$ be a project join tree for PB formula F and $n \in \mathcal{V}(T)$ be a node in T. $S(n) \in \mathcal{V}(T)$ denotes the set of all descendants of n,

including itself. In other words, $S(n)$ is the set of vertices of the subtree rooted at n. Let $P(n) = \bigcup_{o \in S(n) \setminus \mathcal{L}(T)} \pi(o)$ be the set of variables projected in $S(n)$. Let $\Phi(n) = \{\gamma(l) : l \in \mathcal{L}(T) \cap S(n)\}$ be the set of PB constraints of F that are mapped to the leaf nodes of $S(n)$.

Definition 3 (**Projected Valuation**). *Let (F, X, Y) be a PB model counting instance, with projection set X, non-projection set Y, and PB formula F. Let $\mathcal{T}$ be an X, Y graded project join tree of F. The projected-valuation of each node $n \in \mathcal{V}(T)$, denoted g_n, is defined as:*

$$g_n = \begin{cases} [\gamma(n)] & \text{if } n \in \mathcal{L}(T) \\ \sum_{\pi(n)} (\prod_{o \in \mathsf{Ch}(n)} g_o) & \text{if } n \in \mathcal{I}_X \\ \exists_{\pi(n)} (\prod_{o \in \mathsf{Ch}(n)} g_o^W) & \text{if } n \in \mathcal{I}_Y \end{cases}$$

Where $[\gamma(n)]$ is the representation of PB constraint $\gamma(n) \in F$ and $\mathsf{Ch}(n)$ refers to the set of immediate child nodes of n in the project join tree.

Definition 4 (**Early Projection**). *Let X and Y be sets of variables. For all functions f defined over X and g defined over Y, $f : 2^X \to \mathbb{R}$, and $g : 2^Y \to \mathbb{R}$, if $x \in X \setminus Y$ then $\sum_X (f \cdot g) = (\sum_X f) \cdot g$ and $\exists_X (f \cdot g) = (\exists_X f) \cdot g$*

Lemma 1. *Let n be an internal node in a project join tree $\mathcal{T} = (T, r, \gamma, \pi)$. Let o and p be in $\mathsf{Ch}(n)$, and that $o \neq p$. Then $P(o) \cap \mathsf{Var}(\Phi(q)) = \emptyset$.*

Proof. Let x be a variable in $P(o)$, then $x \in \pi(s)$ for some internal node s that is a descendant of o. Suppose that x appears in some arbitrary PB constraint c of PB formula F. By definition of project join tree, the leaf node $\gamma^{-1}(c)$ must be a descendant of s and thus also a descendant of o. As o and q are sibling nodes, variable x does not appear in any descendant leaf node of q and $x \notin \mathsf{Var}(\Psi(q))$. Thus $P(o) \cap \mathsf{Var}(\Phi(q)) = \emptyset$. □

Lemma 2. *Let (F, X, Y) be a weighted projected model counting instance and $\mathcal{T}$ be an X, Y graded project join tree for F. For every node n of T, let*

$$h_n = \prod_{C \in \Phi(n)} [C]$$

Then

$$g_n = \sum_{P(n)\cap X} \exists_{P(n)\cap Y} h_n \tag{5.1}$$

Proof. Let n be an arbitrary node in $\mathcal{V}(T)$.

Case 1: Suppose n is a leaf node, that is $n \in \mathcal{L}(T)$.
If n is a leaf node, then $\Phi(n) = \{\gamma(n)\}$ and $P(n) = \emptyset$. By definition of h_n, $h_n = \prod_{C\in\{\gamma(n)\}}[C] = [\gamma(n)]$. It follows that $\sum_{P(n)\cap X} \exists_{P(n)\cap Y} h_n$ becomes $\sum_{\emptyset} \exists_{\emptyset} h_n = [\gamma(n)]$. Notice that this is exactly g_n by Definition 3, for the case where $n \in \mathcal{L}(T)$.

Case 2: Now we look at the case where n is an internal node of T, that is $n \in \mathcal{V}(T) \setminus \mathcal{L}(T)$.
Suppose for each child node $o \in \mathsf{Ch}(n)$, $g_o = \sum_{P(o)\cap X} \exists_{P(o)\cap Y} h_o$. Then if we take the valuation product, and substituting Equation 5.1, we have

$$\prod_{o\in\mathsf{Ch}(n)} g_o = \prod_{o\in\mathsf{Ch}(n)} \sum_{P(o)\cap X} \exists_{P(o)\cap Y} h_o \tag{5.2}$$

Since for $o, q \in \mathsf{Ch}(n)$ and $o \neq q$ $P(o) \cap \mathsf{Var}(\Phi(q)) = \emptyset$ by Lemma 1, then $P(o) \cap \mathsf{Var}(h_q) = \emptyset$. As a result, by the applying early projection according to Definition 4, we have

$$\prod_{o\in\mathsf{Ch}(n)} g_o = \sum_{A\cap X} \prod_{o\in\mathsf{Ch}(n)} \exists_{P(o)\cap Y} h_o = \sum_{A\cap X} \exists_{A\cap Y} \prod_{o\in\mathsf{Ch}(n)} h_o \tag{5.3}$$

where A is $\bigcup_{o\in\mathsf{Ch}(n)} P(o)$.

Recall that the set of internal nodes, $\mathcal{V}(T) \setminus \mathcal{L}(T)$, of an $\mathcal{T}$ is partitioned by $\mathcal{I}_X$ and $\mathcal{I}_Y$, definition of X, Y-graded project join tree. As such, there are two subcases, where n is $\mathcal{I}_X$ or $\mathcal{I}_Y$.

Case 2a: $n \in \mathcal{I}_Y$. For each $p \in S(n)$, $p \in \mathcal{I}_Y$ and $\pi p \subseteq Y$ by definition of X, Y-graded project join tree. As such $\bigcup_{o\in\mathsf{Ch}(n)} P(o)$ or A is a subset of Y. Thus by Definition 3 and Equation 5.3,

$$g_n = \exists_{\pi(n)} \prod_{o\in\mathsf{Ch}(n)} g_o = \exists_{\pi(n)} \exists_{A} \prod_{o\in\mathsf{Ch}(n)} h_o = \exists_{P(n)} \prod_{o\in\mathsf{Ch}(n)} h_o$$

Therefore,

$$g_n = \exists_{P(n)} \prod_{o\in\mathsf{Ch}(n)} \prod_{C\in\Phi(o)} [C] = \exists_{P(n)} \prod_{C\in\Phi(n)} [C] = \exists_{P(n)} h_n$$

Case 2b: $n \in \mathcal{I}_X$. Notice that $\pi(n) \subseteq X$. Using Definition 3 and Equation 5.3, we have

$$g_n = \sum_{\pi(n)} \prod_{o \in \mathsf{Ch}(n)} g_o = \sum_{\pi(n)} \left(\sum_{A \cap X} \exists_{A \cap Y} \prod_{o \in \mathsf{Ch}(n)} h_o \right)$$

Notice that $\pi(n) \cup A = P(n)$ and also $h_n = \prod_{o \in \mathsf{Ch}(n)} h_o$, therefore

$$g_n = \sum_{P(n) \cap X} \exists_{P(n) \cap Y} \prod_{o \in \mathsf{Ch}(n)} h_o = \sum_{P(n) \cap X} \exists_{P(n) \cap Y} h_n$$

□

Lemma 3. *The computation process in Algorithm 5.17 follows an X, Y graded project join tree to compute projected valuation at each node.*

Proof. We split the proof into two parts, first showing by construction that Algorithm 5.17 follows an X, Y graded project join tree. Subsequently, we show that the computation at each node is exactly according to the definition of projected valuation.

Let T be a tree, with leaf nodes having a bijective relationship with the individual ADDs at the start of computation of Algorithm 5.17 (line 3). Let each merged intermediate ADD ψ at lines 8 and 13 map to an internal node of T, with the internal node's descendants being all the ADDs (denoted φ) involved in the merging process at line 7 and 12 respectively. In addition, let the variable projected away in the merge and project iteration be the label of the respective internal node given by labelling function π. The root node of T would correspond to the final merged ADD ψ at line 15 of Algorithm 5.17.

Recall that an X, Y-graded project join tree $\mathcal{T}$ has two disjoint sets of internal nodes $\mathcal{I}_X$ and $\mathcal{I}_Y$, of X grade and Y graded respectively. The structural requirements of X, Y-graded project join tree is such that nodes in $\mathcal{I}_Y$ cannot have any node of $\mathcal{I}_X$ as descendant in $\mathcal{T}$. We let internal nodes of T mapped in line 7 of Algorithm 5.17 to be of grade I_Y and the internal nodes mapped in line 12 to be of grade I_X. Notice that we labelled all internal nodes, hence $\{\mathcal{I}_X, \mathcal{I}_Y\}$ is a partition of all internal nodes, satisfying property 1 of definition of X, Y-graded project join tree. It is clear that properties 2 and 3 of X, Y-graded project join tree are satisfied as the π labels each internal node n to the variables projected away from it.

Since all internal nodes of grade I_Y are produced before I_X by design of Algorithm 5.17, none of the internal nodes of $\mathcal{I}_X$ would be descendants of internal nodes in $\mathcal{I}_Y$, satisfying property 4 of X, Y-graded project join tree.

Recall that a project join tree's labelling function π should label each internal node with a set of variables such the different labels of internal nodes partition the set of variables of the original PB formula F. For each variable v, notice that Algorithm 5.17 only projects away v once, after merging the corresponding ADDs containing v. Hence, there is no repetition of labels, and the labels of all internal nodes partitions $X \cup Y$.

In addition, an X, Y-graded project join tree has leaf nodes that has one to one mapping γ to constraints in the original formula. In T, recall that each leaf node is one-to-one mapped to an individual ADD at the start of Algorithm 5.17. Since each individual ADD directly represents a constraint in the original PB formula, leaf nodes of T have a bijection to constraints in the PB formula by transitivity. It is clear that property 2 of project join tree holds, because at each merge and project iteration, we merge all ADDs containing the selected variable (lines 7 and 12).

Since tree T arising from the computation process of Algorithm 5.17 meets the specifications of an X, Y-graded project join tree, the computation process of Algorithm 5.17 indeed follows T, an X, Y-graded project join tree.

Notice that at each leaf node $n \in \mathcal{L}(T)$, we have the representation of a constraint $[\gamma(n)]$, which is the individual ADD at the start of Algorithm 5.17. At each node $n \in \mathcal{I}_Y$, we merge the ADDs of descendants using the Apply operation with $\times$ operator and $\exists$-projection away variables labelled at n on line 8, computing $\exists_{\pi(n)}(\prod_{o \in \mathsf{Ch}(n)} g_o)$. Similarly at each node $n \in \mathcal{I}_X$, we again merge ADDs of descendants in the same manner but Σ-projection away variables on line 13, computing $\sum_{\pi(n)}(\prod_{o \in \mathsf{Ch}(n)} g_o)$. Since Algorithm 5.17 computations follow Definition 3 at all nodes, it computes g_n.

Thus, we showed that the computation process in Algorithm 5.17 follows an X, Y graded project join tree to compute projected valuation at each node. □

We restate Theorem 2 for convenience and provide the proof as follows.

Theorem 2. *Let F be a formula defined over $X \cup Y$ such that X is the projection set, and Y is the set of variables not in projection set, then given an instance (F, X, Y), Algorithm 5.17 returns c such that $c = \sum_{\beta \in 2^X}(\max_{\alpha \in 2^Y}[F](\alpha \cup \beta))$*

Proof. Let the root node of $\mathcal{T}$ be r. Notice that because r is the root node, $\Phi(r) = F$ and $P(r) = X \cup Y$. By Lemma 3, the computation process of Algorithm 5.17 follows an X, Y graded project join tree $\mathcal{T}$.

Using Lemma 2,

$$g_r = \sum_X \exists_Y h_r = \sum_X \exists_Y \prod_{C \in F} [C] = \sum_X \exists_Y [F]$$

Notice that $\sum_X \exists_Y [F]$ is exactly $\sum_{\beta \in 2^X} (\max_{\alpha \in 2^Y} [F](\alpha \cup \beta))$. Hence, g_r is the Y-projected count of F and Algorithm 5.17 produces the count $c = \sum_{\beta \in 2^X} (\max_{\alpha \in 2^Y} [F](\alpha \cup \beta))$. □

5.4.3 Incremental Model Counting

Algorithm 5.18: Cache retrieval of PBCount2

Input : PB formula F'
Output : Compute state - set of ADDs retrieved from cache

1: $C' \leftarrow$ getConstraintSet(F');
2: $\mathcal{A}, \mathcal{B} \leftarrow \emptyset$; ▷ Initialize 2 empty sets
3: **for** each ADD ψ in cache **do**
4: | **if** ψ.constraints $\subset C'$ and CheckNoExtraVar(ψ, C') **then** insert ψ into $\mathcal{A}$;
5: **for** each ADD $\psi_{\mathcal{A}} \in \mathcal{A}$ **do**
6: | conflicts $\leftarrow$ *false*;
7: | **for** each ADD $\psi_{\mathcal{B}} \in \mathcal{B}$ **do**
8: | | **if** there exists x which is projected away in $\psi_{\mathcal{A}}$ and not $\psi_{\mathcal{B}}$, or vice versa **then**
9: | | | conflicts $\leftarrow$ *true*;
10: | **if** not conflicts **then** insert $\psi_{\mathcal{A}}$ into $\mathcal{B}$;
11: **for** all $c \in C'$ **do**
12: | **if** c not represented in $\mathcal{B}$ **then** insert constraint ADD of c into $\mathcal{B}$;
13: **return** $\mathcal{B}$;

PBCount2 supports incremental PB model counting via the caching of intermediate ADDs during the handling of model count queries. In particular, PBCount2 supports the removal and addition of constraints in the PB formula. With reference to Algorithm 5.17, we cache the ADDs ψ at lines 8 and 13 respectively. In order to store the compute state associated with an ADD, for cache retrieval purposes, we store 3 pieces of information: 1) the set of constraints in PB formula that the ADD

represents 2) the projection set variables of ADD that have been projected away and 3) the non-projection set variables of ADD that have been projected away.

The cache retrieval mechanism is shown in Algorithm 5.18. When given PB formula F', modified from F by adding or removing constraints, the core idea is to loop through the ADDs in the cache and retrieve the compatible ADDs, replacing the initial ADDs at line 3 of Algorithm 5.17. An ADD ψ is compatible with F' if the set of constraints that ψ represents is a subset of the constraints of F'. In the case where a constraint was removed, the corresponding ADDs will not be considered, as they contain constraints not in F'. In addition, the variables that have been projected out from ψ must not appear in any constraint of F' that ψ does not represent, this is handled by CheckNoExtraVar$(\cdot)$ in Algorithm 5.18 line 4. Subsequently, from lines 5-10, we verify that each ADD that we retrieved is compatible with all other already retrieved ADD candidates in $\mathcal{B}$. Finally, for all constraints that are not represented by an ADD in $\mathcal{B}$, we insert an ADD representing each constraint into $\mathcal{B}$. Cache retrieval replaces lines 1 to 3 of Algorithm 5.17. It is worth noting that caching ADDs requires us to disable preprocessing currently, as there is a need to maintain a unique id for each constraint and also a fixed variable-to-constraint relation. The restriction arises from the fact that we have to maintain ADD to variable mapping for ADDs in the cache to perform retrieval compatibility checks in Algorithm 5.18. Preprocessing might remove variables and modify constraints, thus invalidating cached ADDs. Hence we have currently disabled preprocessing when handling incremental counting.

5.5 Top-Down Model Counting

In this section, we explore the top-down PB model counter design for completeness, in context of PB model counting. We detail the general algorithm of our top-down PB model counter PBMC as well as design decisions that enable a performant PB counter.

5.5.1 Search-Based Algorithm

The overall counting approach of PBMC is shown in Algorithms 5.19 and 5.20. CountPBMC starts with preprocessing the input PB formula F with techniques

Algorithm 5.19: CountPBMC – Counting Algorithm of PBMC

Input : PB formula F
Output : Model count

1: $F \leftarrow \mathsf{Preprocess}(F)$;
2: component $\varphi \leftarrow F$; cache $\zeta \leftarrow \emptyset$; assignment $\tau \leftarrow \emptyset$;
3: $\mathsf{Count}(\varphi, \tau, \zeta)$;
4: **return** $\zeta[\varphi]$;

adapted from existing state-of-the-art counters [YM24]. Subsequently, Count sets the preprocessed F as the root component, and initializes an empty cache ζ and an empty assignment τ. CountPBMC calls Count to search the space of assignments and compute the model count. The main computation process of PBMC in Count comprises of the following steps – propagation (line 2), conflict handling (lines 3–7), variable decision (lines 11–13), and component decomposition (line 9). As shown in Count, PBMC starts with propagation so that variable decisions can be inferred and applied. If the propagation leads to a conflict, PBMC would perform conflict analysis and backjumping using similar techniques as RoundingSat [EN18] (lines 5–6). If the conflict is a top-level conflict, which cannot be resolved by backjumping, PBMC will terminate and return 0 as the formula is unsatisfiable (line 7). Otherwise, PBMC learns a PB constraint that prunes the search space, backjump to the appropriate decision level d, and clear components and recursive calls created after d.

If there are no conflicts, PBMC will attempt to split the current component into variable-disjoint child components, and the count of the current component would be the product of child component counts (lines 9–10). There are two cases of component splitting – (i) there are no new disjoint components and (ii) there are new disjoint components. In case (i) PBMC will proceed to branch on another variable, and the count of the current component would be the sum of counts from the two branches (lines 12–13). If there are no unassigned variables remaining, the count of the current component would be 1 (line 14). Finally, after completing the implicit search tree exploration PBMC returns the count of F in cache ζ.

Conflict Analysis and Backjump We provide a brief overview of the conflict analysis and backjumping mechanism (on lines 5 and 6 of Algorithm 5.20), adapted from RoundingSat [EN18]. On a high level, the Backjump() mechanism backtracks

Algorithm 5.20: Count – helper function of CountPBMC

```
Input  : component φ, assignment τ, cache ζ
if φ entry in ζ then return ζ[φ];
isConf ← propagate(τ);
if isConf then
    ζ[φ] ← 0;
    success ← ConflictAnalysis(τ) ;
    if success then Backjump()   ▷ Clears related cache & recursive calls ;
    else ExitReturnZero()        ▷ Unresolvable conflict, return 0 ;
else
    C ← SplitComponent(φ, τ) ;
    if |C| > 1 then ζ[φ] ← ∏_{c∈C} Count(c, τ, ζ);
    else if φ has unassigned variable then
        l ← pickNextLit(φ, τ);
        ζ[φ] ← Count(φ ∧ l, τ ∪ {l}, ζ) + Count(φ ∧ l̄, τ ∪ {l̄}, ζ) ;
    else ζ[φ] ← 1 ;
return ζ[φ]
```

the variable assignments in chronological order to a state where the learnt constraint is asserting i.e. when the learnt constraint will cause a propagation. The conflict analysis process will determine the variable decision level to backjump to. The conflict analysis mechanism starts with the constraint that is in conflict, as well as the reason constraint that resulted in the conflict-causing propagation. The conflict analysis process involves applying the RountToOne routine introduced in RoundingSat [EN18] to both the conflict constraint and reason constraint. The RountToOne routine of a constraint outputs a weakened constraint that has some literals removed and right-hand side k value adjusted accordingly. The next step would be to resolve the two aforementioned constraints on the last assigned literal in the current partial assignment to get an intermediate constraint. Subsequently the last assignment is undone and the process repeats until the repeatedly updated intermediate constraint is asserting, at which point it is the learnt constraint.

5.5.2 VCIS Variable Decision Heuristic

In our top-down PB model counter PBMC, we introduce a variable decision ordering heuristic that we term *variable coefficient impact score* (VCIS). The intuition for a new variable decision heuristic for PB formulas came from the fact that the

coefficient of each literal affects its impact on the overall counting process. Existing variable decision heuristics from CNF model counting literature do not have to consider the impact of coefficients, and therefore modifications are required to adapt to PB model counting. Additionally, tree decomposition may also be less effective when dealing with PB constraints that are relatively longer or constraints that share a lot of variables as in the case of multi-dimensional knapsack problems.

In our VCIS heuristics, we add a coefficient impact score for each variable as an equal-weighted additional component to prior variable decision heuristics adopted from GPMC. The coefficient impact score for a variable x is given by $\left(\sum_{j\in\Omega_x} b_x^j/k_j\right) \div |\Omega_x|$ where Ω_x is the set of constraints that variable x appears in, b_x^j is the coefficient of x in constraint j, and k_j is the degree of constraint j. At the start of the counting process, the score for each variable is computed, and variables are decided in descending order of scores in the counting process. In addition, the phase preferences of variable decisions are set such that the term coefficient with the largest coefficient impact score is applied, this is reflected in line 12 of Algorithm 5.20. The intuition is that branching on a variable with a larger score would have a greater impact on gaps than that on a variable with a smaller score.

5.5.3 Caching Scheme and Optimization

After computing the count of each component, we store it in the cache to avoid redundant computations when the same component is encountered in a different branch of the counting process. A component in our caching scheme contains the following information for unique identification – 1) the set of unassigned variables in the component, 2) the set of yet-to-be-satisfied PB constraints in the component, and 3) their current gaps. In our implementation, we stored variables and constraints of a component by their IDs.

Lemma 4. *Given a PB formula F and partial assignments τ and τ'. Let component c be that of $F|_\tau$ and c' be that of $F|_{\tau'}$. If $VarIDs(c) = VarIDs(c')$, $CstrIDs(c) = CstrIDs(c')$, and each constraint ω in c satisfies $gap(\omega, \tau) = gap(\omega, \tau')$, then $c = c'$.*

Proof. It is clear that $Vars(c) = Vars(c')$. For each constraint ω with $CstrID(\omega) \in CstrIDs(C)$, ω_τ and $\omega_{\tau'}$ appear in c and c', respectively. As $Vars(c) = Vars(c')$,

the left-hand side of ω_τ is the same as that of $\omega_{\tau'}$. Then we know $\omega_\tau = \omega_{\tau'}$ as $gap(\omega, \tau) = gap(\omega, \tau')$. □

In our implementation, we cache the variable and constraint IDs in ascending order, thereby only needing to record the first ID and the differences between each i^{th} and $i + 1^{th}$. Since a difference is often smaller than an ID, we can use fewer bits to record each component. Since each gap s is positive, we can record it as $s - 1$. Additionally, we do not record gap for a clausal constraint, that is a constraint with coefficients and degree 1. Such a scheme is particularly efficient for the constraints with a lot of literals.

As a cache entry optimization trick to increase the cache hit rate, we perform saturation on the gaps during the generation of component cache IDs. The intuition for gap saturation comes from existing literature on PB solving where saturation is performed on the coefficients of a PB constraint [EN18] i.e. a term ax in the PB constraint is modified to kx where k is the degree when $a > k$. Given a PB constraint $\sum_{j=1}^{m} a_j x_j \geq s$ where s is the gap, our caching scheme changes s in the cache ID to $\min a_j$ over all coefficients a_j of unassigned variables when $0 < s < \min a_j$. This is sound as assigning any of the remaining literals to true would lead to the constraint being satisfied which is the same outcome without modification of s i.e. the resulting gap being 0 or negative.

5.6 Experiments

We performed extensive empirical evaluations to understand the performance impact of our attempt to use alternative logic form for the model counting task. More specifically, we evaluate PBCount to understand the main impact of introducing preprocessing and ADD compilation methods for PB formula input. Subsequently, we evaluate PBCount2 which extends PBCount with projected and incremental counting features via our LOW-MD heuristic and caching mechanism. In addition, we include PBMC in the evaluations to understand the performance of different design paradigms in context of PB model counting. Our empirical evaluation focuses on benchmarks arising from three application domains: sensor placement, auctions, and multi-dimensional knapsack. In particular, we measure the amount of benchmark

instances that each approach could complete, i.e. the number of instances each approach could return counts for in the case of PB model counting.

Through our evaluations and analysis, we sought to answer the following research questions:

RQ 1 How does the runtime performance of PBCount compare to that of the state-of-the-art baselines?

RQ 2 How does the dynamic compilation approach impact the runtime performance of PBCount?

RQ 3 How does PBCount2 perform on projected PB model counting?

RQ 4 How does PBCount2 perform under incremental settings?

RQ 5 How does the top-down PB counter PBMC compare to existing bottom-up PB counters PBCount2 and PBCounter?

RQ 6 How does the VCIS variable decision heuristics impact the runtime performance of PBMC?

Setup We performed our evaluations across machines with AMD EPYC 7713 and 7763 processors, with each set of comparisons performed on machines with the same configuration. Each benchmark instance is provided with 1 core, 16GB memory, and a timeout of 3600 seconds. We implemented our bottom-up PB counter prototypes PBCount and PBCount2 in C++ using the CUDD [Som15] library and double precision, due to the limitations of CUDD not supporting arbitrary precision ADDs. We implemented our top-down PB counter prototype PBMC in C++ with GMP, MPFR, Boost library, and code adapted from RoundingSat and GPMC. Since all the state-of-the-art exact model counters take CNF as input, we employed the CNF model counters with the help of PB to CNF conversion tool PBLib[1] [PS15]. We first evaluated PBCount against state-of-the-art projected counters DPMC, D4[2], and GPMC in **RQ 1** and **RQ 2**. D4 and GPMC are among the winners of the Projected counting track at Model Counting Competition 2022 and 2023. Subsequently in

[1]We used the provided PBEncoder for conversion.
[2]Binary from Model Counting Competition 2022

RQ 3 and **RQ 4**, we evaluated PBCount2 against D4 and GPMC as DPMC was not competitive in the previous comparison. Finally, to have a complete picture of the performance of the two major counter paradigms in context of PB model counting, we evaluated PBMC against existing state-of-the-art PB counters PBCount2, PBCounter, and model counting competition 2024 winner Ganak in **RQ 5** and **RQ 6**.

Benchmarks We generated 3500 benchmarks of the following application areas – sensor placement, auctions, and multi-dimension knapsack. We detail the benchmark statistics (number of variables and constraints) in the Tables 5.1, 5.2 and 5.3.

- The sensor placement benchmark setting (1473 instances after removal of 0 counts from the 1500 generated instances) is adapted from prior work on identifying code sets [LSM23]. Given a network graph, a maximum number of sensors allowed, count the number of ways to place sensors such that failures in the network are uniquely identifiable.
- For the auction benchmark setting (1000 instances), we adapt the combinatorial auction setting [BN07] to a counting variant. There are m participants and n items, each of which can be shared by one or more participants. Given that each participant has a minimum utility threshold, we count the number of ways the n items can be shared such that all participants achieve their minimum threshold. The utilities are additive and can be negative.
- For the multi-dimension knapsack benchmark setting [GL80] (1000 instances), there are n items and constraints on m different features or dimensions of the items in the form of the sum of each dimension should not exceed a given constant. Given such a setting, the goal would be the count the number of subsets of items that satisfy the constraints.

Table 5.1 shows the benchmark statistics for the auction setting. We show the minimum number, 25^{th} percentile, median, 75^{th} percentile, and max value for the number of variables, constraints, and clauses for both PB formula and converted CNF formula. The '# PB variable' and '# PB constraint' columns show statistics for the number of variables and constraints in the 1000 PB formulas (each PB formula is one benchmark), on median ('50%' row) the PB benchmarks have 91

Statistics	# PB variable	# PB constraint	# CNF variable	# CNF clause
Min	5	1	5	1
25%	60	4	15830	28456
50%	91	9	41900.5	77265.5
75%	131	14	80270	147022.75
Max	252	20	234777	474900

Table 5.1: Statistics of the number of variables, number of constraints, and number of clauses for PB and CNF formula in the auction benchmarks

Statistics	# PB variable	# PB constraint	# CNF variable	# CNF clause
Min	5	1	28	38
25%	86.75	6	7866	14871.25
50%	164	10	25212	46340
75%	234	15	50670.5	92525.25
Max	300	20	171632	314821

Table 5.2: Statistics of the number of variables, number of constraints, and number of clauses for PB and CNF formula in the $\mathcal{M}$-dim knapsack benchmarks

Statistics	# PB variable	# PB constraint	# CNF variable	# CNF clause
Min	1	2	1	1
25%	25	291	158	539.75
50%	81.5	1210	1154	3745.5
75%	190	7056.5	3932	12902.5
Max	300	44552	7699	55514

Table 5.3: Statistics of the number of variables, number of constraints, and number of clauses for PB and CNF formula in the sensor placement benchmarks

variables and 9 constraints. Similarly, the numbers for the converted CNF formulas are shown under '# CNF variable' and '# CNF clause' columns. The CNF version of the benchmarks have on median 41900.5 variables and 77265.5 clauses, which is 4 to 5 orders of magnitude larger than that of the PB version, this strongly supports our claim that PB formulas are more succinct than CNF formulas. The observation also holds for $\mathcal{M}$-dim knapsack benchmarks (multidimension knapsack) which we show the stats for in Table 5.2.

However, the difference is not as large in sensor placement benchmarks, because the coefficients and k values are predominantly 1 and -1 except for the budget constraint which indicates the maximum number of sensors one could place. As

such the sensor placement benchmarks are also more amenable to CNF encodings, unlike auction and knapsack benchmarks with different coefficients. We show the benchmark statistics for sensor placement settings in Table 5.3.

Projected Counting and Incremental Counting Benchmarks In projected PB counting and incremental counting evaluations, we use the same aforementioned benchmarks. In projected PB counting evaluations, the projection set is randomly selected from the original variables. The CNF counters use benchmarks converted using PBLib [PS15], with the same projection set. For incremental counting evaluations, each benchmark involves computing the model count for a given PB formula (step 1) and each of the 2 (4) subsequent modification steps for 3-step (5-step) configurations. Each modification step involves modifying an existing constraint, selected at random from the set of constraints of the PB formula. The modifications for the auction benchmarks correspond to updates to utility values, i.e. when preferences change. The modifications for the knapsack benchmarks correspond to changes to constraints of a particular dimension. Finally, the modifications to the sensor placement benchmarks correspond to additional requirements for redundancy of sensors at important locations in the graph. We provided each step of the incremental benchmarks as separate instances to competing approaches, as none of them supported incremental counting.

5.6.1 RQ1: PB Model Counting Performance

We first look at the runtime performance of PBCount, where the core idea is in the ADD compilation methodology for PB constraints, allowing for native PB model counting.

We show the cactus plot of the number of instances completed by each counter out of the 3500 benchmarks in Figure 5.6. The exact number of instances completed by each counter for each benchmark set is shown in Table 5.4. Additionally, we provide individual cactus plots for each set of benchmarks in the Figure 5.7.

In sensor placement benchmarks, PBCount count completed 638 instances, narrowly ahead of DPMC (625 instances), and more than D4 (566 instances) and GPMC (575 instances). In multi-dimension knapsack ($\mathcal{M}$-dim knapsack) and auction benchmarks, PBCount significantly outperforms the competing counters. PBCount

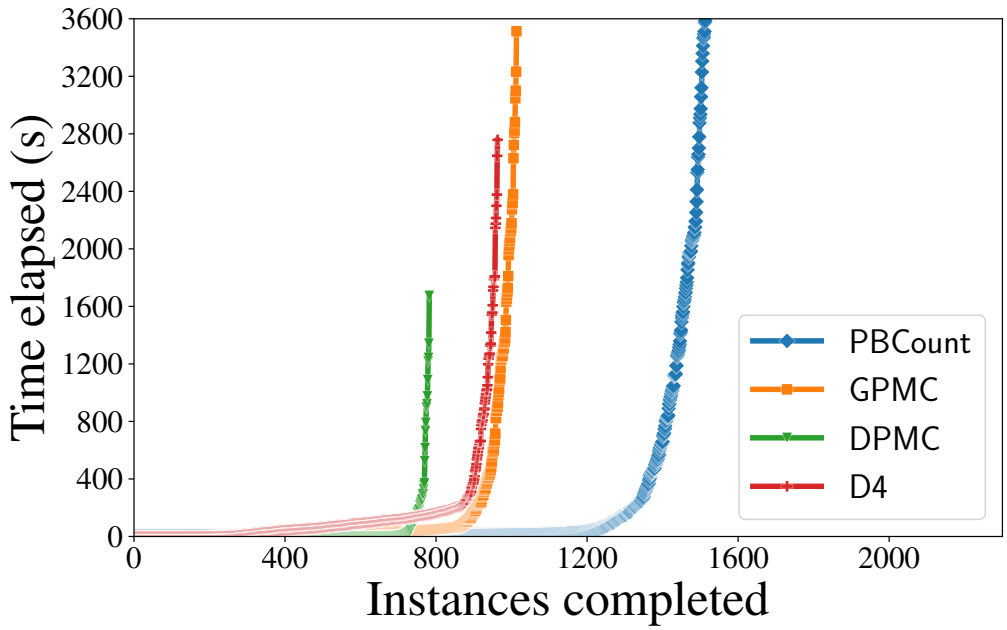
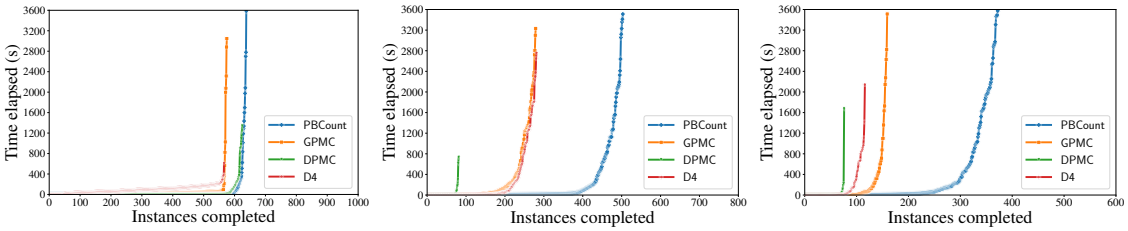


Figure 5.6: Cactus plot of number of benchmark instances completed by different counters. A point (x, y) on each line plot indicates the corresponding counter completes x number of benchmarks after y seconds has elapsed.



(a) Sensor placement (b) $\mathcal{M}$-dim Knapsack (c) Auction

Figure 5.7: Cactus plots of different benchmark sets. A point (x, y) on each line plot indicates the corresponding counter completes x number of benchmarks after y seconds has elapsed.

Benchmarks	DPMC	D4	GPMC	PBCount
Sensor placement	625	566	575	**638**
$\mathcal{M}$-dim knapsack	81	281	279	**503**
Auction	76	116	159	**372**
Total	782	963	1013	**1513**

Table 5.4: Number of benchmark instances completed by each counter in 3600s, higher is better. Excluding 0 count benchmark instances.

completed 503 $\mathcal{M}$-dim knapsack instances, around 1.8× that of GPMC (279 instances) and D4 (281 instances), and 6.2× that of DPMC (81 instances). In auction benchmarks, PBCount completed 372 instances, around 2.3× that of GPMC (159 instances), 3.2× of D4 (116 instances), and 4.9× of DPMC (76 instances). Overall, PBCount completed 1513 instances out of 3500 total instances, around 1.5× that of GPMC, 1.6× of D4, and 1.9× that of DPMC. Note that PBCount achieved superior performance with minimal preprocessing over GPMC, which has advanced preprocessing capabilities.

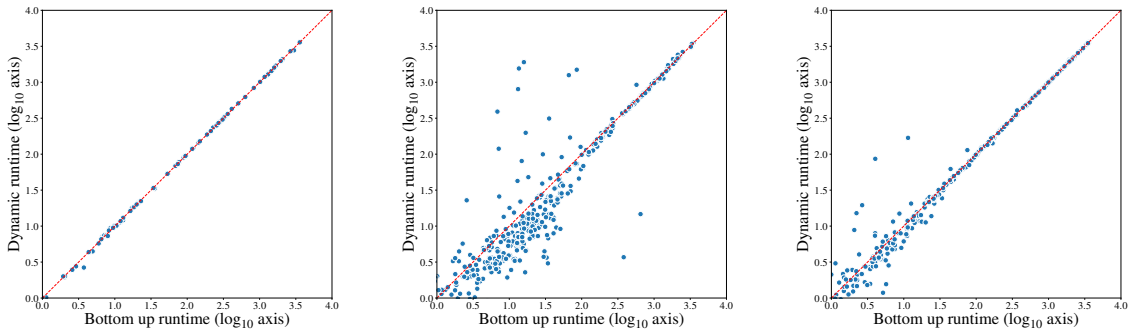
5.6.2 RQ2: Analysis of ADD Compilation Approaches

We now focus on the analysis of different compilation approaches: top-down (Algorithm 5.11), bottom-up (Algorithm 5.10), and dynamic (Algorithm 5.15). The results in Table 5.5 show that for the benchmarks, bottom-up PB constraint compilation outperforms top-down approach significantly in auction and multi-dimension knapsack and to a lesser degree sensor placement. In addition, the evaluation result also highlights that our dynamic compilation heuristic and constraint term optimization closely match the bottom-up approach, with the exception of completing 3 fewer instances in auction benchmarks. However, in the 372 auction instances completed by both bottom-up and dynamic approaches, the dynamic approach with term coefficient optimization completes the counting task faster for 257 instances. We show the individual scatter plots for each benchmark set between dynamic and bottom-up approach in Figure 5.8.

Benchmarks	Top-down	Bottom-up	Dynamic
Sensor placement	580	638	638
$\mathcal{M}$-dim knapsack	109	503	503
Auction	158	375	372

Table 5.5: Number of benchmarks completed by PBCount when employing different compilation strategies, higher number indicates better performance.

Compilation Approach Performance Case Study We provide an example to highlight the performance impact of the choice of compilation approach. The example involves the following PB formula in Equation 5.4 with a single constraint



(a) Sensor placement (b) $\mathcal{M}$-dim Knapsack (c) Auction

Figure 5.8: Scatter plots of runtimes of different benchmark sets between bottom-up and dynamic compilation approaches. Points beneath red diagonal line indicate dynamic compilation is faster, points above otherwise.

that has unique term coefficients:

$$\sum_{i=0}^{12} 2^i x_{i+1} + \sum_{i=1}^{10} 3^i x_{i+13} + \sum_{i=1}^{7} 7^i x_{i+23} \geq k \tag{5.4}$$

We vary the value of k in the above PB constraint from 10^1 to 10^5 and compare the runtime between top-down and bottom-up compilation approaches in Table 5.6. Note that bottom-up compilation takes around the same time irrespective of k as there is no early termination. On the other hand for top-down compilation, the PB constraint is easily satisfied when k is small and thus allows for early termination, leading to significant time savings compared to when k is large. Notice that when top-down compilation is unable to terminate early, it is much slower than bottom-up compilation even when all term coefficients are unique.

Approach	k value				
	10^1	10^2	10^3	10^4	10^5
Top-down	0.005	0.009	0.228	8.586	46.071
Bottom-up	6.927	7.202	7.198	7.434	6.732

Table 5.6: Runtime (seconds) to complete model counting for formula in Equation 5.4. Lower is better

As mentioned previously, bottom-up compilation benefits from having large numbers of same term coefficients or collisions in subset sums of coefficients. To this end, we changed all term coefficients of the PB constraint in equation 5.4 to 1 and compared runtimes in Table 5.7. We observed around three orders of

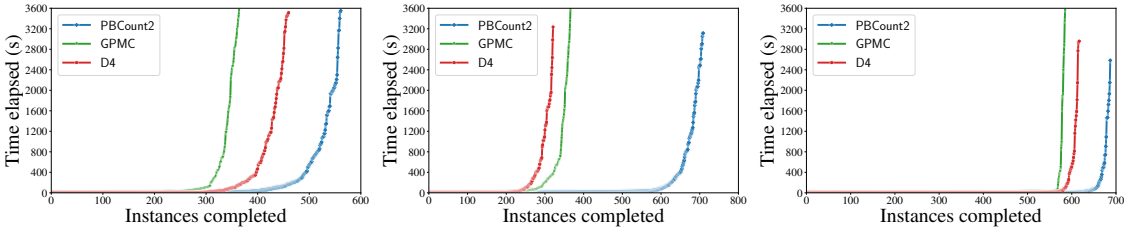
Approach	k value				
	10^1	10^2	10^3	10^4	10^5
Top-down	3.325	61.753	60.530	60.881	64.097
Bottom-up	0.005	0.004	0.004	0.004	0.004

Table 5.7: Runtime (seconds) to complete model counting for formula in Equation 5.4 with all coefficients set to 1.

magnitude reduction in the runtime of the bottom-up compilation approach. In contrast, the top-down approach terminates early only in $k = 10^1$ case and requires full enumeration in other cases. In the absence of early termination, top-down compilation approach is much slower than bottom-up compilation approach, and this is reflected in our dynamic compilation heuristic.

5.6.3 RQ3: Projected PB Model Counting Performance

In this rest of this section, we focus on PBCount2, which is an extension of PBCount with the projected and incremental model counting features.

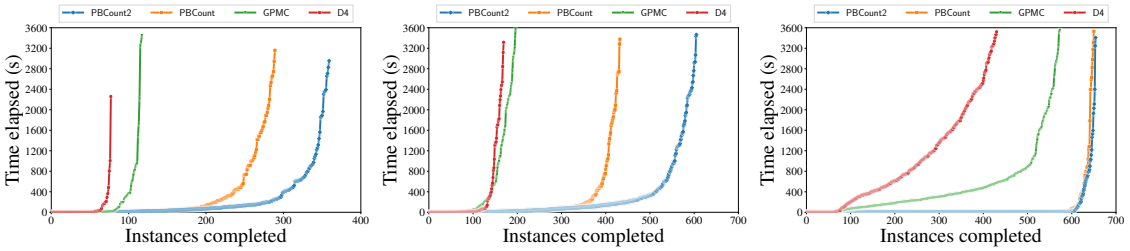


(a) Auction (b) $\mathcal{M}$-dim Knapsack (c) Sensor placement

Figure 5.9: Runtime cactus plots of PBCount2 and competing methods for each benchmark set, for projected model counting.

Benchmarks	D4	GPMC	PBCount2
Auction	460	364	**561**
$\mathcal{M}$-dim knapsack	321	366	**709**
Sensor placement	617	585	**687**
Total	1398	1315	**1957**

Table 5.8: Number of projected benchmark instances completed by D4, GPMC and PBCount2 in 3600s, higher is better.



(a) Auction (b) $\mathcal{M}$-dim Knapsack (c) Sensor placement

Figure 5.10: Runtime cactus plots of PBCount2 and competing methods for each benchmark set, for incremental model counting with 5 steps.

Experiment	Counter	Auction	$\mathcal{M}$-dim knapsack	Sensor placement	Total
3-step	D4	85	179	536	800
	GPMC	130	216	586	932
	PBCount	312	458	657	1427
	PBCount2	**369**	**629**	**660**	**1658**
5-step	D4	77	169	430	676
	GPMC	117	196	573	886
	PBCount	289	432	650	1371
	PBCount2	**359**	**605**	**654**	**1618**

Table 5.9: Number of incremental benchmark instances completed by PBCount2, PBCount, D4, and GPMC in 3600s, higher is better. '3-step' indicates results of incremental PBCount2 with 3 counting steps, and '5-step' indicates that with 5 counting steps.

We conducted extensive evaluations to understand the performance of PBCount2 compared to state-of-the-art projected CNF model counters D4 and GPMC [LM17; RS17]. We show the results in Table 5.8 and cactus plots in Figure 5.9. PBCount2 is able to complete 1957 instances, demonstrating a substantial lead over the 1398 instances of D4 and the 1315 instances of GPMC. Overall, PBCount2 solves around 1.40× the number of instances of D4 and 1.49× that of GPMC, highlighting the efficacy of PBCount2 in projected PB model counting tasks.

5.6.4 RQ4: Incremental Counting Performance

We conducted experiments to analyze the performance of PBCount2's incremental mode against the state-of-the-art PBCount and CNF counters. In our experiments, we looked at the 3-step and 5-step benchmark configurations. The experiments were run with a total timeout of 3600s for the two benchmark configurations. The results

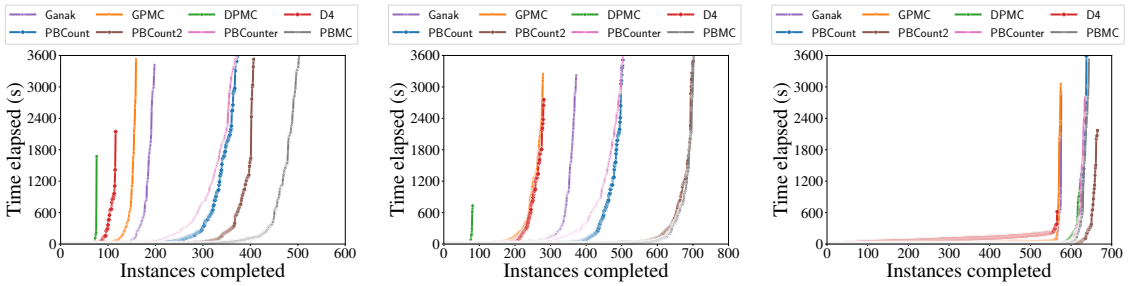
are shown in Table 5.9.

In the 3-step benchmarks, PBCount2 shows a considerable performance advantage over D4, GPMC, and PBCount. More specifically, PBCount2 completed 2.07× the number of benchmarks completed by D4, 1.78× that of GPMC, and 1.16× that of PBCount respectively. Across all 3 incremental benchmark sets for 3-step experiments, PBCount2 performs the same as or better than PBCount. As we move to 5-step benchmarks, we see PBCount2 having a more significant lead over the competing methods. Overall, PBCount2 completes 2.39× the number of instances of D4, 1.83× that of GPMC, and 1.18× that of the original PBCount. Notably when moving from 3-step benchmarks to 5-step benchmarks, the drop in the number of completed incremental benchmarks of PBCount2 is only 40, compared to 56 for PBCount, 46 for GPMC, and 124 for D4. Additionally, we show the cactus plots of each set of incremental benchmarks under 5-step benchmark configuration in Figure 5.10. The plot further highlights the runtime advantages of PBCount2 over competing approaches. Overall, the evaluations demonstrate the efficacy of PBCount2 at handling incremental PB model counting, as opposed to existing approaches that can only treat each incremental step as a separate model counting instance.

5.6.5 RQ5: Top-Down PB Model Counting Performance

In order to have a more complete understanding of how the two major counter designs perform for PB model counting, we revisit our evaluations in **RQ1** and include evaluations for our top-down search-based model counter PBMC. In addition, we include evaluations of more recent counters, namely PB model counters PBCounter [LXY24] and PBCount2 [YM25], as well as CNF model counting competition 2024 winner Ganak [Sha+19]. The appended results are shown in Table 5.10 and cactus plots are shown in Figure 5.11.

Overall, PBMC is able to return counts for 1849 instances, 76 instances more than that of PBCount2, 341 instances more than PBCounter, and 685 instances more than Ganak. There is a clear performance separation between native PB model counters and CNF counters with conversion approach. In the evaluation, PBMC leads in *auction* and *M dim-knapsack* benchmark sets and came in second in the *sensor placement* benchmark set. It is also worth noting that PBCount2 with our



(a) Auction (b) $\mathcal{M}$-dim Knapsack (c) Sensor placement

Figure 5.11: Runtime cactus plots of D4, DPMC, Ganak, GPMC, PBCount, PBCount2, PBCounter, and PBMC with 3600s timeout. A point on the plot indicates the respective counter could return counts for x instances in y time.

Counters	Auction	$\mathcal{M}$-dim knapsack	Sensor placement	Total
DPMC	76	81	625	782
D4	116	281	566	963
GPMC	159	279	575	1013
Ganak	198	372	576	1164
PBCount	372	503	638	1513
PBCount2	407	701	**665**	1773
PBCounter	371	503	634	1508
PBMC	**503**	**702**	644	**1849**

Table 5.10: Number of benchmark instances completed by each of the respective counters in 3600s, higher is better. The first top half counters (DPMC to Ganak) are CNF counters, and the bottom half are native PB counters.

new LOW-MD merge ordering heuristic performs significantly better than the first version of PBCount.

5.6.6 RQ6: Performance Impact of VCIS Heuristics

We conducted further experiments to understand the impact of our VCIS heuristic as introduced previously. In particular, we compared our implementation of PBMC with VCIS against a baseline version of PBMC that uses the existing variable decision heuristics from the GPMC model counter. The results are shown in Table 5.11.

In the evaluations, PBMC with our VCIS variable decision heuristic (VCIS-heu) is able to return counts for 1849 benchmark instances in total, substantially more than the 1772 benchmark instances when PBMC is coupled with the existing heuristics from GPMC (GPMC-heu). In particular, VCIS-heu returned counts for 115 more

Counters	Auction	$\mathcal{M}$-dim knapsack	Sensor placement	Total
GPMC-heu	**541**	587	644	1772
VCIS-heu	503	**702**	644	**1849**

Table 5.11: Number of PB benchmark instances counted by PBMC, using VCIS and GPMC variable decision heuristics in 3600s.

instances than GPMC-heu in *$\mathcal{M}$-dim knapsack* benchmarks, while losing out 38 instances in *Auction* benchmarks. Both heuristics performed the same in *sensor placement* benchmarks. This set of evaluations highlights the tremendous impact of variable decision ordering in the context of PB model counting and demonstrates the performance benefits of our VCIS variable decision heuristic.

5.6.7 Additional Comparisons and Observations

Ganak	PBCounter	PBCount2	PBMC
761	682	289	741

Table 5.12: Number of benchmark instances completed by each of the respective counters in 3600s for benchmark by Lai, Xu, and Yin.

Ganak (no pp)	PBCount2 (Bouquet Tree)	PBMC (no pp)
499	685	740

Table 5.13: Number of benchmark instances completed by each of the respective counters with different configurations in 3600s for benchmark by Lai, Xu, and Yin. A counter with 'no pp' refers to the counter having preprocessing disabled (for Ganak we only disable Arjun preprocessing). 'Bouquet Tree' refers to PBCount2 running with the 'Bouquet Tree' as ADD merge order heuristic.

Apart from our benchmarks, we further conducted comparisons on a different set of benchmarks used by a recently published work [LXY24] for completeness. We compared PBCounter, PBCount2, Ganak, and PBMC on the aforementioned benchmarks. We adapted the benchmarks for Ganak by performing the conversion using the same PBEncoder as we did for our benchmarks and removed the instances that were not successfully converted. We show the number of instances each counter could complete, and disregard the unsatisfiable instances i.e. those with 0 counts. The results are shown in Table 5.12 and 5.13

The results show two main observations, the first being that preprocessing contributes a significant portion to a modern CNF counter's performance as we see Ganak leading with 761 instances counter dropping to 499 instances when preprocessing is disabled. Whereas the same is not true for PB model counters, as we can see from the performance of PBMC with and without preprocessing resulting in a change of only 1 instance. The second observation is that PB model counters tend to be very sensitive to heuristics, as we can see from the drastic performance discrepancy of PBCount2 when using LOW-MD (Table 5.12) and 'bouquet tree' (Table 5.13) ADD merging heuristics.

We studied the benchmarks by Lai, Xu, and Yin in detail and compared it to the benchmarks that we used in prior evaluations. The two different suites of benchmarks are of very different characteristics as we show in Tables 5.14, 5.15, and 5.16.

Percentile	Auction	Knapsack	Sensor	LXY24
25%	0.000	0.000	0.9965	0.3818
50%	0.000	0.000	0.9992	0.8130
75%	0.000	0.100	0.9999	1.0000

Table 5.14: Percentile statistics of ratio of constraints in each PB instance that has only '1' as term coefficients. 'LXY24' and '50%' value of 0.8130 means that on median, a PB instance in LXY24 benchmark set has about 81.3 percent of its PB constraints having only coefficient value of 1. 'LXY24' refers to the benchmark set adapted from PBCounter work [LXY24].

Percentile	Auction	Knapsack	Sensor	LXY24
25%	0.3798	0.9885	0.9600	0.0846
50%	0.5567	0.9939	0.9877	0.1447
75%	0.8024	0.9957	0.9947	0.1688

Table 5.15: Percentile statistics of $\frac{\text{median PG degree}}{\text{num var}}$ of the benchmark instances in different benchmark sets, where PG stands for primal graph. Primal graph have variables as nodes, and nodes are connected if they appear in same PB constraint. 'LXY24' refers to the benchmark set adapted from PBCounter work [LXY24].

There is a clear separation between *Auction*, *Knapsack*, *Sensor* and *LXY24* when looking at the length of PB constraints (number of literals in the constraint) from each set of benchmarks (Table 5.16), with *LXY24* having the smallest median

Percentile	Auction	Knapsack	Sensor	LXY24
25%	22.000	24.000	10.000	2.000
50%	27.000	30.000	11.000	4.000
75%	32.625	36.000	23.000	8.000

Table 5.16: Percentile statistics of median PB constraint lengths of each benchmark PB formula instance, of the different benchmark sets. 'Auction' and '25%' value of 22 means that the 25th percentile of median PB constraint length across all PB formula instances in 'Auction' benchmark is 22. 'LXY24' refers to the benchmark set adapted from PBCounter work [LXY24].

value of 4 for the median PB constraint length among the benchmarks. When looking at the ratio of PB constraints in the benchmark instances that have only 1 as coefficients (Table 5.14), there again is a clear separation between (a) *Auction* and *Knapsack* benchmarks which have almost none and (b) *Sensor* and *LXY24* benchmarks which comprises largely of such constraints. Finally, the primal graph of PB instances in *LXY24* is also much less relatively connected compared to the other benchmarks when looking at Table 5.15. In fact, the median of median primal graph node degree is 21 for *LXY24*, 47 for *Auction*, 80.5 for *Sensor*, and 163 for *Knapsack*.

5.7 Summary

In this chapter, we demonstrate the core idea of the alternative logic pillar of our framework on scaling knowledge compilation applications. We did this over a series of work that introduced the first exact PB model counter PBCount, subsequently the first projected PB model counter and incremental counter PBCount2, and finally the first top-down PB model counter PBMC. The evaluations on PBCount highlight the potential scalability improvements in practice from considering alternative input logic, in this case PB, when designing knowledge compilation applications. In particular, PBCount completed 1.9× the number of benchmark instances of DPMC, which also uses an ADD counting approach but takes CNF formulas as input. Additionally, the evaluations on PBCount and subsequently PBCount2 further demonstrate the potential in considering alternative logic forms, both managed to have a sizable lead over state-of-the-art projected CNF counters arising from decades of works.

Finally, the evaluations on PBMC present the full picture of PB model counting, showing that both top-down and bottom-up approaches when adapted to natively handling PB formula input, have the potential to perform much better than existing state-of-the-art CNF counters. Additionally, the evaluations also highlight some of the current limitations of the PB counters, mainly that they are rather sensitive to heuristics and that the type of PB instances that each heuristic performs well on has yet to be studied in sufficient detail.

Chapter 6

Concluding Remarks and Open Problems

6.1 Concluding Remarks

In the chapters thus far, this thesis presented the ERA framework for improving the scalability of applications involving knowledge compilation techniques. In particular, this thesis has introduced three different perspectives when approaching application scenarios involving knowledge compilation techniques – a) efficient usage of knowledge compilation diagrams b) recoverable approximations when looking at application settings as a whole c) usage of alternative logic input forms to describe problems. The three different perspectives are demonstrated in various scenarios ranging from pseudo-boolean model counting to incremental sampling and recoverable approximations in route sampling. In each of the application scenarios demonstrated, the prototype implementations demonstrated better scalability compared to the relevant state-of-the-art approaches at the time the ideas were published. In summary, this thesis presents the ERA framework for thinking about the scalability of knowledge compilation applications and should be used in tandem with existing techniques to further scale the applications. The ERA framework is orthogonal to both methodologies focusing specifically on improving general knowledge compilation techniques and methodologies focusing on application scenarios in isolation. We hope the ERA framework and underlying techniques can be inspirational when designing and developing knowledge compilation applications.

6.2 Open Problems and Future Directions

While we explored the different pillars of our ERA framework on scaling knowledge compilation applications, there are still numerous open problems that are worth studying in further detail. Some of the notable problems are the following:

1. Constrained sampling with PB formula

2. Finer classification of PB formulas with respect to tool and heuristic performance

3. Applications of PB sampling and counting

4. More sophisticated preprocessing techniques involving PB formulas

In Chapters 3 and 5, we studied applications of knowledge compilation techniques on the incremental constrained sampling task and PB model counting respectively. It would be a natural extension to study the feasibility and applications of a constrained sampler that natively handles PB formula inputs, or even a mixture of CNF and PB constraint inputs. With PB sampling, perhaps it is also worthwhile to revisit the route sampling task in Chapter 4 with PB encodings instead of CNF encodings. In addition, it might also make sense to explore weighted sampling with more succinct KC diagrams that capture symmetry or literal equivalences such as CCDD [LMY22]. In Chapter 5, we introduced two of the early exact PB counters with different techniques to the research community. It is apparent that the topic of PB model counting is in its early days, and there are numerous interesting research topics arising from our initial attempts at PB model counting. A notable topic would be that of a finer classification of PB formulas with respect to heuristics used in PB counters. It is highly possible that there is no single heuristic configuration that is suitable for all kinds of input problem PB formulas, and therefore it would make sense to systematically classify and understand the scenarios whereby different heuristics work well. In addition, there is currently a lack of more sophisticated preprocessing and simplification techniques for PB formulas in contrast to the more established area of CNF model counting whereby the continuous development of preprocessing techniques contributed significantly to performance improvements over the years. As such, it would make sense to study the topic of preprocessing

for PB model counting, whether the preprocessing technique would be in general or knowledge compilation specific.

Bibliography

[Abí+11] I. Abío, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-carbonell. "BDDs for Pseudo-Boolean Constraints - Revisited". In: *International Conference on Theory and Applications of Satisfiability Testing*. 2011.

[Abí+13] I. Abío, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-carbonell. "A Parametric Approach for Smaller and Better Encodings of Cardinality Constraints". In: *International Conference on Principles and Practice of Constraint Programming*. 2013.

[AHT18] D. Achlioptas, Z. Hammoudeh, and P. Theodoropoulos. "Fast Sampling of Perfectly Uniform Satisfying Assignments". In: *SAT*. 2018.

[Ahu+90] R. K. Ahuja, K. Mehlhorn, J. Orlin, and R. E. Tarjan. "Faster algorithms for the shortest path problem". In: *Journal of the ACM (JACM)* 37.2 (1990), pp. 213–223.

[Azi+15] R. A. Aziz, G. Chu, C. Muise, and P. J. Stuckey. "#∃SAT: Projected Model Counting". In: *International Conference on Theory and Applications of Satisfiability Testing*. 2015.

[BDP03] F. Bacchus, S. Dalmao, and T. Pitassi. "Algorithms and complexity results for #SAT and Bayesian inference". In: *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings.* (2003).

[Bah+93] 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". In: *International Conference on Computer Aided Design*. 1993.

[Bal+21] T. Baluta, Z. L. Chua, K. S. Meel, and P. Saxena. "Scalable Quantitative Verification For Deep Neural Networks". In: *Proceedings of International Conference on Software Engineering (ICSE)*. May 2021.

[BRJ15] S. Banerjee, C. Riquelme, and R. Johari. “Pricing in Ride-Share Platforms: A Queueing-Theoretic Approach”. In: *Econometrics: Econometric & Statistical Methods - Special Topics eJournal* (2015).

[BLM20] E. Baranov, A. Legay, and K. S. Meel. “Baital: An Adaptive Weighted Sampling Approach for Improved t-wise Coverage”. In: *Proc. 28th European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2020.

[BJK21] A. Biere, M. Järvisalo, and B. Kiesl. “Preprocessing in SAT Solving”. In: *Handbook of Satisfiability*. 2021.

[BN07] L. Blumrosen and N. Nisan. “Algorithmic Game Theory”. In: *Algorithmic Game Theory*. Cambridge University Press, Sept. 2007. Chap. 11, pp. 267–300.

[Boe17] G. Boeing. “OSMnx: New Methods for Acquiring, Constructing, Analyzing, and Visualizing Complex Street Networks”. In: *Econometrics: Computer Programs & Software eJournal* (2017).

[Boo54] G. Boole. “An Investigation of the Laws of Thought: On which are founded the mathematical theories of logic and probabilities”. In: 1854.

[Bou99] F. Bouquet. “Gestion de la dynamicité et énumération d’impliquants premiers : une approche fondée sur les Diagrammes de Décision Binaire”. In: PhD Dissertation, Aix-Marseille 1. 1999.

[Bry86] R. E. Bryant. “Graph-Based Algorithms for Boolean Function Manipulation”. In: *IEEE Transactions on Computers* C-35 (1986), pp. 677–691.

[CD97] M. Cadoli and F. M. Donini. “A Survey on Knowledge Compilation”. In: *AI Communications* 10 (1997), pp. 137–150.

[CSD17] A. Choi, Y. Shen, and A. Darwiche. “Tractability in Structured Probability Spaces”. In: *NeurIPS*. Vol. 30. 2017, pp. 3477–3485.

[CVD15] A. Choi, G. Van den Broeck, and A. Darwiche. “Probability Distributions over Structured Spaces”. In: *AAAI*. 2015.

[CVB20] Y. Choi, A. Vergari, and G. V. den Broeck. “Probabilistic Circuits: A Unifying Framework for Tractable Probabilistic Models”. In: 2020.

[CM17] R. R. Clewlow and G. Mishra. “Disruptive Transportation: The Adoption, Utilization, and Impacts of Ride-Hailing in the United States”. In: 2017.

[Col20] J. Collison. “The Impact of Online Food Delivery Services on Restaurant Sales”. In: (2020).

[Dar01] A. Darwiche. “Decomposable negation normal form”. In: *J. ACM* 48 (2001), pp. 608–647.

[Dar02] A. Darwiche. “A compiler for deterministic, decomposable negation normal form”. In: *AAAI/IAAI*. 2002.

[Dar11] A. Darwiche. “SDD: A New Canonical Representation of Propositional Knowledge Bases”. In: *IJCAI*. 2011.

[DM02] A. Darwiche and P. Marquis. “A Knowledge Compilation Map”. In: *J. Artif. Intell. Res.* 17 (2002), pp. 229–264.

[Dev20] J. Devriendt. “Watched Propagation of 0-1 Integer Linear Constraints”. In: *International Conference on Principles and Practice of Constraint Programming*. 2020.

[Dev+21] J. Devriendt, S. Gocht, E. Demirovic, J. Nordström, and P. J. Stuckey. “Cutting to the Core of Pseudo-Boolean Optimization: Combining Core-Guided Search with Cutting Planes Reasoning”. In: *AAAI Conference on Artificial Intelligence*. 2021.

[Dix04] H. Dixon. “Automating pseudo-boolean inference within a DPLL framework”. 2004.

[DG84] W. F. Dowling and J. H. Gallier. “Linear-Time Algorithms for Testing the Satisfiability of Propositional Horn Formulae”. In: *J. Log. Program.* 1 (1984), pp. 267–284.

[DPV20a] J. M. Dudek, V. H. N. Phan, and M. Y. Vardi. “ADDMC: Weighted Model Counting with Algebraic Decision Diagrams”. In: *AAAI Conference on Artificial Intelligence*. 2020.

[DPV20b] J. M. Dudek, V. H. N. Phan, and M. Y. Vardi. “DPMC: Weighted Model Counting by Dynamic Programming on Project-Join Trees”. In: *International Conference on Principles and Practice of Constraint Programming*. 2020.

[DPV21] J. M. Dudek, V. H. N. Phan, and M. Y. Vardi. “ProCount: Weighted Projected Model Counting with Graded Project-Join Trees”. In: *International Conference on Theory and Applications of Satisfiability Testing*. 2021.

[Dye+93] M. E. Dyer, A. M. Frieze, R. Kannan, A. Kapoor, L. Perkovic, and U. V. Vazirani. “A Mildly Exponential Time Algorithm for Approximating the Number of Solutions to a Multidimensional Knapsack Problem”. In: *Combinatorics, Probability and Computing* 2 (1993), pp. 271–284.

[EB05] N. Eén and A. Biere. “Effective Preprocessing in SAT Through Variable and Clause Elimination”. In: *International Conference on Theory and Applications of Satisfiability Testing*. 2005.

[ES06] N. Eén and N. Sörensson. “Translating Pseudo-Boolean Constraints into SAT”. In: *J. Satisf. Boolean Model. Comput.* 2 (2006), pp. 1–26.

[EN18] J. Elffers and J. Nordström. “Divide and Conquer: Towards Faster Pseudo-Boolean Solving”. In: *International Joint Conference on Artificial Intelligence*. 2018.

[FMM20] C. Fan, K. Miller, and S. Mitra. “Fast and Guaranteed Safe Controller Synthesis for Nonlinear Vehicle Models”. In: *Computer Aided Verification* 12224 (2020), pp. 629–652.

[FBS19] K. Fazekas, A. Biere, and C. Scholl. “Incremental Inprocessing in SAT Solving”. In: *International Conference on Theory and Applications of Satisfiability Testing*. 2019.

[GL80] G. Gens and E. Levner. “Complexity of approximation algorithms for combinatorial problems: a survey”. In: *SIGACT News* 12 (1980), pp. 52–65.

[Goo+14] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. "Generative Adversarial Networks". 2014. arXiv: 1406.2661 [stat.ML].

[Gup+19] R. Gupta, S. Sharma, S. Roy, and K. S. Meel. "WAPS: Weighted and Projected Sampling". In: *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Apr. 2019.

[HSS08] A. A. Hagberg, D. A. Schult, and P. Swart. "Exploring Network Structure, Dynamics, and Function using NetworkX". In: *Proceedings of the 7th Python in Science Conference*. 2008, pp. 11–15.

[Har+20] C. Harris, K. J. Millman, S. Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. Smith, R. Kern, M. Picus, S. Hoyer, M. Kerkwijk, M. Brett, A. Haldane, J. F. del R'io, M. Wiebe, P. Peterson, P. G'erard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. "Array programming with NumPy". In: *Nature* 585 (2020), pp. 357–362.

[Ino+16] T. Inoue, H. Iwashita, J. Kawahara, and S.-i. Minato. "Graphillion: software library for very large sets of labeled graphs". In: *International Journal on Software Tools for Technology Transfer* (2016).

[Jac19] D. Jackson. "Alloy: a language and tool for exploring software designs". In: *Commun. ACM* 62 (2019).

[JS96] M. Jerrum and A. Sinclair. "The Markov chain Monte Carlo method: an approach to approximate counting and integration". In: 1996.

[JVV86] M. Jerrum, L. G. Valiant, and V. V. Vazirani. "Random Generation of Combinatorial Structures from a Uniform Distribution". In: *Theor. Comput. Sci.* 43 (1986), pp. 169–188.

[Kaw+17] J. Kawahara, T. Inoue, H. Iwashita, and S.-i. Minato. "Frontier-Based Search for Enumerating All Constrained Subgraphs with Compressed Representation". In: *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.* (2017).

[KW19] D. P. Kingma and M. Welling. “An Introduction to Variational Autoencoders”. In: *Foundations and Trends® in Machine Learning* 12.4 (2019), pp. 307–392. ISSN: 1935-8245. URL: http://dx.doi.org/10.1561/2200000056.

[KK07] N. Kitchen and A. Kuehlmann. “Stimulus generation for constrained random simulation”. In: *2007 IEEE/ACM International Conference on Computer-Aided Design*. IEEE. 2007, pp. 258–265.

[Knu05] D. E. Knuth. “The Art of Computer Programming, Volume 4, Fascicle 2: Generating All Tuples and Permutations”. In: 2005.

[KJ21] T. Korhonen and M. Järvisalo. “Integrating Tree Decompositions into Decision Heuristics of Propositional Model Counters”. In: *International Conference on Principles and Practice of Constraint Programming*. 2021.

[LM17] J.-M. Lagniez and P. Marquis. “An Improved Decision-DNNF Compiler”. In: *International Joint Conference on Artificial Intelligence*. 2017.

[LLY17] Y. Lai, D. Liu, and M. Yin. “New Canonical Representations by Augmenting OBDDs with Conjunctive Decomposition”. In: *J. Artif. Intell. Res.* 58 (2017), pp. 453–521.

[LMY21] Y. Lai, K. S. Meel, and R. H. C. Yap. “The Power of Literal Equivalence in Model Counting”. In: *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*. Feb. 2021.

[LMY22] Y. Lai, K. S. Meel, and R. H. C. Yap. “CCDD: A Tractable Representation for Model Counting and Uniform Sampling”. In: *ArXiv* abs/2202.10025 (2022).

[LMY25] Y. Lai, K. S. Meel, and R. H. C. Yap. “Panini: An Efficient and Flexible Knowledge Compiler”. In: *International Conference on Computer Aided Verification*. 2025.

[LXY24] Y. Lai, Z. Xu, and M. Yin. “PBCounter: weighted model counting on pseudo-boolean formulas”. In: *Frontiers of Computer Science* 19 (2024), p. 193402.

[LSM23] A. L. D. Latour, A. Sen, and K. S. Meel. “Solving the Identifying Code Set Problem with Grouped Independent Support”. In: *Proceedings of the 32nd International Joint Conference on Artificial Intelligence*. Aug. 2023.

[Le 01] D. Le Berre. “Exploiting the real power of unit propagation lookahead”. In: *Electron. Notes Discret. Math.* 9 (2001), pp. 59–80.

[Le +18] D. Le Berre, P. Marquis, S. Mengel, and R. Wallon. “Pseudo-Boolean Constraints from a Knowledge Representation Perspective”. In: *International Joint Conference on Artificial Intelligence*. 2018.

[Lee59] C. Y. Lee. “Representation of switching circuits by binary-decision programs”. In: *Bell System Technical Journal* 38 (1959), pp. 985–999.

[LKH06] J. Letchner, J. Krumm, and E. Horvitz. “Trip Router with Individualized Preferences (TRIP): Incorporating Personalization into Route Planning”. In: *AAAI*. 2006.

[Lot+23] K. Lotz, A. Goel, B. Dutertre, B. Kiesl-Reiter, S. Kong, R. Majumdar, and D. Nowotka. “Solving String Constraints Using SAT”. In: *International Conference on Computer Aided Verification*. 2023.

[MRS08] C. D. Manning, P. Raghavan, and H. Schütze. “Introduction to information retrieval”. In: 2008.

[MDM08] R. Mateescu, R. Dechter, and R. Marinescu. “AND/OR Multi-Valued Decision Diagrams (AOMDDs) for Graphical Models”. In: *J. Artif. Intell. Res.* 33 (2008), pp. 465–519.

[Min93] S.-i. Minato. “Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems”. In: *30th ACM/IEEE Design Automation Conference* (1993), pp. 272–277.

[Mui+12] C. Muise, S. A. McIlraith, J. C. Beck, and E. Hsu. “DSHARP: Fast d-DNNF Compilation with sharpSAT”. In: *Canadian Conference on Artificial Intelligence*. 2012.

[Nad22] A. Nadel. “Introducing Intel(R) SAT Solver”. In: *International Conference on Theory and Applications of Satisfiability Testing*. 2022.

[NR12] A. Nadel and V. Ryvchin. “Efficient SAT Solving under Assumptions”. In: *International Conference on Theory and Applications of Satisfiability Testing*. 2012.

[NRS14] A. Nadel, V. Ryvchin, and O. Strichman. “Ultimately Incremental SAT”. In: *International Conference on Theory and Applications of Satisfiability Testing*. 2014.

[Nar+19] N. Narodytska, A. A. Shrotri, K. S. Meel, A. Ignatiev, and J. Marques-Silva. “Assessing Heuristic Machine Learning Explanations with Model Counting”. In: *International Conference on Theory and Applications of Satisfiability Testing*. 2019.

[Nav+07] Y. Naveh, M. Rimon, I. Jaeger, Y. Katz, M. Vinov, E. Marcus, and G. Shurek. “Constraint-Based Random Stimuli Generation for Hardware Verification”. In: *AI Mag.* 28 (2007), pp. 13–30.

[Ope17] OpenStreetMap contributors. “Planet dump retrieved from https://planet.osm.org ”. https://www.openstreetmap.org. 2017.

[Pas+19] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *NeurIPS*. 2019.

[Peh+20] R. Peharz, S. Lang, A. Vergari, K. Stelzner, A. Molina, M. Trapp, G. V. den Broeck, K. Kersting, and Z. Ghahramani. “Einsum Networks: Fast and Scalable Learning of Tractable Probabilistic Circuits”. In: *ICML*. 2020.

[PS15] T. Philipp and P. Steinke. “PBLib - A Library for Encoding Pseudo-Boolean Constraints into CNF”. In: *International Conference on Theory and Applications of Satisfiability Testing*. 2015.

[PHS08] C. Piette, Y. Hamadi, and L. Sais. “Vivifying Propositional Clausal Formulae”. In: *European Conference on Artificial Intelligence*. 2008.

[PD11] H. Poon and P. M. Domingos. “Sum-product networks: A new deep architecture”. In: *2011 IEEE International Conference on Computer Vision Workshops (ICCV Workshops)* (2011), pp. 689–690.

[Pre03] S. Prestwich. “SAT problems with chains of dependent variables”. In: *Discret. Appl. Math.* 130 (2003), pp. 329–350.

[RSL74] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis. “Approximate algorithms for the traveling salesperson problem”. In: *15th Annual Symposium on Switching and Automata Theory (swat 1974)*. IEEE. 1974, pp. 33–42.

[RS17] K. H. Ryosuke Suzuki and M. Sakai. “Improvement of Projected Model-Counting Solver with Component Decomposition Using SAT Solving in Components”. In: *JSAI Technical Report*. Mar. 2017.

[Sha+18] S. Sharma, R. Gupta, S. Roy, and K. S. Meel. “Knowledge Compilation meets Uniform Sampling”. In: *LPAR*. 2018.

[Sha+19] S. Sharma, S. Roy, M. Soos, and K. S. Meel. “GANAK: A Scalable Probabilistic Exact Model Counter”. In: *Proceedings of International Joint Conference on Artificial Intelligence*. Aug. 2019.

[SCD18] Y. Shen, A. Choi, and A. Darwiche. “Conditional PSDDs: Modeling and Learning With Modular Knowledge”. In: *AAAI*. 2018.

[She+19] Y. Shen, A. Goyanka, A. Darwiche, and A. Choi. “Structured Bayesian Networks: From Inference to Learning with Routes”. In: *AAAI*. 2019.

[SSL15] T. Shi, J. Steinhardt, and P. Liang. “Learning Where to Sample in Structured Prediction”. In: *AISTATS*. 2015.

[Sin05] C. Sinz. “Towards an Optimal CNF Encoding of Boolean Cardinality Constraints”. In: *International Conference on Principles and Practice of Constraint Programming*. 2005.

[Smi22] E. V. Smith. “toposort”. 2022. URL: https://pypi.org/project/toposort/.

[Som15] F. Somenzi. “CUDD: CU decision diagram package - release 3.0.0”. Version 3.0.0. 2015.

[Tse83] G. S. Tseitin. “On the Complexity of Derivation in Propositional Calculus”. In: 1983.

[Val79] L. G. Valiant. “The complexity of enumeration and reliability problems”. In: *Siam Journal on Computing* 8.3 (1979), pp. 410–421.

[WFY18] Z. Wang, K. Fu, and J. Ye. “Learning to Estimate the Travel Time”. In: *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (2018).

[WK17] O. White and M. Kuranowski. “Pyroutelib3”. 2017. URL: https://github.com/MKuranowski/pyroutelib3.

[Yam+15] A. Yamada, T. Kitamura, C. Artho, E.-H. Choi, Y. Oiwa, and A. Biere. “Optimization of Combinatorial Testing by Incremental SAT Solving”. In: *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)* (2015), pp. 1–10.

[YM21] J. Yang and K. S. Meel. “Engineering an Efficient PB-XOR Solver”. In: *International Conference on Principles and Practice of Constraint Programming*. 2021.

[YLM25] S. Yang, Y. Lai, and K. S. Meel. “On Top-Down Pseudo-Boolean Model Counting”. In: *nternational Conference on Theory and Applications of Satisfiability Testing*. 2025.

[YLM23] S. Yang, C. Liang, and K. S. Meel. “Scalable Probabilistic Routes”. In: *International Conference on Logic for Programming, Artificial Intelligence and Reasoning*. 2023.

[YLM22] S. Yang, V. Liang, and K. S. Meel. “INC: A Scalable Incremental Weighted Sampler”. In: *Proceedings of the 22nd Conference on Formal Methods in Computer-Aided Design*. 2022.

[YM24] S. Yang and K. S. Meel. “Engineering an Exact Pseudo-Boolean Model Counter”. In: *Proceedings of the 38th Annual AAAI Conference on Artificial Intelligence*. 2024.

[YM25] S. Yang and K. S. Meel. “Towards Projected and Incremental Pseudo-Boolean Model Counting”. In: *Proceedings of the 39th Annual AAAI Conference on Artificial Intelligence*. 2025.

[Yu+17] C. Yu, X. Zhang, D. Liu, M. J. Ciesielski, and D. E. Holcomb. “Incremental SAT-Based Reverse Engineering of Camouflaged Logic Circuits”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 36 (2017), pp. 1647–1659.