Bachelor's Thesis Computing Science

Evaluating Model Counters with Sudoku Puzzles

Evaluating Performance of Model Counters with Respect to Preprocessing and Formula Characteristics Using CNF-SAT Encoding of Partial Sudoku Puzzles as a Benchmark

Milind Patel
s1115305

August 24, 2026

First supervisor/assessor:
Dr. J.S.L. Junges

Second assessor:
Dr. C.L.M. Kop

Radboud University

Abstract

Model counting, or #SAT, is the problem of determining the number of satisfying assignments of a Boolean formula. Unlike satisfiability solving, which only requires finding whether a solution exists, model counting must account for all satisfying assignments and is therefore computationally challenging. This thesis investigates the use of model counters for counting the solutions of partial Sudoku puzzles and studies how the representation and structural properties of the resulting SAT formulas affect model-counter performance.

Sudoku puzzles are encoded as Boolean formulas using two different encoding schemes, referred to as the minimal and extended encoding. A collection of partial Sudoku instances with multiple solutions is used to construct a benchmark set of CNF formulas. Three model counters, `Ganak`, `CNF2OBDD`, and `ADDMC` are evaluated on both the original formulas and formulas simplified using count-preserving preprocessing `Arjun`. The evaluation considers solver runtime, the number of instances solved within a fixed timeout, and the relationships between runtime and formula size, clause-to-variable ratio, treewidth, and number of solutions.

The results show that the effects of preprocessing are strongly dependent on the model counter. While some counters perform similarly or can even benefit from the raw formulas, `ADDMC` shows a substantial improvement after preprocessing. The choice of encoding also affects both the structural properties of the preprocessed formulas and model-counter runtime, with the extended encoding generally producing smaller formulas and lower treewidth after preprocessing. Formula size and treewidth are positively associated with runtime, particularly for `ADDMC`, although these parameters alone do not fully explain the performance of all model counters. The number of Sudoku solutions also shows a positive relationship with runtime, although this relationship is weaker.

Overall, the results demonstrate that reducing formula size or structural complexity does not necessarily lead to uniform performance improvements across model counters. The practical difficulty of a model-counting instance depends on preprocessing, structural properties, and whether the solver used for counting already implements effective preprocessing and inprocessing techniques.

Contents

Chapter 1

Introduction

Boolean satisfiability (SAT) is the problem of determining whether a propositional Boolean formula has at least one satisfying assignment. A closely related problem is *model counting* (or #SAT), which asks for the number of satisfying assignments of a Boolean formula. While SAT only requires determining whether a solution exists, model counting requires accounting for all satisfying assignments. Exact model counting has applications in areas such as probabilistic inference, combinatorial analysis, and uncertainty reasoning, but is computationally challenging for general Boolean formulas.

The practical performance of a model counter depends not only on the logical problem being represented, but also on the structure of the input formula and the algorithm used by the counter. Different model counters exploit different properties of a CNF formula. For example, BDD-based approaches represent the set of satisfying assignments using decision diagrams, while dynamic programming based approaches exploit structural properties such as treewidth. Consequently, logically equivalent formulas can have substantially different solving times, and a transformation that benefits one model counting approach may not necessarily benefit another.

Preprocessing provides one way of changing the structure of a formula before model counting. SAT preprocessing techniques simplify formulas by removing redundancies, propagating assignments, and applying other transformations that reduce the complexity of the input. For model counting, however, it is not sufficient to preserve satisfiability: preprocessing must also preserve the number of satisfying assignments, or otherwise account for any changes to the count. Count-preserving preprocessing can therefore alter the size and structure of a CNF while retaining the model count. An important question is whether such structural simplification consistently translates into improved performance for different model counters.

This question is particularly relevant because model counters can respond differently to the same changes in formula structure. Reductions in the number of variables or clauses may make an instance easier for one approach, while changes to the relationships between variables may affect another approach differently. *Treewidth* provides one measure of such structural complexity and is particularly relevant to decomposition-based methods. Therefore, evaluating only the final runtime does not fully characterize the effect of preprocessing. Examining how preprocessing changes formula size and structural properties can provide additional insight into the observed performance differences.

This thesis investigates these effects using Sudoku as a source of structured SAT instances. A Sudoku puzzle can be encoded as a Boolean formula such that its satisfying assignments correspond to valid completions of the puzzle. By removing clues from valid puzzles, instances with multiple solutions can be obtained. The number of solutions can vary substantially between instances while the underlying problem domain remains fixed. This provides a controlled benchmark for investigating the relationship between solution count, formula structure, preprocessing, and model-counter performance.

Two different Sudoku-to-SAT encodings are considered in this thesis: *minimal* and *extended* encodings (Lynce & Ouaknine, 2006). The resulting CNF formulas are evaluated in both raw and count-preserving preprocessed form. Three model counters are used to evaluate the instances. In addition to comparing their overall performance, the study examines the number of variables, number of clauses, clause-to-variable ratio, treewidth, and number of solutions as potential factors associated with runtime.

The experimental study is guided by the following research questions:

1. To what extent can #SAT solvers count the solutions of partial Sudoku puzzles without explicitly enumerating the solution space?
2. How does the choice between the minimal and extended Sudoku-to-SAT encoding affect model-counter performance?
3. What effect does count-preserving preprocessing have on the performance of different model counters for Sudoku-derived CNF formulas?
4. How do the minimal and extended Sudoku-to-SAT encodings differ in the size and structural properties of their CNF formulas after count-preserving preprocessing?
5. To what extent do CNF formula characteristics and the number of solutions relate to the runtime of different model counters?

This thesis makes several empirical contributions. First, it provides a comparison of three model counters on a large collection of Sudoku-derived CNF instances. Second, it evaluates the effect of count-preserving preprocessing by comparing model-counter performance on raw and preprocessed formulas. Third, it compares two Sudoku-to-SAT encodings with respect to formula size, treewidth, and model-counter runtime after preprocessing. Finally, it examines the relationship between runtime and several properties of the resulting preprocessed CNF formulas, including the number of variables, number of clauses, clause-to-variable ratio, treewidth, and number of satisfying assignments.

The remainder of this thesis is organized as follows. Chapter 2 introduces basic concepts required for the remainder of the thesis. Chapter 3 presents overview of the model counting techniques and the exact model counters evaluated in this thesis. Chapter 4 presents the Sudoku-to-SAT encodings used to construct the benchmark instances. Chapter 5 introduces some preprocessing techniques used in modern SAT and #SAT solvers and model count preserving preprocessor used in this thesis. Chapter 6 introduces the CNF characteristics considered in the analysis, including formula size and treewidth. Chapter 7 covers the experimental methodology, including benchmark generation, preprocessing, and evaluation process. Chapter 8 presents and discusses the experimental results. Chapter 9 places the work in the context of previous research on SAT, #SAT, preprocessing, tree decomposition, benchmarking, and Sudoku encodings. Finally, Chapter 10 summarizes the main findings and discusses directions for future work.

Chapter 2

Preliminaries

This chapter introduces the basic concepts and terminology used throughout this thesis. It provides a brief overview of propositional logic, conjunctive normal form (CNF), Boolean satisfiability, model counting, and CNF preprocessing. These topics are introduced only to the extent required for understanding the remainder of the thesis; more detailed discussions are presented in the relevant chapters.

2.1 Propositional Logic

A proposition is a statement that is either *true* or *false*. Propositional logic is a formal system for reasoning about such statements. A *propositional variable* x is a Boolean variable that takes one of the two truth values. A *literal* is either a variable x or its negation $\neg x$. Literals are combined using logical connectives, including conjunction ($\wedge$), disjunction ($\vee$), negation ($\neg$), implication ($\rightarrow$), and equivalence ($\leftrightarrow$), to form Boolean formulas. The logical constants *true* ($\top$) and *false* ($\bot$) may also appear in formulas.

A *truth assignment* assigns Boolean values to the variables of a propositional formula. A formula is *satisfied* by an assignment if it evaluates to *true* under that assignment. It is *satisfiable* if at least one satisfying assignment exists; otherwise, it is *unsatisfiable*.

The syntax of propositional formulas can be defined recursively as follows:

$$\begin{aligned}
\langle \text{Formula} \rangle \;&::= \langle \text{Literal} \rangle \\
&\quad \mid \langle \text{Constant} \rangle \\
&\quad \mid \neg \langle \text{Formula} \rangle \\
&\quad \mid \langle \text{Formula} \rangle \vee \langle \text{Formula} \rangle \\
&\quad \mid \langle \text{Formula} \rangle \wedge \langle \text{Formula} \rangle \\
&\quad \mid \langle \text{Formula} \rangle \rightarrow \langle \text{Formula} \rangle \\
&\quad \mid \langle \text{Formula} \rangle \leftrightarrow \langle \text{Formula} \rangle \\
&\quad \mid (\langle \text{Formula} \rangle)
\end{aligned} \tag{2.1}$$

$$\langle \text{Constant} \rangle \;::= \top \mid \bot$$

$$\langle \text{Literal} \rangle \;::= \langle \text{Variable} \rangle \mid \neg \langle \text{Variable} \rangle$$

$$\langle \text{Variable} \rangle \;::= x_{\langle i \in \mathbb{N} \rangle}$$

The following simple formula illustrates the notation:

$$\phi \equiv (x_2 \rightarrow x_1) \wedge (\neg x_1 \leftrightarrow \top) \wedge (\neg x_2 \rightarrow (x_3 \rightarrow (x_4 \vee x_5))) \wedge (x_2 \rightarrow (x_3 \vee x_5)). \tag{2.2}$$

The notation introduced in this section is used throughout the remainder of the thesis.

Conjunctive Normal Form (CNF) Although propositional formulas may use arbitrary combinations of logical connectives, most SAT solvers and model counters operate on formulas in *conjunctive normal form* (CNF). A Boolean formula is in CNF if it is expressed as a conjunction ($\wedge$) of clauses, where each clause is a disjunction ($\vee$) of literals.

Formally, a CNF formula with m clauses can be written as

$$\phi = \bigwedge_{i=1}^{m} C_i = C_1 \wedge C_2 \wedge \cdots \wedge C_m,$$

where each clause C_i is of the form

$$C_i = \bigvee_{j=1}^{j \leq k} l_{ij} = l_{i1} \vee l_{i2} \vee \cdots \vee l_{ij},$$

and where each literal l_{ij} is either a atomic variable in pure form or its negation.

The following CNF formula contains $n = 5$ variables and $m = 4$ clauses, with a maximum clause length of $k = 4$. It is logically equivalent to the propositional formula in Equation 2.2.

$$\phi = (x_1 \vee \neg x_2) \wedge (\neg x_1) \wedge (x_2 \vee \neg x_3 \vee x_4 \vee x_5) \wedge (\neg x_2 \vee x_3 \vee x_5) \tag{2.3}$$

Each parenthesized expression is a clause, and the formula is satisfied only if every clause evaluates to *true* under a given truth assignment of variables.

All benchmark instances considered in this thesis are represented as CNF formulas. Their construction from Sudoku puzzles is described in Chapter 4.

2.2 Boolean Satisfiability Problem (SAT)

The Boolean Satisfiability Problem (SAT) is the decision problem of determining whether a Boolean formula ϕ has at least one satisfying assignment.

Let $X = \{x_1, x_2, \ldots, x_n\}$ denote the set of Boolean variables appearing in ϕ. A satisfying assignment is a mapping

$$\sigma : X \to \{0, 1\},$$

such that ϕ evaluates to $\top$ under σ. The notation

$$\sigma \models \phi$$

indicates that σ satisfies ϕ, i.e., σ is a satisfying assignment of ϕ.

SAT was the first problem shown to be NP-complete by Cook's theorem and is the canonical NP-complete problem (Aghayeva, 2025). As a result, many computational problems can be reduced to SAT and solved using highly optimized SAT solvers.

Modern SAT solvers routinely handle industrial instances containing millions of variables and clauses using techniques such as conflict-driven clause learning (CDCL), branching heuristics, and preprocessing (Newsham, Ganesh, Fischmeister, Audemard, & Simon, 2014).

CNF satisfiability can be restricted according to clause length. A formula in which every clause contains at most k literals is called a $\leq k$-*CNF* formula, and the corresponding satisfiability problem is denoted $\leq k$-*SAT*. When every clause contains exactly k literals, the problem is referred to as k-*SAT*. Unless stated otherwise, this thesis uses the simpler notation k-*SAT* to refer to formulas with clauses of at most k literals.

The complexity of SAT depends on the clause length. 2-SAT can be solved in polynomial time (Aspvall, Plass, & Tarjan, 1979), whereas k-SAT is NP-complete for every $k \geq 3$ (Aghayeva, 2025). Moreover, every instance of k-SAT for $k \geq 3$ can be reduced in polynomial time to 3-SAT, making 3-SAT the standard restricted form used in NP-completeness proofs.

Despite these theoretical distinctions, modern SAT solvers operate on general CNF formulas. Accordingly, the term *SAT* refers to CNF-SAT throughout this thesis unless stated otherwise.

SAT determines only whether at least one satisfying assignment exists. Many applications, however, require the total number of satisfying assignments. This problem is known as *model counting* or *#SAT* and is introduced in the next section.

2.3 Model Counting (#SAT)

Model counting, commonly denoted as #SAT, is the counting counterpart of the Boolean satisfiability problem. Formally, for a CNF formula ϕ, the model count is defined as

$$\#(\phi) = |\{\sigma \mid \sigma \models \phi\}|,$$

where σ is a complete truth assignment that satisfies ϕ.

Unlike SAT, which is NP-complete, exact model counting is #P-complete and is generally considered computationally more difficult (Valiant, 1979; Gomes, Sabharwal, & Selman, 2009). Consequently, many counting versions of NP-complete decision problems also belong to the class #P-complete. An interesting example is 2-SAT: although the decision problem is solvable in polynomial time, its counting counterpart (#2-SAT) is #P-complete (Gomes et al., 2009).

A variety of exact and approximate model counting algorithms have been proposed in the literature. This thesis focuses exclusively on exact model counting and evaluates three modern open-source model counters, which are presented in Chapter 3. Like modern SAT solvers, these model counters operate on Boolean formulas represented in Conjunctive Normal Form (CNF).

2.4 CNF Preprocessing

CNF preprocessing is the process of transforming a Boolean formula before SAT solving or model counting in order to simplify its representation. The goal is to reduce the size or structural complexity of the formula while preserving the properties required by the subsequent solving task, potentially improving solver performance.

Common preprocessing techniques include unit propagation, subsumption, clause and variable elimination, and clause strengthening. Depending on the application, these transformations may preserve only satisfiability or also preserve the exact number of satisfying assignments, which is essential for exact model counting.

This thesis employs the count-preserving `Arjun` preprocessor (Soos & Meel, 2024) to simplify the generated Sudoku CNF instances before model counting. Both the original and preprocessed formulas are evaluated experimentally. A detailed discussion of CNF preprocessing techniques and the Arjun preprocessor is provided in Chapter 5.

2.5 Encoding Problems into SAT

Many decision and counting problems can be transformed into SAT instances through polynomial-time reductions. The general idea is to represent the variables and constraints of the original problem using Boolean variables and CNF clauses. Solving the resulting SAT instance then provides a solution to the original problem, where a satisfying assignment can be decoded back into the corresponding solution.

For example, the graph 3-coloring problem can be encoded by introducing Boolean variables representing possible color assignments for each vertex and clauses enforcing the coloring constraints. The graph is 3-colorable if and only if the resulting SAT formula is satisfiable. Similar SAT encodings exist for many classical problems, including CLIQUE, Vertex Cover, and Hamiltonian Cycle. Additionally, constraint based puzzles such as Sudoku, Pattern, Mosaic, Minesweeper, Latin Squares, Flood-it and N-Queens can be also encoded as SAT and solved using SAT solvers (Bright, Gerhard, Kotsireas, & Ganesh, 2019; van Stiphout, 2020; van Oers, 2024; de Jong, 2023). This thesis uses Sudoku puzzles as sources of CNF formulas to evaluate the model counters.

Chapter 3

#SAT and Exact Model Counters

3.1 Introduction

Model counting (#SAT), introduced in Chapter 2, extends the Boolean satisfiability problem by requiring the computation of the total number of satisfying assignments of a Boolean formula. As a result, exact model counting is significantly more challenging and is #P-complete (Valiant, 1979). The exponential number of possible assignments makes explicit enumeration infeasible except for very small instances. Modern exact model counters therefore employ sophisticated search, decomposition, caching, and preprocessing techniques to reduce redundant computation while guaranteeing that every satisfying assignment is counted exactly once.

Several algorithmic paradigms have been developed for exact model counting. Search-based approaches extend SAT-solving techniques such as Davis–Putnam–Logemann–Loveland (DPLL) and Conflict-Driven Clause Learning (CDCL) with recursive counting, component decomposition, and caching. Knowledge compilation approaches instead transform a Boolean formula into a representation, such as a Binary Decision Diagram (BDD) or a Deterministic–Decomposable Negation Normal Form (d-DNNF) that supports efficient model counting (Darwiche & Marquis, 2002).

This chapter introduces the main algorithmic techniques underlying exact model counting and briefly describes the three model counters evaluated in this thesis.

3.2 Algorithmic Techniques for Exact Model Counting

Modern exact model counters employ a variety of algorithmic techniques to mitigate the exponential complexity of model counting. Although different tools combine these techniques in different ways, most are based on a few fundamental ideas:

- search-based reasoning derived from SAT solving,
- optimizations such as component decomposition and caching,
- knowledge compilation into tractable representations,
- exploiting structural properties through dynamic programming.

The following sections provide a brief overview of these techniques.

3.2.1 Search and SAT-Based Reasoning

Many exact model counters build upon search procedures developed for SAT solving. Since SAT solvers are highly optimized for exploring Boolean search spaces, model counters reuse many of the same mechanisms while replacing the objective of finding a satisfying assignment with counting all satisfying assignments.

DPLL-Based Counting

The Davis–Putnam–Logemann–Loveland (DPLL) algorithm is a recursive backtracking procedure for solving Boolean satisfiability problems (Davis, Logemann, & Loveland, 1962; Zhang & Stickel, 2000). DPLL maintains a partial assignment to the variables, denoted as *partial model* σ_p, of a CNF formula and repeatedly simplifies the formula using logical inference rules. In particular, it applies *unit propagation* and, when possible, *pure literal elimination*. If these simplification steps determine that all clauses of the formula are satisfied, the search terminates successfully. If any clause evaluates to false with the current partial assignment (results in an empty clause), conflict is detected and the current branch is unsatisfiable. Otherwise, DPLL selects an unassigned variable and makes a *branching decision*, recursively exploring the two possible truth assignments to that variable.

The basic search process can therefore be viewed as a sequence of *propagation*, *branching*, and *backtracking*. After every branching decision, the consequences of the chosen assignment are propagated through the formula: satisfied clauses are removed from the formula while clauses containing literals evaluating to false under the current partial assignments are simplified by removing the falsified literals. Propagation may force additional variable assignments without requiring further decisions. If a conflict is encountered, these assignments are reverted and the search returns to the most recent branching point, where the alternative truth value of the decision variable is explored. For SAT solving, the search can terminate as soon as one satisfying assignment is found. Figure 3.1 illustrates this process on a small CNF formula, where the initial unit clauses force assignments before an explicit branching decision is required.

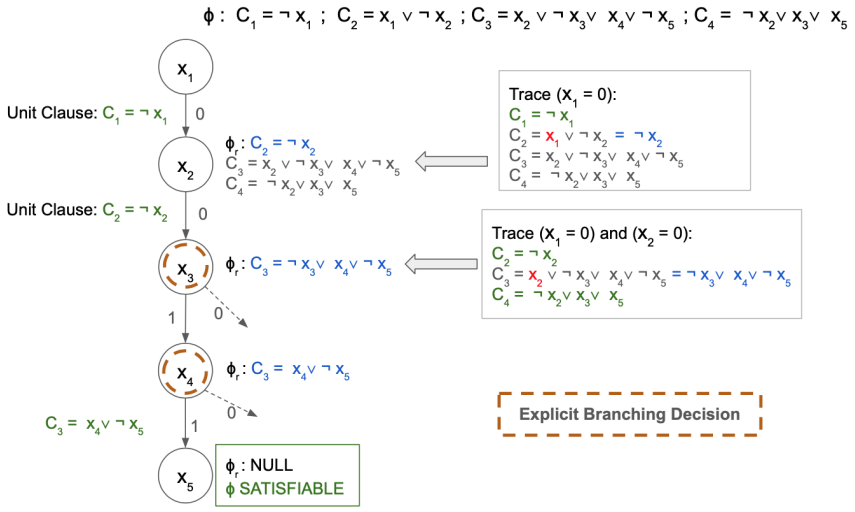


Figure 3.1: DPLL style SAT search

Boolean Constraint Propagation (BCP) A central component of DPLL is BCP, which derives assignments that are logically implied by the current partial assignment. In particular, a clause with exactly one unassigned literal is called a *unit clause*, and the remaining literal must be assigned the value true. This assignment may either remove satisfied clauses from consideration or simplify clauses and generate further unit clauses, resulting in a cascade of forced assignments. For example, in Figure 3.1, the initial unit clause $C_1 = \neg x_1$ forces $x_1 = 0$ and prevents search from exploring branch for $x_1 = 1$. As the goal is to find partial assignments that satisfy all clauses, $x_1 = 0$ also simplifies the clause $(x_1 \vee \neg x_2)$ to $(\neg x_2)$, allowing additional unit propagation. If propagation causes any clause to be false under the current partial assignment, a conflict is detected and the current branch is noted unsatisfiable. BCP is therefore applied after each branching decision until no further assignments can be derived or a conflict is detected. Efficient BCP is essential in practice and is commonly implemented using the *two-watched literals* scheme (Moskewicz, Madigan, Zhao, Zhang, & Malik, 2001).

Pure Literal Elimination Another simplification rule used in basic DPLL is the *pure literal rule*. A variable is pure if it occurs in the (residual) formula with only one polarity. For example, if an unassigned variable x_i occurs only as a positive literal, then assigning $x_i =$ `true` satisfies all clauses containing x_i. Consequently, the variable can be assigned its satisfying polarity without requiring a branching decision. Similarly, a variable occurring only negatively can be assigned false. In DPLL, unit propagation and pure literal elimination are applied first, and explicit branching decision are made only when necessary.

Branching Decisions When propagation of the partial assignment neither satisfies all clauses nor causes a conflict, DPLL selects an unassigned variable and makes an explicit *branching decision*. The search then considers two possible assignments of that variable. In the example in Figure 3.1, after propagation of x_1 and x_2, the residual formula does not contain any unit clauses, thus the algorithm selects x_3 as the next variable, creating $x_3 = 1$ and $x_3 = 0$ branches. In the example figure, the $x_3 = 1$ branch is subsequently simplified by propagation before another branching decision is made. The Search space for $x_3 = 1$ is explored only after fully exploring the $x_3 = 0$ branch. The choice of branching variable can substantially affect the size of the search tree, so practical solvers use branching heuristics to guide this decision.

Chronological Backtracking If a branch leads to a conflict, DPLL *backtracks* by undoing the assignments made since the last relevant branching decision and exploring the alternative branch. For example, assume that the example in figure 3.1 were slightly different, such that the residual formula contained $(x_4 \vee \neg x_5)$ along with other longer clauses, and that the solver chose to branch on x_4. Suppose that exploring the assignment $x_4 = 0$ resulted in the forced propagation of $x_5 = 0$, but eventually led to a conflict due to another clause. When this happens, the solver returns to the last explicit branching decision and reverses the assignment to x_4. The partial assignment and residual formula are reverted to the state at the x_4 branching level before the solver explores the $x_4 = 1$ branch.

Model Counting Extension Unlike SAT solvers, which terminate after finding a satisfying assignment or proving unsatisfiability, exact model counting must explore every relevant branch of the search tree because each branch may contain satisfying assignments. Early DPLL-based model counters include `CDP`, and `Relsat v2.00` (Birnbaum & Lozinskii, 1999; R. J. B. Jr. & Pehoushek, 2000; Sharma, Roy, Soos, & Meel, 2019).

Formally, for a formula ϕ containing variable x_1, the model count is computed recursively as

$$\#(\phi) = \#(\phi|_{x_1=true}) + \#(\phi|_{x_1=false}) \tag{3.1}$$

where $\phi|_{x_1=\texttt{true}}$ and $\phi|_{x_1=\texttt{false}}$ denote the formulas obtained after assigning x_1 to true and false, respectively. Figure 3.2 illustrates this recursion for the example CNF formula from Equation 2.3.

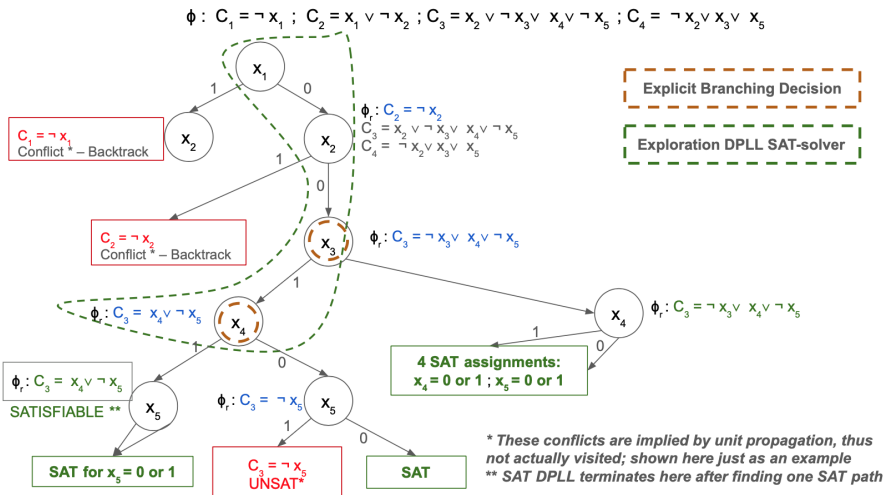


Figure 3.2: DPLL #SAT Exhaustive search

Although conceptually simple, basic DPLL scales poorly to large instances because it relies on relatively simple branching heuristics, does not learn from conflicts, and backtracks chronologically. Modern model counters therefore extend DPLL with more advanced techniques, including conflict-driven clause learning, component decomposition, and caching.

CDCL-Based Reasoning

Conflict-driven clause learning (CDCL) extends DPLL by learning from conflicts encountered during search (Silva & Sakallah, 1999). When a contradiction is detected, the solver analyzes the conflict, derives a *learned clause*, and performs non-chronological backtracking to the most relevant earlier decision level. The learned clause prevents the same conflicting assignment from being explored again, substantially reducing redundant search. Additionally, modern CDCL solvers such as `Chaff` also incorporate clever branching heuristics to select the next unassigned variable to branch on (Moskewicz et al., 2001). Clause learning is typically implemented using a directed acyclic graph (DAG) representation known as an *implication graph*; the details of this mechanism are beyond the scope of this thesis.

Although originally developed for SAT solving, CDCL has become a core component of many modern exact model counters. These counters combine efficient unit propagation, conflict analysis, clause learning, and branching heuristics inherited from SAT solvers with counting-specific techniques. Since learned clauses are logical consequences of the original formula, they preserve the set of satisfying assignments. However, clause learning alone is not sufficient for efficient model counting and suffers from some limitations as every satisfying assignment must still be counted.

Limitations of Pure Search-Based Counting

Although search-based approaches naturally extend SAT solving, they still suffer from the exponential growth of the search tree. Furthermore, many branches contain repeated or independent subproblems, leading to redundant computation.

Modern model counters address these limitations by combining search with additional optimizations. *Component decomposition* exploits conditional independence by partitioning a formula into independent components whose model counts can be computed separately and then multiplied. *Component caching* further improves efficiency by storing and reusing the model counts of previously solved subproblems. These optimizations are discussed next.

3.2.2 Component Decomposition

Many Boolean formulas contain independent substructures. If a formula can be partitioned into components that share no variables, each component can be solved independently and the resulting model counts then can multiplied.

Let a formula ϕ be partitioned into n disjoint subformulas:

$$\phi = \phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n$$

such that, for all $i \neq j$,

$$Vars(\phi_i) \cap Vars(\phi_j) = \emptyset.$$

Since the components share no variables, the satisfying assignments of one component are independent of those of the others. Therefore,

$$\#(\phi) = \#(\phi_1) \times \#(\phi_2) \times \cdots \times \#(\phi_n). \tag{3.2}$$

This allows one large counting problem to be replaced by several smaller independent subproblems. In practice, independent components rarely exist in the original formula but often emerge during search after simplifications resulting from constraint propagation and branching decisions. The residual formula can then be represented as a variable interaction graph, where disconnected components correspond to independent subproblems.

Component decomposition can significantly improve performance because solving several small components is often easier than solving the combined formula. The technique was first incorporated into exact model counting by `Relsat v2.00` (R. J. B. Jr. & Pehoushek, 2000; Gomes et al., 2009) and was subsequently adopted by `Cachet`, `sharpSAT`, `miniC2D`, and `Ganak` (Gomes et al., 2009; Sang, Bacchus, Beame, Kautz, & Pitassi, 2004; Thurley, 2006; Oztok & Darwiche, 2018; Sharma et al., 2019; Soos & Meel, 2025).

3.2.3 Component Caching

While component decomposition reduces the size of counting problems, identical subproblems may still arise multiple times during recursive search, resulting in redundant computation. During search, a partial assignment simplifies the original formula into a residual formula. Different branches of the search tree can produce identical residual formulas despite arising from different partial assignments. Rather than solving these formulas repeatedly, a model counter caches the model count associated with each unique subproblem and queries the cache when the same subproblem is encountered again.

Component caching is analogous to memoization in dynamic programming (DP), where solutions to previously solved subproblems are stored and reused. Unlike classical DP, however, reusable subproblems are discovered dynamically during the search rather than being defined in advance.

To maximize reuse, a model counter must efficiently determine when two subformulas are equivalent. In practice, this is typically achieved using hashing techniques that provide compact representations of subformulas and enable efficient cache lookup.

Combined with search-based reasoning and component decomposition, component caching forms the basis of several successful exact model counters, including `Cachet`, `sharpSAT`, and `Ganak` (Sang et al., 2004; Thurley, 2006; Gomes et al., 2009; Sharma et al., 2019; Soos & Meel, 2025). Its effectiveness depends on the trade-off between the overhead of maintaining the cache and the degree of cache reuse.

3.2.4 Knowledge Compilation

Knowledge compilation provides an alternative to search-based reasoning. Instead of repeatedly exploring the search space through branching and backtracking, it transforms a Boolean formula into a representation that supports efficient reasoning operations.

The main idea is to perform the computationally expensive work during a compilation phase, after which queries such as model counting can be answered efficiently, often in time polynomial in the size of the compiled representation (Gomes et al., 2009). Unlike search-based approaches, where repeated subproblems are discovered dynamically, knowledge compilation captures relevant structural properties explicitly in the compiled representation (Darwiche & Marquis, 2002; Darwiche, 2001).

Several target representations have been proposed. Early approaches include Binary Decision Diagrams (BDDs), which represent Boolean functions as directed decision graphs (Lee, 1959; S. B. A. Jr., 1978; Bryant, 1986). Extensions of BDDs, such as Algebraic Decision Diagrams (ADDs) allow terminal nodes to represent arbitrary values and are useful for weighted reasoning tasks (Slater, Meyer, & Heyninck, 2025; Bahar et al., 1997). Other important representations are based on Negation Normal Form (NNF), including Deterministic-Decomposable Negation Normal Forms (d-DNNFs) and Sentential Decision Diagrams (SDDs) (Darwiche & Marquis, 2002; Darwiche, 2001, 2011). By exploiting properties such as decomposability and determinism, these representations can achieve compactness while supporting efficient model counting and other reasoning tasks.

Binary Decision Diagrams (BDD)

Binary Decision Diagrams (BDDs) represent Boolean functions as directed acyclic graphs, where internal nodes correspond to variables and outgoing edges represent the two possible assignments of those variables. An Ordered BDD (OBDD) fixes the variable order along all its paths. Neither BDDs nor OBDDs are optimal and guarantee compactness of representation, as their size can be exponential in the number of variables in the worst case.

A Reduced Ordered BDD (ROBDD) applies reduction rules that merge equivalent subgraphs and remove redundant nodes. A key property of ROBDDs is their canonicity: for a fixed variable ordering, equivalent Boolean functions have identical ROBDD representations (Bryant, 1992).

This enables efficient operations such as equivalence checking, satisfiability testing, and model counting. Figure 3.3 shows an example BDD for a simple Boolean formula in (a) partially reduced and (b) reduced form. Note that the terminal node for 0 and incoming edges to it are removed in (b) as not relevant for model counting.

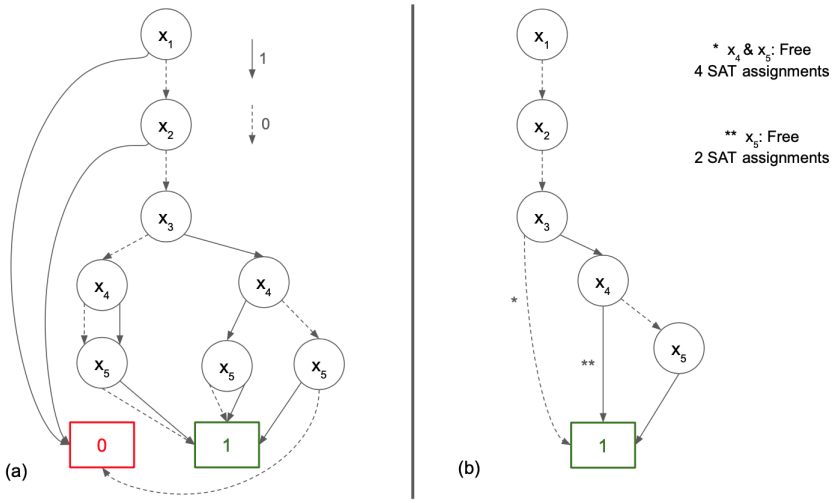


Figure 3.3: BDD Example with naive variable ordering: (a) Partially Reduced (b) Reduced

Model counting using a BDD is performed by recursively traversing the graph and computing the number of assignments represented by each node. Since equivalent subgraphs are shared, repeated computations are avoided automatically.

However, the size of a BDD is highly dependent on the chosen variable ordering and the structure of the input formula. A poor ordering can result in an exponential increase in the number of nodes, making compilation infeasible. Consequently, finding effective variable orderings is a major challenge in BDD-based approaches (Bryant, 1992). Furthermore, some CNF instances require exponentially large OBDD and ROBDD representations even under optimal variable orderings (Bryant, 1992; Wegener, 2000; Bova & Slivovsky, 2017).

Algebraic Decision Diagrams (ADD)

Algebraic Decision Diagrams (ADDs) generalize BDDs by allowing terminal nodes to store arbitrary values from a suitable algebraic domain, such as integers, real numbers, or probabilities, instead of only Boolean values (Bahar et al., 1997; Slater et al., 2025). Consequently, ADDs can represent functions that map Boolean assignments to arbitrary numerical values.

Similar to reduced ordered BDDs, reduced ordered ADDs are canonical for a fixed variable ordering. The shared subgraph structure allows efficient representation and evaluation of functions, while operations such as addition and multiplication can be performed recursively using the *Apply* procedure (Bahar et al., 1997).

ADDs are particularly relevant for *weighted model counting*, where satisfying assignments contribute different weights rather than a unit contribution. By storing these values in terminal nodes, the ADD can be evaluated to compute the total weight of satisfying assignments. However,

as with BDDs, the efficiency of ADD-based approaches depends strongly on variable ordering, and poor orderings may lead to exponentially large representations.

3.2.5 Dynamic Programming Approaches

Dynamic programming approaches exploit structural properties of Boolean formulas to reduce the complexity of model counting (Samer & Szeider, 2010a; Dudek, Phan, & Vardi, 2020). Instead of exploring individual assignments, these methods decompose the formula into smaller subproblems and combine their solutions to compute the final model count.

A common approach is based on *tree decompositions*. A CNF formula can be represented using a graph capturing interactions between variables, where the treewidth of this graph measures the complexity of the interaction structure (Robertson & Seymour, 1986; Bodlaender, 1993). For formulas with small treewidth, dynamic programming can efficiently compute model counts by exploiting the tree decomposition (Samer & Szeider, 2010a; Dudek et al., 2020). Treewidth and its relevance to CNF formulas are discussed in more detail in Chapter 6.

Dynamic programming algorithms associate intermediate results with nodes of the tree decomposition and combine these results while traversing the decomposition. This avoids repeatedly solving identical subproblems. The runtime is typically exponential in the treewidth but polynomial in the size of the formula (Dudek et al., 2020; Samer & Szeider, 2010a). Consequently, these approaches are particularly effective for formulas with small treewidth but may become impractical for arbitrary CNF instances with highly connected variable interactions.

3.2.6 Summary

This section presented the main algorithmic approaches used in modern exact model counting. Search-based methods extend SAT solving techniques such as DPLL and CDCL to count satisfying assignments rather than merely determining satisfiability. However, the exponential complexity of model counting requires additional optimizations to reduce redundant computation.

Component decomposition separates formulas into independent subproblems, while component caching avoids repeated computation of identical subproblems. Knowledge compilation transforms formulas into representations that support efficient counting, although compilation cost and representation size remain important challenges. Dynamic programming approaches exploit structural properties, such as bounded treewidth, to efficiently count suitable classes of instances.

Modern exact model counters typically combine multiple approaches rather than relying on a single method. The following section introduces the specific model counters evaluated in this thesis and the techniques they employ.

3.3 Exact Model Counters Evaluated in This Thesis

This thesis evaluates three open-source exact model counters: `Ganak`, `CNF2OBDD`, and `ADDMC`. These counters represent three different algorithmic approaches to exact model counting.

`Ganak` is primarily a SAT-search based counter that combines CDCL reasoning with techniques such as component decomposition, component caching, and specialized branching heuristics.

`CNF2OBDD` combines SAT-based search with knowledge compilation using OBDDs. In contrast, `ADDMC` follows a dynamic programming approach based on tree decompositions and uses ADDs to represent intermediate computations.

The selected counters provide a diverse set of approaches for evaluating the influence of Sudoku-to-CNF encoding strategies and count-preserving preprocessing. Since they rely on different underlying techniques, their performance may be affected differently by structural properties of CNF instances, such as formula size, clause density, and treewidth. The following sections describe each model counter in more detail, focusing on the characteristics relevant to this thesis.

3.3.1 `Ganak`

Ganak extends previous search-based counters such as `Cachet` and `sharpSAT` by combining CDCL-based SAT reasoning with component decomposition, component caching, and improved search heuristics (Sharma et al., 2019).

Algorithmic Contributions

Ganak introduces several improvements to search-based model counting. The techniques most relevant to this thesis are component caching and improved search heuristics. While Ganak is inherently a search solver, it provides a utility feature with the `--compile` flag that tracks its search path to output a valid d-DNNF file, allowing counting, sampling, and other queries to be performed independently.

Component caching. Ganak improves traditional component caching through *Probabilistic Component Caching* (PCC). Instead of requiring exact comparison of residual formulas, PCC uses hashing to identify previously encountered components and retrieve their associated model counts. This increases cache efficiency while introducing a small bounded probability of incorrect results due to hash collisions (Sharma et al., 2019).

Branching and search heuristics. Variable selection has a significant influence on the search tree and the effectiveness of decomposition. Ganak introduces improved branching heuristics, including the *Cache State and Variable State Aware Decaying Sum* (CSVSADS) variable branching heuristic and improved phase selection strategies, to select variables and assignments, respectively, that simplify the remaining formula more effectively (Sharma et al., 2019). Additional heuristics, such as Independent Support (IS), further improve search performance. Ganak also introduces search strategies such as *Exponentially Decaying Randomness* (EDR) and *Learn and Start Over* (LSO), which improve exploration of the search space through controlled randomness and improved restart strategies by learning from previous restarts (Sharma et al., 2019). The details of these heuristics are beyond the scope of this thesis.

Engineering Improvements

The subsequent version, `Ganak2`, preserves the same search-based counting framework while improving the efficiency of search and residual formula processing (Soos & Meel, 2025). It incorporates additional SAT-solving techniques, including VSIDS-based variable scoring, Luby restart strategies, and polarity caching, to improve recursive search performance.

Ganak2 also separates variables used for SAT-level reasoning from variables used for branching decisions, allowing more effective search decisions. Additionally, it adapts chronological backtracking techniques for the model counting setting while preserving useful learned information (Soos & Meel, 2025).

Relevance to This Thesis

Unless otherwise noted, this thesis uses Ganak to denote the latest version, Ganak2. Among the evaluated counters, Ganak represents the search-based approach to exact model counting. Its performance is influenced by SAT reasoning, branching decisions, component decomposition, and caching behavior. Therefore, changes in CNF structure caused by different Sudoku encodings or preprocessing may affect its performance by changing the available decomposition opportunities, cache reuse, and search complexity.

The version of Ganak used in this thesis additionally performs Arjun preprocessing internally before counting. Consequently, this preprocessing step should be considered when interpreting the experimental results.

3.3.2 BDD-Based Model Counter (CNF2OBDD or BDD_MINISAT_ALL)

CNF2OBDD, also distributed as BDD_MINISAT_ALL, is an exact model counter that combines SAT solving with OBDD (fully reduced OBDD with optional flag) construction. The tool extends the MiniSat SAT solver with AllSAT capabilities and uses OBDDs to represent and cache recurring subproblems during search (Toda, n.d.; Toda & Soh, 2016; Toda & Tsuda, 2015).

The solver combines two approaches discussed in Section 3.2: CDCL-based search for enumerating satisfying assignments and knowledge compilation for constructing partially reduced OBDD representation of the Boolean function. When sufficient structural sharing exists, the resulting decision diagram provides a compact representation of the solution space.

Algorithmic Approach

CNF2OBDD is built on MiniSat (Eén & Sörensson, 2003; Eén, Mishchenko, & Amla, 2010), inheriting techniques such as unit propagation, clause learning, and conflict analysis. Unlike conventional SAT solving, however, CNF2OBDD follows the AllSAT paradigm and continues enumeration until all satisfying assignments have been identified (Toda & Soh, 2016).

During this process, assignments and residual formulas are incrementally compiled into an OBDD, allowing it to avoid explicitly storing every satisfying assignment by merging equivalent substructures (Toda & Tsuda, 2015).

Algorithmic Contributions

CNF2OBDD incorporates several techniques to improve AllSAT-based OBDD construction. A key design choice is *non-blocking enumeration*, which avoids adding blocking clauses for previously found assignments. Instead, specialized backtracking procedures are used, reducing the overhead associated with repeatedly modifying the clause database (Toda & Soh, 2016). The solver also uses non-chronological backtracking inherited from MiniSat's CDCL implementation, allowing the search to return directly to relevant decision levels after conflicts rather than only the most recent decision level (Eén & Sörensson, 2003; Toda & Soh, 2016).

Another important optimization is *Formula–BDD caching*, which stores previously constructed OBDD representations for residual formulas encountered during search. When an equivalent residual formula is found again, the stored representation can be reused instead of being reconstructed. This approach is related to component caching but is applied directly to incremental OBDD construction (Toda & Tsuda, 2015). Additional optimizations, such as lazy caching and finer cutset caching, further improve this process (Toda & Soh, 2016). Details of these and several other optimizations are beyond the scope of this thesis.

Relevance to This Thesis

`CNF2OBDD` represents the knowledge compilation approach among the evaluated counters. While `Ganak` primarily relies on SAT-based search and `ADDMC` uses dynamic programming over tree decompositions, `CNF2OBDD` combines SAT enumeration with incremental OBDD construction.

Its performance depends on both SAT search efficiency and the size of the resulting OBDD. Since OBDD size is strongly influenced by variable ordering and the structural properties of the input formula (Wegener, 2000; Bova & Slivovsky, 2017), changes introduced by different Sudoku encodings or count-preserving preprocessing may affect compilation size and efficiency.

3.3.3 Algebraic-Decision-Diagram Model Counter (`ADDMC`)

`ADDMC` is an exact model counter based on dynamic programming over a tree decomposition of the input CNF formula. Although originally developed for literal-weighted model counting, it also supports unweighted model counting, the configuration used in this thesis (Dudek et al., 2020).

Unlike the previous two model counters, `ADDMC` exploits the structural properties of the input formula through tree decompositions. Intermediate computations are represented using ADDs, allowing shared subfunctions to be represented compactly.

Algorithmic Approach

`ADDMC` follows the dynamic programming framework introduced in Section 3.2.5. The algorithm first obtains a variable ordering from a tree decomposition of the input formula and partitions clauses into clusters according to this decomposition. For each cluster, an ADD representing the corresponding subfunction is constructed. These ADDs are then combined incrementally using algebraic operations while variables are eliminated according to the decomposition structure (Dudek et al., 2020).

Variables are projected during the computation whenever possible rather than only at the end. Early projection can reduce the size of intermediate ADDs and improve runtime and memory efficiency, while preserving correctness (Dudek et al., 2020).

Algorithmic Contributions

The main contribution of `ADDMC` is the integration of ADD representations into dynamic programming for exact model counting. Instead of storing intermediate tables explicitly, `ADDMC` represents intermediate functions symbolically, reducing memory usage when significant structural sharing exists (Bahar et al., 1997; Dudek et al., 2020).

Practical performance is influenced by several heuristics, including tree decomposition construction, variable ordering, clause-to-cluster assignment, and cluster processing order. These choices affect the size of intermediate ADDs and therefore the overall runtime (Dudek et al., 2020). Detailed discussion of these heuristics is beyond the scope of this thesis.

Limitations

Although `ADDMC` can efficiently exploit structural properties of formulas with small treewidth, performance depends strongly on the quality of the tree decomposition and associated variable orderings. Poor decompositions may lead to large intermediate ADDs and reduce the benefits of symbolic compression (Dudek et al., 2020). Since finding an optimal tree decomposition is NP-hard (Arnborg, Corneil, & Proskurowski, 1987; Fichte, Lodha, & Szeider, 2017), practical approaches rely on heuristic decomposition methods.

Relevance to This Thesis

Among the evaluated counters, `ADDMC` represents the dynamic programming approach to exact model counting. Its performance is closely related to structural properties of the input CNF formula, particularly treewidth and the resulting tree decomposition, which are discussed in Chapter 6.

Since count-preserving preprocessing can alter the structural characteristics of CNF formulas, it may affect the resulting decomposition, intermediate ADD sizes, and ultimately `ADDMC` performance.

3.4 Chapter Summary

This chapter introduced the main algorithmic approaches used in exact model counting and described the three model counters evaluated in this thesis.

Modern exact counters combine multiple techniques to overcome the exponential complexity of counting. Search-based approaches extend SAT solving methods such as DPLL and CDCL to count satisfying assignments, while component decomposition and caching reduce redundant computation. Knowledge compilation approaches transform formulas into representations such as BDDs and ADDs, and dynamic programming methods exploit structural properties such as bounded treewidth.

The evaluated counters represent three different approaches: `Ganak` as a SAT-based search counter, `CNF2OBDD` as a hybrid counter combining search-based enumeration with OBDD-based knowledge compilation, and `ADDMC` as a dynamic programming counter using ADDs for symbolic representation. Their algorithmic differences make them suitable for analyzing how CNF encoding strategies and count-preserving preprocessing influence model counting performance.

The next chapter introduces the benchmark problem domain and describes the SAT encoding techniques used to generate the CNF instances evaluated in the experiments.

Chapter 4

SAT encoding of Sudoku

This chapter presents the CNF encoding of Sudoku puzzles used in the experiments. After motivating the choice of Sudoku as a benchmark domain, the Boolean representation of Sudoku is introduced, followed by the construction of the minimal and extended CNF encodings considered in this work. The chapter concludes by describing relevant structural properties of the resulting CNF formulas that are relevant to model counting.

4.1 Sudoku as a Benchmark Domain

Sudoku is a constraint-based puzzle that gained worldwide popularity in the mid-2000s after being introduced in its modern form in 1986 (Lynce & Ouaknine, 2006). Due to its rich and highly structured constraints, Sudoku has become a common benchmark problem in constraint programming, Boolean satisfiability (SAT), and automated reasoning (Simonis, 2005; Lynce & Ouaknine, 2006; Ercsey-Ravasz & Toroczkai, 2012; Defresne, Barbe, & Schiex, 2023).

A standard Sudoku puzzle consists of a 9×9 grid divided into nine 3×3 subgrids, commonly referred to as *blocks*. Some cells contain predefined values, called *clues* or *givens*, while the remaining cells are empty. The objective is to assign digits $1, \ldots, 9$ to all cells such that:

1. each cell contains exactly one digit,
2. each digit appears exactly once in every row,
3. each digit appears exactly once in every column, and
4. each digit appears exactly once in every 3×3 block.

Figure 4.1 shows an example standard Sudoku puzzle (Lynce & Ouaknine, 2006). Sudoku is well suited as a benchmark for evaluating model counters for several reasons. First, its constraint structure is fixed and independent of individual puzzle instances, allowing different encoding schemes to be compared without introducing additional structural differences. Second, instances can be generated by removing clues to generate CNF formulas with different numbers of satisfying assignments. Finally, Sudoku encodings exhibit high structural regularity while remaining sufficiently challenging for modern SAT and #SAT solvers.

The generalized Sudoku problem on an $n^2 \times n^2$ grid is NP-complete (Yato & Seta, 2003). The standard 9×9 Sudoku puzzle corresponds to the special case where $n = 3$.

	1	8				7		
			3			2		
	7							
				7	1			
6							4	
3								
4			5					3
	2			8				
							6	

Figure 4.1: Example Sudoku puzzle

4.2 Boolean Representation

The first step in encoding a Sudoku puzzle as a Boolean satisfiability problem is to define Boolean variables representing possible digit assignments to cells. This thesis adopts the classical variable representation introduced by Lynce and Ouaknine (Lynce & Ouaknine, 2006) due to its simplicity, intuitive interpretation, and widespread use in SAT-based Sudoku encodings.

For every row r, column c, and digit d, where $r, c, d \in \{1, \ldots, 9\}$, a Boolean variable x_{rcd} is introduced. Its interpretation is:

$$x_{rcd} = \begin{cases} \text{true}, & \text{if digit } d \text{ is assigned to cell } (r, c), \\ \text{false}, & \text{otherwise.} \end{cases}$$

A standard 9×9 Sudoku puzzle thus results in $9 \times 9 \times 9 = 729$ Boolean variables. Unlike many SAT encodings, this variable set is independent of the specific puzzle instance. Every puzzle uses the same variables and Sudoku constraints; only the clauses representing the given clues differ between instances.

This representation establishes a one-to-one correspondence between valid Sudoku solutions and satisfying assignments of the complete CNF encoding. Each Sudoku solution determines exactly one assignment to the Boolean variables, and every satisfying assignment of the complete encoding corresponds to a valid Sudoku solution.

The variables can also be viewed as a three-dimensional Boolean array indexed by row, column, and digit. Figure 4.2 illustrates two dimensions of this interpretation (Weber, 2005). For example, the variable x_{257} represents the assignment of digit 7 to the cell in the second row and fifth column. If this variable is true in a satisfying assignment, the corresponding cell contains digit 7. If it is false, the variable only indicates that this particular assignment is not selected; additional constraints are required to determine the digit assigned to the cell.

x_{11}	x_{12}	x_{13}	x_{14}	x_{15}	x_{16}	x_{17}	x_{18}	x_{19}
x_{21}	x_{22}	x_{23}	x_{24}	x_{25}	x_{26}	x_{27}	x_{28}	x_{29}
x_{31}	x_{32}	x_{33}	x_{34}	x_{35}	x_{36}	x_{37}	x_{38}	x_{39}
x_{41}	x_{42}	x_{43}	x_{44}	x_{45}	x_{46}	x_{47}	x_{48}	x_{49}
x_{51}	x_{52}	x_{53}	x_{54}	x_{55}	x_{56}	x_{57}	x_{58}	x_{59}
x_{61}	x_{62}	x_{63}	x_{64}	x_{65}	x_{66}	x_{67}	x_{68}	x_{69}
x_{71}	x_{72}	x_{73}	x_{74}	x_{75}	x_{76}	x_{77}	x_{78}	x_{79}
x_{81}	x_{82}	x_{83}	x_{84}	x_{85}	x_{86}	x_{87}	x_{88}	x_{89}
x_{91}	x_{92}	x_{93}	x_{94}	x_{95}	x_{96}	x_{97}	x_{98}	x_{99}

Figure 4.2: Interpretation of the Boolean variable x_{rcd}. The variable is true if and only if digit d is assigned to cell (r, c).

4.3 Encoding Sudoku Rules

Having introduced the Boolean variables representing possible digit assignments, the next step is to encode the Sudoku rules as a CNF formula, without yet considering clues. The goal is to construct a formula whose satisfying assignments correspond exactly to valid Sudoku solutions.

The encoding is divided into four groups of constraints corresponding to the fundamental Sudoku rules. Throughout this section, the notation introduced in Section 4.2 is used. Unless otherwise stated, $r, c, d \in \{1, \ldots, 9\}$.

4.3.1 Cell Constraints

The first rule of Sudoku requires that every cell contain exactly one digit. This requirement can naturally be decomposed into two separate conditions:

1. every cell contains *at least one* digit (Existence);
2. every cell contains *at most one* digit (Uniqueness).

At least one digit per cell

For a given cell (r, c), at least one of the nine corresponding variables must be assigned the value *true*. This is expressed by the clause

$$\bigvee_{d=1}^{9} x_{rcd}.$$

For the complete grid, the following set of clauses define this constraint for all cells:

$$\bigwedge_{r=1}^{9} \bigwedge_{c=1}^{9} \left(\bigvee_{d=1}^{9} x_{rcd} \right) \tag{4.1}$$

Since the Sudoku board consists of 81 cells, this constraint produces exactly 81 clauses, each containing nine literals (nine-ary clauses).

At most one digit per cell

To prevent a cell from containing multiple digits simultaneously, every pair of distinct digits cannot both be true. For a given cell (r, c), every pair of digits $d_1, d_2 \in \{1, \ldots, 9\}$ satisfying $d_1 < d_2$, the binary clause

$$\neg x_{rcd_1} \vee \neg x_{rcd_2}$$

is introduced. For the complete grid, the following set of clauses is added:

$$\bigwedge_{r=1}^{9} \bigwedge_{c=1}^{9} \left(\bigwedge_{d_1=1}^{8} \bigwedge_{d_2=d_1+1}^{9} (\neg x_{rcd_1} \vee \neg x_{rcd_2}) \right) \tag{4.2}$$

Each cell contains $\binom{9}{2} = 36$ unordered pairs of digits, resulting in $81 \times 36 = 2916$ binary clauses. Together, Equations (4.1) and (4.2) ensure that every cell contains exactly one digit.

4.3.2 Row Constraints

The second Sudoku rule requires every digit to appear exactly once in each row. As before, this property is expressed using two separate constraints.

At least one digit per row

For a given row r and digit d, at least one cell within the row must contain the digit. The corresponding CNF clause is:

$$\bigvee_{c=1}^{9} x_{rcd}$$

For all rows and all possible digits, the following set of clauses is added.

$$\bigwedge_{r=1}^{9} \bigwedge_{d=1}^{9} \left(\bigvee_{c=1}^{9} x_{rcd} \right) \tag{4.3}$$

Since there are nine rows and nine digits, this produces $9 \times 9 = 81$ nine-ary clauses.

At most one digit per row

To prevent duplicate occurrences of a digit within the same row, every pair of distinct columns must be mutually exclusive. For all pairs of columns $c_1, c_2 \in \{1, \ldots, 9\}$ satisfying $c_1 < c_2$ in a given row r and for a given digit d, following clause is generated:

$$\neg x_{rc_1d} \vee \neg x_{rc_2d}$$

Then for the complete grid, set of clauses

$$\bigwedge_{r=1}^{9}\bigwedge_{d=1}^{9}\left(\bigwedge_{c_1=1}^{8}\bigwedge_{c_2=c_1+1}^{9}(\neg x_{rc_1d}\vee\neg x_{rc_2d})\right) \tag{4.4}$$

is added.

Since every row contains $\binom{9}{2}=36$ pairs of columns for each of the nine digits, the uniqueness constraint of the row generate $9\times 9\times 36=2916$ binary clauses.

4.3.3 Column Constraints

Column constraints are analogous to the row constraints. Every digit must appear exactly once in each column.

For the complete grid, the *existence* constraint produces 81 nine-ary clauses as follows:

$$\bigwedge_{c=1}^{9}\bigwedge_{d=1}^{9}\left(\bigvee_{r=1}^{9}x_{rcd}\right) \tag{4.5}$$

The *uniqueness* constraint is enforced by the following set of 2916 binary clauses:

$$\bigwedge_{c=1}^{9}\bigwedge_{d=1}^{9}\left(\bigwedge_{r_1=1}^{8}\bigwedge_{r_2=r_1+1}^{9}(\neg x_{r_1cd}\vee\neg x_{r_2cd})\right) \tag{4.6}$$

4.3.4 Block Constraints

The final Sudoku rule requires that every digit occur exactly once within each 3×3 block. Let b denote the set of nine cells belonging to a particular block. For a given block b and digit d, the existence constraint is

$$\bigvee_{(r,c)\in b}x_{rcd}$$

which again contributes 81 nine-ary clauses that can be shown as general formula below.

$$\bigwedge_{b=1}^{9}\bigwedge_{d=1}^{9}\left(\bigvee_{(r,c)\in b}x_{rcd}\right) \tag{4.7}$$

However, to ensure uniqueness, every pair of distinct cells belonging to the same block must not simultaneously contain the same digit. Thus, for a given pair of cells $(r_1,c_1),(r_2,c_2)\in b$, the clause

$$\neg x_{r_1c_1d}\vee\neg x_{r_2c_2d}$$

is generated for each digit. Since each block contains nine cells, the uniqueness constraints contribute $9\times 9\times\binom{9}{2}=2916$ binary clauses that can be written as a general formula below.

$$\bigwedge_{b=1}^{9}\bigwedge_{d=1}^{9}\left(\bigwedge_{\substack{(r_1,c_1)\in b\\(r_2,c_2)\in b\\(r_1,c_1)<(r_2,c_2)}}(\neg x_{r_1c_1d}\vee\neg x_{r_2c_2d})\right) \tag{4.8}$$

The constraints presented above completely characterize valid completions of an empty Sudoku grid. Any satisfying assignment of the resulting formula corresponds to a valid completion of the Sudoku grid, and every valid Sudoku solution induces exactly one satisfying assignment. Two SAT encoding schemes are obtained by either selecting minimal subsets of these constraints or by introducing additional redundant constraints that preserve logical equivalence while potentially improving solver performance. The minimal and extended encoding schemes considered in this thesis are described in the following sections.

4.4 Encoding the Empty Sudoku Grid

The constraints presented in Section 4.3 define a complete CNF encoding of an empty Sudoku grid. Since several constraint families are logically implied by others, different encodings can be obtained by including or omitting these redundant constraints without changing the set of satisfying assignments. This work considers the two encodings introduced by Lynce and Ouaknine (Lynce & Ouaknine, 2006): the *minimal encoding*, which contains only a sufficient subset of constraints, and the *extended encoding*, which additionally includes redundant constraints to potentially strengthen constraint propagation.

Table 4.1: Constraint families included in the considered Sudoku encodings.

Constraint type	Equation	Clauses		Minimal	Extended	Total
		Nine-ary	Binary			
Cell existence	Eq. 4.1	81	–	✓	✓	81
Cell uniqueness	Eq. 4.2	–	2916		✓	2916
Row existence	Eq. 4.3	81	–		✓	81
Row uniqueness	Eq. 4.4	–	2916	✓	✓	2916
Column existence	Eq. 4.5	81	–		✓	81
Column uniqueness	Eq. 4.6	–	2916	✓	✓	2916
Block existence	Eq. 4.7	81	–		✓	81
Block uniqueness	Eq. 4.8	–	2916	✓	✓	2916
Minimal encoding		**81**	**8748**			**8829**
Extended encoding		**324**	**11664**			**11988**

Table 4.1 above summarizes the constraint types included in both encoding schemes together with their contribution to the total number of CNF clauses. As an empty Sudoku grid admits 6.67×10^{21} valid solutions where solutions related by symmetry are counted as distinct (Felgenhauer & Jarvis, 2005), the CNF formula encoding empty grid will have the same number of satisfying assignments before incorporating clues.

4.4.1 Minimal Encoding

The minimal encoding proposed by Lynce and Ouaknine (Lynce & Ouaknine, 2006) omits constraint families that are logically implied by the remaining formula. Consequently, it produces a smaller CNF representation while preserving the same set of satisfying assignments, and therefore the same model count.

For example, the row existence constraint is implied by the remaining constraint families in the minimal encoding. The proof of this implication is non-trivial and beyond the scope of this thesis. Similar implications apply to the omitted column and block constraints in the minimal encoding.

The resulting encoding contains 81 nine-ary clauses and 8748 binary clauses, for a total of 8829 clauses.

4.4.2 Extended Encoding

The extended encoding contains all constraint families introduced in Section 4.3, including those that are logically redundant. Although these additional clauses do not change satisfiability or the model count, they can strengthen Boolean constraint propagation.

For Sudoku SAT solving, Lynce and Ouaknine (Lynce & Ouaknine, 2006) demonstrated that including redundant constraints can improve solver performance despite increasing the size of the CNF representation. More generally, redundant clauses are known to strengthen Boolean constraint propagation (Alsinet, Béjar, Cabiscol, Fernández, & Manyà, 2002) and improve the effectiveness of local search (Cha & Iwama, 1996; Kask & Dechter, 1995). However, the practical benefit of redundant constraints depends on the solver and instance characteristics (Luo, Li, Xiao, Manyà, & Lü, 2017).

The impact of redundant constraints on exact model counting is less well understood. Unlike SAT solvers, modern #SAT solvers rely not only on Boolean constraint propagation, but also on search, component decomposition, and component caching to efficiently count satisfying assignments (Gomes et al., 2009). Therefore, improvements observed for SAT solving cannot be assumed to transfer directly to model counting. Evaluating the impact of the minimal and extended Sudoku encoding schemes on exact model counting and count-preserving CNF preprocessing is consequently one of the objectives of this thesis.

The extended encoding for Sudoku contains 324 nine-ary clauses and 11664 binary clauses, yielding a total of 11988 clauses.

4.5 Encoding Sudoku Instances

The previous sections described how an empty Sudoku board can be translated into a CNF formula using either the minimal or extended encoding. In practice, however, Sudoku puzzles contain pre-filled cells, referred to as *clues*. These clues restrict the set of valid solutions and distinguish individual puzzle instances.

This section describes how clues are incorporated into the Boolean encoding and explains the procedure used to generate the benchmark formulas evaluated in this thesis.

4.5.1 Incorporating Given Clues

The constraints introduced in the previous sections encode the general rules of Sudoku independently of a specific puzzle instance. To represent a concrete puzzle, the given clues are added to the CNF formula as additional constraints.

Suppose a puzzle specifies that the cell at row r and column c contains digit d. This assignment is encoded by adding the unit clause

$$x_{r,c,d}.$$

Since a unit clause forces its literal to be assigned *true*, the corresponding digit assignment is fixed in every satisfying assignment of the formula. During SAT solving or preprocessing, this assignment can propagate through the remaining constraints via unit propagation, eliminating inconsistent assignments and simplifying the formula.

Therefore, each clue contributes exactly one unit clause to the encoding. The number of variables remains unchanged, while the total number of clauses increases by the number of given clues.

From a model counting perspective, adding clues reduces the number of satisfying assignments by restricting the solution space. A fully solved Sudoku puzzle has a unique satisfying assignment, whereas puzzles with fewer clues may admit multiple valid completions. The relationship between the number of clues and the number of solutions is non-linear and depends on the positions of the clues rather than only their quantity.

4.5.2 Generation of Modified Sudoku Instances

The objective of this thesis is not merely to evaluate model counters on classical Sudoku puzzles with unique solutions. Instead, benchmark instances are generated to produce CNF formulas with varying numbers of satisfying assignments while preserving the underlying Sudoku constraints. More details on this process are described in Chapter 7.

4.6 Properties of the Generated CNF Formulas

The SAT encodings described in the previous sections transform Sudoku instances into structured CNF formulas. Although the minimal and extended encodings represent the same solution space, they differ in their structural characteristics. These differences may influence the behavior of exact model counters, as modern #SAT algorithms rely on exploiting structural properties of the input formula.

This section introduces the main characteristics of the generated CNF formulas considered in this thesis. A more detailed discussion of structural parameters, including graph representations and treewidth, is provided in Chapter 6.

4.6.1 Formula Size

The number of variables and clauses provides a basic measure of CNF formula size and is commonly used when comparing SAT and #SAT encoding schemes.

For both Sudoku encoding schemes before preprocessing, the number of variables remains constant at $n = 729$. The number of clauses, however, depends on the chosen encoding. The minimal encoding contains fewer clauses due to the removal of logically redundant constraints, whereas the extended encoding includes additional clauses intended to improve propagation strength. Table 4.2 summarizes the expected formula sizes after incorporating Sudoku clues. A detailed breakdown of the individual constraints is provided in Table 4.1.

Table 4.2: Size comparison of the minimal and extended Sudoku encodings.

Encoding	Variables	Clauses		
		Nine-ary	Binary	Unary
Minimal encoding	729	81	8748	17–28*
Extended encoding	729	324	11664	17–28*

* Unary clauses correspond to Sudoku givens as well as number of clues removed and therefore range from 17 to 28 in the benchmark instances.

The difference in clause count between the two encodings may affect solver performance in different ways. Additional clauses can provide more opportunities for propagation, but may also increase the computational overhead of processing the formula.

After preprocessing, the number of variables and clauses is no longer fixed and depends on the individual instance and encoding. These changes are analyzed for both encoding schemes in Chapter 8.

4.6.2 Clause Distribution

In addition to the total number of clauses, the distribution of clause lengths is an important characteristic of a CNF formula. The Sudoku encodings considered in this thesis primarily consist of binary and nine-ary clauses, as summarized in Table 4.2.

Binary clauses are particularly important for SAT-based algorithms because they are particularly effective for unit propagation. For example, unit clauses introduced by Sudoku clues may simplify binary clauses into new unit clauses, creating further propagation opportunities.

Nine-ary clauses originate from the "at least one" constraints. The minimal and extended encodings differ mainly in the number of Nine-ary clauses. Since clause distribution affects propagation and preprocessing behavior, it may also influence the performance of model counters.

4.6.3 Number of Satisfying Assignments

The primary quantity of interest in this thesis is the number of satisfying assignments of the generated CNF formulas. The CNF encoding of Sudoku considered in this thesis is *parsimonious*, meaning the model count of the CNF formula equals the number of valid Sudoku solutions of the corresponding puzzle.

Traditional Sudoku puzzles are typically designed to have exactly one solution, resulting in $\#\phi = 1$. However, the benchmark generation procedure used in this thesis intentionally creates instances with multiple solutions by removing selected clues.

The number of solutions may influence the difficulty of model counting, but the relationship is not necessarily linear. A formula with less models is not always easier to count because the solver must still establish that no additional models exist. Conversely, highly structured formulas with many solutions may be easier when the structure allows effective decomposition or caching.

4.6.4 Variable Interaction Structure

The variables and clauses of a CNF formula define an underlying interaction structure. Variables that occur together in clauses are directly related. For Sudoku formulas, this structure is highly regular: variables representing digit assignments to cells within the same row, column, or block are connected through shared constraints. Consequently, Sudoku formulas contain strong global structure compared with randomly generated SAT instances.

These structural properties are relevant for modern #SAT solvers, which often exploit decomposition and component caching. If a formula can be separated into independent components, component decomposition and caching can be exploited. Therefore, variable interaction properties may provide insight into differences observed in solver performance.

A more formal treatment of these properties, including primal graph representation and treewidth, is presented in Chapter 6.

4.7 Chapter Summary

This chapter presented the Sudoku encoding schemes used to generate the CNF formulas evaluated in this thesis. Sudoku was introduced as a structured benchmark domain that provides a controlled setting for generating Boolean formulas with varying numbers of satisfying assignments.

A Boolean representation based on variables $x_{r,c,d}$ was introduced, with each variable represents a possible digit assignment to a Sudoku cell. The Sudoku rules were then translated into CNF constraints describing cell, row, column, and block constraints. Two CNF encoding schemes, minimal and extended, were considered. The extended encoding retains redundant constraints, which may affect the performance of preprocessing and model counting algorithms.

Finally, several characteristics of the generated formulas were discussed, including formula size, clause distribution, model count, and variable interaction structure. These properties provide the basis for analyzing the relationship between CNF structure and the performance of exact model counters.

Chapter 5

Pre-processing CNF-formulas

5.1 Introduction and Motivation

Preprocessing is an important component of modern SAT solving. Before the search begins, a preprocessing algorithm transforms the input CNF formula into an equivalent but often simpler representation. The objective is to reduce formula size, simplify constraints, and expose structural properties that may improve solver performance (Biere, Järvisalo, & Kiesl, 2021).

The distinction between preprocessing and inprocessing is based on when these transformations are applied. Preprocessing is performed before the search procedure starts, whereas inprocessing applies similar simplification techniques dynamically during solving. Although many techniques can be used in both settings, preprocessing remains a standard component of modern SAT solving pipelines (Biere et al., 2021).

Over the past two decades, numerous preprocessing techniques have been developed, ranging from simple logical simplifications to sophisticated redundancy elimination methods. Together, these techniques can substantially reduce the search space and improve the performance of state-of-the-art SAT solvers (Biere et al., 2021; Eén & Biere, 2005).

Most SAT preprocessing techniques are designed to preserve *satisfiability*. That is, the transformed formula is satisfiable if and only if the original formula is satisfiable. This property is sufficient for SAT solving; however, exact model counting requires a stronger correctness condition to also preserve the model count.

Transformations that eliminate variables or modify constraints may preserve satisfiability while changing the number of satisfying assignments, making them unsuitable for direct use in exact model counting. However, some SAT preprocessing techniques have been adapted for count-preserving preprocessing or used as intermediate optimizations within counting algorithms (Biere et al., 2021).

This chapter provides an overview of preprocessing techniques commonly used in SAT solving and discusses selected techniques relevant to exact model counting. Finally, the `Arjun` preprocessor used in the experimental evaluation of this thesis is introduced.

5.2 Overview of SAT Preprocessing Techniques

Modern SAT preprocessors combine numerous simplification techniques that differ in their underlying principles and objectives. These techniques may operate directly on the syntactic structure of a CNF formula or exploit additional representations, such as implication graphs, functional dependencies, and higher-level constraints implicitly encoded by the clauses (Biere et al., 2021).

Modern preprocessors described in the literature and implemented in open-source tools typically combine several of these techniques rather than relying on a single transformation. This section provides a high-level overview of the main categories of SAT preprocessing techniques described in Chapter 9 of the *Handbook of Satisfiability (2021)* (Biere et al., 2021). Since the primary focus of this thesis is count-preserving preprocessing for exact model counting, a detailed discussion of every SAT preprocessing technique is beyond the scope of this work. Readers interested in a comprehensive overview are referred to the handbook.

The remainder of this chapter focuses on preprocessing techniques that are relevant to the `Arjun` preprocessor and likely to influence the structure of the Sudoku SAT encodings considered in this thesis, and briefly introduces the `Arjun` preprocessor.

5.2.1 Classical Preprocessing

The earliest SAT preprocessing techniques aimed to simplify CNF formulas through local logical reasoning. These methods identify variables or clauses that can be removed or simplified without requiring expensive global analysis. Despite their simplicity, they remain fundamental components of modern SAT preprocessors (Biere et al., 2021).

Unit Propagation

Unit propagation was introduced in Section 3.2.1 in the context of SAT solving and model counting as an inprocessing technique. In preprocessing, the same mechanism is applied before search begins, starting from unit clauses that already exist in the input formula. These clauses force the corresponding variables to specific values, which may simplify other clauses, particularly binary clauses containing the corresponding variable with the opposite polarity.

Consider a Sudoku instance where cell $(1, 1)$ is given the digit 1. This clue introduces the unit clause (x_{111}) which forces x_{111} to be true. Consequently, binary clauses from the row uniqueness constraint containing the negated literal $\neg x_{111}$, such as

$$(\neg x_{111} \vee \neg x_{121}), (\neg x_{111} \vee \neg x_{131}), \cdots, (\neg x_{111} \vee \neg x_{191})$$

simplify to the unit clauses

$$(\neg x_{121}), (\neg x_{131}), \cdots, (\neg x_{191})$$

which allow additional unit propagation, resulting in a cascade of further simplifications.

Unit propagation preserves the set of satisfying assignments when the forced assignments are accounted for. However, in the context of model counting, care must be taken when removing clauses that become satisfied. For example, the nine-ary cell existence clause

$$(x_{111} \vee x_{121} \vee \cdots \vee x_{191})$$

is removed after being satisfied during conventional SAT preprocessing and inprocessing. While this operation preserves satisfiability, removing a satisfied clause in general does not necessarily preserve the model count because it may alter the number of valid assignments to the remaining variables. In the case of Sudoku encodings, the combination of exactly-one constraints may ensure that such simplifications remain count preserving, but this property depends on the specific encoding and requires justification.

Subsumption

Subsumption is a simple but effective preprocessing technique that removes logically redundant clauses from a CNF formula. A clause C_1 subsumes another clause C_2 or alternatively C_2 is subsumed by C_1, if every literal appearing in C_1 also appears in C_2. Formally, if

$$C_1 \subset C_2$$

then, C_2 can be removed because every assignment that satisfies C_1 also satisfies C_2. For example, for the clauses

$$C_1 = (a \vee b), \qquad C_2 = (a \vee b \vee c),$$

C_1 subsumes C_2, and C_2 can be removed without changing the satisfiability of the formula.

Subsumption reduces the number of clauses and may improve the efficiency of Boolean constraint propagation by eliminating redundant clauses. Consequently, it is widely implemented in SAT preprocessing systems. Subsumption preserves logical equivalence and thus the model count; provided that the clauses involved were not themselves produced by preprocessing transformations that do not preserve the model count.

Several variants of subsumption are used in modern SAT preprocessors. *Forward subsumption* checks whether a clause is subsumed by existing clauses and removes it when possible. Conversely, *backward subsumption* identifies clauses that are subsumed by the current clause and removes those redundant clauses. These variants are not discussed further in this thesis.

Self-subsuming Resolution Self-subsuming resolution extends subsumption by using resolution to strengthen existing clauses. Consider two clauses:

$$C_1 = (a \vee b \vee x) \qquad C_2 = (a \vee b \vee c \vee d \vee \neg x).$$

These can be seen as

$$C_1 = (A \vee x) \qquad C_2 = (B \vee \neg x) = (A \vee c \vee d \vee \neg x)$$

where $A = (a \vee b)$ subsumes $B = (a \vee b \vee c \vee d)$. This allows self-subsuming resolution to derive the single clause $(a \vee b \vee c \vee d)$ by resolving on x. Self-subsuming resolution can therefore reduce clause size while preserving logical equivalence and model count.

Other

Several other techniques such as *pure literal rule*, *failed literal rule* and *connected components* also fall into this category but are not discussed further in this thesis.

5.2.2 Resolution-Based Preprocessing

Many powerful preprocessing techniques simplify CNF formulas by exploiting the *resolution rule*. Resolution can derive new clauses from existing ones and can be used to eliminate variables, strengthen clauses, or expose further simplification opportunities. Unit propagation and subsumption, discussed in the previous subsection, can also be viewed as techniques related to resolution. This section introduces several additional resolution-based methods that are commonly used in modern SAT solvers and are relevant to the `Arjun` preprocessor.

Binary Implication Graphs (BIG)

A Binary Implication Graph (BIG) is a graph representation of all binary clauses in a CNF formula. Each literal is represented as a vertex, and every binary clause $(a \vee b)$ is interpreted as two implications $\neg a \rightarrow b$ and $a \rightarrow \neg b$.

Representing binary clauses as an implication graph enables efficient reachability analysis and supports several preprocessing techniques. For example, transitive implications and strongly connected components (SCCs) of the BIG, which represent equivalent literals, can be efficiently detected. Binary implication graphs therefore serve as the foundation for several advanced SAT preprocessing methods.

Since a BIG is only an alternative representation of the binary clauses already present in the formula, constructing the graph itself does not modify the formula and therefore preserves the model count. However, transformations based on the BIG do not necessarily preserve the model count depending on the specific operation performed.

Hyper Binary Resolution (HBR)

Hyper Binary Resolution (HBR) derives additional binary clauses that are logical consequences of a CNF formula by combining unit propagation with resolution. During propagation, HBR identifies situations where a set of clauses collectively implies a literal and derives binary implications that are already logically entailed by the formula. These implications are then added explicitly as binary clauses. Consider an n-ary clause $(l_1 \vee l_2 \vee \ldots \vee l_n)$ and $n-1$ binary clauses $(\neg l_1 \vee h)$, $(\neg l_2 \vee h)$, …, $(\neg l_{n-1} \vee h)$, HBR derives the hyper binary resolvent clause $(l_n \vee h)$ by applying a number of resolution steps incrementally (Biere et al., 2021).

The additional binary clauses strengthen the Binary Implication Graph and may enable further simplifications by subsequent preprocessing techniques, such as unit propagation. Since binary clauses can be propagated efficiently, introducing additional implied binary clauses can improve solver performance despite increasing the size of the formula. However, in the worst case, HBR may generate a quadratic number of non-transitive hyper binary resolvents (Biere et al., 2021), causing the overhead of handling additional clauses to outweigh the potential benefits.

Since HBR only adds clauses that are logical consequences of the original formula, the transformation itself preserves the model count. However, when HBR is combined with additional simplification techniques, count preservation depends on whether those subsequent transformations correctly preserve the number of satisfying assignments.

Clause Vivification

Clause vivification, also known as *clause distillation*, is a preprocessing technique that simplifies a CNF formula ϕ while preserving its logical equivalence. Its objective is to shorten clauses or remove clauses that are already implied by the remaining formula. To determine whether a clause can be simplified, vivification repeatedly applies Boolean Constraint Propagation (BCP) (Lagniez & Marquis, 2014).

Consider a clause

$$\alpha = \ell_1 \vee \ell_2 \vee \cdots \vee \ell_k.$$

Vivification examines the literals of α incrementally, constructing progressively smaller subclauses α' by assuming subset of the literals to be false and applying BCP to the formula $\phi \setminus \{\alpha\}$ under those assumptions. Two main simplification rules are applied (Lagniez & Marquis, 2014, 2017) :

- If BCP leads to conflict under the assumptions of subset of literals being false, then it proves that $\phi \setminus \{\alpha\} \models \alpha'$, and the original clause α is replaced by a smaller and stronger clause α'.
- If BCP proves that a remaining literal ℓ_e is implied by the current subclause α', meaning $\phi \setminus \{\alpha\} \models \alpha' \vee \neg\ell_e$, where ℓ_e is a literal appearing in α but not in α', then ℓ_e can be removed from α, producing a shorter clause.

Since vivification preserves logical equivalence over the original set of variables, it also preserves the model count. However, the effectiveness of vivification depends on the ordering of clauses and literals considered during the procedure, which can influence the resulting clause reductions and preprocessing efficiency (Lagniez & Marquis, 2014).

A related technique is *occurrence reduction*. Instead of removing entire clauses, occurrence reduction focuses on eliminating redundant literals from clauses. It can therefore be viewed as a lightweight form of vivification (Lagniez & Marquis, 2014). Similar to vivification, occurrence reduction preserves logical equivalence and the model count when applied without global variable elimination from the formula.

Other Resolution-Based Techniques

Modern SAT preprocessors implement many additional resolution-based simplification techniques. Examples include *Hidden Literal Elimination* (HLE), *Hidden Tautology Elimination* (HTE), and asymmetric literal elimination. These techniques can provide further reductions in formula size and improve propagation strength. However, their suitability to exact model counting depends on whether the resulting transformation preserves the number of satisfying assignments. These techniques are beyond the scope of this thesis and not discussed further.

5.2.3 CNF Preprocessing Beyond Resolution

Many SAT preprocessors extend beyond purely resolution-based techniques by exploiting more general notions of redundancy, where the primary requirement is preservation of satisfiability rather than logical equivalence (Biere et al., 2021). Examples include *Blocked Clause Elimination* (BCE), *Covered Clause Elimination* (CCE), and the more general *Asymmetric Covered Clause Elimination* (ACCE). Although these techniques can substantially simplify CNF formulas for SAT solving, their applicability to exact model counting depends on whether the transformation preserves the number of satisfying assignments. Since these methods are beyond the scope of this thesis, they are not discussed in further detail.

5.2.4 Structure-Based and Other Preprocessing

While the preprocessing techniques discussed so far operate directly on the CNF representation, another class of methods exploits more expressive representations and structural properties of Boolean formulas. These include techniques based on constraints such as XOR ($\oplus$) relations, as well as circuit-level representations involving gates and Boolean networks (Biere et al., 2021). When the input is provided as a CNF formula, algorithms such as *gate extraction* can be used to recover circuit-level structures, allowing circuit-based preprocessing techniques to be applied.

Another technique in this category that is relevant to this thesis is *Backbone Identification*. This technique is briefly discussed because it is used by the *Arjun2* preprocessor and is particularly relevant for preprocessing CNF formulas generated from Sudoku instances.

Other examples of structure-based techniques include *Parity Reasoning*, *Literal Equivalence Detection*, and *Gate Detection and Replacement*. A detailed discussion of these and other structure-based preprocessing techniques is beyond the scope of this thesis.

Backbone Identification

The *backbone* of a satisfiable CNF formula ϕ is the set of literals that are assigned the same truth value in every satisfying assignment of ϕ. If no such literals exist, the backbone is empty. The backbone of an unsatisfiable formula is also trivially empty. Backbone identification is a preprocessing technique that detects these literals and makes them explicit by adding the corresponding unit clauses to the formula.

For example, consider the formula

$$\phi = (a \vee b) \wedge (\neg a \vee b).$$

The variable b must be assigned `true` in every satisfying assignment, independent of the value assigned to a. Therefore, the backbone of ϕ contains the literal b, and the formula can be strengthened by adding the unit clause

$$(b).$$

In practice, backbone literals can be identified by testing whether assigning the opposite value of a candidate literal results in an unsatisfiable formula. If assuming $\neg\ell$ leads to a contradiction, then ℓ is implied by ϕ and is therefore part of the backbone.

Since backbone literals are already logical consequences of the original formula, adding the corresponding unit clauses preserves logical equivalence and consequently the model count. Making backbone literals explicit can also enable additional simplifications, such as stronger unit propagation, clause vivification, and variable elimination, thereby improving the effectiveness of subsequent preprocessing steps.

For Sudoku encodings, many backbone literals arise naturally from the given clues and the associated constraints of Sudoku rules. For example, the unit clauses derived through propagation from a clue, as described in Section 5.2.1, correspond to backbone assignments where these variables only take the value *true*. Additionally, other variables that denote other digits in the cell or the same digit in same row, column or blocks also become backbone literals that are fixed to *false* in all satisfying models. However, identifying all backbone literals of an arbitrary formula is computationally expensive; determining whether a literal belongs to the backbone is a co-NP-hard problem in general (Janota, Lynce, & Marques-Silva, 2015).

5.3 `Arjun` Preprocessor

5.3.1 Overview

`Arjun` is a ppreprocessor for computing independent supports, developed by Soos and Meel for applications such as model counting and sampling (Soos & Meel, 2022). Unlike traditional SAT preprocessors, which primarily focus on preserving satisfiability or logical equivalence, `Arjun` aims to reduce the number of variables that need to be explicitly considered during counting while preserving the relevant solution space.

The motivation behind `Arjun` is that many Boolean formulas contain variables whose values are not independent choices. Instead, some variables are functionally determined by other variables. Explicitly considering such variables during model counting introduces unnecessary complexity, as their values can be derived from the remaining variables.

By identifying a smaller independent support, `Arjun` reduces the effective number of variables over which counting is performed. This reduction can substantially improve the performance of exact model counters, particularly for formulas containing many functionally dependent variables.

`Arjun` was originally developed for *projected model counting* and sampling, where the objective is to count or sample assignments to a selected subset of variables. The technique has subsequently been integrated into practical model counting frameworks to improve scalability.

5.3.2 Independent Support Computation

An independent support provides a mechanism for identifying a subset of variables whose assignments completely determine the assignments of a larger set of variables. Let ϕ be a Boolean formula over variables X, and let $\mathcal{P} \subseteq X$ be a projection set. A subset $\mathcal{I} \subseteq \mathcal{P}$ is called an *independent support* of $\mathcal{P}$ if $\forall \sigma_1, \sigma_2 \in Sol(\phi)$,

$$\sigma_1|_{\mathcal{I}} = \sigma_2|_{\mathcal{I}} \Rightarrow \sigma_1|_{\mathcal{P}} = \sigma_2|_{\mathcal{P}}.$$

In other words, if two distinct satisfying assignments agree on all variables in $\mathcal{I}$, they must also agree on all variables in the projection set $\mathcal{P}$. Therefore, variables in $\mathcal{P} \setminus \mathcal{I}$ do not represent additional independent choices, as their values are uniquely determined by the variables in $\mathcal{I}$.

This property is particularly useful for projected model counting. Instead of counting assignments over the complete projection set, a model counter only needs to consider assignments to the independent support while preserving the number of distinct projected solutions. When $\mathcal{P} = X$, this concept can also be applied to ordinary exact model counting, since propositional model counting is a special case of projected model counting (Soos & Meel, 2022).

For example, consider the formula

$$\phi = (x \leftrightarrow y) \wedge (y \leftrightarrow z)$$

with the equivalent CNF representation

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

Every satisfying assignment of this formula is completely determined by the value assigned to x. If $x =$ `true`, then both y and z must also be assigned `true`; similarly, if $x =$ `false`, then both y

and z must be assigned `false`. Therefore, $\mathcal{I} = \{x\}$ is an independent support for the projection set (in this case $\mathcal{P} = X$)

$$\mathcal{P} = \{x, y, z\}.$$

Two satisfying assignments that agree on x must necessarily agree on all variables in $\mathcal{P}$. Consequently, instead of reasoning about assignments to all three variables, a model counter can perform counting over the single variable in the independent support.

5.3.3 Algorithmic Techniques and Contributions

The main contribution of `Arjun` is an efficient framework for computing independent supports of Boolean formulas. Unlike traditional preprocessing techniques that directly simplify the CNF formula, Arjun focuses on reducing the set of variables that need to be considered during counting while preserving the relevant projected solution space. The computed independent support can subsequently be used by model counters and samplers to reduce the complexity of their computations (Soos & Meel, 2022).

The computation of an independent support is based on the concept of definability. A variable can be removed from the support set if its value is uniquely determined by the remaining variables. `Arjun` exploits both *explicit definability* and *implicit definability*. These concepts are briefly illustrated below without discussing their full theoretical foundations.

Explicit definability identifies variables whose values can be directly inferred from recognizable structures in the formula, such as Boolean gates. For example, if a set of CNF clauses represents the equivalence relation

$$z \leftrightarrow (x \wedge y),$$

then z need not need to be included in the independent support because its value is completely determined by the values of x and y.

When such explicit structures are not present, `Arjun` uses implicit definability based on SAT solving queries. The approach determines whether a variable can take different values while all remaining variables are fixed. If no satisfying assignment exists in which the variable can change while the remaining variables remain fixed, then the variable is functionally determined and can be removed from the support set. This approach is based on *Padoa's theorem* and is implemented using assumption-based SAT solving with a CDCL solver (Soos & Meel, 2022). The detailed theoretical background is beyond the scope of this thesis.

`Arjun` combines these techniques in two main phases. The first phase applies gate identification to detect variables that are explicitly defined by the formula. The second phase uses assumption-based CDCL reasoning to identify additional implicit dependencies between variables. Since explicit and implicit definability are equivalent according to *Beth's theorem*, these phases can be applied sequentially, with the results of one phase reducing the search space for the other (Soos & Meel, 2022).

Several engineering improvements allow `Arjun` to scale to formulas from practical applications. These include applying SAT-based simplifications before independent support computation and optimizing assumption-based SAT queries. Such optimizations reduce the computational overhead of independent support computation while maintaining the correctness guarantees required for model counting (Soos & Meel, 2022).

The follow-up work `Arjun2`, extends the independent support computation approach into a more general preprocessing framework designed specifically for exact model counting. While `Arjun` primarily reduces the effective counting space by identifying independent supports, `Arjun2` combines this approach with additional preprocessing techniques, including redundancy detection, sampling-assisted backbone detection, and redundancy caching to improve propagation. The main motivation is that preprocessing techniques with substantial computational overhead for SAT solving may still be beneficial in model counting settings, where the cost of preprocessing can be amortized over the subsequent counting procedure and the resulting reduction in search complexity can significantly improve overall performance (Soos & Meel, 2024).

`Arjun2` incorporates several preprocessing techniques, including backward subsumption, bounded variable elimination, BCP-based clause vivification, backbone identification and additional simplification methods. Several of these techniques were introduced in previous sections. Since the experimental workflow throughout this thesis uses this version, `Arjun` refers to the latest version, `Arjun2`, except in this section and where otherwise noted.

5.3.4 Chapter Summary

This chapter discussed the role of CNF preprocessing in SAT solving and exact model counting. General SAT preprocessing techniques aim to simplify Boolean formulas by removing redundancies, reducing formula complexity, and improving the efficiency of subsequent solving procedures. Several relevant techniques, including unit propagation, subsumption, resolution-based simplification, and clause vivification, were introduced.

While many preprocessing techniques preserve only satisfiability and are therefore not directly applicable to exact model counting, other techniques preserve logical equivalence and consequently preserve the model count. Some techniques that do not directly preserve count can still be used as intermediate steps when combined with additional transformations that maintain counting correctness. Modern preprocessors typically combine multiple techniques to achieve effective formula simplification while maintaining the required correctness guarantees.

The `Arjun` preprocessor was presented as a specialized technique for computing independent supports of Boolean formulas. Unlike many SAT preprocessors, `Arjun` is designed for model counting and sampling applications where preserving the relevant solution structure is essential. It combines SAT-based reasoning with dependency analysis to identify variables that are sufficient to represent the solution space, reducing the effective number of variables that need to be considered during counting.

In this thesis, `Arjun`-based preprocessing is applied to CNF formulas generated from modified Sudoku puzzles. Both the original and preprocessed formulas are provided to the exact model counters to investigate how count-preserving preprocessing affects the performance of different model counting approaches and whether the structural reductions introduced by the preprocessor provide practical benefits for the considered CNF instances.

Chapter 6

Structural and other Properties of CNF Instances

6.1 Introduction and Motivation

The performance of SAT solvers and exact model counters is strongly influenced by the characteristics of the input CNF formulas. Although two formulas may represent the same underlying problem and have identical numbers of satisfying assignments, differences in their structural properties can significantly affect the computational effort required for solving and counting.

In this thesis, modified Sudoku puzzles are translated into CNF formulas using two different encoding schemes. These encodings preserve the same solution space but may produce formulas with different structural characteristics. Furthermore, the preprocessing techniques discussed in Chapter 5 can modify these characteristics while preserving the model count. Comparing formulas before and after preprocessing therefore provides insight into how structural changes influence model counting performance.

This chapter introduces the CNF properties considered in the experimental analysis. The evaluated characteristics include:

- formula size and density, including the number of variables and clauses, clause-to-variable ratio;
- treewidth of the primal graph representing variable interactions; and
- number of satisfying assignments.

The number of satisfying assignments is considered separately from the structural properties, as it describes the size of the solution space rather than the structure of the CNF representation. However, it is directly related to the objective of model counting and may influence the behavior of exact counting algorithms.

Understanding these characteristics helps interpret the experimental results, as different model counters exploit different structural properties through techniques such as search, knowledge compilation, caching, and dynamic programming, as discussed in Chapter 3. The following sections describe the properties considered in this thesis and their relevance to exact model counting.

6.2 Formula Size and Density

The size of a CNF formula is commonly characterized by the number of variables and clauses it contains. For a formula

$$\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_m$$

over a variable set

$$V = \{x_1, x_2, \ldots, x_n\},$$

the formula contains $n = |V|$ variables and $m = |C|$ clauses. These quantities provide a basic indication of the size of the search space and the amount of explicitly represented logical constraints. A formula with n variables has up to 2^n possible assignments before constraints imposed by clauses are considered.

For SAT solving and exact model counting, increasing the number of variables generally increases the potential search space, while increasing the number of clauses increases the amount of information that must be processed during propagation, search, or compilation. However, the relationship between formula size and computational difficulty is not monotonic. Additional clauses may strengthen propagation and allow invalid assignments to be eliminated earlier, while a smaller formula may require more extensive search due to weaker constraints.

The numbers of variables and clauses are sometimes considered together with the clause-to-variable ratio:

$$R = \frac{|C|}{|V|} = \frac{m}{n}.$$

This ratio provides a normalized measure of formula density and allows comparison between formulas of different sizes. A higher ratio indicates that variables participate in more constraints on average, while a lower ratio corresponds to a sparser constraint structure. The clause-to-variable ratio has been extensively studied in the context of SAT decision problems. For random k-SAT formulas, increasing clause density leads to a phase transition behavior, where instances near a critical ratio can become significantly harder for SAT solvers to classify as satisfiable or unsatisfiable. This behavior has been observed for random 3-SAT and extended to larger fixed clause sizes (Mitchell, Selman, & Levesque, 1992; Mertens, Mézard, & Zecchina, 2006). However, these observations do not directly transfer to structured formulas or exact model counting. In particular, the relationship between clause density and #SAT performance is more complex, as counting algorithms are affected not only by constraint density but also by factors such as decomposition opportunities, variable interactions, and the number of satisfying assignments.

These measures are particularly relevant when comparing the Sudoku encodings considered in this thesis. Before preprocessing, both minimal and extended encodings contain the same number of variables but differ in the number of clauses. Consequently, the extended encoding has a higher clause-to-variable ratio, which may influence propagation and subsequent model counting performance. After preprocessing, both the number of variables and clauses can change depending on the applied transformations and the structure of individual instances.

In this thesis, formula size and density measures are evaluated primarily on preprocessed CNF formulas, as the raw Sudoku encodings have fixed variable counts and clause-to-variable ratios within each encoding scheme. These measurements are used to characterize how encoding choices and count-preserving preprocessing modify the resulting CNF representations and how these changes relate to model counter performance.

6.2.1 Complexity and Current Upper Bounds

The computational complexity of exact model counting is influenced by both the size and structure of the input CNF formula. While formula size measures such as the number of variables, clauses, and clause-to-variable ratio provide useful indicators of instance characteristics, they do not fully determine counting difficulty. The underlying complexity of exact #SAT remains exponential in the general case, and practical algorithms rely on exploiting structural properties of the input formula.

The worst-case time complexity of exact #SAT algorithms is typically expressed in terms of the number of variables (n) or clauses (m). For unrestricted #SAT, exhaustive search yields the trivial $\mathcal{O}^*(2^n)$ upper bound, while more refined branching algorithms achieve improved bounds for restricted formula classes. For example, #2-SAT and #3-SAT admit improved exponential-time algorithms by exploiting their limited clause sizes. Table 6.1 summarizes representative worst-case upper bounds reported in the literature (Zhou, Yin, & Zhou, 2010; Wahlström, 2008; Kutzkov, 2007).

Table 6.1: Representative worst-case upper bounds for exact #SAT algorithms.

Problem	By variables (n)	By clauses (m)
General #SAT	$\mathcal{O}^*(2^n)$	–
#2-SAT	$\mathcal{O}^*(1.2377^n)$	$\mathcal{O}^*(1.1892^m)$
#3-SAT	$\mathcal{O}^*(1.6423^n)$	$\mathcal{O}^*(1.4142^m)$

The $\mathcal{O}^*(\cdot)$ notation suppresses polynomial factors.

6.3 Graph Representation and Treewidth

The size-based parameters introduced in the previous section capture only limited characteristics of a CNF formula. However, they do not capture how variables interact through these constraints. Two formulas with identical numbers of variables and clauses can therefore exhibit substantially different structural complexity.

Graph representations of CNF formulas provide a way to capture these interactions. A central structural parameter used in this context is treewidth. Originally introduced in graph theory by Robertson and Seymour (Robertson & Seymour, 1986), treewidth has since been applied to constraint satisfaction, SAT, and #SAT as a measure of how closely a graph resembles a tree. Many algorithms can exploit bounded treewidth to perform efficient dynamic programming over the resulting tree decomposition (Samer & Szeider, 2010b, 2010a; Ganian & Szeider, 2021).

The intuition behind treewidth is that trees have a simple hierarchical structure that allows efficient processing of local subproblems. Graphs with small treewidth can similarly be decomposed into tree-like structures, enabling algorithms to reason over smaller groups of variables. In contrast, graphs with large treewidth contain highly interconnected regions, making decomposition and intermediate representations more complex.

6.3.1 Graph Representations of CNF Formulas

Several graph representations can be derived from a CNF formula, depending on the structural properties of interest. The two most common representations are the *primal graph* and the *incidence graph*. This thesis focuses on the primal graph because it directly captures interactions between variables that occur together in clauses.

Primal Graph

Given a CNF formula ϕ, the primal graph is defined as:

$$G_P = (V, E),$$

where each vertex corresponds to a Boolean variable. An edge is added between two variables if they occur together in at least one clause. For example, the clause

$$(x_1 \vee x_2 \vee \neg x_3)$$

creates the edges

$$(x_1, x_2), \quad (x_1, x_3), \quad (x_2, x_3).$$

Duplicate edges are represented only once. Figure 6.1 illustrates the primal graph corresponding to the CNF formula in Eq. 2.3.

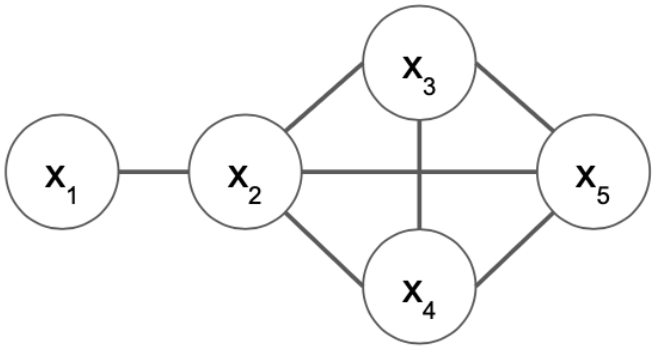


Figure 6.1: Primal graph for the CNF presented in Eq. 2.3

The primal graph is particularly relevant for exact model counting because many counting algorithms exploit decompositions of the variable interaction structure. When the graph separates into weakly connected or independent regions, these components can often be processed separately and combined during counting. Consequently, graph properties such as treewidth provide additional information beyond formula size metrics when analyzing model counting performance.

Other Representations

Another commonly used representation is the incidence graph. Unlike the primal graph, which contains only variable vertices, the incidence graph contains two types of vertices: variable and

clause. An edge connects a variable vertex to a clause vertex whenever the corresponding variable appears in that clause, resulting in a bipartite graph.

Other representations, such as the *dual graph*, *conflict graph*, and *consensus graph*, capture different structural properties of CNF formulas and can be useful for specific algorithms. However, from a parameterized complexity perspective, these graph representations are closely related, and treewidth bounds for one representation can often be related to those of another through polynomial transformations (Samer & Szeider, 2010a, 2010b).

6.3.2 Tree Decomposition and Treewidth

A *tree decomposition* of a graph $G = (V, E)$ is a pair (T, χ), where $T = (B, P)$ is a tree and χ is a mapping that assigns each node $b_i \in B(T)$, called a *bag*, a subset of vertices from G. The notation B and P is used here to distinguish the vertices and edges of the decomposition tree from the vertices and edges of the original graph. A valid tree decomposition must satisfy three properties (Samer & Szeider, 2010a):

- **Vertex coverage:** Each $v \in V(G)$ appears in at least one bag, i.e., $\bigcup_{b_i \in B(T)} \chi(b_i) = V$.
- **Edge coverage:** For every edge $(u, v) \in E(G)$, there exists a bag in T that contains both endpoints of the edge, i.e., there exists $b_i \in B(T)$ such that $\{u, v\} \subseteq \chi(b_i)$.
- **Connectedness:** If two bags $b_i, b_j \in B(T)$ contain the same vertex $v \in V$, then all bags b_k on a unique path between b_i and b_j also contain v.

Intuitively, a tree decomposition represents a graph as overlapping groups of vertices arranged in a tree structure. This allows algorithms to process local regions while preserving the dependencies between them. Figure 6.2 illustrates example tree decompositions.

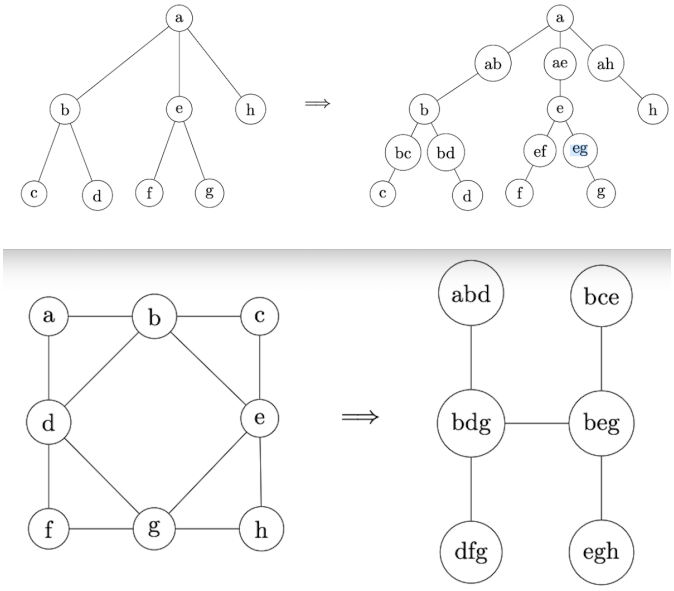


Figure 6.2: Example tree decompositions (Li, 2019).

The *width* of a tree decomposition is defined as the size of its largest bag minus one. The *treewidth* of a graph is the minimum possible width among all its tree decompositions. Low treewidth indicates that the graph admits a tree decomposition with small bags, while high treewidth generally requires larger bags. For example, isolated vertices have treewidth zero, trees have treewidth one, a simple cycle has treewidth two and dense or highly interconnected regions generally require larger bags.

For practical algorithms, tree decompositions are often converted into a normalized form called a *nice tree decomposition*. A *nice* tree decomposition is a rooted tree in which every node represents one of four operations: *leaf*, *introduce*, *forget*, or *join*. This representation simplifies dynamic programming algorithms while preserving the treewidth of the original decomposition. Each node performs exactly one operation on its bag, resulting in simpler and more efficient dynamic programming algorithms. Figure 6.3 shows an example of a NICE tree decomposition.

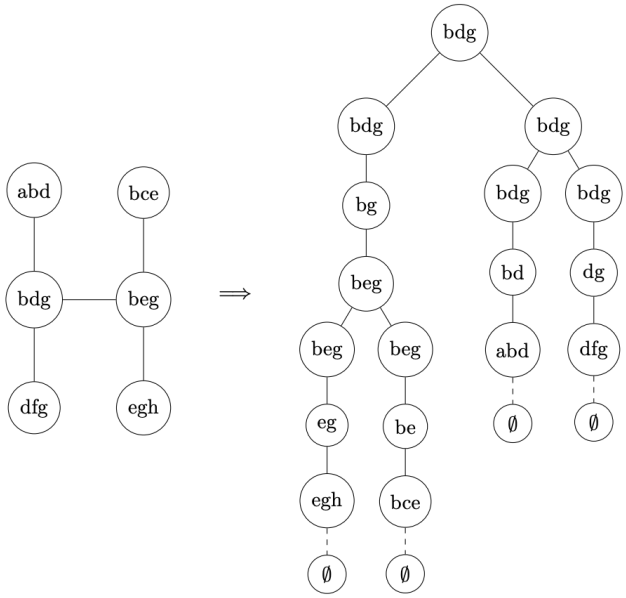


Figure 6.3: Example of a NICE tree decomposition (Li, 2019).

Multiple tree decompositions may exist for the same graph, and finding a decomposition with minimum possible width is NP-hard (Arnborg et al., 1987; Fichte et al., 2017). Therefore, practical applications typically rely on heuristic algorithms that compute decompositions with sufficiently small width.

6.3.3 Primal Treewidth and Its Relevance to #SAT

Throughout this thesis, treewidth refers to the treewidth of the *primal graph* of the CNF formula. Therefore, primal treewidth captures the complexity of the variable interaction structure induced by the clauses rather than only the size of the formula.

The relevance of primal treewidth for exact model counting follows from parameterized complexity results. Although exact model counting is #P-complete in general, it becomes fixed-

parameter tractable when parameterized by the treewidth tw of the primal graph. Dynamic programming algorithms over tree decompositions can solve #SAT in time

$$\mathcal{O}\big(2^{tw} \cdot \mathrm{poly}(n)\big)\,,$$

where the exponential component depends only on the treewidth and the remaining dependence on the formula size is polynomial (Samer & Szeider, 2010b, 2010a; Ganian & Szeider, 2021). Consequently, formulas with small primal treewidth can be solved efficiently even when they contain many variables or clauses.

Primal treewidth is also relevant for knowledge compilation approaches, as representations such as decision diagrams and decomposable circuits often remain compact for formulas with bounded treewidth (Amarilli, Capelli, Monet, & Senellart, 2020). However, even for counters that do not explicitly use tree decompositions, treewidth provides a useful indicator of structural complexity and can help explain performance differences between counting approaches.

In this thesis, primal treewidth values are computed using the `treedecomp` tool developed by the Meel group. The tool applies modern graph decomposition techniques, including FlowCutter-based methods, to compute tree decompositions of the generated primal graphs (Meelgroup, n.d.). The implementation details of `treedecomp` are beyond the scope of this thesis.

6.4 Solution Space Characteristics

Unlike the structural properties discussed in previous sections, the number of satisfying assignments is not a property of the CNF representation, but rather a characteristic of the represented solution space. Nevertheless, it is an important quantity for evaluating exact model counters because it directly represents the size of the solution space that must be counted.

The number of satisfying assignments alone does not determine the difficulty of exact model counting. A formula with many solutions may be efficiently counted if it contains independent components or repeated substructures that can be exploited through decomposition and caching. Conversely, a formula with few solutions may remain difficult if the solver must explore many intermediate subproblems to prove that no additional solutions exist.

Therefore, in this thesis the model count is considered as an additional instance characteristic rather than a direct predictor of computational complexity. Its relationship with preprocessing effectiveness and model counting performance is evaluated empirically in Chapter 8.

6.5 Impact of Encoding and Preprocessing on CNF Structural Properties

The structural properties of the generated CNF formulas depend not only on the underlying Sudoku instances but also on the encoding and preprocessing techniques applied. In this thesis, the main factors affecting the resulting formula structure are:

- the choice between minimal and extended Sudoku encodings,
- the application of count-preserving preprocessing using `Arjun`.

Although these transformations preserve the solution space and therefore model count, they can produce CNF formulas with substantially different structural characteristics. The following subsections briefly describe the expected impact of these transformations on the properties evaluated in this thesis.

6.5.1 Formula Size and Density

The two Sudoku encodings differ primarily in the number of clauses. As shown in Table 4.2, both the minimal and extended encodings contain the same number of variables ($n = 729$) before preprocessing, while the extended encoding introduces additional redundant clauses. Consequently, the extended encoding has a higher clause-to-variable ratio.

6.5.2 Primal Treewidth

The Sudoku encodings considered in this thesis produce highly connected primal graphs due to the interaction between cell, row, column, and block constraints. Variables representing possible digit assignments are connected whenever they participate in a common clause, resulting in a dense interaction structure.

For example, the variable x_{111}, representing whether the first cell contains digit 1, is connected to variables representing alternative digits for the same cell through cell constraints. It is also connected to variables representing digit 1 in other cells of the same row, column, and block through uniqueness constraints. Since each of these variables participates in similar constraints themselves, the resulting primal graph contains many overlapping connections. The same logic applies for extended encoding, which introduces additional redundant constraints. However, these additional constraints may only introduce edges that already exist in the primal graph, in which case they do not increase the treewidth.

This dense structure leads to relatively large treewidth values for the raw Sudoku encodings. Using the `treedecomp` tool, the primal treewidth values of the raw minimal and extended encoding were empirically measured to be approximately 410.[1] Such a large value indicates that the formula does not admit a small tree-like decomposition. Since the underlying Sudoku rules are identical for all instances, the treewidth of the raw encodings is largely independent of the particular clues.

After preprocessing, however, both the formula size and interaction structure may change significantly. Unit propagation and other count-preserving simplifications can remove variables and clauses, thereby modifying the primal graph and potentially reducing treewidth. The resulting treewidth therefore depends on both the encoding choice and the specific Sudoku instance. In this thesis, these changes are measured and analyzed together with model counting performance.

6.6 Chapter Summary

This chapter introduced the CNF properties used to characterize and analyze the benchmark formulas evaluated in this thesis. These properties provide context for interpreting performance differences between exact model counters, which may exploit different aspects of formula structure.

[1] The reported value is obtained using heuristic tree decomposition algorithms and therefore represents an upper bound on the optimal treewidth.

The chapter discussed basic formula-size characteristics, including the number of variables, number of clauses, and clause-to-variable ratio; however, they do not fully capture structural characteristics of a CNF formula. The primal graph and its associated treewidth were introduced as measures of structural complexity. Treewidth is particularly relevant for algorithms based on decomposition, dynamic programming, and knowledge compilation.

In addition to structural properties, the number of satisfying assignments was considered as a solution-space characteristic. Although model count is not a structural property of the CNF representation, it directly relates to the objective of exact model counting and is therefore included in the analysis.

Finally, the impact of Sudoku encoding choices and count-preserving preprocessing on CNF characteristics were discussed. Different encodings and preprocessing configurations preserve the same solution space while producing formulas with different structural properties. These differences form the basis for empirical analysis of how CNF representations and preprocessing relate to model counting performance.

Chapter 7

Experimental Methodology

7.1 Experimental Workflow

This chapter describes the methodology used to generate the benchmark instances, execute the model counting experiments, and collect data. Figure 7.1 illustrates the overall experimental workflow.

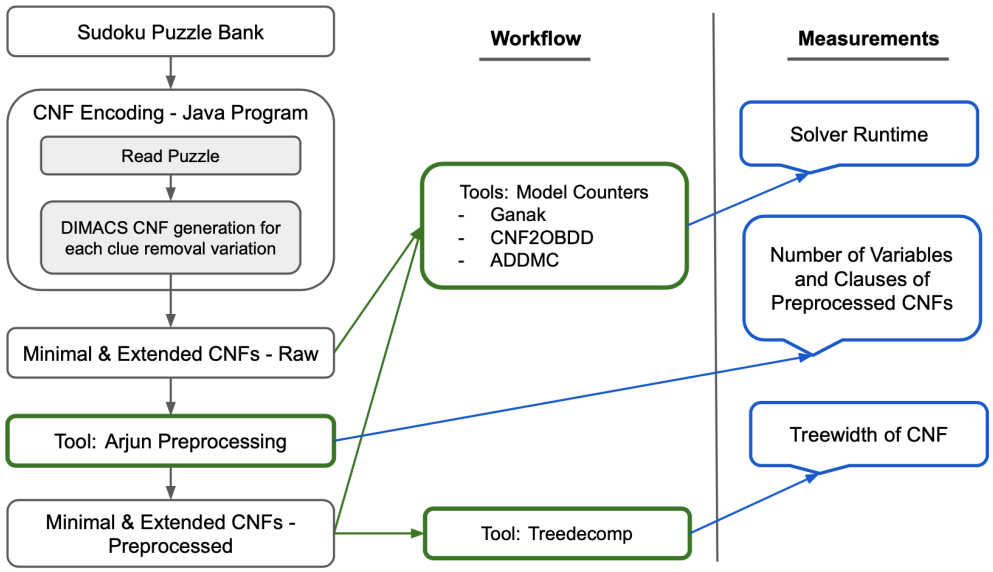


Figure 7.1: Benchmark generation and evaluation workflow

The workflow begins with valid Sudoku puzzles admitting a unique solution. Each puzzle is translated into CNF using both the minimal and extended encodings described in Chapter 4. Additional benchmark instances are generated by removing selected clues before encoding.

The resulting CNF formulas are evaluated both before and after preprocessing. Count-preserving preprocessing is performed using `Arjun`, while structural properties of the preprocessed formulas are obtained using the `treedecomp` tool. Finally, the generated CNF formulas are evaluated using

the selected exact model counters, and the recorded measurements are analyzed to investigate the effects of encoding, preprocessing, and structural properties on model counting performance.

7.2 Benchmark Generation

Sudoku Puzzle Dataset The benchmark instances used in this thesis are based on the Sudoku Exchange Puzzle Bank developed by Grant McLean (McLean, 2021). The dataset contains a large collection of valid Sudoku puzzles covering several difficulty levels and is publicly available on GitHub. To obtain a representative benchmark while keeping the experimental evaluation computationally manageable, two puzzles were selected from each of the four difficulty categories: easy, medium, hard, and diabolical.

Puzzle Variants The original Sudoku puzzles are not modified directly. Instead, additional benchmark instances are generated as part of the CNF generation process by selectively removing clues from each puzzle. Removing clues generally increases the number of satisfying assignments while preserving the Sudoku constraints, producing benchmark instances with different model counts while retaining the same underlying encoding framework. The resulting Sudoku benchmark consists of approximately 1000 modified puzzle instances for each original puzzle, i.e., approximately 8000 puzzle instances total. The generated variants are summarized in Table 7.1.

Table 7.1: Generated puzzle variants for each original Sudoku puzzle.

Variant	**Clues Removed**	**Instances per Puzzle**
Original	0	1
Single removal	1	23–28
Pair removal	2	≈350
Triple removal	3	≈300 ($\frac{1}{10}$ random sample)
Quadruple removal	4	≈300 ($\frac{1}{60}$ random sample)

CNF Generation

The selected Sudoku instances are translated into CNF using a Java program developed as part of this thesis. The implementation constructs the Boolean variables and clauses described in Chapter 4 and outputs the resulting formulas in DIMACS CNF format, the standard input format used by SAT solvers, preprocessors, and model counters (*Satisfiability Suggested Format*, 1993; Froleyks, Heule, Iser, Järvisalo, & Suda, 2021). The implementation internally maps the logical Sudoku variables introduced in Chapter 4 to the consecutive integer identifiers required by the DIMACS format. Details of the DIMACS format and the mapping are provided in Appendix A.2.

For each puzzle variant, both the minimal and extended encodings are generated, producing two DIMACS CNF files. These files are subsequently used as input to the Arjun preprocessor, producing two more DIMACS CNF files for each modified puzzle variant. This results in approximately 32000 CNF instances in total for both encoding schemes in both raw and preprocessed form.

7.3 Count-Preserving Preprocessing

Each generated CNF formula is processed using the `Arjun` preprocessor described in Chapter 5. `Arjun` simplifies the input formula while preserving its model count.

For each CNF instance, `Arjun (v2.7.0)` produces a simplified DIMACS CNF file together with summary information, including the resulting number of variables and clauses. These measurements are extracted automatically and stored for subsequent analysis. Both the original and preprocessed CNF formulas are retained for the model counting experiments.

7.4 Treewidth Computation

Following preprocessing, the primal treewidth of each reduced CNF formula is computed using the `treedecomp` tool. The tool constructs a heuristic tree decomposition of the primal graph using FlowCutter-based algorithms and reports several graph characteristics, including primal treewidth and graph density.

Treewidth computation is performed independently of the model counting experiments and therefore does not influence solver execution. The reported structural properties are recorded for later analysis of their relationship with model counting performance.

7.5 Model Counter Evaluation

The model count for each CNF instance is computed using the three exact model counters introduced in Chapter 3: `Ganak (v2.6.0)`, `CNF2OBDD (v1.0.2)`, and `ADDMC (mc-2020)`. Each model counter is executed on both the original CNF formulas and their corresponding preprocessed versions, producing four experimental configurations for each modified puzzle:

- raw minimal encoding,
- preprocessed minimal encoding,
- raw extended encoding,
- preprocessed extended encoding.

For every execution, the model count, CPU runtime, and execution status are recorded. A timeout limit of five minutes is imposed on each run to ensure that excessively difficult instances do not dominate the experimental evaluation. Instances exceeding this limit are recorded as timeouts.

In addition to successful executions and timeouts, execution failures are also recorded. These include, for example, instances where a model counter terminates because of memory limitations or unsupported input. Recording these outcomes allows differences between the evaluated counters to be analysed beyond runtime alone.

7.6 Experimental Environment

All experiments are executed inside a Docker container to provide a consistent and reproducible software environment across all benchmark instances and model counters. The container includes

the required versions of the evaluated model counters, preprocessing tools, and supporting software used throughout the experimental pipeline.

The benchmark generation, execution of experiments, and data analysis are partly automated using Java, Shell, and Python scripts respectively. The experiments are performed on a 2019 MacBook Air using a Docker container. The complete machine and Docker configuration details can be found on Github repository.

7.7 Recorded Measurements

Several measurements are recorded throughout the experimental pipeline. The numbers of variables and clauses of each preprocessed CNF formula are recorded as described in Section 7.3. The primal treewidth reported by `treedecomp` is recorded as described in 7.4. The model count, CPU runtime, timeouts and failures are recorded as described in 7.5. All measurements are written to CSV files and subsequently imported into Python for statistical analysis and visualization.

Excluded Instances

During preprocessing, `Arjun` occasionally simplifies a CNF formula significantly, producing an empty or very small formula together with a comment of the form `c MUST MULTIPLY BY X`. This indicates that the model count of the original formula is equal to that of the preprocessed formula multiplied by the specified constant.

While `Ganak` correctly supports this comment with empty or very small formula, the evaluated versions of `CNF2OBDD` and `ADDMC` do not. Consequently, these instances cannot be processed consistently across all three model counters.

For analyses comparing the runtime of all evaluated counters, such instances are excluded. Likewise, analyses that directly compare raw and preprocessed formulas omit these instances because the preprocessed CNF no longer represents the complete equivalent counting problem when the multiplier is taken into account.

7.8 Chapter Summary

This chapter described the experimental methodology used throughout the thesis. It presented the benchmark generation process, CNF encoding, count-preserving preprocessing, treewidth computation, evaluation of the selected exact model counters, and the experimental environment.

The next chapter presents the experimental results and analyzes the influence of encoding strategy, preprocessing, and structural properties on the performance of the evaluated exact model counters.

Chapter 8

Results and Discussion

8.1 Overall Performance

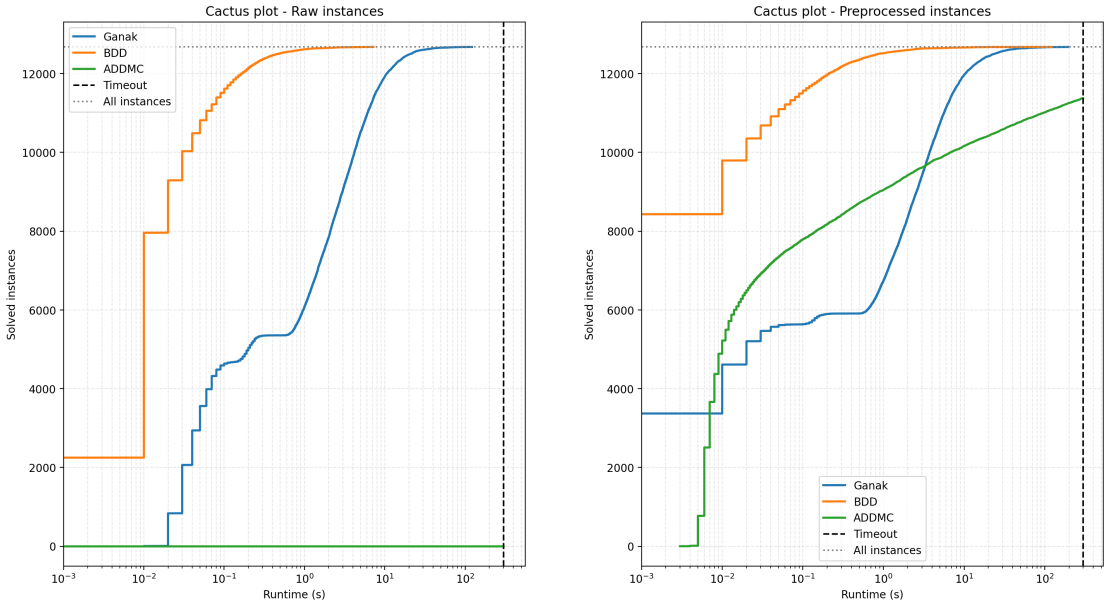
Table 8.1 summarizes the percentage of instances solved within the 5-minute timeout for each of the four CNF variants. After removing unsuitable instances and cleaning the experimental data, 6337 puzzle instances remained, each represented by four CNF variants.

Table 8.1: Percentage of CNF instances solved within the 5-minute timeout by each solver.

	Raw		**Preprocessed**	
Model Counter	**Minimal**	**Extended**	**Minimal**	**Extended**
`Ganak`	100%	100%	100%	100%
`CNF2OBDD`	100%	100%	100%	100%
`ADDMC`	0%	0%	85.45%	94.05%

`Ganak` and `CNF2OBDD` solve all instances across both raw and preprocessed CNFs. In contrast, `ADDMC` does not solve any raw instances within the timeout, but solves 85.45% of the preprocessed minimal instances and 94.05% of the preprocessed extended instances. The substantial improvement for `ADDMC` following preprocessing is consistent with the importance of treewidth for its underlying decomposition-based approach. Since `ADDMC` does not perform the preprocessing or inprocessing unlike `Ganak` and `CNF2OBDD`, the structural simplifications introduced by the preprocessing stage can substantially improve the efficiency of the subsequent decomposition and dynamic programming. The effects of preprocessing on formula size reduction and structural complexity are analyzed further in Section 8.2.

Figure 8.1 shows transposed cactus plots comparing the performance of the three model counters on the raw and preprocessed instances. This plot shows, for all model counters, cumulative number of instances solved on y-axis within given time with a maximum at selected timeout value on x-axis. As shown in Figure 8.1, `CNF2OBDD` shows the best overall performance for both raw and preprocessed instances. Although `ADDMC` solves no raw instances within the timeout, it initially solves preprocessed instances faster than `Ganak`, but `Ganak` eventually overtakes it as the number of solved instances increases. This suggests that preprocessing is particularly beneficial for `ADDMC` on instances where the resulting treewidth is sufficiently low. Solver-specific runtime differences between raw and preprocessed instances are analyzed further in section 8.3.



(a) Raw (b) Preprocessed

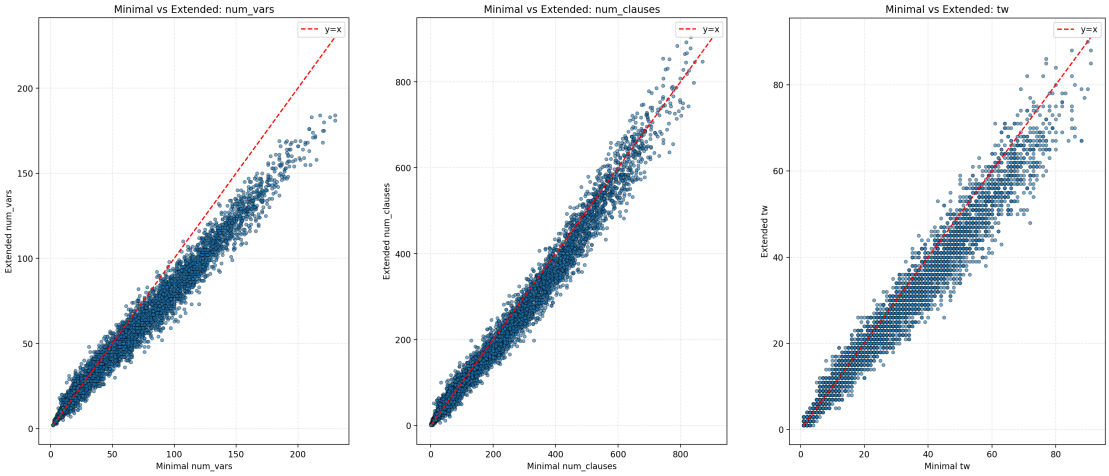
Figure 8.1: Cactus plots comparing solver performance on (a) raw and (b) preprocessed instances.

The initial advantage of `ADDMC` over `Ganak` may also be related to the preprocessing performed internally by Ganak. Ganak invokes the `Arjun` preprocessor even when the input has already been externally preprocessed, introducing additional preprocessing overhead. For instances with sufficiently low treewidth, `ADDMC` can therefore benefit from the absence of an additional preprocessing stage and solve some instances faster than `Ganak`.

8.2 Effects of Encoding on Preprocessing and Runtime

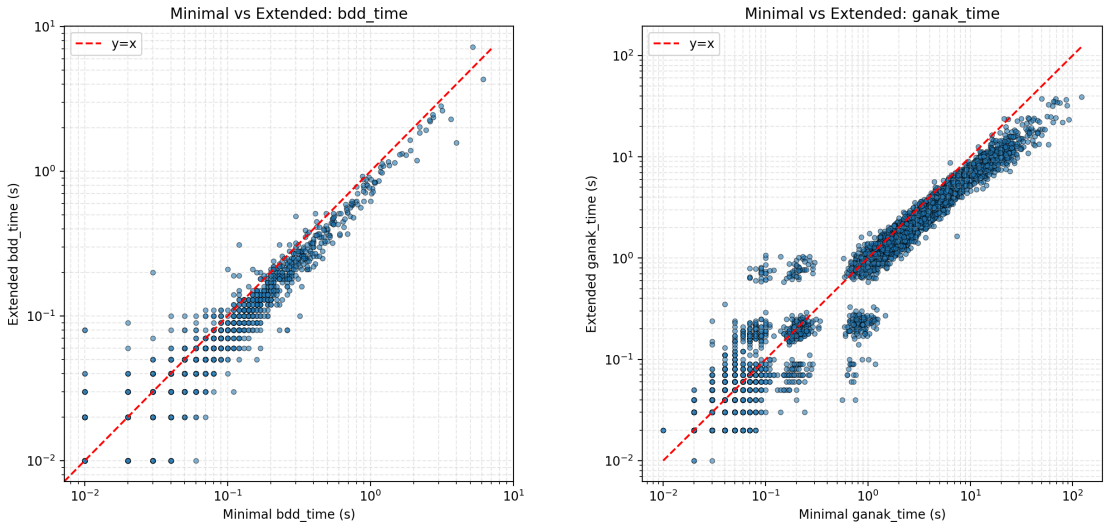
Figure 8.2 compares the number of variables, number of clauses, and treewidth of the minimal and extended encodings after preprocessing. The diagonal in each plot represents equal values for the two encodings. Points below the diagonal indicate that the extended encoding has a lower value for the corresponding parameter, while points above the diagonal indicate the opposite. The extended encoding results in fewer variables and lower treewidth for the majority of instances. However, the presence of points on both sides of the diagonal indicates that the effect of the encoding is not uniform across all instances.

Figure 8.3 compares the runtime of the minimal and extended encodings on the raw CNFs for `CNF2OBDD` and `Ganak`. `ADDMC` is excluded because it times out on all raw CNF instances. As in Figure 8.2, the diagonal represents equal runtimes for the two encodings. Points below the diagonal indicate that the extended encoding is faster. Although points appear on both sides of the diagonal, the majority lie below it for both `CNF2OBDD` and `Ganak`, indicating a tendency for the extended encoding to result in slightly lower runtimes. Thus, while the extended encoding does not consistently outperform the minimal encoding, it shows a modest runtime advantage for most instances.



(a) Number of variables (b) Number of clauses (c) Treewidth

Figure 8.2: Comparison of instance characteristics between minimal and extended encodings after preprocessing.

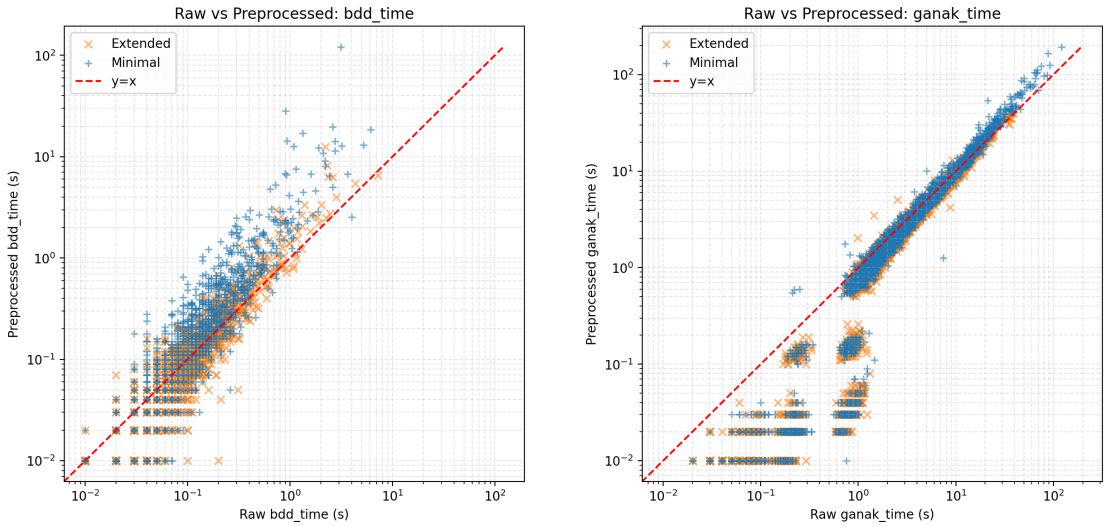


(a) `BDD` runtime (b) `Ganak` runtime

Figure 8.3: Comparison of minimal and extended encoding runtimes before preprocessing. Points below the diagonal indicate that the extended encoding is faster. `ADDMC` is excluded because all raw CNF instances exceed the timeout.

8.3 Effects of Preprocessing

Figure 8.4 compares the runtimes of model counters on raw and preprocessed CNFs for `CNF2OBDD` and `Ganak`. `ADDMC` is excluded because it times out on all raw instances. The results therefore provide a direct comparison of the effect of preprocessing for `CNF2OBDD` and `Ganak`, while the results for `ADDMC` suggest the importance of preprocessing for its performance.



(a) `BDD` runtime

(b) `Ganak` runtime

Figure 8.4: Comparison of runtimes on raw and preprocessed instances. Points below the diagonal indicate a lower runtime for the preprocessed instance. `ADDMC` is excluded because it times out on all raw instances.

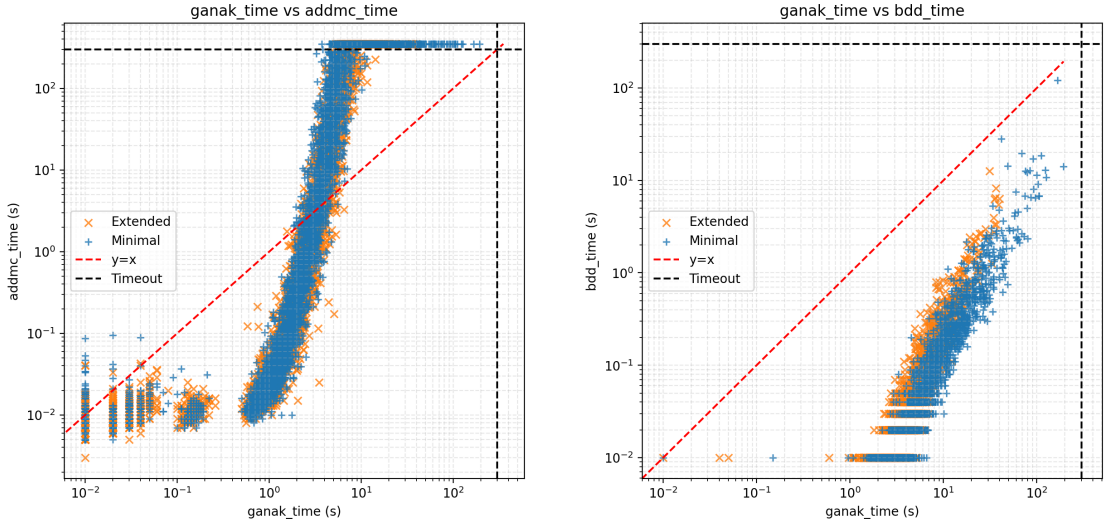
For `Ganak`, preprocessed instances generally have lower runtimes for instances that can be solved in less than approximately 1 second. For runtimes above this range, however, the relationship becomes less consistent, with raw instances showing a slight runtime advantage for instances with longer runtimes. One possible explanation is that `Ganak` applies the `Arjun` preprocessor to every input. Consequently, the preprocessed instances may incur additional preprocessing overhead without necessarily providing sufficient structural simplification to offset this cost.

For `CNF2OBDD`, raw instances have lower runtimes for most data points. One possible explanation is that preprocessing may remove clauses and simplify relationships between variables that are useful to BDD construction. For example, preprocessing may replace groups of clauses with a smaller representation, potentially removing redundancies that could otherwise be exploited during BDD construction. Thus, although preprocessing reduces the size or treewidth of the formula, it does not necessarily produce a representation that is more favorable for `CNF2OBDD`.

It is important to note, however, that the experimental evaluation did not measure or account for time taken by the preprocessor. For some instances, preprocessing time could be significant. However, preprocessing time was excluded from consideration as this is one-time effort that can be amortized over subsequent model-counting evaluations.

8.4 Solver Comparison

Table 8.1 and Figure 8.1 provide an overall comparison of model-counter performance. To examine the differences between individual solvers in more detail, Figure 8.5 compares their runtimes on an instance-by-instance basis for the preprocessed CNFs. As shown in Figure 8.5, `CNF2OBDD`



(a) `Ganak` vs. `ADDMC` runtime

(b) `Ganak` vs. `CNF2OBDD` runtime

Figure 8.5: Pairwise comparison of solver runtimes on preprocessed instances. Points below the diagonal indicate a lower runtime for the solver represented on the y-axis. The `ADDMC`–`CNF2OBDD` comparison is omitted because of the large performance difference between the two solvers.

clearly outperforms `Ganak` on the evaluated preprocessed instances. The same pattern is observed for the raw instances, although those results are omitted for brevity. For the preprocessed instances, `ADDMC` initially outperforms `Ganak` on instances with lower runtimes, but this relationship reverses at approximately 10 seconds, after which `Ganak` is generally faster.

8.5 Effect of CNF Properties

8.5.1 Formula Size

Figures 8.6, 8.7, and 8.8 show the relationship between model-counter runtime and three formula-size parameters: number of variables, number of clauses, and the clause-to-variable ratio. Runtime is shown on a logarithmic scale. The number of variables and clauses show a clear positive relationship with runtime for all model counters, albeit strongest for `ADDMC`.

For `Ganak`, the relationship between runtime and the number of variables or clauses is positive but relatively moderate, while the effect of the clause-to-variable ratio is less pronounced. `CNF2OBDD` shows a similar overall trend, although its dependence on formula size is weaker than that of `ADDMC`. `ADDMC` exhibits the strongest increase in runtime with both the number of variables and the number of clauses. For the clause-to-variable ratio, the relationship is less pronounced overall, although runtimes increase more noticeably for ratios above approximately 3.5.

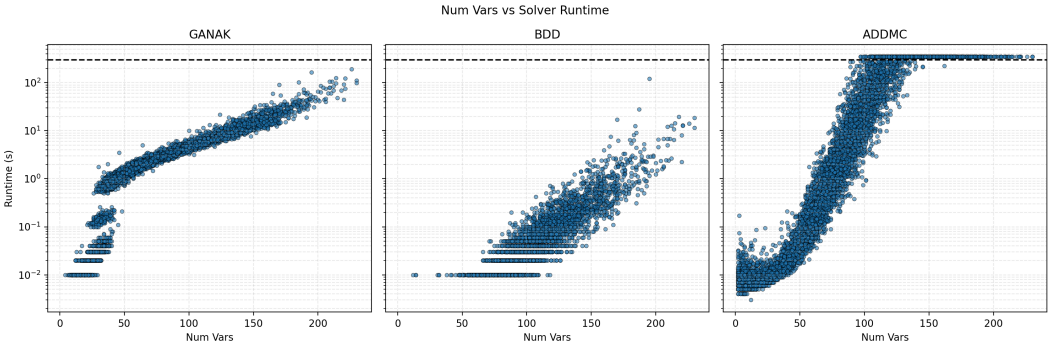


Figure 8.6: Number of variables versus runtime for preprocessed CNFs.

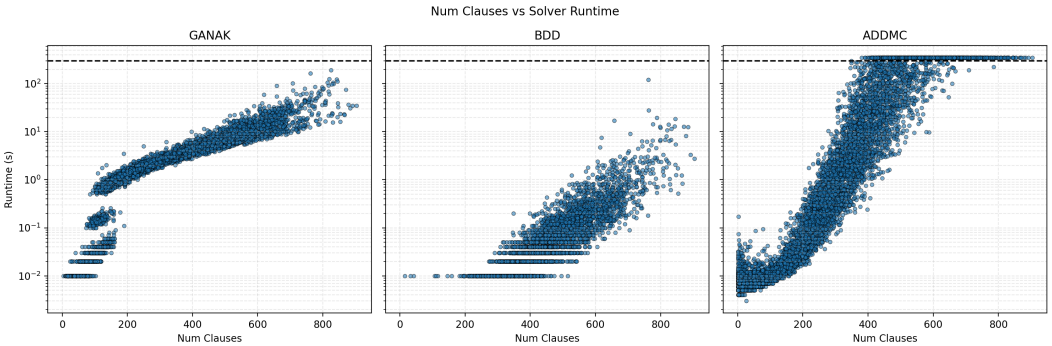


Figure 8.7: Number of clauses versus runtime for preprocessed CNFs.

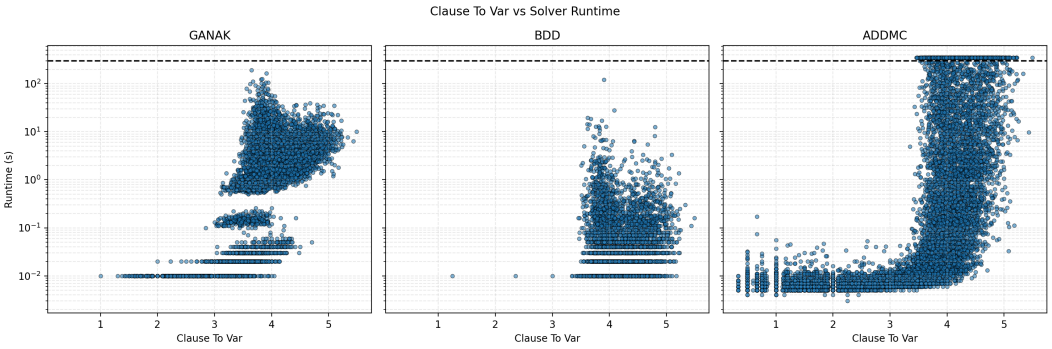


Figure 8.8: Clause-to-variable ratio versus runtime for preprocessed CNFs.

8.5.2 Treewidth

Figure 8.9 shows the relationship between runtime and treewidth for the preprocessed formulas. As with the number of variables and clauses, runtime generally increases with treewidth, although the strength of the relationship differs between model counters.

For example, raw instances all have the same number of variables and clauses within a particular encoding scheme (apart from the unary clause denoting clues) that still admits varying runtime. Count-preserving preprocessing reduces the formulas to a smaller set of variables and clauses, while also potentially changing their structural properties. This introduces greater variation in formula size and treewidth, making the relationship between these properties and runtime more apparent. The relationship between treewidth and formula size is particularly relevant for

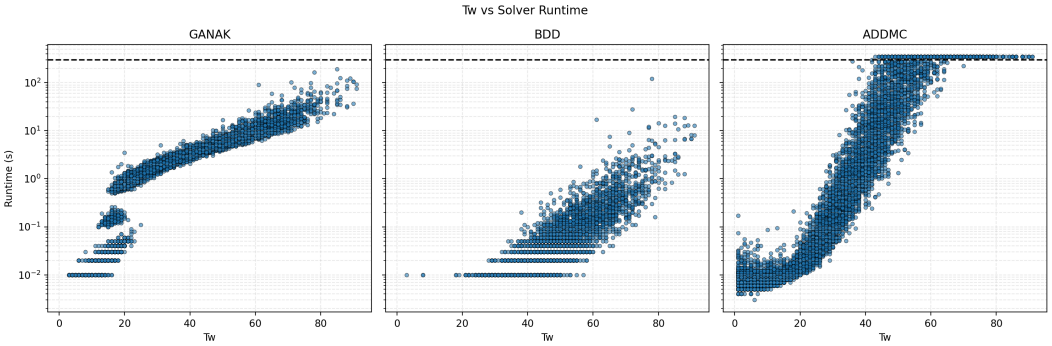


Figure 8.9: Treewidth versus runtime for preprocessed CNFs.

the preprocessed instances. The preprocessing procedure reduces the formula and can substantially alter its structural properties. Consequently, the number of variables, number of clauses, and treewidth are not necessarily independent parameters after preprocessing. This may contribute to the similar trends observed between runtime and these different formula characteristics.

For the raw instances, formulas generated from the same encoding scheme have largely fixed numbers of variables and clauses, apart from variations introduced by the clues. Nevertheless, their runtimes can vary substantially due to amount of unit clauses encoding the clues. After count-preserving preprocessing, the resulting formulas have greater variation in their size and structural properties. This provides a possible explanation for why relationships between formula size, treewidth, and runtime are more visible in the preprocessed instances.

8.5.3 Number of Solutions

Figure 8.10 shows the relationship between the number of solutions and model-counter runtime, with both quantities shown on a logarithmic scale.

A positive relationship between the number of solutions and runtime can be observed for all model counters, although the strength of the relationship varies between solvers. As discussed in Chapter 6, the number of solutions is not a structural property of the CNF formula. Nevertheless, it is considered here because exact model counting requires the solver to account for all satisfying assignments, making the number of solutions a potentially relevant factor in practical runtime. The observed relationship suggests that instances with more solutions tend to have

higher runtimes, although this relationship is weaker than those observed for some structural properties.

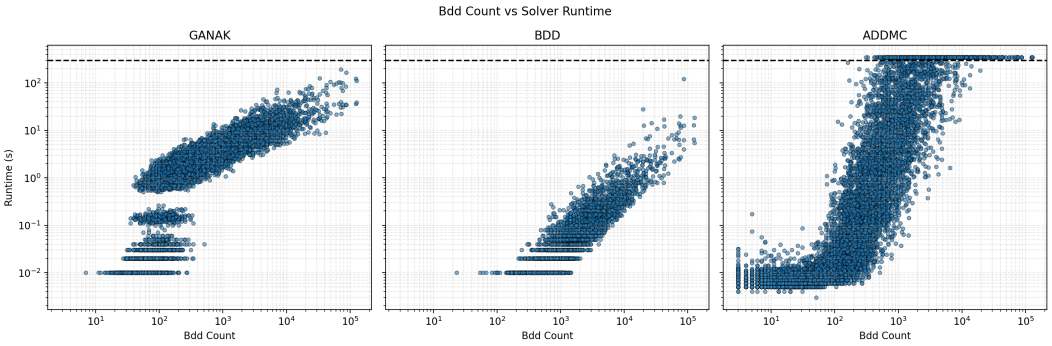


Figure 8.10: Number of solutions (denoted as bdd_count) versus runtime for preprocessed CNFs.

Chapter 9

Related Work

SAT Solving The development of modern propositional reasoning systems has its roots in early procedures for automated theorem proving and satisfiability. The Davis–Putnam procedure introduced a systematic method based on resolution, which was subsequently adapted into the DPLL procedure by Davis, Logemann, and Loveland (Davis et al., 1962). Over the following decades, SAT solving developed from primarily theoretical procedures into highly optimized practical solvers. In particular, the introduction and refinement of techniques such as efficient Boolean constraint propagation, clause-learning, and heuristic variable selection led to substantial improvements in practical performance. The `GRASP` solver proposed non-chronological backtracking with conflict-driven clause learning that became integral part of many modern CDCL solvers (Silva & Sakallah, 1999). The `Chaff` solver with its BCP and branching-heuristic optimizations demonstrated the impact that careful engineering of these techniques could have on difficult benchmark instances, reporting improvements of one to two orders of magnitude over contemporary solvers (Moskewicz et al., 2001). SAT solver development has since been closely connected to systematic empirical evaluation through solver competitions and benchmark collections.

#SAT Solving The success of SAT solving also motivated the development of practical approaches to the corresponding counting problem, #SAT. Early approaches extended the Davis–Putnam procedure directly to counting, resulting in algorithms such as `CDP` (Birnbaum & Lozinskii, 1999). Subsequent work increasingly exploited the structure encountered during search. Component decomposition and caching, for example, allow independent parts of a formula to be counted separately and their counts combined. These ideas were incorporated into practical counters such as Cachet, which combined component caching with clause learning (Sang et al., 2004), and were further developed by `sharpSAT`, which introduced more compact component representations, cache management techniques, and an adaptation of Boolean constraint propagation for model counting (Thurley, 2006). In parallel, knowledge-compilation approaches such as BDDs were developed, however with little success on SAT solving compared to other solvers (Bryant, 1986; Fichte, Berre, Hecher, & Szeider, 2023). However, representations such as BDDs and d-DNNFs (Darwiche & Marquis, 2002; Darwiche, 2004), provided new opportunities to perform model counting efficiently on compiled representations.

Preprocessing Preprocessing has long been an important part of SAT solving, where simplification techniques are applied before or during search to reduce the effective difficulty of an instance. Techniques including variable elimination, clause elimination, subsumption, propagation, and other forms of formula simplification have become standard components of modern

SAT solving (Biere et al., 2021). For example, `SatELite` combined variable elimination with subsumption and self-subsuming resolution (Eén & Biere, 2005). For model counting, however, the additional correctness requirement to preserve the model count has motivated research specifically addressing preprocessing for propositional model counting. Lagniez and Marquis developed `pmc`, a dedicated model-counting preprocessor incorporating techniques such as occurrence reduction, vivification, backbone identification, and equivalence and gate identification, and evaluated their effects on model-counting performance (Lagniez & Marquis, 2014, 2017). Their empirical study is particularly relevant to the present work, as it considered not only reductions in the number of variables and clauses but also changes in the treewidth of the resulting formulas (Lagniez & Marquis, 2017). The `Arjun` preprocessor evaluated in this thesis represents a more recent development on count-preserving preprocessing that combines selected preprocessing techniques with independent-support and definability and became part of `Ganak` model counter (Soos & Meel, 2022, 2024).

CNF Structural Properties A related line of research has investigated the extent to which the structural properties of propositional formulas, such as treewidth can be exploited for model counting. Samer and Szeider presented algorithms for model counting based on the primal, dual, and incidence graphs of a CNF formula, demonstrating a direct relationship between the width of these structures and the complexity of the resulting counting algorithms (Samer & Szeider, 2010a). Subsequent work has further improved algorithms parameterized by treewidth, including approaches based on incidence treewidth (Slivovsky & Szeider, 2020). Treewidth has therefore become both a theoretical parameter for understanding the complexity of model counting and a structural property that can be exploited by practical model counters. More recent counters such as `SharpSAT-TD` have incorporated tree-decomposition information directly into their solving strategies, further demonstrating the practical relevance of structural information (Korhonen & Järvisalo, 2023).

Empirical Evaluation Empirical evaluation has played an important role throughout the development of SAT and #SAT solvers. SAT competitions, such as The International SAT Competition and Model Counting (MC) Competition have provided a common framework in which solvers can be compared on standardized benchmark collections containing industrial, crafted, random, and application-oriented instances (Berre & Simon, 2003; Fichte, Hecher, & Hamiti, 2021). The composition of benchmark collections has consequently evolved alongside solver technology, with new benchmark families being introduced to expose different challenges and structural properties.

Sudoku and #SAT Sudoku has also been used as a structured benchmark for automated reasoning and SAT solving. The two encoding schemes used in thesis was presented and evaluated by Lynce and Ouaknine (Lynce & Ouaknine, 2006). Subsequent work has considered alternative and optimized CNF encodings, motivated in part by the large numbers of variables and clauses produced by basic encoding approaches (Kwon & Jain, 2006). These studies demonstrate the suitability of Sudoku for investigating the interaction between an underlying combinatorial problem, its propositional encoding, and solver performance. The use of Sudoku in the context of model counting has received considerably less attention; nevertheless, Sudoku provides a natural source of structured #SAT instances. Recent work has investigated the capabilities of LLMs for model counting on Sudoku instances, including experiments with different representations and transformations, such as CNF and d-DNNF, using Sudoku benchmark CNF instances from the 2022 Model Counting Competition (Fichte & Hecher, 2025; Corrêa, Fichte, & Hecher, 2026).

Chapter 10

Conclusions and Future Work

This thesis investigated the effects of count-preserving preprocessing and CNF structure on the performance of exact model counters. Sudoku puzzles were used as a source of structured SAT instances, with clues removed to generate puzzles with multiple solutions. Both minimal and extended CNF encodings were considered, and the resulting formulas were evaluated in raw and preprocessed form using three model counters. Somewhat surprisingly, `CNF2OBDD` outperformed `Ganak` on most evaluated instances. This may be explained by the instances being sufficiently small to keep OBDD compilation tractable.

Contrary to initial expectations, the effect of preprocessing was strongly dependent on the model counter. `ADDMC` showed the most substantial improvement, changing from solving no raw instances within the timeout to solving the majority of preprocessed instances. In contrast, preprocessing did not consistently reduce runtime for `Ganak` or `CNF2OBDD`. For `Ganak`, preprocessing was beneficial for some smaller instances but could introduce additional overhead for larger ones, while `CNF2OBDD` was often faster on the raw formulas. These results demonstrate that count-preserving preprocessing can substantially reduce the practical difficulty of a formula, but its benefits depends on the model-counting approach used.

Formula size and treewidth were both positively associated with model-counter runtime as expected, although the strength of these relationships differed between counters. `ADDMC` showed the strongest dependence on the number of variables, clauses, and treewidth, consistent with the importance of structural decomposition in its underlying approach. However, reductions in formula size and treewidth did not universally result in lower runtime. In particular, `CNF2OBDD` could be slower on preprocessed formulas despite reductions in these parameters, indicating that formula size and treewidth alone do not fully explain model-counter performance.

The choice of Sudoku encoding also affected the resulting preprocessed formulas. The extended encoding generally produced fewer variables and lower treewidth after preprocessing and showed a modest runtime advantage for most instances. However, these effects were neither universal nor as strong as initially expected.

Similarly, the number of solutions showed a positive relationship with runtime for all model counters, although this relationship was weaker than those observed for several structural characteristics. Thus, solution count may influence practical model-counting performance but does not by itself determine instance difficulty. Sudoku provided a useful controlled setting for study-

ing this relationship because the number of solutions can be varied through the removal of clues while retaining the same underlying problem domain.

The experiments also revealed opportunities for constructing more challenging Sudoku-derived benchmarks. During pilot runs, a puzzle with as few as 17 clues was encountered for which removing one additional clue resulted in a large increase in the number of solutions and substantially increased the runtime of all model counters. In some cases, `CNF2OBDD` also terminated with memory errors using its basic configuration. These instances were excluded from the main analysis and therefore do not support conclusions about their performance. Nevertheless, they could provide challenging benchmarks for future experiments, particularly with longer timeout. A benchmark collection organized by clue count and number of solutions could provide a controlled way of increasing instance difficulty.

Overall, the experiments demonstrate that model-counter performance cannot be fully explained by a single formula parameter or by preprocessing alone. The benefits of count-preserving preprocessing depend strongly on the model counter, while formula size and treewidth provide useful but incomplete indicators of difficulty. The results therefore highlight the importance of considering both the subtle effects of transformations applied to a formula and the structural characteristics of the resulting formula when evaluating model counters.

Modified Sudoku puzzles provided an interesting benchmark for these experiments because their CNF encodings exhibit complex variable interactions while sharing a largely fixed core structure. Future work could extend the analysis to different Sudoku-to-SAT encoding, investigate harder Sudoku-derived instances, and examine various preprocessing transformations to determine which transformations are responsible for changes in formula structure and model-counter performance. Increasing the timeout could also reveal performance differences that are not visible within the five-minute limit. Even though model counting remains intractable in the general case, studies of this kind could focus on problems from selected practical applications, where different choices of encoding, preprocessing technique, and model counter can be evaluated based on their effects on structural parameters and model-counter performance. Such analyses could help identify deeper relationships between problem structure and solver performance, potentially indicating which instances can be solved efficiently within a particular application domain.

References

Aghayeva, N. (2025). A comprehensive guide to the classical NP-complete problems. doi: 10.5281/zenodo.16795417

Alsinet, T., Béjar, R., Cabiscol, A., Fernández, C., & Manyà, F. (2002). Minimal and redundant SAT encodings for the all-interval-series problem. In *CCIA* (Vol. 2504, pp. 139–144). Springer.

Amarilli, A., Capelli, F., Monet, M., & Senellart, P. (2020). Connecting knowledge compilation classes and width parameters. *Theory Comput. Syst.*, *64*(5), 861–914.

Arnborg, S., Corneil, D. G., & Proskurowski, A. (1987, April). Complexity of finding embeddings in a k-tree. *SIAM J. Algebraic Discrete Methods*, *8*(2), 277–284. Retrieved from `https://doi.org/10.1137/0608024` doi: 10.1137/0608024

Aspvall, B., Plass, M. F., & Tarjan, R. E. (1979). A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Inf. Process. Lett.*, *8*(3), 121–123.

Bahar, R. I., Frohm, E. A., Gaona, C. M., Hachtel, G. D., Macii, E., Pardo, A., & Somenzi, F. (1997). Algebraic decision diagrams and their applications. *Formal Methods Syst. Des.*, *10*(2/3), 171–206.

Berre, D. L., & Simon, L. (2003). The essentials of the SAT 2003 competition. In *SAT* (Vol. 2919, pp. 452–467). Springer.

Biere, A., Järvisalo, M., & Kiesl, B. (2021). Preprocessing in SAT solving. In *Handbook of satisfiability* (Vol. 336, pp. 391–435). IOS Press.

Birnbaum, E., & Lozinskii, E. L. (1999). The good old davis-putnam procedure helps counting models. *J. Artif. Intell. Res.*, *10*, 457–477.

Bodlaender, H. L. (1993). A tourist guide through treewidth. *Acta Cybern.*, *11*(1-2), 1–21.

Bova, S., & Slivovsky, F. (2017). On compiling structured CNFs to OBDDs. *Theory Comput. Syst.*, *61*(2), 637–655.

Bright, C., Gerhard, J., Kotsireas, I. S., & Ganesh, V. (2019). Effective problem solving using SAT solvers. In *MC* (Vol. 1125, pp. 205–219). Springer.

Bryant, R. E. (1986). Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, *35*(8), 677–691.

Bryant, R. E. (1992). Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, *24*(3), 293–318.

Cha, B., & Iwama, K. (1996). Adding new clauses for faster local search. In *Aaai/iaai, vol. 1* (pp. 332–337). AAAI Press / The MIT Press.

Corrêa, A. B., Fichte, J. K., & Hecher, M. (2026). *Can large language models help with model counting?* Retrieved from `https://openreview.net/forum?id=JZCgbU8irx`

Darwiche, A. (2001). Decomposable negation normal form. *J. ACM*, *48*(4), 608–647.

Darwiche, A. (2004). New advances in compiling CNF to decomposable negation normal form. In *Proceedings of the 16th european conference on artificial intelligence* (p. 318–322). NLD: IOS Press.

Darwiche, A. (2011). SDD: A new canonical representation of propositional knowledge bases. In *IJCAI* (pp. 819–826). IJCAI/AAAI.

Darwiche, A., & Marquis, P. (2002). A knowledge compilation map. *J. Artif. Intell. Res.*, *17*, 229–264.

Davis, M., Logemann, G., & Loveland, D. W. (1962). A machine program for theorem-proving. *Commun. ACM*, *5*(7), 394–397.

Defresne, M., Barbe, S., & Schiex, T. (2023). Scalable coupling of deep learning with logical reasoning. *CoRR*, *abs/2305.07617*.

de Jong, T. (2023). *Bachelor thesis: Mosaic as a sat problem.* Retrieved from `https://www.cs.ru.nl/bachelors-theses/2023/Thijs_de_Jong___1015438___Mosaic_as_a_SAT_problem.pdf`

Dudek, J. M., Phan, V., & Vardi, M. Y. (2020). ADDMC: weighted model counting with algebraic decision diagrams. In *AAAI* (pp. 1468–1476). AAAI Press.

Eén, N., & Biere, A. (2005). Effective preprocessing in SAT through variable and clause elimination. In *SAT* (Vol. 3569, pp. 61–75). Springer.

Eén, N., Mishchenko, A., & Amla, N. (2010). A single-instance incremental SAT formulation of proof- and counterexample-based abstraction. In *FMCAD* (pp. 181–188). IEEE.

Eén, N., & Sörensson, N. (2003). An extensible SAT-solver. In *SAT* (Vol. 2919, pp. 502–518). Springer.

Ercsey-Ravasz, M., & Toroczkai, Z. (2012). The chaos within sudoku. *CoRR*, *abs/1208.0370*.

Felgenhauer, B., & Jarvis, F. (2005). Enumerating possible sudoku grids.

Fichte, J. K., Berre, D. L., Hecher, M., & Szeider, S. (2023). The silent (r)evolution of SAT. *Commun. ACM*, *66*(6), 64–72.

Fichte, J. K., & Hecher, M. (2025). The model counting competitions 2021-2023. *CoRR*, *abs/2504.13842*.

Fichte, J. K., Hecher, M., & Hamiti, F. (2021). The model counting competition 2020. *ACM J. Exp. Algorithmics*, *26*, 13:1–13:26.

Fichte, J. K., Lodha, N., & Szeider, S. (2017). SAT-based local improvement for finding tree decompositions of small width. In *SAT* (Vol. 10491, pp. 401–411). Springer.

Froleyks, N., Heule, M., Iser, A., Järvisalo, M., & Suda, M. (2021). SAT competition 2020. *Artif. Intell.*, *301*, 103572.

Ganian, R., & Szeider, S. (2021). New width parameters for SAT and #SAT. *Artif. Intell.*, *295*, 103460.

Gomes, C. P., Sabharwal, A., & Selman, B. (2009). Model counting. In *Handbook of satisfiability* (Vol. 185, pp. 633–654). IOS Press.

Janota, M., Lynce, I., & Marques-Silva, J. (2015). Algorithms for computing backbones of propositional formulae. *AI Commun.*, *28*(2), 161–177.

Jr., R. J. B., & Pehoushek, J. D. (2000). Counting models using connected components. In *AAAI/IAAI* (pp. 157–162). AAAI Press / The MIT Press.

Jr., S. B. A. (1978). Binary decision diagrams. *IEEE Trans. Computers*, *27*(6), 509–516. doi: 10.1109/TC.1978.1675141

Kask, K., & Dechter, R. (1995). GSAT and local consistency. In *IJCAI (1)* (pp. 616–623). Morgan Kaufmann.

Korhonen, T., & Järvisalo, M. (2023). SharpSAT-TD in model counting competitions 2021-2023. *CoRR*, *abs/2308.15819*.

Kutzkov, K. (2007). New upper bound for the #3-SAT problem. *Inf. Process. Lett.*, *105*(1), 1–5.

Kwon, G., & Jain, H. (2006). Optimized CNF encoding for sudoku puzzles.

Lagniez, J., & Marquis, P. (2014). Preprocessing for propositional model counting. In *AAAI* (pp. 2688–2694). AAAI Press.

Lagniez, J., & Marquis, P. (2017). On preprocessing techniques and their impact on propositional model counting. *J. Autom. Reason.*, *58*(4), 413–481.

Lee, C. Y. (1959). Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, *38*(4), 985-999. Retrieved from `https://onlinelibrary.wiley.com/doi/abs/10.1002/j.1538-7305.1959.tb01585.x` doi: https://doi.org/10.1002/j.1538-7305.1959.tb01585.x

Li, J. (2019). *15-859ff: Lecture 17 - coping with intractability cmu, fall 2019.* Retrieved from `https://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15859-f19/www/scribes/lecture17.pdf`

Luo, M., Li, C., Xiao, F., Manyà, F., & Lü, Z. (2017). An effective learnt clause minimization approach for CDCL SAT solvers. In *IJCAI* (pp. 703–711). ijcai.org.

Lynce, I., & Ouaknine, J. (2006). Sudoku as a SAT problem. In *Ai&m.*

McLean, G. (2021). *Grantm/sudoku-exchange-puzzle-bank: Data set of sudoku puzzles.* Retrieved from `https://github.com/grantm/sudoku-exchange-puzzle-bank`

Meelgroup. (n.d.). *Meelgroup/treedecomp: Flow cutter, and other tools needed to perform tree decomposition.* Retrieved from `https://github.com/meelgroup/treedecomp`

Mertens, S., Mézard, M., & Zecchina, R. (2006). Threshold values of random K-SAT from the cavity method. *Random Struct. Algorithms*, *28*(3), 340–373.

Mitchell, D. G., Selman, B., & Levesque, H. J. (1992). Hard and easy distributions of SAT problems. In *AAAI* (pp. 459–465). AAAI Press / The MIT Press.

Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., & Malik, S. (2001). Chaff: Engineering an efficient SAT solver. In *DAC* (pp. 530–535). ACM.

Newsham, Z., Ganesh, V., Fischmeister, S., Audemard, G., & Simon, L. (2014). Impact of community structure on SAT solver performance. In *SAT* (Vol. 8561, pp. 252–268). Springer.

Oztok, U., & Darwiche, A. (2018). An exhaustive DPLL algorithm for model counting. *J. Artif. Intell. Res.*, *62*, 1–32.

Robertson, N., & Seymour, P. D. (1986). Graph minors. II. algorithmic aspects of tree-width. *J. Algorithms*, *7*(3), 309–322.

Samer, M., & Szeider, S. (2010a). Algorithms for propositional model counting. *J. Discrete Algorithms*, *8*(1), 50–64.

Samer, M., & Szeider, S. (2010b). Constraint satisfaction with bounded treewidth revisited. *J. Comput. Syst. Sci.*, *76*(2), 103–114.

Sang, T., Bacchus, F., Beame, P., Kautz, H. A., & Pitassi, T. (2004). Combining component caching and clause learning for effective model counting. In *SAT*.

Satisfiability suggested format. (1993). Retrieved from `https://www.cs.ubc.ca/~babic/doc/dimacs_cnf.pdf`

Sharma, S., Roy, S., Soos, M., & Meel, K. S. (2019). GANAK: A scalable probabilistic exact model counter. In *IJCAI* (pp. 1169–1176). ijcai.org.

Silva, J. P. M., & Sakallah, K. A. (1999). GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, *48*(5), 506–521.

Simonis, H. (2005). Sudoku as a constraint problem.. Retrieved from `https://api.semanticscholar.org/CorpusID:115950646`

Slater, L., Meyer, T., & Heyninck, J. (2025). Toward defeasible reasoning using knowledge compilation techniques. In *NMR* (Vol. 4071, pp. 239–252). CEUR-WS.org.

Slivovsky, F., & Szeider, S. (2020). A faster algorithm for propositional model counting parameterized by incidence treewidth. In *SAT* (Vol. 12178, pp. 267–276). Springer.

Soos, M., & Meel, K. S. (2022). Arjun: An efficient independent support computation technique and its applications to counting and sampling. In *ICCAD* (pp. 71:1–71:9). ACM.

Soos, M., & Meel, K. S. (2024). Engineering an efficient preprocessor for model counting. In *DAC* (pp. 108:1–108:6). ACM.

Soos, M., & Meel, K. S. (2025). Engineering an efficient probabilistic exact model counter. In *CAV (3)* (Vol. 15933, pp. 72–91). Springer.

Thurley, M. (2006). sharpsat - counting models with advanced component caching and implicit BCP. In *SAT* (Vol. 4121, pp. 424–429). Springer.

Toda, T. (n.d.). *CNF2OBDD or BDD_MINISAT_ALL.* Retrieved from `http://www.sd.is.uec.ac.jp/toda/code/cnf2obdd.html`

Toda, T., & Soh, T. (2016). Implementing efficient all solutions SAT solvers. *ACM J. Exp. Algorithmics*, *21*(1), 1.12:1–1.12:44.

Toda, T., & Tsuda, K. (2015). BDD construction for all solutions SAT and efficient caching mechanism. In *SAC* (pp. 1880–1886). ACM.

Valiant, L. G. (1979). The complexity of computing the permanent. *Theor. Comput. Sci.*, *8*, 189–201.

van Oers, N. (2024). *Bachelor's thesis: Pattern as an sat problem.* Retrieved from `https://www.cs.ru.nl/bachelors-theses/2024/Nick_van_Oers___1009378___Pattern_as_an_SAT_problem.pdf`

van Stiphout, M. (2020). *Bachelor's thesis: Flood-it as a sat problem.* Retrieved from `https://www.cs.ru.nl/bachelors-theses/2020/Milan_van_Stiphout___4596269__Flood-It_as_a_SAT_Problem.pdf`

Wahlström, M. (2008). A tighter bound for counting max-weight solutions to 2SAT instances. In *IWPEC* (Vol. 5018, pp. 202–213). Springer.

Weber, T. (2005). A SAT-based sudoku solver.

Wegener, I. (2000). *Branching programs and binary decision diagrams.* SIAM.

Yato, T., & Seta, T. (2003). Complexity and completeness of finding another solution and its application to puzzles. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, *86-A*(5), 1052–1060.

Zhang, H., & Stickel, M. E. (2000). Implementing the davis-putnam method. *J. Autom. Reason.*, *24*(1/2), 277–296.

Zhou, J., Yin, M., & Zhou, C. (2010). New worst-case upper bound for #2-SAT and #3-SAT with the number of clauses as the parameter. In *AAAI* (pp. 217–222). AAAI Press.

Appendix A

Appendix

A.1 Simple CNF Example

Below we show naive enumeration of possible solutions (models) for CNF-SAT formula example from 2.3.

x_1	x_2	x_3	x_4	x_5	ϕ
0	0	0	0	0	**T**
0	0	0	0	1	**T**
0	0	0	1	0	**T**
0	0	0	1	1	**T**
0	0	1	0	0	**F**
0	0	1	0	1	**T**
0	0	1	1	0	**T**
0	0	1	1	1	**T**
0	1	0	0	0	**F**
0	1	0	0	1	**F**
0	1	0	1	0	**F**
0	1	0	1	1	**F**
0	1	1	0	0	**F**
0	1	1	0	1	**F**
0	1	1	1	0	**F**
0	1	1	1	1	**F**

x_1	x_2	x_3	x_4	x_5	ϕ
1	0	0	0	0	**F**
1	0	0	0	1	**F**
1	0	0	1	0	**F**
1	0	0	1	1	**F**
1	0	1	0	0	**F**
1	0	1	0	1	**F**
1	0	1	1	0	**F**
1	0	1	1	1	**F**
1	1	0	0	0	**F**
1	1	0	0	1	**F**
1	1	0	1	0	**F**
1	1	0	1	1	**F**
1	1	1	0	0	**F**
1	1	1	0	1	**F**
1	1	1	1	0	**F**
1	1	1	1	1	**F**

Table A.1: Truth table for $\phi = (x_1 \vee \neg x_2) \wedge (\neg x_1) \wedge (x_2 \vee \neg x_3 \vee x_4 \vee x_5) \wedge (\neg x_2 \vee x_3 \vee x_5)$.

A.2 DIMACS format

In the DIMACS format, each Boolean variable is assigned a unique positive integer identifier. Positive literals are represented by their corresponding integer, while negated literals are represented by the negative value of that integer. Each clause is written on a separate line and terminated by the integer `0`. A header line specifies the number of variables and clauses:
The general structure of a DIMACS CNF file is shown below.

```
p cnf <number_of_variables> <number_of_clauses>
c This is a comment
<clause 1> 0
<clause 2> 0
...
<clause m> 0
```

As DIMACS format works with 1-indexed consecutive integers as variable names; i.e., $V = \{1, 2, 3, \ldots, n\}$ with $|V| = n$, the CNF encoding in experimental pipeline labels variables differently that what was introduced in Chapter 4 that labeled variables in form x_{rcd} where $r, c, d \in 1, 2, \ldots, 9$. To account for the DIMACS supporting variable numbering scheme, the following formula is used within Java program to translate rcd format to DIMACS format.

$$var(r, c, d) = 81 * (r - 1) + 9 * (c - 1) + d$$