KU LEUVEN

FACULTY OF
ENGINEERING SCIENCE

Hierarchical Symmetric Component Caching In Model Counting

Ibrahim El Kaddouri

Thesis voorgedragen tot het behalen
van de graad van Master of Science
in de ingenieurswetenschappen:
computerwetenschappen, hoofdoptie
Artificiële intelligentie

Promotor:
Prof. dr. L. De Raedt

Evaluator:
Dr. H. Bierlee

Begeleider:
Dr. ir. V. Derkinderen

Academiejaar 2025 – 2026

Preface

This thesis would not have been possible without the patience and dedication of my assistant-supervisor. I have nothing but the utmost respect for the profound kindness and guidance he has shown me throughout this journey.

I would like to express my sincere gratitude to my supervisor, professor Luc De Raedt, whose inspiration shaped the broader direction of this research. His prompt responses and continued support throughout my time at the university have been invaluable.

My thanks also extend to Staf Rys, who generously dedicated time from his own demanding schedule to proofread this thesis. Additionally, I am grateful to the nauty community and Brendan McKay for their valuable assistance in addressing various technical questions that arose during this work.

Most importantly, I wish to thank my family for providing both the privilege and opportunity to pursue university studies, and for their unwavering patience and support during the more challenging moments of this academic endeavour.

Ibrahim El Kaddouri

Contents

Abstract

Model counters determine the number of satisfying assignments to Boolean formulas. These solvers employ component caching to avoid redundant computation by storing previously encountered components along with their model counts. The symmetrical scheme represents the state-of-the-art approach to component caching, using canonical graph labels on equivalent graph representations of components to detect structurally identical components that traditional schemes miss. However, computing canonical labels introduces severe computational overhead, with cache access times exceeding traditional schemes by several orders of magnitude.

This thesis investigates whether the benefits of the symmetrical scheme can be preserved while substantially reducing its computational overhead through multi-layer cache architectures. The proposed design introduces intermediate filtering layers that use lightweight graph invariants to reject non-matching components before invoking expensive canonical labelling. Three research questions guide the investigation: which invariants provide optimal trade-offs between discriminative power and computational cost, how multi-layer architectures affect solver performance compared to single-layer caching and whether compact encodings can further reduce memory overhead.

The experimental evaluation across 212 diverse problem instances establishes that invariants combining basic structural properties with average neighbour degree sequences achieve 97% discriminative precision while operating approximately one hundred times faster than canonical labelling. The optimal two-layer cache configuration achieves a 13% improvement in solving time (PAR-2) and solves eleven additional instances compared to single-layer baselines, with particularly dramatic speed-ups on certain problem classes. This configuration employs a combination of number of edges and average degree as graph invariants, paired with cache-aware centrality branching for variable selection and the `xxHash` algorithm for computing hash values.

However, the investigation also reveals negative results. The graph quantization technique that compresses canonical labels degrades rather than improves performance. This outcome indicates that the multi-layered cache approach has already addressed memory pressure, rendering further compression counterproductive. Hash function quality emerges as equally important as invariant precision, with poor hash distribution negating the benefits from even highly discriminative invariants.

Samenvatting

Modelcounters proberen het aantal mogelijke oplossingen voor Booleaanse formules te tellen. Deze programma's gebruiken een cache om redundante berekeningen te vermijden door eerder aangetroffen deelformules op te slaan, samen met hun oplossingsaantallen. Het symmetrisch schema vertegenwoordigt de state-of-the-art codering om componenten in de cache op te slaan. Men zet formules om in hun equivalente graafvorm en berekent canonieke graaflabels, die vervolgens in de cache worden opgeslagen samen met het aantal oplossingen van die formule. Het zijn die canonieke labels die worden vergeleken met andere componenten om te beslissen of de model count van een bepaalde component kan worden hergebruikt. Het berekenen van canonieke labels introduceert echter aanzienlijke rekenkundige kosten.

Deze thesis onderzoekt of de voordelen van het symmetrisch schema behouden kunnen blijven terwijl de rekenkundige kost substantieel wordt verminderd. Dit gebeurt door middel van de implementatie van een meerlagige cache-architectuur. De voorgestelde meerlagige cache gebruikt graaf-invarianten om niet-overeenkomende componenten af te wijzen voordat dure canonieke labels worden berekend. Drie onderzoeksvragen leiden het onderzoek: welke invarianten bieden optimale afwegingen tussen onderscheidend vermogen en rekenkundige kosten, hoe beïnvloeden meerlagige architecturen de prestaties van de solver in vergelijking met enkellagige caching en kunnen compacte coderingen de geheugenkost verder verminderen?

De experimentele evaluatie over 212 diverse probleeminstanties toont aan dat op buren gebaseerde invarianten, in combinatie met kleine structurele eigenschappen, een onderscheidende precisie van 97% bereiken, terwijl ze ongeveer honderd keer sneller berekend kunnen worden dan canonieke labels. De optimale tweelagige cache-configuratie behaalt een verbetering van 13% in oplostijd (PAR-2) en lost 11 extra instanties op vergeleken met een enkellagige cache, met bijzonder dramatische versnellingen voor bepaalde probleemklassen. Deze configuratie maakt gebruik van een combinatie van aantal bogen en gemiddelde graad als graaf-invarianten, cache-bewuste centraliteitsscores als branching heuristiek en het gebruik van `xxHash` voor het berekenen van hashwaarden.

Het onderzoek onthult echter ook negatieve resultaten. De graafkwantisatietechniek die canonieke labels comprimeert, verslechtert de prestaties van de solver, omdat de meerlagige cache de geheugendruk reeds heeft aangepakt. Hashfunctiekwaliteit blijkt even belangrijk als invariantprecisie, waarbij een slechte hashdistributie de voordelen van zelfs zeer onderscheidende invarianten tenietdoet.

List of Figures

List of Tables

List of Abbreviations and Symbols

Abbreviations

SAT	Boolean satisfiability decision problem.
#SAT	Counting the number of satisfying assignments of a Boolean formula.
DPLL	Davis-Putnam-Logemann-Loveland backtracking search algorithm.
#DPLL	DPLL adapted for exact model counting.
CNF	Conjunctive Normal Form, a conjunction of clauses.
NNF	Negation Normal Form.
DNF	Disjunctive Normal Form.
d-DNNF	Deterministic Decomposable Negation Normal Form.
sd-DNNF	Smooth and deterministic DNNF.
STD	Standard encoding scheme for components.
HC	Hybrid coding scheme.
HCO	Hybrid omitting scheme.
HCOP	Hybrid packing scheme.
SS	Symmetrical scheme.
SPS	Symmetrical Packing Scheme.
BCP	Boolean Constraint Propagation.
PAR-2	Penalized Average Runtime.
NAUTY	Graph automorphism / canonization tool.
L1/L2/L3	Cache levels in the multi-layer caching design.
GANAK	Model counter that uses HCOP.
SYMGANAK	GANAK variant that uses SS.
ISYMGANAK	Improved SYMGANAK variant.
VSADS	Variable State Aware Decaying Sum.
CSVSADS	Cache-State and Variable State Aware Decaying Sum.
ICSVSADS	Isomorphic Cache State and Variable State Aware Decaying Sum.

Symbols

σ	Assignment mapping variables to truth values.
$\mathcal{F}\|_\sigma$	Residual formula obtained by applying assignment σ to formula $\mathcal{F}$.
$\text{vars}(\mathcal{F})$	Set of variables occurring in formula $\mathcal{F}$.
$\text{lits}(\mathcal{F})$	Set of literals occurring in formula $\mathcal{F}$.
$R_\mathcal{F}$	Set of all models (satisfying assignments) of formula $\mathcal{F}$.
$\|R_\mathcal{F}\|$	Model count of formula $\mathcal{F}$.
$\text{Gr}(\mathcal{F})$	Graph representation of formula $\mathcal{F}$.
$\text{Canon}(\mathcal{G})$	Canonical labelling of graph $\mathcal{G}$.

Chapter 1

Introduction

The model counting problem, also referred to as #SAT, is about counting the number of possible assignments that would satisfy a Boolean formula. Unlike the decision problem SAT, which only asks whether at least one satisfying assignment exists, #SAT asks how many such assignments there are. This makes #SAT strictly harder in a complexity-theoretic sense: it is #P-complete (Valiant, 1979), meaning it is at least as hard as NP-complete problems. In fact, solving #SAT would allow one to solve SAT, but not necessarily the other way around.

There exist many computational applications that can be solved by representing the problem as a Boolean formula and counting the models as a solution (Sang et al., 2005a). Those applications include probabilistic inference (Chavira and Darwiche, 2008), neural network verification (Baluta et al., 2019), computational biology (Latour et al., 2017) and software verification (Teuber and Weigl, 2021). Some concrete examples include calculating the probability of a query in a probabilistic database (Gribkoff et al., 2014), such as Google's Knowledge Vault (Dong et al., 2014), or in probabilistic logic programs like Problog (De Raedt et al., 2007). In such cases, the logic program is translated into a propositional theory, after which the models of that theory are counted (De Raedt and Kimmig, 2015).

A #SAT solver is a computer program designed to solve the model counting problem. While many such solvers build upon the Davis-Putnam-Logemann-Loveland backtracking search algorithm (DPLL), their true strength comes from more advanced techniques that improve efficiency. One key technique is component decomposition. This involves breaking down a Boolean formula into independent subformulas that can be solved separately. For example, the formula $(A \vee \neg B) \wedge (C \vee D)$ consists of two parts that do not share variables and can therefore be counted independently. The total count is then the product of the counts of these subformulas.

To maximize the benefits of component decomposition, modern solvers employ component caching to avoid redundant computation. When the search algorithm decomposes a formula and solves individual components, it stores the model count of each component. If an identical component appears later in the search tree, the solver can reuse this precomputed count rather than recalculating it.

The effectiveness of component caching depends largely on how components are represented in the cache. Traditional caching schemes such as the *standard scheme* (STD) (Sang et al., 2004) store the contents of clauses explicitly. Unsatisfied clauses are serialized by listing their unassigned literals. The *hybrid coding scheme* (HC) encodes components by the IDs of their unsatisfied clauses and their unassigned variables. More advanced variants include the *hybrid omitting scheme*, which excludes binary clauses and the *hybrid packing scheme* (HCOP), which compresses the representations into a bit stream to reduce memory usage (Thurley, 2006).

These traditional representations share a fundamental limitation: they distinguish between components that are structurally identical. Consider the components $(A \vee \neg B)$ and $(C \vee \neg D)$. Both consist of a single clause containing one positive literal and one negative literal, making them structurally identical. However, traditional caching schemes treat them as distinct entries because they involve different variable names, leading to unnecessary recomputation.

To address this limitation, recent work (van Bremen et al., 2021) has introduced symmetrical schemes that employ canonical graph representations to detect structural equivalence. By representing each component as a coloured graph and computing its canonical labelling, these schemes can recognize isomorphic components regardless of variable naming differences. When the solver encounters a new component, it computes its canonical form and checks whether an isomorphic component has already been cached. If a match is found, the stored model count can be reused, effectively collapsing all structurally equivalent components into a single cache entry. This approach has demonstrated substantial performance improvements on a benchmark suite of combinatorial problem instances.

However, the benefits of symmetrical schemes come at a significant cost. Computing canonical graph labellings is computationally intensive, especially for larger components. Additionally, storing canonical labellings as adjacency lists requires substantially more memory than traditional component encodings. A cached component that is never retrieved again represents a net loss, as the computational effort and memory spent on canonicalization provided no benefit. This observation suggests that a more selective approach might be advantageous, one that computes expensive canonical representations only for components that are likely to be accessed multiple times.

This thesis investigates whether the computational and memory overhead of symmetrical schemes can be reduced while preserving their ability to detect structural

equivalence. The central approach explored is multi-layer caching, which introduces levels of indirection between the initial component representation and the definitive canonical form. Rather than immediately computing the symmetrical scheme for every component, the solver first stores components using a cheaper encoding combined with graph invariants. Only when a component achieves a hit in this higher-level cache, indicating potential for future reuse, does the solver invest in computing its canonical representation and moving it to a lower-level cache that employs the symmetrical scheme.

The introduction of multi-layer caching raises several important research questions. First, the choice of graph invariant for the higher-level cache significantly affects performance. The encoding must be cheap enough to avoid introducing excessive overhead on every component, yet it should provide sufficient discriminative power to filter out non-matching candidates before triggering expensive canonicalization. Understanding which invariants or combinations thereof maximize cache hit rates while minimizing average memory access time across diverse benchmark instances represents the first research question.

Second, the architecture of the cache hierarchy itself requires careful consideration. A two-layer design or adding a third layer might improve the trade-off between lookup cost and discriminative power. Each additional layer introduces overhead in terms of both lookup complexity and memory consumption, but it also provides an opportunity to employ increasingly strong invariants that progressively narrow the candidate set before invoking full canonicalization. The second research question therefore examines how multi-level cache architectures with two or three layers affect solver performance, as measured by PAR-2 score, across a benchmark suite.

Third, even when components reach the canonical cache, opportunities remain to reduce their memory footprint. Compact encoding schemes that pack the canonical labelling into a bitstream using delta encoding and variable-length fields can significantly reduce per-component memory consumption. However, such compression introduces additional computational overhead during encoding and comparison operations. Understanding how these compact representations affect the overall solver performance measured by PAR-2 score will the third research question.

The proposed multi-layer caching approach has been implemented on top of the state-of-the-art model counter `SYMGANAK`, creating a new solver variant called `ISYMGANAK`[1]. The implementation includes the multi-layer cache architecture along with amendments to the variable branching heuristic. Evaluation on a comprehensive benchmark suite of 212 problem instances demonstrates that the multi-layer cache architecture combined with the improved branching heuristic achieves approximately 13% improvement in PAR-2 score compared to `SYMGANAK`.

[1] https://github.com/IbrahimElk/isymganak

The remainder of this thesis is organized as follows. Chapter 2 provides the necessary background on model counting, component caching and graph canonicalization techniques. Chapter 3 formally states the problem, formulates three research questions and motivates the multi-layer caching approach. Chapter 4 describes the implementation details of two-layer and three-layer cache architectures as well as the graph quantization techniques for compact representation. Chapters 5–8 presents experimental results evaluating the proposed approaches across benchmark instances. Finally, Chapter 9 concludes with a summary of findings and directions for future work.

Chapter 2

Background

The chapter begins with Section 2.1, which introduces the basic concepts of propositional logic including syntax, semantics and assignments. Section 2.2 then defines the model counting problem and presents a taxonomy distinguishing exact counters from approximate counters, as well as DPLL-based approaches from knowledge compilation methods.

Section 2.3 examines the core #DPLL algorithm and its essential extensions. This section covers preprocessing techniques, unit propagation, clause learning, component decomposition and traditional component caching establishing the baseline solver architecture upon which modern improvements build. The discussion naturally leads to Section 2.4, which describes how components have historically been encoded in cache systems, progressing from the standard scheme through hybrid encodings to the packed representation that achieves substantial memory reduction.

The chapter then transitions to symmetry-aware techniques. Section 2.5 introduces the concept of structural symmetry between components and demonstrates how graph isomorphism and canonical labeling can identify components that differ in variable names but share identical structure. Section 2.6 details the practical implementation of symmetric caching, explaining how adjacency lists represent graphs and how canonical forms enable symmetry detection.

Section 2.7 provides an overview of the cache architecture, covering hash table organization with chaining, replacement policies and the semantic relationships between cached components that prevent undercounting in the presence of learned clauses. Finally, Section 2.8 addresses branching heuristics, describing component ordering strategies and the evolution of variable selection heuristics from basic literal counting through VSIDS, VSADS and ultimately to the cache-aware CSVSADS and symmetry-aware ICSVSADS heuristics that guide modern solvers.

2.1 Propositional Calculus

This section introduces the basic concepts of propositional logic such as assignments & satisfiability. It will serve as the first building block for explaining the sharp satisfiability problem and the #DPLL algorithm. The following section is heavily based on the book *The Satisfiability Problem* (Schöning and Torán, 2013) and by the course work *Modelling Of Complex Systems* (Denecker, 2025).

Propositional calculus (or propositional logic) formulas are built from propositional variables in combination with propositional connectives. A propositional variable is either true or false. The propositional connectives are negation $(\neg)$, conjunction $(\wedge)$ and disjunction $(\vee)$. These three operators form a complete set of connectives.

Definition 1 (Syntax of propositional calculus)**.** *The set of well-formed formulas is defined inductively as follows (Denecker, 2025):*

- *If P is a propositional variable, then P is a formula*
- *If α, β are formulas, then $(\neg\alpha)$, $(\alpha \wedge \beta)$, $(\alpha \vee \beta)$ are formulas*

Too many parentheses in a formula can harm its readability (Denecker, 2025). The associativity rule can be used to make parentheses implicit. Specifically, negation $(\neg)$ has higher precedence than conjunction $(\wedge)$ and conjunction binds more strongly than disjunction $(\vee)$. As an example, $\neg P \vee Q \wedge R$ is equivalent to $(\neg P) \vee (Q \wedge R)$.

Definition 2 (Semantics of propositional calculus)**.** *A truth table lists every possible assignment of truth values to the propositional variables and shows the resulting truth value of the logical formula (Denecker, 2025).*

TABLE 2.1: Truth tables of logical connectives with two propositional variables.

P	Q	$P \wedge Q$
T	T	T
T	F	F
F	T	F
F	F	F

P	Q	$P \vee Q$
T	T	T
T	F	T
F	T	T
F	F	F

P	$\neg P$
T	F
F	T

A literal refers either to a variable or a negated variable. The polarity or phase of a variable also indicates whether the variable is set to *False* or *True*. The literals of a formula $\mathcal{F}$ are denoted $lits(\mathcal{F})$, and its variables are denoted $vars(\mathcal{F})$.

Definition 3 (Assignment)**.** *An assignment σ is a mapping from (some of) the variables to the values T or F (Schöning and Torán, 2013). This is denoted as:*

$$\sigma = \{x_1 = a_1, x_2 = a_2, \ldots, x_k = a_k\} \quad \text{where } a_1, a_2, \ldots, a_k \in \{T, F\}$$

The residual formula, denoted as $\mathcal{F}|_\sigma$, is the formula obtained by applying the assignment σ to $\mathcal{F}$. It is the formula that results from substituting the values assigned by σ into $\mathcal{F}$. A formula $\mathcal{F}$ is called satisfiable if there exists a total assignment σ, i.e. an assignment that assigns a truth value to every variable in $\mathcal{F}$ such that $\mathcal{F}|_\sigma = T$. A satisfying assignment σ for a formula $\mathcal{F}$ is also called a model of $\mathcal{F}$. Denote by $R_\mathcal{F}$ the set of all models of a formula $\mathcal{F}$. Hence, for the sharp satisfiability problem, all the possible models are counted which is denoted as $|R_\mathcal{F}|$.

Definition 4 (Normal form propositional calculus)**.**

- *A clause is a disjunction $(\vee)$ of literals.*
- *A formula is in conjunction normal form (CNF) if it's a conjunction $(\wedge)$ of clauses.*

In propositional logic, for every formula $\mathcal{F}$ there is an equivalent formula $\mathcal{G}$ in conjunctive normal form (CNF) (Basson and O'Connor, 1959). The size of a clause is the number of literals it contains, which is denoted by $|C|$. Clauses with only one literal are called unit clauses. CNF provides a standardized way to express propositional formulas. There exist other normal forms such as negation normal form (NNF), disjunctive normal form (DNF) and others such as DNNF and sd-DNNF.

2.2 #SAT

Propositional model counting, also known as #SAT, involves determining the total number of models for a given propositional formula. This refers to counting the distinct truth assignments to the formula's variables that result in the formula evaluating to true (Gomes Carla P. et al., 2009).

Model counting generalizes the SAT problem and is recognized as the prototypical #P-complete problem (Valiant, 1979). Since each variable is Boolean, there are 2^N total possible assignments, where N is the number of distinct variables in the formula.

2.2.1 Taxonomy Of Model Counters

There are two major categories of model counters, exact model counters and approximate model counters (Gomes Carla P. et al., 2009). Within exact counting,

approaches are further differentiated between those that rely on DPLL-style exhaustive search and those that leverage knowledge compilation (Gomes Carla P. et al., 2009). A brief overview will be presented below, emphasizing the distinctions between these methods.

DPLL based model counting

The DPLL algorithm, initially conceived for SAT solvers (Davis et al., 1962), was later adapted for the task of #SAT and renamed #DPLL. The #DPLL algorithm is defined on conjunctive normal form (CNF) formulas. At its core, #DPLL is a backtracking search algorithm that tries to find all assignments that satisfy a formula $\mathcal{F}$. A more detailed explanation of this process follows in subsequent sections.

Example 1. *To illustrate, consider the following formula in CNF:*

$$\mathcal{F} = (A \vee B) \wedge (\neg A \vee \neg B)$$

The algorithm initially attempts to assign A the value $true$. Afterward, it may assign B to $true$ as well. However, upon evaluation, it realizes that the latest assignment does not satisfy the formula. Consequently, the algorithm backtracks, revising B to $false$ instead. Finally, it determines that $\{A = true, B = false\}$ is a model of the formula. Unlike standard DPLL, which halts upon finding the first satisfying assignment, #DPLL continues backtracking to enumerate all satisfying assignments.

Knowledge compilation based model counting

A different approach for exact model counting consists of compiling the given CNF formula into a deterministic decomposable negation normal form (d-DNNF) (Darwiche, 2004). Once in d-DNNF form, the model count can be deduced in polynomial time in the size of the compilation (Darwiche, 2000). However, despite the advantage of polynomial-time counting, the compilation process itself remains computationally challenging. A useful connection between knowledge compilation and DPLL-based approaches is that d-DNNF compilation can be achieved by capturing the execution trace of DPLL-based model counters (Huang and Darwiche, 2005).

Example 2. *$\mathcal{F}$ can be rewritten in disjunctive normal form (DNF) as:*

$$\mathcal{F} = (\neg A \wedge B) \vee (A \wedge \neg B)$$

In this case, an equivalent smooth d-DNNF representation can be constructed, as depicted in following Figure 2.1.

Model counting on the sd-DNNF circuit proceeds bottom-up: each leaf literal contributes 1*, conjunction nodes multiply the counts of their children and disjunction nodes sum the counts of their children. For the circuit in Figure 2.1, the left* `AND` *node has count* 1*, the right* `AND` *node has count* 1*, and the root* `OR` *node sums these to give the total model count* 2 *for $\mathcal{F}$ (Kimmig et al., 2017).*

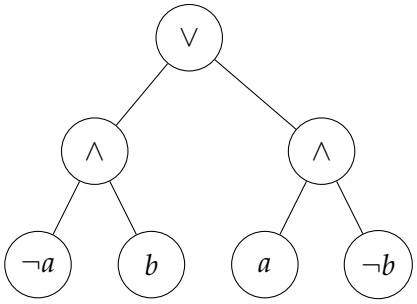


FIGURE 2.1: The sd-DNNF circuit of formula $\mathcal{F}$ (Kimmig et al., 2017).

Approximate model counting

In practical applications, precise counts may not always be necessary. Approximate solutions that provide reasonably accurate estimations suffice. As long as the method is computationally efficient, approximate counting can serve as a viable alternative to exact computation (Gomes Carla P. et al., 2009).

Several algorithms have been developed for this purpose, including `ApproxMC` (Chakraborty et al., 2013), which uses random hashing (via XOR constraints) and SAT solving to estimate the number of models with high confidence. The solver `Ganak` (Sharma et al., 2019), on the other hand, implements a probabilistic component cache scheme and is the first probabilistic exact model counter: given a formula $\mathcal{F}$ and a confidence parameter δ, it returns a count that is guaranteed to be the number of solutions of $\mathcal{F}$ with confidence at least $1 - \delta$. Hashes provide a more compact representation of components to improve cache utilization, but they can produce incorrect counts due to hash collisions (Sharma et al., 2019).

2.3 #DPLL Based Solver

This section outlines how the basic #DPLL algorithm works and explains certain extensions added on top of #DPLL including component decomposition, clause learning and component caching.

2.3.1 Preprocessing

Despite advances in #SAT solving, formulas encountered in practical applications can still be too large for solvers. Tools such as automated encoding techniques can introduce redundant clauses and variables, which would enlarge the search space and in turn make the solver performance less efficient (Biere, Armin and Järvisalo, Matti and Kiesl, Benjamin, 2021). To mitigate this issue, preprocessing techniques are employed, applying automated formula simplifications prior to solver execution to enhance overall efficiency (Biere, Armin and Järvisalo, Matti and Kiesl, Benjamin, 2021). A range of preprocessing strategies for CNF formulas has been developed, including unit propagation and other basic clause elimination techniques.

2.3.2 #DPLL Algorithm

The basic backtracking algorithm works by initially choosing a variable and assigning it a truth value. Afterwards, the formula is simplified by removing all the clauses that become true and all literals that become false from the remaining clauses. In the next step, it checks if the simplified formula is satisfiable. If this is the case, the original formula is satisfiable, the count is updated and the search is resumed by backtracking.

However, if the formula is unsatisfied at any point, #DPLL backtracks to the most recent decision point, flips the value of the last assigned variable, and resumes the search from there. In the following subsections, a number of features that improve upon the #DPLL algorithm are described (Gomes Carla P. et al., 2009).

2.3.3 Partial Counts

Consider a formula $\mathcal{F}$ with n variables, where the #DPLL algorithm has already assigned truth values to t of those variables. If the residual formula is satisfied yet there are remaining unset variables, the algorithm can assign arbitrary values to these variables since the formula is already satisfied (Gomes Carla P. et al., 2009).

As a result, the algorithm associates 2^{n-t} solutions with this branch. These solutions represent all possible ways of assigning values to the remaining $n - t$ unset variables (Gomes Carla P. et al., 2009).

2.3.4 Unit Propagation

If a formula consists of a unit clause, it can only be satisfied by an assignment that sets the clause's literal to true. Upon detecting of a unit clause in the CNF formula $\mathcal{F}$, the unit clause rule eliminates all clauses that contains this literal and removes its negation from any remaining clauses. Unit propagation can also be referred to as Boolean constraint propagation (BCP) (Biere, Armin and Järvisalo, Matti and Kiesl, Benjamin, 2021).

Unit propagation systematically applies this rule in an iterative manner until one of two outcomes is reached. Either no unit clauses remain or the process results in the derivation of an empty clause. In the latter case, a conflict arises, meaning that every literal within the conflicting clause evaluates to false (Biere, Armin and Järvisalo, Matti and Kiesl, Benjamin, 2021).

Example 3. *Consider the following simple formula* $\mathcal{F} = (A \vee B) \wedge (A) \wedge (\neg B \vee C) \wedge (\neg C)$. *The formula contains the unit clause* (A). *By applying the unit clause rule, any clause containing* A *is eliminated, leaving* $\mathcal{F} = (\neg B \vee C) \wedge (\neg C)$.

2.3.5 Clause Learning

One of the main reasons for the widespread adoption of solvers in various applications is due to clause learning (Marques-Silva et al., 2021). During the execution of a #SAT algorithm on a CNF formula $\mathcal{F}$, it is possible for a partial assignment σ to be constructed in such a way that one of the clauses C_i in $\mathcal{F}$ becomes a conflict clause. When this occurs, it becomes impossible to extend the current assignment σ.

When a conflict clause arises, *resolution* is applied to construct a clause that gives the reason for the conflict. This clause is called the *learned clause* and allows to perform *backjumping*, i.e. to backtrack over multiple levels. Moreover, the clause is stored to avoid such failed assignment to be repeated (Denecker, 2025).

For an intuitive explanation and visual demonstrations of the underlying mechanisms of clause learning, the reader is referred to online visualisation resources. Clause learning is also depicted in Algorithm 2.

2.3.6 Component Analysis

> **Definition 5** (Component). *Consider a partitioning of a formula $\mathcal{F}$ into sets of clauses $\gamma = C_1 \cup \cdots \cup C_n$ such that $vars(C_i) \cap vars(C_j) = \emptyset$ for $i \neq j$. Then each C_i is called a component of $\mathcal{F}$, and the number of models satisfies $|R_{\mathcal{F}}| = \prod_{i=1}^{n} |R_{C_i}|$ (van Bremen et al., 2021).*

Consider the constraint graph G associated with a CNF formula $\mathcal{F}$. In this graph, the vertices represent the variables of $\mathcal{F}$ and an edge exists between two vertices if the corresponding variables appear together in any clause of $\mathcal{F}$.

Example 4. *An example of component decomposition on a formula $\mathcal{F}$ is provided to illustrate this concept (Gomes Carla P. et al., 2009).*

$$\mathcal{F} = (A \vee B) \wedge (B \vee C) \wedge (D \vee E) \wedge (E \vee F) \wedge (\neg D \vee F) \wedge (G \vee H) \wedge (\neg G \vee \neg H)$$

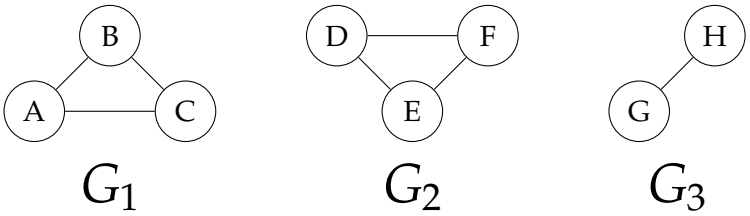


FIGURE 2.2: Constraint graph showing the independent sets of variables

Calculating the model count of $\mathcal{F}$ can be achieved by identifying the disjoint components of $\mathcal{F}$, calculating the model count for each component, and then multiplying the individual results. The identification of components occurs dynamically as the #DPLL procedure extends a partial assignment, as can be seen in Algorithm 1. With each new assignment, certain clauses are satisfied, leading to a dynamic simplification of the constraint graph, potentially creating new components (Gomes Carla P. et al., 2009).

Input : A formula $\mathcal{F}$, An empty set of learned clauses $\mathcal{G} = \emptyset$
Output: Model count of $\mathcal{F}$

```
1  function #DPLL(F):
2      encoding ← Encode(F);
3      cached ← Lookup(encoding);
4      if cached ≠ -1 then
5          return cached;
6      end
7      pick a literal l in F;
8      x ← CountConditioned(F, l);
9      y ← CountConditioned(F, ¬l);
10     CacheInsert(encoding, x + y);
11     return x + y;
12 end

13 function CountConditioned(F, l):
14     F_l ← BCP (F|_l);
15     if F_l contains empty clause then
16         return 0;
17     end
18     else if F_l contains no clauses then
19         v ← number of unassigned variables in F_l;
20         return 2^v;
21     end
22     else
23         count ← 1;
24         C ← FindDisjointComponents(F_l);
25         foreach C_i ∈ C do
26             count ← count × #DPLL(C_i);
27         end
28         return count;
29     end
30 end
```

Algorithm 1: Extended #DPLL Algorithm

Input: A formula $\mathcal{F}$, A set of learned clauses $\mathcal{G}$

```
1  function BCP(F):
2      changed ← true ;
3      while changed do
4          U ← set of unit literals in F and G;
5          changed ← false;
6          foreach u ∈ U do
7              F ← F|_u;
8              G ← G|_u;
9              changed ← true;
10             if F or G contains an empty clause then
11                 conflict-clause ← ExtractConflictClause(F, G);
12                 learned ← AnalyseConflict(F, conflict-clause);
13                 G ← G ∪ {learned};
14                 Backjump(F);
15                 break;
16             end
17         end
18     end
19     return F;
20 end
```

Algorithm 2: Clause Learning And Boolean Constraint Propagation

2.3.7 Component Caching

As the #DPLL-based model counter traverses the search tree, setting variables and simplifying the formula, it may encounter subformulas that have already appeared in a previous branch. Recognizing such repetitions and avoiding the need to recompute the model count for these subformulas is highly advantageous (Gomes Carla P. et al., 2009).

Component caching stores a mapping from a representation of a component (the key) to the model count of that component (the value). This allows for efficient reuse of previously computed model counts for subformulas encountered again during the search.

A component is represented as a set of unsatisfied clauses with falsified literals removed (Sang et al., 2004). There is a substantial number of components processed throughout the execution of the search algorithm. The storage of cached entries poses a significant challenge if outdated entries are never removed from the cache (Sang et al., 2004).

Tian Sang et al. demonstrated a concrete example of this issue arises when solving a randomly generated 3-CNF formula with 75 variables at a clause-to-variable ratio close to the challenging region of 1.8. In one such instance, the process encountered over 9 million distinct components (Sang et al., 2004).

2.4 Traditional Component Representation

Components stored in the cache can be stored in the *standard scheme* (STD) (Thurley, 2006). Components are represented by strings that omit clauses that are satisfied and literals that have been assigned (Thurley, 2006).

Example 5. *Consider the following formula:*

$$\mathcal{F} = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_4 \vee \neg x_5) \wedge (x_6 \vee x_2 \vee x_3) \wedge (x_6 \vee \neg x_4 \vee \neg x_5)$$

The corresponding encoding for this formula is provided below.
In this representation, zeros mark the end of each clause (Sang et al., 2004).

$$Encode(\mathcal{F}) = (1, 2, 3, 0, 1, -4, -5, 0, 6, 2, 3, 0, 6, -4, -5, 0)$$

On the other hand, in the *hybrid encoding scheme* (HC), only *sound* components are stored. Sound components contain at least two unassigned literals. This restriction is justified because clauses with a length of zero, empty clauses, indicate conflicts and unit clauses are already addressed via unit propagation (Thurley, 2006).

In the HC scheme, the components of a formula are represented using two strings, denoted as a and b. Specifically, string a contains the indices of the variables included in the component and string b contains the indices of the clauses that are part of the component. To reconstruct and determine whether each variable appears in its positive or negated form, the original formula $\mathcal{F}$ is consulted using the indices stored in a and b (Thurley, 2006).

Example 6. *The corresponding encoding of formula $\mathcal{F}$ in the HC encoding is provided below. In this representation, the zero acts as a separator between the two parts a and b.*

$$Encode(\mathcal{F}) = (1, 2, 3, 4, 5, 6, 0, 1, 2, 3, 4)$$

In this representation, variable and clause identifiers are used rather than clause contents. As a result, two components may look identical in the STD format but differ in the HC format.

Example 7. *Take the assignment $\sigma = \{x_6 = T,\ x_1 = F\}$. Under σ the remaining formula is $F_\sigma = (x_2 \vee x_3) \wedge (\neg x_4 \vee \neg x_5)$, which is encoded in HC format as $(2, 3, 4, 5, 0, 1, 2)$. If instead, the assignment $\tau = \{x_6 = F,\ x_1 = T\}$ is used, the reduced formula is still $F_\tau = (x_2 \vee x_3) \wedge (\neg x_4 \vee \neg x_5)$, but its HC encoding becomes $(2, 3, 4, 5, 0, 3, 4)$. Thus, the clause identifiers differ even though the components themselves are the same (Thurley, 2006).*

While this approach may occasionally prevent HC from recognizing certain components as identical, as opposed to the STD encoding would, the HC encoding nonetheless leads to a significant reduction in storage to represent a component than STD, allowing more components to be stored in the cache (Gomes Carla P. et al., 2009). For example, STD uses 16 symbols to represent the formula $\mathcal{F}$, whereas HC requires only 11 symbols.

Additionally, the *hybrid encoding scheme omitting binary clauses* (HCO) excludes all binary clauses from the original formula due to redundancy (Thurley, 2006). Suppose the formula $\mathcal{F}$ contains a binary clause $(x_1 \vee x_2)$. The binary clause appears in the component if and only if both literals remain unassigned. Hence, the occurrence of a clause identifier for this binary clause can be reconstructed directly from the presence of the individual variables in the component (Thurley, 2006).

If the original formula $\mathcal{F}$ contains any binary clauses, its HCO representation would differ from that of HC.

Example 8. *Consider the following formula $\mathcal{G}$:*

$$\mathcal{G} = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_5) \wedge (x_6 \vee x_2 \vee x_3) \wedge (x_6 \vee \neg x_4 \vee \neg x_5)$$

The HCO encoding for this formula would be $(2, 3, 4, 5, 6, 0, 1, 2)$ where the first two binary clauses are derived rather than explicitly stored. As shown, this approach reduces the size of the encoding, from $(1, 2, 3, 4, 5, 6, 0, 1, 2, 3, 4)$ to $(2, 3, 4, 5, 6, 0, 1, 2)$, effectively shortening it by 3 symbols out of 8.

Each literal in formula $\mathcal{G}$ maintains a list of literals with which it forms binary clauses. In this case, literal x_2 has a list containing (1), literal $\neg x_5$ has a list containing (1) and literal x_1 has a list containing $(2, -5)$. One might assume that these binary links would require additional space in the cache, however, they do not need to be stored in the cache! Instead, these binary links can be stored globally and shared across all components (Thurley, 2006).

The *hybrid encoding scheme omitting binary clauses packed* (HCOP) stores each component as a compact bitstream rather than as an explicit list of variable and clause identifiers (Thurley, 2006). The idea is closely related to variable-length encoding in coding theory. Instead of storing all identifiers explicitly, only the first identifier and the differences between consecutive identifiers are stored. This is efficient because variable and clause identifiers within a component are always sorted.

Example 9. *Consider again the formula*

$$\mathcal{G} = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_5) \wedge (x_6 \vee x_2 \vee x_3) \wedge (x_6 \vee \neg x_4 \vee \neg x_5).$$

The variables in the component are $(1,2,3,4,5,6)$ *and the clause indices are* $(1,2,3,4)$. *Since both sequences are consecutive, all differences equal* 1. *Thus, the total packed size is* 13 *bits*[1], *which requires one 32-bit block, instead of 10 32-bit blocks in the HCO format.*

By storing only small deltas instead of full identifiers, HCOP achieves a substantial reduction in memory usage per component, allowing significantly more components to be stored in the cache (Thurley, 2006).

2.5 Symmetric Component Representation

Traditional cache indexing schemes register a cache hit only when there is an exact match between component representations. However, an important observation is that many components, although differing in the specific variables they contain, and therefore in their representations (STD, HC, etc.), are actually *structurally identical*.

Example 10. *Consider two distinct components encountered at different locations within the search tree.*

$$\mathcal{C}_1 = (\neg x \vee y) \wedge (\neg y \vee z)$$
$$\mathcal{C}_2 = (\neg r \vee s) \wedge (t \vee \neg s)$$

Although these two components have different HCO representations, they are structurally identical. If the variables are renamed such that $z \mapsto \neg r$, $y \mapsto \neg s$ *and* $x \mapsto \neg t$, *then the second component is obtained. This transformation can be expressed as:*

$$\pi = \{z \mapsto \neg r,\ y \mapsto \neg s,\ x \mapsto \neg t\}$$

This implies that all possible assignments satisfying $\mathcal{C}_2$ *are identical to those satisfying* $\pi(\mathcal{C}_1)$. *Consequently, their corresponding solution sets and model counts are equal:*

$$R_{\mathcal{C}_2} = R_{\pi(\mathcal{C}_1)} \quad \textit{and} \quad |R_{\mathcal{C}_2}| = |R_{\pi(\mathcal{C}_1)}|$$

The advantage of symmetric component representation is that it can reuse cached model counts of structurally identical components (van Bremen et al., 2021).

Definition 6 (Symmetric Components)**.** *Two formulas* ψ_1 *and* ψ_2 *are said to be symmetric (structurally identical) if there exists a bijection (van Bremen et al., 2021)*

$$\pi : \mathit{lits}(\psi_1) \to \mathit{lits}(\psi_2)$$

such that

$$R_{\psi_2} = R_{\pi(\psi_1)} \quad \textit{and} \quad \forall l \in \mathit{lits}(\psi_1) : \neg\pi(l) = \pi(\neg l).$$

[1] 3 bits for the first variable ID and 1 bit for each consecutive variable ID $(3 + 1 \times 5)$.
2 bits for the first clause ID and 1 bit for each consecutive clause ID $(2 + 1 \times 3)$.

Before explaining how to identify structurally identical components, several concepts from graph theory that are useful for recognizing structural symmetries are introduced.

Definition 7 (Coloured Graph)**.** *A coloured graph is a triple $G = (V, E, P)$, where (V, E) defines an undirected graph, and $P = \{V_i\}_{i=1}^{k}$ is a partition of the vertices into k distinct colour classes. For any vertex $v \in V_i$,* $\text{colour}(v) = i$*, (van Bremen et al., 2021).*

Given two coloured graphs, one might ask whether they are *isomorphic*.

Definition 8 (Coloured Graph Isomorphism)**.** *Given two coloured graphs $G = (V_1, E_1, P_1)$ and $H = (V_2, E_2, P_2)$, G and H are said to be isomorphic if there exists a bijection $\varphi : V_1 \to V_2$ satisfying the following conditions (van Bremen et al., 2021):*

1. *For all $v, w \in V_1$, $(v, w) \in E_1 \iff (\varphi(v), \varphi(w)) \in E_2$*
2. *For all $v \in V_1$, $colour(v) = colour(\varphi(v))$.*

The core idea is that finding symmetries in a formula can be reduced to the problem of *coloured graph automorphism*[2]. In this approach, each variable is represented by two vertices, one for the positive literal and one for the negative literal, while each clause is represented by a single vertex. The clause vertices connect to the literal vertices if the literals appear in that clause. Clause vertices are assigned a colour and literal vertices are assigned another colour. Vertices representing opposite literals are directly connected by an edge to ensure Boolean consistency (Aloul et al., 2002).

Using such graph representation, permutational symmetries can be detected alongside phase shifts (Aloul et al., 2002).

Example 11. *Phase-shift symmetries of the form $a \leftrightarrow \neg a$ can be detected. Consider the formula $\mathcal{G} = (x_1 \vee x_2) \wedge (\neg x_1 \vee x_3)$, a phase shift on x_1 gives $\mathcal{G}' = (\neg x_1 \vee x_2) \wedge (x_1 \vee x_3)$ which is structurally identical to the original formula and thus provides the same model count. A permutational symmetry can be illustrated with swapping $x_1 \leftrightarrow x_2$ and $x_3 \leftrightarrow x_4$ which gives $\mathcal{G}'' = (x_2 \vee x_1) \wedge (x_4 \vee x_3)$, which is structurally identical to $\mathcal{G}$.*

Another simplification allows arbitrary two-literal clauses to be represented by a single edge directly connecting the two literals, instead of using two edges and a clause vertex (Aloul et al., 2002).

Going back to graph theory, a closely related problem to graph isomorphism is *graph canonization*.

[2] A graph automorphism is an isomorphism from a graph to itself.

Definition 9 (Canonical Labelling). *Given a graph G, a canonical labelling of G is a graph $Canon(G)$ such that for any graph H (van Bremen et al., 2021):*

$$H \text{ is isomorphic to } G \iff Canon(G) = Canon(H).$$

As the name suggests, $\text{Canon}(G)$ is essentially a relabelled version of G. Given an oracle for graph canonization, checking whether two graphs are isomorphic becomes straightforward: compute the canonical labelling for each graph and simply compare the results to see if they are identical.

In short, to detect symmetric components, the formulas $\mathcal{C}_1$ and $\mathcal{C}_2$ are first encoded as graph representations, $Gr(\mathcal{C}_1)$ and $Gr(\mathcal{C}_2)$. Their canonical labellings are then computed and compared for equality (van Bremen et al., 2021).

The transformation of a formula into a graph, denoted $Gr(\mathcal{F})$, is achieved through the following steps (Aloul et al., 2002, van Bremen et al., 2021):

1. Add a red node for each non-binary clause $\mathcal{C}_i$ in $\mathcal{F}$.
2. Add a blue node for each literal l_i in $\mathcal{F}$.
3. Add an edge between each literal l_i and its negated counterpart l_j.
4. Add an edge between literal l_i & l_k of a binary clause.
5. Add an edge between l_i and $\mathcal{C}_i$ if it occurs in a non-binary clause $\mathcal{C}_i$.

Example 12. *Consider the formula*

$$\mathcal{F} = (\neg x \vee y) \wedge (\neg y \vee z) \wedge (x \vee y \vee z)$$

The following figure represents the graph representation of formula $\mathcal{F}$.

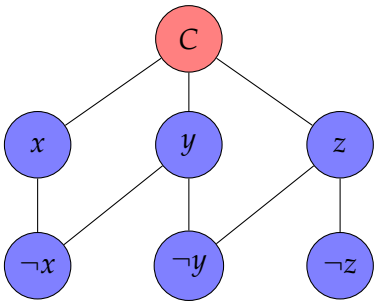


FIGURE 2.3: The graph representation of formula $\mathcal{F}$

For the oracle that provides canonical labellings of graphs, SYMGANAK (van Bremen et al., 2021) makes use of existing graph automorphism tools, specifically NAUTY (McKay and Piperno, 2014). Prior research (Puget, 2005) has demonstrated that, despite the unresolved theoretical complexity of graph isomorphism, such software performs very efficiently in practice. Furthermore, applying hashing techniques to Canon$(G(\mathcal{C}))$ allows for near-constant-time retrieval of isomorphic components (Kitching and Bacchus, 2007).

2.6 Symmetric Caching

It is important to understand how graphs are represented and how their canonical forms are obtained, as well as why hashing is necessary for efficient retrieval, as mentioned in the previous Section.

There are two standard ways to represent an undirected graph $G = (V, E)$: using adjacency lists or an adjacency matrix. The adjacency-list representation is generally preferred for sparse graphs, where $|E| \ll |V|^2$, because it provides a more compact and memory-efficient representation (Cormen et al., 2022).

In the adjacency-list representation of a graph $G = (V, E)$, there is an array e of $|V|$ lists, one for each vertex in V. For each vertex $u \in V$, the adjacency list $e[u]$ contains all vertices v such that $(u, v) \in E$. In other words, $e[u]$ lists all vertices adjacent to u in G (Cormen et al., 2022). Importantly, the vertices in each adjacency list are *not* stored in an arbitrary order !

When constructing a graph, the adjacency list is built starting from the variable with the smallest index. For each variable, the negative literal is listed first, followed by the positive literal.

Example 13. *Consider the two formulas γ and ϕ:*

$$\gamma = (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3)$$

$$\phi = (\neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_1)$$

Even though the clauses in ϕ appear in a different order, the resulting adjacency list of the graph will be identical to that of γ. This shows that the graph representation is invariant under clause reordering or literal position changes.

This property also explains why traditional cache indexing schemes can handle *permutational symmetries*, as discussed in an earlier section. The key requirement is simply to maintain a consistent construction order for literals and clauses when building any kind of representation.

Example 14. *The adjacency list and matrix for both ϕ and γ is shown below.*

Adjacency List:

$$[[x_2], [], [x_3], [\neg x_1], [\neg x_2]]$$

Adjacency Matrix:

	$\neg x_1$	x_1	$\neg x_2$	x_2	$\neg x_3$	x_3
$\neg x_1$	0	0	0	1	0	0
x_1	0	0	0	0	0	0
$\neg x_2$	0	0	0	0	1	0
x_2	1	0	0	0	0	0
$\neg x_3$	0	0	0	0	0	0
x_3	0	0	1	0	0	0

When two components differ by structural symmetries, their raw adjacency lists will differ. Graph canonization produces a canonical relabelling so that all isomorphic graphs share the same adjacency-list representation! In other words, computing the canonical labelling of two graphs is equivalent to finding a common adjacency-list form for that entire isomorphism class!
However, directly comparing adjacency matrices or lists between components quickly becomes impractical, especially when dealing with thousands of cached components. To address this, the canonical labels (i.e., the adjacency lists) are hashed. If two graphs differ, their hashes will differ as well. If their hashes match, a direct comparison of their adjacency lists is performed to confirm equivalence.

Algorithm 3 presents the `Encode` function (as referenced in Algorithm 1).

```
1 function Encode(ψ):
2     graph ← Gr(ψ)
3     canonical_label ← Canon(graph)
4     return hash(canonical_label), canonical_label
```

Algorithm 3: Encoding Symmetric Components

Isomorphic component caching inevitably introduces computational overhead during the search process. Despite the overhead, the approach proves highly effective in practice.

To quantify solver performance across benchmarks, van Bremen et al. used the *PAR-2 score* (van Bremen et al., 2021). This metric not only captures the runtime on solved instances but also penalizes unsolved instances, allowing a fair comparison between solvers. The PAR-2 score is formally defined as follows.

Definition 10 (PAR-2 Score)**.** *The PAR-2 score (Penalized Average Runtime) of a solver over a set of benchmark instances is defined as*

$$\text{PAR-2} = \frac{1}{N} \sum_{i=1}^{N} t'_i \tag{2.1}$$

where N is the total number of benchmark instances and

$$t'_i = \begin{cases} t_i, & \text{if instance } i \text{ is solved within the time limit,} \\ 2T, & \text{otherwise.} \end{cases} \tag{2.2}$$

where, t_i denotes the runtime of instance i, and T is the imposed time limit.

The results clearly showed that `SYMGANAK` (van Bremen et al., 2021) outperforms `GANAK` on certain combinatorial instances. Although it does not always dominate across all benchmarks, `SYMGANAK` achieves higher performance in terms of both the PAR-2 score and the number of solved instances. This demonstrates that, despite the overhead introduced by graph-based symmetry detection, the gains in cache effectiveness and reduced redundant computation can compensate for it.

2.7 Cache Structure

A cache implemented in software differs fundamentally from the hardware caches used in modern CPUs, even though the underlying principles are quite similar.

Modern processors employ several levels of hardware caches, most notably L1, L2 and L3 to store data close to the CPU and reduce access latency compared to main memory. Software caches are inspired by the same idea: keeping frequently accessed data readily available to avoid expensive recomputation or retrieval. However, the way they are implemented is quite different.

In software, a cache is essentially a *list* stored in main memory. The kernel allocates space for this list and as more data is added, it dynamically allocates additional memory to accommodate it. Since main memory is finite, a true software cache must enforce a size limit.

Once a list is bounded in size, a *replacement policy* must be defined to decide which entries to evict when new data needs to be inserted. Unlike hardware caches, software implementations have the luxury of maintaining rich metadata, allowing for more flexible and sophisticated eviction strategies.

Furthermore, another difference between hardware and software caches lies in how data is stored and managed. Hardware caches operate on fixed-size *cache lines*,

meaning every stored block of data occupies a uniform amount of space. In contrast, software caches are far more flexible, each entry (key-value pair) can vary greatly in size, from tiny objects to very large ones.

In summary, a software cache is simply a bounded list of data stored in main memory, managed by a replacement policy that decides which entries to evict when new ones are added. The elegant part is that many of the principles and metrics used to analyse hardware-level cache performance, such as hit rate, miss rate and latency, can also be applied to software caches.

2.7.1 Hashing And Chaining

The cache used by `SYMGANAK` is implemented as a hash table. Keys are computed from canonical labels and each key maps to a cache entry that holds the cached data for that canonical label.

Because the hash function is not perfect, multiple keys can map to the same bucket. A common and efficient method to handle collisions is *chaining*: all elements whose keys map to the same bucket are stored together in a chain associated with that bucket (Knuth, 1998).

For performance reasons related to memory locality, `SYMGANAK` implementation does not allocate one list per bucket. Instead, all cache entries are allocated in a single contiguous array called the *cache* and each entry contains a pointer to the next element in its bucket chain. Figure 2.4 illustrates this layout and Example 15 demonstrates how a cache lookup traverses such a chain during access.

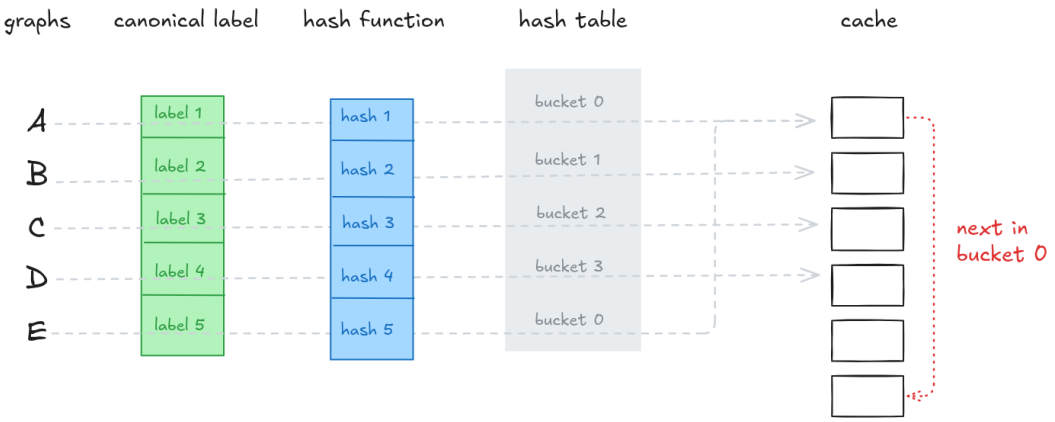


FIGURE 2.4: Hash table and chaining

As the number of stored entries grows while the number of buckets remains fixed, chains become longer. To maintain efficient lookup and insertion performance, the implementation enforces an upper bound $\alpha_{\max}$ on the load factor.

Definition 11. *The* load factor *α of a hash table is defined as*

$$\alpha = n/m$$

where n denotes the number of stored entries and m denotes the number of buckets in the cache (Cormen et al., 2022). The load factor measures the average number of entries per bucket. Larger values of α typically imply longer chains and degraded average lookup performance.

Whenever α approaches $\alpha_{\max}$, the table is resized and rehashing is performed, increasing the number of buckets and reducing average chain length (Cormen et al., 2022).

Example 15. *Consider Figure 2.4. Assume the hash table has $m = 4$ buckets and contains five entries corresponding to graphs A–E. Let the hash encodings computed from the canonical labels of each graph be:*

$$A = 12, \quad B = 1, \quad C = 2, \quad D = 3, \quad E = 8$$

Bucket indices are computed by taking each hash modulo 4. Graphs A and E both map to bucket 0, forming a chain, while B, C, and D each occupy distinct buckets.

A lookup for graph E first selects bucket 0. The chain is traversed starting from the first entry (A). The hash encoding of A does not match, so the algorithm follows the `next_bucket_element` *pointer to E. The hash encoding matches, the canonical labels are equal, and the cached value is returned. If no matching entry were found before the chain ended, the lookup would result in a cache miss.*

```
1  function Lookup(component):
2      bucket ← component.hash mod #buckets ;          // select bucket
3      idx ← hash_table[bucket] ;                      // walk the chain
4      while idx > 0 do
5          entry ← cache[idx];
6          if entry.hash_encoding = hash_encoding then
7              if entry.canonical_label = canonical_label then
8                  return entry.model_count ;          // cache hit
9              end
10         end
11         idx ← entry.next_bucket_element ;           // follow chain
12     end
13     return −1 ;                                     // cache_miss
```

Algorithm 4: Accessing Symmetric Components Cache

Algorithm 4 presents the `CacheGet` function (as referenced in Algorithm 1). The explicit comparison of hash encodings inside a bucket is required because bucket

selection uses a modulo operation. Different hash encodings may therefore map to the same bucket, making a secondary equality check necessary before performing the more expensive canonical-label comparison.

2.7.2 Replacement Policy

When the cache becomes full, a decision must be made about which existing entries to evict in order to make room for new data. The performance of this replacement policy has a significant impact on overall performance and should be chosen based on the application. Common strategies include:

- First In, First Out (FIFO) evicts the oldest data entry.
- Last In, First Out (LIFO) evicts the youngest data entry.
- Least Recently Used (LRU) evicts the least recently referenced data entry.

In `SYMGANAK`, an approximation of FIFO is implemented. It timestamps entries with a monotonically-increasing sequence number. When the cache needs space, the implementation computes a global cut-off (median of timestamps) and removes roughly half of the candidate entries (those older than the cut-off), then rehashes the hash tables.

2.7.3 Semantic Structure

Inside the cache, there are several semantic relations between components.
The relations shown in Figure 2.5 (`father`, `first_descendant` and `next_sibling`) implement these parent/child and sibling links across buckets.

Example 16. *Let ϕ be a component whose formula decomposes into two disjoint components ϕ_A and ϕ_B (so ϕ is their `father`). If the model counts are $\#\phi_A = M_A$ and $\#\phi_B = M_B$, then the model count of the father is $\#\phi = \#\phi_A \times \#\phi_B$. In this situation ϕ_A and ϕ_B are siblings of each other, whichever of them was cached most recently is the `first sibling`.*

These semantic relations are introduced because of a subtle interaction between component caching and clause learning that can cause the solver to compute a *lower bound* rather than the exact model count. The issue and the solution (sibling pruning) were described in (Sang et al., 2004).

Learned clauses $\mathcal{G}$ are entailed by the original formula $\mathcal{F}$. Satisfying assignments of $\mathcal{F}$ correspond one-to-one with those of $\mathcal{F} \wedge \mathcal{G}$. This is one of the reasons why components are detected with respect to $\mathcal{F}$ only, ignoring $\mathcal{G}$. However, learned clauses are still used during search for unit propagation and pruning, as can be seen in Algorithm 2. As a result, the search below a partial assignment σ will only see assignments that satisfy $\mathcal{F}|_\sigma \wedge \mathcal{G}|_\sigma$, and these may form a strict subset of the satisfying assignments of $\mathcal{F}|_\sigma$. This mismatch is the source of the undercount.

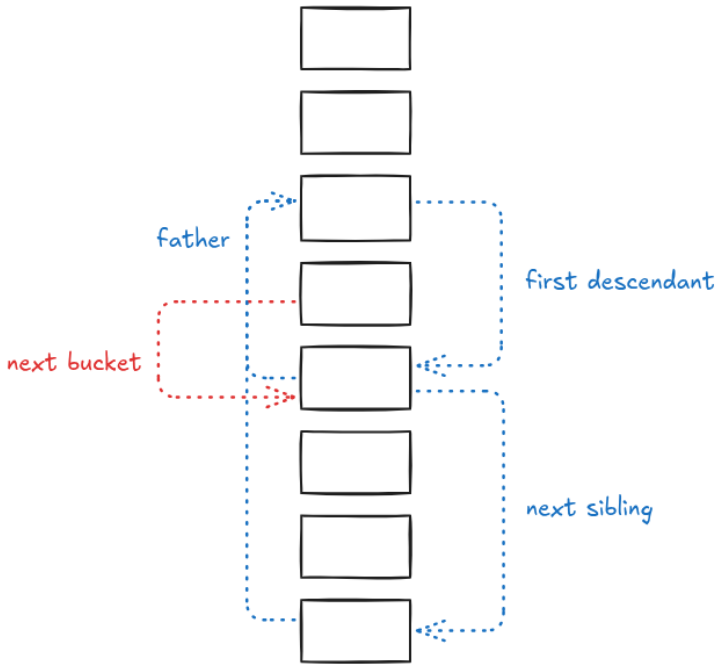


FIGURE 2.5: Semantic relations in the cache

Example 17.

$$\mathcal{F} = (p_0 \vee \neg a_1 \vee p_1) \wedge (p_0 \vee \neg p_2 \vee a_2) \wedge (a_1 \vee a_2 \vee a_3) \\ \wedge (\neg p_1 \vee b_1) \wedge (\neg b_1 \vee b_2) \wedge (\neg b_2 \vee \neg p_2)$$

$$\mathcal{G} = (p_0 \vee a_1 \vee a_2)$$

First branch $\sigma = \{p_0{=}0, p_1{=}1, p_2{=}0\}$
$\mathcal{A} = (a_1 \vee a_2 \vee a_3), \quad \mathcal{G}|_\sigma = (a_1 \vee a_2)$
$|\mathcal{A} \wedge \mathcal{G}|_\sigma| = 5$

Second branch $\sigma' = \{p_0{=}1\}$
$\mathcal{A} = (a_1 \vee a_2 \vee a_3), \quad \mathcal{G}|_{\sigma'} \equiv T$
$|\mathcal{A} \wedge \mathcal{G}|_{\sigma'}| = 7$

Thus the true count of component $\mathcal{A}$ is 7, reusing the cached value 5 would undercount due to learned clauses that falsify. Learned clauses can only falsify if one of the siblings of component $\mathcal{A}$ also falsify! Since learned clauses are entailed from formula $\mathcal{F}$.

This particular undercounting issue can be solved by removing all of its cached siblings and their descendants, as illustrated in Algorithm 5. In short, the relations (e.g., `first_descendant`, `next_sibling`) can be used in order to remove cache pollution (Sang et al., 2004).

```
1  function CacheInsert(component, model_count):
       // if cache is full, apply FIFO
2      if model_count = 0 then
3          Remove all sibling subtrees of component from cache and hash_table;
4          if component is the last branched component of its parent then
5              Add (component, 0) to the cache;
6              AddToHashTable(component);
7          end
8          else
9              Remove all descendants of component from cache and hash table;
10         end
11     end
12     else
13         Add (component, model_count) to the cache;
14         AddToHashTable(component);
15     end

16 function AddToHashTable(component):
17     bucket ← component.hash mod #buckets;
18     old_head ← hash_table[bucket];
19     cache[idx].next_bucket_element ← old_head;
20     hash_table[bucket] ← idx;
```

Algorithm 5: Insert Model Count In Cache & Remove Cache Pollution

2.8 Branching Decision Strategy

Branching decisions consist of three choices (Sang et al., 2005b):

1. *Which component* to explore.
2. *Which variable* inside the chosen component to branch on.
3. *Which polarity* is the chosen variable.

2.8.1 Component Ordering

At each decision point, the residual formula decomposes into disjoint components. Each component can be treated almost independently, although learned clauses can create cross-component interactions (Sang et al., 2005b). If a component is satisfiable, its child components will also be satisfiable. Therefore, every child must eventually be explored and cached results can be reused (Sang et al., 2005b). By contrast, an unsatisfiable component implies that at least one child is unsatisfiable, making work on other children unnecessary for proving the parent's unsatisfiability (Sang et al., 2005b).

In addition, cached values for satisfiable children need to be removed if an unsatisfiable sibling is later discovered. For these reasons, component selection should favour strategies that reduce wasted effort when unsatisfiable siblings exist (Sang et al., 2005b).

Predicting which child is unsatisfiable without exploration is difficult, so heuristics are used. Sang et al. initially tried selecting the component with the largest clause-to-variable ratio, but this proved ineffective. A more practical approach is to select the component with the fewest variables first. Smaller components are cheaper to analyse, and abandoning their work because of an unsatisfiable sibling limits wasted computation (Sang et al., 2005b).

2.8.2 Variable Branching Heuristic

The second decision is choosing the variable to branch on. Variable-branching heuristics are crucial to the performance of DPLL-based #SAT solvers. `GANAK` and `SYMGANAK` can both use the *Cache State and Variable State Aware Decaying Sum* (CSVSADS) heuristic (Sharma et al., 2019). To understand CSVSADS, it helps to first look at the heuristics that inspired it.

Literal-count heuristics
Literal-count heuristics (Silva, 1999) choose branching variables by counting how often each variable appears as a positive literal C_p and as a negative literal C_n in the current component, then scoring variables either by the combined count $C_p + C_n$ or by the larger of the two counts $max(C_p + C_n)$. The combined approach, known as *Dynamic Largest Combined Sum* (DLCS), picks the variable with the highest $C_p + C_n$ and sets it true when $C_p \geq C_n$, and false otherwise. The alternative, *Dynamic Largest Individual Sum* (DLIS), selects the variable whose single polarity appears most often and similarly sets its value according to whether $C_p \geq C_n$. Both are dynamic because the counts are updated during search, and both aim simply to eliminate as many clauses as possible at each decision without explicitly accounting for the effects of unit propagation (Sang et al., 2005b).

Variable State Independent Decaying Sum
VSIDS heuristic (Moskewicz et al., 2001) scores each literal and updates those scores only when new learned conflict clauses are added. Each literal in a newly added learned clause has its counter incremented. It periodically divides all counters by a constant factor. The solver picks the unassigned literal with the highest counter with breaking ties randomly. After many decay steps the scores mainly reflect recent conflict clauses (Sang et al., 2005b). The polarity is then chosen using a DLCS-style rule. If the positive literal occurs at least as often as the negative literal in the current formula ($C_p \geq C_n$), the variable is assigned `true`, otherwise `false` (Sang et al., 2005b).

Variable State Aware Decaying Sum
VSADS heuristic (Sang et al., 2005b) bridges the gap between the conflict-driven focus of VSIDS and the formula-aware greediness of DLCS. The VSADS score is calculated as a weighted sum:

$$score(VSADS,\ v) = p \cdot score(VSIDS,\ v) + q \cdot score(DLCS,\ v) \tag{2.3}$$

where p and q are constant factors and v an unassigned variable. This allows the solver to behave like VSIDS during conflict-heavy phases and like DLCS when the search is smoother. Once a variable is selected by this combined score, the solver applies the same polarity rule as in DCLS: if the positive literal's occurrence count satisfies $C_p \geq C_n$, the decision is `true`, otherwise `false` (Sang et al., 2005b).

Cache State and Variable State Aware Decaying Sum
The CSVSADS heuristic (Sharma et al., 2019) extends VSADS by incorporating component-cache awareness to improve cache hit rates. CSVSADS discourages branching on variables whose components were recently added to the cache (Sharma et al., 2019). It introduces two guiding parameters: a cache score which prioritizes variables whose components were not recently cached and a hyperparameter r, which determines the top percentage of variables by VSADS score to consider. The third step selects among candidates the variable with the highest cache score (Sharma et al., 2019).

$$VSADS_{max} = max\{score(VSADS,\ v) \mid v \in \text{unassigned variables}\} \tag{2.4}$$

The cache score (CS) itself is maintained dynamically. Whenever a cache hit or cache store occurs, the cache score of all variables appearing in that component is decremented (Sharma et al., 2019). Periodic increments are applied to all variable scores by multiplying them by a constant decay factor. This mechanism ensures that variables whose components are already cached receive lower priority, effectively discouraging future branching on them and improving overall cache utilization (Sharma et al., 2019).

$$score(CSVSADS, v) = \begin{cases} score(CS, v), & \text{if } score(VSADS, v) > r \cdot VSADS_{max} \\ -\infty, & \text{otherwise} \end{cases} \tag{2.5}$$

After selecting a variable, polarity is also determined by comparing occurrence counts. However, CSVSADS introduces randomization to weaken this choice when the preference is not strong. The `GANAK` and `SYMGANAK` solvers maintain a polarity cache, similar to (Pipatsrisawat and Darwiche, 2007), recording previously assigned polarities. For a variable v that appears in this cache, if neither polarity dominates (i.e., neither exceeds the other by a factor of two), the solver randomly selects from $\{DLCSpolarity,\ true,\ false\}$. Otherwise, the standard DLCS polarity is used, positive when $C_p \geq C_n$, negative otherwise (Sharma et al., 2019).

Isomorphic Cache State and Variable State Aware Decaying Sum
ICSVSADS (van Bremen et al., 2021) is motivated by CSVSADS but is also symmetry-aware. When a cache hit occurs, decrement the scores of all variables in that component and also decrement the scores of variables that have previously formed a symmetric component (van Bremen et al., 2021). For example, if $(x \vee y \vee z)$ yields a cache hit and a symmetric component $(a \vee \neg b \vee c)$ was seen earlier, decrement the scores for x, y, z and for a, b, c, otherwise the heuristic follows CSVSADS (van Bremen et al., 2021).

2.8.3 Improved DFS

The algorithm explores the component tree using depth-first search (DFS) (Sang et al., 2005b). A third optimization to accelerate the detection of unsatisfiable components involves a modification to this DFS traversal (Sang et al., 2005b).

In the baseline DFS approach, the solver fully explores a child component before moving to the next sibling. This strategy can waste significant effort if a later sibling turns out to be unsatisfiable, because earlier children may be completely analysed before the decisive unsatisfiable child is discovered (Sang et al., 2005b).

The improved approach employs a lightweight exploration strategy. Rather than fully completing one child before starting the next, the solver begins exploring a child and, once the first satisfying assignment for that child is found, temporarily pauses further deep exploration and moves to the next sibling (Sang et al., 2005b). This light exploration continues until either a child is found unsatisfiable or all children are proved satisfiable (Sang et al., 2005b). If an unsatisfiable child is discovered, the parent is immediately known to be unsatisfiable and search can backtrack. If all children are found satisfiable, their explorations are completed (Sang et al., 2005b).

This improved DFS strategy reduces the risk of expending substantial effort on a child that will later be rendered irrelevant by an unsatisfiable sibling (Sang et al., 2005b).

Chapter 3

Problem Statement

In the Chapter 2, it was shown that any given component can be represented as a graph and that structurally identical components correspond to isomorphic graphs. By differentiating between non-structurally identical components encountered during the search process, redundancy is minimized, preventing the recalculation of model counts already stored in the cache.

It was argued that the overhead of computing the symmetrical scheme was justified on certain combinatorial instances. It does pay off to reduce the search space by incurring additional computational overhead. In Chapter 4, this overhead will be examined on certain problem instances.

Additionally, the symmetrical scheme leverages the canonical labelling of graphs to detect isomorphic graphs, identifying components that are structurally identical. Given that the canonical label effectively represents the adjacency list of a graph, it follows logically that storing adjacency lists for every cached component incurs significantly greater space overhead compared to traditional schemas. In Chapter 4, this overhead will be quantified in greater detail.

As such, a critical challenge arises: computing the symmetrical scheme is computationally intensive and demands significantly more cache space than traditional approaches. This raises the following question: **is it feasible to maintain the benefits of the symmetrical scheme while simultaneously reducing its computational overhead and optimizing space utilization, thereby enhancing the solver's overall cache efficiency and thus performance ?**

One way to mitigate unnecessary computations of the symmetrical scheme involves adding an additional level of indirection, i.e. implementing a multi-layer cache. As previously outlined in Chapter 2, components are cached through a mapping that associates a representation of the component with their respective model counts. If a component is cached but never retrieved again, calculating its symmetrical scheme became an inefficient use of resources. The effort is justified only when a stored

component is later found to be structurally identical to another, allowing the reuse of its precomputed model count.

The higher-level cache refrains from computing the symmetrical scheme and instead computes another encoding that is cheap but generalizes more than traditional representations. Components are only stored in the lower-level cache with their symmetrical scheme representation after a hit has occurred in the higher level cache. This mechanism tries to ensure that the symmetrical scheme is only computed for components that are more likely to be accessed in the future. Although a single hit does not guarantee repeated future access !

The introduction of a multi-layer cache system leads to several interesting research questions:

- **RQ1**: Which key representation for the higher-level cache should be used to maximize cache hit rate and minimizes average memory access time across a benchmark ?
- **RQ2**: How does a multi-level cache architecture (2, 3 levels) affect solver performance measured by PAR-2 score, compared to a single-level cache, across a benchmark?
- **RQ3**: How does a compact bitstream encoding of the symmetrical scheme affect the PAR-2 score and the number of components cached ?

Chapter 4

Approach

This chapter investigates the computational and memory overhead of different encoding schemes and presents different methods to reduce these overheads. In Section 4.1, the time and space overhead of the symmetrical scheme (SS) and the hybrid packing scheme (HCOP) is discussed. In Section 4.2, a 2-layer cache is introduced to reduce these overheads, followed by a discussion on a 3-layer cache in Section 4.3. Finally, the chapter explores graph quantization techniques in Section 4.4 to minimize the space overhead of the SS scheme.

4.1 Premises

4.1.1 Time Overhead

The symmetrical encoding is computationally expensive, as argued in previous chapters. Specifically, converting each component encountered in the search tree into a graph, then into a canonical label and finally into a hash takes time. In other words, cache access time is slower when using the symmetrical scheme.

To illustrate this overhead, the average cache access time is measured and compared across two solvers on five problem instances. The solvers are identical except for the encoding schemes they use, which are the symmetrical scheme and hybrid packing scheme.

TABLE 4.1: Average cache access time per problem and encoding type

Problem	**SS** (μs)	**HCOP** (μs)
apex5	2517.22	1.62
count15-3	1021.44	3.62
fpga10_8_sat_rcr	326.54	3.64
count21-3	126 513.66	5.61
50-12-3-q	1039.14	2.26

As shown in Table 4.1, accessing the cache takes significantly longer on average when using the symmetrical scheme. This overhead occurs consistently, regardless of the problem instance being solved. The table highlights that the hybrid packing scheme yields consistently low and stable cache access latencies, single-digit microseconds, whereas the symmetrical scheme incurs substantially higher and much more variable latencies.

These numbers indicate that the cost of canonicalisation dominate cache-access time and depends heavily on components encountered during search. By contrast, HCOP delivers low, predictable access times. Practically, this means that if the solver performs many cache lookups, SS's extreme latencies will be most pronounced on instances that generate large canonical labels.

In Figure 4.1, the latency of each cache access is measured and recorded in a histogram for both solvers. The histograms from the five problem instances are then superimposed to produce the figure. The difference is striking, the symmetrical scheme has a lot of outliers that contribute to the higher average cache access latencies in Table 4.1.

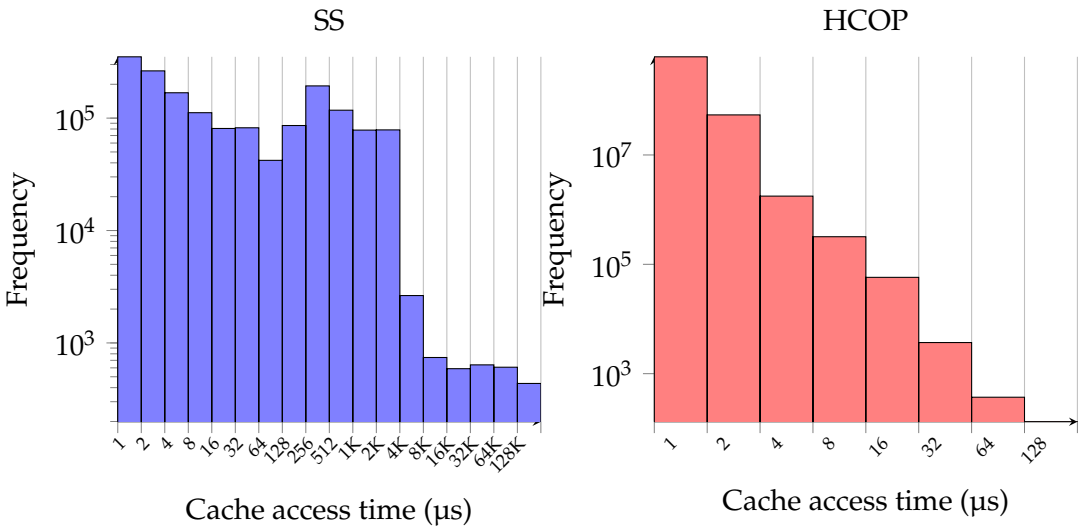


FIGURE 4.1: Histogram of cache access time across the five problem instances.

The previous results establish that cache access is indeed slow when using the SS scheme. It is therefore important to quantify how much of the total computation time is spent on cache operations. Table 4.2 reports the proportion of total solver runtime devoted to cache access. In some instances, this proportion is surprisingly high, reaching nearly 90% of the total execution time.

TABLE 4.2: Total time spent constructing the encoding and accessing the cache (TO = timed out).

Problem	**SS** (%)	**Time**$_{SS}$ (s)	**HCOP** (%)	**Time**$_{HCOP}$ (s)
apex5	99.08	135.14	1.66	32.35
count15-3	89.86	3.48	0.05	681.47
fpga10_8_sat_rcr	34.78	248.91	0.23	TO
count21-3	95.40	94.75	0.04	TO
50-12-3-q	97.07	376.44	5.71	TO

4.1.2 Space Overhead

The previous chapter established that the symmetrical scheme requires significantly more memory. Specifically, the amount of space required per component is higher under the symmetrical scheme. To illustrate this space overhead, the average memory usage per component is measured and compared between the two solvers across five problem instances.

TABLE 4.3: Size per component (mean $\pm$ std in bytes) and number of components

	SS			**HCOP**		
Problem	**Mean**	**Std**	**#Comp**	**Mean**	**Std**	**#Comp**
apex5	21400	26309	3431	235	40	4248290
count15-3	143064	146983	397	215	25	285794
fpga10-8-sat-rcr	6069	2786	243	212	9	492878
count21-3	943126	904183	959	218	28	1197842
50-12-3-q	23718	8145	4875	405	27	4620022

Table 4.3 reveals a dramatic disparity in per-component footprint and in the number of stored components. For every instance, the mean SS component size exceeds the mean HCOP component size by factors ranging from about 29 to 4,326. Also, the standard deviations for SS are extremely large in several cases (notably `count21-3`), indicating heavy-tailed size distributions driven by a few very large canonical labels.

Figure 4.2 shows the per-component size distributions for the five problem instances and confirms the heavy-tailed nature of the SS scheme. Practically, this means SS tends to store far fewer but much larger entries, while HCOP stores huge quantities of compact entries. These observations motivate multi-layer caching and the graph-quantization strategies described in Sections 4.2-4.4. Reducing the SS footprint or avoiding full SS representations unless necessary can reclaim effective cache capacity.

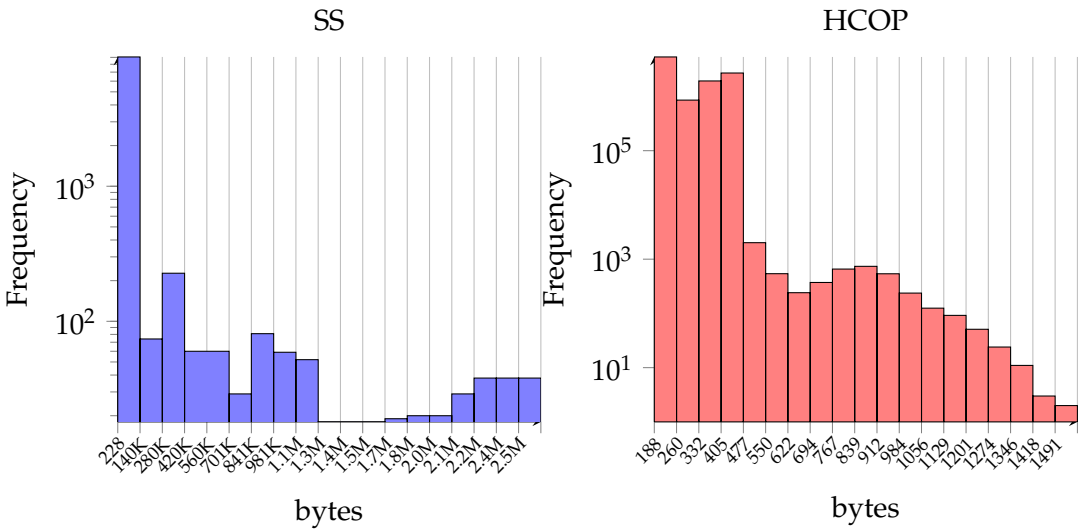


FIGURE 4.2: Histogram of per component cache size across the problem instances

4.1.3 Performance

Despite its computational and memory overhead, the symmetrical scheme (SS) can still yield performance benefits. Figure 4.3 shows a cactus plot comparing two solvers on a large benchmark suite. `GANAK` uses the HCOP scheme, whereas `SYMGANAK` uses the symmetrical scheme.

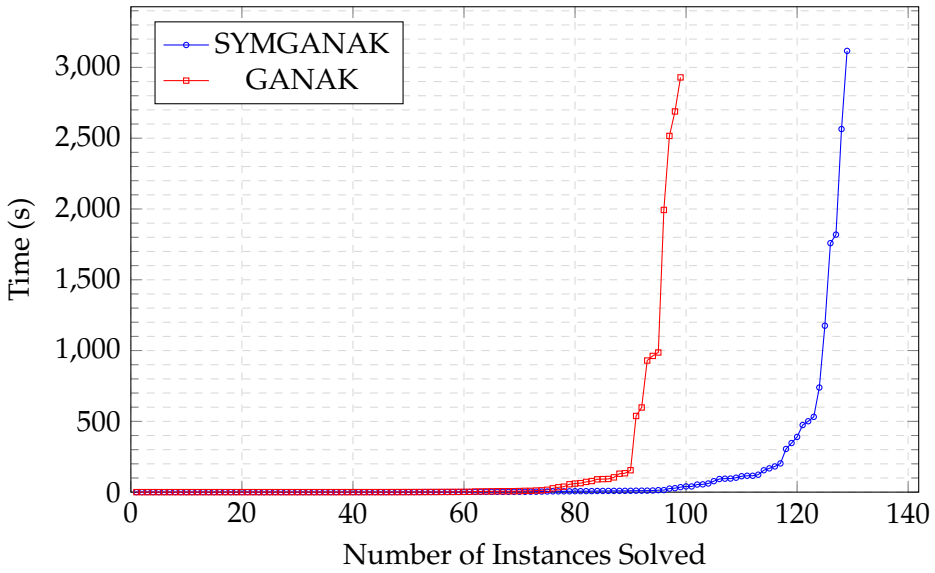


FIGURE 4.3: Cactus plot comparing the performance of different solvers.

The methodology for generating this plot is as follows: each problem p_i is solved by both solvers, and the solving time t_i is recorded (up to a time limit T). The solving times t_i are sorted in increasing order, with timeouts excluded. The points $(t_1, 1), (t_2, 2), \ldots, (t_i, k)$ are then plotted, where each point represents the k-th solved instance. Note that the same set of problems is not necessarily solved by both solvers. Therefore, the plots may include different subsets of instances (Brain et al., 2017).

From Figure 4.3, with a time budget of 3600 seconds, the solver using HCOP solves 99 problems, while the solver using SS (`SYMGANAK`) manages to solve about 129. This trend aligns closely with the results reported by van Bremen et al. (van Bremen et al., 2021). Remarkably, even with a much smaller time budget of around 100 seconds, the SS solver already outperforms the HCOP solver, demonstrating its efficiency on certain problem instances.

4.2 2-Layer Caching

As discussed in chapter 3, both time and space overhead can be improved by introducing an additional level of indirection. Up to this point, only a single cache was used, which will be referred to as the *L3 cache*. The role of the L3 cache is to store components encoded in the symmetrical scheme together with their corresponding model counts.

Input : A formula $\mathcal{F}$, An empty set of learned clauses $\mathcal{G} = \emptyset$
Output: Model count of $\mathcal{F}$

```
1  Function #DPLL(F)
2      encoding ← EncodeL2(F) ;      // HCO encoding + graph invariant
3      model_count ← LookupL2(encoding);
4      if model_count ≠ -1 then
5          return model_count;
6      end
7      pick a literal l in F;
8      x ← CountConditioned (F, l);
9      y ← CountConditioned (F, ¬l);
10     CacheInsert (encoding, x + y);
11     return x + y;
```

Algorithm 6: #DPLL algorithm with L2 / L3 caches

In the new `ISYMGANAK` solver, a higher-level *L2 cache* is introduced which is accessed before L3. Algorithm 6 shows the updated #DPLL algorithm that uses this cache. Before explaining the encoding used in the L2 cache, the following definition is required.

Definition 12 (Graph Invariant)**.** *A* graph invariant *is a function or property $f(G)$ of a graph G such that for any two isomorphic graphs G_1 and G_2,*

$$f(G_1) = f(G_2)$$

In other words, graph invariants are preserved under graph isomorphism (Diestel, 2025).

Remark. *Graph isomorphism is simply a relabelling of vertices. When two graphs are identical up to vertex names, they are isomorphic and their structural properties are preserved. This can also be shown for a property using definition 6. For example, there exists no bijection between two components that differ in their number of literals. Similarly, by definition 8, there can be no bijection between two graphs that differ in their number of nodes.*

Trivially, if two graphs differ in the number of vertices, they cannot be isomorphic. If the number of vertices do equal, no definitive conclusion can be drawn. Such properties that are cheap to compute are effective filters that allow the solver to reject many non structural identical components without performing expensive canonization!

The encoding used for L2 therefore combines a lightweight structural encoding such as HCO with a graph invariant. L2 lookups, as illustrated in Algorithm 7, first use the invariant-based hash to retrieve a small bucket of candidate entries from the L2 hash table. Only when an L2 candidate's invariant matches, is canonicalization triggered and the candidate is moved to L3 to perform the definitive check. After canonicalization the HCO encoding is removed but the graph invariant is kept. This design lets, components that are in L2, to still compare against entries in L3 that are transformed.

It is important to emphasise that components are stored exactly once in the `cache`, which is a single, contiguous array. The L2 and L3 hash tables only hold indices into that array. A bucket is formed by looking into the L2 hash table and retrieving the index of the first cache entry in that bucket's chain, and each cache entry maintains a `next_l2_bucket_element` index linking it to the next entry in the same L2 bucket. The L3 hash table is structured analogously, using `next_l3_bucket_element` index to form bucket chains.

The two hash tables differ in *what representations they index*. Entries reachable from the L2 hash table may correspond either to components stored in their HCO encoding with a graph invariant or to components that have already been transformed into the symmetrical scheme while still retaining their invariant. In contrast, entries reachable from the L3 hash table always correspond to components in the symmetrical scheme together with an invariant. Because both hash tables index the same underlying `cache` array, a single cache entry may simultaneously participate in an L2 bucket chain and an L3 bucket chain.

```
1  function LookupL2(component):
2      bucket ← component.l2_hash mod #l2_buckets;
3      idx ← l2_hash_table[bucket];
4      while idx ≠ -1 do
5          entry ← cache[idx];
6          if entry.l2_hash = component.l2_hash  and
7              entry.invariant = component.invariant then
               // add SS + remove HCO + keep invariant
8              ComputeCanonicalLabel(component);
               // add SS + remove HCO + keep invariant + add to L3 hashtable
9              MoveEntryToL3(entry);
10             model_count ← LookupL3(component);
11             if model_count ≠ -1 then
12                 return model_count;
13             end
14         end
15         idx ← entry.next_l2_bucket_element;
16     end
17     return −1;
```

Algorithm 7: Lookup a component model count in L2

Figure 4.4 illustrates the new cache structure. The fast *L2 cache*, which filters candidates using inexpensive features and the slower but definitive *L3 cache*. The following paragraphs walk through the three characteristic outcomes that can occur during a lookup so the reader can form a concrete mental model.

Case 1: equal invariants, equal canonical labels.
Consider the two components $\mathcal{H}$ and $\mathcal{K}$ from Figure 4.4. Component $\mathcal{K}$ is already present in the cache and the solver encounters component $\mathcal{H}$ in the search tree. As in Algorithm 6, the solver computes the L2 encoding of $\mathcal{H}$. The L2 encoding is the combination of the lightweight HCO scheme and the graph invariant. A hash of the graph invariant is also computed to index the L2 hash table and to do fast invariant comparisons.

It turns out that $\mathcal{H}$'s bucket contains $\mathcal{K}$ as a candidate. The lookup inspects $\mathcal{K}$'s stored invariant and compares it to the invariant of $\mathcal{H}$. If the two invariant values are equal, the lookup proceeds to the definitive stage. Per Algorithm 7, both the query component and the candidate are put through canonicalisation. The entry $\mathcal{K}$ has already undergone canonicalisation and is also part of the L3 hash table, so in this case only $\mathcal{H}$ needs to be canonicalized as denoted in Algorithm 8.

The next step is to inspect the L3 cache as shown in function `LookupL3` in Algorithm 10. Based on the hash value of the canonical label of component $\mathcal{H}$, its bucket starts

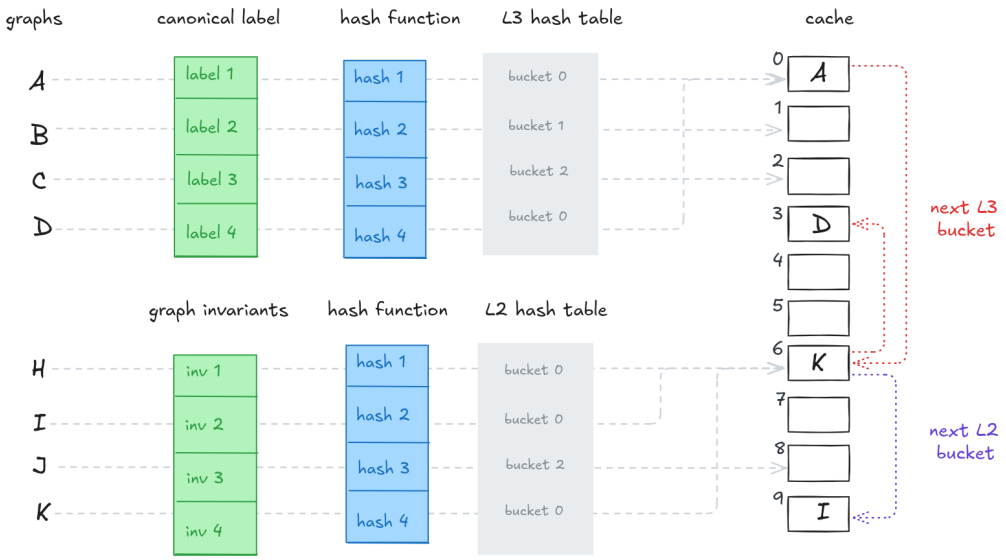


FIGURE 4.4: Two-layer hash table and chaining.

with index 0 in the cache. In that bucket, the components $\mathcal{A}$, $\mathcal{K}$ and $\mathcal{D}$ are present. The canonical labels of $\mathcal{A}$ and $\mathcal{H}$ do not match, so the next entry in the chain is considered. It turns out that the canonical label of $\mathcal{H}$ matches that of $\mathcal{K}$. An isomorphism is found and the algorithm returns the model count stored with the existing cache entry of $\mathcal{K}$. No new canonical entry is created, and no further work is required.

Case 2: equal invariants, different canonical labels.
If the L2 invariant values are equal but the canonical labels differ, the lookup has performed the heavier canonicalisation work but found that that the components are not isomorphic. The algorithm therefore continues scanning the remaining entries in the L2 bucket chain looking for another candidate whose invariant matches. It turns out $\mathcal{I}$ is also in this bucket. If no candidate in that bucket yields an invariant match, the queried component may be inserted into the L2 and L3 tables. Keep in mind that $\mathcal{H}$ is already canonicalized, that's why it also needs to be in the L3 hash table, as shown in Algorithm 9. In this case, future queries sharing its canonical form will hit directly in L3 or if future queries sharing its invariant will hit directly in L2.

Case 3: different invariants.
Consider the component $\mathcal{J}$ from Figure 4.4. The solver computes the L2 encoding of $\mathcal{J}$, which includes the HCO encoding and the graph invariant. The hash of the invariant is used to index the L2 hash table. However, the bucket selected based on this hash does not contain any candidate components (with matching invariants). Since the invariants do not match, the solver does not proceed to the canonicalization step. The solver then computes the model count for $\mathcal{J}$ and inserts it into only in L2 cache for future lookups.

```
1  function ComputeCanonicalLabel(component):
2      if not component.has_canonical then
3          component.canonical ← CanonicalLabel(component.encoding);
4          component.l3_hash ← Hash(component.canonical);
5          component.has_canonical ← true;
6      end
7  function MoveEntryToL3(entry):
8      if not entry.has_canonical then
9          entry.canonical ← CanonicalLabel(entry.encoding);
10         entry.l3_hash ← Hash(entry.canonical);
11         entry.has_canonical ← true;
12         InsertEntryIntoL3(entry);
13     end
14 function InsertEntryIntoL3(entry):
15     bucket ← entry.l3_hash mod #l3_buckets;
16     old_head ← l3_hash_table[bucket];
17     entry.next_l3_bucket_element ← old_head;
18     l3_hash_table[bucket] ← entry_idx;
```

Algorithm 8: Move cache entry from L2 to L3

As for which graph invariant to choose, there are many choices. Simple invariants include the number of vertices, number of edges and the sorted degree sequence. Slightly stronger but still inexpensive invariants include the multiset of average neighbour degrees. More computationally expensive yet more discriminative invariants include spectral signatures.

Choosing which invariants to use is a trade-off between computation cost and discriminative power: cheap invariants keep L2 queries fast but increase the workload of L3 since more false positives will occur, while stronger invariants slow down L2 slightly but reduce how often L3 must be invoked since we have less false positives. In certain cases, computing canonical labels is even cheaper than computing graph invariants, especially for small graphs.

4.3 3-Layer Caching

Some invariants are strong discriminators but too expensive to compute on every lookup. Others are cheap to compute but weak discriminators. Combining multiple invariant levels can improve the trade-off between lookup cost and discriminative power. A very cheap test (L1) quickly rejects the majority of non-matching candidates, a moderately expensive test (L2) filters the remaining candidates before

```
1  function CacheInsert(component, model_count):
       // if cache is full, apply FIFO
2      if model_count = 0 then
3          Remove all sibling subtrees of component from cache and hash_table;
4          if component is the last branched component of its parent then
5              Add (component, 0) to the cache;
6              AddToHashTables(component);
7          end
8          else
9              Remove all descendants of component from cache and hash table;
10         end
11     end
12     else
13         Add (component, model_count) to the cache;
14         AddToHashTables(component);
15     end

16 function AddToHashTables(component):
17     bucket ← component.l2_hash mod #L2_buckets;
18     old_head ← l2_hash_table[bucket];
19     cache[idx].next_l2_bucket_element ← old_head;
20     l2_hash_table[bucket] ← idx;
21     if entry.has_canonical then
22         bucket ← component.l3_hash mod #L3_buckets;
23         old_head ← l3_hash_table[bucket];
24         cache[idx].next_l3_bucket_element ← old_head;
25         l3_hash_table[bucket] ← idx;
26     end
```

Algorithm 9: Insert Model Count In Cache & L2 / L3 Hash Tables

invoking the definitive canonical check in L3. Figure 4.5 illustrates a three-layer cache that implements this idea. It uses the same principles as in the two-layer case in Section 4.2.

L1 stores components in the HCO scheme with a very cheap invariant. L2 stores components in the same HCO scheme together with a stronger property. L3 stores components in the symmetrical scheme. Because all hash tables index the same `cache` array, a single cache entry may participate simultaneously in L1, L2 and L3 bucket chains.

Algorithm 11 drives the multi-layer lookup. Given a query component, the routine computes an L1 encoding and uses the hash computed from the L1 invariant to select an L1 bucket. Each candidate entry in that bucket is compared against the

```
1  function LookupL3(component):
2      bucket ← component.l3_hash mod #l3_buckets;
3      idx ← l3_hash_table[bucket];
4      while idx ≠ -1 do
5          entry ← cache[idx];
6          if component.l3_hash = entry.l3_hash  and
7              component.canonical = entry.canonical then
8              return entry.model_count;
9          end
10         idx ← entry.next_l3_bucket_element;
11     end
12     return −1;
```

Algorithm 10: Lookup a component model count in L3

query using the cheap L1 invariant. If an L1 candidate matches, the L2 property is also computed for the query component and ensures that the cache entry also has an L2 property. If not, the cache entry is moved to L2 by inserting it into the L2 hash table and computing the L2 property. After these steps the lookup descends to `LookupL2` and follows the same overall pattern as in the two-layer case.

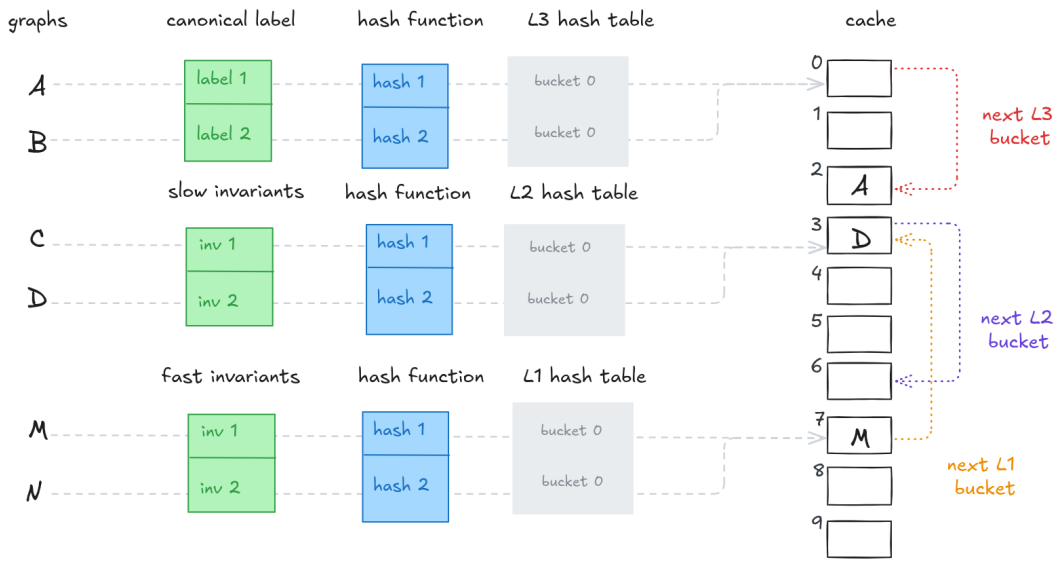


FIGURE 4.5: Three-layer hash table and chaining.

```
Input  : A formula F, An empty set of learned clauses G = ∅
Output : Model count of F
Function #DPLL(F)
    encoding ← EncodeL1(F);
    model_count ← LookupL1(encoding);
    if model_count ≠ -1 then
        return model_count;
    end
    pick a literal l in F;
    x ← CountConditioned (F, l);
    y ← CountConditioned (F, ¬l);
    CacheInsert (encoding, x + y);
    return x + y;
Function LookupL1(component)
    bucket ← component.l1_hash mod #l1_buckets;
    idx ← l1_hash_table[bucket];
    while idx ≠ -1 do
        entry ← cache[idx];
        if entry.l1_hash = component.l1_hash and entry.l1_invariant =
          component.l1_invariant then
            ComputeL2Property(component);
            MoveEntryToL2(entry);
            model_count ← LookupL2(component);
            if model_count ≠ -1 then
                return model_count;
            end
        end
        idx ← entry.next_l1_bucket_element;
    end
    return −1;
```

Algorithm 11: #DPLL algorithm with L1 / L2 / L3 caches

4.4 Graph Quantization

At the beginning of Chapter 4, the space overhead was discussed. The cache space consumed per component by each encoding scheme was reported in Table 4.3. The symmetric encoding requires substantially more space than the alternative schemes. When a multi-layer cache is used, the average space occupied by a component decreases. This effect is demonstrated below and will be examined in greater detail in Chapter 8.

Table 4.4 reports the mean component size in bytes and the number of components for each problem instance. Components encoded with the symmetric scheme, i.e. components stored in the L3 cache exhibit a substantially larger mean than components stored in the L2 cache (HCO + property).

TABLE 4.4: space per component (mean ± std in bytes)

	L3 (SS)			**L2** (HCO + property)		
Problem	**Mean**	**Std**	**#Comp**	**Mean**	**Std**	**#Comp**
apex5	12952	18913	689	7473	8028	2742
count15-3	21102	35710	30	4482	2903	367
fpga10-8-sat-rcr	6455	3312	119	2321	706	124
count21-3	173483	351919	31	928	12885	8228
50-12-3-q	30338	12787	2176	2426	9934	2699

In some cases, the mean component sizes in L3 are 2 to 5 times larger than the corresponding means in L2. In one instance the difference exceeds two orders of magnitude. The data also shows that HCO is considerably less space-efficient than HCOP (see Table 4.3), while SS occupies markedly less space than the values listed in Table 4.3, reducing the per-component footprint using a multi-layer cache.

Figure 4.6 also illustrates that components in L3 require less space per component across the problem instances. The figure also highlights a substantial reduction in aggregate L3 space usage, however, L3 still contains occasional very large components that occupy far more space than typical L2 components.

The dataset used for this initial demonstration is small. Certain problem instances may contain many highly similar graphs, which would populate L3 more rapidly and change the observed distributions. In such cases it may be instructive to investigate how reductions in L3 component footprint affect overall performance on larger or more diverse datasets.

Each component stored in L3 uses the symmetric encoding and must store its canonical label, i.e. a graph adjacency list. The approach proposed here to compactly represent the canonical label is analogous to the hybrid packing scheme (HCOP), which compactly encodes a formula using fewer bits. The proposed variant is referred to as the *Symmetrical Packing Scheme* (SPS).

4.4.1 Theoretical Bounds

First, note that decompression of the packed canonical label is unnecessary. It is not required for an isomorphism check, nor is any information extracted from the canonical label. The focus, therefore, is on lightweight packing schemes that reduce the per-component footprint enough to increase effective cache capacity while preserving fast access and comparison operations.

Secondly, in information-theoretic terms, the question is whether a lower bound exists on the number of bits required to represent a graph with n vertices and m edges without loss of information. The answer involves the concept of *Kolmogorov*

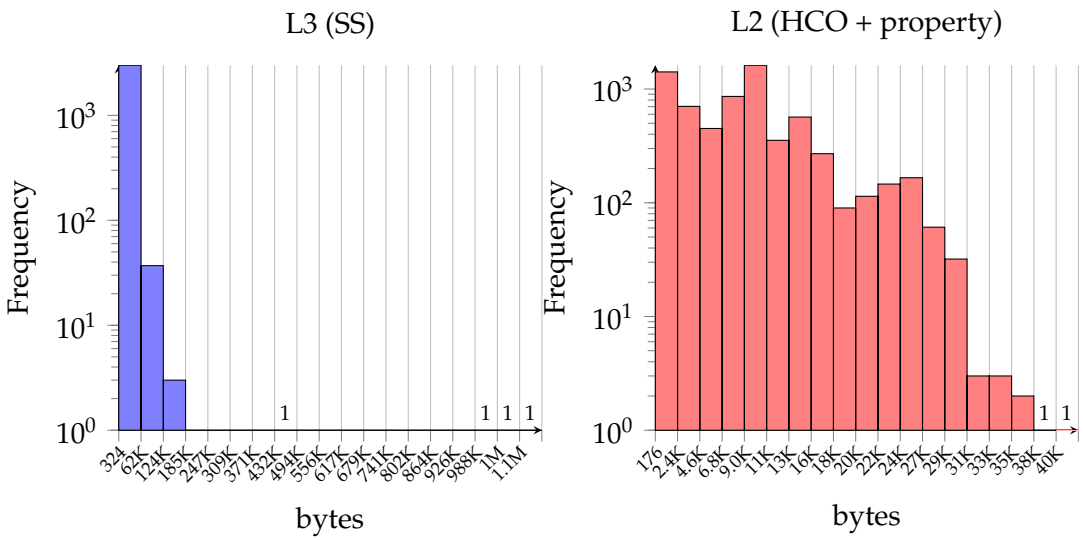


FIGURE 4.6: Memory footprint per cached component.

complexity. Kolmogorov complexity (Cover and Thomas, 2005) defines the absolute minimal description length but is uncomputable in general. Consequently, no algorithm can be guaranteed to achieve the Kolmogorov-optimal length for every input graph.

Empirical evaluation is required to quantify the trade-off: a comparison showing memory footprint, encoding time and impact on end-to-end solver runtime for representative cache sizes (e.g. 2 GB and 4 GB) would be particularly informative, such an evaluation would be presented in Chapter 8.

4.4.2 Quantization

Consider the following formula $\mathcal{F}$:

$$\begin{aligned}\mathcal{F} = &(x_1 \vee x_2 \vee x_3) \\ \wedge\ &(x_3 \vee x_4 \vee x_5) \\ \wedge\ &(\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee \neg x_4 \vee \neg x_5)\end{aligned}$$

The graph representation of this formula is visualized in Figure 4.7. This graph has 13 vertices and is represented as a `NAUTY` sparse graph in `SYMGANAK`, which uses the adjacency-list representation described in Chapter 2. An adjacency list must be constructed for each of the 13 vertices where vertex identifiers and neighbouring vertex identifiers are represented as integers. On most commonly used systems, integers require 32 bits or 4 bytes.

The amount of bytes needed to represent a simple adjacency-list layout can be expressed as the sum of the neighbour array and an index (or offset) array that

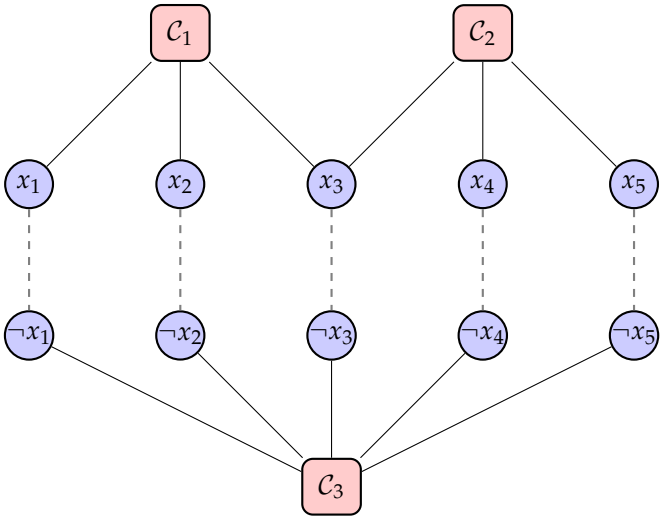


FIGURE 4.7: Graph representation of the CNF formula $\mathcal{F}$.

indicates the start position of each vertex's neighbour list in the neighbour array. For an undirected graph with n vertices and m edges, the neighbour array length equals $2m$ (each edge contributes two entries). The index array typically has n entries. If 32-bit integers are used, the total byte requirement becomes

$$\text{Bytes} \;=\; 4 \cdot (2m + n) \;=\; 4\,(2m + n). \tag{4.1}$$

Applying this to the example ($n = 13$, $m = 16$) yields

$$\text{Bytes} \;=\; 4\,(2 \cdot 16 + 13) \;=\; 4 \cdot 45 \;=\; 180\,\text{bytes}.$$

Alternative representations can be far more compact. For example, the `graph6` format (an ASCII encoding used by `NAUTY`) packs the upper-triangle adjacency matrix into printable characters. A detailed description of the `graph6` encoding scheme is provided in Appendix B. The equivalent graph representation of formula $\mathcal{F}$ in `graph6` format is `L‘?G?C??Ig?iig`, which occupies 16 bytes in memory. The ASCII-based encoding benefits from readability and portability, but it carries the overhead of representing packed bits as printable characters.

By contrast, a bit-level encoding that stores the adjacency matrix or the packed upper-triangle directly as a sequence of bits removes the ASCII-character overhead. Using such a packed bit representation for the same graph can reduce the footprint further. For example, a carefully constructed bit-packed representation requires only 9 bytes instead of 16! In the following section, the symmetrical packing scheme will be explained.

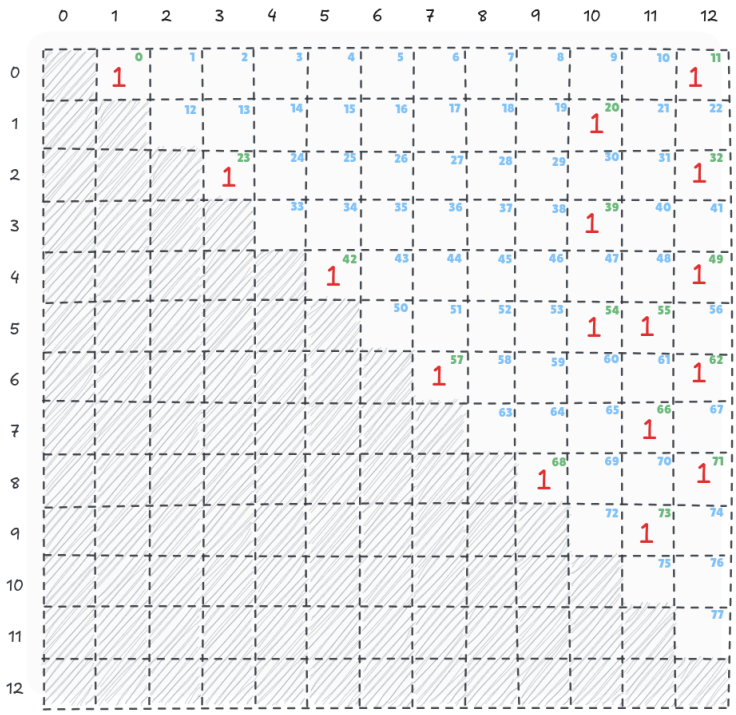


FIGURE 4.8: Adjacency matrix of a graph representation of a formula. Blue numbers in the upper-right corner of each relevant cell denote the index assigned to that position in the strict upper triangle.

4.4.3 Symmetrical Packing Scheme

Given an undirected simple graph with n vertices and m edges and assume the vertices are numbered $0, 1, \ldots, n-1$. Map each undirected edge (i, j) with $0 \leq i < j < n$ to a unique integer index in $\{0, \ldots, \binom{n}{2} - 1\}$ by enumerating the strict upper triangle of the adjacency matrix in lexicographic row-major order, as illustrated in Figure 4.8.

The resulting list of indices of the edges is sorted in increasing order and then delta-encoded by storing the first index in full and representing each subsequent index as the difference from its predecessor. Fixed bit-widths are chosen for the full index and for the deltas. All fields are concatenated and packed into a single bitstream together with a compact header that records the required metadata.

A convenient bijection $\text{idx} : (i, j) \mapsto k$ for $0 \leq i < j < n$ enumerates the strict upper triangle in lexicographic row-major order:

$$(0,1), (0,2), \ldots, (0,n-1), (1,2), \ldots, (1,n-1), (2,3), \ldots$$

The closed form for the index of pair (i, j) is

$$\mathrm{idx}(i, j; n) = \sum_{t=0}^{i-1} (n - 1 - t) \; + \; (j - i - 1) = \frac{i(2n - i - 1)}{2} + (j - i - 1). \tag{4.2}$$

Write the sorted indices as

$$a_0 < a_1 < \cdots < a_{m-1}, \qquad a_t \in \{0, \ldots, \binom{n}{2} - 1\},$$

and define deltas by

$$\delta_0 := a_0, \qquad \delta_t := a_t - a_{t-1} \quad (t \geq 1). \tag{4.3}$$

Let $B := \binom{n}{2}$. Choose bit-widths

$$b := \lfloor \log_2 B \rfloor + 1 \tag{4.4}$$

$$\Delta_{\max} := \max_{t \geq 1} \delta_t, \qquad d := \lfloor \log_2 \Delta_{\max} \rfloor + 1. \tag{4.5}$$

With these definitions the packed data bit-length equals

$$\mathrm{data\_bits} = b + (m - 1)\, d. \tag{4.6}$$

The stored bitstream begins with a compact header that encodes the necessary metadata which includes m, n and the chosen bit-widths b and d. Followed by the first full index a_0 stored in b bits and then by the sequence of $m - 1$ delta values $\delta_1, \ldots, \delta_{m-1}$, each encoded using d bits. The concatenated bitstream is laid out into fixed-size blocks of width W bits (typically $W = 32$ or 64).

If deltas are small relative to B, i.e. $d \ll b$, storing the first index once together with $m - 1$ small deltas requires fewer bits than storing m full indices. Consequently, the delta-encoded, bit-packed representation can be significantly more compact than storing each index in full.

Example 18. *Consider once again formula $\mathcal{F}$:*

$$\mathcal{F} = (x_1 \vee x_2 \vee x_3) \wedge (x_3 \vee x_4 \vee x_5) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee \neg x_4 \vee \neg x_5)$$

The vertices of the graph representation, as illustrated in Figure 4.7, are numbered as follows:

$$\neg x_1,\ x_1,\ \ldots,\ \neg x_5,\ x_5 \mapsto 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 \qquad C_1, C_2, C_3 \mapsto 10, 11, 12.$$

Assume this numbered graph is the canonical graph whose adjacency matrix is shown in Figure 4.8. In the figure, ones indicate edges and zeros are omitted for readability, block indices are highlighted in blue. The adjacency-matrix representation is used for encoding because each edge can be represented by a single integer index, whereas an adjacency-list representation typically requires two integers per vertex.

For this graph the total number of unordered vertex pairs is $B = \binom{13}{2} = 78$. *Using Equation* (4.2) *with* $n = 13$ *yields a sorted list of edge indices (upper-triangular indexing). The indices that correspond to edges are*

$$[0,\ 11,\ 20,\ 23,\ 32,\ 39,\ 42,\ 49,\ 54,\ 55,\ 57,\ 62,\ 66,\ 68,\ 71,\ 73],$$

which contains $m = 16$ *entries and therefore produces* $m - 1 = 15$ *successive differences. The deltas* $\delta_t = a_t - a_{t-1}$ *for* $t \geq 1$ *form the sequence*

$$[11,\ 9,\ 3,\ 9,\ 7,\ 3,\ 7,\ 5,\ 1,\ 2,\ 5,\ 4,\ 2,\ 3,\ 2],$$

whose maximum is $\Delta_{\max} = 11$.
The full-index bit-width is

$$b = \lfloor \log_2 B \rfloor + 1 = \lfloor \log_2 78 \rfloor + 1 = 7,$$

and the delta bit-width is

$$d = \lfloor \log_2 \Delta_{\max} \rfloor + 1 = \lfloor \log_2 11 \rfloor + 1 = 4.$$

Store the first index $a_0 = 0$ *using* $b = 7$ *bits as* `0000000`, *then encode each of the* $m - 1 = 15$ *deltas using* $d = 4$ *bits. The relevant 4-bit encodings are*

$11 \mapsto$ `1011`, $9 \mapsto$ `1001`, $7 \mapsto$ `0111`, $5 \mapsto$ `0101`, $4 \mapsto$ `0100`, $3 \mapsto$ `0011`, $2 \mapsto$ `0010`, $1 \mapsto$ `0001`.

Concatenating the initial full index and the 15 delta fields yields the bitstring

$$\underbrace{0000000}_{a_0}\ \underbrace{1011\ 1001\ 0011\ 1001\ 0111\ 0011\ 0111\ 0101\ 0001\ 0010\ 0101\ 0100\ 0010\ 0011\ 0010}_{\textit{deltas}},$$

which is then appended to the header containing the number of variables and clauses as well as the bit-widths b and d.

Chapter 5

Invariants

Chapter 4 established that the symmetrical scheme incurs substantial computational overhead, with cache access times exceeding those of traditional schemes by orders of magnitude. The proposed solution introduces a multi-layer cache architecture in which a higher-level cache uses computationally inexpensive graph invariants to filter candidates before invoking the expensive canonical labelling procedure. The viability of this approach depends on identifying invariants that strike an appropriate balance between two competing objectives: minimizing computation cost while maximizing discriminative power.

This chapter addresses research question 1 by systematically evaluating a range of graph invariants across three major categories: basic structural properties such as vertex and edge counts, neighbourhood-derived properties and centrality measures such as betweenness and closeness. The evaluation examines both individual invariants and pairwise combinations, assessing their performance across approximately 450 million graph pairs drawn from 212 problem instances.

The experimental results reveal a three-tiered performance landscape. Basic structural invariants achieve precision rates around 75 % while remaining four orders of magnitude faster than canonical labelling. Neighbourhood-based invariants reach precision rates near 97 % at computation costs approximately three orders of magnitude faster than canonical labelling. Centrality-based invariants achieve precision rates exceeding 99 % but require computation times comparable to or exceeding canonical labelling itself. These findings suggest that neighbourhood-based invariants, particularly the average neighbour degree sequence, represent the most promising candidates for the L2 cache layer, offering substantial discriminative power at acceptable computational cost.

5.1 Related Work

Dehmer et al. analysed combinations of distance-based and information-theoretic invariants on sets of non-isomorphic graphs with identical sizes (e.g., 5-node and

9-node graphs). They found that invariants effectively distinguish most graphs in such small collections. However, as the cardinality of the graph set increases, the same invariants often produce more collisions and lose discriminative power (Dehmer et al., 2013).

This suggests that for applications such as model counters, which store limited sets of components rather than exhaustively enumerating all non-isomorphic graphs of a certain size, simple invariants can still serve as fast and effective heuristics to rule out non-matches before applying full isomorphism checks.

Similarly, Araújo et al. developed an invariant-based filtering system for algebraic structures, achieving order-of-magnitude speed-ups by partitioning models into blocks using hand-crafted and randomly generated invariants (Araújo et al., 2022). While these studies address the graph isomorphism problem, underlying the detection of structurally identical components in model counting, they primarily optimize for discriminative power in batch processing scenarios rather than real-time cache queries.

Notably, these studies measure success by minimizing the number of post-partitioning isomorphism checks (Dehmer et al., 2013, Araújo et al., 2022), accepting a 20-30% runtime overhead for invariant calculation (Araújo et al., 2022). In contrast, model counting requires millions of cache lookups during search, where the overhead of invariant computation accumulates significantly. This suggests that for model counting applications, the trade-off between discriminative power and computation time may differ substantially from these related works, potentially favouring cheaper invariants that tolerate more false positives.

5.2 Methodology

The experiment uses a staged processing pipeline to assess both the discriminative power and computation cost of selected graph invariants, individually and in pairs. The solver first extracts components during search, after which each component undergoes invariant computation and canonical labelling. The results enable two forms of comparison: invariant-based predictions and label-based ground-truth validation. From these comparisons, metrics are calculated to quantify both discriminative power and computational overhead. The full workflow is illustrated in Figure 5.1. Each stage writes to an intermediate database, ensuring that progress is resumable and enabling independent inspection of every processing step.

5.2.1 Component Collection

Each component encountered by the solver is converted to a graph representation and stored. Do note that the graphs stored are not the canonical graphs. The graphs are serialized into a compact ASCII encoding, `graph6`, accompanied by a separate bit-string that represents the clause-literal colouring. Additional details on the `graph6`

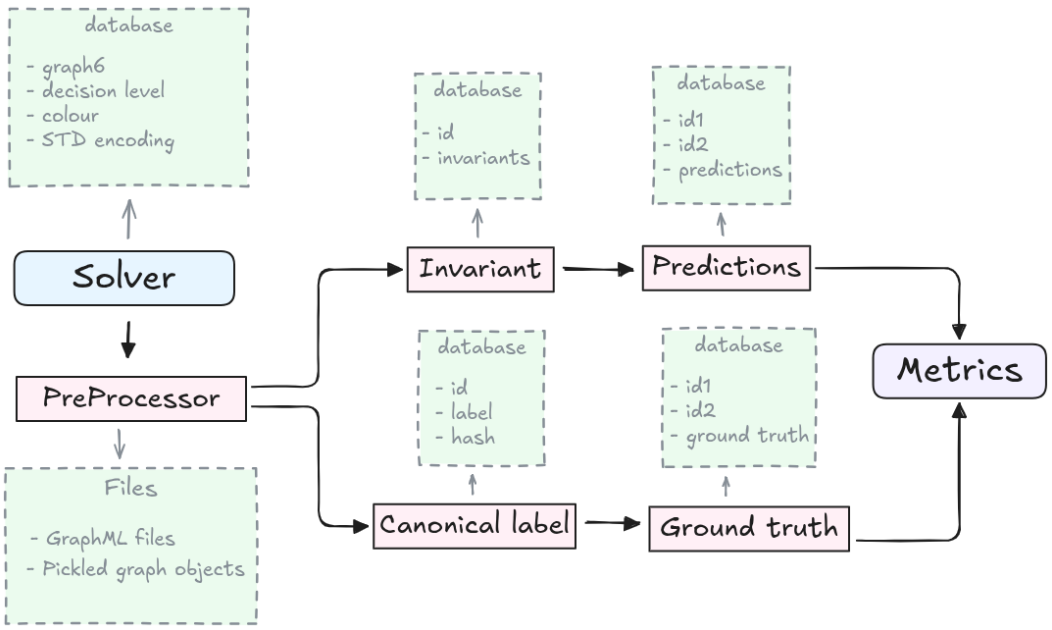


FIGURE 5.1: Staged pipeline illustrating the experimental workflow for invariant computation, canonical labelling, pairwise comparison and evaluation.

format are provided in Appendix B. These two artifacts together allow the original graph structure to be reconstructed losslessly by standard `Python` graph libraries.

All graph data used in this study are produced by the `SYMGANAK` solver that is instrumented to mine components during search. The solver is configured to use the symmetrical scheme, the VSADS variable branching heuristic and a time bound of an hour. The cache size is limited to 3,2 GB. To control storage and downstream computation for problems instances that generate many components, the component miner applies pseudo-random sampling.

A random number is drawn when a new component is encountered, if this random number is below the sampling percentage, the component is stored. Problems that generated on the order of thirty thousand components or more were assigned a sampling threshold of 0.01 while smaller problems had a sampling threshold of 1. Concretely, 120 problem instances were processed under the reduced sampling threshold of 0.01 and 92 instances were processed with no sampling.

The probabilistic sampling technique might introduce some sampling bias. To this case, a statistical test is conducted where a simple invariant distribution is compared between a sampled dataset and a dataset where no sampling was performed (see Appendix C.1 for details). The analysis confirmed that indeed there is a slight bias towards larger components in the sampled dataset.

5.2.2 Analysis

The analysis begins by filtering the collected components based on their STD encoding. This filtering step prevents overcounting isomorphic components and avoids creating the false impression that an invariant is effective simply due to the presence of repeated instances.

Next, different invariants are computed for each graph. Specifically, these include basic structural properties, neighbourhood-derived signatures, clustering & triangle statistics and classical centrality measures. All selected invariants are available in the `IGRAPH` library (Csárdi and Nepusz, 2006) and are computationally feasible, with at most polynomial-time complexity.

Some invariants produce sequences rather than scalars. In such cases, the sequences are sorted to ensure a consistent representation. Each invariant computation is timed and averaged. Note that graph construction time is excluded from these measurements. Some invariants can be computed without constructing the full graph, whereas others require the complete graph structure. The impact of this overhead is evaluated later in Chapter 7.

Afterwards, pairwise comparisons are performed to evaluate how effectively each invariant distinguishes between non-isomorphic graphs. To be precise, pairwise comparisons are processed in blocks to avoid materializing the full $\binom{n}{2}$ matrix of pairs in memory. For each block, queries fetch invariant values for the involved graphs and compare equality for each invariant. Moreover, all combinations of invariants are computed by taking logical conjunctions of single-invariant equality results and the time of the combined invariants is calculated as the sum of the average time of each individual invariant.

The canonical labelling procedure is also computed for each vertex-coloured graph to produce exact isomorphism classes. The canonical graph is serialized into a compact canonical textual representation and a cryptographic hash is computed (SHA-256) for fast comparison. Given two graphs, their isomorphism relationship can be determined by comparing their canonical labels. As mention in Chapter 2, two graphs are isomorphic if and only if their canonical labels are identical.

Each canonical labelling computation is also timed and recorded. These timings serve as a practical baseline for evaluation. An invariant that is more expensive to compute than canonical labelling itself offers little pragmatic benefit, as one could simply use canonical labelling directly for definitive results.

To evaluate, predictions are compared against the ground truth canonical labels. First, all graph pairs of interest are assembled and each pair is marked with its actual isomorphism status based on whether their canonical encodings are identical. A true positive (TP) occurs when an invariant predicts that a graph pair is isomorphic

(the invariant values are equal) and the graphs are indeed isomorphic according to the ground truth (their canonical labels match). Conversely, a false positive (FP) occurs when an invariant predicts isomorphism but the graphs are actually non-isomorphic (their canonical labels differ). Similarly, a true negative (TN) correctly identifies non-isomorphic pairs, while a false negative (FN) incorrectly predicts non-isomorphism for actually isomorphic pairs.

Finally, evaluation of predictions against ground truth proceeds by computing the classification metrics (TN, FP, FN, TP) for each invariant. This pipeline is applied to every problem instance in the benchmark, which contains 212 instances. The confusion-matrix counts are then summed across all datasets, resulting in approximately 450 million graph pairs evaluated in total. It should be noted that some overcounting may occur in these aggregated totals. Although duplicate graphs are filtered within each individual problem instance, the same graph may appear across multiple datasets.

Once the total TP and FP counts are known for each invariant across the benchmark, the precision score is computed. Precision is defined as TP / (TP + FP) and represents the proportion of predicted isomorphic pairs that are actually isomorphic. This metric is particularly fitting for measuring discriminative power because it directly quantifies how often an invariant's prediction of isomorphism is correct. An invariant with high precision produces few false positives, meaning that when it predicts two graphs are isomorphic (based on equal invariant values), this prediction is highly reliable.

5.3 Graph Invariants

The following provides formal definitions of the invariants employed in this experiment, all of which are computed for the vertex-coloured graphs constructed from the CNF formulas.

Let $G = (V, E)$ be a finite, simple and undirected graph with $|V| = n$ vertices and $|E| = m$ edges. Vertices are partitioned into two disjoint colour classes $V = V_c \cup V_\ell$, corresponding to clause-nodes and literal-nodes.

Size and Degree
The order and size of the graph are $n = |V|$ and $m = |E|$ (Diestel, 2025). The degree of a vertex $v \in V$ is $\deg(v) = |\{u \in V : (u, v) \in E\}|$. The degree sequence of G is the multiset $\{\deg(v) : v \in V\}$. Restricting to the two colour classes yields the clause-degree sequence $\{\deg(v) : v \in V_c\}$ and the literal-degree sequence $\{\deg(v) : v \in V_\ell\}$.

Average Degree
The average degree is $\frac{1}{n}\sum_{v\in V}\deg(v) = \frac{2m}{n}$, and the graph density is $\rho(G) = \frac{2m}{n(n-1)}$ corresponding to the fraction of edges present relative to the complete graph on n vertices.

Neighbourhood degree
The neighbourhood $N(v)$ for a vertex v is defined as its immediately connected neighbours as follows: $N(v) = \{u \in V : (u,v) \in E\}$. For a vertex v with neighbourhood $N(v)$, the average neighbour degree is $k_{\text{nn}}(v) = \frac{1}{|N(v)|}\sum_{u\in N(v)}\deg(u)$, defined for vertices of non-zero degree. The average neighbour degree sequence is the multiset $\{k_{\text{nn}}(v) : v \in V\}$, with analogous variants restricted to V_c and V_ℓ.

Clustering
The local clustering coefficient represents the likelihood that two neighbours of a vertex share an edge and is defined as the ratio of the number of triangles to the number of connected triples (Watts and Strogatz, 1998). The global clustering coefficient is the average $C = \frac{1}{n}\sum_{v\in V} C(v)$. (Wasserman and Faust, 1994).

Centralities
Let $d(u,v)$ denote the shortest-path distance between vertices u and v. The betweenness centrality of a vertex v is $B(v) = \sum_{\substack{s,t\in V\\ s\neq t\neq v}} \frac{\sigma_{st}(v)}{\sigma_{st}}$, where σ_{st} is the number of shortest paths between s and t and $\sigma_{st}(v)$ is the number of those paths passing through v (Freeman, 1977).

Closeness centrality quantifies how close a vertex is to all other vertices in the network. The closeness centrality of v is $C(v) = 1/\left(\sum_{u\in V\setminus\{v\}} d(v,u)\right)$. It is defined as the reciprocal of the average shortest-path distance between the vertex and every other vertex (Freeman, 1978).

Eigenvector centrality measures the importance of a vertex by assigning it a score that depends on the scores of its neighbours. Connections to highly ranked vertices contribute more to a vertex's centrality than connections to less influential ones (Csárdi and Nepusz, 2006). Let A denote the adjacency matrix of G. The eigenvector centrality vector x satisfies $Ax = \lambda_{\max}x$, where $\lambda_{\max}$ is the largest eigenvalue of A, and x_v gives the centrality of vertex v (Bonacich, 1987).

Centralization
Given a vertex-level centrality measure $f(v)$, the centralization of the graph is defined as $\text{Cent}(G,f) = \sum_{v\in V}\big(\max_{u\in V} f(u) - f(v)\big)$. Normalized versions divide this quantity by the maximum attainable value over all graphs of order n, which is achieved by the star graph (Freeman, 1978).

Triangles
A triangle is a 3-clique $\{u, v, w\} \subset V$. The triangle count per vertex is $T(v) = |\{\text{triangles containing } v\}|$ and the total triangle count is $T(G) = \frac{1}{3}\sum_{v \in V} T(v)$, since each triangle is counted once at each of its vertices (Csárdi and Nepusz, 2006).

Distances
The eccentricity of a vertex v is $\varepsilon(v) = \max_{u \in V} d(v, u)$. The eccentricity sequence is the multiset $\{\varepsilon(v) : v \in V\}$. The diameter and radius of the graph are $\text{diam}(G) = \max_{v \in V} \varepsilon(v)$, $\text{rad}(G) = \min_{v \in V} \varepsilon(v)$, respectively. The girth of G is the length of its shortest cycle, with girth infinite for acyclic graphs (Csárdi and Nepusz, 2006).

5.4 Problem Instances

The problem instances or the CNF formulas used to evaluate `SYMGANAK` are the same ones introduced by van Bremen et al. (van Bremen et al., 2021). These instances include a diverse set of benchmarks from various sources and some are generated using different methods, as detailed in the following paragraph.

- **ProbLog Problems:** These problems were created by compiling ProbLog programs. They are used to evaluate the handling of probabilistic logic (Derkinderen, 2020).
- **N-Queens Problems with Symmetry Breaking:** These problems were generated as described in *A Study of Symmetry Breaking Predicates and Model Counting* (Wang et al., 2020). They focus on symmetry breaking predicates in the context of the N-Queens problem.
- **Classic N-Queens Problems:** These instances use a classical encoding method for the N-Queens problem, where the grid is encoded as a set of Boolean variables (Shen, 2011).
- **Latin Squares:** These problems involve completing Latin squares (Gomes, 2012).
- **Grid Problems:** These problems involve grid networks and were obtained from the CRIL PMC archive (Lagniez and Marquis, 2014). These instances are directed grid networks of size $N \times N$ where each node has two outgoing edges (to the right and down). The upper-left node is the source and the bottom-right node is the sink. The benchmark query asks for the probability that the sink is true given no evidence.
- **FPGA Problems:** These instances are related to the configuration of FPGA switch-boxes (Aloul et al., 2002).
- **CNFgen Problems:** These include instances generated using the CNFgen tool (Lauria et al., 2017). They cover a range of problems such as counting principles, graph colouring and Tseitin transformations.

5.5 Results

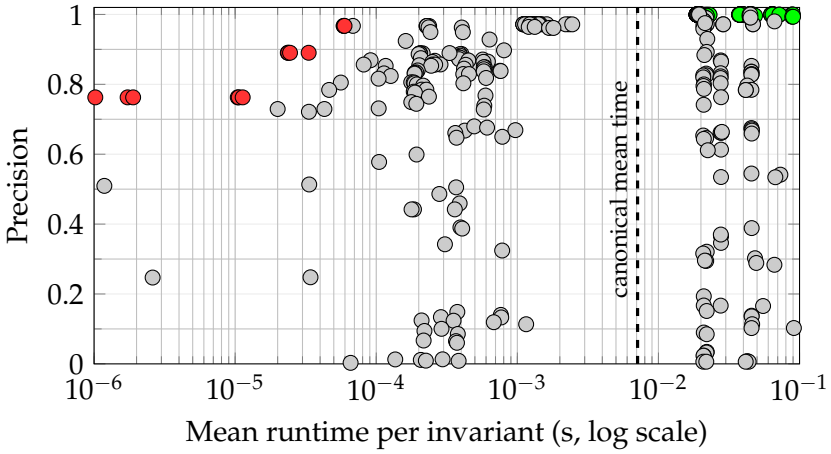


FIGURE 5.2: Precision versus mean computation time per component (seconds, log scale). Each point is an invariant (or invariant combination). The vertical dashed line marks the average runtime of canonical labelling, invariants to the left and near the top are Pareto-efficient (high precision, low cost).

Figure 5.2 presents the precision versus mean computation time for all evaluated invariants and invariant combinations across approximately 450 million graph pairs. Each point represents either a single invariant or a combination of two invariants. The vertical dashed line indicates the average runtime of canonical labelling, which serves as the baseline for comparison. Invariants positioned to the left of this line are computationally cheaper than canonical labelling, while those near the top of the plot demonstrate high discriminative power. The Pareto-efficient invariants, located in the upper-left region, achieve high precision at low computational cost.

The results reveal three distinct performance tiers. First, invariants highlighted in red represent the best-performing combinations in terms of balancing precision and computational efficiency. These include pairwise combinations such as `n_nodes` with `avg_neighbour_degree_sequence`, `n_edges` with `avg_neighbour_degree_sequence` and similar variants.

As shown in Table 5.2, these combinations achieve precision scores of approximately 96.7% while maintaining computation times roughly three orders of magnitude faster than canonical labelling, as illustrated in Table 5.1.

Second, invariants highlighted in green indicate combinations achieving precision scores exceeding 99%. These high-precision combinations, detailed in Table 5.3, include most of the time centrality-based measures such as closeness, betweenness or eigenvector centrality.

Third, simpler structural invariants such as `avg_degree`, `n_nodes` and `n_edges` achieve more modest precision scores around 72–76%, yet remain computationally inexpensive.

TABLE 5.1: Computation time comparison showing mean runtime per component for invariants from each performance tier. All times are compared against the canonical labelling baseline of 7.11 milliseconds.

Invariant	Mean Time (s)	Speed-up Factor
Canonical labelling baseline		
Canonical labelling	0.00711	1.0×
High-precision centrality-based		
n_clauses & closeness_sequence	0.01885	0.38×
n_literals & betweenness_sequence	0.04452	0.16×
Mid-tier neighbourhood-based		
avg_degree & avg_neighbour_degree_sequence	0.0000587	121×
n_edges & avg_neighbour_degree_sequence	0.0000589	121×
n_nodes & avg_neighbour_degree_sequence	0.0000596	119×
Basic structural invariants		
n_nodes & avg_degree	0.00000173	4110×
n_nodes & n_edges	0.00000189	3762×

A total of 162 invariant combinations achieve precision scores exceeding 99%. The invariant `avg_neighbour_degree_sequence` provides particularly noteworthy performance which achieves precision scores within two percentage points of the highest-performing centrality-based combinations (96.7% versus 99.9%).

TABLE 5.2: Performance metrics of select invariant combinations, showing true positives (TP), false positives (FP), true negatives (TN), false negatives (FN) and precision scores.

Index	TP	FP	TN	FN	Precision
Neighbourhood-based combinations					
avg_degree & avg_neighbour_degree_seq	1287245	43579	448616967	0	0.967
density & avg_neighbour_degree_seq	1287245	43579	448616967	0	0.967
n_edges & avg_neighbour_degree_seq	1287245	43579	448616967	0	0.967
n_nodes & avg_neighbour_degree_seq	1287245	43579	448616967	0	0.967
Basic structural combinations					
n_edges & avg_degree	1287245	400477	448260069	0	0.763
n_nodes & avg_degree	1287245	400477	448260069	0	0.763
n_nodes & n_edges	1287245	400477	448260069	0	0.763
Single invariants					
avg_degree	1287245	498234	448162312	0	0.721

These results align with the intuition presented by Dehmer et al. (Dehmer et al., 2013), who observed that invariants can be highly effective when applied to diverse collections of graphs rather than exhaustive enumerations of all non-isomorphic graphs of a fixed size. The benchmark employed in this study exhibits significant diversity, containing isomorphism classes spanning a wide range of graph sizes and structures rather than being restricted to graphs of uniform order. Under these conditions, even simple structural invariants such as neighbourhood properties demonstrate considerable discriminative power without requiring expensive centrality computations.

TABLE 5.3: Performance metrics for selected high-precision invariant combinations (precision $\geq$ 99%). All high-performing combinations include at least one centrality-based measure.

Index	TP	FP	TN	FN	Precision
Colour-centrality combinations					
n_clauses & closeness_seq	1287245	40	448660506	0	0.999969
n_literals & closeness_seq	1287245	40	448660506	0	0.999969
n_literals & betweenness_seq	1287245	43	448660503	0	0.999967
n_literals & closeness_centralization	1287245	210	448660336	0	0.999837
Centrality-based combinations					
betweenness_seq	1287245	8560	448651986	0	0.993394
betweenness_seq & triangle_counts	1287245	8560	448651986	0	0.993394
degree_sequence & betweenness_seq	1287245	8560	448651986	0	0.993394
literal_degree_seq & betweenness_seq	1287245	8560	448651986	0	0.993394
clause_degree_seq & eccentricity_seq	1287245	9285	448651261	0	0.992839

However, if the precision rate of centrality measures could be achieved at the computational cost of neighbourhood-based invariants, the resulting combination would be optimal for real-time model counting applications. This possibility motivates the use of centrality measures as variable branching heuristics in Chapter 6, which investigates whether incorporating centralities into the branching strategy can offset their computational overhead.

Chapter 6

Variable Branching Heuristic

The evaluation of graph invariants in Chapter 5 revealed that centrality-based invariants achieve precision rates exceeding 99 %, a bit higher than the 97 % achieved by neighbourhood-based measures. However, this superior discriminative power comes at a steep computational cost. The average time required to compute centrality measures exceeds the cost of canonical labelling itself, rendering these invariants impractical for cache lookup operations that must be performed millions of times during search.

This observation suggests an alternative application for centrality measures. Rather than computing them solely for cache lookups, centrality scores might serve a dual role as both variable-branching heuristics and graph invariants. If centrality-informed branching produces more balanced component splits and elevates cache hit rates, the performance gains from improved search efficiency could offset the computational cost of computing centrality measures for every component encountered during search.

The experimental design distinguishes between global and local centrality computation strategies. Global strategies compute centrality measures once on the primal graph representing the initial formula. Local strategies recompute centrality measures for each component encountered during search, providing component-specific information.

The results addresses research question 1 by demonstrating that centrality-based invariants, despite their superior discriminative power shown in Chapter 5, cannot achieve the computational efficiency required for practical cache lookup operations. Specifically, local recomputation of centrality measures on components encountered during search fails to improve performance. Even when achieving lower decision counts in some cases, the overhead of repeated centrality calculation outweighs any benefits from component-specific guidance.

However, the investigation yields a complementary positive finding. Betweenness centrality computed once on the primal graph achieves an 8% improvement in PAR-2 score over the CSVSADS baseline, solving seven additional problem instances. This result validates the hypothesis that centrality-guided branching produces more balanced component decompositions.

6.1 Related Work

Bliem and Järvisalo explored centrality-based search heuristics for SAT-based exact model counting. Their experiments employed betweenness centrality on the *primal* graph, i.e. the graph representation of the root formula, and showed empirical improvements in the model counter `sharpSAT` (Bliem and Järvisalo, 2019).

Betweenness centrality, originally formalized in social-network analysis, identifies nodes that lie on many shortest paths and therefore act as intermediaries between different parts of the graph (Freeman, 1977). Nodes with high betweenness often sit between communities. Branching on such variables can therefore lead to more balanced component decompositions, which are believed to contribute to the observed performance gains in `sharpSAT` (Bliem and Järvisalo, 2019). A concrete example of such a balanced split is shown in Appendix D.1.

Despite these promising results, the ongoing consensus in the literature is that the best variable-selection heuristic depends strongly on the problem instance (van Bremen et al., 2021). Centrality-based heuristics are a promising option but not a universal remedy.

6.2 Methodology

Bliem et al. only explored using centrality measures on the primal graph and not on the local graphs encountered during search. This was identified as future work in their paper, which is investigated here. The experimental design is structured hierarchically across five levels, as illustrated in Figure 6.1.

Level 1: Strategy
The top-level choice determines when centrality measures are computed. A *global* strategy computes centrality scores once on the primal graph. A *local* strategy recomputes centrality measures for each component encountered during search.

Level 2: Centrality Computation
After choosing the strategy, the next decision concerns which centrality measure to compute and whether to use exact or approximate computation. Two primary centrality measures are evaluated:

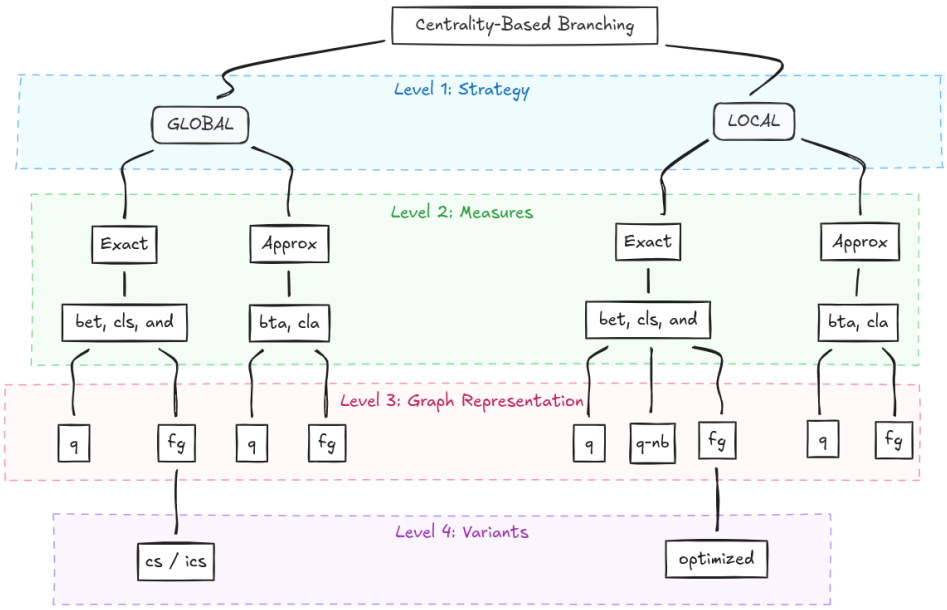


FIGURE 6.1: Hierarchical structure of the experimental design. Level 1 distinguishes global versus local branching strategies. Level 2 specifies the centrality computation method: exact (bet = betweenness, cls = closeness, and = average neighbour degree) or approximate with cut-off (bta = betweenness-approximate, cla = closeness-approximate). Level 4 defines the graph representation: fg = full graph with separate literal nodes, q = quotient graph with merged positive/ negative literals, q-nb = quotient without binary edges. Level 5 encompasses implementation variants such as cache-awareness (cs/ics), optimization such as using centrality scores as initial colouring for canonical labelling.

Betweenness centrality quantifies how often a vertex lies on shortest paths between other vertex pairs. For a vertex v, it counts the fraction of all shortest paths that pass through v. Vertices with high betweenness act as bridges between different graph regions (Brandes, 2001). Exact betweenness has time complexity $O(|V||E|)$ with V and E the vertex set and edge set of a graph.

Closeness centrality measures how easily a vertex can reach (or be reached from) all other vertices, defined as the inverse of the mean distance to all other vertices. Exact closeness requires $O(n|E|)$ time where n is the number of vertices for which centrality is computed.

To reduce computational overhead, *cut-off-based approximations* can be used. They compute range-limited variants that consider only shortest paths of length at most k (the cut-off parameter). This approximation maintains the same asymptotic worst-case complexity but can achieve substantial practical speed-ups.

Level 4: Graph Representation
Before computing centralities, the component must be converted to a graph representation. Three representations are tested, ordered from most to least precise:

Full graph (`-fg`): Each positive and negative literal receives a distinct node and clause nodes are included. Binary and longer clauses form the edges. This representation preserves complete structural information but is most expensive to construct and analyse. The graph construction follows Section 2.5.

Quotient graph (`-q`): Positive and negative literals of the same variable are merged into a single variable node. Clause nodes remain, with edges connecting variables to the clauses containing them. This reduces the vertex count by approximately half while retaining clause structure.

Quotient without binary edges (`-q-nb`): Further simplification removes binary clause edges, retaining only connections through clauses of length three or greater. The hypothesis is that omitting binary edges may reduce computation time without significant information loss for branching.

Since branching requires variable scores, centrality scores must be aggregated appropriately. For full graphs where each literal has a separate node, the centrality scores of the positive and negative literal nodes are summed to produce a single variable score. For quotient graphs, the single variable node's centrality score is used directly. In both cases, centrality scores of clause nodes are not used for branching decisions.

Level 5: Implementation Variants
The final level encompasses additional implementation choices that modify the base configurations:

Cache-aware variants (`cs`/`ics`): These incorporate cache-awareness mechanisms similar to CSVSADS or ICSVSADS, where variable cache scores (CS) are decremented when their components are cached.

$$centrality_{max} = max\{score(centrality,\ v) \mid v \in \text{unassigned variables}\} \tag{6.1}$$

$$score(CSC, v) = \begin{cases} score(CS, v), & \text{if } score(centrality, v) > r \cdot centrality_{max} \\ -\infty, & \text{otherwise} \end{cases} \tag{6.2}$$

$$score(ICSC, v) = \begin{cases} score(ICS, v), & \text{if } score(centrality, v) > r \cdot centrality_{max} \\ -\infty, & \text{otherwise} \end{cases} \tag{6.3}$$

Optimized variants: Centrality vectors may be used to provide initial colourings for nauty-based canonical labelling, potentially speeding up canonical computation by

providing a better starting partition of vertices. This optimization is particularly relevant for local strategies where both centrality and canonicalization are performed per component.

There are several suitable metrics for evaluating variable-branching heuristics. PAR-2 scores are used which are complemented with decision and conflicts count (Moskewicz et al., 2001).

6.3 Setup

All experiments were executed on the Genius compute cluster, part of the Flemish Supercomputer Center (VSC). The software environment consisted of the following modules: `Boost` version 1.85.0, `SQLite` version 3.45.3, GCCcore toolchain version 13.3.0, and `GMP` version 6.3.0. Runtime dependencies include an `igraph` shared library version 0.10.15 and `nauty` version 2.9.0.

All centrality measures are computed over the full vertex set, with `igraph` centrality normalization disabled. Approximate variants of closeness and betweenness centrality are computed using a cut-off parameter of 4. In cases where closeness-based measures produce undefined values, all `NaN` entries are replaced with -1 to ensure deterministic behaviour.

Components are processed in ascending order of size during the splitting procedure. The cache score parameter r was fixed to 0.9. The benchmark dataset consists of 212 instances and is identical to the one used in Chapter 5. Each instance was executed under the same configuration on the Genius cluster, using the specified versions of `igraph` and `nauty`.

6.4 Results

6.4.1 Global Centrality Measures

Table 6.1 presents results for solvers using centrality measures computed once on the primal graph. The baseline `ss-csvsads` solver achieves a PAR-2 score of 2894.00 seconds across 129 solved instances, with an average of 91,076 decisions and 13,376 conflicts per instance.

Several global centrality variants outperform this baseline. The best performer is `ss-glbal-bet-fg`, which computes betweenness centrality on the full graph representation and achieves a PAR-2 score of 2675.29 seconds while solving 136 instances, seven more than the baseline. This represents an 8% improvement in PAR-2 score.

TABLE 6.1: PAR-2 and descriptive stats (mean and standard deviation) for decisions and conflicts.

Solver	Inst.	PAR-2 [s]	Dec. mean	Dec. std	Conf. mean	Conf. std
hpc-csvsads	99	3910.86	1,288,926	6,123,032	10,110	37,211
ss-csvsads	129	2894.00	91,076	597,571	13,376	43,990
ss-glbal-bet-q	134	2730.43	75,513	333,058	61,581	310,035
ss-glbal-bet-fg	136	2675.29	79,933	297,136	65,679	274,157
ss-glbal-cls-q	133	2766.73	95,048	360,108	64,013	330,068
ss-glbal-cls-fg	131	2824.13	85,945	323,355	50,945	280,795
ss-glbal-and-q	117	3290.39	96,586	322,390	39,722	187,442
ss-glbal-and-fg	132	2777.24	161,875	1,221,925	41,933	268,949
ss-glbal-cla-q	102	3793.20	154,555	937,099	25,479	86,015
ss-glbal-cla-fg	99	3878.94	74,733	324,431	22,242	88,483
ss-glbal-bta-q	126	2895.33	36,238	118,586	22,546	109,433
ss-glbal-bta-fg	128	2919.37	47,576	193,052	19,679	78,579

Notably, the betweenness-based variants show substantially higher conflict counts than the baseline (approximately 62,000–66,000 versus 13,376), yet achieve better overall performance. The decision counts remain slightly lower than the baseline.

Closeness centrality variants (`ss-glbal-cls-q` and `ss-glbal-cls-fg`) also show improvements over baseline, though less pronounced than betweenness. The full-graph closeness variant solves 131 instances with a PAR-2 of 2824.13 seconds.

Average neighbour degree (`and`) and closeness-approximate (`cla`) variants generally perform worse than betweenness and closeness, with the closeness-approximate quotient variant solving only 102 instances.

The comparison between quotient graphs and full graphs reveals mixed results. For betweenness centrality, the full-graph representation marginally outperforms the quotient representation (PAR-2 of 2675.29 versus 2730.43), suggesting that the additional structural detail in full graphs justifies the modest increase in computation time. For other centrality measures, the differences are less consistent.

6.4.2 Local Centrality Measures

Table 6.2 shows the results for solvers that recompute centrality measures at each component during search. In contrast to the global approach, local centrality computation generally fails to improve upon the baseline, and in many cases performs substantially worse.

The best local variant is `ss-local-cls-q`, which matches the baseline by solving 129 instances but achieves a slightly worse PAR-2 score of 2936.49 seconds compared to the baseline's 2894.00 seconds. This variant shows significantly higher decision counts (176,573 versus 91,076 on average), indicating that the recomputed centrality scores may be leading to less balanced splits that require more branching decisions.

TABLE 6.2: PAR-2 and descriptive stats (mean and standard deviation) for decisions and conflicts for the local solver variants.

Solver	Inst.	PAR-2 [s]	Dec. mean	Dec. std	Conf. mean	Conf. std
hpc-csvsads	99	3910.86	1,288,926	6,123,032	10,110	37,211
ss-csvsads	129	2894.00	91,076	597,571	13,376	43,990
ss-local-bet-q-nb	111	3535.76	131,461	569,122	26,589	110,593
ss-local-bet-q	119	3317.46	82,068	229,777	42,270	139,767
ss-local-bet-fg	113	3497.36	66,305	192,198	33,195	112,524
ss-local-bet-fg-optimised	104	3744.98	47,200	186,985	16,125	63,287
ss-local-cls-q-nb	122	3179.94	149,076	819,029	16,173	59,832
ss-local-cls-q	129	2936.49	176,573	1,184,605	22,586	82,658
ss-local-cls-fg	124	3097.86	48,035	125,630	24,290	78,212
ss-local-cls-fg-optimised	123	3133.02	47,387	125,936	24,026	78,476
ss-local-and-q-nb	46	5670.69	1,154,929	6,672,156	8,926	36,990
ss-local-and-q	99	3897.79	215,291	1,071,474	6,401	28,780
ss-local-and-fg	79	4559.68	194,605	1,051,996	4,308	11,205
ss-local-cla-q-nb	103	3776.63	170,778	891,235	13,210	69,618
ss-local-cla-q	101	3852.75	71,406	351,710	4,592	19,844
ss-local-cla-fg	93	4126.82	43,002	172,425	24,727	146,293
ss-local-bta-q-nb	107	3664.92	136,505	507,715	38,128	158,324
ss-local-bta-q	97	3994.44	99,289	474,684	15,920	60,844
ss-local-bta-fg	90	4209.68	51,971	208,967	4,388	16,636

The worst performer is `ss-local-and-q-nb`, which solves only 46 instances with a PAR-2 score of 5670.69 seconds, nearly twice the baseline PAR-2. The average decision count for this variant exceeds 1.1 million, comparable to the slower `hpc-csvsads` baseline. The 'nb' suffix denotes variants that do not include binary clauses in the graph construction.

The failure of local centrality measures to improve performance can be attributed to computational overhead. Recomputing centrality measures for every component encountered during search is expensive, particularly for larger components. The cost of these repeated computations outweighs any benefit gained from having component- specific branching guidance. Even when local variants achieve lower decision counts (e.g., `ss-local-bet-fg` with 66,305 decisions versus baseline's 91,076), the PAR-2 scores remain worse due to the per-component computation overhead.

6.4.3 Cache-Aware Variants

Table 6.3 examines the interaction between global betweenness centrality and cache-aware branching heuristics. The baseline comparison includes three variants: VSADS, CSVSADS and ICSVSADS (the symmetry-aware extension of CSVSADS). Interestingly, all three solve 129 instances with nearly identical PAR-2 scores around 2894-2906 seconds, consistent with van Bremen's finding that cache-aware heuristics show instance-dependent rather than uniformly superior performance (van Bremen et al., 2021).

TABLE 6.3: PAR-2 and descriptive statistics (mean and standard deviation) for decisions and conflicts for the CS/ICS/global solver variants.

Solver	Inst.	PAR-2 [s]	Dec. mean	Dec. std	Conf. mean	Conf. std
ss-vsads	129	2896.68	83,241	465,303	16,026	58,596
ss-csvsads	129	2894.00	91,076	597,571	13,376	43,990
ss-icsvsads	129	2905.75	95,374	553,953	16,071	53,561
ss-glbal-bet-fg	137	2667.26	86,437	338,200	64,060	313,801
ss-cs-glbal-bet-fg	136	3234.79	77,864	286,560	65,972	275,206
ss-ics-glbal-bet-fg	134	2722.13	59,926	212,347	48,680	204,711

The pure global betweenness variant `ss-glbal-bet-fg` achieves the best overall performance in the entire experimental suite, solving 137 instances with a PAR-2 of 2667.26 seconds. This represents an 8% improvement over the CSVSADS baseline.

The cache-aware global betweenness variant `ss-cs-glbal-bet-fg` solves 136 instances but with a substantially worse PAR-2 of 3234.79 seconds, worse than both the pure global betweenness and the baseline CSVSADS.

Surprisingly, the symmetry-aware variant `ss-ics-glbal-bet-fg` performs better than the cache-aware version, solving 134 instances with a PAR-2 of 2722.13 seconds. While this is still slightly worse than the pure global betweenness approach, it is substantially better than the cache-aware variant. The ICSVSADS mechanism, which decrements scores for variables in both the cached component and previously seen symmetric components, appears to interact more favourably with betweenness centrality than the simpler CSVSADS approach. The reduced decision count (59,926 on average) and conflict count (48,680) suggest that symmetry awareness helps avoid redundant exploration.

These results demonstrate that cache awareness does not uniformly improve centrality-based heuristics and may in fact degrade performance when the mechanisms conflict. The symmetry-aware approach shows more promise but still does not improve upon pure global betweenness centrality.

6.5 Discussion

The experimental results lead to several important conclusions regarding the use of centrality measures as variable-branching heuristics for model counting.

Global centrality measures provide consistent improvements. Computing betweenness centrality once on the primal graph and using these scores throughout the search yields the best overall performance, with an 8% improvement in PAR-2 score and seven additional solved instances compared to the CSVSADS baseline. This validates Bliem et al.'s findings and demonstrates that centrality-based branching can improve component decomposition quality (Bliem and Järvisalo, 2019).

Local recomputation is prohibitively expensive. Despite the theoretical appeal of component-specific centrality scores, recomputing centrality measures during search fails to offset its computational cost. Even when local variants achieve lower decision counts, the overhead of repeated centrality calculations results in worse wall-clock performance. This answers Bliem et al.'s open question about local centrality computation: on this benchmark set, it is not practical.

Cache awareness interferes with centrality benefits. Incorporating CSVSADS-style cache awareness into global betweenness branching degrades rather than improves performance. The mechanism of decrementing scores for variables in cached components conflicts with the topological guidance provided by betweenness centrality. Symmetry awareness (ICSVSADS) mitigates this interference somewhat but still does not improve upon pure global betweenness.

Implications for multi-layer caching. The original motivation was to explore whether centrality measures could serve as graph invariants for multi-layer component caching. The experimental results indicate that centrality measures, while providing discriminating power, are too expensive to compute during search.

Chapter 7

Multi-Layer Cache

The investigations in Chapters 5 and 6 established two critical findings that inform the design of multi-layer caching systems. First, the invariant analysis revealed that combinations involving average neighbour degree sequences achieve precision scores of 96.7 %, approaching the near-perfect 99.9% percent precision of centrality-based invariants while requiring orders of magnitude less computation time. Second, the variable branching experiments demonstrated that centrality measures, despite their superior discriminative power, prove impractical for local computation due to their computational overhead. These findings suggest that lightweight neighbourhood-based invariants represent the most promising foundation for intermediate cache layers that filter candidates before invoking expensive canonical labelling.

This chapter addresses research question 2 by evaluating the practical performance of multi-layer cache architectures in the context of a complete solver implementation. The experimental evaluation examines several critical factors that influence multi-layer cache performance. The choice of invariant for L2 filtering determines both the discriminative power and the computational overhead of the intermediate layer. The quality of the hash function affects collision rates and bucket chain lengths, which directly impact lookup efficiency. The interaction between cache architecture and variable branching heuristics may produce synergistic or antagonistic effects that alter overall solver performance. By systematically varying these factors across a diverse benchmark suite of 212 problem instances, the experiments identify configurations that achieve optimal trade-offs between filtering effectiveness and computational overhead.

The results demonstrate that multi-layer caching produces substantial performance improvements when appropriately configured. The optimal two-layer architecture achieves a PAR-2 score of 2514.66 seconds with 140 solved instances, representing a 13 % improvement over the single-layer baseline's 2896.68 seconds and 129 solved instances. This configuration employs a combination of basic structural properties as the L2 invariant, paired with the cs-centrality branching heuristic and an improved

hash function. However, the experiments also reveal that performance gains depend on the quality of the hash function used in L2, with poor hash distribution negating the benefits of even highly discriminative invariants. Three-layer architectures show mixed results, with certain configurations matching two-layer performance while others degrade substantially, indicating that additional cache levels introduce complexity with no real performance gains.

7.1 Methodology

The experimental design compares multiple cache configurations across a diverse benchmark suite to determine which invariant strategies yield the best end-to-end solver PAR-2 performance. Each configuration combines the L2 cache with a specific invariant, while a baseline uses only the L3 cache. The solver runs on the same 212 benchmark instances used in Chapter 5.

The invariants under evaluation are the cheap but powerful invariants identified in Table 5.2. Additionally, some very simple invariants are also tested, these include scalar invariants such as the number of variables. This simple property requires no graph construction and can be computed directly from the component structure in negligible time. Stronger invariants require some form of constructing the graph's adjacency list.

Performance evaluation focuses on two metrics. The amount of components found in the L2 cache at termination, which measures how effectively each invariant filters candidates before canonical labelling, with higher amounts indicating better discrimination. More critically, the PAR-2 metric, captures the combined effect of all factors: invariant computation overhead, canonical labelling frequency and cache lookup efficiency.

All experiments execute on the Genius compute cluster under identical resource constraints. Each job receives a single CPU core, 8 GB of memory and a one-hour time limit. The cache size remains fixed at 3.2 GB across all configurations to isolate the effect of the invariant choice from cache capacity considerations. The solver uses either global betweenness centrality branching heuristic or the classic VSADS heuristic. Sometimes, the cache score annotated CS from CSVSADS is used in tandem with centrality scores, which is annotated as cs-centrality.

7.2 Results

7.2.1 Two-Layer Cache

Table 7.1 presents the PAR-2 scores for a two-layer cache solver using centrality-based branching. The baseline is a solver with centrality based branching but with only one layer cache. The baseline configuration solves 137 instances with a PAR-2 score of 2667.26 seconds. Several invariant choices demonstrate competitive or improved performance relative to this baseline.

TABLE 7.1: PAR-2 scores of a two-layer cache solver with centrality based branching

ID	Solver	Inst.	PAR-2 [s]
	baseline	137	2667.26
1	n_vars	136	2666.94
2	n_vars & n_long_clauses	137	2646.52
3	n_vars & n_long_clauses & n_bin_clauses	138	2615.47
4	n_edges & avg_degree	138	2611.03
5	n_nodes & avg_degree	137	2641.32
6	n_nodes & n_edges	139	2584.95
7	avg_degree	137	2658.89
8	avg_degree & avg_neighbour_degree_sequence	134	2723.68
9	density & avg_neighbour_degree_sequence	135	2704.57
10	n_edges & avg_neighbour_degree_sequence	134	2722.87
11	n_nodes & avg_neighbour_degree_sequence	135	2703.49

The results reveal that the simplest invariants show modest improvements: using n_vars alone solves 136 instances with a nearly identical PAR-2 of 2666.94 seconds. The triple combination using n_bin_clauses further improves to 2615.47 seconds with 138 solved instances, representing a 1.9% improvement over baseline.

Invariants that require adjacency list construction show strong performance. The configuration using n_nodes with n_edges achieves the best PAR-2 score of 2584.95 seconds while solving 139 instances, marking a 3.1% improvement over baseline.

The average neighbour degree sequence combinations demonstrate an unexpected outcome. Despite achieving 96.7% precision in the discriminative power analysis, these invariants underperform relative to simpler invariants. Invariant with ID 8 solves only 134 instances with a PAR-2 of 2723.68 seconds, worse than baseline. Similar degradation appears across all four neighbour degree sequence variants (IDs 8-11), with PAR-2 scores ranging from 2703.49 to 2723.68 seconds.

To isolate the performance benefits of the two-layer cache from the gains provided by the centrality heuristic, Table 7.2 presents results using the VSADS branching heuristic. This configuration reveals the pure improvement from multi-layer caching against a different baseline.

TABLE 7.2: PAR-2 scores of solver with VSADS branch heuristic

ID	Solver	Inst.	PA-R2 [s]
	baseline	129	2896.68
1	n_vars	131	2835.04
2	n_vars & n_long_clauses	134	2731.30
3	n_vars & n_long_clauses & n_bin_clauses	134	2757.57
4	n_edges & avg_degree	134	2727.42
5	n_nodes & avg_degree	135	2687.66
6	n_nodes & n_edges	135	2699.07
7	avg_degree	132	2813.22
8	avg_degree & avg_neighbour_degree_sequence	127	2984.66
9	density & avg_neighbour_degree_sequence	126	3014.38
10	n_edges & avg_neighbour_degree_sequence	128	2966.73
11	n_nodes & avg_neighbour_degree_sequence	126	3010.50

The VSADS baseline solves 129 instances with a PAR-2 of 2896.68 seconds. Under this heuristic, the relative performance of invariants shifts noticeably. The best performer is n_nodes with avg_degree (ID 5), which solves 135 instances with a PAR-2 of 2687.66 seconds, representing a 7.2% improvement. Several other invariants also show substantial gains.

Notably, the average neighbour degree sequence combinations (IDs 8-11) perform even worse under VSADS than under centrality branching. These invariants solve only 126-128 instances with PAR-2 scores exceeding 2984 seconds, representing a degradation of 3-4% relative to the VSADS baseline.

The two-layer design is intended to reduce the number of expensive L3 checks by introducing a cheaper L2 filter. Whether this helps in practice depends on two factors. First, how selective the L2 invariant is to avoid large candidate sets, and second, the hash function used to select the buckets of elements to compare in the L2 cache. If the buckets are too large, or if the hash values for many graph invariants are identical, it would create one large bucket such that each cache access requires traversing the entire L2 cache instead of a small candidate set. The following tables characterize these effects using the cache state observed at solver termination, aggregated across the benchmark suite.

Table 7.3 reports the number of cached components per cache level. Each solver variant uses the same centrality-based branching heuristic, while differing only in the invariant used for the L2 cache.

TABLE 7.3: Cache composition per cache level (mean ± std).

Solver	L3 count (mean ± std)	L2 count (mean ± std)
n_vars	46522.11 ± 117499.86	133.72 ± 302.85
avg_degree	45906.19 ± 121673.81	318.65 ± 1131.08
n_edges & avg_degree	43340.38 ± 114077.65	3808.54 ± 10725.58
avg_degree & avg_neighbour_degree_sequence	11934.62 ± 35894.90	19911.00 ± 46979.92

Table 7.4 reports the average number of variables stored in a cached component ($|var|$), again split by cache level. Again, each solver variant uses the same centrality-based branching heuristic, while differing only in the invariant used for the L2 cache.

TABLE 7.4: Average $|var|$ per cache level (mean ± std).

Solver	L3 avg $\lvert var \rvert$ (mean ± std)	L2 avg $\lvert var \rvert$ (mean ± std)
n_vars	451.85 ± 1597.52	524.63 ± 856.00
avg_degree	460.65 ± 1469.41	421.55 ± 752.15
n_edges & avg_degree	453.75 ± 1675.17	510.56 ± 1141.45
avg_degree & avg_neighbour_degree_sequence	310.32 ± 1267.81	460.38 ± 1125.59

The cache composition data reveals how different invariants affect component promotion from L2 to L3. The simplest invariant, n_vars, produces a cache with predominantly L3 components and relatively few components remaining at L2. This suggests that the `n_vars` invariant, while cheap to compute, provides insufficient discrimination. Most components must be promoted to L3 for canonical verification, resulting in minimal benefit from the two-layer architecture. The average component size is similar across both levels, at approximately 450-525 variables.

The most striking pattern appears with the `avg_degree` & `avg_neighbour_degree_seq` combination. This invariant produces 11,935 L3 components on average while maintaining only 19,911 components in L2 on average. Understanding this result requires examining how hash collisions affect cache performance.

Both cache levels are accessed through hashing. When multiple entries map to the same hash bucket, they are stored in the same cache but linked through pointers. A collision is counted whenever an entry is not the tail of its bucket list. Equivalently, a bucket chain of length k contributes $k - 1$ collisions. This metric captures the amount of pointer-chasing and per-candidate comparisons introduced by hashing, which can offset the benefits of intermediate filtering.

Table 7.5 presents collision analysis for each solver configuration, showing collision counts per cache level. The solver uses the centrality-based heuristic with the invariants mentioned in the table for the L2 cache. The data is collected from the cache state when the solver finishes.

TABLE 7.5: Collision counts per cache level (mean ± std).

Solver	L3 collisions (mean ± std)	L2 collisions (mean ± std)
n_vars	10108.70 ± 37011.64	1.07 ± 9.15
avg_degree	10713.79 ± 39812.82	11.81 ± 45.97
n_edges & avg_degree	9237.70 ± 34383.32	2622.34 ± 8546.97
avg_degree & avg_neighbour_degree_sequence	1859.96 ± 7139.35	21514.32 ± 48283.80

The collision data exposes a critical trade-off between invariant discrimination and hash distribution. The `n_vars` invariant generates approximately 10,109 collisions in L3 but only 1 collision in L2 on average. However, the minimal L2 collisions come at the cost of poor filtering, as evidenced by the low L2 component count in Table 7.3.

The `avg_degree` invariant shows similar L3 collision behaviour (10,714) but experiences an order of magnitude more L2 collisions (12). The combination of `n_edges` & `avg_degree` demonstrates the following behaviour: L3 collisions remain comparable (9,238), but L2 collisions explode to 2,622 on average. This high collision rate might explains the modest performance gains observed in Table 7.1 despite the invariant's reasonable discrimination power. The hash function struggles to distribute the combined invariant values uniformly across buckets, forcing excessive pointer traversal during L2 lookups.

The `avg_neighbour_degree_sequence` combination exhibits the most severe collision problem, with 21,514 L2 collisions on average and standard deviation of 48,284. This massive collision rate, combined with the high L2 component count (19,911), could create extremely long bucket chains that must be traversed during cache lookups. Despite the invariant's excellent 96.7% precision for detecting non-isomorphic pairs, the practical benefits could be negated by collision-induced overhead. The data suggests that the hash function fails to capture the structural diversity encoded in the neighbour degree sequences, mapping many distinct sequence values to the same buckets. Interestingly, the L3 collisions for this configuration are much lower (1,860), indicating that canonical labels might distribute more uniformly than the graph invariant values.

In short, these collision counts appear excessively high and might stem from the hash function causing too many collisions. Each graph invariant is hashed and based on this hash value, the bucket or chain for a given component is determined. If the hash function lacks sufficient diversity and causes excessive collisions, performance degrades substantially. To test this hypothesis, `xxHash` is used instead for hashing the graph invariants in L2. The hash value for L3, which is based on the canonical graph, remains unchanged. More detailed explanation can be found in Appendix D.1 regarding both hash types and the specific hash functions employed. Previously, a linear or polynomial hash was used, but the polynomial hash likely suffered from overflow issues resulting in erroneous hash values.

Table 7.6 presents the PAR-2 performance when `xxHash` replaces the original hash function for L2 invariant hashing. The solver continues to use the centrality-based heuristic with the invariants mentioned in the table for the L2 cache. The data is collected from the cache state when the solver finishes.

TABLE 7.6: PAR-2 scores of solver with centrality and xxHash.

Solver	**Inst.**	**PAR-2 [s]**
baseline with VSADS	129	2896.68
baseline with centrality	137	2667.26
n_vars	135	2709.29
avg_degree	138	2627.56
n_edges & avg_degree	135	2735.26
avg_degree & avg_neighbour_degree_sequence	137	2648.15

The introduction of `xxHash` produces notable improvements across two configurations. The baseline centrality solver achieves 137 solved instances with a PAR-2 of 2667.26 seconds, while the VSADS baseline solves 129 instances at 2896.68 seconds. With `xxHash` applied to L2 caching, several patterns emerge that differ substantially from the original hashing scheme.

The `n_vars` invariant with `xxHash` solves 135 instances with a PAR-2 of 2709.29 seconds, representing a marginal 1.6% degradation over the baseline centrality solver. The avg_degree invariant demonstrates more substantial benefits from `xxHash`, solving 138 instances with a PAR-2 of 2627.56 seconds. This represents a 1.5% improvement over baseline and notably outperforms the same invariant under the original hash function.

Most strikingly, the `avg_neighbour_degree_sequence` combination solves 137 instances with a PAR-2 of 2648.15 seconds under `xxHash`, improving slightly over the baseline performance. This represents a dramatic improvement over the PAR-2 score of 2723.68 seconds observed with the original hash function, where only 134 instances were solved. The transformation from worst-performing to competitive configuration confirms that the invariant's poor prior performance stemmed primarily from hash-induced collisions rather than fundamental algorithmic limitations.

Table 7.7 presents the collision analysis when `xxHash` is employed for L2 invariant hashing, revealing how the improved hash function affects bucket chain lengths.

TABLE 7.7: collision counts per cache level (mean $\pm$ std) (xxHash).

2 layer cache solver	**L3 collisions (mean $\pm$ std)**	**L2 collisions (mean $\pm$ std)**
n_vars	10503.22 $\pm$ 38908.65	1.06 $\pm$ 9.12
avg_degree	10299.74 $\pm$ 38399.03	11.89 $\pm$ 46.77
n_edges & avg_degree	8500.38 $\pm$ 34420.48	127.12 $\pm$ 505.83
avg_degree & avg_neighbour_degree_sequence	4602.75 $\pm$ 21304.74	4448.72 $\pm$ 15495.30

The collision data with `xxHash` reveals the mechanism behind the performance improvements. The `n_vars` and `avg_degree` invariants show minimal change in collision behaviour, with L2 collisions remaining at approximately 1 and 12 respectively. This stability confirms that these simple invariants already achieved reasonable hash distribution under the original scheme. The L3 collision counts remain similar across both hash functions, as expected, since L3 hashing is based on canonical labels and remains unchanged.

The most dramatic collision reduction occurs with the `avg_neighbour_degree_sequence` combination, where L2 collisions plummet from 21,514 to 4,449 on average. This nearly five-fold reduction in collisions explains the corresponding performance recovery observed in the PAR-2 scores.

Another possible factor limiting the gains from strong invariants is the computational overhead of adjacency list construction. The adjacency list must be constructed from every component encountered during search in order to compute the L2 property. Furthermore, each time an L2 cache entry requires conversion to L3, the edge list must be constructed once again before performing the canonical computation. Simpler invariants, such as `n_vars`, avoid this overhead and only construct the edge list once for canonical computation.

This limitation stems from using HCO encoding during search, which requires more expensive operations to retrieve the edge list. Addressing this optimization would require a significant overhaul of the codebase. During component detection, edge lists would need to be stored explicitly, and each component would need to maintain this edge list representation, which is not as space-efficient as the HCO scheme.

An unexpected finding emerges when combining cache scores with centrality measures in a two-layer cache configuration. Table 7.8 demonstrates that this combination produces notably strong results across all invariants tested. The mechanism behind this synergy remains poorly understood and merits further investigation. One hypothesis is that cs-centrality alters the solver's search path in ways that reduce problematic numerical edge cases or hash collisions for the stronger graph invariants. The modified branching behaviour may generate components that are more amenable to L2 filtering, though the specific interaction requires deeper analysis. Future work should explore whether the symmetry-aware ICS variant produces similar or superior results when combined with two-layer caching.

The cs-centrality configurations reveal a surprising interaction between cache-aware branching and multi-layer caching. The baseline with cs-centrality alone solves 136 instances with a PAR-2 of 3234.79 seconds, representing a substantial degradation compared to the standard centrality baseline (137 instances, 2667.26 seconds). The baselines are from previous Chapter 6, where cache-aware variants interfered with the benefits of betweenness-guided branching.

TABLE 7.8: PAR-2 scores of solver with cs-centrality and xxHash.

Solver	Inst.	PAR-2 [s]
baseline with VSADS	129	2896.68
baseline with centrality	137	2667.26
baseline with cs-centrality	136	3234.79
n_vars	139	2561.75
avg_degree	141	2525.65
n_edges & avg_degree	140	2514.66
avg_degree & avg_neighbour_degree_sequence	140	2516.93

When cs-centrality is paired with two-layer caching and `xxHash`, the performance transforms dramatically. The `n_vars` invariant solves 139 instances with a PAR-2 of 2561.75 seconds, representing a 4.0% improvement over the standard centrality baseline and a remarkable 21% improvement over the cs-centrality baseline.

The strongest performance comes from the `n_edges` & `avg_degree` combination, which achieves 140 instances at 2514.66 seconds. This represents a 5.7% improvement over the standard centrality baseline and a 22% improvement over the cs-centrality baseline, making it the best overall PAR-2 score.

Figure 7.1 presents a scatter plot comparing the runtime of individual instances between the `ISYMGANAK` and `SYMGANAK` solvers. The `ISYMGANAK` configuration uses a two-layer L2/L3 cache with `n_edges` & `avg_degree` as the L2 invariant, combined with cache scores and centrality-based branching. The `SYMGANAK` baseline uses only an L3 cache with VSADS branching.

The scatter plot reveals a systematic performance advantage for `ISYMGANAK`. Many points lie above the diagonal line, indicating that `ISYMGANAK` solves instances faster than `SYMGANAK`. The majority of these points fall within the shaded region corresponding to speed-ups between $1\times$ and $10\times$, demonstrating consistent but moderate improvement across the benchmark suite.

The performance advantage is particularly pronounced for FPGA problems, where `ISYMGANAK` achieves a median runtime of 4.57 seconds compared to `SYMGANAK`'s median of 175 seconds. This represents a dramatic 38-fold speed-up on this problem class, visible in the figure as a cluster of points far above the diagonal.

The `ISYMGANAK` solver times out on 8 instances compared to 14 timeouts for `SYMGANAK`. Notably, there exist instances that `SYMGANAK` solves but `ISYMGANAK` does not, visible as red points along the vertical timeout line.

To conclude, the overall performance comparison shows that `ISYMGANAK` achieves a PAR-2 score of 2514.66 seconds compared to `SYMGANAK`'s 2896.68 seconds, repre-

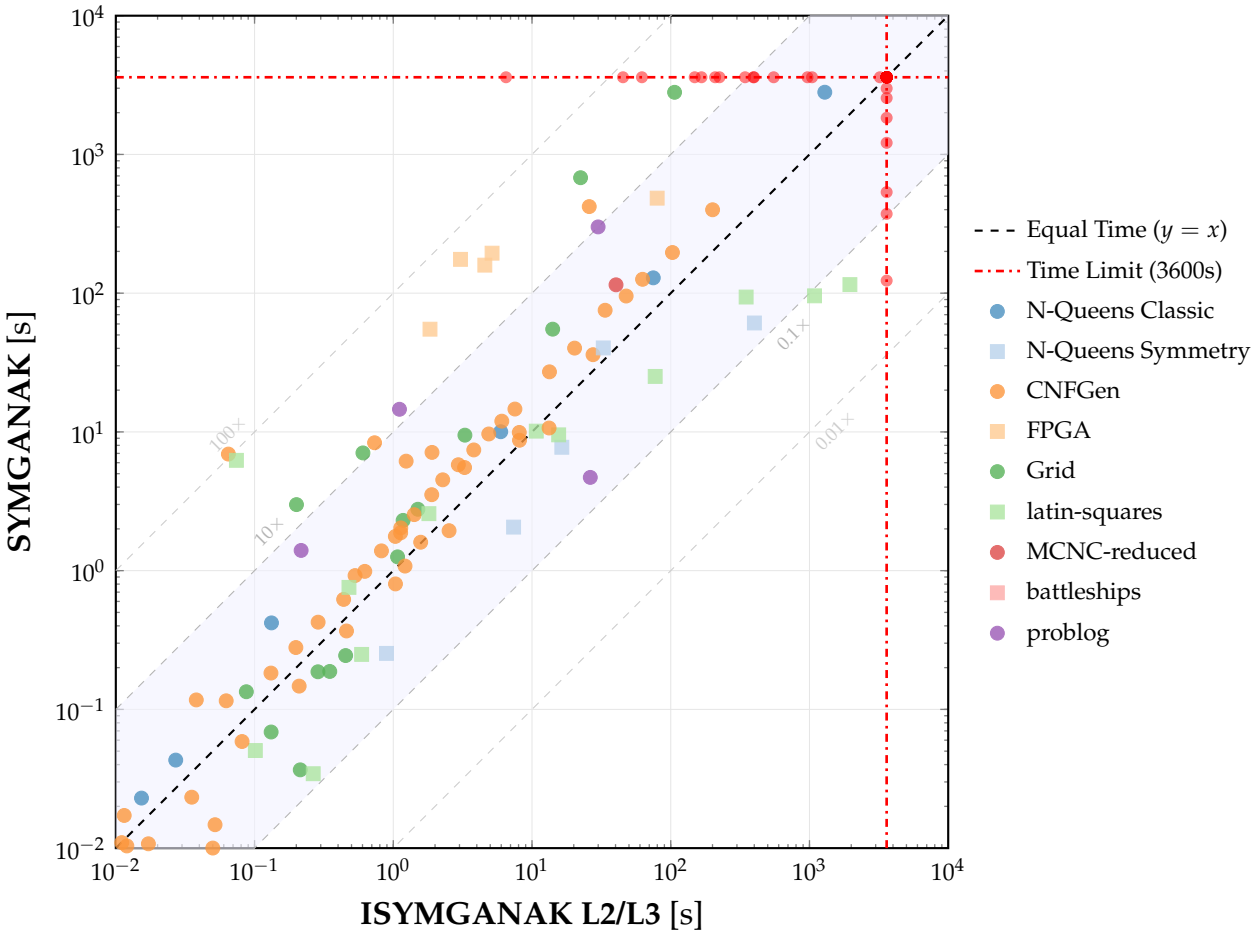


FIGURE 7.1: Per-instance runtime comparison between `ISYMGANAK` and the `SYMGANAK` baseline. `ISYMGANAK` employs two-layer L2/L3 caching with `xxHash` and the n_edges and avg_degree invariant at L2, combined with cache scores and centrality-based branching, while `SYMGANAK` uses a single L3 cache with VSADS branching.

senting a 13% improvement. Additionally, `ISYMGANAK` solves 140 instances while `SYMGANAK` solves 129 instances, demonstrating both improved efficiency and broader coverage across the benchmark suite.

7.2.2 Three-Layer Cache

Simple scalar invariants such as `n_vars` have minimal computation overhead, but provide insufficient discrimination, allowing many non-isomorphic components to reach the L3 canonical labelling stage. Conversely, stronger graph-based invariants such as `avg_neighbour_degree_sequence` offer superior discrimination power, but incur substantial overhead from adjacency list construction.

A three-layer cache architecture offers a potential solution to this trade-off by implementing a cascading filter strategy. As discussed in Section 4.3, the L1 cache layer can employ an ultra-lightweight scalar invariant that requires no graph construction whatsoever. This L1 filter quickly rejects obviously non-matching components with negligible computational cost. Only components that pass the L1 filter proceed to L2, where a stronger graph-based invariant provides more refined discrimination. Components that survive both L1 and L2 filtering finally undergo canonical labelling at L3. This graduated approach aims to minimize the number of components that require edge list construction while maintaining effective discrimination throughout the cache hierarchy.

The experimental design evaluates two three-layer configurations to test this hypothesis. Both configurations use cs-centrality branching and `xxHash` for L2 hashing to ensure consistency with the best-performing two-layer setup. The baseline for comparison is the optimal two-layer configuration identified previously: `n_edges` & `avg_degree` at L2 with cs-centrality and `xxHash`, which achieved 140 solved instances with a PAR-2 of 2514.66 seconds.

Table 7.9 includes the three one-layer baselines (VSADS, centrality, and cs-centrality), the three two-layer baselines that demonstrated strong performance with cs-centrality branching and two three-layer cache configurations.

TABLE 7.9: PAR-2 scores and solved instances across one-layer, two-layer and three-layer cache architectures using cs-centrality branching and `xxHash`. All configurations maintain identical solver settings except for invariant selection.

Cache	L1 invariant	L2 invariant	Inst.	PAR-2 [s]
One-layer cache (L3)				
L3 baseline with VSADS	–	–	129	2896.68
L3 baseline with centralities	–	–	137	2667.26
L3 baseline with cs-centralities	–	–	136	3234.79
Two-layer cache (L2/L3)				
L2/L3 baseline	–	n_vars	139	2561.75
L2/L3 baseline	–	n_edges & avg_deg	140	2514.66
L2/L3 baseline	–	avg_deg & avg_n_deg_seq	140	2516.93
Three-layer cache (L1/L2/L3)				
L1/L2/L3	n_vars	n_edges & avg_deg	125	3010.24
L1/L2/L3	n_vars & n_b_cls & n_l_cls	n_edges & avg_deg	115	3366.03
L1/L2/L3	n_vars	avg_deg & avg_n_deg_seq	126	2988.90
L1/L2/L3	n_vars & n_b_cls & n_l_cls	avg_deg & avg_n_deg_seq	139	2542.25

The three-layer cache results in Table 7.9 reveal that additional cache layers generally fail to improve upon the two-layer architecture. Most three-layer configurations exhibit substantial performance degradation, with the worst variant solving only 115 instances at a PAR-2 of 3366.03 seconds. The majority of three-layer configurations achieve PAR-2 scores between 2988.90 and 3010.24 seconds while solving between 115 and 126 instances, representing significant regressions compared to the two-layer

baseline of 2514.66 seconds with 140 solved instances. One three-layer configuration using `n_vars` & `n_bin_clauses` & `n_long_clauses` at L1 combined with `avg_degree` & `avg_neighbour_degree_sequence` at L2 achieves competitive performance with 139 solved instances and a PAR-2 of 2542.25 seconds, though this still falls short of the best two-layer configuration.

The disappointing performance of the three-layer configuration lacks a clear explanation. The most likely cause is that the L1 layer fails to provide sufficient filtering, which results in unnecessary computational overhead when constructing adjacency lists at L2 and possibly once again at L3. Future research should investigate whether the performance degradation stems from inadequate L1 discrimination and whether the overhead of adjacency list construction can be minimized.

To conclude, the optimal configuration identified through these experiments employs a two-layer L2/L3 cache architecture with `n_edges` & `avg_degree` as the L2 invariant, cs-centrality branching and `xxHash` for L2 hashing. This configuration, designated `ISYMGANAK`, achieves a PAR-2 score of 2514.66 seconds with 140 solved instances, representing a 13% improvement over the `SYMGANAK` baseline. Figure 7.2 demonstrates with a cactus plot the performance advantage of `ISYMGANAK` across the benchmark suite. The cactus plot shows `ISYMGANAK` solving 140 instances compared to 129 for `SYMGANAK` and 99 for the `GANAK` baseline.

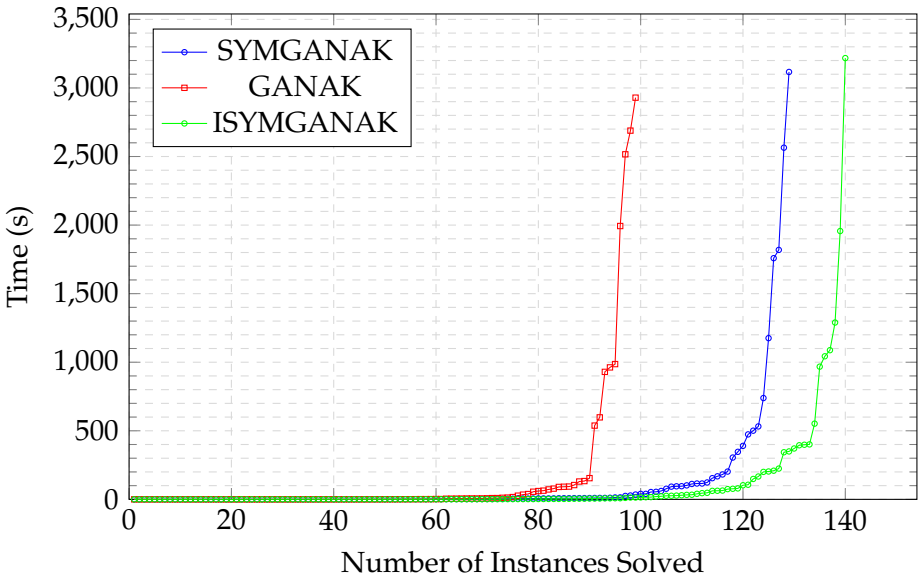


FIGURE 7.2: Cactus plot comparing the performance of different solvers.

Chapter 8

Graph Quantization

In Section 4.1, several benchmark instances were observed to produce very large canonical labels. Even with an adjacency list representation, these graphs can consume substantial memory once cached. Section 4.4 introduced a compact representation intended to reduce the per-component memory footprint, thereby increasing the effective cache capacity and lowering eviction pressure.

This chapter addresses research question 3 by investigating whether compact bitstream encodings of canonical labels can reduce the space overhead of the symmetrical scheme resulting in better cache utilisation, and thus better performance.

The cache composition results in Chapter 7 already suggest that strong L2 invariants reduce the number of components reaching L3. For example, Table 7.3 showed that the strongest L2 invariant maintains an L3 cache with approximately 10,000 components on average, compared to over 45,000 for simpler invariants. At the same time, Table 7.4 indicated that the L3 components observed with a strong L2 invariant are not exceptionally large on average, measuring roughly 310 ± 1,200 variables. This suggests that the L2 cache already filters many large components, potentially reducing the headroom for additional memory savings through graph compression. However, the high standard deviation indicates substantial variability across instances, with some problems generating much larger components that could benefit from compact encoding.

The experimental design evaluates this hypothesis by comparing solver performance across packed and unpacked canonical label representations at two different cache sizes. The unpacked baseline employs the standard adjacency list representation used throughout previous experiments, while the packed variant applies the Symmetrical Packing Scheme to compress canonical labels in the L3 cache. Both configurations use identical solver settings, including the L2 invariant (`avg_degree` & `avg_neighbour_degree_sequence`), cs-centrality branching and `xxHash` for L2 hashing.

The results reveal that graph packing consistently degrades rather than improves solver performance. At 2 GB cache capacity, the packed variant solves three fewer instances and achieves a 3.1 % worse PAR-2 score compared to the unpacked baseline. At 4 GB cache capacity, the degradation persists with two fewer solved instances and 2.6 % worse PAR-2 score. These findings indicate that the overhead of encoding packed representations outweighs any benefits from reduced memory consumption under current cache configurations. The analysis suggests that multi-layer filtering has already addressed the memory pressure that graph quantization was designed to alleviate.

8.1 Methodology

The analysis consists of two complementary investigations. The first investigation measures cache statistics at solver termination to assess the remaining optimization potential for graph quantization. For each benchmark instance, three key metrics are recorded: the total memory footprint occupied by cached components measured in bytes, the number of components stored in the cache and the average size of individual cache entries in bytes along with their standard deviation.

These per-instance statistics are subsequently aggregated and averaged across the entire benchmark suite to yield representative values for each cache configuration. Two configurations are examined: a baseline employing a single-layer L3 cache paired with VSADS branching, and the best-performing multi-layer configuration which consists of a two-layer architecture with `n_edges` & `avg_degree` as the L2 invariant, cache-aware centrality branching (cs-centrality) and `xxHash` for L2 hashing. For the multi-layered cache, metrics are collected independently for each layer.

The objective is to determine whether L3 components in the multi-layered cache still occupy substantial space compared to the single-layered cache. If significant memory consumption persists at the L3 level despite the additional filtering layers, this indicates viable opportunities for applying graph quantization techniques to further reduce memory overhead.

The second investigation directly tests solver performance through PAR-2 scores by evaluating the impact of graph quantization at different cache capacity limits. The experiment compares packed and unpacked graph representations at two cache capacities: 2 GB and 4 GB. To maintain consistent memory availability for non-cache operations, instances with 2 GB cache allocation receive 4 GB total RAM on the Genius cluster, while instances with 4 GB cache allocation receive 6 GB total RAM.

The unpacked baseline stores canonical labels using the standard adjacency list representation employed throughout the previous experiments. The packed variant applies the Symmetrical Packing Scheme to compress canonical labels in the L3 cache. Both variants use the same L2 invariant (`avg_degree` & `avg_neighbour_degree_sequence`) and cs-centrality branching to ensure that performance differences reflect only the impact of graph compression.

8.2 Results

Table 8.1 presents cache occupancy statistics for the VSADS baseline configuration, which employs a single-layer L3 cache only.

TABLE 8.1: Cache occupancy statistics for single-layer cache with VSADS branching.

Metric	**Mean**	**Std**
Total cache (KB)	127,464	219,892
L3 entries	101,018	211,683
L3 per entry (KB)	55.83	130.42

This baseline maintains an average of 101,018 components in the L3 cache, with each component occupying approximately 55.83 KB on average. The total cache occupancy averages 127 MB but exhibits substantial variability across instances. Table 8.2 presents comparable statistics for the best multi-layer configuration, using `n_edges` and `avg_degree` as the L2 invariant with cs-centrality branching and `xxHash`.

TABLE 8.2: Cache occupancy statistics for two-layer cache.

Metric	**Mean**	**Std**
Total cache (KB)	42,887	127,360
L2 Cache (HCO + invariant)		
L2 entries	3,919	11,229
L2 per entry (KB)	7.49	16.25
L3 Cache (SS + canonical label)		
L3 entries	43,673	111,934
L3 per entry (KB)	50.84	129.55

The multi-layer configuration achieves a substantial reduction in total cache occupancy, averaging 42.9 MB compared to 127.5 MB for the baseline. This represents a 66% reduction in memory consumption. The reduction stems primarily from filtering: only 43,673 components reach L3 on average (compared to 101,018 in the baseline), while 3,919 components remain in L2.

The per-component L2 footprint averages only 7.49 KB, approximately seven times smaller than L3 components. This confirms that the HCO encoding with invariant requires substantially less space than the symmetrical scheme with canonical labels. Interestingly, the average L3 component size in the multi-layer configuration (50.84 KB) is only slightly smaller than in the baseline (55.83 KB), suggesting that the L2 filter does not strongly bias toward smaller components. Rather, the filter rejects components based on invariant matching regardless of size.

Despite the substantial reduction in total cache occupancy, the presence of 43,673 L3 entries consuming approximately seven times more space per entry than L2 components suggests potential room for optimisation through graph quantization. Table 8.3 compares PAR-2 performance across packed and unpacked graph representations at two cache capacities. All configurations use the L2 invariant avg_degree and avg_neighbour_degree_sequence with cs-centrality branching.

TABLE 8.3: PAR-2 scores comparing unpacked (standard) and packed (SPS) graph representations at 2 GB and 4 GB cache capacities. All configurations use L2 invariant `avg_degree` & `avg_neighbour_degree_sequence` with cs-centrality branching.

Configuration	**Inst.**	**PAR-2 [s]**
2 GB Cache Capacity		
Unpacked graph (baseline)	139	2555.70
Packed graph (SPS)	136	2637.42
4 GB Cache Capacity		
Unpacked graph (baseline)	139	2530.50
Packed graph (SPS)	137	2599.00

The results reveal a consistent pattern across both cache capacities: graph packing degrades rather than improves solver performance. At 2 GB cache capacity, the unpacked baseline solves 139 instances with a PAR-2 of 2555.70 seconds, while the packed variant solves only 136 instances with a PAR-2 of 2637.42 seconds. This represents a 3.1% degradation in PAR-2 and a loss of three solved instances. The pattern persists at 4 GB cache capacity, where the unpacked configuration achieves 139 instances at 2530.50 seconds compared to 137 instances at 2599.00 seconds for the packed variant, marking a 2.6% degradation.

The performance degradation occurs despite the theoretical memory savings offered by the Symmetrical Packing Scheme. This finding suggests that the overhead of encoding packed representations appears to outweigh the memory savings.

Chapter 9

Conclusion

9.1 Conclusion

This thesis investigated whether the benefits of the symmetrical scheme for component caching in model counting could be preserved while reducing its substantial computational and memory overhead. The symmetrical scheme detects structurally identical components through canonical graph labelling, enabling sophisticated cache reuse that traditional schemes miss. However, the cost of computing canonical labels for every cache access introduces severe performance penalties, with cache access times exceeding traditional schemes by several orders of magnitude.

The multi-layer cache architecture proposed and evaluated in this work addresses this fundamental tension by introducing intermediate layers that use lightweight graph invariants to reject non-matching components before invoking expensive canonical labelling. The experimental evaluation systematically addressed three research questions examining invariant selection, cache architecture design and memory optimization strategies across a diverse benchmark suite of 212 problem instances.

The first research question investigated which graph invariants provide optimal trade-offs between discriminative power and computational cost for higher-level cache filtering. The analysis of approximately 450 million graph pairs revealed three distinct performance tiers. Simple structural invariants such as vertex and edge counts achieved modest precision rates around 75 percent. Neighbourhood-based invariants combining basic properties with average neighbour degree sequences achieved 96.7 percent precision at computation costs approximately one hundred times lower than canonical labelling. Centrality-based invariants achieved precision exceeding 99 percent but required computation times comparable to or exceeding canonical labelling itself, rendering them impractical for cache lookup operations performed millions of times during search.

The second research question evaluated how multi-layer cache architectures affect solver performance compared to single-layer caching. The optimal two-layer configuration achieved a PAR-2 score of 2514.66 seconds while solving 140 instances, representing a 13 % improvement and eleven additional solved instances compared to the single-layer baseline of 2896.68 seconds and 129 instances. This configuration employed basic structural properties combined with average degree as the L2 invariant, paired with cache-aware centrality branching and an improved hash function implementing the `xxHash` algorithm.

The evaluation of three-layer cache architectures produced mixed results, with certain configurations matching but not exceeding two-layer performance. The configuration using only simple invariants at L1 showed substantial degradation, solving only 125 instances with a PAR-2 of 3010.24 seconds. However, a carefully tuned three-layer design achieved 139 instances at 2542.25 seconds, approaching the optimal two-layer performance.

The third research question examined whether compact bitstream encoding of canonical labels could reduce space overhead and improve performance through increased cache capacity utilization. The Symmetrical Packing Scheme applied delta encoding and bit-level packing to canonical labels, but when evaluated in the context of the complete multi-layer solver architecture, it consistently degraded rather than improved performance across both 2 GB and 4 GB cache capacities. The failure of graph quantization stems from the effectiveness of multi-layer filtering at addressing memory pressure.

9.2 Future Work

9.2.1 Construction of Graph Structure

To detect disjoint components in the search tree, variables are traced from the super component and disjoint sets are created. From these disjoint sets, components are formed and encoded using the HCO encoding. This approach was chosen because HCO representations require minimal space, an important consideration given that the search tree can become quite large. However, this design introduces inefficiencies when components must be cached.

When a component requires caching in the symmetrical scheme, the HCO encoding must be converted into a graph structure to construct the canonical label. Similarly, components stored in the lightweight HCO encoding at the L2 level still require edge list construction when computing graph invariants such as average neighbour degree. If such a component is subsequently promoted to the L3 cache, the edge list must be reconstructed again to compute the canonical label. This duplicated work represents a significant inefficiency in the current implementation.

The system would benefit from encoding components in the search stack using a format that enables rapid edge list construction. Alternatively, if memory requirements can be accommodated, storing edge lists directly would eliminate the need for recomputation.

9.2.2 Caching Before BCP

The symmetrical scheme is used in order to find structurally identical components that have the same model count. An important point is that only components with a given model count are cached. This principal can be extended to include also unsatisfiable components **but in a different cache**.

In the second chapter, in section 2.7.3, the argument was made that components from unsatisfiable subtrees are not included in the cache. Those components may have model counts that are undercounted because they have been polluted by learned clauses. But, due to pruning, components that caused the subtree to become unsatisfiable are never recorded. It would be beneficial if before Boolean constraint propagation is performed, the current component was not found in a cache containing unsatisfiable components.

Example 19. *Consider the following CNF formula $\mathcal{F}$:*

$$\begin{aligned}\mathcal{F} =&(x_1 \vee x_2 \vee x_3)\\ &\wedge (x_1 \vee \neg x_2 \vee x_3)\\ &\wedge (x_1 \vee x_2 \vee \neg x_3)\\ &\wedge (x_1 \vee \neg x_2 \vee \neg x_3)\\ &\wedge (\neg x_1 \vee x_4 \vee x_5)\\ &\wedge (\neg x_1 \vee \neg x_4 \vee x_5)\\ &\wedge (\neg x_1 \vee x_4 \vee \neg x_5)\\ &\wedge (\neg x_1 \vee \neg x_4 \vee \neg x_5).\end{aligned}$$

In practice, this formula produces only a single cached component, namely the full formula, which is undesirable. Assume the solver first decides on the literal x_1. This creates two branches: $\{x_1 = true\}$ and $\{x_1 = false\}$. In the branch $\{x_1 = true\}$, the remaining component is:

$$\begin{aligned}&(x_2 \vee x_3)\\ \wedge\ &(\neg x_2 \vee x_3)\\ \wedge\ &(x_2 \vee \neg x_3)\\ \wedge\ &(\neg x_2 \vee \neg x_3).\end{aligned}$$

In the branch $\{x_1 = false\}$, the remaining component is:

$$
\begin{aligned}
&(x_4 \vee x_5) \\
\wedge\ &(\neg x_4 \vee x_5) \\
\wedge\ &(x_4 \vee \neg x_5) \\
\wedge\ &(\neg x_4 \vee \neg x_5).
\end{aligned}
$$

Once the first branch is proven `UNSAT`*, the second branch must also be* `UNSAT` *by symmetry, yet the solver fails to exploit this redundancy and repeats the same work due to cache pollution management, see Algorithm 5.*

9.2.3 Symmetric Clause Learning

Take the previous formula $\mathcal{F}$. The solver first branches on the literal x_1. In the branch $\{x_1 = \text{true}\}$ a conflict may yield the clause $(x_1 \vee x_2 \vee x_3)$. From this conflict, the learned clause $\neg x_1$ can be derived, i.e. the solver concludes that x_1 must be false.

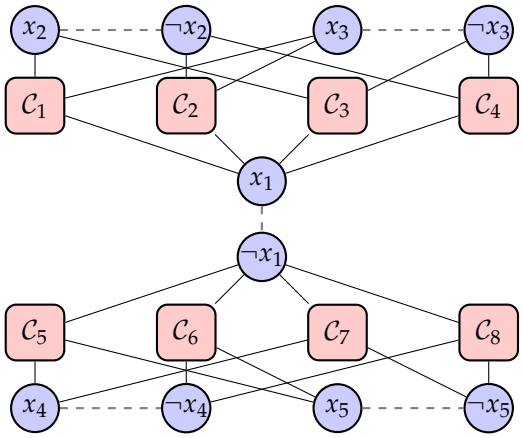


FIGURE 9.1: Graph representation $Gr(\mathcal{F})$ of the CNF formula $\mathcal{F}$.

Furthermore, it can visually be seen that x_1 and $\neg x_1$ belong to the same automorphism orbit. This means that one can replace the node x_1 with the node $\neg x_1$ and the resulting graph remains isomorphic, i.e. it belongs to the same automorphism group.
Since x_1 and $\neg x_1$ belong to the same automorphism orbit of the primal graph, the symmetric counterpart of the learned clause would be $\neg x_1 \wedge x_1$, which is unsatisfiable. In that case computation of the other branch is avoided.

This symmetry-based technique has been studied extensively in SAT. Variants such as *Symmetry Propagation* (Devriendt et al., 2012) and *Symmetric Explanation Learning* (Devriendt et al., 2017) can be adapted to #SAT solvers since they preserve the model count.

9.2.4 Cache-Aware Neural Variable Branching

CSVSADS and ICSVSADS are variable branching heuristics that try to include the cache state when considering branching. They try to bias the search away from re-exploring components that have recently been solved, by penalizing variables that frequently appear in cached components or in their symmetric counterparts. The underlying assumption is that exploring new, syntactically distinct components can help expose more opportunities for decomposition and avoid repeatedly reconstructing the same subproblems.

An alternative approach is explicitly aiming to maximize the reuse of cached components. The central idea is to treat the score of a candidate branching variable as an estimate of the utility of the *residual formula* that would result if that variable were chosen. Intuitively, a variable should be preferred if choosing it is likely to produce a residual component that already exists (or closely matches) an entry in the component cache, and thus yields a cache hit that saves expensive recomputation.

To make the proposal concrete, consider a tiny illustrative scenario. Suppose the cache currently contains a component represented by the two clauses on the left, while the component encountered at decision time consists of the clauses on the right.

Cache	**Root of Current Search Tree**
$(\neg D \vee E)$	$(A \vee \neg B \vee C)$
$\wedge\, (D \vee \neg E)$	$\wedge\, (B \vee \neg C)$

If the solver chooses the assignment $\{A = False\}$, the residual formula would be isomorphic to the cached component. Branching on A therefore produces an almost immediate cache hit, making it clearly the optimal choice. While this may not matter much in a small example, the advantage becomes obvious if the components were larger. Moreover, centrality-driven choices can spectacularly fail to exploit available cache hits in such cases. CSVSADS can also fail if its cache-penalization mechanism pushes the solver away from precisely those assignments that would produce immediate, large cache hits. This doesn't mean the previous heuristics are wrong or inferior, rather they should be applied at different times.

The only solution capable of generalizing to specific contexts is a neural network. Reinforcement learning is particularly well-suited for this task. The RL agent learns through sequential decision-making which strategies to prioritize in different contexts, automatically discovering when to favour cache hits over centrality or when to heed learned clauses, all by maximizing the reward signal.

All this to say, this isn't a novel idea. There has been some effort in this direction. Vaezipoor et al. introduced *Neuro#*, which uses graph neural networks (GNNs) and reinforcement learning to learn variable-branching policies for model counting

(Vaezipoor et al., 2021). In their experiments *Neuro#* achieved large reductions in the number of branching steps and, on several hard problem families, produced net wall-clock speed-ups relative to traditional heuristics (measured with `GANAK`) despite the runtime overhead incurred by model queries.

The authors nevertheless acknowledge that inference latency is a practical bottleneck in some settings and suggest that query overhead could be reduced by optimizations such as GPU acceleration and tighter integration (e.g., loading the model into the solver's C++ code instead of making out-of-process Python calls).

However, *Neuro#* primarily focuses on the structural representation of the formula. A significant extension beyond this work involves adding the *cache state* or the learned clause database into the neural architecture. This could be achieved through embeddings of cached components and computing a 'distance to cache' metric. By incorporating these signals, the network can learn when to prioritize centrality and when to pursue branches likely to lead toward existing cache entries.

$$score(NN,\ v) = NN(current\ component,\ centrality\ values,\ cache\ state,\ learned\ clauses\ database,\ v)$$

Another interesting avenue, inspired by *AlphaZero* (Silver et al., 2017), involves using Monte Carlo Tree Search and a model to predict future cache states based on a series of future variable assignments to guide the variable selection. Although the query overhead would be too high, it is still interesting to see how small the search tree and effective the cache can become.

Appendices

Appendix A

Measuring Methods

A.1 Measuring Average Access Time

Table 4.1 was shown illustrating the average cache access time across five benchmark problem instances. The following explains how these measurements were taken.

Average access times were measured by attaching user-level probes to the two functions of interest, `ComponentCache::ManageNewComponent` and `ComponentManager::ConvertComponent`, using `bpftrace`. To ensure that each probe captured the complete execution of the target function, both functions were compiled with the `noinline` attribute, preventing inlining by the compiler and guaranteeing the presence of stable symbols in the binary. Symbol locations were verified in advance using `nm -C`.

For each function, a paired `uprobe/uretprobe` was installed. On function entry, a high-resolution timestamp (`nsecs`) was recorded. On function return, the elapsed execution time was computed as the difference between the return timestamp and the corresponding entry timestamp. Per-call durations were accumulated in global counters storing the total elapsed time and the number of calls and were additionally recorded in histograms to capture the distribution of execution times.

Tracing was performed by launching the solver under `bpftrace`, ensuring that the full execution of each benchmark instance was measured. Each benchmark problem was executed independently, and for a given problem the average access time was computed as the ratio of the total accumulated execution time to the total number of observed calls. The values reported in Chapter 4, Section 4.1 correspond to these per-problem averages.

All measurements were conducted under identical build conditions to minimize external sources of variability. The default settings of `SYMGANAK` were used in all cases, except for the encoding.

A.2 Measuring Total Proportion of Cache Access Time

Table 4.2 was shown illustrating the fraction of each benchmark's runtime that was spent performing the cache-access operations. The values reported there were obtained as follows.

The data presented was obtained using Valgrind's `callgrind` profiling tool, which records executed-instruction counts and attributes them to functions. These counts can be used as a proxy for the relative time spent in different parts of the program.

The raw profiling data was collected in `callgrind` output files and subsequently analysed using `KCachegrind` for visualization.

A.3 Measuring memory usage per component

Table 4.3 reports the average and standard deviaton of memory usage per cached component across some benchmark instances. These metrics were obtained by measuring the size of every cached component when the solver is finished. The statistics (count, mean and standard deviation) are computed online using Welford's algorithm (Welford, 1962) to avoid numeric instability.

Figure 4.2 shows a histogram of the memory usage of cached components across several benchmark instances. This data was collected at the end of each run by measuring the memory usage of every cached component and assigning the measurements to a 30-bin histogram. To obtain an aggregated histogram across multiple benchmark instances, the per-instance histograms are combined by summing the bin counts.

Analogously, Table 4.4 and Figure 4.6 present results for both L2 and L3 caches, using a single encoding scheme per cache. L3 uses the SS encoding, while L2 uses the HCO encoding together with an additional graph property (average degree and average neighbour degree).

Appendix B

Encoding and Hashing

B.1 Graph6 Encoding

`graph6` (McKay, 2022) is a compact ASCII-based format for representing simple undirected graphs. Each graph is written using only printable ASCII characters in the numeric range 63–126. The graph encoding is a concatenation of two parts: `N(n) R(x)` where n is the number of vertices and x is the bitstring formed from the upper-triangle of the adjacency matrix (McKay, 2022). In what follows, the `graph6` encoding will be explained by applying it to formula $\mathcal{F}$.

$$\mathcal{F} = (x_1 \vee x_2 \vee x_3) \wedge (x_3 \vee x_4 \vee x_5) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee \neg x_4 \vee \neg x_5)$$

Example 20. *To derive the* `graph6` *encoding for the given formula, start by identifying the nodes and edges in the graph representation of the formula. Given the graph representation, which is the same as in Figure 4.7, the adjacency matrix with* $n = 13$ *vertices, the upper-triangle edge positions are*

$$\{0, 11, 20, 23, 32, 39, 42, 49, 54, 55, 57, 62, 66, 68, 71, 73\},$$

which corresponds with Figure 4.8. The number of upper-triangle bits is $B = \binom{13}{2} = 78$. *The first step is to build the 78-bit upper-triangle vector* x, *which sets bit* i *to 1 if and only if position* i *is an edge. The resulting 78-bit string is*

```
100000000001 00000000100 1000000001 000000100 10000001 0000110 100001 00010 1001 010 00 0
```

The following step is to split this long bitstring into 6-bit groups and form $R(x)$. *Also, if the total bitstring is not divisible by 6, then the bitstring is padded to be a multiple of 6. In this case, the bitstring is 78 bits long and divisible by 6. The result will be 13 groups of 6 bits as follows:*

group index	*6-bit group*	*value*
1	100000	32
2	000001	1
3	000000	0
4	001001	9
5	000000	0
6	001000	8
7	000100	4
8	100000	32
9	010000	16
10	110100	52
11	001000	8
12	101001	41
13	010000	16

Each 6-bit value v is encoded as a single byte by adding 63 to it. Since a 6-bit number can represent values from 0 to 63, the largest possible encoded value is $63 + 63 = 126$. Because a byte can represent values up to 255, this encoding never overflows. Consequently, every resulting byte lies in the range 63–126, which corresponds exactly to printable ASCII characters. This ensures that the entire encoding consists of readable symbols.

$$\begin{aligned}
32 + 63 &= 95\ (\,'_'\,)\\
1 + 63 &= 64\ (\,'@'\,)\\
0 + 63 &= 63\ (\,'?'\,)\\
9 + 63 &= 72\ (\,'H'\,)\\
0 + 63 &= 63\ (\,'?'\,)\\
8 + 63 &= 71\ (\,'G'\,)\\
4 + 63 &= 67\ (\,'C'\,)\\
32 + 63 &= 95\ (\,'_'\,)\\
16 + 63 &= 79\ (\,'O'\,)\\
52 + 63 &= 115\ (\,'s'\,)\\
8 + 63 &= 71\ (\,'G'\,)\\
41 + 63 &= 104\ (\,'h'\,)\\
16 + 63 &= 79\ (\,'O'\,)
\end{aligned}$$

Hence

$$R(x) = \texttt{\_@?H?GC\_OsGhO}.$$

The final step is to encode the number of vertices n. For values $0 \le n \le 62$, the `graph6` *format represents n using a single byte with numeric value $63 + n$. In this case, $n = 13$, so the encoded value is $63 + 13 = 76$, which corresponds to the ASCII character* `L`*. Thus, $N(13) =$* `L`*. The complete* `graph6` *representation is obtained by concatenating the encoded vertex count $N(n)$ with the remaining data $R(x)$.*

$$\texttt{L\_@?H?GC\_OsGhO}.$$

B.2 Hash Functions for Graph Invariants

The L2 cache encoding requires efficient computation of hash values from graph invariants. The choice of hash function impacts both cache lookup speed and the potential for collisions, which can make chains in the cache longer which increases lookup time, see Figure 4.4 or Figure 4.5. Therefore, the quality of the hash distribution can affect performance. This section discusses the hash functions used for computing hash keys from graph properties.

B.2.1 Linear Hash

When graph invariants consist of scalar values such as variable counts and clause counts, a simple linear combination hash function has been used:

$$h = \sum_{i=1}^{n} c_i \cdot v_i$$

where v_i are the invariant values and c_i are constant coefficients. The implementation typically uses small integers such as 1 or 2 for these coefficients. For example, a property combining the number of variables and clauses computes its hash as:

$$h = 2 \cdot n_{vars} + n_{clauses}$$

B.2.2 Polynomial Rolling Hash

Graph invariants containing arbitrary numbers of scalar values, such as degree sequences, require a different approach. A polynomial rolling hash operates on sequences of values by iteratively applying the recurrence relation:

$$h_{i+1} = h_i \cdot c + v_i$$

where c is a constant multiplier, v_i is the i-th value in the sequence, and h_0 is initialized with a seed value. For the graph invariants consisting of a scalar value and a vector such as the average degree and degree sequence combination, the hash was computed as:

```
1 function PolynomialHash(invariant_constant, invariant_sequence):
2 |   hashkey ← invariant_constant;
3 |   for each value v in invariant_sequence do
4 |   |   hashkey ← (hashkey × 3) + v;
5 |   end
6 |   return hashkey;
```

Algorithm 12: Polynomial rolling hash for graph invariants

The multiplier constant of 3 was chosen for its simplicity and computational efficiency. Polynomial hashing is straightforward to implement and requires minimal

computational overhead, each iteration involves only one multiplication and one addition operation.

However, polynomial hashing with small constant multipliers can suffer from several limitations. The hash values exhibit poor avalanche properties, meaning that small changes in input values do not sufficiently disperse the resulting hash values across the hash space. This can increase the collision rate in hash tables, which can lead to longer bucket chains and degraded lookup performance.

B.2.3 xxHash

`xxHash` (Collet, 2018), which stands for extremely fast hash, uses a non-cryptographic hash function designed by Yann Collet [1]. `xxHash` prioritizes speed and distribution quality over cryptographic security, making it well-suited for hash table applications.

`xxHash` subdivides input data into multiple independent streams that are processed in parallel (Collet, 2018). This approach minimizes data dependencies and allows the algorithm to approach the theoretical RAM bandwidth limit (Collet, 2018). For hashing graph invariants, Algorithm 13 depicts the procedure for graph invariants that are a combination of a sequence and a scalar value.

function *GetHashkey(invariant_constant, invariant_sequence)*:
    $n \leftarrow$ sizeof(invariant_sequence);
    v_hash $\leftarrow$ `XXH3_64bits(`*invariant_sequence*, $n \times$ *sizeof*(*double*)`)`;
    total_hash $\leftarrow$ `XXH3_64bits_withSeed(`*invariant_constant*, *sizeof(double)*, *v_hash*`)`;
    return (total_hash $\oplus$ (total_hash $\gg$ 32));

Algorithm 13: XXH3 hashing for graph invariants (sequence + scalar).

B.2.4 Canonical Graph Hashing for L3

While the L2 and L1 cache use `xxHash` or the polynomial hash to process hash graph invariants, the L3 cache operates on canonical graph representations and requires a different hashing strategy. The L3 hash function must compute hash values directly from the graph structure in its canonical form, ensuring that isomorphic graphs produce identical hash values after canonicalization (McKay, 2024).

The L3 hash is computed using the `hashgraph_sg` function from the `NAUTY` library, which operates on sparse graph representations. This function is specifically designed to hash graph structures producing consistent results.

[1] https://github.com/Cyan4973/xxHash

The graph hashing algorithm processes the sparse graph representation by examining the degree sequence and edge lists of all vertices. The computation maintains an accumulator that is updated based on the structural properties of each vertex:

```
1  function HashGraphSG(sg, key):
2      accum ← sg.num_vertices;
3      for i ← 0 to sg.num_vertices − 1 do
4          if sg.degree[i] = 0 then
5              accum ← accum + FUZZ1(i);
6          else
7              accum ← (accum ≫ 7) | ((accum ≪ 24) & 0x7FFFFFFF);
8              val ← ListHash (sg.edges[i], sg.degree[i], key);
9              val ← (val + i) & 0x7FFFFFFF;
10             accum ← accum + FUZZ2(val);
11         end
12     end
13     return accum & 0x7FFFFFFF;
```

Algorithm 14: Canonical graph hashing for L3 cache

The algorithm distinguishes between isolated vertices (degree zero) and connected vertices. For isolated vertices, the accumulator is updated by applying a fuzz function to the vertex index. For connected vertices, the algorithm performs a circular bit rotation on the accumulator before processing the vertex's edge list. The rotation operation $(accum \gg 7) \mid ((accum \ll 24)$ & 0x7FFFFFFF$)$ shifts the accumulator right by 7 bits while wrapping the discarded bits to the high-order positions. The edge list of each vertex is hashed using the `listhash` subroutine, which computes a hash value from a set of integers:

```
1  function ListHash(edges, count, key):
2      accum ← count;
3      lkey ← key & 0x7FFFFFFF;
4      for i ← 0 to count − 1 do
5          val ← edges[i] & 0x7FFFFFFF;
6          val ← (val + lkey) & 0x7FFFFFFF;
7          accum ← accum + FUZZ1(val);
8      end
9      return accum & 0x7FFFFFFF;
```

Algorithm 15: Hash function for edge lists

The `listhash` function initializes its accumulator with the edge count, then iterates through the edge array, which contains the neighbour vertices of a particular vertex. Each neighbour vertex is masked to 31 bits, combined with `lkey`, and passed through

the FUZZ1 function before being added to the accumulator. Two sets of constants are defined:

$$\text{fuzz1} = \{1984625421, 971524688, 1175081625, 377165387\}$$
$$\text{fuzz2} = \{2001381726, 1615243355, 191176436, 1212176501\}$$

The FUZZ1 and FUZZ2 operations are defined as:

$$\text{FUZZ1}(x) = x \oplus \text{fuzz1}[x \bmod 4]$$
$$\text{FUZZ2}(x) = x \oplus \text{fuzz2}[x \bmod 4]$$

These operations XOR the input value with a constant selected based on the two least significant bits of the input.

Appendix C

Sampling Bias

C.1 Sampling Bias in Collecting Components

This analysis is restricted to the first six minutes of the solver's execution to ensure computational feasibility. The first test aggregates information per instance by computing the median of the n_nodes[1] invariant values in the full and in the sampled dataset and then applies a paired Wilcoxon signed-rank test to those per-instance medians. This nonparametric paired test checks whether the distribution of $\text{median}_{full} - \text{median}_{sample}$ is symmetric about zero.

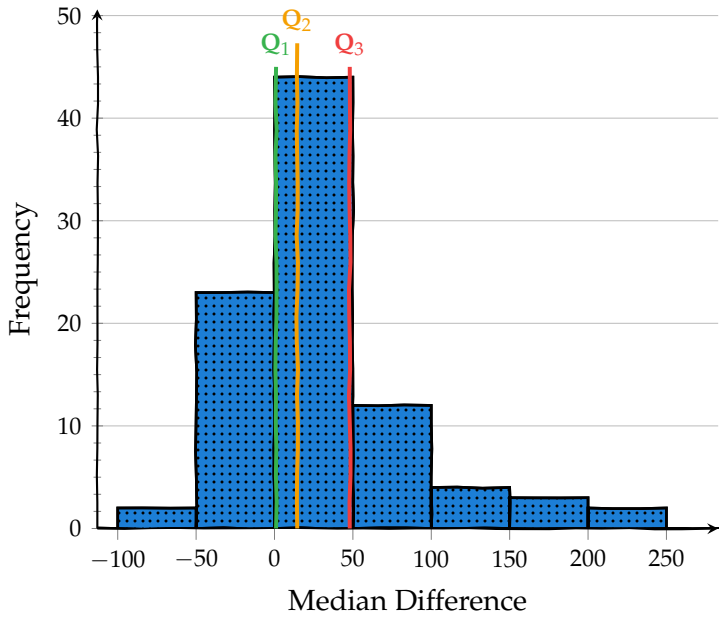


FIGURE C.1: Distribution of median differences ($Q_1 = 1, Q_2 = 14.5, Q_3 = 48$)

The second test is at the instance level: for every instance the code runs a two-sample Kolmogorov-Smirnov (KS) test comparing the empirical n_nodes distributions from

[1] number of nodes of the graph representation of a component

full vs sampled data. Per-instance p-values are corrected for multiple comparisons using Benjamini–Hochberg (FDR) procedure and the number of rejections is illustrated in Figure C.2.

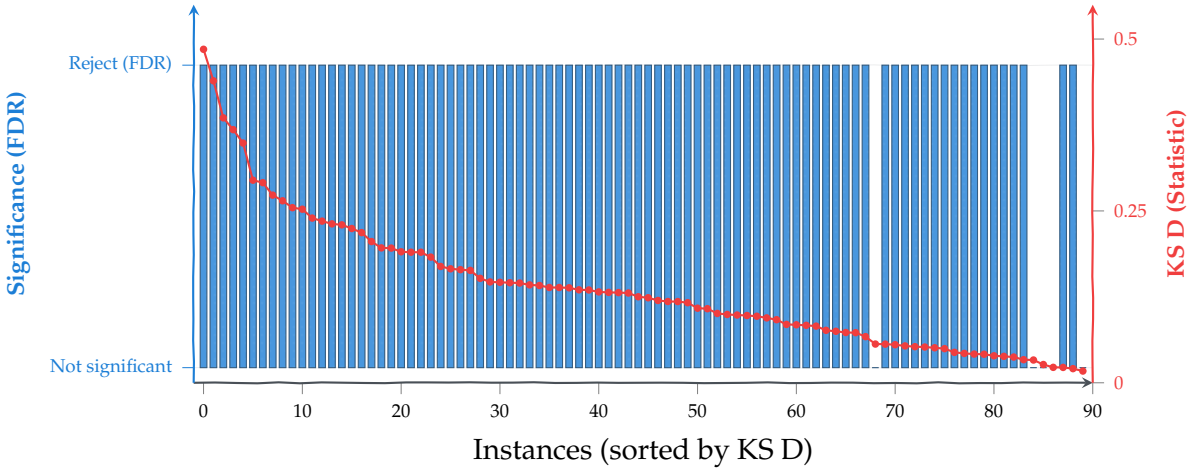


FIGURE C.2: Significance (FDR) and KS *D* statistics across instances, 85 out of 90 instances reject null hypothesis after FDR

Appendix D

Branching

D.1 Centrality and Balanced Search Trees

This appendix compares variable-branching choices and explains how centrality measures can yield a more balanced search tree. There is also an interesting coincidence with branching on a central node when the graph representation is symmetrical around the central node.

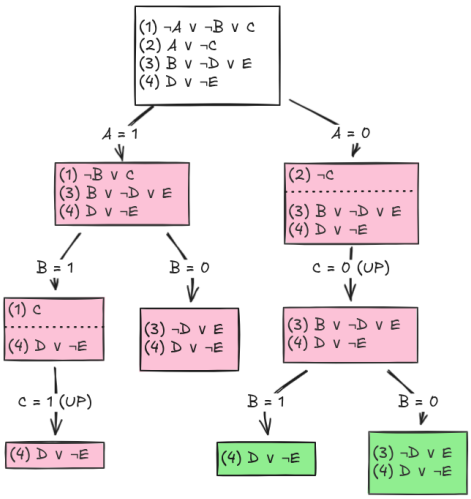
The search performed by a model-counting solver can be represented as a binary search tree whose nodes correspond to variable assignments and represent subproblems, as illustrated in Figure D.1. The shape of such search trees are heavily influenced by the branching heuristic. A poor heuristic may produce a highly skewed tree, where one branch quickly terminates while the other remains large. A more balanced branching choice yields similarly sized subproblems on both sides of the split which may result in less decisions and better solver performance, as will be illustrated in the example below.

Example 21. *Consider the formula*

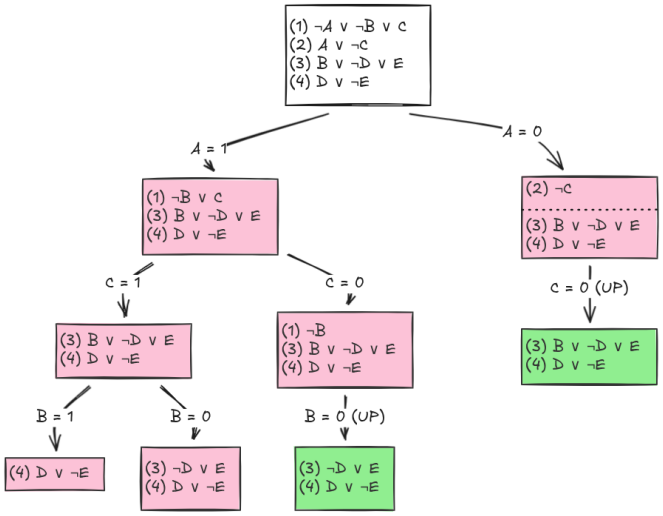
$$\begin{aligned}\mathcal{F} = &(\neg A \vee \neg B \vee C)\\ &\wedge (A \vee \neg C)\\ &\wedge (B \vee \neg D \vee E)\\ &\wedge (D \vee \neg E).\end{aligned}$$

Initially this formula is a single connected component, as illustrated by the graph representation in Figure D.2. Five variables are available for branching: A, B, C, D, E*. Structurally, the meaningful branching choices reduce to either branching on* B *or branching on any variable from* $\{A, C, D, E\}$*.*

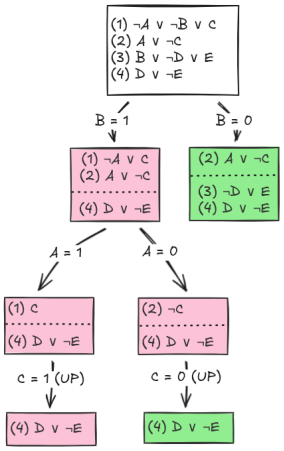
The figures below visualize the different search trees produced by these options. Branching on B yields well-balanced subcomponents. Both the true and false branches split into components of three variables. Branching on other variables often produces more unbalanced trees and fewer repeated large substructures.



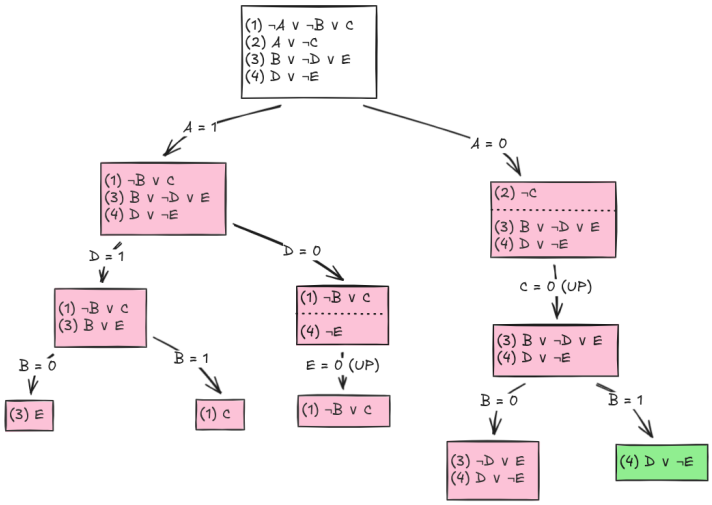
(A) Branch on A, B



(B) Branch on A, C



(C) Branch on B



(D) Branch on A, D

FIGURE D.1: Different search trees. Green indicates that the component is present in the cache, while red indicates absence. Each box represents a component. depths shown are limited to components containing at least two variables for readability. Dotted lines inside component boxes indicate disjoint components. Labels such as *A*, *B* denote branching first on variable *A* at the root, followed by branching on *B* at the next decision level.

Figure D.2 shows the graph representation of the CNF instance and highlights how B is centrally located between two communities. In this instance, branching on B splits the component into two equal sized components, which leads directly to reuse in the cache and to shorter decision trees as illustrated in Table D.1.

Branch on	Total steps	Total decisions	Cache hits
A, B	16	3	2
A, C	16	3	2
A, D	20	4	1
B	11	2	2

TABLE D.1: Cost metrics for the example search trees. *'A, B'* denotes branching first on variable *A* at the root, followed by branching on *B* at the next decision level.

As a side effect, if the component is symmetrical around the central node such as in this case, splitting on the central node may not only produce equal sized components but may also correspond to finding structurally identical components. In this case, as can be seen in Figure D.1, branching on variable B, resulted in a cache hit direclty under this decision level.

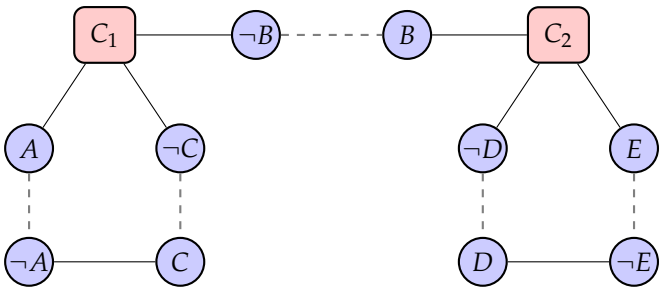


FIGURE D.2: Graph representation of the CNF instance. The *B* node is centrally located between two communities.

Appendix E

Use of Generative AI

Code of conduct and transparency statement on the use of GenAI for KU Leuven-students (academic year 2025-2026)

Generative AI (GenAI) assistance tools can be used to generate various types of content, including text, images, code, video, music, or combinations thereof. Common examples of such tools include ChatGPT, Google Gemini, Microsoft Copilot, Midjourney, Claude.ai, Perplexity.ai, and DALL·E, among others.

This code of conduct is a tool that helps students to be transparent about the use of GenAI and fits within the university's principles on academic integrity.

Important guidelines and remarks

- **Sensitive or personal data:** Some GenAI tools protect your input and sensitive or personal data better than others. There is often no transparency on what the owners of the AI applications do with the data entered. Therefore, **do not enter sensitive or personal data in free GenAI tools.** More info about what sensitive and personal data are, is described in the three confidentiality levels of the KU Leuven data classification model (non-confidential, confidential, strictly confidential). For confidential data you should preferably use **M365 Copilot Chat with your KU Leuven account**. If you use another GenAI tool, you must be **absolutely certain** it does **not store or reuse the data you enter**. **Try to avoid entering these data. Do so only if strictly necessary, and only to the extent required. For personal data, work with anonymous or pseudonymized data.** Additionally, in case of strictly confidential data this should first be discussed with the teaching staff of the course or your thesis supervisor.
- **Copyrighted materials**: For **lawfully obtained copyrighted material** you should preferably use **M365 Copilot Chat with your KU Leuven account**. If you use another GenAI tool, you must be **absolutely certain** it does **not store or reuse the data you enter**.
- GenAI assistance may not be used for data or topics covered by a **Non-Disclosure Agreement (NDA)**. Even in the absence of an NDA, certain information may still need to be treated as confidential—for example, due to regulatory requirements or the risk of significant harm to the university if disclosed. In case of doubt, check with your teaching staff or supervisor.
- If your master's thesis is under **embargo**, you should first discuss with your supervisor whether the use of GenAI is permitted.
- Before using a GenAI tool, always consider whether its use is responsible, including from a **sustainability perspective** (e.g. when using GenAI as a search engine, language assistant,…).
- **Take a scientific and critical attitude** when interacting with GenAI assistance and interpreting its output, that may not always be correct.

- As a student you are responsible for complying with Article 84 of the Regulations on Education and Examinations: your report or thesis should reflect your own knowledge, understanding and skills. Be aware that plagiarism rules also apply to (work that is the result of) the use of GenAI assistance tools.

  > ***Exam Regulations Article 84**: "Every conduct individual students display with which they (partially) inhibit or attempt to inhibit a correct judgement of their own knowledge, understanding and/or skills or those of other students, is considered an irregularity which may result in a suitable penalty. A special type of irregularity is plagiarism, i.e. copying the work (ideas, texts, structures, designs, images, plans, codes , …) of others or prior personal work in an exact or slightly modified way without adequately acknowledging the sources. Every possession of prohibited resources during an examination (see article 65) is considered an irregularity."*

- In order to maintain academic integrity and avoid plagiarism, **more information about being transparent on the use of GenAI assistance and about correctly citing and referencing GenAI** can be found on this website for students (Dutch/English).
- **Additional reading: KU Leuven guidelines on responsible use of Generative AI tools, and other information** (Dutch/English)

A few final words

If you are uncertain whether or not you should declare your use of GenAI tools, we suggest that you discuss this with your instructor or supervisor. It is always safer to declare GenAI use, even when it is not strictly required. However, declaring GenAI use does not entail that its use is allowed; the right column in the table below provides more detailed instructions in this regard (code of conduct).

Moreover, advanced AI tools are evolving rapidly, and their capabilities have expanded significantly in a short period of time. As a result, we do not yet have all the answers about their responsible use. Finally, it is important to follow-up on the most recent evolutions in AI technologies, to have an open mind but also to be a bit cautious, to communicate with instructors, teaching assistants, supervisors and peers, to be as transparent as we can, and to learn together as we move along.

Student name: El Kaddouri Ibrahim.....**Student number: R0855183**.....

Please indicate with "X" whether it relates to a course assignment or to the Bachelor's or Master's thesis:

O This form is related to a **course assignment**.

Course name: ..... **Course number:** .....

This form is related to **my** O **Bachelor's or** ☒ **Master's thesis**.

Title Bachelor's or Master's thesis: Hierarchical Symmetric Component Caching In Model Counting

Supervisor: Prof. dr. Luc De Raedt.....

Daily supervisor: Dr. ir. Vincent Derkinderen.....

Please indicate with "X":

O **I did not use** any GenAI assistance tool.

☒ **I did use** GenAI Assistance. In this case **specify which ones** (e.g. ChatGPT, M365 Copilot,...):

GenAI assistance used as/for:	**Name of the GenAI tool(s) used.** If helpful, also describe in which way you were using GenAI related to what is specified as code of conduct.	**Code of conduct:**
		For each of the categories below, always take into account the important guidelines and remarks mentioned above (e.g. copyrighted data, sensitive or personal data,…).
As a language assistant for reviewing or improving texts I wrote myself	ChatGPT, Claude, Mistral	This use is similar to using spelling and grammar check tools. In general, you do not have to refer to such kind of GenAI use in the text. **However, be careful:** - When using GenAI tools on texts you did not write yourself to improve the text, **you have to refer to the original source or author**, otherwise you are committing plagiarism and thus an irregularity.
As a paraphrasing tool	ChatGPT, Claude, Mistral	This use is allowed except when it is prohibited by the teacher or the program of study. You may paraphrase your own text or texts by an author other than yourself and take inspiration from what a GenAI tool or another tool suggests (unless it is not allowed). In general, you do not have to refer to such kind of GenAI use or other paraphrasing tools in the text. **However, be careful:** - If it entails text by an author other than yourself, **you are not allowed to include that paraphrased text without reference to the original source or author.** Without such reference, you would be committing plagiarism

		and thus an irregularity.
For translation aid to improve texts I wrote myself or to better understand text from others	ChatGPT, Claude, Mistral	This use is allowed except when it is prohibited by the teacher or the program of study. It is similar to using translation tools (Google translate, DeepL, …). In general, you do not have to refer to such kind of GenAI use in the text. **However, be careful:** - **You are not allowed to include that translated text without reference to the original source or author.** Without such reference, you would be committing plagiarism and thus an irregularity. - Always check the translated text for correctness and meaning.
As a search engine to get information on a topic or to search for existing research on the topic	ChatGPT	This use is similar to e.g. a Google search or checking Wikipedia. If you write your own text based on this information, you do not have to refer to the use of GenAI in the text. **You only have to refer to the existing research and references you have checked and used** (without such references you would be committing plagiarism and thus committing an irregularity). **However, be careful:** - Be aware that the output of the GenAI tool cannot be guaranteed as a 100% reliable source of information. The output may not be entirely correct and/or be limited due to the databases it uses. Moreover, knowledge evolves and may change over time; therefore, the database of the GenAI tool may not be up to date. Therefore,

		verify the information and do not just copy-paste it as you should understand and critically process everything you are writing.
For literature search	ChatGPT	This use is comparable to e.g. a Google Scholar search. You do not have to refer to such kind of GenAI use; **you only have to refer to the literature references you have checked and used** (without such references you would be committing plagiarism and thus committing an irregularity).. **However, be careful:** - Be aware that the search output is restricted to the database the GenAI tool is built on. After this initial search, look for scientific sources and conduct your own analysis of the source documents. **Interpret, analyse and process the information you obtained; verify it and do not just copy-paste it** as you should understand and critically process everything you are writing. - Be aware that some GenAI tools may **output no or wrong references**. As a student you are responsible for further checking and verifying the absence or correctness of references; do not just copy-paste it.
For generating programming code	ChatGPT	Use of GenAI for coding is allowed except when it is prohibited by the teacher or the program of study. If used for coding, correctly mention the use of GenAI assistance in accordance with the instructions on the page on being transparant about the use of GenAI.
For generating new (research) ideas		This use of GenAI is allowed except when it is prohibited by the teacher or the program of study. Correctly

		mention the use of GenAI assistance in accordance with the instructions on the page on being transparant about the use of GenAI. **Be careful:** - Further verify in this case whether the idea is novel or not. It is likely that it is related to existing work. **If so, that existing work should be correctly referenced in the text** (without such reference you would be committing plagiarism and thus committing an irregularity). -
For generating synthetic data		Use of GenAI for generating synthetic data is allowed, **provided that it is methodologically and ethically justifiable**, except when it is prohibited by the teacher or the program of study. Always correctly mention the use of GenAI assistance in accordance with the instructions on the page on being transparant about the use of GenAI. **Be careful:** - Always **carefully evaluate** the generated synthetic data for quality and possible bias since the output is highly dependent on the quality of the data on which the models are trained.
For generating blocks of text (other than the allowed use without referencing mentioned above)		According to Article 84 of the Regulations on Education and Examinations your text should allow to correctly and properly assess your own knowledge, understanding and skills. **Therefore, inserting blocks of text without quotes and a reference to GenAI assistance in your work is not allowed.** **Be careful:** - If it is really needed to insert a block of text from a

		GenAI tool, for instance because of the nature of your assignment, mention it as a citation by using quotes and correctly mention the use of GenAI assistance in accordance with the instructions page on being transparant about the use of GenAI. - However, in general, such GenAI use should be kept to an **absolute minimum**; you should always check the original sources.
For generating visuals, video or audio		Use of GenAI for generating visuals, audio or video is allowed except when it is prohibited by the teacher or the program of study. **Be careful:** - If used, refer to GenAI for visuals according to the usual referencing style, following the instructions on the page on referencing GenAI. - If you work with existing visuals, audio or video, **that existing work should be correctly referenced in the text** (without such reference you would be committing plagiarism and thus an irregularity). - Explain the usage in the methods section (if there is one) and optionally attach (or link to) the prompts with the full output (or history).
Other use (specify here; this may also include a combination of types of use mentioned above):		To make sure other use of GenAI is allowed within the course or thesis, and if so, the conditions that may apply, contact the teaching staff of the course or the supervisor of the thesis **beforehand** and explain the intended GenAI use. Also inform the programme director. Motivate how you would comply with Article 84 of the exam regulations. Explain the use and the added value of

		the AI tool you consider to use and how it is in accordance with the assignment or thesis and the KU Leuven guidelines on responsible use of Generative AI tools. Depending on the kind of GenAI use, it may be needed to properly reference it in the text, in accordance with the instructions page on being transparant about the use of GenAI.

Bibliography

F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah. Solving difficult sat instances in the presence of symmetry. In *Proceedings of the 39th Annual Design Automation Conference*, DAC '02, page 731–736, New York, NY, USA, 2002. Association for Computing Machinery. ISBN 1581134614. doi: 10.1145/513918.514102. URL https://doi.org/10.1145/513918.514102.

J. Araújo, C. Chow, and M. Janota. Boosting isomorphic model filtering with invariants. *Constraints*, 27(3):360–379, 2022.

T. Baluta, S. Shen, S. Shinde, K. S. Meel, and P. Saxena. Quantitative verification of neural networks and its security applications. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1249–1264, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367479. doi: 10.1145/3319535.3354245. URL https://doi.org/10.1145/3319535.3354245.

A. H. Basson and D. J. O'Connor. *Introduction to Symbolic Logic*. University Tutorial Press, London, 3rd edition, 1959. ISBN 0-7231-0456-5.

Biere, Armin and Järvisalo, Matti and Kiesl, Benjamin. Preprocessing in sat solving. In *Handbook of Satisfiability*, Frontiers in artificial intelligence and applications, pages 391–435. IOS Press, 2021. doi: 10.3233/FAIA200992.

B. Bliem and M. Järvisalo. Centrality heuristics for exact model counting. In *2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 59–63, 2019. doi: 10.1109/ICTAI.2019.00017.

P. Bonacich. Power and centrality: A family of measures. *American Journal of Sociology*, 92(5):1170–1182, 1987. ISSN 00029602, 15375390. URL http://www.jstor.org/stable/2780000.

M. Brain, J. H. Davenport, and A. Griggio. Benchmarking solvers, sat-style. In SC^2@ *ISSAC*, 2017.

U. Brandes. A faster algorithm for betweenness centrality*. *The Journal of Mathematical Sociology*, 25(2):163–177, 2001. doi: 10.1080/0022250X.2001.9990249. URL https://doi.org/10.1080/0022250X.2001.9990249.

S. Chakraborty, K. S. Meel, and M. Y. Vardi. A scalable approximate model counter. In C. Schulte, editor, *Principles and Practice of Constraint Programming*, pages 200–216, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-40627-0.

M. Chavira and A. Darwiche. On probabilistic inference by weighted model counting. *Artificial Intelligence*, 172(6):772–799, 2008. ISSN 0004-3702. doi: https://doi.org/10.1016/j.artint.2007.11.002. URL https://www.sciencedirect.com/science/article/pii/S0004370207001889.

Y. Collet. xxHash algorithm description. https://github.com/Cyan4973/xxHash/blob/dev/doc/xxhash_spec.md, 2018. Accessed: 2025-10-20.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2022.

T. M. Cover and J. A. Thomas. *Kolmogorov Complexity*, chapter 14, pages 463–508. John Wiley and Sons, Ltd, 2005. ISBN 9780471748823. doi: https://doi.org/10.1002/047174882X.ch14. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/047174882X.ch14.

G. Csárdi and T. Nepusz. The igraph software package for complex network research. *InterJournal*, Complex Systems:1695, 2006. URL https://igraph.org.

A. Darwiche. On the tractable counting of theory models and its application to belief revision and truth maintenance, 2000. URL https://arxiv.org/abs/cs/0003044.

A. Darwiche. New advances in compiling cnf to decomposable negation normal form. In *Proceedings of the 16th European Conference on Artificial Intelligence*, ECAI'04, page 318–322, NLD, 2004. IOS Press. ISBN 9781586034528.

M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962. ISSN 0001-0782. doi: 10.1145/368273.368557. URL https://doi.org/10.1145/368273.368557.

L. De Raedt and A. Kimmig. Probabilistic (logic) programming concepts. *Mach. Learn.*, 100(1):5–47, July 2015. ISSN 0885-6125. doi: 10.1007/s10994-015-5494-z. URL https://doi.org/10.1007/s10994-015-5494-z.

L. De Raedt, A. Kimmig, and H. Toivonen. Problog: a probabilistic prolog and its application in link discovery. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence*, IJCAI'07, page 2468–2473, San Francisco, CA, USA, 2007. Morgan Kaufmann Publishers Inc.

M. Dehmer, M. Grabner, A. Mowshowitz, and F. Emmert-Streib. An efficient heuristic approach to detecting graph isomorphism based on combinations of highly discriminating invariants. *Advances in Computational Mathematics*, 39(2):311–325, 2013.

M. Denecker. Modelling of complex systems. Course notes, KU Leuven, mar 2025. URL https://people.cs.kuleuven.be/~marc.denecker/A-PDF/MCS.pdf.

V. Derkinderen. ProbLog CNF Generator. DTAI, KU Leuven, 2020. CNF generation script (`cnf-generator.py`), June 26, 2020.

J. Devriendt, B. Bogaerts, B. De Cat, M. Denecker, and C. Mears. Symmetry propagation: Improved dynamic symmetry breaking in sat. In *2012 IEEE 24th International Conference on Tools with Artificial Intelligence*, volume 1, pages 49–56. IEEE, 2012.

J. Devriendt, B. Bogaerts, and M. Bruynooghe. Symmetric explanation learning: Effective dynamic symmetry handling for sat. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 83–100. Springer, 2017.

R. Diestel. *Graph theory*. Springer Nature, 2025.

X. Dong, E. Gabrilovich, G. Heitz, W. Horn, N. Lao, K. Murphy, T. Strohmann, S. Sun, and W. Zhang. Knowledge vault: a web-scale approach to probabilistic knowledge fusion. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, page 601–610, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450329569. doi: 10.1145/2623330.2623623. URL https://doi.org/10.1145/2623330.2623623.

L. C. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 40 (1):35–41, 1977. ISSN 00380431, 23257938. URL http://www.jstor.org/stable/3033543.

L. C. Freeman. Centrality in social networks conceptual clarification. *Social Networks*, 1(3):215–239, 1978. ISSN 0378-8733. doi: https://doi.org/10.1016/0378-8733(78)90021-7. URL https://www.sciencedirect.com/science/article/pii/0378873378900217.

C. P. Gomes. lsencode v1.1 quasigroup completion problem generator. GitHub mirror and archived Wayback copy, June 2012. Original software no longer available at Cornell. Archived URL: https://web.archive.org/web/20120616064725/http://www.cs.cornell.edu:80/gomes/SOFT/lsencode-v1.1.tar.Z; GitHub mirror: https://github.com/HelgeS/lsencode?tab=readme-ov-file.

Gomes Carla P., S. Ashish, and S. Bart. Model counting. In *Handbook of Satisfiability*, Frontiers in artificial intelligence and applications, page 633–654. IOS Press, 2009. doi: 10.3233/978-1-58603-929-5-633.

E. Gribkoff, D. Suciu, and G. Van den Broeck. Lifted probabilistic inference: A guide for the database researcher. *Bulletin of the Technical Committee on Data Engineering*, 37(3):6–17, Sept. 2014. URL http://starai.cs.ucla.edu/papers/GribkoffDEBUL14.pdf.

J. Huang and A. Darwiche. Dpll with a trace: from sat to knowledge compilation. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, IJCAI'05, page 156–162, San Francisco, CA, USA, 2005. Morgan Kaufmann Publishers Inc.

A. Kimmig, G. Van den Broeck, and L. De Raedt. Algebraic model counting. *Journal of Applied Logic*, 22:46–62, 2017. ISSN 1570-8683. doi: https://doi.org/10.1016/j.jal.2016.11.031. URL https://www.sciencedirect.com/science/article/pii/S157086831630088X. SI:Uncertain Reasoning.

M. Kitching and F. Bacchus. Symmetric component caching. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence*, IJCAI'07, page 118–124, San Francisco, CA, USA, 2007. Morgan Kaufmann Publishers Inc.

D. E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., USA, 1998. ISBN 0201896850.

J.-M. Lagniez and P. Marquis. Preprocessing for propositional model counting. *Proceedings of the AAAI Conference on Artificial Intelligence*, 28(1), Jun. 2014. doi: 10.1609/aaai.v28i1.9116. URL https://ojs.aaai.org/index.php/AAAI/article/view/9116.

A. L. D. Latour, B. Babaki, A. Dries, A. Kimmig, G. Van den Broeck, and S. Nijssen. Combining stochastic constraint optimization and probabilistic programming. In J. C. Beck, editor, *Principles and Practice of Constraint Programming*, pages 495–511, Cham, 2017. Springer International Publishing. ISBN 978-3-319-66158-2.

M. Lauria, J. Elffers, J. Nordström, and M. Vinyals. Cnfgen: A generator of crafted benchmarks. In S. Gaspers and T. Walsh, editors, *Theory and Applications of Satisfiability Testing – SAT 2017*, pages 464–473, Cham, 2017. Springer International Publishing. ISBN 978-3-319-66263-3.

J. Marques-Silva, I. Lynce, and S. Malik. Chapter 4. conflict-driven clause learning sat solvers. In *Handbook of Satisfiability*, Frontiers in artificial intelligence and applications, pages 133–182. IOS Press, Feb. 2021. doi: 10.3233/faia200987.

B. McKay. Description of graph6, sparse6 and digraph6 encodings. https://users.cecs.anu.edu.au/~bdm/data/formats.txt, 2022. Updated Jun 2015; Apr 2022. Accessed 2025-01-05.

B. D. McKay. Re: [Nauty] uniqueness of canonical labeling. Nauty mailing list archive, February 2024. URL https://mailman.anu.edu.au/pipermail/nauty/2024-February/000962.html. Accessed: 2025-10-25.

B. D. McKay and A. Piperno. Practical graph isomorphism, ii. *Journal of Symbolic Computation*, 60:94–112, 2014. ISSN 0747-7171. doi: https://doi.org/10.1016/j.jsc.2013.09.003. URL https://www.sciencedirect.com/science/article/pii/S0747717113001193.

M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient sat solver. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, page 530–535, New York, NY, USA, 2001. Association for Computing Machinery. ISBN 1581132972. doi: 10.1145/378239.379017. URL https://doi.org/10.1145/378239.379017.

K. Pipatsrisawat and A. Darwiche. A lightweight component caching scheme for satisfiability solvers. In J. Marques-Silva and K. A. Sakallah, editors, *Theory and Applications of Satisfiability Testing – SAT 2007*, pages 294–299, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-72788-0.

J.-F. Puget. Automatic detection of variable and value symmetries. In P. van Beek, editor, *Principles and Practice of Constraint Programming - CP 2005*, pages 475–489, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-32050-0.

T. Sang, F. Bacchus, P. Beame, H. A. Kautz, and T. Pitassi. Combining Component Caching and Clause Learning for Effective Model Counting. In *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing*, pages 20–28, 2004.

T. Sang, P. Beame, and H. Kautz. Solving bayesian networks by weighted model counting. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05)*, volume 1, pages 475–482. AAAI Press, 2005a.

T. Sang, P. Beame, and H. Kautz. Heuristics for fast exact model counting. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing*, SAT'05, page 226–240, Berlin, Heidelberg, 2005b. Springer-Verlag. ISBN 3540262768. doi: 10.1007/11499107_17. URL https://doi.org/10.1007/11499107_17.

U. Schöning and J. Torán. *The Satisfiability Problem : Algorithms and Analyses*. Mathematik Für Anwendungen Series ; v.3. Lehmanns Fachbuchhandlung GmbH, Abt. Verlag, Berlin, 1st ed. edition, 2013. ISBN 9783865416483.

S. Sharma, S. Roy, M. Soos, and K. S. Meel. Ganak: A scalable probabilistic exact model counter. In *IJCAI*, volume 19, pages 1169–1176, 2019.

H. Shen. Solve a n-queens problem using a sat solver. https://web.archive.org/web/20201027062513/https://sites.google.com/site/haioushen/search-algorithm/solvean-queensproblemusingsatsolver, Dec. 2011. Original post: 2011-12-02 (updated 2011-12-11). Archived via Internet Archive (snapshot: 2020-10-27). Accessed: 2025-12-22.

J. a. P. M. Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *Proceedings of the 9th Portuguese Conference on Artificial Intelligence: Progress in Artificial Intelligence*, EPIA '99, page 62–74, Berlin, Heidelberg, 1999. Springer-Verlag. ISBN 354066548X.

D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017. URL https://arxiv.org/abs/1712.01815.

S. Teuber and A. Weigl. Quantifying software reliability via model-counting. In A. Abate and A. Marin, editors, *Quantitative Evaluation of Systems*, pages 59–79, Cham, 2021. Springer International Publishing. ISBN 978-3-030-85172-9.

M. Thurley. sharpsat: counting models with advanced component caching and implicit bcp. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing*, SAT'06, page 424–429, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3540372067. doi: 10.1007/11814948_38. URL https://doi.org/10.1007/11814948_38.

P. Vaezipoor, G. Lederman, Y. Wu, C. Maddison, R. B. Grosse, S. A. Seshia, and F. Bacchus. Learning branching heuristics for propositional model counting. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(14):12427–12435, May 2021. ISSN 2159-5399. doi: 10.1609/aaai.v35i14.17474. URL http://dx.doi.org/10.1609/aaai.v35i14.17474.

L. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8(2):189–201, 1979. ISSN 0304-3975. doi: https://doi.org/10.1016/0304-3975(79)90044-6. URL https://www.sciencedirect.com/science/article/pii/0304397579900446.

T. van Bremen, V. Derkinderen, S. Sharma, S. Roy, and K. S. Meel. Symmetric component caching for model counting on combinatorial instances. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(5):3922–3930, May 2021. doi: 10.1609/aaai.v35i5.16511. URL https://ojs.aaai.org/index.php/AAAI/article/view/16511.

W. Wang, M. Usman, A. Almaawi, K. Wang, K. S. Meel, and S. Khurshid. A study of symmetry breaking predicates and model counting. In A. Biere and D. Parker, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 115–134, Cham, 2020. Springer International Publishing. ISBN 978-3-030-45190-5.

S. Wasserman and K. Faust. *Social Network Analysis: Methods and Applications*. Structural Analysis in the Social Sciences. Cambridge University Press, 1994.

D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world'networks. *nature*, 393(6684):440–442, 1998.

B. P. Welford. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3):419–420, 1962. doi: 10.1080/00401706.1962.10490022. URL https://www.tandfonline.com/doi/abs/10.1080/00401706.1962.10490022.