Symmetric Component Caching for Model Counting on Combinatorial Instances

Timothy van Bremen[1*], **Vincent Derkinderen**[1*], **Shubham Sharma**[2*], **Subhajit Roy**[3], **Kuldeep S. Meel**[4]

[1]KU Leuven [2]Nutanix Software India Pvt. Ltd.
[3]Indian Institute of Technology Kanpur [4]National University of Singapore

Abstract

Given a propositional formula ψ, the *model counting* problem, also referred to as #SAT, seeks to compute the number of satisfying assignments (or models) of ψ. Modern search-based model counting algorithms are built on conflict-driven clause learning, combined with the caching of certain subformulas (called *components*) encountered during the search process. Despite significant progress in these algorithms over the years, state-of-the-art model counters often struggle to handle large but structured instances that typically arise in combinatorial settings. Motivated by the observation that these counters do not exploit the inherent symmetries exhibited in such instances, we revisit the component caching architecture employed in current counters and introduce a novel caching scheme that focuses on identifying *symmetric* components. We first prove the soundness of our approach, and then integrate it into the state-of-the-art model counter GANAK. Our extensive experiments on hard combinatorial instances demonstrate that the resulting counter, SYMGANAK, leads to improvements over GANAK both in terms of PAR-2 score and the number of instances solved.

1 Introduction

Given a propositional formula ψ, the *model counting* problem, also referred to as #SAT, seeks to compute the number of satisfying assignments (or models) of ψ. Model counting is a fundamental problem in artificial intelligence with a wide variety of applications such as probabilistic inference (Chavira and Darwiche 2008), neural network verification (Baluta et al. 2019), computational biology (Sashittal and El-Kebir 2020), and the like. Consequently, the problem of model counting has been subject to intense theoretical and practical investigations over the past four decades. The seminal work of Valiant (1979) showed that model counting is #P-complete. Subsequently, Toda (1991) proved that every problem in the polynomial hierarchy can be efficiently solved using only a single call to a #P oracle; more formally, $\mathsf{PH} \subseteq \mathsf{P}^{\#\mathsf{P}}$.

Practical strategies for model counting span a variety of approaches, from approximate techniques (Stockmeyer 1983; Soos and Meel 2019) with probabilistic error bounds, to exact counting (Birnbaum and Lozinskii 1999; Bayardo Jr and Pehoushek 2000; Sang et al. 2004; Thurley 2006; Aziz et al. 2015; Oztok and Darwiche 2015; Lagniez and Marquis 2017). Many solvers use variants of the classic DPLL algorithm for SAT solving (Davis, Logemann, and Loveland 1962), with optimizations geared towards model counting (Birnbaum and Lozinskii 1999). One prominent optimization used in such algorithms is *component caching*: during the search process subsets of clauses that can be solved independently (referred to as *components*) are identified, solved, and cached. When the same component appears again along a different search path, the model count of the component can simply be returned from the cache, alleviating the need to recompute it (Bacchus, Dalmao, and Pitassi 2003).

The exact representation scheme used for storing components in the cache differs between solvers: CACHET (Sang et al. 2004; Sang, Beame, and Kautz 2005) uses a simple encoding where the literals in each clause are represented as integers with clauses separated by a sentinel. SHARPSAT (Thurley 2006) uses a *hybrid encoding* that achieves a more compact representation. GANAK (Sharma et al. 2019) introduced the notion of a *probabilistic* cache: the component encodings are hashed into a yet smaller representation to enable better cache utilization but, in the process, paying a price with a (small) probability of incorrect counts due to hash collisions. The algorithm is parametrized by the probability of collision, which can be set as small as the user desires, at the expense of poorer cache utilization due to longer hash lengths. In addition, GANAK adds several other optimizations that allow it to significantly outperform other state-of-the-art model counters.

However, all existing cache indexing schemes (including that of GANAK) declare a cache hit only on *exact* matches on components. We make an important observation that there are often components that are structurally identical but differ only in the variables appearing in the formula. Due to the fact that components employ variables disjoint from the rest of the formula, the model counts can also be transferred across such structurally identical components. Such *symmetric components* occur naturally in many instances, particularly those arising from combinatorial problems. It is worth remarking that the counting variants of many combinatorial problems also enjoy straightforward reductions to #SAT, such as n-queens, quasigroup (Latin square) completion, and graph k-colouring (Aloul et al. 2002; Yang 1991; Lauria et al. 2017;

*equal contribution

Gomes and Shmoys 2002; Wang et al. 2020).

Our primary contribution is exploiting the inherent symmetry exhibited in combinatorial problems for component caching-based model counters. To this end, we propose and formalise the notion of *symmetric component caching*—allowing for the use of cached model counts even across components that are only structurally identical (*symmetric*) and not exact matches. We first prove that the proposed scheme is sound when combined with clause learning. We, then, augment the state-of-the-art counter GANAK with *symmetric component caching*, along with several low-level but crucial technical improvements. The resulting counter, called SYMGANAK, outperforms the state-of-the-art model counter GANAK on PAR-2 score and number of instances solved, achieving significant performance gains in terms of runtime.

The performance improvement of SYMGANAK over GANAK in the context of combinatorial instances should be viewed in the context of the recent SAT solver-assisted breakthroughs in mathematics. The scalability of SAT solving has allowed questions of existence of particular structures to be *carefully* reduced to SAT queries (Brakensiek et al. 2020; Heule, Kullmann, and Marek 2016); since counting is a fundamental combinatorial object, the resolution of several open problems in combinatorics via automated reasoning techniques would significantly benefit from the availability of efficient model counters for such instances.

The rest of the paper is organized as follows: we present related work in Section 2. We then present notations and preliminaries in Section 3. We present the primary technical contribution, symmetric component caching, in Section 4. We then present empirical analysis in Section 5 and finally conclude in Section 6.

2 Related Work

We are not the first to explore symmetry in propositional logic: the use of precomputed symmetry-breaking predicates to speed up SAT solving dates back to the 1990s (Crawford et al. 1996). More recent work has extended this idea with the use of more efficient symmetry-breaking formulas (Devriendt et al. 2016). Taking a different approach, others have examined how symmetry information can be used at runtime for SAT solvers (Sabharwal 2009). Outside of SAT solving, Kitching and Bacchus (2007) explored symmetry in the context of solving constraint optimization problems with decomposable objective functions. Bart et al. (2014) exploited symmetry to achieve space savings in knowledge compilation.

However, relatively little work has focused on using symmetry in the context of model counting. We are aware only of very recent work by Wang et al. (2020), who studied the use of existing model counting algorithms on formulas conjoined with symmetry-breaking predicates, thus effectively counting models up to isomorphism. SYMGANAK differs in that it counts all models of the formula, and does not rely on symmetries of the input formula itself—rather, it exploits symmetry amongst the components encountered *during runtime* of the algorithm. Note that, in principle, this does not require symmetry to be present in the input formula for our approach to be effective: if a variable ordering can be chosen in such a way that propagating variables in this order leads to structurally identical components, this will suffice to see performance gains over existing counters. We compare different variable ordering heuristics later in this paper, and also examine the implications of our approach when integrating with many of the features (such as clause learning) present in modern model counters.

3 Background

In this section, we provide some background on propositional logic and model counting, and review some preliminaries on graph isomorphism and the #DPLL algorithm with component caching.

Propositional logic We deal with propositional logic throughout this paper. *Variables* are symbols which can take the value true or false. A *literal* is a positive variable x or its negation $\neg x$. A *propositional formula* ψ is defined inductively as either a literal, the conjunction or the disjunction of two formulas. We define the support of a formula ψ, denoted by $vars(\psi)$, as the set of variables that occur in ψ, e.g. if $\psi = x_1 \wedge \neg x_2$ then $vars(\psi) = \{x_1, x_2\}$. Similarly, we define $lits(\psi)$ as the set of all literals that can be formed using variables from $vars(\psi)$. A *clause* is a disjunction of literals, e.g. $x_1 \vee \neg x_2 \vee x_3$. A formula is said to be in *conjunctive normal form* (CNF) if it is a conjunction of clauses. As is standard in the model counting literature, we assume in the remainder of this paper that all formulas are specified in CNF. When applying π, an assignment of literals to truth values, to ψ, we denote the resulting formula as $\psi|_\pi$.

We follow the usual semantics of propositional logic, which we omit here for brevity. Denote by R_ψ the set of all models of a formula ψ, and by $R_{\psi\downarrow P}$ the projection of R_ψ onto P (that is, the models in R_ψ restricted to literals formed only from P). The satisfiability problem is to determine whether or not a given formula ψ has a model, that is, decide if $|R_\psi| > 0$. Analogously, the model counting problem is to determine the number of models of a given formula (i.e., determine $|R_\psi|$).

Graph isomorphism Below we describe a few notions relating to graph theory that we will use to detect structural symmetries in formulas. We first review the definition of a coloured graph.

Definition 1 (Coloured graph)**.** *A coloured graph is a three-tuple $G = (V, E, P)$, where (V, E) specifies an undirected graph and $P = \{V_i\}_{i=1}^k$ is a partition of the vertices into k distinct colours. We further denote* $\mathsf{colour}(v) = i$ *if* $v \in V_i$.

Given two coloured graphs, one can ask if they are *isomorphic*.

Definition 2 (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 $\phi : G \to H$ such that:*

- $\forall v, w \in V_1$, $(v, w) \in E_1 \iff (\phi(v), \phi(w)) \in E_2$
- $\forall v \in V_1$ $\mathsf{colour}(v) = \mathsf{colour}(\phi(v))$

Algorithm 1: #DPLL algorithm with component caching.

```
1  function GetModelCount(ψ):
2      encoding ← Encode(ψ)
3      if encoding in cache then
4          return CacheGet(encoding)
5      else
6          pick a literal l in ψ
7          |R_{ψ_l}| ← CountConditioned(ψ, l)
8          |R_{ψ_¬l}| ← CountConditioned(ψ, ¬l)
9          CacheInsert(encoding, |R_{ψ_l}| + |R_{ψ_¬l}|)
10         return |R_{ψ_l}| + |R_{ψ_¬l}|
11     end
12 function CountConditioned(ψ, l):
13     ψ_l ← propagate units on ψ|_l
14     if ψ_l contains empty clause then
15         return 0
16     else if ψ_l contains no clauses then
17         v ← number of unassigned variables in ψ_l
18         return 2^v
19     else
20         |R_{ψ_l}| ← 1
21         C ← DisjointComponents(ψ_l)
22         for C_i ← C do
23             |R_{ψ_l}| ← |R_{ψ_l}| × GetModelCount(C_i)
24         end
25         return |R_{ψ_l}|
26     end
```

The (coloured) graph isomorphism problem is to determine whether or not two (coloured) graphs are isomorphic. The coloured graph isomorphism problem is polynomial-time reducible to its uncoloured counterpart (Schweitzer 2009), so we will omit the word "coloured" when appropriate. Although the complexity theoretic status of the graph isomorphism problem remains open, relatively efficient algorithms exist in practice (McKay and Piperno 2014). A closely related problem is that of *graph canonization*, which is to compute the *canonical labelling* of a given graph.

Definition 3 (Canonical labelling)**.** *Given graphs G and H, a canonical labelling of a graph G is a new graph* $\mathsf{Canon}(G)$*, such that H is isomorphic to G if and only if* $\mathsf{Canon}(G) = \mathsf{Canon}(H)$.

As implied by the name, $\mathsf{Canon}(G)$ is effectively a relabelling of G. Thus, given an oracle for graph canonization, verifying isomorphism between graphs can be done by computing the canonical labelling for each graph, and checking whether the resulting graphs are identical.

#DPLL algorithm with component caching A popular model counting approach is a variant of the classic DPLL algorithm for SAT solving (Davis, Logemann, and Loveland 1962; Birnbaum and Lozinskii 1999). This algorithm (Algorithm 1) repeatedly chooses a literal l and creates two branches (lines 6–8): one in which the variable is conditioned to be true ($\psi|_l$), and another where it is conditioned to be false ($\psi|_{\neg l}$). When assigning true (false) to l, any clause containing the l ($\neg l$) is satisfied and thus removed; from the remaining clauses $\neg l$ (l) is eliminated. In the process, if all clauses are removed, the formula is satisfied and the model count under the current partial assignment is 2^v, where v is the number of unassigned variables. Instead, if a clause becomes empty, the formula is unsatisfied and further assignments do not lead to any model (lines 13–18).

Arguably one of the most impactful optimizations to the above algorithm has been *component caching*. Component caching identifies subformulas (or *components*) that can be solved independently and memoizes them, allowing for a shallower search tree. The component caching step happens each time before a variable is selected and propagated.

Definition 4 (Component)**.** *Consider a partitioning of a formula* ψ *into sets of clauses* $\psi = \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n$ *such that* $vars(\mathcal{C}_i) \cap vars(\mathcal{C}_j) = \emptyset$ *for* $i \neq j$*. Then each* $\mathcal{C}_i$ *is called a component of* ψ*, and we have* $|R_\psi| = \prod_{i=1}^{n} |R_{\mathcal{C}_i}|$.

Example 1. *Consider the formula* ψ *and its conditioning* $\psi|_b$ *:*

$$\psi \begin{cases} a \vee \neg c \\ \neg a \vee \neg b \vee c \\ d \vee \neg e \\ \neg d \vee e \vee b \end{cases} \qquad \psi|_b \begin{cases} a \vee \neg c \\ \neg a \vee c \\ d \vee \neg e \end{cases}$$

Notice that the first two clauses of $\psi|_b$ *do not share any variables with the third clause. This means that* $\psi|_b$ *can be split up into the two components* $\mathcal{C}_1 = \{a \vee \neg c, \neg a \vee c\}$ *and* $\mathcal{C}_2 = \{d \vee \neg e\}$ *such that* $|R_{\psi|_b}| = |R_{\mathcal{C}_1}||R_{\mathcal{C}_2}|$.

After solving and computing the model count of a component (lines 20–25), the result is cached so that the model count can be reused for an identical component encountered elsewhere in the search tree (line 9). The cache is indexed by an encoding for a component ψ ($\mathsf{Encode}(\psi)$ in line 2). Designing efficient encodings for the components has been an important research direction (Sang et al. 2004; Sang, Beame, and Kautz 2005; Thurley 2006; Sharma et al. 2019). More compact encodings improve cache utilization, allowing memoization of more components for a given cache size.

There exist many *branching heuristics* that dictate the selection of variables for branching decisions (line 6). VSADS (Sang, Beame, and Kautz 2005) is a popular heuristic that selects the decision variable by considering the number of instances of a variable in the current subformula and giving priority to variables that are present in recently generated conflict clauses. CSVSADS (Sharma et al. 2019) proposes a component-cache aware heuristic that adapts VSADS by discouraging variables appearing in cached components in an attempt to improve the cache hit-rate.

The core #DPLL algorithm with component caching has been optimised and extended with additional features over the years. Such optimisations include *conflict driven clause learning* (Sang et al. 2004), a variety of *variable and phase selection heuristics* (Sharma et al. 2019; Thurley 2006), and *implicit BCP* (Thurley 2006), along with many others. We will touch on some of these later in the paper.

4 Approach

In this section, we introduce *symmetric component caching* as well as some additional heuristics and optimizations added in SYMGANAK.

4.1 Symmetric component caching

Let us formally define the notion of *symmetric components*.

Definition 5 (Symmetric components)**.** *Two formulas ψ_1 and ψ_2 are said to be (semantically) symmetric if there is a bijection $\pi : lits(\psi_1) \to lits(\psi_2)$ such that $R_{\psi_2} = R_{\pi(\psi_1)}$ and $\forall l \in lits(\psi_1) : \neg\pi(l) = \pi(\neg l)$.*

Example 2. *Suppose we observe the following two components in different places in our search tree. We can show that the two components are semantically symmetric: for the mapping $\pi = \{a \mapsto \neg c, c \mapsto a, d \mapsto b\}$ (fixing all other literals), we have $R_{\mathcal{C}_2} = R_{\pi(\mathcal{C}_1)}$.*

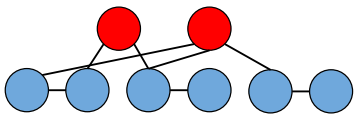
$$\mathcal{C}_1 \begin{cases} \neg a \vee c \\ a \vee c \vee d \end{cases} \qquad \mathcal{C}_2 \begin{cases} \neg c \vee a \vee b \\ a \vee c \end{cases}$$

Detecting symmetries We can employ the following two-step process to detect if two components are symmetric: (i) first encode the formulas ψ_1 and ψ_2 as graphs $\mathsf{Gr}(\psi_1)$ and $\mathsf{Gr}(\psi_2)$; then (ii) check whether their canonical labellings are equal, i.e. $\mathsf{Canon}(\mathsf{Gr}(\psi_1)) = \mathsf{Canon}(\mathsf{Gr}(\psi_1))$. We first outline the encoding in step (i).

Definition 6. *(Aloul et al. 2002) The graph representation of ψ, denoted $\mathsf{Gr}(\psi) = (V, E, P)$, is a coloured graph constructed in the following manner:*

1. *Add a node n_{c_i} to V with $\mathsf{colour}(n_{c_i}) =$ red for each clause c_i in ψ.*
2. *Add a node n_{l_i} to V with $\mathsf{colour}(n_{l_i}) =$ blue for each literal l_i in $lits(\psi)$.*
3. *Add an edge (n_{l_i}, n_{l_j}) joining each literal l_i with its negated counterpart l_j.*
4. *Add an edge (n_{c_i}, n_{l_i}) if l_i occurs in the clause c_i, thus joining every clause node with its constituent literal nodes.*

Example 3. *Using the definition above, the graph form of either $\mathcal{C}_1$ or $\mathcal{C}_2$ from Example 2 both yield a graph with the following structure:*



We now state our primary soundness argument for the symmetric component cache:

Theorem 1. *Given two components ψ_1 and ψ_2 if $\mathsf{Canon}(\mathsf{Gr}(\psi_1)) = \mathsf{Canon}(\mathsf{Gr}(\psi_2))$ then $|R_{\psi_1}| = |R_{\psi_2}|$.*

Proof Sketch. It will suffice to show that if $\mathsf{Canon}(\mathsf{Gr}(\psi_1)) = \mathsf{Canon}(\mathsf{Gr}(\psi_2))$, then ψ_1 and ψ_2 are semantically symmetric. Any formula recovered from a graph $\mathsf{Gr}(\psi_1)$ ($\mathsf{Gr}(\psi_2)$) is semantically symmetric to ψ_1 (ψ_2): this is because it is unique up to a reordering of clauses and literals, and relabelling of literals. The same statement holds after canonical labelling: that is, any formula reconstructed from $\mathsf{Canon}(\mathsf{Gr}(\psi_1))$ ($\mathsf{Canon}(\mathsf{Gr}(\psi_2))$) is semantically symmetric to ψ_1 (ψ_2), since a canonical labelling of a graph yields a bijection on the nodes such that colours and edges are preserved (cf. Definition 2 and 3). Thus, putting the two statements together we get that if $\mathsf{Canon}(\mathsf{Gr}(\psi_1)) = \mathsf{Canon}(\mathsf{Gr}(\psi_2))$, then ψ_1 and ψ_2 are semantically symmetric. □

Algorithm 2: Encoding symmetric components

```
1 function Encode(ψ):
2     graph ← Gr(ψ)
3     canonial_label ← Canon(graph)
4     h ← H_cl(n, m)
5     return h(canonial_label)
```

Probabilistic symmetric component caching (PSCC) To improve cache utilization, SYMGANAK calculates an m-bit hash of each canonical labelling using the hash family $H_{cl}(n, m)$ mapping $\{0,1\}^n \to \{0,1\}^m$ (Lemire and Kaser 2016). While probabilistic component caching (PCC) was initially proposed in GANAK, we adapt the scheme to work with cached graphs. The string that is hashed is created from the vertices and edges of the canonical labelling in sorted order. This hash (rather than the canonical labelling itself) is stored in the cache. Hashing makes the solver probabilistic due to the risk of a hash collision, but the confidence δ (influencing the hash length m) is configurable by the user and can be set to a small value for high confidence. The probabilistic guarantees proven for PCC in GANAK (Sharma et al. 2019) continue to hold for PSCC in SYMGANAK.

Final scheme Algorithm 2 shows the final algorithm of the $\mathsf{Encode}(\cdot)$ function (referred in Algorithm 1) for SYMGANAK. To compute an encoding for a component ψ, SYMGANAK computes the canonical labeling of the graph representation of ψ (line 3). SYMGANAK then randomly samples a hash function from the hash function family $H_{cl}(n, m)$ (line 4), and finally computes a hash of the canonical labelling to add to the component cache (line 5).

4.2 Other heuristics and optimizations

In this section, we discuss additional optimizations in SYMGANAK that are directed at improving the efficacy of symmetric component caching.

Bounded component analysis Modern model counters typically integrate some form of *conflict-driven clause learning*, recording failed search paths as conflict clauses that guide backtracking. SYMGANAK employs *bounded component analysis* (Sang et al. 2004), such that, the learned clauses are used to prune the search space but are not included in the cached component representation. This is similar to what is

done in prior model counters (like SHARPSAT and GANAK). However, note that even for "classical" component caching schemes, as employed in previous solvers, bounded component analysis is not trivial: Sang et al. (2004) showed that extra care must be taken when integrating component caching with clause learning. When exploring unsatisfiable parts of the search tree, model counts found for components under this part of the tree should be discarded from the cache, as reusing them may lead to incorrect results. Fortunately, as long as all components under a given assignment are satisfiable, the approach is sound: below, we prove that this result continues to hold for caching of symmetric components.

Lemma 1. *Let π be a partial assignment such that $F|_\pi$ is satisfiable, and let A be a component of $F|_\pi$, and $G|_\pi$ a set of learned clauses of F reduced by π. Then $|R_A| = |R_{A \wedge G|_\pi \downarrow vars(A)}|$.*

Proof. This lemma follows from 1 and 2 below which respectively prove that any projected model of $A \wedge G|_\pi$ is also a model of A, and vice versa.

1. Any model of $R_{A \wedge G|_\pi \downarrow vars(A)}$ is clearly a model of A (because $A \wedge G|_\pi$ implies A).
2. By Theorem 1 of Sang et al. (2004), any model ρ_A of A can be extended to a model of $F|_\pi \wedge G|_\pi$. Now since $A \wedge G|_\pi$ is implied by $F|_\pi \wedge G|_\pi$ (A is component of $F|_\pi$), ρ_A can also be extended to a model of $A \wedge G|_\pi$. Hence, it follows that for any model ρ_A of A we have $\rho_A \in R_{A \wedge G|_\pi \downarrow vars(A)}$.

□

Theorem 2. *Symmetric component caching, in combination with bounded component analysis and clause learning, still yields the correct model count as long as we remove all sibling components and their descendants from the cache when encountering an unsatisfiable component.*

Proof. Using clause learning, when encountering component A as part of a satisfiable formula $F|_\pi$, its model count will be computed as $|R_{A \wedge G|_\pi \downarrow vars(A)}|$. This is because the model count is computed using guidance from the learned clauses (which may contain variables not in $vars(A)$). In bounded component analysis, this value will be cached as the model count of component A: the soundness of this, even for symmetric component caching, is guaranteed by Lemma 1 (subject to pruning unsatisfiable siblings and their descendants). Any component B, symmetric to A, can safely reuse this value because $|R_A| = |R_B|$ by Theorem 1. □

Handling binary clauses The component encoding scheme used in CACHET, a precursor to SHARPSAT represents cached components as a combination of the unassigned variables and an identifier for each clause in the component (Sang et al. 2004). Thurley (2006) observed that the presence of binary clauses can be inferred from the presence of the unassigned variables, and therefore proposed a sound caching scheme that did not store the identifier corresponding to binary clauses. Interestingly, the arguments about soundness of Thurley's encoding scheme do not hold under symmetric caching scheme. Therefore, in a significant departure from SHARPSAT and its derivatives such as GANAK, we store the identifiers corresponding to the binary clauses. Fortunately, the probabilistic component caching scheme introduced in GANAK alleviates potential space efficiency drawbacks as our cache consists of the hashes of components. Designing an encoding scheme for symmetric components without encoding the identifiers corresponding to binary clauses is an interesting challenge for future work.

Hybrid thresholding Searching for a component in the SYMGANAK cache is computationally expensive as compared to previous caching schemes. Thus, there exists a delicate balance between the time spent on cache lookups and the gains from a cache hit. For this purpose, SYMGANAK employs the following scheme, which we call *hybrid thresholding*: we fix configurable parameters l and u (empirically determined), for the minimum and maximum number of variables a component must contain to be eligible for symmetric component caching. If the number of variables in a component lies outside of these bounds (either $|vars(\mathcal{C})| > u$ or $|vars(\mathcal{C})| < l$), SYMGANAK instead uses the traditional caching scheme of GANAK. This scheme is motivated by the following observations:

- the overhead of computing the canonical labeling for small components is often higher than simply solving these components from scratch;
- large components have a high cost of computing the canonical labeling and a small likelihood of obtaining a cache hit.

Hence, in both of the above cases, we ignore symmetry detection and resort to PCC (as used in GANAK) which is both fast and has high cache utilization.

Variable selection heuristics Along with packaging existing heuristics like VSADS (Sang, Beame, and Kautz 2005) and CSVSADS (Sharma et al. 2019), we introduce a novel variable selection heuristic, *Isomorphic Cache State and Variable State Aware Decaying Sum* (ICSVSADS), that is motivated by CSVSADS but is also *symmetry aware*. More concretely, whenever a cache hit occurs, we decrease the scores of all variables in that component, as well as the scores of all variables that have previously formed a component symmetric to it.

For example, in GANAK, when $(x \vee y \vee z)$ yields a cache hit under the CSVSADS heuristic, the score of x, y, and z is decreased to discourage branching on those variables in the future. If we were to branch on any of those variables in the future, it would be impossible to obtain a cache hit on the same component below that point in the search tree. We extend this idea to SYMGANAK with ICSVSADS, such that when $(x \vee y \vee z)$ is hit as a result of the symmetric component $(a \vee \neg b \vee c)$, we not only discourage branching on x, y and z, but also on a, b and c. The heuristic is otherwise identical to CSVSADS.

5 Experiments

We integrated the above proposed scheme on top of the existing state of the art model counter, GANAK[1]. We employ NAUTY (McKay and Piperno 2014) to perform computation of canonical labelling. It is worth remarking that SYMGANAK also provides an option to turn off PSCC, and thereby behave as a deterministic counter. We perform detailed empirical evaluation on a large suite of benchmarks arising from combinatorial instances, and the objective of our empirical study was to answer the following research questions:

RQ1 How do different variable branching heuristics impact the performance of SYMGANAK?

RQ2 How does the runtime performance of SYMGANAK compare with respect to the state of the art model counter GANAK?

Our empirical study leads to a surprising conclusion: First, we observe that VSADS heuristic achieves better runtime performance than the other alternate branching heuristics. We then observe that SYMGANAK significantly outperforms GANAK both in terms of PAR-2 score[2] and number of instances solved. Our results are in line with often observed behavior in the context of SAT solving: the choice of heuristics depend on the class of benchmarks. As pointed out in Section 1, combinatorial benchmarks not only serve as important challenging problems but improvements in automated reasoning has paved a way for discovery and proofs of challenging mathematical theorems. In this context, we expect our empirical study to motivate further studies on designing efficient counting schemes for combinatorial instances.

5.1 Implementation and experimental setup

We evaluated SYMGANAK on 212 instances from a wide range of combinatorial benchmark classes: Battleships, n-queens, grid Bayesian networks, k-colouring of grid graphs, quasigroup (Latin square) completion, FPGA switch-boxes, and logic synthesis, among several others.

We performed our experiments on a high-performance computer cluster, with each node having an Intel Xeon E5-2690 v3 CPU with 24 cores and 96GB of RAM. We used all 24 cores per node, with memory limit set to 4GB per core. Every instance, for each tool, was executed on a single core.

For GANAK and SYMGANAK, we set the default value of $\delta = 0.05$, a maximum component cache size of 2GB, and a timeout of 5000 seconds. For SYMGANAK, we empirically determined 10 and 250 to be good lower and upper bound values for hybrid thresholding (see Section 4.2). SYMGANAK (similar to GANAK) uses the *independent support* (IS) (Ivrii et al. 2016) of the formula to accelerate the search; due to cost considerations, IS is used only if fewer than 500 conflicts are detected after 500 000 decisions. We run both GANAK and SYMGANAK with this setting. All other parameters were set to their default values as in GANAK.

The cactus plots (Figures 1 and 2) show the number of instances solved (x-axis) by the respective tool in a given amount of time (y-axis); a point (x, y) on the plots represents that x benchmarks were solved by the counter in y seconds.

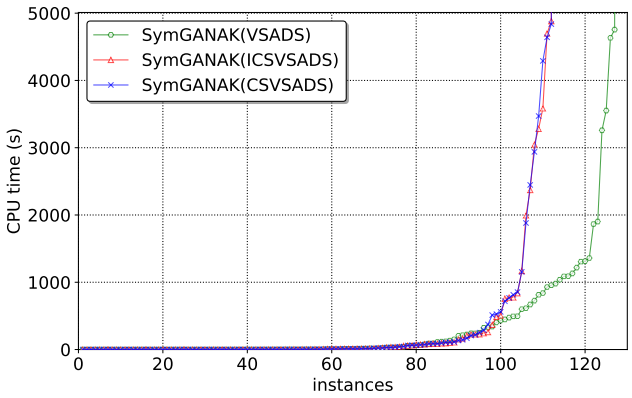


Figure 1: Cactus plot comparing different variable branching heuristics in SYMGANAK

5.2 Results

RQ1: Comparing branching heuristics Figure 1 compares the different branching heuristics available in SYMGANAK. We found it surprising that VSADS outperforms both the cache aware heuristics (CSVSADS and ICSVSADS). A detailed analysis of these heuristics on our set of benchmarks shows that these heuristics are incomparable: of all the instances, VSADS, CSVSADS and ICSVSADS was found to be the most effective heuristic in 34, 16 and 12 instances respectively; they had comparable performance[3] in 60 instances, while they all timed out for 85 instances. So, though VSADS is the dominant heuristic in 34 instances, one of the two cache aware heuristics emerges as the winner in 28 instances.

Among CSVSADS and ICSVSADS, our symmetry aware heuristic ICSVSADS has the same performance as CSVSADS. A deeper examination revealed that SYMGANAK (ICSVSADS) made fewer decisions on average than SYMGANAK (CSVSADS) (149 000 vs 181 000), suggesting that the gains of an improved cache performance was likely outweighed by the additional bookkeeping required to keep track of variables used in each component. Translating this improved cache performance to runtime improvements seems to be an interesting challenge for future work.

RQ2: Impact of the symmetric component cache (SYMGANAK versus GANAK) As the VSADS branching heuristic performs the best for both SYMGANAK and GANAK on our benchmarks, we compare both of these tools with VSADS. Figure 2 shows that SYMGANAK outperforms GANAK: while SYMGANAK solves 16 more instances and

[1] In the recent *1st International Competition on Model Counting* (https://mccompetition.org), the entry built on GANAK won the model counting track.

[2] The PAR-2 score (penalized average runtime), as used in the SAT 2018 Competition (Heule, Järvisalo, and Suda 2019), is the average runtime assigning a runtime of two times the time limit (instead of a "not solved" status) for each unsolved benchmark.

[3] Finishing within one second of each other.

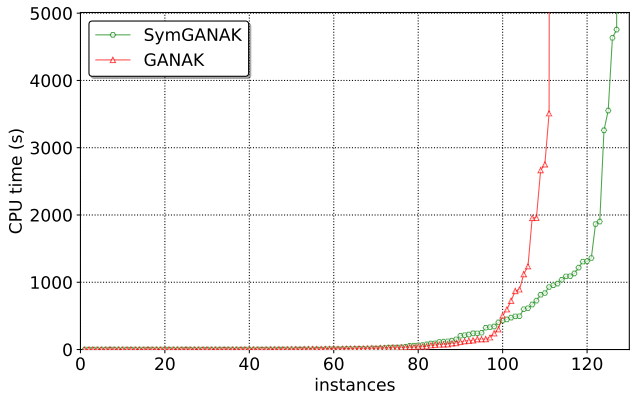


Figure 2: Cactus plot comparing SymGanak and Ganak

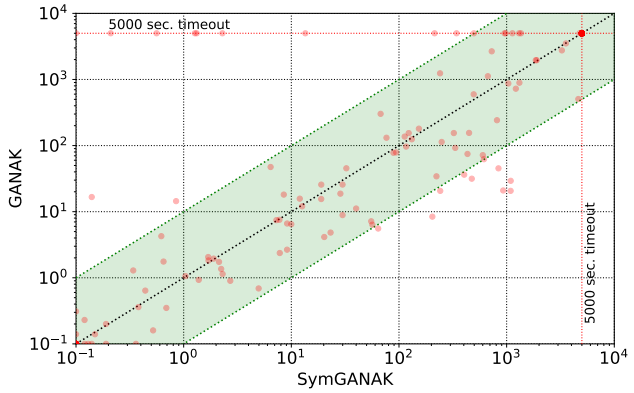


Figure 3: Scatter plot comparing SymGanak and Ganak

achieves a lower PAR-2 score of $0.87\times$ that of Ganak, there was only a single instance solved by Ganak that timed out on SymGanak. Figure 3 shows a scatter plot comparing their runtime on individual instances.

The performance of SymGanak can be attributed to the fact that by exploiting symmetries, SymGanak is able to obtain both a higher number of cache hits as well as cache hits on larger components[4]. Over all instances solved by both Ganak and SymGanak, the average component size of each cache hit was 79.7 for SymGanak and only 58.2 for Ganak; the mean number of decisions made by SymGanak was approximately 302 000, compared to 1.4 million for Ganak.

To understand the results above in greater detail, Figure 4 shows a detailed distribution of cache hits for a representative n-queens instance: SymGanak has component cache hits with over 100 variables, about an order of magnitude larger than the largest components hit by Ganak. Even on the smaller components, SymGanak manages to obtain substantially more cache hits than Ganak.

[4]We define the size of a component as the number of variables appearing in it.

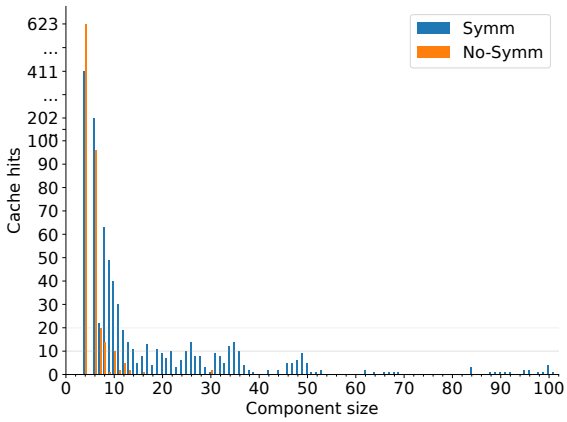


Figure 4: Cache hit distribution for an n-queens instance ($n = 12$, 144 variables). For each component size (x-axis), the number of cache hits for components of that size (y-axis). The figure was generated using the CSVSADS heuristics, turning off both random restarts and hybrid thresholding for easier analysis.

6 Conclusion

We investigated the effect of caching symmetric components in #DPLL-based model counting algorithms. To evaluate our approach, we implemented the concept into a new counter SymGanak, an extension of Ganak, and compared their performance.

While detection of symmetries comes with a computational cost, we showed that detecting larger components more often can reduce the overall time needed to solve them, as illustrated by a reduced PAR-2 score and a greater number of benchmarks solved by SymGanak. This opens the door to further research in faster methods for detecting symmetric components. We also evaluated the performance of SymGanak under several variable selection heuristics. While we made some first steps in identifying a novel variable selection heuristic (ICSVSADS) that could work well with symmetric component caching, improving this remains an open problem for future research. In future work we would also investigate the extension of our approach to weighted counting.

While our choice of Ganak as the base tool was in line with the typical practice in SAT community where improvements are shown on top of winning solvers of recent years, it would be interesting to pursue integration of symmetric component caching in other state-of-the-art model counting systems such as D4 (Lagniez and Marquis 2017).

Acknowledgements

TvB is supported by the Research Foundation – Flanders [G095917N]. VD is supported by an SB PhD fellowship at the Research Foundation – Flanders [1SA5520N]. This work was supported in part by the National Research Foundation Singapore under its NRF Fellowship Programme [NRF-NRFFAI1-2019-0004] and the AI Singapore Programme [AISG-RP-2018-005], NUS ODPRT Grant [R-252-000-685-13]. The computational work for this study was performed on

resources of the National Supercomputing Centre, Singapore (https://www.nscc.sg). The authors thank Luc De Raedt for his involvement in the early stages of this project.

References

Aloul, F. A.; Ramani, A.; Markov, I. L.; and Sakallah, K. A. 2002. Solving difficult SAT instances in the presence of symmetry. In *DAC*, 731–736. ACM.

Aziz, R. A.; Chu, G.; Muise, C. J.; and Stuckey, P. J. 2015. #∃SAT: Projected Model Counting. In *SAT*, volume 9340 of *Lecture Notes in Computer Science*, 121–137. Springer.

Bacchus, F.; Dalmao, S.; and Pitassi, T. 2003. DPLL with Caching: A new algorithm for #SAT and Bayesian Inference. *Electron. Colloquium Comput. Complex.* 10(003).

Baluta, T.; Shen, S.; Shinde, S.; Meel, K. S.; and Saxena, P. 2019. Quantitative Verification of Neural Networks and Its Security Applications. In *CCS*, 1249–1264. ACM.

Bart, A.; Koriche, F.; Lagniez, J.; and Marquis, P. 2014. Symmetry-Driven Decision Diagrams for Knowledge Compilation. In *ECAI*, volume 263 of *Frontiers in Artificial Intelligence and Applications*, 51–56. IOS Press.

Bayardo Jr, R. J.; and Pehoushek, J. D. 2000. Counting models using connected components. In *AAAI/IAAI*, 157–162.

Birnbaum, E.; and Lozinskii, E. L. 1999. The Good Old Davis-Putnam Procedure Helps Counting Models. *J. Artif. Intell. Res.* 10: 457–477.

Brakensiek, J.; Heule, M.; Mackey, J.; and Narváez, D. 2020. The Resolution of Keller's Conjecture. In *IJCAR (1)*, volume 12166 of *Lecture Notes in Computer Science*, 48–65. Springer.

Chavira, M.; and Darwiche, A. 2008. On probabilistic inference by weighted model counting. *Artif. Intell.* 172(6-7): 772–799.

Crawford, J. M.; Ginsberg, M. L.; Luks, E. M.; and Roy, A. 1996. Symmetry-Breaking Predicates for Search Problems. In *KR*, 148–159. Morgan Kaufmann.

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

Devriendt, J.; Bogaerts, B.; Bruynooghe, M.; and Denecker, M. 2016. Improved Static Symmetry Breaking for SAT. In *SAT*, volume 9710 of *Lecture Notes in Computer Science*, 104–122. Springer.

Gomes, C. P.; and Shmoys, D. 2002. Completing Quasigroups or Latin Squares: A Structured Graph Coloring Problem.

Heule, M. J. H.; Järvisalo, M.; and Suda, M. 2019. SAT Competition 2018. *J. Satisf. Boolean Model. Comput.* 11(1): 133–154.

Heule, M. J. H.; Kullmann, O.; and Marek, V. W. 2016. Solving and Verifying the Boolean Pythagorean Triples Problem via Cube-and-Conquer. In *SAT*, volume 9710 of *Lecture Notes in Computer Science*, 228–245. Springer.

Ivrii, A.; Malik, S.; Meel, K. S.; and Vardi, M. Y. 2016. On computing minimal independent support and its applications to sampling and counting. *Constraints* 21(1): 41–58.

Kitching, M.; and Bacchus, F. 2007. Symmetric Component Caching. In *IJCAI*, 118–124.

Lagniez, J.; and Marquis, P. 2017. An Improved Decision-DNNF Compiler. In *IJCAI*, 667–673.

Lauria, M.; Elffers, J.; Nordström, J.; and Vinyals, M. 2017. CNFgen: A Generator of Crafted Benchmarks. In *SAT*, volume 10491 of *Lecture Notes in Computer Science*, 464–473. Springer.

Lemire, D.; and Kaser, O. 2016. Faster 64-bit universal hashing using carry-less multiplications. *J. Cryptogr. Eng.* 6(3): 171–185.

McKay, B. D.; and Piperno, A. 2014. Practical graph isomorphism, II. *J. Symb. Comput.* 60: 94–112.

Oztok, U.; and Darwiche, A. 2015. A Top-Down Compiler for Sentential Decision Diagrams. In *IJCAI*, 3141–3148. AAAI Press.

Sabharwal, A. 2009. SymChaff: exploiting symmetry in a structure-aware satisfiability solver. *Constraints* 14(4): 478–505.

Sang, T.; Bacchus, F.; Beame, P.; Kautz, H. A.; and Pitassi, T. 2004. Combining Component Caching and Clause Learning for Effective Model Counting. In *SAT*.

Sang, T.; Beame, P.; and Kautz, H. A. 2005. Heuristics for Fast Exact Model Counting. In *SAT*, volume 3569 of *Lecture Notes in Computer Science*, 226–240. Springer.

Sashittal, P.; and El-Kebir, M. 2020. Sampling and summarizing transmission trees with multi-strain infections. *Bioinform.* 36(Supplement-1): i362–i370.

Schweitzer, P. 2009. *Problems of Unknown Complexity: Graph isomorphism and Ramsey theoretic numbers*. Ph.D. thesis, Saarland University.

Sharma, S.; Roy, S.; Soos, M.; and Meel, K. S. 2019. GANAK: A Scalable Probabilistic Exact Model Counter. In *IJCAI*, 1169–1176. ijcai.org.

Soos, M.; and Meel, K. S. 2019. BIRD: Engineering an Efficient CNF-XOR SAT Solver and Its Applications to Approximate Model Counting. In *AAAI*, 1592–1599. AAAI Press.

Stockmeyer, L. J. 1983. The Complexity of Approximate Counting. In *STOC*, 118–126. ACM.

Thurley, M. 2006. sharpSAT - Counting Models with Advanced Component Caching and Implicit BCP. In *SAT*, volume 4121 of *Lecture Notes in Computer Science*, 424–429. Springer.

Toda, S. 1991. PP is as Hard as the Polynomial-Time Hierarchy. *SIAM J. Comput.* 20(5): 865–877.

Valiant, L. G. 1979. The Complexity of Enumeration and Reliability Problems. *SIAM J. Comput.* 410–421.

Wang, W.; Usman, M.; Almaawi, A.; Wang, K.; Meel, K. S.; and Khurshid, S. 2020. A Study of Symmetry Breaking Predicates and Model Counting. In *TACAS (1)*, volume 12078 of *Lecture Notes in Computer Science*, 115–134. Springer.

Yang, S. 1991. Logic Synthesis and Optimization Benchmarks User Guide: Version 3.0. Technical report, MCNC Technical Report.