Universiteit Leiden

Master Computer Science

Caching in Model Counters: A Journey through Space and Time

Name:	Jeroen G. Rook
Student number:	s1265091
Date:	[02/08/2021]
Specialisation:	Computer Science and Advanced Data Analytics
1st supervisor:	Prof. dr. Holger H. Hoos
2nd supervisor:	Anna L.D. Latour MSc.
2nd reader:	Prof. dr. Siegfried Nijssen

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

Model counting is the task of counting all distinct truth assignments for a propositional Boolean formula and can solve many important AI problems like Bayesian inference and software verification problems. One way to solve these problems is by using DPLL-style algorithms with component caching. We investigate the role of component caching in these solvers and how the available memory for the cache impacts solving instances. We find that the size of the cache does not influence the performance much, which is attributed to the characteristics of good branching heuristics. These characteristics produce a small search tree and favour components that are only encountered in small parts of the tree. Finally, we show that future component encounters are predictable and demonstrate that predicting this component relevance can improve existing caching schemes. To visualise the cache interactions with the solver under different circumstances and for learning how to predict component relevance, we take a machine learning approach that collects characteristics of components and the moments these components interact with the solver.

Contents

Abstract **i**

Acknowledgement 1

1 Introduction **2**

2 Background **5**

2.1 Notation and Preliminaries 5

2.2 Complexity of model counting 6

2.3 Model counting variants 7

2.3.1 Weighted model counting 7

2.3.2 Projected model counting 8

2.3.3 Approximate Model Counting 8

2.4 Summary 9

3 DPLL-style model counting methods **10**

3.1 DPLL-style algorithms 11

3.2 Component decomposition 12

3.3 Clause Learning 13

3.4 Component Caching 13

3.4.1 Combining caching with clause learning 14

3.4.2 Component encoding 15

3.4.3 Caching schemes 15

3.5 Branching heuristics 17

3.5.1 Combining heuristics 18

3.6 Model counters overview 20

4 Approach **22**

4.1 Pipeline 22

4.2 Component mining 23

4.2.1 Features 23

4.2.2 Component structure features . . . 23
4.2.3 Search-state features . . . 25
4.2.4 Measuring component relevance . . . 26
4.2.5 Sampling . . . 27
4.3 Classification . . . 27
4.4 Automated algorithm configuration . . . 28
4.5 Extending GANAK . . . 28
4.5.1 Configurability . . . 28
4.5.2 Data collection . . . 29
4.5.3 Classifier-based caching schemes . . . 29

5 Experiments **31**
5.1 Experimental setup . . . 31
5.1.1 Hardware and software . . . 31
5.1.2 Benchmark sets . . . 32
5.1.3 Configuration . . . 33
5.1.4 Classification . . . 33
5.1.5 Classification embedding . . . 34
5.2 Trade-off between memory and solving time . . . 34
5.3 Influence of branching heuristic on cache behaviour . . . 35
5.4 Influence of caching schemes on cache behaviour . . . 39
5.5 Predictability of component relevance . . . 41
5.6 Online classifier performance . . . 43

6 Conclusions and future work **44**
6.1 Future work . . . 45

Bibliography **46**

A Parameter Importance Table. **52**

Acknowledgements

I would like to express my gratitude towards everyone who was directly and indirectly involved towards the completion of this thesis. A special thanks goes out to Anna. Her involvement in the project and me as a person made it for me a really meaningful learning experience in both academic hard and soft skills. Also, the encouragements and positive inputs during challenging moments during the project is something I appreciate deeply. I want to thank Holger and Siegfried for their advice, mentoring and sharp insights. Further, I want to thank all members of the ADA research group for adopting me into their research group, and all the help and advice I got from them. Of course, also thank you for the games of Hanabi, Code Names and Skribble during our social gatherings. At last, I want to thank all the students and staff of LIACS with whom I was involved during my time at the institute.

Chapter 1

Introduction

Model counting is the task of counting the number of distinct satisfying truth assignments for a given Boolean propositional formula. Also known as #SAT; it is one of the most prominent #$\mathcal{P}$-complete [73,74] problems within artificial intelligence and serves numerous real-world applications, such as probabilistic inference [16, 55], software verification [7, 50], computational biology [62], and network reliability planning [24, 41]. The model counting problem is #$\mathcal{P}$-complete [73, 74], which means that within the polynomial complexity hierarchy [66] it is likely to be harder to solve than $\mathcal{NP}$-complete problems, such as SAT.

The last two decades witnessed many innovations in solving #SAT. Along these innovations are model counting algorithms [2, 5, 8] that are based on the Davis-Putmann-Logemann-Loveland (DPLL) algorithm [22, 23]. These DPLL-style solvers are *divide-and-conquer* algorithms, which divide the formula into smaller subformulae that are referred to as *components* and have become increasingly powerful due to *conflict-driven clause learning* (CDCL) [59], component decomposition [3], and component caching [2]. Further efforts on improving these algorithms focused on developing new variable branching heuristics [9, 63, 69] and on more efficient [11, 69] and probabilistic [63] ways for encoding components.

Other lines of research studied preprocessing [44, 45], enumeration [70], approximation [13, 32, 33], bounding [34], knowledge compilation [20, 21, 43] and parallel model counting techniques [28]. In contrast to this work, we focus on exact DPLL-style model counters, such as Cachet [59], sharpSAT [69], and specifically, GANAK; which is the current state of the art solver.

To the best of our knowledge, this work performs the first thorough investigation on the role that component caching plays in DPLL-style model counters. Component caching stores solved components along with the resulting model count. Once the same component is encountered later in the search, the stored model count is used instead of recomputing it. That way less computations are needed in solving the instances. In a way, this memoisation of components makes model counters with components

caching *dynamic programming* algorithms.

The caching of components requires memory, which is a limited resource. Therefore, model counters need to manage the cached counts to remain within the memory limits. Having more memory available for the cache, means that more component counts can be kept in the cache at the same moment. It is intuitive to assume that by having more model counts cached at once can result in quicker solving times because there is a higher chance that a component's count is in the cache. This would suggest that having an as large as possible cache results in that a problem instance is likely to be solved more quickly. We also note that time is a somewhat soft constraint, whereas memory is a harder constraint: after all, it is easier to let the computer run longer than to install extra memory. This motivates the efforts towards developing more efficient component cache storing methods over the last 15 years [11,63,69]. We empirically show that this intuition only sometimes aligns with reality. Specifically we investigate the following research questions:

Q1 How do running times of a DPLL-based #SAT solver depend on the maximum allowed cache size?

Q2 Why do smaller cache sizes not necessarily decrease solving times?

Q3 How do branching heuristics influence cache usage?

Q4 How do caching scheme influence cache usage

Q5 To what degree can it be predicted how beneficial it is to the solver to keep the model count of a component stored in the cache?

Q6 How can component relevance predictions be leveraged in DPLL-based #SAT solvers to solve with less decision nodes?

#SAT solvers have many parameters that may influence the running time of the solver. To focus our attention on the impact of one feature, such as cache size, in this work we propose to use *automated algorithm configuration (AAC)* [37] to determine the optimal value for the other parameters of these solvers. This allows us to evaluate how we can minimise running time for different maximum cache sizes, and to gain insights on how the cache is being used by the solver under different circumstances.

We then take inspiration from earlier work that used a supervised machine learning approach to optimise the use of learnt clauses in SAT solvers [65], work that used reinforcement learning to learn new branching heuristics for #SAT solvers [72], and work that used machine learning to improve cache management heuristics to increase the number of cache hits in a generic caching problem [47]. Specifically, we employ supervised machine learning techniques similar to those used in CrystalBall [65] to predict the 'relevance' of a component count, such that we can determine if it should be stored in the cache or discarded. We also demonstrate that this can be applied in practice into these solvers. Further, we argue that our findings represent a promising first step towards designing new, cheap-to-compute cache management heuristics.

The remainder of this thesis is structured as follows. In Chapter 2, we introduce important concepts, notation, and model counting variants. Next, in Chapter 3, we give an detailed overview of DPLL-style model counters and how they work. Then, in Chapter 4, we set out our approach and explain how we extended GANAK. In Chapter 5, we present our experimental results and answer our research questions. We end with Chapter 6 where we make some general conclusions and give directions for future work.

Chapter 2

Background

In this chapter, we discuss preliminaries and how model counting relates to other fundamental problems within computer science. We discuss different the variants of model counting; weighted model counting and projected model counting, that are able to handle other problem representations.

2.1 Notation and Preliminaries

Our notations, definitions and preliminaries are aligned with the ones found elsewhere in literature. However, in order to have a common basis, we recite them here.

A propositional formula $\phi(V)$ consists of propositional variables and logical operators. Each variable can be `true`, `false` or remain unassigned. The logical operators we use are negation ($\neg$), conjunction ($\wedge$) and, disjunction ($\vee$). These three operators are together a functionally complete set [57]. Meaning that, all possible truth tables can be expressed as a Boolean formula by only using these operators. Other logical operators can be rewritten into a form that only uses the three aforementioned operators. For example, implication ($\rightarrow$) in $p \rightarrow q$ can be rewritten to $\neg(p \wedge q)$ while remaining semantically equivalent [39].

Propositional formulas can even be stricter defined with a syntactic structure, which is referred to as *language* [19, 39]. The structural properties of these languages allow for a more systematic approach in finding solutions for problems on a formula.

Let $x \in V$ represent a variable from the set of all distinct variables from formula ϕ. We define a **literal** l as a variable (x) or its negation ($\neg x$) and a **clause** c as a set of literals that are joined with disjunctions wherein each variable in V can occur at most once.

We define Conjunctive Normal Form (CNF) as a set of clauses combined with conjunctions. Equation

2.1 is an example of a formula in CNF and has 3 variables, 8 literals, and 4 clauses:

$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge (x_1) \tag{2.1}$$

Individual clauses and variables are referred to as c_j and x_i, with $1 \leq j \leq |\phi|$ and $1 \leq i \leq |V|$, respectively. An assignment σ is a mapping of the variables in V accompanied with a truth assignment: $\sigma : V \mapsto \{\texttt{true}, \texttt{false}\}$. This is notated in a form of: $\{x_1, \neg x_2, \dots\}$, which translates to an assignment of `true` for x_1 and `false` for x_2. Note that a literal $\neg x_2$ in ϕ with this assignment is equal to `true`. When $\phi|_\sigma$ evaluates to `true`, σ is a *model*, *witness* or *satisfying assignment* of ϕ. To solve the model counting problem we need to count all models for ϕ.

Generally, ϕ can be any propositional formula but in this work, we only count the number of models on propositional formulae in CNF. For formulae in CNF an assignment σ is a model when all its clauses have at least one literal that evaluates to `true`.

We use $\phi|_\pi$ to denote the *residual formula* obtained after substituting all variables v from a subset $W \subset V$ with the values under the partial truth assignment $\pi : W \mapsto \{\texttt{true}, \texttt{false}\}$ and simplifying the resulting formula. Given a partial assignment π, if $\phi|_\pi = \texttt{true}$, any extension of π to a full assignment σ is a model of ϕ. If we can write $\phi|_\pi = \bigwedge_i \phi_i(V_i)$ with $V_i \bigcap V_j = \emptyset$ for all pairs of distinct i and j, we call the subformulae ϕ_i disjoint components of $\phi|_\pi$. With these definitions we can make the following observations about $mc(\phi)$, the number of models (or model count) of a formula ϕ. Firstly, an unsatisfiable formula has a model count of 0. Secondly, when $\phi|_\pi = \texttt{true}$ then there are $2^{|V/\pi|}$ models for $\phi|_\pi$. Thirdly, $mc(\phi|_\pi) = mc(\phi|_{\pi \cup \{v\}}) + mc(\phi|_{\pi \cup \{\neg v\}})$ with a variable v in $\phi|_\pi$. Finally, if $\phi|_\pi$ has disjoint components, then $mc(\phi|_\pi) = \prod_i mc(\phi_i)$

2.2 Complexity of model counting

The model counting problem belongs to the complexity class $\#\mathcal{P}$ which was introduced by Valiant [73,74]. To the class $\#\mathcal{P}$ belong problems that count the number of solutions of $\mathcal{NP}$-complete problems. However, the problems in $\#\mathcal{P}$ are not restricted to the latter. Problems belonging to class $\mathcal{P}$ can have a corresponding counting problem that is in $\#\mathcal{P}$, such as counting the number of cycles in a graph or counting all satisfying truth assignments for a formula in Disjunctive Normal Form (DNF). Similar to problems in $\mathcal{NP}$ they can be *hard* or *complete*. A problem is $\#\mathcal{P}$-complete when it is in $\#\mathcal{P}$ and for every problem in $\#\mathcal{P}$ there exists a polynomial counting reduction from one to another.

Valiant showed that counting the number of satisfiable assignments of a propositional formula in CNF is $\#\mathcal{P}$-complete [74]. More surprising is that the counting problem for formulas in DNF and CNFs with only Horn clauses are also $\#\mathcal{P}$-complete [18], while the complexity class for solving the SAT problem on these instances belongs to $\mathcal{P}$. For clarity, Horn clauses have one negated literal and the other literals are

positive. Horn clauses can be seen as an implication and that property is used to solve SAT for these formulas in a time complexity $\mathcal{P}$.

$\mathcal{NP}$-complete problems are at least as hard to solve as problems of the class $\mathcal{P}$, unless $\mathcal{P} = \mathcal{NP}$. This also hold for problems that are $\#\mathcal{P}$-complete compared to $\mathcal{NP}$-complete problems. Although, even if $\mathcal{P} = \mathcal{NP}$, $\#\mathcal{P}$ is probably still harder to solve. Only if $\mathcal{P} = \mathcal{PP}$ is proven, $\#\mathcal{P}$ can be solved in polynomial time.

Model Counting is the canonical $\#\mathcal{P}$-complete problem and is an interesting test-bench to test computational methods to solve problems that belong to this class. This is similar as to SAT solving is used for solving problems that are in $\mathcal{NP}$-complete. Advances on the efficiency of solving these problems translate to the ability of solving larger problems for all other problems within the same complexity class.

Besides CNF, there are other propositional languages that can be used to solve the model counting problems. With each language having their own set of rules and logical operators. For example, DNF uses the same set of operators, but opposed to CNF it joins sets of literals with conjunctions and combines these with disjunctions. Languages are not mutually exclusive, meaning that a formula can belong to multiple languages at once. Furthermore, languages can be a subset of another language by having stricter rules and/or operator sets. Both CNF and DNF are inherited from Negation Normal Form (NNF), where the only rule is that negations are only allowed on variables. Formulas can be rewritten/compiled to each other and is referred to as *knowledge compilation* [21].

The time complexities for answering different problems on formulas differs for each language. For some languages, the model count can be computed in polynomial time. Unfortunately, there is no free lunch since compiling a formula to another language is often not a procedure that is done in polynomial time and can even become harder to solve since the resulting formula can experience exponential blow-up.

2.3 Model counting variants

Besides counting the exact number of models for a given formula, there are several variants to the model counting problem. We briefly discuss three of those variants. First, we discuss weighted and projected model counting problems, which allow for more efficient problem encodings. At last we discuss approximate model counting, which weakens the strictness of the model counting task.

2.3.1 Weighted model counting

Analogous to model counting, Bayesian inference is $\#\mathcal{P}$-complete [58]. An encoding that transforms Bayesian networks to CNF's [19] made it possible to solve these using model counters. There are some limitations to this encoding. Firstly, the reduction produces exponentially large formulas. Secondly, the encoding is limited to cover a discrete set of probabilities found in the Bayesian networks. In order

to overcome these limitations a weighted variant of model counting was introduced called Weighted Model Counting (WMC) [61].

The maximum number of models a formula can have is $2^{|V|}$. Besides explicitly counting the models of a formula ϕ, it can also be represented as the probability that any given full assignment will satisfy the formula. In exact model counting the truth assignment for each variable is chosen uniformly at random. Other probability distributions can be assigned to the truth assignments, which make it weighted. A weight function $W(\cdot)$ where each variable has two mappings to a corresponding weight; one for each of the truth value; `true` and `false`. For a variable x the weights are either *normal weighted* $W(x) + W(\neg x) = 1$ with $0 \leq W(x) \leq 1$, or they are *indifferent* $W(x) = W(\neg x) = 1$. The latter is especially useful to represent state variables in Bayesian networks [12]. Weighted model counting does not limit itself to only relying on probabilities, but for any truth value of a variable any weight can be assigned to it.

Weighted model counting does not change the time complexity for solving the problem and is merely a variation. However, it allows for more efficient encoding of problems without compromises as was demonstrated with probabilistic inference problems [61].

2.3.2 Projected model counting

Projected Model Counting (PMC) [1, 50] allows for more efficient encodings of reduced problems. Currently, all models are counted over all variables in the formula. This restricts model counting to encodings which do not hold any variables that are not functionally defined on the variables of the original problem. Functionally defined variables do not carry information with regard to the model count, but they can be beneficial for an efficient encoding of the problem. For example, Tseitin transformations [71] use such variables in order to produce efficient formulas in CNF.

Let $P \subseteq V$ be the set that represents the variables which we want to count the number of different models on. That is, to count different assignments the variables P can produce within the context of the whole formula. A projected model count is a number between 0 and $2^{|P|}$.

2.3.3 Approximate Model Counting

Currently we only considered the model counting problem to count the exact number of satisfiable assignments. Due to the time complexity of this task, the scalability of instances that can be solved within reasonable time is limited. Instead of finding more computational efficient methods that can handle larger instances, approximation methods can be used [13,75]. These methods do not compute the exact count, but instead try to find a count which is close to the exact count. This alleviates the problem and requires less computations. It was long believed that approximation is in theory not harder than $\mathcal{NP}$-hard problems [58,67]. More recently, these claims have been altered by Chakraborty et al. [15] by showing that this is only holds in particular cases. There are two main types of approximation; with and

without guarantees. Every approximated count M' can have a *quality* based how close it is to the exact count, and a *confidence* based the correctness. A lower bound of 0 can be given with a confidence of 100% (guarantee), but this bound can be very far off from the exact count (low quality). With guarantees, you want to guarantee that the approximated count is within a certain window around the exact count M. Let the allowed error (quality) be denoted as $0 < \delta \leq 1$ then the approximated count M' should be between:

$$M/(1+\delta) \leq M' \leq M \cdot (1+\delta) \tag{2.2}$$

Here, $M/(1+\delta)$ is the lower bound of M and $\leq M \cdot (1+\delta)$is the upper bound [42].

Some applications that rely on model counting, such as probabilistic reasoning, can perform well with an approximation that has certain guarantees [17]. However, there are many other applications that do rely on an exact count. In this work we will refrain from approximated model counting and focus on the exact variants.

2.4 Summary

The model counting problem with all its variants is able to solve many different problems within the class $\#\mathcal{P}$-complete. All these problems have a reduction where the problem is encoded to a propositional language. Often this language is in CNF and we only study these instances. There can be different encodings for a given a problem and they can greatly influence the difficulty of solving the problem. That is why variants such a weighted model counting, projected model counting or a combination of both can be very helpful. As seen with probabilistic inference, variants allow problem encodings where no compromises need to be made.

Chapter 3

DPLL-style model counting methods

In the last two decades model counters have become increasingly good at solving #SAT. Many of these solvers share a common design paradigm; the Davis-Putman-Logemann-Loveland (DPLL) algorithm [22, 23]. The DPLL algorithm was originally proposed for solving SAT on propositional formulae in CNF. This chapter gives an overview of how the DPLL algorithm was modified in order to be able to solve #SAT. We then describe how mechanisms such as component decomposition, clause learning, component caching, and different branching heuristics work and how they are implemented into solvers. Afterwards, we make a comparison between different model counters to emphasise their differences and similarities.

Figure 3.1 provides a visual overview of the search tree that DPLL-based algorithms produce and which steps are taken at each branch of the search tree.

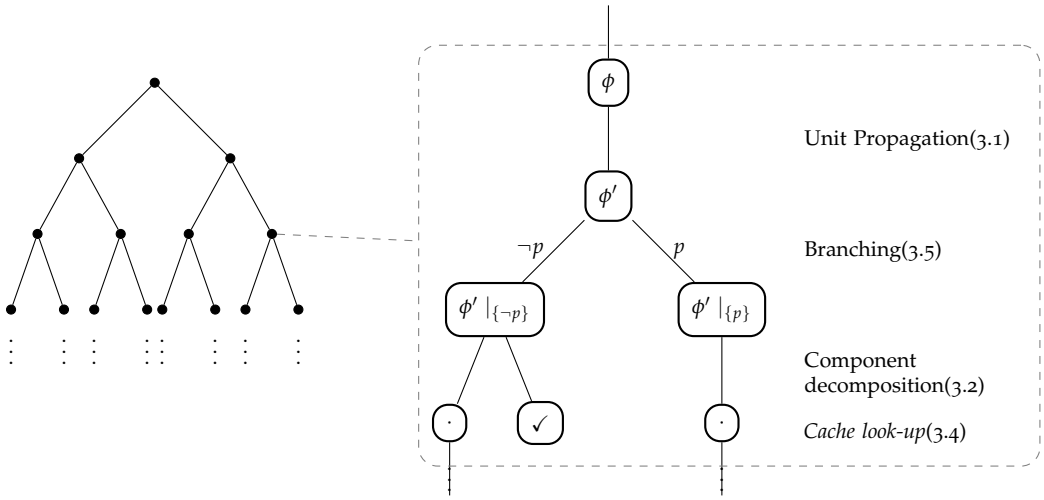


Figure 3.1: Diagram of the DPLL algorithm with caching for model counting. The procedure (right) is performed at each node in the search tree (left). A component is only solved when it is not found in the cache. Otherwise it returns the count from the cache (✓). Relevant sections for parts are given within parenthesis.

3.1 DPLL-style algorithms

We first give a brief description on how the DPLL algorithm [22, 23] works for SAT. The DPLL algorithm checks satisfiability of a formula by recursively selecting variables and assigning truth values to those variables in the formula. Each value assignment produces a residual formula by removing clauses that are satisfied and removing literals that become falsified from the other clauses. The algorithm repeats itself until the formula is empty or until there is an empty clause. An empty formula means that all clauses are satisfied and an empty clause means that the clause, and therefore the formula, cannot become satisfied anymore. When the partial assignment of variables is not able to satisfy the formula, the DPLL algorithm backtracks to the closest branch in the search tree where it did not previously assigned the opposite truth value to that variable. The algorithm continues until a satisfying assignment is found or when all branches in the search tree visited both truth values that could be assigned, which means that there is no solution that satisfies the formula. When a partial set of assigned variables satisfies the formula, it does not matter what the value of other variables is.

To further simplify the residual formula after assigning a truth value to a variable, the DPLL algorithm leverages two properties of CNF:

Pure literal elimination When the (residual) formula has a literal l without any negation $\neg l$ or vice versa. All clauses containing that literal can be deleted from the formula, since these clauses can always be satisfied without the chance that other clauses will become unsatisfiable.

Unit clause propagation (UP) All clauses with only one literal (*unit clauses*) can only be satisfied if the corresponding variable of that literal is given a truth value that satisfies the clause. Therefore only that truth value needs to be assigned the variable, as assigning the opposing truth value will always result in an empty clause.

One of the first algorithms used for solving #SAT is based on the DPLL algorithm. Birnbaum and Lozinskii [8] proposed a modified version of the DPLL algorithm named *Counting Davis Putman* (CDP). The psuedo code of is found in Algorithm 1. With three modifications to the original DPLL algorithm the CDP algorithm is able to count the number of models instead of checking satisfiability. These modifications are:

1. Once a satisfying assignment is found, this means that there is at least one model. Unassigned variables can be given any polarity, meaning that there are multiple models. Let π be the (partial) assignment of variables V in ϕ. If the residual formula of $\phi|_\pi$ is empty, then there are $2^{|V|-|\pi|}$ models for $\phi|_\pi$.

2. The original DPLL algorithm terminates once a satisfying assignment is found. For model counting the algorithm must continue to find all satisfying assignments. That means that for each variable that is branched on the model counts for both truth values must be computed. The model counts of each of the resulting residual formulae are summed together, as is explained in Section 2.1.

3. Pure literal elimination is removed. Clauses with pure literals can be satisfied without impacting other clauses. This should help in finding a solution quicker. However, the residual formula from the opposing value can still hold solutions, which makes pure literal an unwanted optimisation.

Algorithm 1: Counting David-Putman (CDP)

Input: A set of clauses ϕ, with n variables
Output: $mc(\phi)$
Function *CDP(ϕ,n)*
  if ϕ *is empty* **then**
    return 2^n
  if ϕ *has empty clause* **then**
    return 0
  if ϕ *has unit clause (l)* **then**
    return CDP($\phi|_{\{l\}}$, $n-1$) *#Unit Propagation (UP)*
  choose v from the variables in ϕ *#Branching*
  return CDP($\phi|_{\{\neg v\}}$, $n-1$) + CDP($\phi|_{\{v\}}$, $n-1$)

The order in which the variables are chosen can have a high impact on the performance of a counter [8,60] and are guided by heuristics. We discuss different heuristics used in model counters in more detail in Section 3.5. The CDP algorithm is the first DPLL-style model counter and serves as a base where the mechanisms, we discuss in this chapter, are build upon.

3.2 Component decomposition

We define a component as a set of clauses in ϕ that do not contain variables that appear in other clauses of ϕ. A formula can consist out of multiple components that are disjoint from each other: $\bigwedge_i \phi_i(V_i)$, with $V_i \bigcap V_j = \emptyset$. Logically, a formula can consist of only one component.

Components can be solved independently and since formulas with a small number of variables tend to be easier to solve than ones with a large number of variables, this should reduce the overall effort to solve the complete formula. Beyardo et al. [3] extended the CDP algorithm with this principle and named it *Decomposing Davis Putnam* (DDP), but is often referred to as RelSAT, since it was implemented in this SAT solver [4]. Recall that the model count of a residual formula with multiple disjoint components is obtained by taking the product of the component's model counts (Section 2.1).

The order in which components are solved can affect performance. For example, when a formula has an unsatisfiable component, all the model counts of other components become irrelevant, because the component counts are multiplied. Therefore, RelSAT solves components that are more constraint first, since a higher constrained formula is more likely to be unsatisfiable. The level constrainedness of a formula ϕ can be expressed by how many distinct variable pairs occur together in the clauses of ϕ [3]. Additionally, RelSAT checks if each component of a formula is SAT before it starts with model counting.

Although the DDP algorithm is represented as a recursive algorithm in Algorithm 2, it is implemented in an iterative manner. This allows to change the order in which the search is performed to become a

Algorithm 2: Decomposing-David-Putman (DDP)

Input: A set of clauses ϕ, with n variables
Output: $mc(\phi)$
Function *DDP(ϕ,n)*
  if ϕ *is empty* **then**
    return 2^n
  if ϕ *has empty clause* **then**
    return 0
  if ϕ *has unit clause* l **then**
    return $\text{DDP}(\phi|_{\{l\}}, n-1)$ *#UP*
  for *disjoint component* ϕ_i *in* ϕ **do**
    choose v from $V(\phi_i)$;
    $n_i = |V(\phi_i)|$;
    $c_i = \text{DDP}(\phi_i|_{\{\neg v\}}, n_i - 1) + \text{DDP}(\phi_i|_{\{v\}}, n_i - 1)$
  return $\prod_i c_i$

best-first search instead of a *depth-first* search. The order of this search is based on a variable branching heuristic score (Section 3.5).

3.3 Clause Learning

Another method that RelSAT uses is *clause learning*. Once an assignment on a formula results in an empty clause, also referred to as *conflict*, the solver derives from the implication graph [52] which variable assignments caused the conflict. By adding these variables as a new clause to the formula futile search is prevented, since this new clause results in a conflict before the same combination of variables were to be assigned again. In other terms; clause learning *prunes* the search space from futile parts of the search space of the formula.

Like many of the mechanisms we discuss here, clause learning was originally proposed in the context of solving SAT. The first solvers that used this were GRASP [64] and zChaff [78] and have been a key mechanism for many state-of-the-art SAT solvers ever since. In order to fully exploit clause learning the branching heuristic of RelSAT is chosen to trigger this behaviour as often as possible. This is referred to a *Conflict-Driven-Clause-Learning* (CDCL).

3.4 Component Caching

The work on component decomposition in RelSAT yields an interesting observation that the same components can be encountered more than once during solving formulae. By storing solved components along with their model count in memory, equal components that are found elsewhere in the search do not have to be solved again. Instead, the saved model count can be used, preventing redundant computations. Bacchus et al. [2] proposed three different methods to use caching with DPLL-style algorithms in which *component caching* was the theoretically most competitive. The other two methods,

simple caching and *linear-space caching*, either do not use component decomposition or remove all child components once the count of their parent was computed in order to attain linear space usage. Beame et al. [5] proposed a methods for caching as well, but here the only store the residual formulae that were UNSAT.

Unfortunately, theoretical competitiveness does not directly translate to practical feasibility. Theoretically, *component caching* can store all components with their counts. In practice only a selection of all encountered components can be simultaneously in the cache due to memory limitations of computers.

3.4.1 Combining caching with clause learning

Clause learning and component caching are two seperate mechanisms that were proposed to solve model counting problems more efficiently with DPLL-style solvers. Sang et al. showed with their solver Cachet that component caching and clause learning can be combined into a competitive solver [59]. During the search, clause learning adds clauses to the original formula. This means that if a component from a partial assignment of π is seen again, the solver might not be able to recognise that component, due to the added clauses between these two encounters. In order to resolve this, learned conflict clauses are kept in a separate formula θ and only decomposed components, originating from the (residual) formula ϕ, are cached. The model count of these components holds its correctness because added clauses only prune the unsatisfiable search spaces. This means that the model count is not affected by conflict clauses, so for a partial assignment π the count is the same for $\phi|_\pi$ and $\phi|_\pi \wedge \theta|_\pi$. This approach is called *bounded component analysis* and holds two other useful properties:

1. Adding clauses results in a larger formula, which requires more memory to be stored. The number of conflict clauses that are added to the formula can be exponentially larger compared to the number of clauses in the original formula. The components of these large formulae would require more space, resulting in fewer counts that can be stored within the same amount of memory. By keeping the clause in a separate cache, the space required to store counts is not affected.
2. Clauses in θ increase the constrainedness of ϕ, which reduces the likelihood of decomposing a formula into multiple components.

To maintain correctness over the count with caching and conflict clause learning, sometimes cached components need to be removed. Otherwise the model count can become a *lower bound* of the actual count. Recall that, sometimes, a (residual) formula can be decomposed into multiple components, which are solved as independent problems. When one of these components is unsatisfiable, this means that for the whole residual formula there are no models. However, sibling components of an unsatisfiable component can already be solved and cached. The count of those components have been solved under false assumptions and could yield a lower model count than the component actually has. Cachet proved that this only occurs when one components in the decomposition is unsatisfiable and is prevented by removing cached counts of all sibling components and all the other components that were cached in underling the search trees of these components.

3.4.2 Component encoding

The representation, or *encoding* of a component can have a great influence on the performance of the model counter. These encoding are used as a key in the cache that maps to the corresponding model count for that component. There is one requirement for the component encoding; it needs to be sound with regard to the model count, which means that there for any two components with a different model count, the encoding cannot be the same. This means that other properties for a component do not have to be preserved necessarily.

Other aspects of component encoding are the size of the encoding and the computational cost to produce the encoding from the component. A denser encoding allows for more counts to be stored within the same amount of available memory and a low computation encoding cost results in a lower computational overhead for component caching.

There is a trade-off between the encoding time and the resulting compression of the representation. Cachet used a *simple encoding scheme* where components are not encoded and the solving representation is stored in the cache as is.

Another model counter called sharpSAT uses a couple of steps to obtain a denser encoding. Here, the clauses and variables in the input formula are both given an identifier and only the identifiers of active clauses and unassigned variables are stored in a vector [69]. The resulting vector is further reduced by only storing the numerical differences between consecutive elements in the vector and by *packing* the vector in a bitstring. Obviously, this *hybrid encoding scheme* is requires more computations, but it generally allows for storing more components on the same amount of memory compared to the *simple encoding scheme*.

Yet another model counter GANAK [63] uses *probabilistic component caching* to compress components even more. Each component is hashed, and this hash is stored along with the model count of the component. This greatly reduces the size of cached components but comes at a price. Hash functions have the risk of mapping two components to the same hash, which causes collisions. Once a collision occurs during search, this can lead to a faulty count. Although, the bounded probabilities of collisions are very small, there is a chance that the count is incorrect. Therefore this approach should be used with caution.

More recently, symmetric component caching was proposed by Van Bremen et al. [11] for model counting. Here, components are encoded into a canonical isomorphic graph representation. Since isomorphic graphs have the same model count, this can be used as encoding. These encodings increase the likelihood of components being in the cache, but requires expensive computations.

3.4.3 Caching schemes

The available space on computer systems is limited. Hence, the cache can become full during solving. Therefore, solvers require strategies on how to cope with a limited cache size.

Caching (also known as *paging*) is a prominent problem within computer science [10] and is generally conceptualised on a system (*i.e.* a database or webserver) that has a cache. The cache is limited in size, but can quickly return a stored item. The system is considered to be much slower in returning a requested item. Note that every requested item can either already be in the cache (a *cache hit*) or not (a *cache miss*). The objective is to have a strategy that chooses which items to keep in the cache (without knowing which requests will come in the future), such that it maximises the number of cache hits.

In the context of caching in model counting, several caching schemes have been used:

First in first out (FIFO) is used in Cachet [59] and gives each component an timestamp at the moment that its count is stored in the cache. Once the cache becomes full, FIFO evicts components from the cache that have timestamps most distant from the timestamp at that moment. This is based on the assumption that older components are requested less frequently compared to newer once.

Least recently used (LRU) is a scheme that removes components based on the last time the solver had an encounter with a component and has been used by Majercik and Littman [48]. This encounter can be when storing the count in the cache or at a cache hit. The timestamps are updated upon each encounter to reflect the time of that interaction.

HitDecay gives a score to components, which based on its past encounters. The score is increased each time that component is encountered again during search. The scores are periodically divided in order to let more recent hits have more impact on the score than hits that occurred further in the past. It can be seen as a combination between the *least-recently-used (LRU)* and *least-frequently-used (LFU)* caching schemes. Once the cache becomes full all components below a certain threshold score are truncated. sharpSAT [69] and GANAK [63] use this method, where the threshold score is set such that half of all cached counts are deleted from the cache.

Marker is a scheme that runs in phases. At the beginning of each phase, all cached components are unmarked. When a component is encountered where its model count is cached, the marker algorithm [26] marks that entry in the cache. If the model count is not in the cache, the caching scheme randomly evicts an unmarked cache entry from the cache. This continues until no unmarked cache entries remain. Then a new phase starts with all components being unmarked again. The marker scheme is theoretically expected to be more competitive than the LRU and FIFO schemes [26]. To the best of our knowledge this schemes is not used before in the context of model counting.

Random is the random baseline scheme that does not keep any metric on components, but randomly removes a predefined percentage of cached counts from the cache.

Clearing the cache and maintaining the score for components yields computational overhead. Hence, one could question if it is even worth to store all component counts in the cache. For example: small components are not always stored since it is often faster to just compute the model count for them [3]. The same observation has been made in [48]. There Majercik and Littman propose a *smart* LRU, where components with a low number of variables are not cached. Additionally, components with a high

number of variables are kept out of the cache as well, since they reason that these components are less likely to be encountered later on in the search. In their experiments, the smart LRU scheme outperformed FIFO and LRU schemes for small cache sizes, but eventually all the three schemes seemed to converge to the same performance.

It would also be interesting to know if components are to be encountered again during search, since it is only relevant to store it in the cache when that is the case. This is not applied successfully in model counters yet. However, predicting if cached counts will be used in the future resembles with the work of Soos et al. [65] who showed that it is possible to predict if conflict clauses were to be used in the future in a SAT solving context. Removing conflict clauses in SAT solvers is needed because the problem instances can produce many conflict clauses. Too many of these can have a negative impact on the performance, while not all conflicts are that relevant in practice.

For efficient implementation of the cache, the cache is supplemented with a hash table where each key has a bucket in which multiple component counts can be stored. The hash table is used by all three solvers Cachet, sharpSAT and GANAK and do not require the solver to traverse a list of components each time it searches for a component in the cache. As both components and model count are stored in these buckets instead of only the model count, this approach cannot suffer from hash collisions.

3.5 Branching heuristics

In DPLL-style algorithms the search order is determined by choosing the variable in the component to branch on, which truth value is assigned to the branch variable first, and the order in which components are solved after decomposition. The search order is critical for efficient solving [8, 60] and these decisions are guided by heuristics. Heuristics score each option (*i.e.* variable, phase or component) that can be chosen and determines the order based on that score. These heuristics are designed with the solver's mechanisms in mind, since they are also implemented to solve problems more efficiently. For example, the AUPC heuristic targets unit propagation and the VSIDS heuristic targets clause learning.

Heuristics can be very effective in producing a small search tree, but they come with computational overhead. Some heuristics are more expensive (*e.g.*higher computational overhead) than others. Practically, this means that a cheap to compute heuristic with a larger resulting search tree can outperform an expensive heuristic with a smaller search tree.

There are many heuristics that can be used for model counting. In this work we consider the following:

Dynamic largest combined sum (DLCS) counts the number of times a literal of a variable appears in a (residual) formula. A higher score indicates that this variable will probably affect the most number of clauses when it is branched over, resulting in a small residual formulae. Overall, DLCS is expected to result in smaller search tree [64].

Dynamic largest individual sum (DLIS) is similar to DLCS, but instead of combining the score for

each polarity it uses the maximum number of occurrences of both corresponding literal polarities in the residual formula.

Exact unit propagation count (EUPC) is a heuristic which aims to direct the search such that unit propagation is induced and conflicts are triggered. It is an expensive heuristic to compute, since for each literal of the variables in the component, the number of simplifications it yields in unit propagations needs to be computed. This approach is used in Relsat [3].

Approximate unit propagation count (AUPC) scores variables by estimating the amount of unit propagations that occur when assigning a truth value to that variable [31]. The estimation procedure start by assigning a truth value to the variable v. Now, count each binary clause that contains a literal of v that becomes falsified by this truth assignment. In that case, the accompanying literals in those binary clauses must become `true` in order to satisfy that clause. Consequently, we also count the number of binary clauses that hold the complementary literals of the aforementioned accompanied literals. In conclusions, the AUPC score is the sum of the estimation procedure for both truth values `true` and `false`.

Variable state independent decaying sum (VSIDS) keeps a score for each variable of the formula during search and the score of a variable is increased when that variable is part of a conflict clause. The scores are decayed periodically, giving higher priority to conflict that occurred more recent. One advantage of VSIDS is that the residual formula should not be analysed, which makes it very efficient. VSIDS was one of the success factors of the SAT-solver zChaff [78]. For model counting it is considered to not work well, since instead of finding only one solutions, the full search space needs to be explored.

Centrality uses the *primal* (variable) graph representation of the input formula to compute the betweenness centrality of each variable. The primal graph is a graph where each node represents a variable of the formula and edges represents the presence of two variables in a clause of the formula. Betweenness centrality is the number of times a node (variable) is part of the shortest path between two other nodes in the graph [29]. The intuition behind this heuristic is that a high centrality score corresponds to having a strong influence on the community structure [30] of the formula. *i.e.* it targets component decomposition. Computing the betweenness centrality is quite expensive. Hence, the scores are only computed over the input formula once and used throughout the search. Nevertheless, the performance of the centrality heuristics is quite competitive [9].

3.5.1 Combining heuristics

Thus far, the described heuristics only focus one mechanism that they aim to utilise. Modern model counters often carry multiple mechanisms and in order to target more than one mechanism, heuristics are combined.

One way in which heuristics are combined is by using a weighted sum. One example is *variable state aware decaying sum* (VSADS), that combines VSIDS and DLCS and is used in Cachet [59]. When there are

not much conflicts in the search, the VSIDS score for all variables can behave randomly [60]. By also including DLCS scores, VSADS is also *aware* of the formula it is handling and alleviates the random behaviour of VSIDS. For VSADS the weights are empirically set in sharpSAT (and GANAK) to:

$$VSADS(v) := DLCS(v) + 10 \cdot VSIDS(v)$$

Bliem and Järvisalo [9] combined Centrality as a weighted sum with either DLCS, VSIDS, or VSADS. For the combination with VSADS this looks like:

$$score(v) := DLCS(v) + 10 \cdot VSIDS(v) + w \cdot Centrality(v)$$

For DLCS and VSIDS the same weights were used as found in [60]. The weight for Centrality w is chosen such that, after multiplication, the score is at most the number of clauses of the input formula.

Another way of combining heuristics is to have a *two-step* process. Here, the variables are first ranked by a chosen heuristic. After that, a second heuristic is used to select a variable from the subset of variables that are on top of the ranking. Variables are in this subset if their score is above a certain threshold, which is often a percentage of the highest score found. One advantage of this two-step approach over a weighted sum is that one heuristics needs to be computed for a smaller number of variables. Another advantage can be that there is a smaller change that the two heuristics diminish each other.

We discuss two heuristics that are used a secondary heuristics in model counters. Both have been proposed in GANAK:

Exponentially decaying randomness (EDR) randomly selects a variable from the selected subset from the ranking of the first heuristic. The threshold is a percentage of the best score from the ranking and this percentage become more strict as the search progresses. In GANAKthe first heuristic is VSADS and the percentage is chosen by the *exponentially increasing heuristic equivalence parameter*: $100 - 10 \exp^{-0.0001 \cdot \#Decisions}$. After 20 000 induced decision nodes, EDR is switched off and only the VSADS is used to select the variable to branch on.

CacheScore is, similar to VSIDS, a decaying sum. When a component is added to the cache, the score of the variable in that components are increased. A high CacheScore indicates that that variable is prominently present in the components in the cache. Branching over such an variable would render those components useless for the underlying search because that variable would not be present anymore. Hence, branching over a variable with a low CacheScore would be better to do, since it increases the chances of getting a cache hit.

Combining heuristics can reinforce the overall utilisation of mechanisms in the solver such that the solver performs better [9, 60]. However, this is not guaranteed and combining heuristics could have a counter productive effect on performance if the scores annihilate each other. We see evidence of such

behaviour in Section 5.3.

In this work we allow for all weigthed sums that are described in literature [9,60], which are combinations of VSIDS, DLCS, and Centrality. For the two step approach we allow all single heuristics, as well as the weighted sums to act as the first heuristic. For the second heuristic in this approach, we allow EDR or CacheScore. EDR is not a good candidate to be used in a weighted sum, since it adds randomness to the selection procedure. GANAK uses a a two step process with VSADS and CacheScore as the first and second heuristics, respectively and is referred to as CSVSADS. We acknowledge that the selections of heuristics for both combination methods can be extended, but that is outside the scope of this thesis.

3.6 Model counters overview

Table 3.1: An overview of a selection of model counting solvers that have a DPLL-style basis.

Solver	Added/modified mechanisms	Branching	Year	Extends
CDP [8]	DPLL style search	DLCS	1999	DPLL [22]
RelSAT (DDP) [3]	Component decomposition, Clause learning, Iterative implementation	EUCP	2000	CDP
Cachet [59]	Component caching, FIFO caching scheme	VSADS	2000	RelSAT
sharpSAT [69]	IBCP, hybrid component encoding, HitDecay	VSADS	2004	Cachet
Centrality [9]	*none*	Centrality	2018	sharpSAT
GANAK [63]	Probabilistic component caching, LSO , Independent Support	CSVSADS	2019	sharpSAT
SYMGANAK [11]	Symmetric component caching	CSVSADS	2021	GANAK

Table 3.1 provides an overview of the different exact model counters that have been discussed in this chapter. Over the last two decades these model counters have been representing the state-of-the-art for solving model counting problems, and each solver conceptually shares the same origin. Practically, this does not directly translate that their code implementations are also built upon each other. RelSAT, Cachet and sharpSAT all have different code bases, that are highly optimised to efficiently implemented the mechanisms and branching heuristics of those solvers. On the contrary, the code of Centrality, GANAK and SYMGANAK are all based on sharpSAT. Most of the mechanisms and heuristics in the table have been discussed in this chapter, but not all. We briefly describe the ones that are not covered in the previous sections.

sharpSAT proposed *implicit Boolean constraint propagation (IBCP)*. This is a *look-ahead* technique that checks if assigning a truth value to a variable will cause a conflict. In that case only the opposite truth value can be assigned by the solver, resulting in a propagation step.

GANAK introduced many mechanisms and modifications to the solver it built upon, sharpSAT. *Learn and start over (LSO)* runs the solver for a short period of time after which the solver start over with the search. The motivation for LSO is that are heuristic counters get initialised, which should improve the performance of the solver. Independent Support is only used for 'hard' instances and can be considered

as an optimisation step. Once solving takes too long (after 5 millions induced decision nodes), the solver stops and computes the independent support [14] over the variables in the input formula. Independent support find a subset of variables $\mathcal{I} \subseteq V$, where the model count of the variables in $\mathcal{I}$ projected on ϕ is equal to model count of all the variables in ϕ. This basically makes the independent support optimisation a projected model counting problem and has demonstrated that it can be quite effective.

Chapter 4

Approach

In this chapter we set out our approach to gain a better understanding of the cache and describe the methods we use to answer our research questions. With these insights we plan to find answers to how the cache behaves under different circumstances, which identifying characteristics relevant components have and to better understand the trade-off between cache size and running time performance.

4.1 Pipeline

Our approach is structured in three phases and is illustrated in Figure 4.1. The first phase is the extraction phase where we build databases out of encountered components when solving instances with the solver. For these components we mine characteristics of the component structure and solver search state. Furthermore, we track the components interactions with the solver after the first encounter. In the second phase, we use these databases to visualise how the cache interacts with the rest of the solver

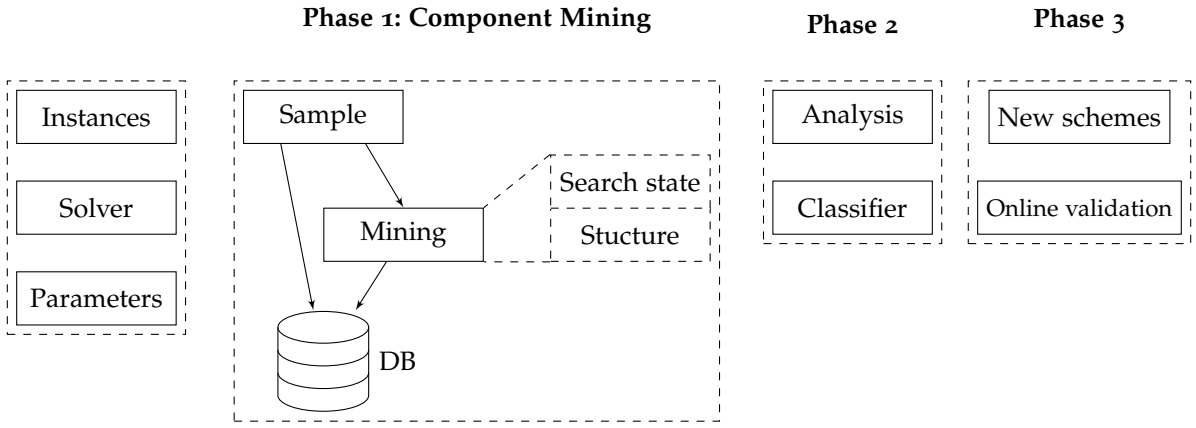


Figure 4.1: Diagram of the pipeline that is used for finding characteristics of cache behaviour and how we identify relevant component features, such that better caching schemes can be made.

during search. We also use the databases to train classification models in order to see how predictable different types of components are. In the third and also the last phase we use the insights and classifiers from the second phase to construct new caching schemes with the aim to be more efficient with the available space and to improve solving effectiveness.

4.2 Component mining

In this section we introduce the features that we calculate for each component, including three definitions of component relevance. These features will be used in the following phases in our pipeline. We begin with describing the features, followed by the definitions of relevance. At last, we discuss how we collect these features during instance solving, which includes the sampling of components.

4.2.1 Features

To characterise a component we collect features from the component's structure and from the solver's search state in which components are encountered. The structure of the input formula is also used to normalise features against. For example, the number of variables in the component versus the number of variables in the input formula. The motivation to normalise features is that it could help to generalise feature distributions when combining features that originate from different input formulas. In total we mine 228 different features from a component of which 96 are normalised against features from the input formula.

4.2.2 Component structure features

Every component is a subformula of the input formula, which is in CNF. Hence, each component is in CNF as well. For the component structure features we use a subset of the features proposed by SATzilla [76]. We selected features that capture structural properties, such as component size, clause types, literal polarities, and how variables and clauses are related to each other. Features that rely on probing using different search techniques are not used because those often need considerable amounts of time to produce meaningful metrics. Since we need to compute these features often during search that would render impracticable. In addition to those features we also describe several other features for capturing component structure. We collect the following features for each component c:

3 problem size features Number of variables $|V(c)|$, number of clauses $|\mathcal{C}(c)|$, ratio of number of variables to number of clauses $|V(c)|/|\mathcal{C}(c)|$.

12 variable-clause graph features For the variable node degree, we compute the mean, median, variation coefficient, minimum, maximum and entropy. We do the same for the clause node degree.

12 variable graph features We compute the mean, median, variation coefficient, minimum, maximum and entropy of the node degrees, and of the node eccentricities. The eccentricities are computed

using the bounding eccentricities algorithm [68].

12 clause graph features We compute the mean, median, variation coefficient, minimum, maximum and entropy of the node degrees, and of the node clustering coefficients.

15 balance features We compute the mean, median, variation coefficient, minimum, maximum and entropy of the ratio of the number of positive to negative literals in each clause in c, and over the ratio of the number of positive to negative literals in c for each variable in $V(c)$. A ratio is computed by p/n, with p denoting the positive number and n the negative number. When n is 0, the ratio is equal to p. We also collect the ratios that represent the fractions binary clauses ($|\mathcal{C}_2(c)|/|\mathcal{C}(c)|$), ternary clauses ($|\mathcal{C}_3(c)|/|\mathcal{C}(c)|$), and Horn clauses ($|\mathcal{C}_H(c)|/|\mathcal{C}(c)|$). Here $|\mathcal{C}_2(c)|$ denotes the number of binary clauses in the component. (and analogous for ternary clauses and Horn clauses).

3 clause fraction features The number of binary, ternary and Horn clauses divided by the total number of clauses $|\mathcal{C}(c)|$.

6 Horn formula proximity For each variable in $V(c)$ we collect the number of occurrences in a Horn clause $\mathcal{C}_H(c)$. For the resulting distribution of numbers of occurrences, we then compute the mean, median, variation coefficient, minimum, maximum and entropy.

As described in the paper, we also normalise all the aforementioned features by dividing the component-specific feature values by the corresponding value of the input CNF ϕ. Three other feature categories to characterise the component:

3 clause balance features This category of features is inspired by SATzilla's balance features, but instead dividing the number of binary, ternary or Horn clauses by the number of clauses in the component we divide by the number of clauses in the input CNF. Note that the features in this category are already normalised.

11 timing features We measure the CPU time that is needed to compute features. We categorise the total time that was needed to compute the component features into 5 partitions; component encoding parsing, graph conversion, eccentricity computation, clustering coefficients computation, and computing the ratio features and Horn proximity features. Note that when parsing the component encoding the problem size features and variable-clause graph features, are collected simultaneously. We further compute the fractions of each timing partition divided by the total CPU time to compute the component features of c.

1 solution density We compute the solution density [35] as $log_2(mc(c))/|V(c)|$. Because the solution of the input instance is unknown until after it is solved, it is not feasible to normalise it by dividing by the solution density of the input CNF.

4.2.3 Search-state features

We introduce a number of new features that are more specific to component caching in model counters, which are divided into 8 different categories. The focus lays on the context in which the component c is first encountered $\tau_0(c)$. Each categories is described as follows:

5 progression features We collect the total number of decision nodes that the solver induced at $\tau_0(c)$, and the numbers of variable assignments that the solver made through *unit propagation* (UP) or *Boolean constraint propagation* (BCP) at $\tau_0(c)$. We also collect the sum of the UP and BCP variable assignments and also the total number of variable assignments due to decisions, UP or BCP.

5 partial assignment features From the partial assignment π at $\tau_0(c)$ we count the number of assignments that are induced by decisions, UP or BCP. Like with the progression features we compute the sum of the number of UP and the number BCP assignments and the total number of truth value assignments in $|\pi|$.

5 max truth assignment features We keep track of the maximum numbers of the partial assignment features seen between the start of solving and $\tau_0 c$.

7 truth assignment balance features We compute the ratios between the truth values assignments in the partial assignment π at $\tau_0(c)$ induced by UP, BCP, the sum of UP and BCP, or decisions. Furthermore, we compute the fractions of the number of truth assignments induced by UP, BPC, the sum of UP and BCP, or decisions divided by the total number of truth value assignments in π.

4 search tree features We collect the *ancestors* $\mathcal{A}$, which are all nodes from the root of the search tree to the current branch node at $\tau_0(c)$. For each ancestor, we record in which direction the solver has branched to at the moment of mining the search state, and how balanced that is over all the ancestors: $1/|\mathcal{A}| \cdot \sum_{a \in \mathcal{A}} dir(a)$, with $dir(a) := 0$ if the solver branched left in node a, and $dir(a) := 1$ if the solver branched right. Similarly, we collect the truth value assignments made to the nodes in $\mathcal{A}$ and measure how balanced that is: $1/|\mathcal{A}| \cdot \sum_{a \in \mathcal{A}} val(\pi(a))$, with $val(\pi(a)) := 1$ if $\pi(a) = \top$, and $val(\pi(a)) := -1$ if $\pi(a) = \bot$. Additionally, we collect the balance of all the truth value assignments in the partial assignment π: $1/|\pi| \cdot \sum_{a \in \pi} val(\pi(a))$. Finally, we collect the number of disjoint components at the branch node at $\tau_0(c)$.

4 conflict clause features We collect the total numbers unit, binary and longer conflict clauses that were learned up to $\tau_0(c)$, as well as the number of longer conflict clauses that are still active at $\tau_0(c)$.

12 VSIDS features We compute the mean, median, variation coefficient, minimum, maximum and entropy of the VSIDS scores of the variables in $V(c)$ and analogously for $V(\phi)$.

12 CacheScore features We compute the mean, median, variation coefficient, minimum, maximum and entropy of the CacheScore scores of the variables in $V(c)$ and analogously for $V(\phi)$.

We only normalise the first three feature categories and do this in two ways: by dividing it by the

number of variables $|V|$ in the input CNF, and by dividing it by the number of clauses $|C|$ in the input CNF. This results in 30 normalised features.

4.2.4 Measuring component relevance

The features described above are either search-agnostic and static, or search-aware but still static, as they are recorded only the first time a component is encountered. We now describe the metrics that we use to characterise the component's journey throughout the solving process. In Phase 2, we use these metrics to define the class labels in supervised learning tasks. For each component that we mine, we record when the solver *encounters* the component during the search.

For each encounter we notate the moment at which it occurred in terms of the total number of decision nodes that have been created thus far. At this point, either the component count can be retrieved from the cache, or it must be (re)computed before it is stored in the cache. We denote the total number of these encounters with N_e.

Additionally, we record a proxy for the computational cost $cc(c)$ for computing the model count of the component in terms of number of decision nodes. Specifically, we compute the absolute difference $\Delta d(c)$ the solver had taken between when c was encountered for the first time and when its resulting model count $mc(c)$ is found. Since DPLL-based model counters have a depth-first search order, this is a proxy for $cc(c)$. Note that $\Delta d(c)$ is influenced by the parameters settings of the model counter, since they have effect on the search tree, and may cause different sub formulae of the component to already have stored model counts in the cache.

We use the measurements from above to define three different proxies of the relevance of component c:

Number of encounters $N_e(c)$. The number of times the same component is encountered by the solver during solving.

Relevance window length $\ell_r(c)$. The time difference, in terms of number of decision nodes, between the first encounter of the component and its last encounter.

Computational cost savings $R_{cc}(c)$. We compute a proxy for the computational costs, in terms of number of decision nodes, that we would save if each encounter of component c leads to a cache hit, as follows: $R_{cc}(c) = \Delta d(c) \cdot N_e(c)$.

Each relevance measure captures different aspects of relevance. When $N_e(c)$ is large, keeping the component in the cache leads to many cache hits. If $\ell_r(c)$ is small, we only need to store the component for a short while. Finally, if $cc(c)$ is high, or $N_e(c)$ is large, we can save much computational cost by keeping the component in the cache.

4.2.5 Sampling

When solving a single instance, the number of distinct encountered components can grow exponentially with the number of variables in the input CNF [6]. For all but the smallest instances, mining and tracing all encountered components would result in massive databases. This makes processing them computationally infeasible. For this reason, we take a sample of all components that are encountered during solving an instance, using hash-based sampling [25].

Specically, we use a hash function $h : \mathcal{C} \mapsto [0, 1]$ over the space of components $\mathcal{C}$, a sampling rate $0 \leq r \leq 1$, and for each *first* time that the model counter encounters a component $c \in \mathcal{C}$, we test if $h(c) < r$. If this is the indeed the case, we collect the features and further trace its encounters in order to compute relevance measures. We use `CLHASH` [46], to hash components c.

To verify that hash-based sampling remains free from distribution biases on the features we performed a preliminary experiment where we compared the feature distributions between a sampled database and a database with no sampling. For each feature pair we performed a bivariate two-sample Kolmogorov–Smirnov test [54] between the two databases that were mined from a single instance. The bivariate two-sample Kolmogorov–Smirnov test tests if two distributions originate from the same distribution. Two sample rates of 1% and 10% have been tested on 10 randomly selected instances. After multiple-test correction of $1/20$ and an α of 0.05 we see that for all instances the samples do not result in significantly different distributions.

4.3 Classification

Besides analysis of the databases we also plan to learn predictive models where features are used as input vector and a single relevance measure act as a target. With these models we can do three things. Firstly, the predictability performance can be assessed and compared over different classifications specifications. Secondly, the models can be analysed to find directions to features that have a high importance in predicting relevance. These insights are then used to define new heuristics. Thirdly, the predictive models can be assessed on their performance when embedded into the solver as an oracle. We propose several caching schemes that depend on such an oracle later in Section 4.5.3.

All our prediction tasks are structured as a binary classification tasks, by setting a threshold on one of the relevance measures. Components below this threshold are considered irrelevant and the ones that are equal or above the threshold are considered relevant. Instead of hand-picking thresholds for each performance metric, we chose them such that for a specified relevance metrics, a chosen percentage of components is relevant. We refer to this percentage as the *cutoff percentage*.

4.4 Automated algorithm configuration

The reason for having many configurable parameters to control the caching part in our solver is to support the paradigm of Programming by Optimisation(PbO) [38]. Here, the idea is that many algorithms have many hard-coded parameters (*magic constants*) that are set based on empirical experimentation and intuition by its creators. This is often a tedious process and requires a lot of expert knowledge to obtain a good set of parameter values that perform well. Additionally, it makes an algorithm less versatile to different problem inputs. Alternately, finding good parameters for on a given instance set and performance measure can be seen as an optimisation problem. This problem is referred to as Automated Algorithm Configuration(AAC) [37], and has showed that the parameter configurations found with AAC often outperform the expert's parameter configurations on a given instance set and a targeted performance measure. Another advantage of AAC is that it makes experimentation less dependent on a fixed set of parameters, which allows for more general conclusions.

4.5 Extending GANAK

In this work, we use the CDCL-based model counter GANAK [1] as our basis for our anlysis. GANAKwas the best performing model counter in the exact solving track of the Model Counting Competition 2020 [27]. We extend GANAK in three different directions.

4.5.1 Configurability

The first dimension it that of the configurability of the solver. One of our interests is to understand how model counters could make more efficient use of the cache. This is subject to the how the rest of the solver approaches to solve the problem, which is defined by the parameter settings of the solver. In order to reduce potential effects of specifically fixed parameters, we must have high flexibility in the configurability of the parameters of the solves.

The version of the version of GANAK we built on has a number of hard-coded magic constants. These magic constants are often empirically set with experimentation and expert intuition. Some examples of these constants are decay intervals of heuristical values and the weights for combining two heuristics. We made many of these magic constants tune-able by the user.

We also extended the solver. GANAK only deploys the VSADS heuristic with hardcoded weights in combining DLCS and VSIDS. We extend this by allowing to chose individual heuristics DLCS, DLIS, VSIDS, AUPC, Centrality, and Random. Additionally, the DLCS, VSIDS, and/or Centrality heuristics can be combined in a weigthed sum. Note that the weights are are also configurable (Section 3.5).

The cache strategy of GANAK is a FIFO scheme. Here, the oldest entries to the cache are removed to clear 50% of the cache space once it it full. We extend this by adding the LRU, Hitdecay, Marker, and

[1] https://github.com/meelgroup/ganak (accessed on 12 March 2020)

Random schemes to be chosen as well (Section 3.4.3).

The resulting solver is highly configurable with 24 tune-able parameters, instead of the original 5 we choose to be configurable. A full overview of the parameters is available in Appendix A. All parameters can be set by using command-line arguments.

We also restrict some functionalities of GANAK, in order to do our analysis. We turn PCC, LSO and independent support off, because those functionalities overcomplicate the mining and classification tasks. We always turn component caching on, since that is the mechanism we are investigating.

4.5.2 Data collection

The second dimension is that of data collection. In order to compute the component structure features (Section 4.2.2) , search state features(Section 4.2.3) and component relevance features (Section 4.2.4), we implemented a data collection functionality into GANAK. The data collection pipeline consists out of computing the features for a given component c and search state $\mathcal{S}$, tracking components encounters during solving, and a functionality to export the collected data to a database.

The data collection can be turned on or off with a command line argument. With an additional argument, the sample rate (Section 4.2.5) can be set. When the solver is using a caching scheme that relies on component classifications only the computation of features is activated, in order to feed the classifier a full or partial set of features as input.

Note that during the tracking of components, we track if for the full duration of the solving process. Even when the component and its model count are removed during a cache removal and is encountered again later on, it is not considered as a *new* component.

4.5.3 Classifier-based caching schemes

We now describe four caching schemes that make use of component's relevance classifiers:

Oracle is the naive classification scheme. When the cache is cleaned, each cached component is classified with the relevance classifier to predict its relevance, given the search state of the solver at that moment, and the structure of that component. Based on the classification, it is discarded from or kept in the cache. This scheme has limited control over the removal proportion since it does not use a score ranking. To overcome the situations where only a very few components are considered irrelevant, and an almost full cache remains after clean up, a back-up scheme removes a minimum proportion of the cache. The back-up scheme can be any of the schemes mentioned in Section 3.4.3, except the Marker scheme.

OracleConfidence is a variant of the Oracle scheme, but instead of only using the binary classification prediction it uses the classification confidence. All components that are predicted to be relevant are

ranked based on the confidence of the prediction. Using the confidence has two advantages: it gives more control to the desired proportion of the cache that is removed, as lower-ranked components can be discarded and it allows narrowing the selection of components after classification to be of higher expected relevance. The latter is set with a minimal confidence threshold.

SeededHitDecay is conceptually equal to HitDecay, except for the initialisation of the component's score. When a component enters the cache it is given to the relevance classifier. A classified relevant component is given fixed initial score, whereas non-relevant components are initialised with the default value of 1. The initial score for relevant component is a fixed value that can be set by the user.

Hybrid is a scheme that uses two separate caches, one cache for regular components and one cache for (highly) relevant components. Each new component that is added to the cache is classified and, based on the prediction, is placed in one of the two caches. Each cache has its own scheme that handles cache removal and can be FIFO, LRU, HitDecay, SeededHitDecay or Random. The motivation behind the hybrid scheme is to reduce the number of calls to the classifier, as this yields computational overhead. The scheme with regular components can be one with a small computational overhead, for example, FIFO, whereas the (smaller) relevant cache can also use the other classifier based schemes. When the relevant cache uses a classifier based scheme the same classification model is used that determines in which of the two caches the components belongs.

In addition to the broad selection of caching schemes, parameters are made available to be configured. For all schemes these are the minimal and desired percentages of components that gets removed when a cache removal procedure is initiated. For specific schemes other parameters are configurable as well. In total, the parameter space of the solver is extended with 12 new parameters that are related to the caching scheme.

Chapter 5

Experiments

In this chapter, we present our experimental setup and show results we that we use to answer the research questions we stated in Chapter 1. For convenience, we briefly repeat them here again:

Q1 How do running times of a DPLL-based #SAT solver depend on the maximum allowed cache size?

Q2 Why do smaller cache sizes not necessarily decrease solving times?

Q3 How do branching heuristics influence cache usage?

Q4 How do caching scheme influence cache usage

Q5 To what degree can it be predicted how beneficial it is to the solver to keep the model count of a component stored in the cache?

Q6 How can component relevance predictions be leveraged in DPLL-based #SAT solvers to solve with less decision nodes?

5.1 Experimental setup

Here, we describe the hardware and software where we performed our experiments with. We also describe the benchmarks we used and describe the set up we use for automated algorithm configuration, classification and online classifier validation.

5.1.1 Hardware and software

All experiments were executed on the *grace* compute cluster which has 32 nodes. Each node has two 16-core *Intel(R) Xeon(R) CPU E5-2683 v4* processors that run at 2.1GHz and are each equipped with 94GB of RAM. All solving runs are executed on a single core with a memory limit of 3GB, unless mentioned otherwise. Preliminary experimentation showed no significant ($\alpha < 0.01$) effect on CPU running time when solvers ran in parallel compared to solving them sequentially.

We use the extended version of GANAK, as described in Section 4.5. For our configuration experiments, we chose the general-purpose configurator SMAC 2 [40], because it is one of the best-performing algorithm configurators. SMAC 2 is a model based algorithm configurator that is able to handle both continues and categorical parameter domains. Additionally, SMAC 2 integrates with Sparkle, a tool that automates the automated configuration process.[1] We used SQLite 3.30.1 as our database system. We used scikit-learn 0.23.1 [56] to train the classifiers and scikit-optimize 0.8.1 [36] for the hyperparameter tuning of these classifiers, all operating on Python 3.8.3.

5.1.2 Benchmark sets

We use three benchmarks for our experiments that are known from literature. Specifically, we select two artificially structured benchmarks, GRID [61] and DQMR [61] and one benchmark that originate from industrial problems; *program analysis* (PA) [51]. Here, we give a brief overview of the structure and base problem of instances in each benchmark.

DQMR [61] These instances are CNF encodings of Bayesian networks that simulate diagnostic networks. This are two layer bipartite networks, where the nodes in the first layer represent diseases and the nodes in the second layer represent symptoms. The number of diseases and symptoms vary between 50 and 100. Each symptom is randomly linked to 4 diseases. Each instance solves a query that computes the marginal probabilities for having a disease given a selection of observed symptoms. The benchmarks was originally constructed for weighted model counting. But by ignoring the variable weights, we solve it in as exact instances.

GRID [61] Like DQMR, these instances originate from Bayesian networks. The networks are a square grid of size $N \times N$, where each node $X_{i,j}$ for $1 \leq i, j \leq N$ has an incoming edge from $X_{i-1,j}$ and $X_{i,j-1}$ as long as the indices are greater than zero. A proportion of the nodes is deterministic, meaning that the conditionals are set to a probability of 0 or 1 at random. The other nodes are given uniformly random probabilities. The networks were encoded with bif2cnf [59]. We extended the benchmark set by generating new grid networks that we encode using bn-to-cnf [2]. The encoding method between bif2cnf and bn-to-cnf is different, which reduces the encoding homogeneity of the benchmark. The size of the grids vary between 10 and 50 and the probabilities lie are .5, .75 or .9 These instances are weighted model counting problem that we approach as exact problems.

Program Analysis (PA) [51] These instances originate from symbolic execution problems that explore the frequency execution paths in software given a random input. With this, bugs and leaks can be detected. The symbolic execution problems originate from seven different programs and are generated with the symbolic execution bug finding tool CAnalyze [77]. The problems are encoded into CNF with Boolector 3.2.0 [53]. All problem instances that were solved under a second by the sharpSAT [69] were excluded from this benchmark set.

[1] Accessible through `ada.liacs.nl/projects/sparkle`.
[2] `Accessible through www.github.com/gisodal/bayes-to-cnf`

Table 5.1: *Mining*, *Training*, *Test* and *Online* set sizes for the three different benchmarks. Table also shows indication of the problem instance sizes in each benchmark set, in terms of number of variables $|V|$ and number of clauses $|C|$.

benchmark	*Mining*	*Training*	*Test*	*Online*	$\lvert V\rvert$	$\lvert C\rvert$
DQMR [61]	13	57	57	54	100–200	300–639
GRID [61]	19	81	80	30	389–4 256	650–14 901
PA [51]	11	48	48	87	883–27 754	2 429–81 547

From each benchmark we select the instances that ran between 5*s* and 3600*s* using GANAK with default parameters. We made this selection such that we have problem instances whose difficulties are within a reasonable range. We divided these selection into *Mining*, *Training*, and *Test* partitions as can be seen in Table 5.1.

We also have a fourth partition *Online* that we use in Section 5.6. These instances are selected from the complete benchmark set and have at least one time that the cache becomes full in a run with GANAK with configured parameters and a cache size of $128MB$, are solved within 600*s*. There are no overlapping instances in the *Online* partition with the other partitions.

5.1.3 Configuration

The default configuration of GANAK as well as the parameters being tuned were already discussed in Section 4.5 and are detailed in Appendix A. In our configuration experiments, the objective is to minimise the CPU running time, measured in seconds and report on the penalised average runtime with a penalty factor of 10 (PAR10). Each configuration experiment consisted of 15 separate runs that each had a budget of 24 hours for configuration on the instances of the *Training* partitions. Each instance run had a cutoff time of 600 seconds. From the 15 runs, the configuration with the best PAR10 performance on all the instances of the *Training* partition was selected as the optimised configuration. We reported results on running the optimised configuration on the instances in the *Test* partition.

5.1.4 Classification

For the classification experiments, we took the instances from the *Mining* partitions and constructed component databases from them using a cache limit of 128MB and the optimised configurations. The sample rate is chosen such that for each instance the expected number of components that will mined and traced is 100 000. The minimum sample rate is 0.1%. All instance databases from each partition are concatenated into one large database. The rows in the database are then randomly split into a training set and a test set, containing 80% and 20% of the data entries, respectively.

We then use Bayesian optimisation from scikit-optimize with a budget of 200 iterations and 3-fold cross validation to find the best classification model on the training database with the specified relevance label.

The classifier can be a decision tree, a random forest or a *stochastic gradient descent* (SGD) linear model.

We allow the hyperparameters of these models to be tuned by the Bayesian optimisation, but we limit the tree depth and the maximum number of trees in the forest to prevent the model in becoming too large.

The objective of the optimisation process is the F1-score of the relevant(`true`) labels. The F1-score is the harmonic mean of precision and recall over the predicted labels $\hat{y}$ and the true labels y:

$$F1score(\hat{y}, y) = 2 \cdot \frac{Precision(\hat{y}, y) \cdot Recall(\hat{y}, y)}{Precision(\hat{y}, y) + Recall(\hat{y}, y)}$$

All classification tasks for prediction component relevance are imbalanced ranging from $1/4$ to $1/100$ for the rows being labelled as relevant. Hence, we use precision-recall (PR) curves to validate the performance of the classifiers.

5.1.5 Classification embedding

To leverage the classifier in model counters itself, we embedded the classifiers back into the solver's code base. For this we used a modified version of sklearn-porter[3] that ports the models into c++ code. The resulting code was embedded into the solver code. These classifiers were used by the classifier schemes that make use of classifier predictions (Section 4.5.3).

We assessed the effectiveness of the caching schemes that rely on classifier advice by comparing the induced decision nodes the solver produces to solve problem instances. Instead of using the absolute number of decision nodes, we made use of a competitive ratio. Specifically, the number decision nodes in a run divided by the decision nodes of a run with no cache limit. The run with no cache limit acted as a proxy for being an optimal run.

5.2 Trade-off between memory and solving time

First we answer **Q1**, by gauging how much running time reduction we can achieve with for each benchmark set and different cache limits. We present the results in Table 5.2. We see that, for all benchmarks set and all cache limits, automatically configuring GANAK is effective in reducing the PAR10 scores. Especially for the GRID and PA benchmark sets we see drastic improvements speed-ups of two orders of magnitude. Interestingly, we do not observe any clear relationship between the cache limit and the PAR10 scores. Especially after configuration. This suggest that the constraint on the cache size is not that important, as long as the configurator can find a good configuration of the other parameters. However, we do note that the improvements after configuration make the solver so effective, that, even for the smallest cache limit, most instances of GRID and PA the cache never becomes full. Nevertheless, the PAR10 scores do not chance much for the default configurations as well.

[3]Available through www.github.com/nok/sklearn-porter

Table 5.2: Penalised Average Running times (PAR10) for the default configuration and best found configuration for different benchmarks and different cache sizes in *MB*.

Benchmark	Config	Metric	Cachesize								
			32	64	128	256	512	1 024	2 048	4 096	8 192
DQMR (57)	default	PAR10	1 441	1 345	1 347	1 345	1 345	1 344	1 344	1 342	1 346
		TO	8	6	6	6	7	8	7	6	8
	optimised	PAR10	732	633	633	630	731	631	631	632	634
		TO	2	2	2	2	2	2	2	2	2
GRID (80)	default	PAR10	1 509	1 493	1 067	1 266	1 197	1 183	1 053	1 189	1 120
		TO	29	31	31	23	30	26	23	24	27
	optimised	PAR10	34	85	160	14	12	16	18	24	16
		TO	2	1	0	0	0	0	0	0	0
PA (48)	default	PAR10	1 772	1 767	1 765	1 651	1 649	1 649	1 651	1 646	1 648
		TO	11	9	9	9	9	9	8	8	8
	optimised	PAR10	28	38	182	13	35	49	28	18	24
		TO	1	1	1	1	1	1	1	1	1

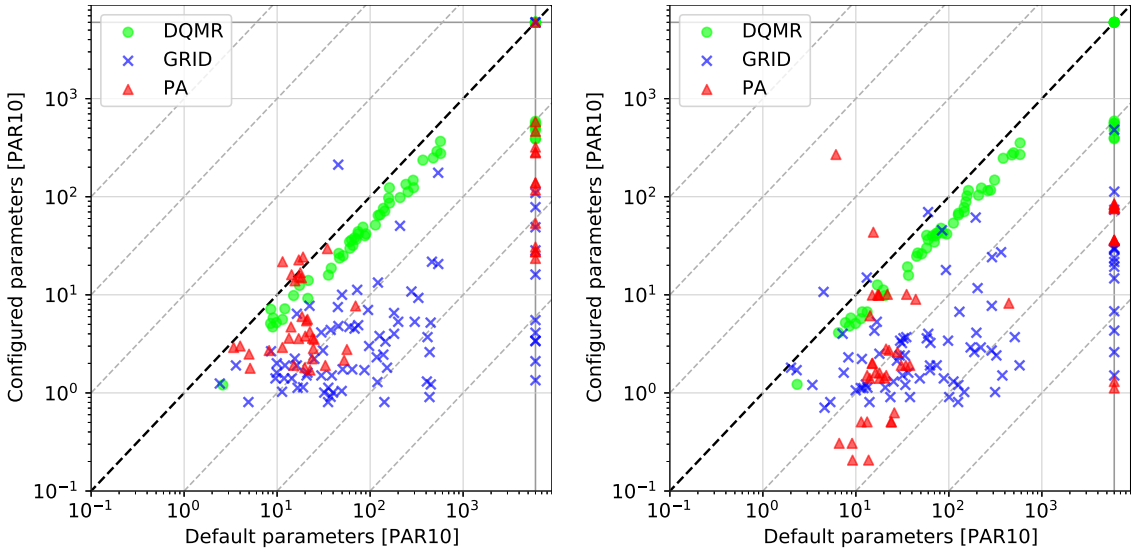
The consistence in performance is confirmed by the individual comparison plots in Figure 5.1. Here, we see that for both cache limits the improvements are quite similar. Only the PA instances seem to be solver quicker for the runs with a cache limit of 8 192*MB*.

5.3 Influence of branching heuristic on cache behaviour

We continue with answering **Q2**. One of the most important parameters in GANAK is the branching heuristic, because it has a direct effect on the size of the search tree. Since each node in the search tree corresponds to a component, it therefore also has a direct influence on the number of components that might enter the cache, which components are found, and in what order. Additionally, much of the effort in improving model counters has been focused on improving branching heuristics [9, 60, 63, 72], sometimes with an explicit emphasis on improving the efficacy of caching, *e.g.*, by aiming to increase the number of cache hits [63], as described in Section 4.5. To the best of our knowledge, in what follows, we are the first to investigate the effect of different branching heuristics on the behaviour of the cache.

In this experiment, we fixed the maximum allowed cache size to 64 MB, to make sure that the cache size was small enough to increase the likelihood to be full at least once during the search. We also fixed the branching heuristic to AUPC, DLCS, DLIS, Centrality, VSIDS, VSADS, or CSVSADS (see Section 4.5). For the combined branching heuristics VSADS and CSVSADS we use the same weights as used by the solvers they appeared in. Then, we automatically configured all of GANAK's other parameters on the *Training* partition of each of the three benchmarks sets of Table 5.1.

For each benchmark and branching heuristic, we subsequently ran GANAK with the appropriate optimised configuration on the instances in the *Test* partition. During these runs, we choose the sample rate for each instance such that the expected number of components that will be mined and traced is



(a) Cachesize 128 MB

(b) Cachesize 8 192 MB

.

Figure 5.1: Running time comparison between default and optimised configurations of GANAK for instances of two benchmarks. Cutoff time: 600 CPU sec.

Table 5.3: PAR10 values (lowest in **bold**, cut-off time is 600 s), average number of components $|\mathcal{C}|$ ($\times$1000) and percentage of components that are encountered exactly once, for the default configuration of GANAK, and optimised configurations with fixed branching heuristic. Cache size restricted to 64 MB.

	DQMR			GRID			PA		
config.	**PAR10**	$\lvert\mathcal{C}\rvert$	$N_e = 1$	**PAR10**	$\lvert\mathcal{C}\rvert$	$N_e = 1$	**PAR10**	$\lvert\mathcal{C}\rvert$	$N_e = 1$
default	1 345	189	74	1 493	7 059	74	1 767	6 236	83
AUPC	2 756	19	71	3 285	6 858	68	1 755	1 246	92
DLCS	1 124	184	74	1 141	5 183	74	2 004	1 513	92
DLIS	**1 117**	315	74	1 142	5 464	74	2 012	1 327	92
Centrality	1 435	3	15	3 084	383	53	1 876	1 558	94
VSIDS	1 548	342	65	**16**	6 314	70	**1 384**	3 798	81
VSADS	1 121	181	74	1 071	6 161	75	1 649	6 368	84
CSVSADS	1 231	157	74	964	7 262	74	1 534	6 080	84

100k (as described in Section 4.2.5). This was done by solving the instance with a cache size of 8 GB and compute how large a proportion 100k is over all component entries to the cache. We used a minimal sample rate of 0.1%.

Table 5.3 shows that, in the default configuration, the majority of the components (74%-83%) are only encountered once; consequently, their counts are added to the cache, and take up space there until they are deleted, but cannot lead to cache hits. After configuration, these percentages remain roughly equal for DQMR and GRID, increase slightly for some PA - branching pairs. Interestingly, we see that the Centrality branching heuristic shows very different behaviour for DQMR and GRID in terms of number of components and the percentage of components that are only encountered once are intriguingly low.

We see that the different benchmark sets need the cache to different degrees. For the default configuration, the solver encounters much less components in the DQMR instances, than in the GRID and PA instances.

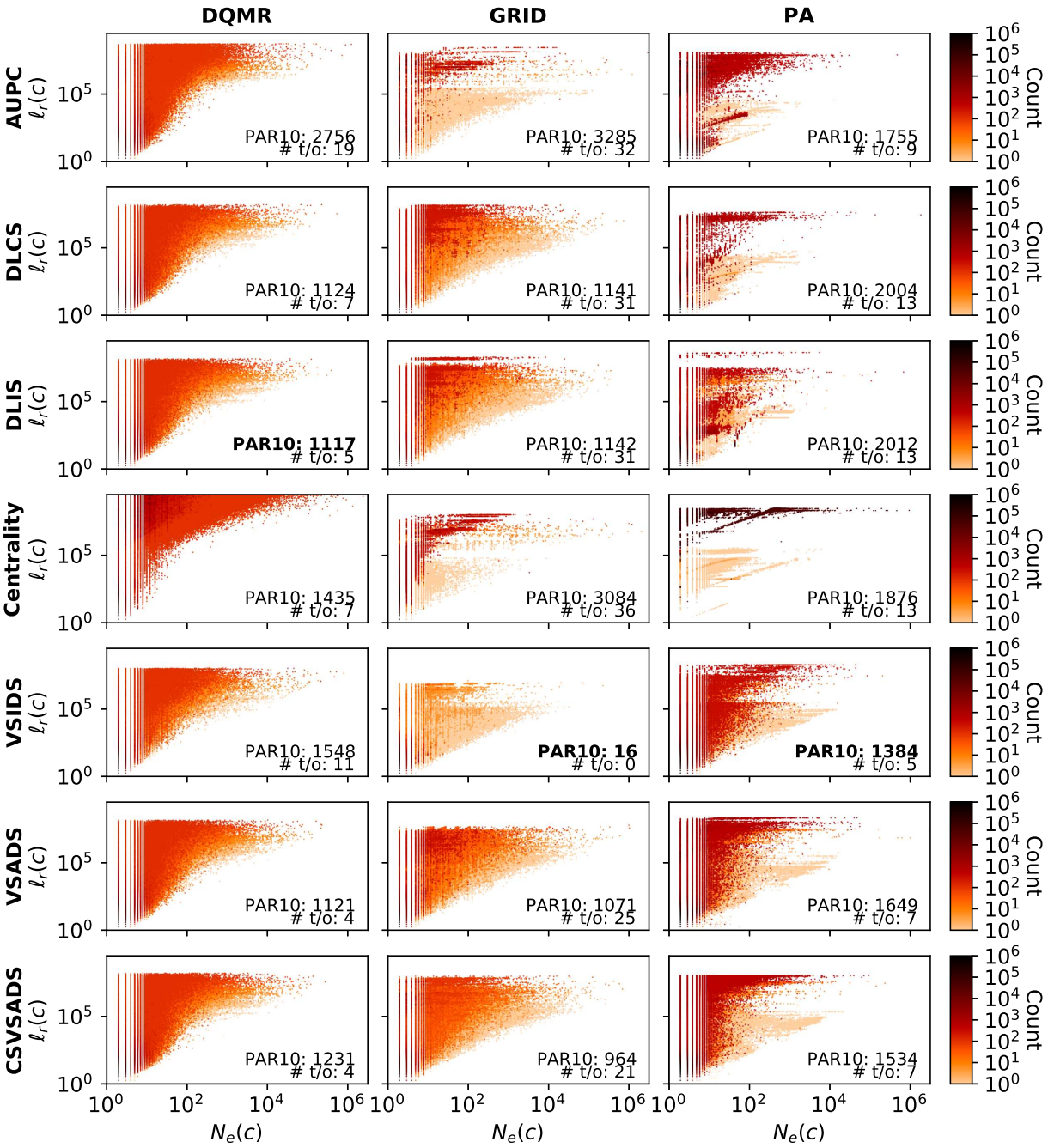


Figure 5.2: Relevance window length ℓ_r as a function of number of encounters N_e, for results in Table 5.3. Figures also show PAR10 value and number of timeouts.

After configuration, this difference tends to remain the same. For different branching heuristics we do see large differences in the number of component encounters. This is not directly tied with PAR10 performance.

We see in this one explanation for why smaller cache sizes do not necessarily yield smaller PAR10 values (**Q2**), especially for GRID: for good enough branching heuristics, the cache is not much needed. This point is underscored by looking at the percentage of component counts that is removed during cache clean-up after configuration. The default is 50%, but this is increased to 97%, 67% and even 71% in the best-performing optimised configurations on the DQMR, GRID and PA benchmarks, respectively.

We interpret this to mean that, for some benchmarks and given good branching heuristics, the benefits of storing counts in the cache for long do not outweigh the costs of having to evaluate those counts for cache removal. For smaller clean-up percentages, we expect the cache to be full more often, and the component counts in that cache to be evaluated to determine if they should be removed more often, even though nothing much might have changed in the mean time.

This suggests that good branching heuristics are perhaps the ones that not only yield small search trees (and thus fewer components), but also search trees in which components are only relevant for a short while, so we do not expect performance to suffer from removing their component counts from the cache.

To test this hypothesis, we take a closer look at these results in Figure 5.2, which are heatmaps in which we visualise how the length of the relevance window $\ell_r(c)$ depends on the number of encounters $N_e(c)$ of each component c. The figure only shows data of the components that were encountered more than once during the search, representing only those components that had the potential to lead to cache hits.

We would expect a successful configuration to lead to smaller values for both N_e and ℓ_r. After all, fewer encounters per component may indicate a smaller search tree. Similarly, a shorter relevance window indicates that component counts can be deleted from the cache sooner, making room for new counts.

This is indeed what we see in Figure 5.2. If we consider, for example, the heatmaps for PA configured with branching fixed to DLCS and VSADS, we see a shift. In the DLCS plot, there are many components with long relevance windows, but few encounters. Consequently, these either are stored in the cache for a long time, without yielding many cache hits, or they are removed in between encounters, and thus have to be recomputed later. However, in the VSADS plot, we see a shift towards the lower left of the square, where components are relevant for a short while, and are encountered only a few times.

From the heatmaps of the Centrality heuristic we see that they have relatively long relevance windows. That can be a possible explanation for why the PAR10 performance of this heuristic is relatively bad.

Recall from Section 4.5 that CSVSADS is the only branching heuristic that directly takes the cache into account. However, the optimised configuration in which we fix the branching heuristic to CSVSADS never outperforms all the other optimised configurations.

Interestingly, we do not see any clear trends indicating which of the cache management schemes yield smaller PAR10 values, we do see that all caching schemes are chosen quite uniformly Figure 5.2, but Marker scheme is chosen most often and HitDecay the least. However, we observe that much more effort has been invested into developing good branching heuristics for #SAT solvers than into developing good cache management heuristics. Especially because, in our experiments, the optimised configurations for the DQMR benchmark are more conservative in their cache clean-ups than the optimised configurations for the GRID and PA benchmark sets. The solver also encounters less components for the DQMR instances than for the others. We conclude that there are problem types for which the solver needs to rely more on the cache than for others, regardless of branching heuristic.

Overall, these results seem to indicate that branching heuristics that yield components that have smaller relevance windows and fewer encounters, yield smaller PAR10 values. This answers **Q3**.

5.4 Influence of caching schemes on cache behaviour

We take a similar approach as before to see how different caching schemes influence the usability of the cache in the solver. We configured the parameters on the *Training* partitions of DQMR, GRID, and PA. In each configuration experiment we fix the caching scheme, which can be FIFO, LRU, HitDecay, Marker, or Random. Other cache related parameters, such as the percentage of the cache that is removed when the cache becomes full remain configurable if applicable.

Interestingly, we see in Table 5.4 that for GRID and PA, the Marker scheme has worse performance, while yielding the best performance for DQMR. The number of encountered components decrease compared to the default. For PA this is even a order of magnitude in decrease of encountered components. This suggests, that the caching schemes we compare here do not have much effect on the number of components that a being encountered.

Again, we make heatmaps to visualise how the length of the relevance window $\ell_r(c)$ depends on the number of encounters $N_e(c)$ of each component c in Figure 5.3. As expected, the heatmaps for each benchmark do not show much differences. We also see that the PAR10 scores are closer to each other, as well as that they are close the best PAR10 scores found after configuring with the complete parameter space, found in Table 5.2.

We do note that the instances of the *Training* partitions of GRID and PA are solved much more efficiently after configuration. The consequence is that for the majority of instances the cache does not become full anymore. This makes the analysis a bit more complicated, and one could argue that a parameter configurator is not able to decide on the best caching scheme in this situation. However, for DQMR we see that caching schemes are not dominantly preferred of the others as well, while here the cache still becomes full once for almost instances. This answers **Q4**

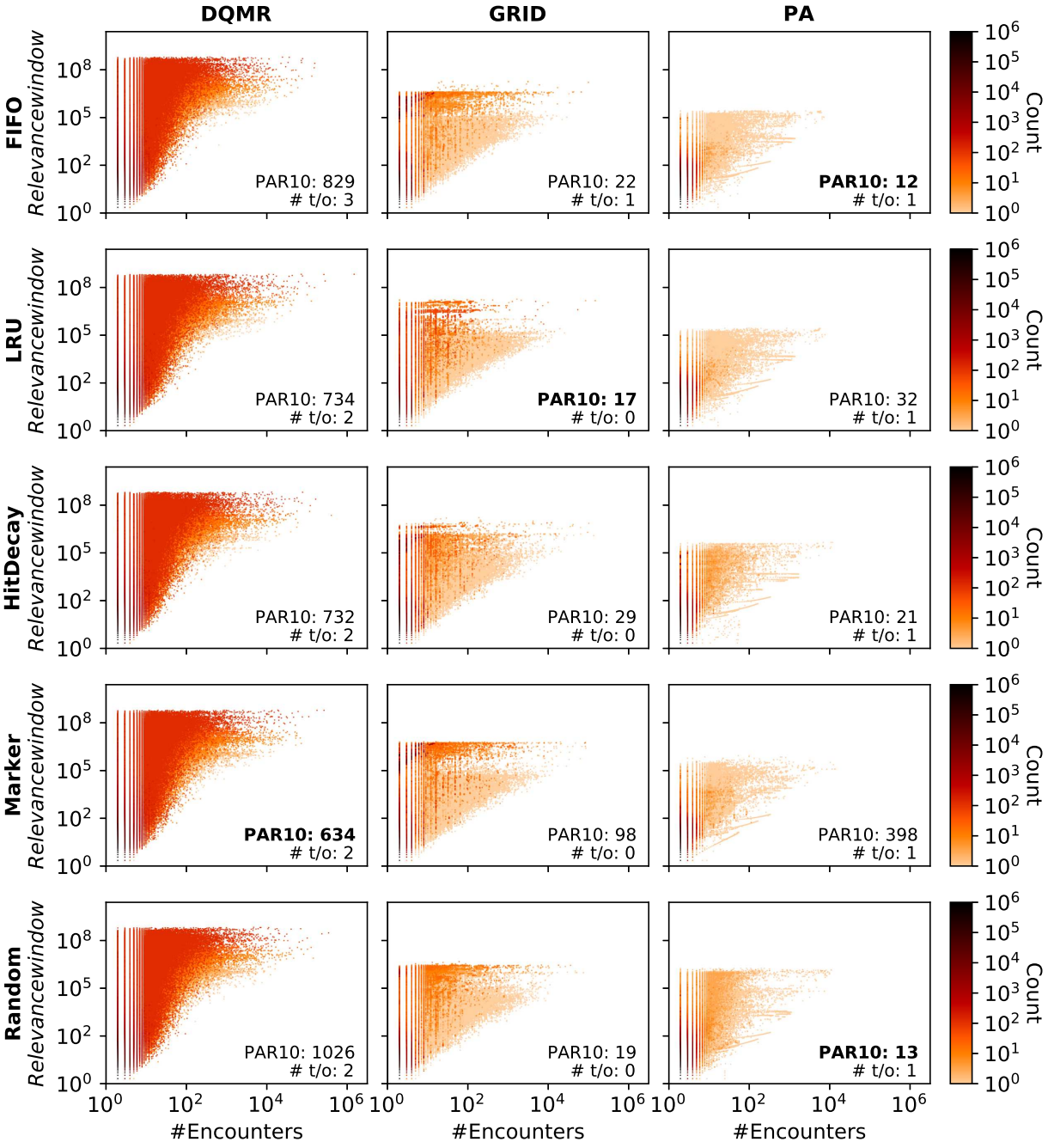


Figure 5.3: Relevance window length ℓ_r as a function of number of encounters N_e, for results in Table 5.4. Figures also show PAR10 value and number of timeouts.

Table 5.4: PAR10 values (lowest in **bold**, cut-off time is 600 s), average number of components $|\mathcal{C}|$ ($\times 1000$) and percentage of components that are encountered exactly once, for the default configuration of GANAK, and optimised configurations with fixed cache management schemes. Cache size restricted to 64 MB.

	DQMR			GRID			PA		
config.	**PAR10**	$\lvert\mathcal{C}\rvert$	$N_e = 1$	**PAR10**	$\lvert\mathcal{C}\rvert$	$N_e = 1$	**PAR10**	$\lvert\mathcal{C}\rvert$	$N_e = 1$
default	1 345	189	74	1 493	7 059	74	1 767	6 236	83
FIFO	829	32	74	22	3 504	68	**12**	727	72
LRU	734	34	74	**17**	6 766	73	32	719	72
HitDecay	732	27	74	29	3 265	66	21	875	70
Marker	**634**	25	74	98	4 218	70	398	822	76
Random	1 026	26	74	19	4 005	69	**13**	671	72

5.5 Predictability of component relevance

The results in Section 5.3 and Section 5.4 motivate our next experiments. In both experiments we saw that the majority of components are only encountered once and, when added to the cache, take up space without contributing to reducing the number of computations to solve the instances. Therefore, we ask whether we can, given a component, we can predict how relevant it is **Q5**. If we could, this indicates that there is room for the development of better caching management heuristics. Note that to obtain better run times, the heuristics should also be efficient to calculate. In this section we explicitly do not consider this dimension, as our focus is on identifying the potential for new cache heuristics; the efficient calculation of these heuristics is left as future work.

We described the experimental setup in section 5.1.4. For each benchmark and for each relevance measure; number of encounter $N_e(c)$s, relevance window length $\ell_r(c)$, and Computational cost savings $R_{cc}(c)$ we train four classifiers. Each of these classifier has a different threshold for which components are labelled as relevant. We choose the thresholds, such that the top percentage of the components for that relevance measure are labelled as relevant. We choose cutoff percentages of 100, 30, 10, 5. In the case of a cutoff percentage of 100, this means that all components that are encountered more than once, have a relevance window of at least 1, or have at least one less decision node, for each relevance metric, respectively.

In Table 5.5 we present the performance metrics of all the classifiers we trained. Since, each classification task has different labels, the scores are not comparable between each other, but we can compare the area under the PR curve against the baseline (imbalance). Here, we see that for all tasks the classifier beats this baseline, often by at least a factor of two. For the tasks with the strictest definitions of relevance (cutoff = 5), we observe even larger differences.

We see that most classifiers have a higher recall than precision on the relevant class. This can be interpreted as that there are proportionally less true negatives compared to false positives. This is favourable, since false positives are components that aren't used and true negatives are components that will save computations, but are seen as not relevant.

We, also look at which and how much features contribute towards making these predictions in the classifiers, since that can be a important signal in what direction new cache management heuristics can be crafted. In Figure 5.4 we show all features that appeared at least twice in the top four of most influential features of each classifier. Here, we see that the normalised number of variables of a component is often the most important feature. We do note that the majority of features that is the most important are problem size features, like the number of (Horn) clauses and are normalised against the corresponding features derived from the input CNF.

Further, we see that often the second most important features is a search state feature. Together, this gives us concrete directions for constructing new cache management schemes.

Table 5.5: Offline classifier performance on held-out validation set for different benchmark, target, cutoff triplets.

Benchmark	Target	Cutoff	Precision	Recall	F1-score	ROC_auc	PR_auc	Imbalance
DQMR	expected	5	0.065	0.384	0.111	0.818	0.058	0.012
		10	0.086	0.492	0.147	0.800	0.100	0.024
		30	0.195	0.661	0.301	0.811	0.266	0.073
		100	0.508	0.774	0.613	0.850	0.651	0.245
	hits	5	0.318	0.594	0.414	0.955	0.403	0.012
		10	0.344	0.600	0.437	0.943	0.432	0.025
		30	0.408	0.651	0.501	0.906	0.504	0.082
		100	0.506	0.779	0.613	0.850	**0.652**	0.245
	longevity	5	0.290	0.489	0.364	0.945	0.302	0.012
		10	0.299	0.547	0.387	0.925	0.351	0.024
		30	0.413	0.517	0.459	0.885	0.445	0.073
		100	0.503	0.779	0.611	0.851	0.650	0.242
GRID	expected	5	0.322	0.751	0.450	0.959	0.514	0.015
		10	0.302	0.830	0.443	0.900	0.563	0.029
		30	0.332	0.870	0.481	0.926	0.629	0.088
		100	0.608	0.829	0.701	0.899	0.806	0.288
	hits	5	0.143	0.960	0.249	0.978	0.429	0.015
		10	0.083	0.726	0.149	0.791	0.411	0.030
		30	0.597	0.935	0.729	0.981	**0.891**	0.110
		100	0.421	0.710	0.529	0.715	0.527	0.288
	longevity	5	0.347	0.995	0.514	0.996	0.743	0.014
		10	0.459	0.975	0.624	0.991	0.762	0.029
		30	0.173	0.647	0.273	0.676	0.425	0.087
		100	0.578	0.826	0.68	0.88	0.752	0.287
PA	expected	5	0.162	0.566	0.252	0.925	0.255	0.013
		10	0.281	0.359	0.315	0.878	0.304	0.026
		30	0.197	0.771	0.314	0.837	0.366	0.079
		100	0.449	0.747	0.561	0.789	**0.569**	0.263
	hits	5	0.378	0.571	0.455	0.953	0.465	0.014
		10	0.285	0.643	0.395	0.938	0.436	0.028
		30	0.254	0.769	0.382	0.872	0.425	0.081
		100	0.435	0.748	0.550	0.779	0.553	0.263
	longevity	5	0.216	0.854	0.345	0.977	0.416	0.013
		10	0.227	0.860	0.360	0.962	0.429	0.026
		30	0.278	0.651	0.390	0.860	0.379	0.079
		100	0.443	0.760	0.560	0.792	0.562	0.260

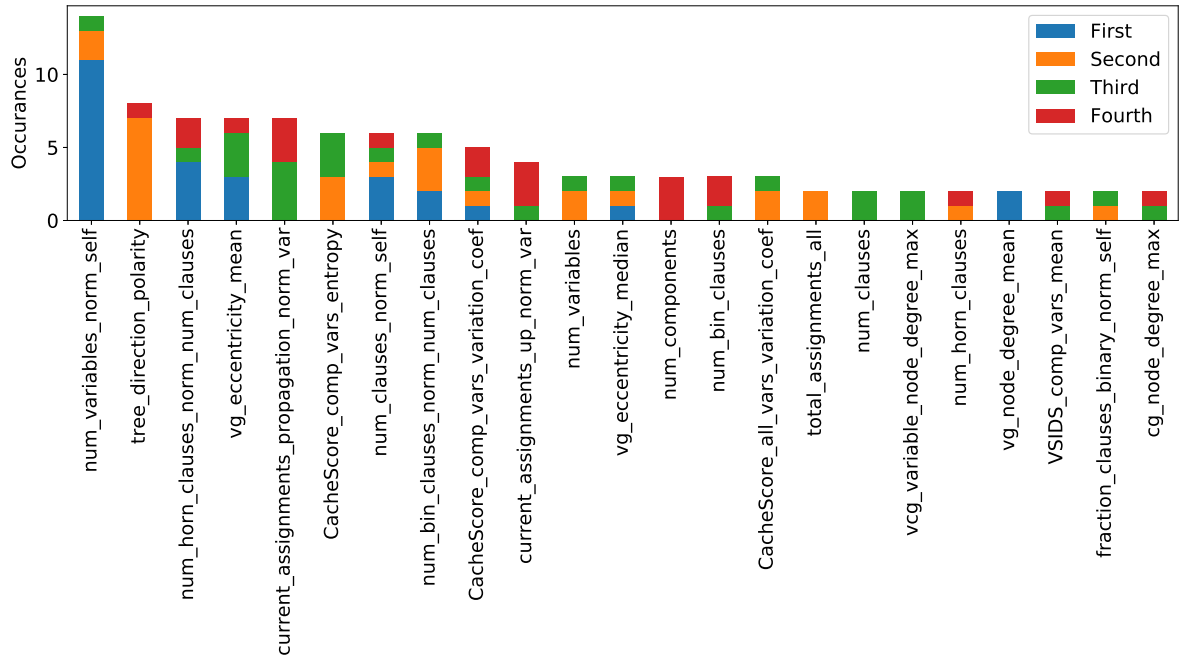


Figure 5.4: Ranking of the occurences features that appear in the top four of most importance features in the classification model at least twice.

Table 5.6: Descriptive statistics over the competitive ratios for the runs with the optimised configuration and one using the predictions of the classifiers to assist the cache scheme. A lower score is better.

	Default		Classifier	
	Mean	Median	Mean	Median
DQMR	1.06	**1.05**	1.12	1.12
GRID	1.07	**1.03**	4.01	2.69
PA	9.51	3.62	1.73	**1.37**

5.6 Online classifier performance

We now answer (**Q6**) by asking ourselves if and how much these classifiers can reduce the number of computations compared to the existing caching schemes. To make this comparison we ran the instances under three different solver configurations. The first run is with optimised parameters and a cache size of 25GB, the second run is with optimised parameters and a cache size of 128MB, and the last run is with optimised parameters, a cache size of 128MB and the cache scheme being substituted by Oracle. We use the first run with the large cache as a baseline to compare the runs of the other two against. This is used as a proxy to estimate the competitive ratios of both runs.

In Table 5.6 we see that each benchmark shows differences in which run is better. For DQMR there is not much difference in the competative ratios. This is likely because DQMR does not rely much on cached components. GRID shows that the classifier advice can be counter productive, whereas for PA we see the opposite.

Chapter 6

Conclusions and future work

In this work, we presented an study on DPLL-based model counters and we investigated the role of component caching in these model counters. By adding branching heuristics, caching schemes that were proposed to work well in model counters and by making parameterising fixed constants in GANAK, we obtained a highly configurable solver that is able to yield massive performance improvements in terms of the CPU time needed to solve instances.

Opposed to what we expected, we showed that increasing the cache size does not always result in reductions in solving times. To understand why, we explored the behaviour of the cache with the rest of the solver under different circumstances. Particularly, we evaluated the cache usage when using different branching heuristics and different caching schemes on three different benchmark sets.

We saw that more than 70% of cached component's counts were encountered again later in the search. These components only take up space in the cache and do not result in less computations to solve the problem. Another finding was that caching schemes do not have a large effect on the ability to solve instances more or less quickly. Using automated algorithm configuration, we found that an effective way of coping with this is to use a branching strategy that ensures that components are only encountered very locally in the search tree.

Furthermore, we showed that by mining features from components and the state of the solver, it is possible to predict if a component will be relevant in the future using three definitions of relevance. When we feed the classifiers directly into the solver we saw that the number of decision nodes the solver sometimes reduced compared to the caching schemes without classification advice. However, classification each component once the cache is full is computationally very expensive and the solving time for solving with classifier advice is not competitive.

From the classifiers we also looked at which features had influenced the predictions the most. Here, we saw that cheap features that describe the component size; number of variables and the number of

(Horn) clauses, are often good predictors for component relevance. This aligns with the smart LRU scheme that was proposed by [48], where the number of variables in components were used as well to cache more cleverly. We believe that this knowledge can provide a good basis for designing new caching schemes that make use of these cheap-to-compute features to triage components on whether or not their counts should be stored in the cache.

6.1 Future work

Based on our finding, we see many directions for future work. In this work, we assessed component relevance on an individual basis. This neglects the fact that the components all originate from the same input CNF, making them dependent from each other. It would be interesting to consider the cache as a whole when designing new caching schemes. One direction could be to only cache components that are on a even depth in the search tree. Alternatively, more features can be made that capture the state of the cache.

Another limitation in this work is that we only used a single component encoding method. It would be interesting to see how different encoding methods affect the caching mechanism in model counters.

To answers our research questions, we built a framework that meticulously builds a snapshot of the search. This framework could me used to find insights on other parts in the solver as well. One example could be on how to schedule the order in which components are solved.

Our approach is designed to provide insights and answers, such that better caching schemes can be designed. An alternative approaches to decide whether or not a component's model count should be cached can be found by casting is as a probabilistic knapsack problem and possibly solved using a predict-and-optimise approach [49].

In this work we learned to predict relevant components in an off-line fashion after we feed those classifiers back into the solver. It would be interesting to incorporate this relevance prediction learning in an on-line fashion during solving problem instances, like is done in [47]. One first step could be an adaptive threshold on, for example, the number of variables in the component, that learns during solving to which size components are more likely to be encountered again.

Bibliography

[1] Aziz, R.A., Chu, G., Muise, C.J., Stuckey, P.J.: #∃sat: Projected model counting. In: SAT. Lecture Notes in Computer Science, vol. 9340, pp. 121–137. Springer (2015)

[2] Bacchus, F., Dalmao, S., Pitassi, T.: DPLL with caching: A new algorithm for #SAT and Bayesian inference. Electronic Colloquium on Computational Complexity (ECCC) **10**(003) (2003)

[3] Bayardo Jr., R.J., Pehoushek, J.D.: Counting models using connected components. In: AAAI/IAAI. pp. 157–162. AAAI Press / The MIT Press (2000)

[4] Bayardo Jr., R.J., Schrag, R.: Using CSP look-back techniques to solve real-world SAT instances. In: AAAI/IAAI. pp. 203–208. AAAI Press / The MIT Press (1997)

[5] Beame, P., Impagliazzo, R., Pitassi, T., Segerlind, N.: Memoization and DPLL: formula caching proof systems. In: Computational Complexity Conference. pp. 248–259. IEEE Computer Society (2003)

[6] Beame, P., Impagliazzo, R., Pitassi, T., Segerlind, N.: Formula caching in DPLL. ACM Trans. Comput. Theory **1**(3), 9:1–9:33 (2010)

[7] Biondi, F., Enescu, M.A., Heuser, A., Legay, A., Meel, K.S., Quilbeuf, J.: Scalable approximation of quantitative information flow in programs. In: VMCAI. Lecture Notes in Computer Science, vol. 10747, pp. 71–93. Springer (2018)

[8] Birnbaum, E., Lozinskii, E.: The Good Old Davis-Putman Procedure Helps Counting Models. Journal of Artificial Intelligence Research **2**(27), 457–477 (1999)

[9] Bliem, B., Järvisalo, M.: Centrality heuristics for exact model counting. In: ICTAI. pp. 59–63. IEEE (2019)

[10] Borodin, A., El-Yaniv, R.: Online computation and competitive analysis. Cambridge University Press (1998)

[11] van Bremen, T., Derkinderen, V., Sharma, S., Roy, S., Meel, K.S.: Symmetric component caching for model counting on combinatorial instances. In: AAAI. pp. 3922–3930. AAAI Press (2021)

[12] Chakraborty, S., Fried, D., Meel, K.S., Vardi, M.Y.: From weighted to unweighted model counting. In: IJCAI. pp. 689–695. AAAI Press (2015)

[13] Chakraborty, S., Meel, K.S., Vardi, M.Y.: A scalable approximate model counter **8124**, 200–216 (2013)

[14] Chakraborty, S., Meel, K.S., Vardi, M.Y.: Balancing scalability and uniformity in SAT witness generator. In: DAC. pp. 60:1–60:6. ACM (2014)

[15] Chakraborty, S., Meel, K.S., Vardi, M.Y.: On the hardness of probabilistic inference relaxations. In: AAAI. pp. 7785–7792. AAAI Press (2019)

[16] Chavira, M., Darwiche, A.: Compiling Bayesian networks with local structure. In: IJCAI. pp. 1306–1312. Professional Book Center (2005)

[17] Chistikov, D., Dimitrova, R., Majumdar, R.: Approximate counting in SMT and value estimation for probabilistic programs. Acta Informatica **54**(8), 729–764 (2017)

[18] Creignou, N., Hermann, M.: Complexity of generalized satisfiability counting problems. Inf. Comput. **125**(1), 1–12 (1996)

[19] Darwiche, A.: A logical approach to factoring belief networks. In: KR. pp. 409–420. Morgan Kaufmann (2002)

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

[21] Darwiche, A., Marquis, P.: A knowledge compilation map. CoRR **abs/1106.1819** (2011)

[22] Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Communications of the ACM **5**(7), 394–397 (jul 1962)

[23] Davis, M., Putnam, H.: A computing procedure for quantification theory. J. ACM **7**(3), 201–215 (jul 1960)

[24] Dueñas-Osorio, L., Meel, K.S., Paredes, R., Vardi, M.Y.: Counting-based reliability estimation for power-transmission grids. In: AAAI. pp. 4488–4494. AAAI Press (2017)

[25] Duffield, N.G., Grossglauser, M.: Trajectory sampling for direct traffic observation. IEEE/ACM Trans. Netw. **9**(3), 280–292 (2001)

[26] Fiat, A., Karp, R.M., Luby, M., McGeoch, L.A., Sleator, D.D., Young, N.E.: Competitive paging algorithms. J. Algorithms **12**(4), 685–699 (1991)

[27] Fichte, J.K., Hecher, M., Hamiti, F.: The model counting competition 2020. CoRR **abs/2012.01323** (2020)

[28] Fichte, J.K., Hecher, M., Zisser, M.: An improved GPU-based SAT model counter. In: CP. Lecture Notes in Computer Science, vol. 11802, pp. 491–509. Springer (2019)

[29] Freeman, L.C.: A set of measures of centrality based on betweenness. Sociometry **40**(1), 35–41 (1977)

[30] Girvan, M., Newman, M.E.J.: Community structure in social and biological networks. Proceedings of the National Academy of Sciences **99**(12), 7821–7826 (2002)

[31] Goldberg, E.I., Novikov, Y.: Berkmin: A fast and robust SAT-solver. In: DATE. pp. 142–149. IEEE Computer Society (2002)

[32] Gomes, C.P., Hoffmann, J., Sabharwal, A., Selman, B.: From sampling to model counting. In: IJCAI. pp. 2293–2299 (2007)

[33] Gomes, C.P., Hoffmann, J., Sabharwal, A., Selman, B.: Short XORs for model counting: From theory to practice. In: SAT. Lecture Notes in Computer Science, vol. 4501, pp. 100–106. Springer (2007)

[34] Gomes, C.P., Sabharwal, A., Selman, B.: Model counting: A new strategy for obtaining good bounds. In: AAAI. pp. 54–61. AAAI Press (2006)

[35] Gupta, R., Roy, S., Meel, K.S.: Phase transition behavior in knowledge compilation. In: CP. Lecture Notes in Computer Science, vol. 12333, pp. 358–374. Springer (2020)

[36] Head, T., Kumar, M., Nahrstaedt, H., Louppe, G., Shcherbatyi, I.: scikit-optimize/scikit-optimize (Sep 2020), `https://doi.org/10.5281/zenodo.4014775`

[37] Hoos, H.H.: Automated algorithm configuration and parameter tuning. In: Autonomous Search, pp. 37–71. Springer (2012)

[38] Hoos, H.H.: Programming by optimization. Communications of the ACM **55**(2), 70–80 (2012)

[39] Huth, M., Ryan, M.: Logic in Computer Science: Modelling and Reasoning About Systems. Cambridge University Press, New York, NY, USA (2004)

[40] Hutter, F., Hoos, H.H., Leyton-Brown, K.: Sequential model-based optimization for general algorithm configuration. In: LION. Lecture Notes in Computer Science, vol. 6683, pp. 507–523. Springer (2011)

[41] Kabanza, F., Filion, J., Benaskeur, A.R., Irandoust, H.: Controlling the hypothesis space in probabilistic plan recognition. In: IJCAI. pp. 2306–2312. IJCAI/AAAI (2013)

[42] Karp, R.M., Luby, M.: Monte-carlo algorithms for the planar multiterminal network reliability problem. J. Complexity **1**(1), 45–64 (1985)

[43] Koriche, F., Lagniez, J., Marquis, P., Thomas, S.: Knowledge compilation for model counting: Affine decision trees. In: IJCAI. pp. 947–953. IJCAI/AAAI (2013)

[44] Lagniez, J., Lonca, E., Marquis, P.: Improving model counting by leveraging definability. In: IJCAI. pp. 751–757. IJCAI/AAAI Press (2016)

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

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

[47] Lykouris, T., Vassilvitskii, S.: Competitive caching with machine learned advice. In: ICML. Proceedings of Machine Learning Research, vol. 80, pp. 3302–3311. PMLR (2018)

[48] Majercik, S.M., Littman, M.L.: Using caching to solve larger probabilistic planning problems. In: AAAI/IAAI. pp. 954–959. AAAI Press / The MIT Press (1998)

[49] Mandi, J., Demirovic, E., Stuckey, P.J., Guns, T.: Smart predict-and-optimize for hard combinatorial optimization problems. In: AAAI. pp. 1603–1610. AAAI Press (2020)

[50] Möhle, S., Biere, A.: Dualizing projected model counting pp. 702–709 (2018)

[51] Möhle, S., Ge, C., Biere, A.: Program analysis benchmarks submitted to the model counting competition mc 2020 (2020), `www.mccompetition.org`, retrieved in August 2020

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

[53] Niemetz, A., M, P., Biere, A.: Boolector 2.0 system description. Journal on Satisfiability, Boolean Modeling and Computation **9**, 53–58 (2015)

[54] Peacock, J.A.: Two-dimensional goodness-of-fit testing in astronomy. Monthly Notices of the Royal Astronomical Society **202**(3), 615–627 (03 1983)

[55] Pearl, J.: Probabilistic reasoning in intelligent systems — networks of plausible inference. Morgan Kaufmann series in representation and reasoning, Morgan Kaufmann (1989)

[56] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., VanderPlas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine learning in python. J. Mach. Learn. Res. **12**, 2825–2830 (2011)

[57] Pelletier, F.J., Martin, N.M.: Post's functional completeness theorem. Notre Dame Journal of Formal Logic **31**(3), 462–475 (1990)

[58] Roth, D.: On the hardness of approximate reasoning. Artif. Intell. **82**(1-2), 273–302 (1996)

[59] Sang, T., Bacchus, F., Beame, P., Kautz, H.A., Pitassi, T.: Combining component caching and clause learning for effective model counting. In: presented at SAT. p. 9 (2004), `http://www.satisfiability.org/SAT04/programme/21.pdf`

[60] Sang, T., Beame, P., Kautz, H.A.: Heuristics for fast exact model counting. In: SAT. Lecture Notes in Computer Science, vol. 3569, pp. 226–240. Springer (2005)

[61] Sang, T., Beame, P., Kautz, H.A.: Performing Bayesian inference by weighted model counting. In: AAAI. pp. 475–482. AAAI Press / The MIT Press (2005)

[62] Sashittal, P., El-Kebir, M.: SharpTNI: Counting and sampling parsimonious transmission networks under a weak bottleneck. bioRxiv (2019). https://doi.org/10.1101/842237

[63] Sharma, S., Roy, S., Soos, M., Meel, K.S.: GANAK: A scalable probabilistic exact model counter. In: IJCAI. pp. 1169–1176. IJCAI (2019)

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

[65] Soos, M., Kulkarni, R., Meel, K.S.: Crystalball: Gazing in the black box of SAT solving. In: SAT. Lecture Notes in Computer Science, vol. 11628, pp. 371–387. Springer (2019)

[66] Stockmeyer, L.J.: The polynomial-time hierarchy. Theor. Comput. Sci. **3**(1), 1–22 (1976)

[67] Stockmeyer, L.J.: On approximation algorithms for #p. SIAM J. Comput. **14**(4), 849–861 (1985)

[68] Takes, F.W., Kosters, W.A.: Computing the eccentricity distribution of large graphs. Algorithms **6**(1), 100–118 (2013)

[69] Thurley, M.: sharpSAT - counting models with advanced component caching and implicit BCP. In: SAT. Lecture Notes in Computer Science, vol. 4121, pp. 424–429. Springer (2006)

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

[71] Tseitin, G.S.: On the Complexity of Derivation in Propositional Calculus, pp. 466–483. Springer Berlin Heidelberg (1983)

[72] Vaezipoor, P., Lederman, G., Wu, Y., Maddison, C.J., Grosse, R.B., Seshia, S.A., Bacchus, F.: Learning branching heuristics for propositional model counting pp. 12427–12435 (2021)

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

[74] Valiant, L.G.: The complexity of enumeration and reliability problems. SIAM J. Comput. **8**(3), 410–421 (1979)

[75] Wei, W., Selman, B.: A new approach to model counting. In: SAT. Lecture Notes in Computer Science, vol. 3569, pp. 324–339. Springer (2005)

[76] Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla: Portfolio-based algorithm selection for SAT. J. Artif. Intell. Res. **32**, 565–606 (2008)

[77] Xu, Z., Zhang, J., Xu, Z., Wang, J.: Canalyze: a static bug-finding tool for C programs. In: ISSTA. pp. 425–428. ACM (2014)

[78] Zhang, L., Madigan, C.F., Moskewicz, M.W., Malik, S.: Efficient conflict driven learning in Boolean satisfiability solver. In: ICCAD. pp. 279–285. IEEE Computer Society (2001)

Appendix A

Parameter Importance Table.

Table A.1: Parameter space. Emphasised rows are constraints that need to hold for the indented parameters below. Parameter values marked with * are excluded in configuration experiments without classifier schemes.

Parameter	Range	Default	Description
	Regular parameters		
PP	{`true`, `false`}	`true`	Preprocessing
LSO	{`true`, `false`}	`true`	Learn-and-Start-Over
IBCP	{`true`, `false`}	`true`	Implicit BCP
componentOrder	ascending, descending, random	ascending	
ccLongIntervalStart	[100,10^5]	10000	Component caching
ccLongIntervalIncreaseFactor	[1,10^4]	10	
ccLongPercentage	[1, 95]	50	
ccShortIntervalStart	[100,10^6]	100000	
	Phase selection parameters		
pol	{`true`, `false`, DLIS, activity, polaritycache}		
branching	{DLCS, DLIS, VSIDS, AUPC, Centrality, Random, Mix}		
branching == Mix			
MixDLCS	{`true`, `false`}	`false`	
MixDLCSweight	[0.01,100]		
MixVSIDS	{`true`, `false`}	`false`	
MixVSIDSweight	[0.01,100]		
MixCentrality	{`true`, `false`}	`false`	
MixCentralityWeightType	{`true`, `false`}		
MixCentralityWeight	[0.01,100]		
branching == VSIDS ∨ MixVSIDS == `true`			
VSIDSdecayInterval			
VSIDSdecayWeight			
EDR	{`true`, `false`}	`false`	
EDR == `true`			
EDRalpha	[0.01, 1]	0.1	
EDRduration	[0,10e6]		
CSVSADS	{`true`, `false`}	`true`	
CSVSADS == `true`			
CSVSADSthreshold	[0.01, 0.95]	0.9	
CSVADSdacayWeight	[0.01, 0.9999]	0.5	
CSVSADSdecayInterval	[8, 262144]	128	
	Caching parameters		
primaryCacheScheme	{FIFO, LRU, HitDecay, SeededHitDecay*, Random, Marker, Oracle*, OracleConfidence*, Hybrid*}	FIFO	
primaryCacheScheme ∈ {Oracle, OracleConfidence, Hybrid}			
secondaryCacheScheme	{FIFO, LRU, HitDecay, SeededHitDecay, Random}		
primaryCacheScheme == Hybrid			
tertairyCacheScheme	{FIFO,LRU,HitDecay, SeededHitDecay, Random}		
primaryCacheScheme != Marker			
cacheDelPerMax	[1,100]	50	
cacheDelPerMin < cacheDelPerMax && primaryCacheScheme ∈ {Oracle,OracleConfidence,Hybrid}			
cacheDelPerMin	[1,100]	10	
primaryCacheScheme == Hybrid			
min_classifier_confidence	[0,1]	0	
primaryCacheScheme == Hybrid			
longer_term_cache_proportion	[0.01, 1]	0.3	
primaryCacheScheme == Hybrid			
reclassifyLongTermCache	{`true`, `false`}	`true`	
primaryCacheScheme in {HitDecay,SeededHitDecay} || secondaryCacheScheme ∈ {HitDecay,SeededHitDecay}			
HitDecayWeight	[0.01,0.9999]	0.5	
SeededHitDecayStart	[1,1000]	50	