Testing for Satisfiability in Modal Logics using a Subset-matching Size-bounded Cache

Enrico Giunchiglia [a] Armando Tacchella [a]

[a] *DIST, V.le Causa 13 16145 Genova, Italy*
E-mail: {enrico,tac}@dist.unige.it

The implementation of efficient decision procedures for modal logics is a major research problem in automated deduction. Caching the result of intermediate consistency checks has experimentally proved to be a very important technique for attaining efficiency. Current state-of-the-art systems implement caching mechanisms based on hash tables.

In this paper we present a data type —that we call "bit matrix"— for caching the (in)consistency of sets of formulas. Bit matrices have three distinguishing features: (*i*) they can be queried for subsets and supersets; (*ii*) they can be bounded in size; and (*iii*) if bounded, they can easily implement different policies to resolve which results have to be kept. We have implemented caching mechanisms based on bit matrices and hash tables in *SAT. In *SAT, the bit matrix cache is bounded, and keeps the latest obtained (in)consistency results. We experiment with the benchmarks proposed for modal logic K at the "TABLEAUX Non Classical Systems Comparison (TANCS) 2000". On the basis of the results, we conclude that *SAT performances are improved by (*i*) caching the result of intermediate consistency checks, (*ii*) using bit matrices instead of hash tables, and (*iii*) storing a small number of results in the bit matrices.

Keywords: Modal logics, Satisfiability, Caching structures

1. Introduction

The implementation of efficient decision procedures for modal logics is a major research problem in automated deduction. Several systems have been developed, like KRIS [1,2], KSAT [3,4], LWB [5], FaCT [6], and more recently *SAT [7,8], KK [9], DLP [10], HAM-ALC [11], KtSeqC [12], and MSPASS [13]. In the last few years, a competition is run in conjunction with the Tableaux conferences, aiming at stimulating the development of effective systems. The last four of the above cited systems participated at the last competition, and —as the organizer Fabio Massacci writes[1]—

> The "winner" is undoubtedly the DLP prover by Peter Patel-Schneider.

[1] See `http://www.dis.uniroma1.it/~tancs/results.shtml`.

As reported in [14], one of the reasons for DLP effectiveness is its ability to cache — for later reuse— the results of intermediate consistency checks of sets of formulas. The DLP caching mechanism is based on hash tables.

In this paper, we present a data type —that we call "bit matrix"— for caching intermediate consistency checks of sets of formulas. Caching mechanisms based on bit matrices and hash tables have been implemented in *SAT. Bit matrices have three distinguishing features:

1. they can be queried for subsets and supersets;
2. they can be bounded in size; and
3. if bounded, they can easily implement different policies to resolve which results have to be kept.

The first feature is important because it allows, e.g., to determine the consistency of any subset of a set of formulas whose consistency has been cached. The second feature allows for limiting (either before or during the execution of the algorithm) the dimension of the data structure. This is important because in many modal logics (e.g., K) the number of consistency checks (and of results to store) can be exponential in the dimension of the input formula. The third feature is important because —depending on the particular class of formulas and/or search strategy implemented by the decision procedure— different policies may turn out to be more effective. Given that *SAT search algorithm is based on chronological backtracking, we keep the latest obtained (in)consistency results. This policy is very simple to implement, and embodies the idea that the oldest results are likely to be the least useful in the future.

We experiment with the benchmarks proposed for modal logic K at the "TABLEAUX Non Classical Systems Comparison (TANCS) 2000". On the basis of the results, we conclude that *SAT performances are improved by

- caching the result of intermediate consistency checks,
- using bit matrices instead of hash tables, and
- storing a small number of results in the bit matrices.

The first result confirms previous experimental analysis (see, e.g., [14]). The second result is due to the fact that *SAT with bit matrices performs less consistency check than *SAT with hash tables. The last result is to be expected given that —as we will see— the cost of querying a bit matrix linearly increases with the number of stored sets.

The paper is structured as follows. In Section 2, we review some notions that are necessary to comprehend the rest of the paper. In particular —besides some basic logic notions— we review the structure of the decision procedure for K implemented in *SAT. In Section 3, we discuss some design issues behind caching for modal logics, and for K in particular. We also show how *SAT decision procedure for K has been modified in order to cache intermediate consistency checks. In Section 4, we present bit matrices and how they meet the issues raised in the previous Section. Section 5 describes how caching mechanisms based on bit matrices and hash tables have been implemented in *SAT. The results of the experimental analysis are reported in Section 6. We end the paper in Section 7 with some conclusions.

This paper builds on and extends [15].

2. Basics

In this Section we briefly review some basic notions necessary for the comprehension of the rest of the paper. See [16] and [8] for more extensive presentations of the topics discussed in the first and in the second Subsections respectively.

2.1. *Definitions and notation*

The set of formulas is constructed starting from a given set of propositional letters and applying the 0-ary operators $\top$ and $\bot$ (representing truth and falsity respectively); the unary operators $\neg$ and $\Box$; and the binary operators $\wedge$, $\vee$, $\supset$ and $\equiv$.

The *modal logic* K is the smallest set of formulas (called *theorems*) closed under tautological consequence and the rule

$$\frac{(\varphi_1 \wedge \ldots \wedge \varphi_n) \supset \psi}{(\Box\varphi_1 \wedge \ldots \wedge \Box\varphi_n) \supset \Box\psi}\,RK$$

with $n \geq 0$ [16].

We say that a conjunction μ of propositional literals and formulas of the form $\Box\varphi$ or $\neg\Box\varphi$ is an *assignment* if, for any pair ψ, ψ' of conjuncts in μ, it is not the case that $\psi = \neg\psi'$. An assignment μ *satisfies* a formula φ if μ entails φ by propositional reasoning. A formula φ is *consistent* or *satisfiable* if $\neg\varphi$ is not a theorem, i.e., if $\neg\varphi \notin$ K.

As notational conventions, we use variously decorated lowercase and uppercase greek letters (e.g., $\varphi, \alpha, \alpha_1, \Delta, \Gamma, \Gamma'$) to denote formulas and sets of formulas respectively. Given a formula φ,

- the *size* of φ is the number of symbols in φ,
- the *modal depth* (or simply *depth*) $d(\varphi)$ of φ is the maximum number of nested $\Box$ operators occurring in φ,
- ψ is a *modal subformula* of φ if $\Box\psi$ occurs in φ.

2.2. *DPL-based algorithm for modal satisfiability*

Consider a formula φ.

Following the SAT-based approach (see, e.g., [3,8]) the problem of determining whether φ is consistent is decomposed in two steps:

function KSAT(φ)
  return KSAT$_{\mathrm{DPL}}$($cnf(\varphi)$, $\top$).

function KSAT$_{\mathrm{DPL}}$(φ, μ)
  if $\varphi = \{\}$ **then return** KCONSIST(μ); /* base */
  if $\{\} \in \varphi$ **then return** *False*; /* backtrack */
  if { a unit clause $\{l\}$ belongs to φ } /* unit propagation */
    then return KSAT$_{\mathrm{DPL}}$($assign(l, \varphi)$,$\mu \wedge l$);
  if not KCONSIST(μ) **then return** *False*; /* early pruning */
  $l :=$ *choose-literal*(φ, μ);
  return KSAT$_{\mathrm{DPL}}$($assign(l, \varphi)$,$\mu \wedge l$) **or** /* split */
      KSAT$_{\mathrm{DPL}}$($assign(\bar{l}, \varphi)$,$\mu \wedge \bar{l}$).

Figure 1. KSAT and KSAT$_{\mathrm{DPL}}$

- *generate* a set of assignments, each satisfying φ, and
- *test* whether at least one of the generated assignment is consistent.

Testing the consistency of an assignment μ amounts to determining the consistency of other formulas whose depth is strictly smaller than $d(\mu)$. This implies that we can check the consistency of these other formulas by recursively applying the above methodology, at the same time ensuring the termination of the overall process. From an implementation point of view, this amounts to two mutually recursive procedures:

- KSAT(φ) for the generation of assignments satisfying φ, and
- KCONSIST(μ) for testing the consistency of each generated assignment μ.

Although the generation of assignments satisfying φ can be based on any procedure for SAT (see [8]), the KSAT procedure implemented in *SAT (represented in Figure 1) is based on the Davis-Putnam-Longemann-Loveland (DPL) procedure [17]. In Figure 1:

- $cnf(\varphi)$ is the set of clauses obtained from φ by applying Plaisted and Greenbaum's conversion [18].
- *choose-literal*(φ, μ) returns a literal occurring in φ and chosen according to some heuristic criterion.
- if l is a literal, $\bar{l}$ stands for A if $l = \neg A$, and for $\neg A$ if $l = A$;
- for any literal l and formula φ, $assign(l, \varphi)$ is the set of clauses obtained from φ by
  - $*$ deleting the clauses in which l occurs as a disjunct, and
  - $*$ eliminating $\bar{l}$ from the others.

function KCONSIST(μ)
  $\Delta := \{\alpha \mid \Box\alpha$ is a conjunct of $\mu\}$;
  $\Gamma := \{\beta \mid \neg\Box\beta$ is a conjunct of $\mu\}$;
  foreach $\beta \in \Gamma$ **do**
    if not KSAT($\bigwedge_{\alpha\in\Delta} \alpha \wedge \neg\beta$) **then return** *False*
  return *True*.

Figure 2. KCONSIST

As can be observed, the procedure KSAT_{DPL} in Figure 1 is the DPL-procedure modulo

- the call to KCONSIST(μ) when it finds an assignment μ satisfying the input formula ($\varphi = \{\}$), and
- the *early pruning* step, i.e., a call to KCONSIST(μ) that forces backtracking after each unit propagation when incomplete assignments are not consistent.

The procedure KSAT is correct and complete, i.e., it returns *True* if the input formula is consistent, and *False* otherwise. See [8].

3. Caching in modal logics

Consider the procedure KCONSIST in Figure 2. Given two assignments μ and μ', it may be the case that KCONSIST(μ) and KCONSIST(μ') perform some equal subtests, i.e., recursive calls to KSAT. This is the case, e.g., when μ and μ' differ only for the propositional conjuncts and there is at least one conjunct of the form $\neg\Box\beta$. To prevent recomputation, the standard solution is to cache both the formula whose consistency is being checked and the result of the check. Then, the cache is consulted before performing each subtest, to determine whether its result can be assessed on the basis of the formulas already checked and stored.

In the following, we assume to have two different caching mechanisms, each using a separate caching structure:

- S-cache to store and query about consistent formulas, and
- U-cache to store and query about inconsistent formulas.

In this way, storing a subtest amounts to storing the formula in the appropriate caching structure. Of course, the issue is how to implement effective caching mechanisms allowing to reduce the number of subtests as much as possible. To this extent, the following considerations are in order:

1. if a formula $\bigwedge_{\alpha\in\Delta'} \alpha \wedge \neg\beta$ has already been determined to be consistent then, if $\Delta \subseteq \Delta'$, we can conclude that also $\bigwedge_{\alpha\in\Delta} \alpha \wedge \neg\beta$ is consistent, and

2. if a formula $\bigwedge_{\alpha\in\Delta'} \alpha \wedge \neg\beta$ has already been determined to be inconsistent then, if $\Delta \supseteq \Delta'$, we can conclude that also $\bigwedge_{\alpha\in\Delta} \alpha \wedge \neg\beta$ is inconsistent.

The above observations suggest the usage of caching structures that can be efficiently queried for subsets and supersets. In other words, given a subtest

$$\textsc{Ksat}(\bigwedge_{\alpha\in\Delta} \alpha \wedge \neg\beta), \tag{1}$$

we want to be able to query our $\mathsf{S\text{-}cache}$ about the presence of a formula

$$\bigwedge_{\alpha\in\Delta'} \alpha \wedge \neg\beta \tag{2}$$

with $\Delta \subseteq \Delta'$ (query for subsets or *subset-matching*). Analogously, given the subtest (1), we want to be able to query our $\mathsf{U\text{-}cache}$ about the presence of a formula (2) with $\Delta \supseteq \Delta'$ (query for supersets or *superset-matching*). In this way, caching a subtest avoids the recomputation of the very same subtest, *and* of the possibly many "subsumed" subtests.

Observations 1 and 2 are independent of the particular modal logic being considered. They are to be taken into account when designing caching structures for satisfiability in any modal logic. Of course, depending on the particular modal logic considered, some other considerations might be in order. For example, in K, we observe that in $\textsc{Kconsist}$ there is a natural unbalance between satisfiable subtests and unsatisfiable ones. In fact, with reference to Figure 2, when testing an assignment μ

3. many subtests can be determined to be satisfiable, all sharing the same set Δ, and

4. at most one subtest may turn out to be unsatisfiable.

Observation 3 suggests that $\mathsf{S\text{-}cache}$ should be able to store satisfiable subtests sharing a common set Δ in a compact way. Therefore, $\mathsf{S\text{-}cache}$ associates the set Δ to the set $\Gamma' \subseteq \Gamma$, representing the "computed" satisfiable subtests $\bigwedge_{\alpha\in\Delta} \alpha \wedge \neg\beta$ for each $\beta \in \Gamma'$. Observation 4 suggests that $\mathsf{U\text{-}cache}$ should not care about subtests sharing a common Δ. Therefore, $\mathsf{U\text{-}cache}$ associates Δ to the single β for which the subtest $\bigwedge_{\alpha\in\Delta} \alpha \wedge \neg\beta$ failed.

Given the design issues outlined above, we modified $\textsc{Kconsist}$ to yield the procedure $\textsc{Kconsist}_{\mathrm{C}}$ shown in Figure 3. In the Figure:

- $\mathsf{U\text{-}cache\_get}(\Delta, \Gamma)$ returns *True* if $\mathsf{U\text{-}cache}$ contains a set Δ' such that $\Delta \supseteq \Delta'$, Δ' is associated with β and $\beta \in \Gamma$;
- $\mathsf{S\text{-}cache\_get}(\Delta, \Gamma)$ returns the set $\Gamma \setminus \Gamma'$, where Γ' is the union over all the sets Γ'' such that for some set $\Delta' \supseteq \Delta$, Γ'' is associated to Δ' in $\mathsf{S\text{-}cache}$.
- $\mathsf{U\text{-}cache\_store}(\Delta, \beta)$ stores in $\mathsf{U\text{-}cache}$ the set Δ and associates β to it;
- $\mathsf{S\text{-}cache\_store}(\Delta, \Gamma)$ stores in $\mathsf{S\text{-}cache}$ the set Δ and associates to it the set Γ.

function $\textsc{Kconsist}_\mathrm{C}(\mu)$
  $\Delta := \{\alpha \mid \Box\alpha$ is a conjunct of $\mu\}$;
  $\Gamma := \{\beta \mid \neg\Box\beta$ is a conjunct of $\mu\}$;
  if U-cache_get(Δ, Γ) **return** *False*;
  $\Gamma_r :=$ S-cache_get(Δ, Γ);
  $\Gamma_s := \emptyset$;
  foreach $\beta \in \Gamma_r$ **do**
    if not $\textsc{Ksat}(\bigwedge_{\alpha\in\Delta} \alpha \wedge \neg\beta)$ **then**
      if $\Gamma_s \neq \emptyset$ **then** S-cache_store(Δ, Γ_s);
      U-cache_store(Δ, β);
      return *False*
    else $\Gamma_s := \Gamma_s \cup \{\beta\}$;
  if $\Gamma_s \neq \emptyset$ **then** S-cache_store(Δ, Γ_s);
  return *True*.

Figure 3. $\textsc{Kconsist}_\mathrm{C}$: consistency checking for K with caching

The new issue is now to implement effective data structures for S-cache and U-cache supporting the above functions. Clearly, we expect that the computational costs associated to the above functions will be superior to the computational costs associated to other caching structures designed for "equality-matching", i.e., effectively supporting the functions obtained from the above by substituting "$\supseteq$" with "$=$". There is indeed a trade-off between "smart but expensive" and "simple but efficient" data-structures for caching. Of course, depending on

- the particular search strategy implemented by the decision procedure,
- the particular logic being considered, and
- the characteristics of the particular formula being tested,

we expect that different caching mechanisms will lead to faster procedures.

Independently from the data-structure being used, the following (last) observation needs to be taken into account when dealing with modal logics whose decision problem is not in NP (e.g., K, S4):

5. testing the consistency of a formula may require an exponential number of subtests.

This is the case for the Halpern and Moses formulas presented in [19] for various modal logics. Observation 5 suggests that it may be necessary to bound the size of the cache, and introduce mechanisms for deciding which formulas to discard when the bound is reached.

$$
(i)\quad \begin{array}{cccccc}
 & 1 & \hat{2} & 3 & \cdots & 32 \\
\psi_1 & 1 & 0 & 0 & \cdots & 0 \\
\psi_2 & 0 & 0 & 0 & \cdots & 0 \\
\psi_3 & 1 & 0 & 0 & \cdots & 0 \\
\psi_4 & 0 & 0 & 0 & \cdots & 0 \\
\psi_5 & 0 & 0 & 0 & \cdots & 0 \\
\psi_6 & 1 & 0 & 0 & \cdots & 0 \\
\psi_7 & 0 & 0 & 0 & \cdots & 0 \\
\psi_8 & 1 & 0 & 0 & \cdots & 0
\end{array}
\qquad
(ii)\quad \begin{array}{cccccc}
 & 1 & 2 & \hat{3} & \cdots & 32 \\
\psi_1 & 1 & 1 & 0 & \cdots & 0 \\
\psi_2 & 0 & 1 & 0 & \cdots & 0 \\
\psi_3 & 1 & 0 & 0 & \cdots & 0 \\
\psi_4 & 0 & 0 & 0 & \cdots & 0 \\
\psi_5 & 0 & 0 & 0 & \cdots & 0 \\
\psi_6 & 1 & 1 & 0 & \cdots & 0 \\
\psi_7 & 0 & 0 & 0 & \cdots & 0 \\
\psi_8 & 1 & 0 & 0 & \cdots & 0
\end{array}
$$

Figure 4. Bit matrix B with $N = 8$, (i) before and (ii) after storing the set $\psi_1 \wedge \psi_2 \wedge \psi_6$ ("$\hat{}$" denotes the current column).

4. Bit matrices for modal satisfiability

Bit matrices are a data type for storing subsets of a finite set of objects (in our case formulas). With reference to the observations made in Section 3, we first show (Subsection 4.1) how bit matrices can be

- queried for subsets and supersets, and
- either unlimited or bound in size.

Thus, bit matrices allow for taking into account the observations 1, 2 and 5. Then, in Subsection 4.2 [resp. 4.3], we show how bit matrices can be adapted for S-cache [resp. U-cache] to take into account observation 3 [resp. 4].

4.1. Bit matrices

A *vector* is a sequence of elements of the same kind. A N*-vector* is a vector with N elements. In the following, Latin letters A,B,C ... are used to denote vectors and, given an N-vector A and a number $i \leq N$, we write A$[i]$ to denote its i-th element.

A $N \times W$ *bit matrix* is an N-vector B, where each B$[i]$ is a bit vector of length W (i.e., is a vector of W bits). Given a $N \times W$ bit matrix B, we write B$[i][k]$ (with $i \leq N$ and $k \leq W$) to denote the k-th bit (k-th column) of the i-th vector (i-th row). Intuitively, N is the cardinality of the set from where the objects are extracted, and W (called the *window size* of B) is the number of subsets that can be stored. W can be either fixed or dynamically increased, allowing different types of storage policies.

To understand storage and retrieval algorithms for bit matrices, consider a finite set Σ of formulas. Let $\psi_1, \psi_2, \ldots \psi_N$ be the elements of Σ, listed according to a fixed enumeration (thus, the cardinality of Σ is $|\Sigma| = N$). Then, we have an $N \times W$ bit matrix B. We say that a bit B$[i][k]$ with $i \leq N$ and $k \leq W$ is *set* when B$[i][k]$= 1 and that it is *clear* otherwise. The *current column* $\hat{k}$ is a variable and tells us where the incoming set is going to be stored. Initially, $\hat{k} = 1$ and each B$[i][k]$ is clear.

Storing a set Δ: Given that an $N \times W$ bit matrix can contain up to W subsets, there are two cases depending on whether the matrix is full and new subtests need to be stored, or not. In order to distinguish these two cases, we use a boolean variable *full*. If *full* is false, then we set each bit B$[i][\hat{k}]$ if and only if $\psi_i \in \Delta$. Then, $\hat{k}$ is incremented and, if $\hat{k} = W + 1$, *full* is set to true. If *full* is true, then we can

- either dynamically allocate more space, and allow more sets to be cached without overwriting previous data. In this case, we set *full* to false, $\hat{k}$ to $W + 1$ and W to the new value determined by the dynamic allocation. Then we apply the procedure above;
- or overwrite previously cached data with some policy, generally depending on the contents of B. In other words, we simply fix $\hat{k}$ to be $f(\mathrm{B})$, where f is a given function that decides which sets are to be overwritten. Then, we set each bit B$[i][\hat{k}]$ such that $\psi_i \in \Delta$, and clear the others.

For example, assume that Σ consists of 8 formulas $\psi_1, \psi_2, \ldots \psi_8$ listed according to a fixed enumeration. Intuitively, we consider an 8×32 bit matrix B, in which each B$[i]$ corresponds to ψ_i. Figure 4 - (i), (in which each row index i has been replaced by the corresponding formula ψ_i) shows the case in which the set $\{\psi_1, \psi_3, \psi_6, \psi_8\}$ is already stored and $\hat{k} = 2$. Assume that we are to store $\{\psi_1, \psi_2, \psi_6\}$ in B. As depicted in Figure 4 - (ii) we consider the current column ($\hat{k} = 2$) and we set the bits B[1][2], B[2][2] and B[6][2]. The current column becomes $\hat{k} + 1$.

Subset matching for a set Δ: For the retrieval, we first define the intersection between two bit vectors A_1 and A_2 as the bit vector $\mathrm{A} = (\mathrm{A}_1 \cap \mathrm{A}_2)$ such that A$[i]$ is set if and only if both $\mathrm{A}_1[i]$ and $\mathrm{A}_2[i]$ are set. We assume that the bit matrix B is not empty, i.e., that there is at least one element already stored. To see if there exists a set Δ' such that $\Delta \subseteq \Delta'$ is stored in B, it is sufficient to consider the conjunction C of all the bit vectors B$[i]$ such that $\psi_i \in \Delta$, i.e., $C = \bigcap_{\psi_i \in \Delta} \mathrm{B}[i]$. C is a bit vector and, if at least one of its elements is set, then $\Delta \subseteq \Delta'$ for some set Δ' previously stored. We call this a *hit*. Otherwise, if every element of C is clear we have a *miss*. Notice how bit matrices naturally allow for subset-matching. This capability comes at no additional cost since it is an intrinsic property of the retrieval algorithm. As an example of hit, considering B in Figure 4 - (ii) and $\Delta = \{\psi_1, \psi_6\}$, we have that $C = (\mathrm{B}[1] \cap \mathrm{B}[6])$. Since C[1] and C[2] are set, we conclude that B contains two supersets of Δ and this is clearly the case in Figure 4 - (ii).

Superset matching for a set Δ: Superset-matching can be reduced to subset-matching for Δ by exploiting the following set-theoretic property: given a finite set Σ and two sets $\Delta, \Delta' \subseteq \Sigma$ we have that $\Delta \supseteq \Delta'$ if and only if $(\Sigma \setminus \Delta) \subseteq (\Sigma \setminus \Delta')$.

We remark that different policies to handle the window size (either resizable or fixed) have an impact both on the space occupied by the structure and on the time spent for retrieval operations. Clearly,[2] with B and Σ as above, the space occupied by the bit matrix is $O(NW)$ and the basic retrieval operation, i.e., $\bigcap_{\psi_i \in \Delta}$ B$[i]$, is $O(NW)$. If we let W grow, it is possible that the retrieval time becomes unacceptably high: if all the subsets of Σ are stored, retrieval is $O(N2^N)$. On the other hand, if we know in advance that the number of stored sets will be small, we can save memory by beginning with $W = 0$ and extending it on demand only. If we leave W fixed we may need to overwrite existing data. In this case, if W is too small, trashing may occur, i.e., relevant data may be lost too early. If W is too big, memory space is wasted and retrieval time is unnecessarily increased.

4.2. Bit matrices for S-cache

Consider a formula φ. Let $\psi_1, \ldots, \psi_N$ be a listing of the modal subformulas of φ. Consider an assignment μ satisfying φ generated by KSAT. Consider the routine KCONSIST$_{\text{C}}$ in Figure 3. We write α_i (resp. β_j) for ψ_i (resp. ψ_j).

In order to take into account observation 3, S-cache can be implemented as a bit matrix B with $2N$ rows. Every column of B encodes the satisfiable subtests resulting from a single call to KCONSIST$_{\text{C}}$: given a column k and $i \leq N$, B$[i][k]$ is set if and only if $\alpha_i \in \Delta$, while B$[N+j][k]$ is set if and only if both $\beta_j \in \Gamma_r$ and the subtest $\bigwedge_{\alpha_i \in \Delta} \alpha_i \wedge \neg\beta_j$ is known to be satisfiable. Rows in the range $[1, N]$ encode members of Δ, while rows in the range $[N+1, 2N]$ encode members of Γ_r.

The bit matrix based algorithm for S-cache_get(Δ, Γ) is:

1. compute the conjunction C of all the bit vectors B$[i]$ such that $\alpha_i \in \Delta$, i.e., C$= \bigcap_{\alpha_i \in \Delta}$ B$[i]$: this step takes time $|\Delta| W$;
2. initialize $\Gamma_r := \Gamma$;
3. for each $\beta_j \in \Gamma$ and for each k such that C$[k]$ is set, test if B$[N+j][k]$ is set and, if so, $\Gamma_r := \Gamma_r \setminus \{\beta_j\}$. Since deletion of β_j can be performed in constant time, this step[3] takes time $|\Gamma| s + W$ where s is the number of detected supersets of Δ;
4. return Γ_r.

For S-cache_store(Δ, Γ_s) we have:

1. take the current column $\hat{k}$ and clear each bit B$[i][\hat{k}]$ for $i \leq 2N$. Clearly, this step takes time $2N$;

[2] We recall that $O(g(n)) = \{\text{f(n)} |$ there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$. We say that $f(n)$ is $O(g(n))$ and we write $f(n) = O(g(n))$ meaning that $f(n)$ is a member of $O(g(n))$.

[3] The calculation is done in the case of a hit. In the case of a miss the step takes time W.

2. set each bit $\mathrm{B}[i][\hat{k}]$ such that $\alpha_i \in \Delta$. This step takes time $|\Delta|$;

3. for each $\beta_j \in \Gamma_s$ set $\mathrm{B}[N+j][\hat{k}]$. Clearly, this step takes time $|\Gamma_s|$.

Summarizing, bit matrices allow us to obtain an algorithm for S-cache_get that takes time $|\Delta|W + |\Gamma_r|s + W \leq NW + Ns + W \leq (2N+1)W = O(NW)$ and an algorithm for S-cache_store that takes time $2N + |\Delta| + |\Gamma_s| \leq 4N = O(N)$. The time required to cache *all* the satisfiable subtests of μ is $O(N)$: if we were using a bit matrix associating a single formula to each Δ, this time would have been $O(N^2)$.

4.3. Bit matrices for U-cache

Consider the hypotheses written in the first paragraph of the previous Subsection.

In order to take into account observation 4, U-cache can be implemented using a bit matrix B of N rows, and a vector D of W elements. Every column of B encodes part of an unsatisfiable subtests resulting from a call to $\text{K}\textsc{consist}_\text{C}$: given a column k and $i \leq N$, $\mathrm{B}[i][k]$ is set if and only if $\alpha_i \in \Delta$. The index j such that $\bigwedge_{\alpha_i \in \Delta} \alpha_i \wedge \neg\beta_j$ is unsatisfiable, is stored in $\mathrm{D}[k]$. As before, rows in the range $[1, N]$ encode members of Δ, but we need a single number to remember β_j.

The bit matrix based algorithm for U-cache_get(Δ, Γ) is:

1. compute the conjunction C of all the bit vectors $\mathrm{B}[i]$ such that $\alpha_i \notin \Delta$, i.e., $\mathrm{C}= \bigcap_{\alpha_i \notin \Delta} \mathrm{B}[i]$. This step takes time $(N - |\Delta|)W$;

2. for each $\beta_j \in \Gamma$ and for each k such that $\mathrm{C}[k]$ is set, test if $\mathrm{D}[k]= j$ and, if so, return *True*. This step takes time $W + |\Gamma|s$ where s is the number of detected subsets of Δ;[4]

3. return *False*;

For U-cache_store(Δ, β_j) we have:

1. take the current column $\hat{k}$ $(k < W)$ and for $i \leq N$ set each bit $\mathrm{B}[i][\hat{k}]$. This step takes time N;

2. clear each bit $\mathrm{B}[i][\hat{k}]$ such that $\alpha_i \in \Delta$. This step takes time $|\Delta|$;

3. $\mathrm{D}[\hat{k}] := j$. Clearly, this step takes constant time.

Summarizing, bit matrices allow us to obtain an algorithm for U-cache_get that takes time $(N - |\Delta|)W + W + |\Gamma|s \leq NW + W + Ns \leq (2N+1)W = O(NW)$ and an algorithm for U-cache_store that takes time $N + |\Delta| \leq 2N = O(N)$.

[4] The calculation is done in the case of a hit. In the case of a miss the step takes time W.

5. Implementation in *sat

We implemented in our system *SAT[5] the data structures and algorithms presented in Sections 3 and 4. *SAT is a platform for building and experimenting SAT-based decision procedures for modal logics (see [7] and [8]). The core of *SAT is a C implementation of the procedures KSAT and KCONSIST$_{\mathrm{C}}$ in Figures 1 and 3.

Both bit matrices and hash tables can be used interchangeably. In the following, φ is the formula in input, and N is the number of subformulas in φ.

Caching with bit matrices in *SAT is implemented according to the data structures and the algorithms presented in Section 4, with the following assumptions:

1. only a small number of subtest results are useful, and
2. the latest obtained results are more useful.

Assumption 1 suggests a size-bounded cache, i.e., W is fixed to a value defined by the user. We recall that caching all the subtests in modal logic K may lead to an exponential increase of W (Section 3) and the running time of retrieval algorithms for bit matrices linearly depends on W (Section 4). Since W is fixed, *SAT features $O(N)$ implementations of S-cache_get and U-cache_get (S-cache_store and U-cache_store are $O(N)$ even if W is dynamically increased). A fixed window size also implies that, whenever the number of subtests exceeds W, we must resolve which of the previously stored results need to be overwritten. Assumption 2 suggests that we can tackle this situation by accessing the bit matrix with a first in - first out policy. Results are stored in chronological order and, when the window size is exceeded, the latest obtained result overwrites the first, then the second and so on. With bit matrices, this access policy is implemented simply by incrementing $\hat{k}$ modulo W, i.e., whenever $\hat{k} > W$ the next current column is $\hat{k} := 1$. One last point to consider, is the (real) memory allocated by bit matrices in *SAT. We say that a formula ψ *occurs at level* l *in* φ if ψ occurs under the scope of exactly l nested boxes in φ ($l \geq 0$). Now we can state the following assumption:

3. it is unlikely that a subtest involving formulas of a given level is recomputed with the same formulas occurring at a different level.

Then, S-cache and U-cache are implemented in *SAT as vectors of $d(\varphi)$ bit matrices. Each one holds the results of subtests involving formulas occurring at a given level in φ. Clearly, the memory allocation for such structures is $O(Nd(\varphi))$.

Caching with hash tables in *SAT is implemented by replacing subset and superset-matching with equality matching in the algorithms presented in Section 3. The data structure is a standard hash table: a vector H of length N

[5] You can find *SAT latest version and available documentation at:
http://www.mrg.dist.unige.it/~tac/StarSAT.html

whose elements are lists called *buckets*. A suitable function $h(\Delta) \leq N$ is used to calculate the hash code associated to each set Δ. Two sets $\Delta \neq \Delta'$ may share the same hash code, i.e., $h(\Delta) = h(\Delta')$ and they are both stored in the bucket $H[h(\Delta)]$. In S-cache (resp. U-cache) a set Γ is associated to each Δ such that for each $\beta \in \Gamma$ the subtest $\bigwedge_{\alpha \in \Delta} \alpha \wedge \neg\beta$ is consistent (resp. inconsistent). Worst case storage and retrieval time are $O(1)$ and $O(2^N)$ respectively. In practice, what makes hash tables interesting is the *expected* retrieval time, that is $O(M)$ where $M << 2^N$ is the average bucket size. Worst case memory allocation for hash tables is $O(2^N)$, but, again, we expect it to be $O(MN)$. More precisely, the expected memory allocation is $O(MNd(\varphi))$ because, exploiting assumption 3 above, *SAT features a vector of length $d(\varphi)$ of hash tables.

*Selection of a caching method in *SAT* can be done via the command line switch `-m<n>`. In the following we assume that *SAT is used as a decision procedure for modal K. When *SAT is run with all caching optimization turned off, we denote it with kSAT. Other choices are:

- `n=1,2,3` activates caching with bounded bit matrices. In this case the window size can be modified with the `-w<m>` switch. The default window size is $W = 512$, and there are three caching schemas: with `n=1` only S-cache is activated, with `n=2` only U-cache is activated, and `n=3` activates both. We denote *SAT featuring bit matrices with kSAT+SB if `n=1`, kSAT+UB if `n=2`, and kSAT+USB if `n=3`.
- `n=4,5,6` activates caching with unbounded hash tables. Again, with `n=4` only S-cache is activated, with `n=5` only U-cache is activated, and `n=6` activates both. We denote *SAT featuring hash tables with kSAT+SH if `n=4`, kSAT+UH if `n=5`, and kSAT+USH if `n=6`.

6. Experimental analysis

To test the effectiveness of the caching mechanisms that we have described, we consider kSAT, kSAT+USB and kSAT+USH performances on the TANCS 2000 benchmarks problems, category "MODAL PSPACE Division", sub-category "Problems for Modal QBF", classes "Easy/Medium", "Modalized Easy/Medium", "Medium/Hard", "Modalized Medium/Hard".[6] The name of the sub-category refers to the fact that these formulas are generated starting from Quantified Boolean Formulas (QBFs). QBFs are generated according to the following parameters: the number of clauses C, the alternation depth D, and the maximum number of variables allowed for each alternation V. Each clause consists of four literals, and care is taken in order to avoid the generation of trivial tests. Then, each QBF is translated into a modal formula: the class to which

[6] They can be retrieved at `ftp://ftp.dis.uniroma1.it/pub/tancs/problems/qbf-cnf-tancs.tar.gz`.

the resulting formula belongs, depends on the encoding being used. Currently, six encodings are possible:

1. "vanilla": this encoding is based on Ladner's original translation [20], enhanced with some smart tricks so that no extra variable is introduced. The resulting formulas are considered to be the hardest (of "medium/hard" difficulty) among the distributed benchmarks.
2. "Ladner": also this encoding is based on Ladner translation. However, the resulting formulas are considered to be of "medium" difficulty because they have been simplified by performing some modal resolution step.
3. "SSS": this encoding is similar to the Schmidt-Schauss and Smolka translation [21]. These formulas are considered to be of "easy/medium" difficulty because in the encoding process they are simplified even further by doing some extra modal resolution steps.

By setting a switch, each of the above three encodings can be "modalized" to produce formulas which are considered to be harder than the originals. In the modalized version, each propositional variable is replaced with a special modal formula with only one propositional variable. In the following we say that a formula is "easy/medium", "medium", "medium/hard" to mean that it has been generated with the SSS, Ladner, vanilla encoding respectively. When a formula has been generated with an encoding having the modalizing switch on, we specify that the formula is "modalized".

Each class consists of various sets of formulas. Each set

- corresponds to one setting of the parameters C, V and D, and
- consists of 8 formulas, except when they are medium/hard or modalized medium/hard. In these last two cases, the formulas in the set are 4.

See [22] for more details.

We performed the tests on a Personal Computer `PII350MHz` with `256 MbRAM`, running `Linux SUSE 6.2`. For practical reasons, the execution of each system on a formula has been stopped either because it reached a timelimit fixed to 1200 seconds of CPU time, or because it exhausted the 256MB of RAM available on the computer. To save space, here we consider only the results on the easy/medium, modalized easy/medium, medium/hard and modalized medium/hard formulas. For the medium/hard formulas we do not report the results on all the sets, but only those for which we have some result either in the non-modalized or in the modalized version of the set. See *SAT home page for the performances of KSAT+USB on the medium and modalized medium formulas.

To help focusing, we structure the analysis of the results in three Subsections. In the first, we show that caching the result of intermediate consistency checks improves *SAT performances, independently from whether we are using bit matrices or hash tables. In the second, we analyze the differences in *SAT

C	V	D	Sz	d	Ms	kSAT S/U/F	kSAT Cpu	kSAT Cons	kSAT+USB S/U/F	kSAT+USB Cpu	kSAT+USB Cons	kSAT+USH S/U/F	kSAT+USH Cpu	kSAT+USH Cons
10	4	4	7202	20	1013	0/0/4	–	–	4/0/0	102.05	26663	4/0/0	133.82	41898
20	4	4	8471	20	1259	0/0/4	–	–	3/1/0	321.15	75967	3/1/0	493.66	131996
30	4	4	9289	20	1418	0/3/1	87.50	53472	0/4/0	18.08	3953	0/4/0	29.36	8253
10	8	4	31281	40	2728	0/0/4	–	–	0/0/4	–	–	0/0/4	–	–
20	8	4	247845	80	11275	0/0/4	–	–	0/0/4	–	–	0/0/4	–	–
30	8	4	308422	80	14553	0/0/4	–	–	0/0/4	–	–	0/0/4	–	–
10	4	6	15698	28	1717	0/0/4	–	–	1/0/3	981.05	154892	0/0/4	–	–
20	4	6	19395	28	2284	0/0/4	–	–	0/0/4	–	–	0/0/4	–	–

Table 1
kSAT, kSAT+USB, and kSAT+USH on TANCS 2000 medium/hard problems

C	V	D	Sz	d	Ms	kSAT S/U/F	kSAT Cpu	kSAT Cons	kSAT+USB S/U/F	kSAT+USB Cpu	kSAT+USB Cons	kSAT+USH S/U/F	kSAT+USH Cpu	kSAT+USH Cons
10	4	4	20452	40	973	0/0/4	–	–	0/4/0	30.20	20398	0/4/0	101.46	98330
20	4	4	27137	41	1311	0/0/4	–	–	0/4/0	28.99	12796	0/4/0	176.98	134319
30	4	4	30140	41	1479	0/0/4	–	–	0/4/0	48.49	28928	0/3/1	338.78	197789
10	8	4	459144	159	7056	0/0/4	–	–	0/3/1	704.37	128334	0/0/4	–	–
20	8	4	95742	79	2761	0/0/4	–	–	0/0/4	–	–	0/0/4	–	–
30	8	4	140703	81	4138	0/0/4	–	–	0/1/3	727.93	96175	0/0/4	–	–
10	4	6	46054	57	1773	0/0/4	–	–	0/3/1	64.98	17100	0/4/0	155.37	140415
20	4	6	59526	57	2312	0/0/4	–	–	0/1/3	295.58	213999	0/1/3	875.62	664688

Table 2
kSAT, kSAT+USB, and kSAT+USH on TANCS 2000 modalized medium/hard problems

performances when using bit matrices instead of hash tables. As we said in the introduction, the former lead to better performances than the latter. Finally, we further analyze bit matrices, and show that the window size W is indeed a critical parameter which affects kSAT+USB behavior. However, on the tests that we consider here, fixing W to 512 seems to be a good choice on the average.

6.1. *Caching vs no caching*

We first consider the medium/hard (Table 1 and modalized medium/hard formulas (Table 2). In both Tables, for each set of formulas corresponding to one setting of C, V, D, we report the geometric mean of

- the size (column "Sz"),
- the depth (column "d"), and
- the number of modal subformulas (column "Ms")

of the formulas in the set. The geometric mean is the way to average results at TANCS. Then we show the results for kSAT, kSAT+USB, and kSAT+USH on the various sets. For each system, we report

						kSAT			kSAT+USB			kSAT+USH		
C	V	D	Sz	d	Ms	S/U/F	Cpu	Cons	S/U/F	Cpu	Cons	S/U/F	Cpu	Cons
10	4	4	378	20	183	8/0/0	7.76	115566	8/0/0	0.08	246	8/0/0	0.07	302
20	4	4	637	20	336	6/2/0	38.68	353029	6/2/0	0.48	945	6/2/0	0.54	1494
30	4	4	907	20	481	2/6/0	17.17	120200	2/6/0	0.66	1036	2/6/0	0.80	1831
40	4	4	1157	20	610	1/7/0	23.66	111885	1/7/0	0.96	1268	1/7/0	1.30	2553
50	4	4	1403	20	734	0/8/0	9.32	37074	0/8/0	0.66	726	0/8/0	0.80	1374
10	8	4	650	39	375	0/0/8	–	–	8/0/0	0.25	643	8/0/0	0.22	783
20	8	4	1100	40	701	0/0/8	–	–	8/0/0	3.28	4037	8/0/0	4.98	8991
30	8	4	1503	40	979	0/0/8	–	–	8/0/0	42.72	30507	8/0/0	91.49	98399
40	8	4	1910	40	1265	0/0/8	–	–	2/6/0	58.20	32698	2/6/0	162.45	134390
50	8	4	2323	40	1529	0/0/8	–	–	0/8/0	75.71	37776	0/8/0	206.99	145379
10	16	4	1189	77	754	0/0/8	–	–	8/0/0	0.46	906	8/0/0	0.48	1203
20	16	4	1966	79	1397	0/0/8	–	–	8/0/0	6.57	6684	8/0/0	9.97	14769
30	16	4	2723	80	2013	0/0/8	–	–	8/0/0	83.77	43580	7/0/1	177.23	138971
40	16	4	3404	80	2553	0/0/8	–	–	2/0/6	604.25	234336	0/0/8	–	–
50	16	4	4212	79	3208	0/0/8	–	–	0/0/8	–	–	0/0/8	–	–
10	4	6	498	28	266	6/0/2	155.18	2156024	8/0/0	0.16	410	8/0/0	0.19	686
20	4	6	827	28	480	0/0/8	–	–	8/0/0	1.78	2437	8/0/0	2.85	6009
30	4	6	1145	28	680	0/0/8	–	–	7/1/0	7.46	7435	7/1/0	14.79	22988
40	4	6	1453	28	865	0/1/7	349.22	2147620	2/6/0	5.83	5396	2/6/0	9.45	13295
50	4	6	1789	28	1057	0/1/7	256.84	1080890	0/8/0	4.66	3755	0/8/0	11.86	13772
10	8	6	853	54	509	0/0/8	–	–	8/0/0	0.29	679	8/0/0	0.27	839
20	8	6	1439	55	971	0/0/8	–	–	8/0/0	4.81	4808	8/0/0	8.10	11784
30	8	6	2000	56	1397	0/0/8	–	–	8/0/0	90.65	52982	8/0/0	225.74	200873
40	8	6	2573	56	1835	0/0/8	–	–	4/1/3	396.80	163496	1/0/7	369.96	273971
50	8	6	3061	56	2196	0/0/8	–	–	0/0/8	–	–	0/0/8	–	–
10	16	6	1557	108	998	0/0/8	–	–	8/0/0	0.73	1316	8/0/0	0.67	1648
20	16	6	2626	110	1918	0/0/8	–	–	8/0/0	8.93	6701	8/0/0	14.13	16686
30	16	6	3693	111	2840	0/0/8	–	–	8/0/0	176.68	72254	6/0/2	304.58	193313
40	16	6	4699	112	3694	0/0/8	–	–	4/0/4	561.58	168941	0/0/8	–	–
50	16	6	5674	111	4518	0/0/8	–	–	0/0/8	–	–	0/0/8	–	–

Table 3
kSAT, kSAT+USB, and kSAT+USH on TANCS 2000 easy/medium problems

- the number of instances determined to be satisfiable by the system (column "S"),
- the number of instances determined to be unsatisfiable by the system (column "U"),
- the number of instances for which the system failed, i.e., for which it has been stopped (column "F"),
- the geometric average of the CPU times (column "CPU"), but considering only the formulas on which the system did not fail. When the system fails on all the samples, we write "–".
- the geometric average of the number of consistency checks (column "Cons"), i.e., of recursive calls to KSAT in within KCONSIST_C. As for the CPU times,

						kSAT			kSAT+USB			kSAT+USH		
C	V	D	Sz	d	Ms	S/U/F	Cpu	Cons	S/U/F	Cpu	Cons	S/U/F	Cpu	Cons
10	4	4	1171	22	230	0/0/8	–	–	8/0/0	0.36	750	8/0/0	0.36	1007
20	4	4	2041	22	369	0/0/8	–	–	4/4/0	1.64	2231	4/4/0	1.92	3558
30	4	4	2887	22	515	0/0/8	–	–	3/5/0	2.83	2871	3/5/0	3.93	5320
40	4	4	3710	22	647	0/0/8	–	–	1/7/0	4.43	3461	1/7/0	7.28	8000
50	4	4	4516	22	787	0/0/8	–	–	0/8/0	4.41	3163	0/8/0	5.91	5349
10	8	4	2236	42	440	0/0/8	–	–	8/0/0	0.77	1302	8/0/0	0.78	1598
20	8	4	3775	42	761	0/0/8	–	–	8/0/0	12.77	10773	8/0/0	16.11	17303
30	8	4	5200	42	1042	0/0/8	–	–	8/0/0	92.74	49964	8/0/0	152.93	111660
40	8	4	6601	42	1348	0/0/8	–	–	2/5/1	517.84	203594	1/4/3	623.78	332768
50	8	4	7998	42	1620	0/0/8	–	–	0/3/5	197.68	71422	0/3/5	331.41	154176
10	16	4	4242	82	845	0/0/8	–	–	8/0/0	1.29	1840	8/0/0	1.21	2210
20	16	4	7762	82	1478	0/0/8	–	–	8/0/0	75.98	45052	8/0/0	79.06	62894
30	16	4	10470	82	2136	0/0/8	–	–	6/0/2	703.53	244983	4/0/4	691.19	331341
40	16	4	13334	82	2728	0/0/8	–	–	0/0/8	–	–	0/0/8	–	–
50	16	4	15851	82	3315	0/0/8	–	–	0/0/8	–	–	0/0/8	–	–
10	4	6	1598	30	314	0/0/8	–	–	8/0/0	0.50	973	8/0/0	0.53	1278
20	4	6	2775	30	529	0/0/8	–	–	8/0/0	4.86	5279	8/0/0	6.61	9549
30	4	6	3839	30	726	0/0/8	–	–	5/3/0	22.85	16895	5/3/0	32.21	33517
40	4	6	4856	30	921	0/0/8	–	–	1/7/0	33.29	18658	1/7/0	55.54	43488
50	4	6	5912	30	1132	0/0/8	–	–	0/8/0	46.07	21509	0/8/0	77.32	49231
10	8	6	3044	58	598	0/0/8	–	–	8/0/0	0.78	1258	8/0/0	0.75	1473
20	8	6	5357	58	1061	0/0/8	–	–	8/0/0	22.59	16482	8/0/0	26.34	26749
30	8	6	7322	58	1523	0/0/8	–	–	5/0/3	681.74	296324	3/0/5	579.76	318183
40	8	6	9295	58	1914	0/0/8	–	–	0/0/8	–	–	0/0/8	–	–
50	8	6	11136	58	2318	0/0/8	–	–	0/0/8	–	–	0/0/8	–	–
10	16	6	6113	114	1141	0/0/8	–	–	8/0/0	1.75	2064	8/0/0	1.62	2453
20	16	6	10657	114	2054	0/0/8	–	–	8/0/0	54.70	25887	8/0/0	55.37	38118
30	16	6	15188	114	2966	0/0/8	–	–	5/0/3	502.92	152995	4/0/4	508.82	204417
40	16	6	19047	114	3925	0/0/8	–	–	0/0/8	–	–	0/0/8	–	–
50	16	6	23091	114	4706	0/0/8	–	–	0/0/8	–	–	0/0/8	–	–

Table 4
kSAT, kSAT+USB, and kSAT+USH on TANCS 2000 modalized easy/medium problems

we consider only the formulas on which the system did not fail, and write "–" when the system fails on all the samples.

As it can be seen, caching produces dramatic improvements. Many tests that are not solved within the timelimit by kSAT, are solved when using a caching mechanism.

Indeed, the sets in Tables 1 and 2 are just a few. Consider also Tables 3 and 4, showing kSAT, kSAT+USB and kSAT+USH results on the easy/medium and modalized easy/medium formulas respectively. These Tables make clear the qualitative differences between kSATon one side, and kSAT+USB and kSAT+USH on the other. kSAT solves only 51 (9%) out of the 544 samples on which it has been tested. kSAT+USB and kSAT+USH solve 414 (76%) and 390 (71%)

formulas respectively. kSAT fails on many tests which are efficiently solved both by kSAT+USB and kSAT+USH. For example, in the easy/medium problems, when C=10, V=8 and D=4, kSAT does not solve any of the eight samples; On the other hand kSAT+USB and kSAT+USH solve them in a time well below 1s. Even considering the sets on which kSAT does not timeout, its performances are sometime much worse than those of the other two systems. The data about the CPU times reflect the data about the number of consistency checks: kSAT always perform many more of them. The fact that kSAT+USH performs less consistency checks than kSAT, points out that this last system indeed repeats the same computation many times.

Finally, it is interesting to notice that kSAT, kSAT+USB and kSAT+USH perform better on the non-modalized tests than on the corresponding modalized tests. This seems to confirm the conjecture that the modalized tests are harder.

6.2. Bit Matrices vs Hash Tables

To better evaluate the differences between kSAT+USB and kSAT+USH, here we consider some other data from the previously described tests. In particular, we now consider the following parameters:

- The number of times the solver exceeded the timeout (column "T"), or the available RAM (column "M");
- The average cache hit ratio (column "Hit"), i.e., the average of the ratio:

$$\frac{\text{Number of cache hits}}{\text{Number of cache lookups}}$$

  where a "lookup" happens any time the cache is searched for a previously computed result, and a "hit" means that the search was successful.
- The average amount of memory in Kbytes allocated by the system (column "Mem"). For kSAT+USH, this value is (roughly) proportional to the number of stored results. For kSAT+USB, this value is proportional to the depth and the size of the sample.

Tables 5, 6, 7 and 8 report the above data for the medium-hard, modalized medium-hard, easy-medium and modalized easy-medium sets. In the Tables, all the parameters that are not listed above, have the same meaning of the previous Subsection.

Consider Tables 5, 6, 7 and 8. The first observation is that kSAT+USB is able to solve more samples than kSAT+USH: kSAT+USB fails on 130 samples, while kSAT+USH fails on 154. Of the 154 samples on which kSAT+USH fails, 29 are because of the memory limit of 256MB. On the other hand, kSAT+USB space occupancy is always below 32MB. Taking into account that kSAT+USB CPU times are (when bigger than 2s, and on the average) always lower than kSAT+USH,

			kSAT+USB						kSAT+USH					
C	V	D	T/M	Cpu	Cons	Hit	Stored	Mem	T/M	Cpu	Cons	Hit	Stored	Mem
10	4	4	0/0	102.05	26663	0.51	24060	5018	0/0	133.82	41898	0.52	35369	11561
20	4	4	0/0	321.15	75967	0.40	69556	5442	0/0	493.66	131996	0.42	111632	36608
30	4	4	0/0	18.08	3953	0.40	3717	5664	0/0	29.36	8253	0.39	6854	6371
10	8	4	4/0	–	–	–	–		4/0	–	–	–	–	–
20	8	4	4/0	–	–	–	–		4/0	–	–	–	–	–
30	8	4	4/0	–	–	–	–		4/0	–	–	–	–	–
10	4	6	3/0	981.05	154892	0.44	142816	9640	4/0	–	–	–	–	–
20	4	6	4/0	–	–	–	–		4/0	–	–	–	–	–

Table 5
kSAT+USB and kSAT+USH on TANCS 2000 medium/hard problems

we consider this to be a very positive experimental result for kSAT+USB.[7] Notice that we have better results for kSAT+USB even on very large formulas. For example, in the modalized medium/hard problems, for C=10, V=8 and D=4 we have formulas with more than 7000 modal subformulas, whose size is more than 450000 symbols, and with depth 159. To give an idea, storing one of these formulas takes roughly 1MB of disk space. Indeed, on very deep formulas, the subset-matching mechanism implemented by bit matrices can cause a hit which saves a lot of computation.

A closer look at the other parameters of the two systems, points out that the following facts:

- kSAT+USB always performs less consistency checks than kSAT+USH.
- The two systems have roughly the same hit-ratio on all samples (the biggest difference between the two hit-ratio, is 0.08 for C=10, V=16 and D=4 in the easy-medium problems).

The first item is not surprising, and agrees with the above discussed results on the CPU times. The second item points out that even if kSAT+USB performs less consistency checks (and thus on some data kSAT+USB has a hit and kSAT+USH a fail when accessing the cache), the ratio between the hits and number of accesses for the two systems remains the same.

6.3. Big W vs small W

To evaluate whether it is true that only few results are useful and, among these, the latest are the most important, we run kSAT+USB on previous benchmarks, but with different window sizes. Besides the default window size

[7] At first glance, it may seem that kSAT+USH performs better than kSAT+USB on some set. This is the case of C=40, V=8 and D=6 in the easy/medium tests. However, a closer look reveals that in this set, kSAT+USH solves only one of the 8 samples, while kSAT+USB does 3. Indeed, on the sample solved by kSAT+USH, kSAT+USB takes 97.20s.

			kSAT+USB						kSAT+USH					
C	V	D	T/M	Cpu	Cons	Hit	Stored	Mem	T/M	Cpu	Cons	Hit	Stored	Mem
10	4	4	0/0	30.20	20398	0.64	13333	5892	0/0	101.46	98330	0.68	8030	5643
20	4	4	0/0	28.99	12796	0.64	8168	6601	0/0	176.98	134319	0.69	7155	5425
30	4	4	0/0	48.49	28928	0.65	18477	7018	1/0	338.78	197789	0.67	11621	12825
10	8	4	1/0	704.37	128334	0.58	75810	22101	4/0	–	–	–	–	–
20	8	4	4/0	–	–	–	–		4/0	–	–	–	–	–
30	8	4	3/0	727.93	96175	0.63	62782	31641	4/0	–	–	–	–	–
10	4	6	1/0	64.98	17100	0.65	9849	11709	0/0	155.37	140415	0.72	10524	7057
20	4	6	3/0	295.58	213999	0.74	164461	13482	3/0	875.62	664688	0.74	41566	27327

Table 6

kSAT+USB and kSAT+USH on TANCS 2000 modalized medium/hard problems

			kSAT+USB						kSAT+USH					
C	V	D	T/M	Cpu	Cons	Hit	Stored	Mem	T/M	Cpu	Cons	Hit	Stored	Mem
10	4	4	0/0	0.08	246	0.59	208	1276	0/0	0.07	302	0.55	137	1164
20	4	4	0/0	0.48	945	0.41	780	1545	0/0	0.54	1494	0.39	707	1438
30	4	4	0/0	0.66	1036	0.33	822	1676	0/0	0.80	1831	0.34	925	1539
40	4	4	0/0	0.96	1268	0.33	983	1788	0/0	1.30	2553	0.31	1327	1681
50	4	4	0/0	0.66	726	0.34	574	2030	0/0	0.80	1374	0.33	706	1842
10	8	4	0/0	0.25	643	0.63	563	2029	0/0	0.22	783	0.60	286	1811
20	8	4	0/0	3.28	4037	0.49	3551	2964	0/0	4.98	8991	0.47	2871	3131
30	8	4	0/0	42.72	30507	0.41	25986	3902	0/0	91.49	98399	0.42	29913	9429
40	8	4	0/0	58.20	32698	0.42	28238	4348	0/0	162.45	134390	0.41	41067	14316
50	8	4	0/0	75.71	37776	0.39	31326	5274	0/0	206.99	145379	0.41	45634	19295
10	16	4	0/0	0.46	906	0.85	853	3826	0/0	0.48	1203	0.77	356	3434
20	16	4	0/0	6.57	6684	0.61	6045	5325	0/0	9.97	14769	0.55	5175	6644
30	16	4	0/0	83.77	43580	0.48	38224	6954	0/1	177.23	138971	0.46	38389	24971
40	16	4	6/0	604.25	234336	0.38	201991	9428	1/7	–	–	–	–	–
50	16	4	8/0	–	–	–	–		1/7	–	–	–	–	–
10	4	6	0/0	0.16	410	0.57	354	1628	0/0	0.19	686	0.56	253	1484
20	4	6	0/0	1.78	2437	0.43	2098	2056	0/0	2.85	6009	0.42	2070	2046
30	4	6	0/0	7.46	7435	0.40	6240	2431	0/0	14.79	22988	0.40	8458	3283
40	4	6	0/0	5.83	5396	0.35	4446	2910	0/0	9.45	13295	0.35	5656	3614
50	4	6	0/0	4.66	3755	0.33	2992	3001	0/0	11.86	13772	0.35	5936	4006
10	8	6	0/0	0.29	679	0.81	624	2681	0/0	0.27	839	0.77	251	2393
20	8	6	0/0	4.81	4808	0.60	4350	4014	0/0	8.10	11784	0.54	3705	4482
30	8	6	0/0	90.65	52982	0.46	45885	4963	0/0	225.74	200873	0.45	49537	19711
40	8	6	3/0	396.80	163496	0.42	141021	6667	7/0	369.96	273971	0.37	95712	43223
50	8	6	8/0	–	–	–	–		8/0	–	–	–	–	–
10	16	6	0/0	0.73	1316	0.79	1255	5038	0/0	0.67	1648	0.73	445	4534
20	16	6	0/0	8.93	6701	0.70	6257	6893	0/0	14.13	16686	0.62	5250	9224
30	16	6	0/0	176.68	72254	0.49	65628	9345	0/2	304.58	193313	0.46	56111	44964
40	16	6	4/0	561.58	168941	0.43	152276	12197	0/8	–	–	–	–	–
50	16	6	8/0	–	–	–	–		4/4	–	–	–	–	–

Table 7

kSAT+USB and kSAT+USH on TANCS 2000 easy/medium problems

			kSAT+USB					kSAT+USH				
C	V	D	T/M	Cpu	Cons	Hit	Mem	T/M	Cpu	Cons	Hit	Mem
10	4	4	0/0	0.36	750	0.71	1644	0/0	0.36	1007	0.68	1518
20	4	4	0/0	1.64	2231	0.48	1936	0/0	1.92	3558	0.48	1826
30	4	4	0/0	2.83	2871	0.47	2091	0/0	3.93	5320	0.47	2015
40	4	4	0/0	4.43	3461	0.47	2194	0/0	7.28	8000	0.48	2213
50	4	4	0/0	4.41	3163	0.46	2823	0/0	5.91	5349	0.48	2711
10	8	4	0/0	0.77	1302	0.74	2496	0/0	0.78	1598	0.75	2271
20	8	4	0/0	12.77	10773	0.56	3639	0/0	16.11	17303	0.55	3936
30	8	4	0/0	92.74	49964	0.46	4558	0/0	152.93	111660	0.48	9153
40	8	4	1/0	517.84	203594	0.46	5030	3/0	623.78	332768	0.49	19140
50	8	4	5/0	197.68	71422	0.45	6342	5/0	331.41	154176	0.50	14607
10	16	4	0/0	1.29	1840	0.81	4391	0/0	1.21	2210	0.78	3982
20	16	4	0/0	75.98	45052	0.57	5893	0/0	79.06	62894	0.58	10442
30	16	4	2/0	703.53	244983	0.47	8133	4/0	691.19	331341	0.50	33819
40	16	4	8/0	–	–	–		8/0	–	–	–	–
50	16	4	8/0	–	–	–		8/0	–	–	–	–
10	4	6	0/0	0.50	973	0.66	1888	0/0	0.53	1278	0.64	1756
20	4	6	0/0	4.86	5279	0.48	2387	0/0	6.61	9549	0.49	2390
30	4	6	0/0	22.85	16895	0.46	2814	0/0	32.21	33517	0.48	3908
40	4	6	0/0	33.29	18658	0.45	3752	0/0	55.54	43488	0.47	5647
50	4	6	0/0	46.07	21509	0.45	3885	0/0	77.32	49231	0.47	6415
10	8	6	0/0	0.78	1258	0.80	3047	0/0	0.75	1473	0.78	2730
20	8	6	0/0	22.59	16482	0.55	4707	0/0	26.34	26749	0.58	5613
30	8	6	3/0	681.74	296324	0.45	6754	5/0	579.76	318183	0.49	26041
40	8	6	8/0	–	–	–		8/0	–	–	–	–
50	8	6	8/0	–	–	–		8/0	–	–	–	–
10	16	6	0/0	1.75	2064	0.82	5590	0/0	1.62	2453	0.80	5043
20	16	6	0/0	54.70	25887	0.65	7791	0/0	55.37	38118	0.65	12036
30	16	6	3/0	502.92	152995	0.55	11211	4/0	508.82	204417	0.60	36155
40	16	6	8/0	–	–	–		6/2	–	–	–	–
50	16	6	8/0	–	–	–		8/0	–	–	–	–

Table 8
kSAT+USB and kSAT+USH on TANCS 2000 modalized easy/medium problems

			W=32			W=256			W=512			W=1024		
C	V	D	F	Cpu	Cons	F	Cpu	Cons	F	Cpu	Cons	F	Cpu	Cons
10	4	4	0	89.85	27860	0	93.96	26663	0	102.05	26663	0	120.88	26663
20	4	4	0	352.54	98572	0	306.45	78459	0	321.15	75967	0	372.04	75081
30	4	4	0	18.25	4554	0	17.26	3970	0	18.08	3953	0	19.94	3931
10	8	4	4	–	–	4	–	–	4	–	–	4	–	–
20	8	4	4	–	–	4	–	–	4	–	–	4	–	–
30	8	4	4	–	–	4	–	–	4	–	–	4	–	–
10	4	6	3	882.31	159339	3	924.92	155948	3	981.05	154892	3	1128.69	154892
20	4	6	4	–	–	4	–	–	4	–	–	4	–	–

Table 9
kSAT+USB on TANCS 2000 medium/hard problems

			W=32			W=256			W=512			W=1024		
C	V	D	F	Cpu	Cons	F	Cpu	Cons	F	Cpu	Cons	F	Cpu	Cons
10	4	4	2	98.06	95035	0	32.63	24461	0	30.20	20398	0	36.00	18982
20	4	4	3	237.01	128555	0	28.76	13836	0	28.99	12796	0	33.28	12321
30	4	4	2	389.78	304689	0	57.03	40008	0	48.49	28928	0	55.70	25995
10	8	4	4	–	–	2	804.62	145312	1	704.37	128334	1	620.89	89416
20	8	4	4	–	–	4	–	–	4	–	–	4	–	–
30	8	4	4	–	–	4	–	–	3	727.93	96175	3	730.64	92482
10	4	6	4	–	–	1	60.16	17475	1	64.98	17100	1	73.64	16616
20	4	6	4	–	–	3	308.19	237896	3	295.58	213999	1	648.46	287018

Table 10
kSAT+USB on TANCS 2000 modalized medium/hard problems

			W=32			W=256			W=512			W=1024		
C	V	D	F	Cpu	Cons	F	Cpu	Cons	F	Cpu	Cons	F	Cpu	Cons
10	4	4	0	0.06	246	0	0.07	246	0	0.08	246	0	0.09	246
20	4	4	0	0.42	998	0	0.45	945	0	0.48	945	0	0.55	945
30	4	4	0	0.65	1132	0	0.60	1036	0	0.66	1036	0	0.72	1036
40	4	4	0	0.94	1364	0	0.92	1268	0	0.96	1268	0	1.06	1268
50	4	4	0	0.62	773	0	0.61	726	0	0.66	726	0	0.70	726
10	8	4	0	0.25	643	0	0.23	643	0	0.25	643	0	0.29	643
20	8	4	0	4.41	6533	0	3.03	4037	0	3.28	4037	0	3.55	4037
30	8	4	0	110.36	106738	0	44.07	34140	0	42.72	30507	0	47.16	29163
40	8	4	0	139.32	109670	0	61.45	37476	0	58.20	32698	0	62.23	30988
50	8	4	1	157.84	111311	0	82.04	44521	0	75.71	37776	0	79.18	35402
10	16	4	0	0.41	906	0	0.44	906	0	0.46	906	0	0.50	906
20	16	4	0	7.48	8467	0	6.57	6684	0	6.57	6684	0	6.85	6684
30	16	4	2	182.72	117424	0	93.64	50259	0	83.77	43580	0	86.59	41951
40	16	4	8	–	–	7	486.70	226642	6	604.25	234336	5	702.74	217854
50	16	4	8	–	–	8	–	–	8	–	–	8	–	–
10	4	6	0	0.12	410	0	0.14	410	0	0.16	410	0	0.17	410
20	4	6	0	1.82	2852	0	1.67	2437	0	1.78	2437	0	1.97	2437
30	4	6	0	11.65	14420	0	7.43	7858	0	7.46	7435	0	8.37	7396
40	4	6	0	7.09	7633	0	5.62	5463	0	5.83	5396	0	6.51	5390
50	4	6	0	5.39	4905	0	4.44	3780	0	4.66	3755	0	5.18	3754
10	8	6	0	0.27	679	0	0.28	679	0	0.29	679	0	0.30	679
20	8	6	0	6.60	7168	0	4.71	4808	0	4.81	4808	0	5.05	4808
30	8	6	1	273.40	222914	0	106.14	68400	0	90.65	52982	0	94.33	46901
40	8	6	7	230.88	143811	3	498.64	219931	3	396.80	163496	2	438.99	155608
50	8	6	8	–	–	8	–	–	8	–	–	8	–	–
10	16	6	0	0.67	1316	0	0.79	1316	0	0.73	1316	0	0.78	1316
20	16	6	0	9.57	7767	0	8.97	6701	0	8.93	6701	0	9.26	6701
30	16	6	3	229.75	111451	1	145.28	61544	0	176.68	72254	0	184.35	70267
40	16	6	8	–	–	5	543.57	170179	4	561.58	168941	4	539.78	151134
50	16	6	8	–	–	8	–	–	8	–	–	8	–	–

Table 11
kSAT+USB on TANCS 2000 easy/medium problems

			W=32			W=256			W=512			W=1024		
C	V	D	F	Cpu	Cons	F	Cpu	Cons	F	Cpu	Cons	F	Cpu	Cons
10	4	4	0	0.29	757	0	0.43	750	0	0.36	750	0	0.39	750
20	4	4	0	1.56	2514	0	1.97	2231	0	1.64	2231	0	1.81	2231
30	4	4	0	2.89	3420	0	3.58	2872	0	2.83	2871	0	3.16	2871
40	4	4	0	4.87	4473	0	5.52	3478	0	4.43	3461	0	4.86	3461
50	4	4	0	4.98	4087	0	5.41	3173	0	4.41	3163	0	4.74	3163
10	8	4	0	0.72	1325	0	0.90	1302	0	0.77	1302	0	0.83	1302
20	8	4	0	17.36	17047	0	16.85	10888	0	12.77	10773	0	13.76	10745
30	8	4	1	183.97	127385	0	126.42	56186	0	92.74	49964	0	97.77	46512
40	8	4	6	646.82	341448	4	600.88	203170	1	517.84	203594	1	468.93	159376
50	8	4	6	342.99	150130	5	306.75	87421	5	197.68	71422	4	303.16	99996
10	16	4	0	1.15	1841	0	1.57	1840	0	1.29	1840	0	1.33	1840
20	16	4	0	134.19	94407	0	99.08	48311	0	75.98	45052	0	77.40	42750
30	16	4	8	–	–	4	744.42	254053	2	703.53	244983	2	649.31	208462
40	16	4	8	–	–	8	–	–	8	–	–	8	–	–
50	16	4	8	–	–	8	–	–	8	–	–	8	–	–
10	4	6	0	0.45	988	0	0.63	973	0	0.50	973	0	0.56	973
20	4	6	0	5.80	7603	0	5.92	5289	0	4.86	5279	0	5.40	5279
30	4	6	0	34.81	32295	0	29.46	18303	0	22.85	16895	0	25.44	16349
40	4	6	0	54.57	36145	0	43.99	20258	0	33.29	18658	0	35.51	17975
50	4	6	0	70.62	38110	0	61.11	22867	0	46.07	21509	0	49.08	20763
10	8	6	0	0.73	1262	0	0.99	1258	0	0.78	1258	0	0.84	1258
20	8	6	0	33.18	29027	0	28.33	16681	0	22.59	16482	0	24.19	16445
30	8	6	8	–	–	5	924.42	335789	3	681.74	296324	3	620.52	232005
40	8	6	8	–	–	8	–	–	8	–	–	8	–	–
50	8	6	8	–	–	8	–	–	8	–	–	8	–	–
10	16	6	0	1.71	2064	0	2.21	2064	0	1.75	2064	0	1.85	2064
20	16	6	0	80.45	44305	0	53.65	26656	0	54.70	25887	0	54.82	25613
30	16	6	7	834.32	295436	3	546.50	175342	3	502.92	152995	3	492.00	143179
40	16	6	8	–	–	8	–	–	8	–	–	8	–	–
50	16	6	8	–	–	8	–	–	8	–	–	8	–	–

Table 12
kSAT+USB on TANCS 2000 modalized easy/medium problems

(W=512), we look at W=1024 and W=256 sizes to better understand the trade-off between the access time and the persistency of cached data. We also try W=32, i.e., the smallest window size that we can ask for in kSAT+USB. Tables 9 through 12 show the results. For each window size, we consider the number of failed samples ("F"), the Cpu time ("Cpu") and the number of Consistency checks ("Cons").

By comparing the results for different window sizes we can easily see that even with the smallest possible window size, the performance of kSAT on the benchmark formulas is greatly improved. Looking at Table 9 (resp. Table 10) we see that kSAT+USB with W=32 can solve 40% (resp. 21%) of the samples in the medium/hard category (resp. modalized medium/hard) within the time limit. The same percentages drop to 3% (resp. 0%) when we consider kSAT results in

Table 1 (resp. Table 2). Notice that:

- on the medium/hard formulas the percentage of samples solved within the time limit does not increase when increasing the window size, while
- on the modalized medium/hard formulas, we see some noticeable improvements only for substantial increases in the window size (e.g. C=10, V=8, D=4 and C=10,20 V=4, D=6 with W=512,1024).

Considering the medium/easy benchmarks, we get more or less the same overall picture. By looking at Table 11 (resp. Table 12) we see that kSAT+USB with W=32 is able to solve 78% (resp. 65%) of the samples in the medium/easy category (resp. modalized medium/easy) within the time limit. In Table 1 (resp. Table 2 we can observe that kSAT is lagging behind kSAT+USB on these benchmarks too: 20% (resp. 0%) of samples is solved within the time limit. As opposed to the medium/hard case, increasing the window size, always increases the number of samples that kSAT+USB is able to solve. Summing up, kSAT+USB with W=32:

- always ameliorates performance when compared to kSAT, and
- in some cases, leads to better computation times than kSAT+USB featuring bigger window sizes; this is particularly true when the number of performed consistency checks does not decrease with the increase of the window size.

While the good performance of kSAT+USB with W=32 w.r.t kSAT+USB with W bigger than 32 could be attributed to the efficiency of the implementation,[8] the improvement over kSAT is a clear evidence that, at least on these benchmarks, the ability to store even a small number of recently computed results can dramatically improve the overall performances.

In Tables 9 through 12, when looking at window sizes bigger than 32, we immediately notice that the average number of consistency checks is a monotonely decreasing function of the window size. This is true of all the sets (50 out of 76 total) on which kSAT+USB with W bigger than 32 failed to complete the reasoning before the time limit on 25% of the samples at most. By focusing on these sets, we see that the CPU time is never a monotone decreasing function of the window size. Clearly, bigger window sizes augment the overhead and the caching may not pay off any more. For instance, it turns out that kSAT+USB with W=1204 never runs faster than kSAT+USB with W=512 on the selected sets, and it can solve only 5 samples more. The difference between kSAT+USB with W=256 and W=512 is more encouraging: there are only 7 selected sets for which having a bigger window significantly degrades the performance, and kSAT+USB with W=512 solves 11 samples more than kSAT+USB with W=256.

[8] Since most of the current machines feature 32 bit integers (at least) the setting W=32 enables us to implement bit matrices using integers and low-level instructions to handle them as set of bits.

Finally, comparing kSAT+USB with W=256 and kSAT+USB with W=32 we see that the former is nearly always faster and it is able to solve 31 samples more than the latter. Judging from these number, we decided that W=512 offers a good compromise between the reduced number of consistency checks and the increased overhead.

7. Conclusions

We have presented bit matrices. They are a simple caching structure that allows for subset and superset-matching. They can be bounded in size, and different policies for forgetting results can be easily implemented. Bit matrices have been implemented in *SAT and are available for experimentation. Following what has been done in other state of the art systems, we have also implemented a caching mechanism based on hash tables. Hash tables are not bounded in size. Experimenting with the TANCS 2000 benchmarks for modal K, we have showed that *SAT performances are improved by

1. caching intermediate consistency checks,
2. using bit matrices instead of hash tables, and
3. when using bit matrices, storing only a small number of results. storing only a small number of results in the bit matrices.

There is a huge literature focusing on data-structures for efficiently manipulating sets. Most of the these data-structures are variations of the familiar trees from standard textbooks, and it is not easy to use them for subset and superset-matching. Recently, [23] presents a data structure, called UBTree, that allows for subset and superset-matching, and studies its application in IPP, a state-of-the-art planning system. Besides the different application domain, the main difference between UBTree and bit matrices is that for the first it is not clear how to implement effective strategies for storing new elements when the cache is bounded.

For the future, we point out that we have explored only some of the potentials of bit matrices. For example, having a different window size at each depth might lead to better performances. Using a different strategy when forgetting subtests (e.g., before overwriting, we might look whether some cached subtest is subsumed by some other cached subtest) might improve performances as well. Of course, the trade-off between "smart but expensive" and "simple but efficient" remains for all the strategies considered.

Finally, a last remark on the experimental results: the tests that we have performed and of which we have showed the results are dependent

- on the particular algorithms implemented by the procedures, and
- on the particular formulas being tested.

Indeed, as observed, e.g., in [24], even when checking the satisfiability of a propositional formula, if we consider small variations

- in the search procedure, or
- in the formula being tested,

we may get dramatic changes in the computational behavior of the system. We expect this to be true also in our setting, since we are dealing with an extension of propositional logic. Nevertheless, bit matrices have proved to be an effective data structure for caching at least in *SAT and on the tests we consider. More results, the samples that we have used, and *SAT are available on line at:

`http://www.mrg.dist.unige.it/~tac/StarSAT.html`.

Acknowledgements

The authors are partially supported by the MURST project "Model checking e satisfiability: sviluppo di nuove procedure di decisione e loro valutazione ed analisi sperimentale comparata in ambito applicativo" ("Model checking and satisfiability: development of novel decision procedures and comparative validation and experimental analysis in significant application areas").

References

[1] B. Hollunder, W. Nutt, and M. Schmidt-Schauß. Subsumption Algorithms for Concept Description Languages. In *Proc. 8th European Conference on Artificial Intelligence*, pages 348–353, 1990.

[2] F. Baader, E. Franconi, B. Hollunder, B. Nebel, and H.J. Profitlich. An Empirical Analysis of Optimization Techniques for Terminological Representation Systems or: Making KRIS get a move on. *Applied Artificial Intelligence. Special Issue on Knowledge Base Management*, 4:109–132, 1994.

[3] F. Giunchiglia and R. Sebastiani. Building decision procedures for modal logics from propositional decision procedures - the case study of modal K. In *Proc. CADE-96*, Lecture Notes in Artificial Intelligence, New Brunswick, NJ, USA, August 1996. Springer Verlag.

[4] E. Giunchiglia, F. Giunchiglia, R. Sebastiani, and A. Tacchella. More evaluation of decision procedures for modal logics. In *Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, 1998.

[5] A. Heuerding, G. Jager, S. Schwendimann, and M. Seyfried. The Logics Workbench LWB: A Snapshot. *Euromath Bulletin*, 2(1):177–186, 1996.

[6] I. Horrocks. Using an expressive description logic: FaCT or fiction? In *Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 636–647, 1998.

[7] Armando Tacchella. *SAT system description. In P. Lambrix, A. Borgida, M. Lenzerini, R. Möller, and P. Patel-Schneider, editors, *Collected Papers from the International Description Logics Workshop (DL'99)*. CEUR, July 1999.

[8] E. Giunchiglia, F. Giunchiglia, and A. Tacchella. SAT-Based Decision Procedures for Classical Modal Logics. *Journal of Automated Reasoning*, 2000. To appear.

[9] Andrei Voronkov. KK: a theorem prover for K. In Harald Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, volume 1632 of *LNAI*, pages 383–387, Berlin, July 7–10, 1999. Springer-Verlag.

[10] P. F. Patel-Schneider. DLP system description. In E. Franconi, G. De Giacomo, R. M. MacGregor, W. Nutt, C. A. Welty, and F. Sebastiani, editors, *Collected Papers from the International Description Logics Workshop (DL'98)*, pages 87–89. CEUR, May 1998.

[11] V. Haarslev and R. Moeller. HAM-ALC. In E. Franconi, G. De Giacomo, R. M. MacGregor, W. Nutt, C. A. Welty, and F. Sebastiani, editors, *Collected Papers from the International Description Logics Workshop (DL'98)*. CEUR, May 1998.

[12] Vijay Boyapati and Rajeev Goré. KtSeqC: System description. In Neil V. Murray, editor, *Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX-99)*, volume 1617 of *LNAI*, pages 29–31, Berlin, June 07–11 1999. Springer.

[13] U. Hustadt, R. A. Schmidt, and C. Weidenbach. MSPASS: Subsumption testing with SPASS. In P. Lambrix, A. Borgida, M. Lenzerini, R. Möller, and P. Patel-Schneider, editors, *Collected Papers from the International Description Logics Workshop (DL'99)*, pages 136–137. CEUR, July 1999.

[14] I. Horrocks and P. F. Patel-Schneider. Advances in propositional modal satisfiability, 1999. Manuscript.

[15] Enrico Giunchiglia and Armando Tacchella. A subset-matching size-bounded cache for satisfiability in modal logics. In *Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX-2000)*, 2000.

[16] B. F. Chellas. *Modal Logic – an Introduction.* Cambridge University Press, 1980.

[17] M. Davis, G. Longemann, and D. Loveland. A machine program for theorem proving. *Journal of the ACM*, 5(7), 1962.

[18] D.A. Plaisted and S. Greenbaum. A Structure-preserving Clause Form Translation. *Journal of Symbolic Computation*, 2:293–304, 1986.

[19] Joseph Y. Halpern and Yoram Moses. A guide to completeness and complexity for modal logics of knowledge and belief. *Artificial Intelligence*, 54:319–379, 1992.

[20] R. Ladner. The computational complexity of provability in systems of modal propositional logic. *SIAM J. Comp.*, 6(3):467–480, 1977.

[21] M. Schmidt-Schauss and G. Smolka. Attributive concept descriptions with complements. *Artificial Intelligence*, 48(1):1–26, February 1991.

[22] Fabio Massacci. Design and results of the Tableaux-99 Non-classical (Modal) Systems comparison. In Neil V. Murray, editor, *Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX-99)*, volume 1617 of *LNAI*, pages 14–18, Berlin, June 07–11 1999. Springer.

[23] Jörg Hoffman and Jana Koehler. A new method to index and query sets. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 462–467, 1999.

[24] Carla P. Gomes, Bart Selman, and Henry Kautz. Boosting combinatorial search through randomization. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98) and of the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI-98)*, pages 431–437, Menlo Park, July 26–30 1998. AAAI Press.