A Subset-matching Size-bounded Cache for Satisfiability in Modal Logics

Enrico Giunchiglia and Armando Tacchella

DIST, Università di Genova, Viale Causa, 13 – 16145 Genova, Italy
{enrico,tac}@dist.unige.it

Abstract. 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 query for subsets and supersets; (*ii*) they can be bounded in size; and (*iii*) if bounded, the latest obtained (in)consistency results can be kept. We have implemented a caching mechanism based on bit matrices in *SAT. Experimenting with the TANCS 2000 benchmarks for modal logic K, we show that bit matrices (*i*) allow for considerable speedups, and (*ii*) lead to better performances both in space and time than the natural alternative, i.e., hash tables.

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]. A competition is run in conjunction with the Tableaux conferences, aiming at stimulating the development of effective systems. At the last competition, the last four of the above cited systems participated, and —as the organizer Fabio Massacci of the competition writes [14]

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

As reported in [15], one of the reasons for DLP effectiveness is its ability to cache —for later reuse— the result 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. A caching mechanism based on bit matrices has been implemented in *SAT. Bit matrices have three distinguishing features:

1. they can query for subsets and supersets;
2. they can be bounded in size; and
3. if bounded, they the latest obtained (in)consistency results can 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 —given that *SAT search algorithm is based on chronological backtracking— we expect the oldest results to the least useful in the future.

Experimenting with the TANCS 2000 benchmarks for modal logic K, we show that

- caching yields considerable speedups, and
- bit matrices allow for less consistency checks and better timings than hash tables.

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 implement caching. In Section 4, we present bit matrices and how they meet the issues raised in the preceding Section. Section 5 describes how caching mechanisms based on bit matrices and hash tables have been implemented in *SAT. Then, the results of the experimental analysis are reported. We end the paper in Section 6 with some conclusions.

2 Basics

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.

```
function KSAT(φ)
  return KSAT_DPL(cnf(φ), ⊤).

function KSAT_DPL(φ, μ)
  if φ = {} then return KCONSIST(μ);                    /* base */
  elsif {} ∈ φ then return False;                       /* backtrack */
  elsif { a unit clause {l} belongs to φ } then         /* unit propagation */
    return KSAT_DPL(assign(l, φ),μ ∧ l);
  elsif not KCONSIST(μ) then return False;              /* early pruning */
  else
    l := choose-literal(φ, μ);                          /* split */
    return KSAT_DPL(assign(l, φ),μ ∧ l) or
           KSAT_DPL(assign(l̄, φ),μ ∧ l̄).
```

Fig. 1. KSAT and KSAT_{DPL}

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,

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

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:

- *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:

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*.

Fig. 2. KCONSIST

- *cnf*(φ) 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, $\overline{l}$ stands for A if $l = \neg A$, and for $\neg A$ if $l = A$;
- for any literal l and formula φ, *assign*(l, φ) is obtained from φ by
  - deleting the clauses in which l occurs as a disjunct, and
  - eliminating $\overline{l}$ from the others.

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 store 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 enabling 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 suggests caching mechanisms which can store sets of formulas and can efficiently query about subsets or 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 S-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 U-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 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 S-cache should be able to store satisfiable subtests sharing a common set Δ in a compact way. Therefore, S-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 U-cache should not care about subtests sharing a common Δ. Therefore, U-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 KCONSIST to yield the procedure $\textsc{Kconsist}_{\textsc{C}}$ shown in Figure 3. In the Figure:

- U-cache_get(Δ,Γ) returns *True* if U-cache contains a set Δ' such that $\Delta\supseteq\Delta'$, Δ' is associated with β and $\beta\in\Gamma$;

```
function KConsist_C(μ)
  Δ := {α | □α is a conjunct of μ};
  Γ := {β | ¬□β is a conjunct of μ};
  if U-cache_get(Δ, Γ) return False;
  Γ_r := S-cache_get(Δ, Γ);
  Γ_s := ∅;
  foreach β ∈ Γ_r do
    if not KSat(⋀_{α∈Δ} α ∧ ¬β) then
      if Γ_s ≠ ∅ then S-cache_store(Δ, Γ_s);
      U-cache_store(Δ, β);
      return False
    else Γ_s := Γ_s ∪ {β};
  if Γ_s ≠ ∅ then S-cache_store(Δ, Γ_s);
  return True.
```

Fig. 3. KConsist$_C$: consistency checking for K with caching

- S-cache_get(Δ, Γ) 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 S-cache.
- U-cache_store(Δ, β) stores in U-cache the set Δ and associates β to it;
- S-cache_store(Δ, Γ) stores in S-cache the set Δ and associates to it the set Γ.

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 logic being considered, and
- on the characterstics of the particular formula being tested,

we expect that one caching mechanism will lead to a faster decision process than the others.

Indipendently 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.

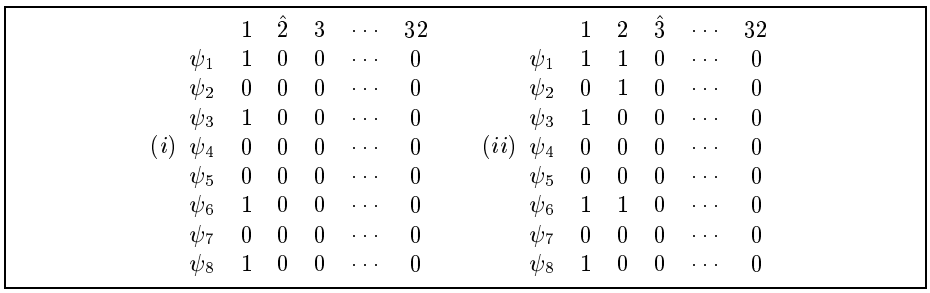

Fig. 4. Bit matrix B with $N = 8$, (i) before and (ii) after storing the set $\psi_1 \wedge \psi_2 \wedge \psi_6$ ("$\hat{}$" denotes the current column).

4 Bit matrices for modal satisfiability

4.1 Bit matrices

A *vector* is a sequence of elements of the same kind. An 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 A, we write A$[i][k]$ (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* $\hat{k}$ 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 a $N \times W$ 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 dynamically allocated space). Then we apply the procedure above;

– or overwrite previously cached data with some policies, generally depending on the contents of B. In other words, we simply fix $\hat{k}$ to be $f(B)$, 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 and B as the bit vector C = (A ∩ B) such that C$[i]$ is set if and only if both A$[i]$ and B$[i]$ are set. 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., $C = \bigcap_{\psi_i \in \Delta}$ 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 implement 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 = (B[1] ∩ 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 $(\Sigma \setminus \Delta) \subseteq (\Sigma \setminus \Delta')$.

In the following, we assume that the routine KCONSIST$_C$ is used in KSAT to establish consistency of assignments and we describe how to use bit matrices for S-cache and U-cache respectively.

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. Let μ be an assignment satisfying φ. (We assume that each conjunct in μ is a subformula of φ.) Δ, Γ and Γ_r have the same meaning as 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$_C$: 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 $\beta_j \in \Gamma_r$ 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 works as follows:

1. the conjunction C of all the bit vectors B[i] such that $\alpha_i \in \Delta$ is computed, i.e. C= $\bigcap_{\alpha_i \in \Delta}$ B[i];
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, $\Gamma_r := \Gamma_r \setminus \{\beta_j\}$;
4. return Γ_r.

For S-cache_store we have:

1. take the current column $\hat{k}$ and clear each bit B[i][$\hat{k}$] for $i \leq 2N$;
2. set each bit B[i][$\hat{k}$] such that $\alpha_i \in \Delta$:
3. while Δ does not change, i.e., while in the same KCONSIST$_C$ call, set B[$N+j$][$\hat{k}$] whenever $\bigwedge_{\alpha_i \in \Delta} \wedge \neg\beta_j$ is satisfiable.

The time required to cache *all* the satisfiable subtests of a given assignment μ is proportional to the number of modal subformulas in μ.

4.3 Bit matrices for U-cache

Consider the hypotheses written in the first paragraph of the preceding subsection.

In order to take into account observation 4, U-cache can be implemented using a bit matrix B of N rows, and and a vector D of W elements. Every column of B encodes part of an unsatisfiable subtests resulting from a call to KCONSIST$_C$: given a column k and $i \leq N$, B[i][k] is set if and only if $\alpha_i \in \Delta$. The index j such that $\bigwedge_{\alpha_i \in \Delta} \supset \beta_j$, is stored in 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 works as follows:

1. the conjunction C of all the bit vectors B[i] such that $\alpha_i \notin \Delta$ is computed, C= $\bigcap_{\alpha_i \notin \Delta}$ B[i];
2. for each $\beta_j \in \Gamma$ and for each k such that C[k] is set, test if D[k]= j and, if so, return *True*;
3. return *False*;

For U-cache_store we have:

1. take the current column $\hat{k}$ ($k < W$) and for $i \leq N$ set each bit B[i][$\hat{k}$];
2. clear each bitB[i][$\hat{k}$] such that $\alpha_i \in \Delta$;
3. D[k] := j since $\bigwedge_{\alpha_i \in \Delta} \supset \beta_j$.

The time required to cache an unsatisfiable subtest from a given assignment μ is proportional to the number of modal subformulas *not appearing* in μ.

5 Implementation and experimental analysis

5.1 Caching in *SAT

We implemented in our system *SAT 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 in Tables 1 and 2.

The current version of *SAT [1] features a caching otpimization based on the algorithm KCONSIST_C described in section 3. Bit matrices, as well as hash tables, are available to support such optimization. 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 is useful, and
2. 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. A fixed window size 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 owerwrites 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 is 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.

The data structure for S-cache and U-cache in *SAT is a vector of $d(\varphi)$ bit matrices. Each one holds the results of subtests involving formulas occurring at a given level in φ.

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 lenght N whose elements are lists called *buckets*. A suitable function $1 \leq h(\Delta) \leq N$ is used to calculate the hash code associated to each set Δ. Two sets $\Delta \neq \Delta'$ may share the same

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

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 $\bigwedge_{\alpha \in \Delta} \wedge \neg\beta$ is consistent (resp. inconsistent). Notice that, according to assumption 3 above, *SAT features a vector of length d of hash tables.

The selection of caching methods in *SAT includes the following options that we used in our experimental analisys:

kSAT denotes *SAT with all caching optimization turned off;
kSAT+USB denotes *SAT plus caching with bounded bit matrices: the window size is $W = 512$ unless otherwise specified, and both S-cache and U-cache are active;
kSAT+USH denotes *SAT plus caching with unbounded hash tables: as in the case of bit matrices, both S-cache and U-cache are active.

5.2 Experimental analysis

To test the effectiveness of the caching mechanisms that we have described and implemented in *SAT, we consider the TANCS 2000 benchmarks problems, category "MODAL PSPACE Division", sub-category "Problems for Modal QBF", class "Medium/Hard".[2] 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. By default, 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 using a polynomial encoding based on Ladner original translation ([20]), enhanced with some smart tricks so that no extra variable is introduced. Each formula, can be also "modalized": each propositional variable is replaced with a special modal formula with only one propositional variable. See [21] for more details. We will refer to the formulas resulting after Ladner encoding as "unbounded QBFs", and to their modalized version as "unbounded modalized QBFs".

Table 1 shows the results for kSAT, kSAT+USB, and kSAT+USH on a collection of "unbounded QBFs" (top table) and "unbounded modalized QBFs" (bottom table) benchmarks. For each selected value of C,V, and D the four samples found in the TANCS2000 repository were run and we report the geometric mean of the size (column "Sz"), the depth (column "d"), and the number of modal subformulas ("Ms") of the samples. For practical reasons, a time limit of 1200 seconds of CPU time is enforced for each run. For the systems, we report the number of failures ("F"), i.e. the number of samples for which either the time limit was exceeded or the physical memory was exhausted by *SAT.[3] The average CPU time ("Cpu") in seconds, and the average number of consistency

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

[3] All the tests have been run on PCs `PII350MHz` with `256 MbRAM`, running `Linux SUSE 6.2`.

						kSAT			kSAT+USB			kSAT+USH		
C	V	D	Sz	d	Ms	F	Cpu	Cons	F	Cpu	Cons	F	Cpu	Cons
10	4	4	7202	20	1013	4	–	–	0	102.05	26663	0	133.82	41898
20	4	4	8471	20	1259	4	–	–	0	321.15	75967	0	493.66	131996
30	4	4	9289	20	1418	1	87.50	53472	0	18.08	3953	0	29.36	8253
10	8	4	31281	40	2728	4	–	–	4	–	–	4	–	–
20	8	4	247845	80	11275	4	–	–	4	–	–	4	–	–
30	8	4	308422	80	14553	4	–	–	4	–	–	4	–	–
10	4	6	15698	28	1717	4	–	–	3	981.05	154892	4	–	–
20	4	6	19395	28	2284	4	–	–	4	–	–	4	–	–
10	4	4	20452	40	973	4	–	–	0	30.20	20398	0	101.46	98330
20	4	4	27137	41	1311	4	–	–	0	28.99	12796	0	176.98	134319
30	4	4	30140	41	1479	4	–	–	0	48.49	28928	1	338.78	197789
10	8	4	459144	159	7056	4	–	–	1	704.37	128334	4	–	–
20	8	4	95742	79	2761	4	–	–	4	–	–	4	–	–
30	8	4	140703	81	4138	4	–	–	3	727.93	96175	4	–	–
10	4	6	46054	57	1773	4	–	–	1	64.98	17100	0	155.37	140415
20	4	6	59526	57	2312	4	–	–	3	295.58	213999	3	875.62	664688

Table 1. kSAT, kSAT+USB, and kSAT+USH performances

checks ("Cons"), are calculated on the samples that did not fail. When the system exceeds the resources on all the samples, the entry "–" replaces the average CPU time and number of consistency checks.

As it can be seen, caching produces dramatic speedups. Many tests that are not solved within the time limit by kSAT, are solved when using a caching mechanism. For the unbounded QBFs, we have a one order of magnitude speedup. For the modalized version, we have a speedup of two orders of magnitude. Interestingly, we see that kSAT+USB performs better than kSAT+USH. The difference in performances is not dramatic. However, we recall that kSAT+USB uses a cache bounded in size, while kSAT+USH does not: kSAT+USH failed on some tests because it exhausted the physical memory of the computer. Also notice that kSAT performs always many more subtests than kSAT+USH: this means that some equal subtests are indeed repeated by kSAT. Moreover, kSAT+USH performs more subtest than kSAT+USB, even if kSAT+USB uses a size bounded cache. This means that a subset-matching and superset-matching cache is indeed a winning bet at least in these tests which involve pretty big formulas, both in terms of depth and size.

To evaluate whether it is the case that only a small number of results is useful, and the latest obtained results are the most important, we run kSAT+USB on previous benchmarks, but with different window sizes. Table 2 shows the results. As it can be observed, on the first test, having $W = 32$ leads to the best computation times. Indeed, by incrementing the window size, the number of performed subtests decreases. However, the time saved does not compensate the additional costs for querying a bigger cache. However, the most interesting

			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	–	–
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 2. kSAT+USB performances when varying the window size

result is that (by observing these few data), it seems that beyond a certain value for W, the number of performed subtests decreases very slowly with W. This seems to be the case for all the tests which do not exceed the available resources. Of course, this points out that

- beyond a certain value for W, furtherly increasing W will not pay off, and
- our strategy for forgetting results is good, in the sense that we are not throwing away useful data.

6 Conclusions

We have presented bit matrices. They are a simple caching structure that features 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 DLP, 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

- caching allows for considerable speedups, and
- bit matrices allow for less consistency checks and better timings than hash tables.

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, [22] presents a data structure, called UBTree, that allows for subset and superset-matching, and study 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. Bit matrices can be seen as a tree in which each row corresponds to a path. The particular implementation of bit matrices makes queries particular effective.

For the lack of space we reported only a few test results: a much wider set of benchmarks was run and the results on kSAT+USB are reported in [23]. In this work, the performance is evaluated on the full set of TANCS2000 benchmarks, thus including formulas of the "Medium" and "Easy/Medium" categories (as opposed to the "Medium/Hard" category that we analyzed here). The results of this evaluation plus some unpublished preliminary results that we have on kSAT+USH essentially confirm the results that we presented. We also remark that only some of the potentials of bit matrices have been explored. 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. Of course, there is always a trade-off between "smart but expensive" and "simple but efficient" strategies.

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. Fabio Massacci. Results of the TANCS comparison. `http://www.dis.uniroma1.it/~tancs/results.shtml`, 1999. Accessed March 9, 2000.
15. I. Horrocks and P. F. Patel-Schneider. Advances in propositional modal satisfiability, 1999. Manuscript.
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. 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.
22. 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.
23. Armando Tacchella. Evaluating *SAT on TANCS2000 benchmarks. In *Proc. of International Conference on Theorem Proving with Analytic Tableaux and Related Methods (Tableaux 2000)*, July 2000.