Using BDDs to Implement Propositional Modal Tableaux

Kerry Olesen
Australian National University

Abstract. We present a method for using Binary Decision Diagrams (BDDs) to implement the tableau method of automated reasoning for propositional modal logics K and S4. We evaluate an implementation of such a reasoner that uses BDDs as the basic data structure. We show, with comparison to FaCT++ and InKreSAT, that it can compete with other state of the art methods of reasoning in propositional modal logic.

1 Introduction

The systems which people design are becoming increasingly complex. From business processes [9] to digital circuit design [5], modern systems are often incredibly complex, and also safety-critical; it is of the utmost importance that some of these systems always perform exactly as they were intended to, else there could be significant consequences. Thus, verifying the correctness of these systems is important, but is becoming more and more difficult.

The field of automated reasoning seeks to manage this complexity by automating parts of the verification process. Given a logical encoding of a system $\Gamma$ and a desired property $\varphi$, an automated reasoner seeks to determine whether the property $\varphi$ is indeed enforced by $\Gamma$. In other words, it determines whether $\varphi$ is a logical consequence of $\Gamma$, in some given logic $L$.

To be effective, such automated reasoners need to be efficient and fast, but this task is often not trivial, even for powerful modern computers. The design of efficient reasoners is an active research topic [4] [8], to which we contribute.

There are many approaches to automated reasoning, but we will focus on the tableau method [3]. Similarly, there are many formal logics within which automated reasoning is done, but we will only consider $L$ as propositional modal logic. Propositional modal logic can represent a system of propositional states, with the modal aspect capturing the notion of transitions between states. We assume the reader is familiar with both tableaux and propositional modal logic.

Tableaux can implemented in various ways. A naive implementation may simply use hash sets to represent sets of formulae, and an explicit coding for each of the rules, creating new hash sets for every node in the tableau.

We present a method for using Binary Decision Diagrams (BDDs) to implement the tableau method in propositional modal logic. We use BDDs for both representing sets of formulae, and for applying the propositional tableau rules to manipulate those sets. Similar work has been done by Kaminski and Tebbi.
[8], wherein the problem is reduced to a Boolean satisfiability problem and then solved by a SAT solver [2], in an incremental fashion. Our use of BDDs has similarities to their use of a SAT solver, but differs in the following ways.

Kaminski and Tebbi use a labelled tableau calculus, and a single SAT solver for the entire system. To instantiate a tableau rule, they add a clause to their SAT encoding of the tableau. For example, the \((\lor)\) rule is instantiated by adding the clause \(\neg l_{\sigma;\varphi_1} \lor \varphi_1 \lor l_{\sigma;\varphi_2}\). Their construction of the tableau is done by picking a branch, instantiating all pending rules that can be instantiated at that branch, and repeating.

The tableau calculus we use is unlabelled. This means we have separate BDDs for each world, instead of a single SAT solver for the entire system.

The manner in which we construct the tableau is also quite different. We always construct the tableau for an entire saturation phase in one hit, and then explore modal jumps in a depth-first fashion. We have a similar master/slave relationship with BDDs as they do with their SAT solver, incrementally requesting BDD operations and directing the process at a high level, but the granularity of our increments is quite different.

Finally, the internal mechanics of BDDs are fundamentally quite different to those of a SAT solver.

We will begin by providing details of modal logic, tableau and BDDs, and then describe our method and implementation.

We will then provide an evaluation, which shows that our method can be competitive with other state-of-the-art automated reasoners for modal logic.

2 Preliminaries

2.1 Modal Logic Syntax and Semantics

We assume the reader is familiar with modal logic, so here we present the syntax and semantics of modal logic as presented by Goré [3].

A formula \(\varphi\) in modal logic is any expression built out of atomic propositions and connectives as follows, where \(p\) is any element of the set of atomic propositions \(\text{Atm}\):

\[
p ::= p_0 \mid p_1 \mid p_2 \mid \ldots
\]

\[
\varphi ::= p \mid \neg \varphi \mid \Box \varphi \mid \varphi \land \varphi \mid \varphi \lor \varphi
\]

The semantics of modal logic are defined by Kripke models. A Kripke model is a tuple \(\langle W, R, \vartheta \rangle\), where \(W\) is a non-empty set of worlds, \(R \subseteq W \times W\) is a binary reachability relation over the worlds of \(W\), and \(\vartheta : W \times \text{Atm} \rightarrow \{\text{true, false}\}\) is a valuation that provides the truth value of every atomic proposition at every world of \(W\). We write \(wRv\) to denote that \(v\) is an \(R\)-successor of \(w\).

At each world, the semantics of the connectives \(\neg, \land, \lor\) are defined recursively as in the classical propositional case:
The semantic definitions of the □ and ◊ connectives make use of the reachability relation $R$:

$$\vartheta(w, ◊\varphi) = \begin{cases} t & \text{if } \vartheta(v, \vartheta) = t \text{ for some } v \in W \text{ with } wRv \\ f & \text{otherwise} \end{cases}$$

$$\vartheta(w, □\varphi) = \begin{cases} t & \text{if } \vartheta(v, \vartheta) = t \text{ for every } v \in W \text{ with } wRv \\ f & \text{otherwise} \end{cases}$$

Propositional modal logic includes a semantic forcing relation $\vdash$, which is defined for a world $w$, a model $M = \langle W, R, \vartheta \rangle$ and given formula $\varphi$ or formula set $\Gamma$ as follows:

$$w \vdash \varphi = \begin{cases} t & \text{when } \vartheta(w, \varphi) = t \\ f & \text{otherwise} \end{cases}$$

$$M \vdash \varphi = \begin{cases} t & \text{when } \forall w \in W. w \vdash \varphi \\ f & \text{otherwise} \end{cases}$$

$$M \vdash \Gamma = \begin{cases} t & \text{when } \exists \varphi \in \Gamma. M \vdash \varphi \\ f & \text{otherwise} \end{cases}$$

From this, both the logical consequence relation $\models$ and satisfiability are defined, each over some set of models $\mathcal{K}$:

$$\Gamma \models \varphi \text{ iff } \forall M \in \mathcal{K}. M \vdash \Gamma \Rightarrow M \vdash \varphi$$

$$\varphi \text{ is } \mathcal{K}\text{-satisfiable} \text{ iff } \exists M = \langle W, R, \vartheta \rangle \in \mathcal{K}. \exists w \in W. w \vdash \varphi$$

It can be shown that logical consequence and satisfiability are related in the following way:

$$\Gamma \models \varphi \text{ iff } \Gamma \land \lnot \varphi \text{ is not } \mathcal{K}\text{-satisfiable}$$

Thus, a question of logical consequence can be translated into a question of satisfiability.

Sets of models are often characterised by restrictions on the reachability relation $R$. We shall consider two such sets of models: the set of all models (with no restrictions on $R$); and the set of all models for which $R$ is both reflexive and transitive. We shall denote the logics corresponding to these restrictions by $K$ and $S4$ respectively.
As the more difficult case, we shall focus on S4 from here onward.

Different propositional modal logic formulae may be semantically equivalent, so it is important to define a canonical form. For this we shall use Negation Normal Form, which can be recursively defined as follows:

\[
\begin{align*}
nnf(p) &= p \\
nnf(\neg p) &= \neg p \\
nnf(\varphi \land \psi) &= nnf(\varphi) \land nnf(\psi) \\
nnf(\varphi \lor \psi) &= nnf(\varphi) \lor nnf(\psi) \\
nnf(\Box \varphi) &= \Box nnf(\varphi) \\
nnf(\Diamond \varphi) &= \Diamond nnf(\varphi) \\
nnf(\neg(\varphi \land \psi)) &= nnf(\neg \varphi) \lor nnf(\neg \psi) \\
nnf(\neg(\varphi \lor \psi)) &= nnf(\neg \varphi) \land nnf(\neg \psi) \\
nnf(\neg \Box \varphi) &= \Diamond nnf(\neg \varphi) \\
nnf(\neg \Diamond \varphi) &= \Box nnf(\neg \varphi) \\
nnf(\neg \neg \varphi) &= nnf(\varphi)
\end{align*}
\]

We also define a shorthand for sets of formulae:

\[
nnf(Y) = \{nnf(\varphi) \mid \varphi \in Y\}
\]

### 2.2 Tableaux

We assume the reader is familiar with modal tableaux, so here we present the procedure of tableau as presented by Goré [3].

Tableaux are an inference procedure that can be used for determining satisfiability (and thus logical consequence) in modal logic. Broadly, the procedure consists of constructing a tree of nodes (a K-tableau) for a given formula set \(Y\), where each branch from a parent node to a child is created by instantiating one of several rules of inference. The resulting tree is said to be either open or closed, which corresponds to the original formula set \(Y\) being either satisfiable or unsatisfiable respectively.

Each node in a K-tableau consists of a multiset of modal formulae. However, the following lemma allows us to treat them as sets:

**Lemma 1.** \(\varphi; X \text{ has a closed K-tableau} \iff \varphi; \varphi; X \text{ has a closed K-tableau}\)

Each rule (shown below for S4, for formulae in negation normal form) consists of a numerator and a denominator, which mean that if the numerator is \(K\)-satisfiable, then some denominator (separated by ‘|’) must be \(K\)-satisfiable.

\[
\begin{align*}
(id) &:\quad \neg \varphi; X \quad \quad (\land) &:\quad \varphi \land \psi; X \\
&\quad \varphi; \psi; X \\
&\quad \varphi \lor \psi; X \\
&\quad \Box \varphi; X \\
&\quad \varphi; (\Box \varphi)^*; X \\
&\quad \Box \varphi \text{ not starred}
\end{align*}
\]
(\langle S4 \Gamma \rangle \models \varphi; \Box X^*; Z) \forall \psi. \Box \psi, \Box \psi \notin Z

The rules can be categorised into static rules and transitional rules. The rules (id), (\lor), (\lor) and (T) are static rules. This means that the numerator and denominator refer to the same world.

The (\langle S4 \Gamma \rangle) rule is a transitional rule, which means that the denominator does not refer to the same world as the numerator; instead it refers to an R-successor of the numerator world.

To construct a K-tableau for a formula set Y, first a root node is created that contains Y. Then, child nodes are created by instantiating rules of inference. If every leaf of the resulting tree is an instance of the (id) rule, then the K-tableau is said to be closed, otherwise it is open.

\[ \Box p \land \Box \neg p \lor \Box p \land \neg p \]
\[ \Box p; \Box \neg p (\lor) \]
\[ \Box p; \Box \neg p \lor \Box p (T) \]
\[ \Box p; \Box \neg p \lor \Box p (\langle S4 \Gamma \rangle) \]
\[ \Box p^*; p; \Box \neg p (T) \]
\[ \Box p; \Box \neg p (T) \]
\[ \Box p^*; p; \Box \neg p (T) \]
\[ \Box p^*; p; \Box \neg p (T) \]
\[ \Box p^*; p; \Box \neg p (T) \]
\[ \Box p^*; p; \Box \neg p (T) \]
\[ \Box p^*; p; \Box \neg p (T) \]
\[ \Box p^*; p; \Box \neg p (T) \]

The above example shows a closed S4-tableau for the formula set Y = {nnf(\neg(\Box p \to \Box \Box p \land \Box p \to p))}

In practice, this construction can be done in phases. First, there can be a saturation phase, where only static rules (id), (\lor) and (\lor) are instantiated. This phase is purely propositional; no modal aspects are involved. When there are no propositional rules left to instantiate, then the (T) rule can be instantiated, which may start a new saturation phase. Only when there are no possible static rules to instantiate at a leaf is the transitional rule instantiated, which is called a modal jump. This creates a node for which there may be static rules to instantiate, so the phases repeat.

It can be shown that this procedure is both sound and complete:

**Theorem 1.** If there is a closed K-tableau for Y then Y is not K-satisfiable.

**Theorem 2.** If there is no closed K-tableau for Y then Y is K-satisfiable.

**Corollary 1.** A formula \( \varphi \) is a logical consequence of a formula set \( \Gamma \) iff there is a closed K-tableau for \( \Gamma \uplus \neg \varphi \).

\[ \Box a; \neg a \lor \Box (b \land \neg b) \]
\[ \Box a; \Box (b \land \neg b) (\lor) \]
\[ \Box a^*; a; \Box (b \land \neg b) (T) \]
\[ \Box a; b \land \neg b (\lor) \]
\[ \Box a^*; a; \Box (b \land \neg b) (T) \]
\[ \Box a; b \land \neg b (\lor) \]
\[ \Box a; \neg a (T) \]
\[ \Box a^*; a; \Box (b \land \neg b) (T) \]
\[ \Box a; b \land \neg b (\lor) \]
\[ \Box a; \neg a (T) \]
\[ \Box a^*; a; \Box (b \land \neg b) (T) \]
\[ \Box a; b \land \neg b (\lor) \]
\[ \Box a; \neg a (T) \]
\[ \Box a^*; a; \Box (b \land \neg b) (T) \]
\[ \Box a; b \land \neg b (\lor) \]
\[ \Box a; \neg a (T) \]
\[ \Box a^*; a; \Box (b \land \neg b) (T) \]
\[ \Box a; b \land \neg b (\lor) \]
\[ \Box a; \neg a (T) \]
\[ \Box a^*; a; \Box (b \land \neg b) (T) \]
\[ \Box a; b \land \neg b (\lor) \]
\[ \Box a; \neg a (T) \]
Consider the above S4-tableau for $\Gamma = \{ \square a \}$, $\varphi = a \land \square (b \lor \neg b)$ and $mnf(\Gamma \cup \neg \varphi) = \{ \square a; \neg a \lor \Diamond (b \land \neg b) \}$. Since all the leaves of this tableau are instances of (id), this tableau is closed. We can then conclude that $\Gamma \cup \neg \varphi$ is not S4-satisfiable, or, equivalently (as shown above), that $\Gamma \models \varphi$ in S4.

2.3 Binary Decision Diagrams

At a high level, a Binary Decision Diagram (BDD) is a compact representation of a boolean function of boolean variables $f : \text{Var} \to \{ t, f \}$ [13]. At a low level, a BDD is a directed acyclic graph of bdd-nodes, which are either a variable-node, the true-node or the false-node. Variable-nodes are a tuple of three things $\langle v, \text{high}, \text{low} \rangle$: a variable number $v$, and two branches to other bdd-nodes, high and low. The true- and false-nodes are special terminal nodes that denote true and false respectively.

The high and low branches of a node represent the assignment of true and false to that node’s variable, respectively. So any path down a BDD represents a (potentially partial) valuation on the variables of the BDD, and the value of the function on that valuation is given by the terminal node reached - true for the true-node, false for the false-node.

BDDs as they are used here are more formally known as ROBDDs, for Reduced Ordered Binary Decision Diagrams. Ordered means that, on all paths from the root, all the variables appear in the same order. Reduced means that any equivalent sub-graphs have been merged, and any nodes whose high and low children were isomorphic have been eliminated.

These two properties make the BDD for a given function canonical, for a particular variable order. In practice, this means that equivalence checking between BDDs is simple, as only the root nodes need to be compared [11]. If the root nodes are different objects, then the BDDs must be different.

We shall denote the BDD that consists only of the false-node as the false-bdd, and that which consists only of the true-node as the true-bdd.

The next most simple BDDs are then the BDDs of exactly one variable. Due to the reduced property, there are only two forms such a BDD can take: either representing the same value as the variable, or representing it’s negation. We shall define a shorthand for these two BDDs:

$$v = BDD\langle v, \text{true-node}, \text{false-node} \rangle$$

$$\neg v = BDD\langle v, \text{false-node}, \text{true-node} \rangle$$

As boolean functions, BDDs can be negated and composed together using ‘and’ and ‘or’ connectives. So we can recursively define a mapping $[\cdot]$ from propo-
sitional modal formulae to BDDs:

\[
\begin{align*}
[p] &= v_p \\
[\square \varphi] &= v_{\square \varphi} \\
[\Diamond \varphi] &= \neg [\square \neg \varphi] \\
[\neg \varphi] &= \neg [\varphi] \\
[\varphi \land \psi] &= [\varphi] \land [\psi] \\
[\varphi \lor \psi] &= [\varphi] \lor [\psi]
\end{align*}
\]

It is important to note that this mapping is both purely propositional and modally shallow: any modal formula $\square \varphi$ is simply represented by a BDD variable $v_{\square \varphi}$, and the mapping “stops” when it reaches a modal formula. (Note also that any $\Diamond \varphi$ are translated into equivalent $\neg \neg \varphi$, so as to have a canonical BDD representation of modal formulae.) The point to stress is that such a BDD treats modal formulae as if they were atomic propositions, and requires that they have a BDD variable number $v_{\square \varphi}$ as atomic propositions do.

As constructed, $[\varphi]$ then represents a function that is true on exactly all valuations that satisfy $\varphi$ at a propositional level. As such, all paths to the true-node in $[\varphi]$ (all valuations on which $[\varphi]$ is true) represent all the open leaves of a fully saturated tableau for $\varphi$.

The correspondence between valuations and tableau leaves is not quite one-to-one, as the following example demonstrates.

On the left we have the tableau for $a \lor b$, and on the right we have the BDD $[a \lor b]$ (where the low branch is represented by a dashed line, the high branch is represented by a solid line). Each of the two tableau branches only contain one of the two atomic propositions, due to the $(\lor)$ rule. But $[a \lor b]$, with the variable order $v_a < v_b$, contains the valuation $a = false$, $b = true$, which would be equivalent to the tableau node ‘$a; \neg b$’.

This is acceptable because, no matter the variable ordering, the BDD will always represent all propositionally satisfying valuations, just not necessarily in the same form as they would appear in tableau. The set of valuations represented by both is semantically equivalent, but may be represented differently, so the satisfiability of both representations is also equivalent.

3 Method

The basics of our method are to follow the tableau procedure as described in Section 2.2, but to use BDDs to perform the saturation phase whenever it is needed, using the processes described in Section 2.3. Thus the basic algorithm is to compute a saturation phase using BDDs, get a satisfying valuation from the
saturation, then compute either an unboxing phase via \( (T) \) or a modal jump via \( (\diamond S4\Gamma) \) as necessary. Both of these cases are explored in a recursive, depth-first fashion; unboxing is followed by further exploration of the tableau branch that included the boxes, and modal jumps are followed by exploration of the tableau branch beyond the modal jump. Prominent base cases are when a saturation phase closes completely (results in the false-bdd, unsatisfiable), and when there are no \( (T) \) or \( (\diamond S4\Gamma) \) rules to apply to a leaf (satisfiable). Typical actions on return from recursion are passing along satisfiability (all necessary child branches were satisfiable), or eliminating the explored leaf and choosing another (child branches were unsatisfiable). Specifics vary from case to case.

An important part of the way BDDs are used is in the case when a modal jump closes. This means that a branch that was created in the saturation phase is now closed, and another one needs to be explored. The naive way of achieving this would be to traverse the BDD for another satisfying valuation, until all have been explored. Instead, we refine the BDD in such a way that the closed leaf will be removed from the set of satisfying valuations, and then ask for a new satisfying valuation from the refined BDD. In general, this refinement is of the form \( BDD_f = BDD_i \land \neg (v_0 \land v_1 \land \ldots) \), where \( BDD_i \) is the initial BDD, \( BDD_f \) is the final BDD, and \( v_i \) are some of the bdd-variables of the satisfying valuation branch that closed.

Consider the following simple example:

\[
\frac{a \lor \diamond(b \land \neg b)}{a} \quad \frac{\diamond(b \land \neg b)}{b \land \neg b} \quad \frac{b; \neg b}{(\diamond S4\Gamma)} \quad (\lor)
\]

On the left we have the tableau for \( \{a \lor \diamond(b \land \neg b)\} \). The first BDD on the right is \( [a \lor \diamond(b \land \neg b)] \). As shown in the tableau, exploring the modal jump over \( \diamond(b \land \neg b) \) will close, so on return we would refine \( [a \lor \diamond(b \land \neg b)] \) with \( [\neg (\diamond(b \land \neg b))] \). This would give us the BDD \([a]\), from which we would get a new satisfying valuation, and continue the procedure. Thus the refined BDD would represent the open branch \( a \), just as in the tableau.

The idea behind this lies in the fact that this process has the potential to eliminate many more branches than the one actually explored. For instance, if \( a \) in the previous example were some more complicated formula, we would have also eliminated every other branch that included \( \diamond(b \land \neg b) \).

The flipside of this is that the refinement process may introduce variables to a branch that were otherwise irrelevant. This is similar to the case discussed in Section 2.3, and, again, is a purely representational difference.

### 3.1 Optimisations

Given the choice of refining with \( \neg(v_0) \) or \( \neg(v_0 \land v_1) \), we would prefer to use \( \neg(v_0) \) as it is a stronger result; refining with \( \neg(v_0) \) would eliminate at least as
many branches as \( ¬(v_0 \land v_1) \). Thus a significant optimisation of this algorithm is to reduce the number of variables included in any refinement.

In the worst case, a modal jump refinement must include every modal variable before the modal jump. It may be that every one of those variables are required for the modal jump to close; if any one of them were not present, it may have stayed open. The task is then to find a minimal subset of the variables, and refine over those. A minimal subset is a subset which is unsatisfiable (would close) and for which removing any element would make it satisfiable (would stay open).

To achieve this, we include with any return of unsatisfiable a set of variables that were deemed directly “responsible” for the unsatisfiable result. As our algorithm is recursive overall, this aspect is also recursive.

The base case is when a modal jump closes immediately because the subsequent saturation returned the false-bdd. In this case we can incrementally reconstruct the saturation phase from the modal variables, and determine a definitely minimal subset that still yields the false-bdd. The algorithm we use to do this gives us a minimal subset, but it does not guarantee that it will be the smallest such subset.

In the general case, we will only be given which variables beyond the jump were responsible. From this, we can only over-approximate which modal variables were responsible for those variables, as the details of those relationships are hidden within the BDDs we construct. The approximation we use is to consider all the subformulae of a modal variable - the variables that are created from an application of the \((\lozenge S4\Gamma)\) or \((T)\) rules. If one of its subformulae is a responsible variable, then we consider it a responsible variable. But we must also consider how that modal variable interacted with other modal variables, so we also include any other modal variables who share subformulae with it, and could have potentially interacted with it.

Consider the following example:

\[
\begin{align*}
\lozenge (b \lor \lozenge (a \land \neg a)) ; \Box (\neg b)^* ; \neg b \\
\vdots \\
\lozenge (a \land \neg a) ; \Box (\neg b)^* ; \neg b \\
\vdots \\
\times \text{ (id)}
\end{align*}
\]

From the middle to the bottom, we have a modal jump (modal jump 2) that closed immediately due to \( a \land \neg a \). This is our base case, and we would identify that \( \lozenge (a \land \neg a) \) is the only variable responsible for modal jump 2 closing. As there is only one branch, this result and the responsible variable would be returned to the modal jump at the top (modal jump 1). We would then identify that \( \lozenge (b \lor \lozenge (a \land \neg a)) \) is responsible for modal jump 1 closing, because its subformula \( \lozenge (a \land \neg a) \) was returned as a responsible variable from beyond modal jump 1. We would then identify \( \Box (\neg b)^* \) as also responsible for modal jump 1 closing, as it shares a subformula \( b \) with \( \lozenge (b \lor \lozenge (a \land \neg a)) \).
This case demonstrates the necessity of considering interactions between modal variables, as $\lozenge (b \lor \lozenge (a \land \neg a))$ is not unsatisfiable by itself, even though it is the only modal variable that creates $\lozenge (a \land \neg a)$. It is the interaction between $\lozenge (b \lor \lozenge (a \land \neg a))$ and $\Box (\neg b)^*$ that leads to unsatisfiability.

Some less detailed optimisations involve storing results to be reused later.

The most simple in $\textbf{S4}$ is to store each $\Box \varphi$ and the result of its unboxing $\varphi$, as every modal jump after they are created will include them, due to the transitivity of $\textbf{S4}$. Storing them avoids recomputing the $(T)$ rule for every $\Box \varphi$ at every modal jump.

On any result of satisfiable, we cache the BDD that was satisfiable. At every modal jump we can lookup this cache to see if we have determined a result for this BDD already, and avoid recomputing it. This lookup is efficient, as it is just a comparison of BDD root nodes (integers), for which a hash table can be used. To prevent large memory usage, we limit the size of the cache, beyond which we remove an element before inserting the next one. Elements are removed on a first-in-first-out basis.

On any unsatisfiable modal jump, we cache the refinement we make upon return. For all modal jumps explored, we iterate through this cache, and if the variables in a cached refinement are a subset of the variables of the modal jump, we apply that refinement to the modal jump. The subset check is there to avoid introducing otherwise irrelevant variables, as discussed above. This cache is also size-limited, with a first-in-first-out removal policy.

Consider again the example shown previously in Section 3:

\[
\begin{array}{c}
\alpha \lor \lozenge (b \land \neg b) \\
\hline
\lozenge (b \land \neg b) \quad \lor \\
\hline
\Box \neg (b \land \neg b) \quad \Box S4 \Gamma \\
\hline
\Box b \land \neg b \quad \land \\
\hline
\neg b \quad \Box \neg b \\
\hline
\end{array}
\]

In this example the modal jump over $\lozenge (b \land \neg b)$ was explored, which closed. On return we refined our initial BDD with $\neg \lozenge (b \land \neg b)$. We would cache $\neg \Box \neg (b \land \neg b)$, and then apply it to every future modal jump which contains the variable $\neg \Box (b \land \neg b)$. Thus, from this point in time onward, no branches will ever be explored that contain $\lozenge (b \land \neg b)$, as we have learnt that it is unsatisfiable.

4 Results

We evaluate the performance of our BDD based method against InKreSAT [8] and FaCT++ [12]. Both are representative of state-of-the-art reasoners in this field. We ran InKreSAT with default settings.

We have followed Kaminski and Tebbi [8] in their choice of benchmarks. Our tests were performed on an Intel 3.47GHz CPU with 4GB of memory, and were run with a time limit of 30 seconds per formula.

Table 1 contains results for the Logic Work Bench (LWB) benchmark sets for $\textbf{K}$ and $\textbf{S4}$ [6]. Each subclass in the LWB consists of a formula shape which
accepts are argument \( n \) and generates a formula. As \( n \) increases, these formulae become harder to solve. The entries in Table 1 mark the highest problem instance solved by each program, where each program was given a 30 second time limit per formula. For each subclass the best result is marked in bold. The LWB benchmarks are a common benchmark set for evaluating modal reasoners.

On the LWB benchmarks we see that our BDD-based approach achieves results comparable to both FaCT++ and InKreSAT. We complete every subclass that both FaCT++ and InKreSAT complete, and gain the highest mark for several subclasses. There are some subclasses where our BDD-based approach performs quite poorly, but these are only few.

In the case of the \( \text{ph}_n \) subclass for K, this poor result is to be expected. In this class the formulae are primarily propositional, so the saturation phase is large. The correct result when using tableau methods for this class is that they are satisfiable, so tableau-based approaches can halt as soon as they find a single open branch. Our approach, however, always computes the entirety of the saturation phase in one hit, and so in this class will waste a lot of time finding every open branch, when only one is needed.

Figure 1 provides a little more detail into the LWB results. It shows actual time taken by each program per instance for the \( \text{K}_{d4,n} \) and \( \text{K}_{\text{path},n} \) subclasses. They demonstrate that Table 1 does not tell the whole story; even within subclasses where all provers make it to instance 21, there can be significant differences in the performance of each program.

Figure 2 shows results for a set of randomly generated 3CNF \( K \) formulae of modal depth 2, 4 and 6, as well as for a subset of the TANCS-2000 [10]

Table 1. Results on the LWB benchmarks for K (left) and S4 (right)

<table>
<thead>
<tr>
<th>subclass</th>
<th>BDD-based</th>
<th>FaCT++</th>
<th>InKreSAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>branch_n</td>
<td>16</td>
<td>10</td>
<td>13</td>
</tr>
<tr>
<td>branch_p</td>
<td>21</td>
<td>9</td>
<td>16</td>
</tr>
<tr>
<td>d4_n</td>
<td>21</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>d4_p</td>
<td>21</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>dum_n</td>
<td>21</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>dum_p</td>
<td>21</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>grz_n</td>
<td>21</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>grz_p</td>
<td>21</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>lin_n</td>
<td>21</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>lin_p</td>
<td>21</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>path_n</td>
<td>21</td>
<td>21</td>
<td>10</td>
</tr>
<tr>
<td>path_p</td>
<td>21</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>ph_n</td>
<td>9</td>
<td>13</td>
<td>21</td>
</tr>
<tr>
<td>ph_p</td>
<td>9</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>poly_n</td>
<td>21</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>poly_p</td>
<td>21</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>t4p_n</td>
<td>21</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>t4p_p</td>
<td>21</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>45_n</td>
<td>21</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>45_p</td>
<td>21</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>branch_n</td>
<td>18</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>branch_p</td>
<td>21</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>grz_n</td>
<td>21</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>grz_p</td>
<td>21</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>ipc_n</td>
<td>12</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>ipc_p</td>
<td>21</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>md_n</td>
<td>11</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>md_p</td>
<td>17</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>path_n</td>
<td>3</td>
<td>21</td>
<td>4</td>
</tr>
<tr>
<td>path_p</td>
<td>4</td>
<td>21</td>
<td>5</td>
</tr>
<tr>
<td>ph_n</td>
<td>5</td>
<td>8</td>
<td>14</td>
</tr>
<tr>
<td>ph_p</td>
<td>6</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>s5_n</td>
<td>21</td>
<td>20</td>
<td>15</td>
</tr>
<tr>
<td>s5_p</td>
<td>21</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>t4p_n</td>
<td>21</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>t4p_p</td>
<td>21</td>
<td>21</td>
<td>21</td>
</tr>
</tbody>
</table>
Fig. 1. LWB classes $K_{\text{path}}$ and $K_{\text{d4}}$.

Fig. 2. 3CNF$_K$ and MQBF benchmarks

Unbounded Modal QBF (MQBF) benchmarks for K. The 3CNF$_K$ formulae are intended to show how performance depends on modal depth.

On the 3CNF$_K$ benchmark set, our BDD-based approach shows the best performance. However, our approach shows the worst performance on the MQBF benchmarks.

5 Further Work

There are several avenues for further work in this area.

Determining an effective BDD variable reordering scheme could cut BDD operation times. Some naive schemes were tested here without any significant results, such as using the automatic reordering provided by our BDD package.

Creating good heuristics for modal jump exploration order or tableau branch exploration order could potentially see much improvement.

The caches could be improved by using a better policy for removing elements.
Huang [7] presents efficient techniques, using BDDs, for finding the smallest minimally unsatisfiable subset. Using these techniques could improve upon the algorithm used here, that does not guarantee to find the smallest such set.

6 Conclusion

Our work shows that BDDs can be an effective base data structure for computing tableaux for propositional modal logic. The results explicitly show that this is the case for K and S4 with respect to the state-of-the-art as represented by FaCT++ and InKreSAT. These results could be improved upon, but they demonstrate that this approach can at least be competitive against other methods.
Bibliography