Optimising Propositional Modal Satisfiability for Description Logic Subsumption

Ian Horrocks[1] and Peter F. Patel-Schneider[2]

[1] University of Manchester, Manchester, UK (horrocks@cs.man.ac.uk)
[2] Bell Labs Research, Murray Hill, NJ, U.S.A. (pfps@research.bell-labs.com)

Abstract. Effective optimisation techniques can make a dramatic difference in the performance of knowledge representation systems based on expressive description logics. Because of the correspondence between description logics and propositional modal logic many of these techniques carry over into propositional modal logic satisfiability checking. Currently-implemented representation systems that employ these techniques, such as FaCT and DLP, make effective satisfiable checkers for various propositional modal logics.

1 Introduction

Description logics are a logical formalism for the representation of knowledge about individuals and descriptions of individuals. Description logics represent and reason with descriptions similar to "all people whose friends are both doctors and lawyers" or "all people whose children are doctors or lawyers or who have a child who has a spouse". The computations performed by systems that implement description logics are based around determining whether one description is more general than (subsumes) another. There have been various schemes for computing this subsumption relationship, depending on the expressive power of the description logic and the degree of completeness of the system. As description logic systems perform numerous subsumption checks in the course of their operations, they need to have a highly-optimised subsumption checker.

Recent work [16] has shown that determining subsumption in expressive description logics is equivalent to determining satisfiability of formulae in propositional modal or dynamic logics. Thus one part of a system that implements a description logic is equivalent to a satisfiability checker for a propositional modal or dynamic logic. Several description logic systems have been built for such description logics, and thus include what is essentially a satisfiability checker, including KRIS [2] and CRACK [5]. These two systems have incorporated a number of optimisations to achieve better performance of their subsumption checkers.

Description logic systems are also optimised in other ways. In particular, their operations are optimised to avoid the potentially-costly subsumption checks whenever possible. There are also other optimisations to subsumption possible in description logic systems, having to do with the nature of the representation of knowledge in a description logic, but these have little or nothing to do with optimising propositional modal satisfiability.

We have built two systems that explore the optimisations required to build an expressive description logic system, namely FaCT [11], a full description logic system,

and DLP [14], an experimental system providing only a limited description logic interface. FaCT is available at `http://www.cs.man.ac.uk/~horrocks`; DLP is available at `http://www-db.research.bell-labs.com/user/pfps`.

We have incorporated a range of known, adapted and novel optimisation techniques into the subsumption checkers for these two systems. The optimisation techniques include: lexical normalisation, semantic branching search, boolean constraint propagation, dependency directed backtracking, heuristic guided search and caching.

These optimisations techniques make a drastic difference to the performance of the overall system. As evidence, KRIS is not able to load a modified version of the GALEN knowledge base because it gets stuck trying to determine one of the thousands of subsumptions required to load the knowledge base. FaCT and DLP, which have higher levels of optimisation, are able to easily load this knowledge base, classifying over two thousand definitions in about two hundred seconds.

We have also performed experiments with both FaCT and DLP on several test suites of propositional modal formulae. The optimisations built into the two systems qualitatively change their behaviour on the test suites, indicating that the optimisations have considerable utility simply taken as optimisations for reasoning in propositional modal logics.

2 Background

FaCT and DLP are designed to build and maintain taxonomies of named concepts. Given a collection of definitions of named concepts and statements about these concepts, they determine the subsumption partial order for the named concepts. To do this they have to determine subsumption relationships between descriptions in a description logic.

The description logic that DLP implements is called $\mathcal{ALC}_{R^+}$. FaCT implements a considerably more-expressive logic, but most of the satisfiability optimisations in FaCT are demonstrable in $\mathcal{ALC}_{R^+}$. $\mathcal{ALC}_{R^+}$ is built up from atomic concepts and two kinds of atomic roles, non-transitive roles and transitive roles. Concepts in $\mathcal{ALC}_{R^+}$ are formed using the grammar $\mathsf{A} \mid \top \mid \bot \mid \neg C \mid C \sqcap D \mid C \sqcup D \mid \exists R.C \mid \forall R.C \mid \exists T.C \mid \forall T.C$,[1] where A is an atomic concept, C and D are concept expressions, R is a non-transitive role, and T is a transitive role.

The semantics of $\mathcal{ALC}_{R^+}$ is a standard extensional semantics, using an interpretation $\mathcal{I}$ that is a pair $(\Delta^{\mathcal{I}}, .^{\mathcal{I}})$ consisting of a domain and a mapping from concepts to subsets of the domain and from roles to binary relations on the domain (transitive relations for transitive roles, of course). The semantics for concept expressions are given in Table 1. One concept then subsumes another if and only if the extension of the first concept includes the extension of the second in all interpretations.

The semantics of $\mathcal{ALC}_{R^+}$ is a simple transformation of the possible world semantics for propositional modal logics. In this transformation elements of the domain correspond to possible worlds, atomic concepts correspond to propositional variables, and

[1] Throughout the paper, we will be using the syntax of description logics. To translate into the syntax of modal propositional logics, replace $\forall R$ with $\Box_R$ and $\exists R$ with $\Diamond_R$ and perform several other obvious replacements.

Syntax	Semantics
A	$\mathsf{A}^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
$\top$	$\Delta^{\mathcal{I}}$
$\bot$	$\emptyset$
$\neg C$	$\Delta^{\mathcal{I}} - C^{\mathcal{I}}$
$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
$C \sqcup D$	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$
$\exists R.C$	$\{d \in \Delta^{\mathcal{I}} \mid R^{\mathcal{I}}(d) \cap C^{\mathcal{I}} \neq \emptyset\}$
$\forall R.C$	$\{d \in \Delta^{\mathcal{I}} \mid R^{\mathcal{I}}(d) \subseteq C^{\mathcal{I}}\}$
$\exists T.C$	$\{d \in \Delta^{\mathcal{I}} \mid T^{\mathcal{I}}(d) \cap C^{\mathcal{I}} \neq \emptyset\}$
$\forall T.C$	$\{d \in \Delta^{\mathcal{I}} \mid T^{\mathcal{I}}(d) \subseteq C^{\mathcal{I}}\}$

Table 1. Semantics of $\mathcal{ALC}_{R^+}$ concept expressions

roles correspond to modalities. This transformation shows that fragments of $\mathcal{ALC}_{R^+}$ correspond to $\mathbf{K}_{(\mathbf{m})}$ and $\mathbf{K4}_{(\mathbf{m})}$. Transitive roles in $\mathcal{ALC}_{R^+}$ are used for $\mathbf{K4}_{(\mathbf{m})}$ and non-transitive roles are used for $\mathbf{K}_{(\mathbf{m})}$. $\mathcal{ALC}_{R^+}$ can also express formulae in $\mathbf{KT}_{(\mathbf{m})}$ and $\mathbf{S4}_{(\mathbf{m})}$ via the usual encoding that maps $\forall R.C$ into $C \sqcap \forall R.C$, etc.

Determining subsumption in $\mathcal{ALC}_{R^+}$ is PSPACE-complete [15], as is the related problem of determining whether a concept in $\mathcal{ALC}_{R^+}$ is satisfiable. However, it is possible to build practical description logic systems based on expressive description logics [2, 5, 11] that have this sort of computationally intractable subsumption. Systems that are based on description logics like $\mathcal{ALC}_{R^+}$ generally determine whether a subsumption holds by transforming the subsumption question into a satisfiability question and then attempting to construct a model for this concept, just as a tableaux satisfiability checker for a propositional logic attempts to construct a model for a formula. During this process, various nodes are created, where each node represents an individual (possible world), and tells whether the individual belongs to various concepts (gives values to formulae at this world). This set of concepts is said to form the label of the node—we will use $\mathcal{L}(x)$ to denote the label of a node x. The nodes are connected by modal relationships in a tree fashion, starting at a root node. If a node is related to another node via role R, the second node is called an R-successor of the first.

The basic algorithm starts out with a single node representing an individual (possible world) that must be in the extension of the concept being tested for satisfiability (must have a formula evaluate to true at it). This concept (formula) is expanded to produce simpler concepts that must have the individual in their extension (simpler formulae that evaluate at the world). Disjunctive concepts (formulae) give rise to choice points in the algorithm (branches in the tableau). Existential role concepts, $\exists R.C$, (existential modal formulae) cause the creation of new successor nodes representing other individuals (possible worlds).

Universal role concepts (universal modal formulae) augment the concepts that these individual must belong to (formulae that are true at these possible worlds). In order to guarantee termination, transitive roles (transitive modalities) require filtration or *blocking*: a check to ensure that no other node has the same set of concepts (formulae)—if so,

the two nodes can be collapsed into a cycle. If the algorithm constructs a collection of nodes where there are no compound concepts (formulae) that have not been expanded and where there are no obvious contradictions, called *clashes*, at any of the nodes, then the collection of nodes corresponds to a model for the initial concept (formula). If the algorithm fails to construct such a collection then the initial concept (formula) has no model—it is said to be *unsatisfiable*.

The details of the algorithm, including precise termination conditions, are fairly standard, and can be found in [15].

3 Optimisation Techniques

The basic algorithm given above is too slow to form the basis of a useful description logic system. We have therefore investigated and employed a range of known, adapted and novel optimisations that improve the performance of the satisfiability testing algorithm, including lexical normalisation, semantic branching search, boolean constraint propagation, dependency directed backtracking, heuristic guided search, and caching.

Theoretical descriptions of tableaux algorithms generally assume that the concept expression to be tested is in negation normal form, with negations applying only to atomic concepts. This simplifies the (description of the) algorithm but it means that a clash will only be detected when an atomic concept and its negation occur in the same node label. For example, when testing the satisfiability of the concept expression $\exists R.(C \sqcap D) \sqcap \forall R.\neg C$, where C is an atomic concept, a clash would be detected when the algorithm creates an R-successor y because $\{C, \neg C\} \subseteq \mathcal{L}(y)$. However, if C is a concept expression, then the clash would not be detected immediately because $\neg C$ would have been transformed into negation normal form. If C is a large or complex expression this could lead to costly wasted expansion.

This problem is addressed by transforming concept expressions into a lexically normalised form, and by identifying lexically equivalent expressions. All concepts can then be treated equally, whether or not they are atomic, with a clash being detected whenever a concept expression and its negation occur in the same node label.[2] In lexically normalised form, concept expressions consist only of (possibly negated) atomic concepts, conjunction concepts and universal role concepts: expressions of the form $\exists R.C$ are transformed into $\neg(\forall R.\neg C)$ and expressions of the form $(C_1 \sqcup, \ldots, \sqcup C_n)$ are transformed into $\neg(\neg C_1 \sqcap, \ldots, \sqcap \neg C_n)$, where the $C_1, \ldots, C_n$ are sorted and duplicates are eliminated. The normalisation process can also include simplifications such as $\forall R.\top \longrightarrow \top$, $(\bot \sqcap \ldots) \longrightarrow \bot$ and $(C \sqcap \neg C \sqcap \ldots) \longrightarrow \bot$; in extreme cases the need for a tableau expansion can be completely eliminated by simplifying expressions to $\top$ or $\bot$. Efficiency can be further enhanced by tagging each lexically distinct expression with a unique code so that equivalent expressions can be identified simply by comparing tags.[3]

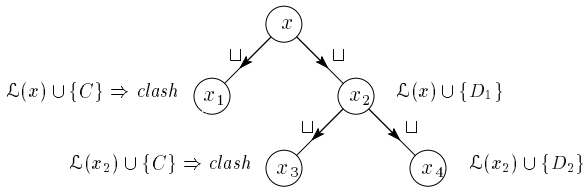
Tableau expansion of concepts in this form is no more complex than if they are in negation normal form: $\neg(\forall R.C)$ can be dealt with in the same way as $\exists R.\neg C$ and

[2] KRIS addresses the same problem, in a less complete manner, by lazily expanding named concepts, and retaining their names in node labels [1].

[3] A similar technique is used in KSAT, but without the benefit of tagging [9].

$\neg(C_1 \sqcap, \ldots, \sqcap C_n)$ can be dealt with in the same way as $(\neg C_1 \sqcup, \ldots, \sqcup \neg C_n)$. The expression $\exists R.(C \sqcap D) \sqcap \forall R.\neg C$ would be transformed into $\neg(\forall R.\neg(C \sqcap D)) \sqcap \forall R.\neg C$, and the $\neg(\forall R.\neg(C \sqcap D))$ term would lead directly to the creation of an R-successor whose label contained both C and $\neg C$. As the two occurrences of C will be lexically normalised and tagged as the same concept, a clash will immediately be detected, regardless of the structure of C.

Standard tableaux algorithms are inherently inefficient because they use a search technique based on syntactic branching. When expanding the label of a node x, syntactic branching works by choosing an unexpanded disjunction in $\mathcal{L}(x)$ and searching the different models obtained by adding each of the disjuncts. As the alternative branches of the search tree are not disjoint, there is nothing to prevent the recurrence of an unsatisfiable disjunct in different branches [9]. The resulting wasted expansion could be costly if discovering the unsatisfiability requires the solution of a complex sub-problem. For example, tableau expansion of a node x, where $\{(C \sqcup D_1), (C \sqcup D_2)\} \subseteq \mathcal{L}(x)$ and C is an unsatisfiable concept expression, could lead to the search pattern shown below, where the unsatisfiability of C must be demonstrated twice.



This problem is dealt with by using a semantic branching technique adapted from the Davis-Putnam-Logemann-Loveland procedure (DPLL) commonly used to solve propositional satisfiability (SAT) problems [6, 8]. Instead of choosing an unexpanded disjunction in $\mathcal{L}(x)$, a single disjunct D is chosen from one of the unexpanded disjunctions in $\mathcal{L}(x)$. The two possible sub-trees obtained by adding either D or $\neg D$ to $\mathcal{L}(x)$ are then searched. Because the two sub-trees are strictly disjoint, there is no possibility of wasted search as in syntactic branching.

An additional advantage of using a DPLL based search technique is that a great deal is known about the implementation and optimisation of this algorithm. In particular, both *boolean constraint propagation* and *heuristic guided search* can be used to try to minimise the size of the search tree.

Boolean constraint propagation (BCP) is a technique used to maximise deterministic expansion, and thus pruning of the search tree via clash detection, before performing non-deterministic expansion (branching) [8]. Before semantic branching is applied to the label of a node x, BCP deterministically expands disjunctions in $\mathcal{L}(x)$ which present only one expansion possibility and detects a clash when a disjunction in $\mathcal{L}(x)$ has no expansion possibilities. The number of expansion possibilities presented by a disjunction $(C_1 \sqcup \ldots \sqcup C_n) \in \mathcal{L}(x)$ is equal to the number of disjuncts C_i such that $\neg C_i \notin \mathcal{L}(x)$. In effect, BCP is using the inference rule $\frac{\neg C, C \sqcup D}{D}$ to simplify the expression represented by $\mathcal{L}(x)$.

For example, given a node x such that $\{(C \sqcup (D_1 \sqcap D_2)), (\neg D_1 \sqcup \neg D_2), \neg C\} \subseteq \mathcal{L}(x)$, BCP deterministically expands the disjunction $(C \sqcup (D_1 \sqcap D_2))$ because $\neg C \in$

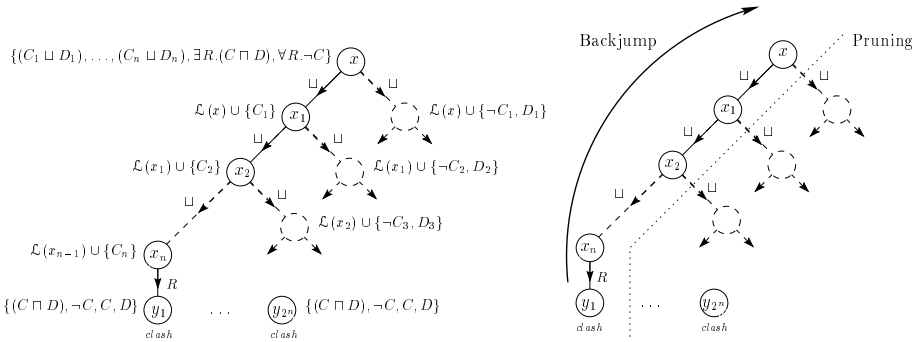


Fig. 1. Thrashing in backtracking search/Backjumping

$\mathcal{L}(x)$. The expansion of $(D_1 \sqcap D_2)$ adds both D_1 and D_2 to $\mathcal{L}(x)$, allowing BCP to identify $(\neg D_1 \sqcup \neg D_2)$ as a clash without any branching having occurred.

Inherent unsatisfiability concealed in sub-problems can lead to large amounts of unproductive backtracking search known as thrashing. The problem is exacerbated when blocking is used to guarantee termination, because blocking may require that sub-problems only be explored after all other forms of expansion have been performed. For example, expanding a node x, where $\mathcal{L}(x) = \{(C_1 \sqcup D_1), \ldots, (C_n \sqcup D_n), \exists R.(C \sqcap D), \forall R.\neg C\}$, would lead to the fruitless exploration of 2^n possible R-successors of x before the inherent unsatisfiability is discovered. The search tree created by the tableau expansion algorithm is illustrated in Fig. 1.

This problem is addressed by adapting a form of dependency directed backtracking called *backjumping*, which has been used in solving constraint satisfiability problems [3] (a similar technique was also used in the HARP theorem prover [13]). Backjumping works by labeling concept expressions with a dependency set indicating the branching points on which they depend. A concept expression $C \in \mathcal{L}(x)$ depends on a branching point when C was added to $\mathcal{L}(x)$ at the branching point or when $C \in \mathcal{L}(x)$ depends an another concept expression $D \in \mathcal{L}(y)$, and $D \in \mathcal{L}(y)$ depends on the branching point. A concept expression $C \in \mathcal{L}(x)$ depends on a concept expression $D \in \mathcal{L}(y)$ when C was added to $\mathcal{L}(x)$ by a deterministic expansion which used $D \in \mathcal{L}(y)$, e.g., if $A \in \mathcal{L}(x)$ was derived from the expansion of $(A \sqcap B) \in \mathcal{L}(x)$, then $A \in \mathcal{L}(x)$ depends on $(A \sqcap B) \in \mathcal{L}(x)$.

When a clash is discovered, the dependency sets of the clashing concepts can be used to identify the most recent branching point where exploring the other branch might alleviate the cause of the clash. The algorithm can then jump back over intervening branching points *without* exploring alternative branches.

For example, when expanding the node x from the previous example, the search algorithm will perform a sequence of n branches, eventually leading to the node x_n with $\{\exists R.(C \sqcap D), \forall R.\neg C\} \subset \mathcal{L}(x_n)$. When $\exists R.(C \sqcap D) \in \mathcal{L}(x_n)$ is expanded the algorithm will generate an R-successor y_1 with $\mathcal{L}(y_1) = \{(C \sqcap D), \neg C\}$. The concept expression $(C \sqcap D)$ will then be expanded and a clash will be detected because

$\{C, \neg C\} \subset \mathcal{L}(y_1)$. As neither C nor $\neg C$ in $\mathcal{L}(y_1)$ will have the branching points leading from x to x_n in their dependency sets, the algorithm can either return *unsatisfiable* immediately (if both the dependency sets were empty) or jump directly back to the most recent branching point on which one of C or $\neg C$ did depend. Figure 1 illustrates how the search tree below x is pruned by backjumping, with the number of R-successors explored being reduced by $2^n - 1$.

Heuristic techniques can be used to guide the search in a way which tries to minimise the size of the search tree. A method which is widely used in DPLL SAT algorithms is to branch on the disjunct which has the Maximum number of Occurrences in disjunctions of Minimum Size [8]. By choosing a disjunct which occurs frequently in small disjunctions, this heuristic tries to maximise the effect of BCP. For example, if the label of a node x contains the unexpanded disjunctions $\{C \sqcup D_1, \ldots, C \sqcup D_n\} \subseteq \mathcal{L}(x)$, then branching on C leads to their deterministic expansion in a single step: when C is added to $\mathcal{L}(x)$, all of the disjunctions are fully expanded and when $\neg C$ is added to $\mathcal{L}(x)$, BCP will expand all of the disjunctions. Branching first on any of $D_1, \ldots, D_n$, on the other hand, would only cause a single disjunction to be expanded.

Unfortunately this heuristic interacts adversely with the backjumping optimisation by overriding any "oldest first" order for choosing disjuncts: older disjuncts are those which resulted from earlier branching points and will thus lead to more effective pruning if a clash is discovered [11]. Moreover, the heuristic itself is of little value because it relies for its effectiveness on finding the same disjuncts recurring in multiple unexpanded disjunctions: this is likely in SAT problems, where the disjuncts are propositional variables, and where the number of different variables is usually small compared to the number of disjunctive clauses (otherwise problems would, in general, be trivially satisfiable); it is unlikely in concept satisfiability problems, where the disjuncts are concept expressions, and where the number of different concept expressions is usually large compared to the number of disjunctive clauses. As a result, the heuristic will often discover that all disjuncts have similar or equal priorities, and the guidance it provides is not particularly useful.

An alternative strategy is to employ a heuristic which tries to maximise the effectiveness of backjumping by using dependency sets to guide the expansion. Whenever a choice is presented, the heuristic chooses the concept whose dependency set includes the earliest branching points. This technique can be used both when selecting disjuncts and when ordering R-successors. The use of heuristics is an area of continuing research, but preliminary results suggest that the dependency heuristic is a promising technique.

During a satisfiability check there may be many successor nodes created. These nodes tend to look considerably alike, particularly as the R-successors for a node x each have the same concept expressions for the universal role concepts in $\mathcal{L}(x)$. Considerable time can thus be spent re-performing the computations on nodes that end up having the same label. As the satisfiability algorithm only cares whether a node is satisfiable or not, this time is wasted.

If successors are only created when other possibilities at a node are exhausted, then the entire set of concept expressions that come into a node label can be generated at one time. The satisfiability status of the node is then completely determined by this set of concept expressions. Then, if there exists another node with the same set of initial

formulae the two nodes will have the same satisfiability status [7]. Thus work need be done only on one of the two nodes, potentially saving a considerable amount of processing, as not only is the work at one of the nodes saved, but also the work at any of the successors of this node.

The downside of caching is that the dependency information required for backjumping cannot be effectively calculated for the nodes that are not expanded. This happens because the dependency set of any clash detected depend on the dependency sets of the incoming concept expressions, which will differ between the two nodes. Backjumping can still be performed, however, by combining the dependency sets of all incoming concept expressions and using that as the dependency set for the unsatisfiable node.

Another problem with caching is that it requires that nodes, or at least sets of formulae, be retained until the end of a satisfiability test, changing the storage requirements of the algorithm from polynomial to exponential in the worst case.

4 Testing

All the above optimisations are implemented in FaCT and DLP, and we have tested their efficacy on several test suites. (FaCT and DLP differ on their implementation details, how well they implement some of the above optimisations, and the exact heuristic optimisation they do.) All times reported are for runs on machines with approximately the speed of a SPARC Ultra 1.

We would prefer to test on actual description logic knowledge bases, as that is what FaCT and DLP are designed for. However, there are very few description logic knowledge bases that use the more-powerful constructs provided by FaCT and DLP. One test that we have been able to do is to take the GALEN knowledge base and construct versions of it that are acceptable to FaCT, DLP and KRIS, by, among other things, making all roles non-transitive and eliminating inclusion axioms. To illustrate the importance of backjumping, caching, and the heuristics, times are also given for DLP with these optimisations disabled—we will refer to this system as DLP*. FaCT and DLP processed the knowledge base in 210 seconds, classifying over two thousand concept definitions requiring tens of thousands of satisfiability tests. Both DLP* and KRIS were unable to complete the processing of the knowledge base in four hours.

Our other testing has been against test suites for propositional modal logics, using the propositional modal logic interface for FaCT and DLP. We have tested against the test suite for the Tableaux'98 propositional modal logic comparison [10] and against a collection of random formulae initially generated by Hustadt and Schmidt [12].

The Tableaux'98 test suite consists of several classes of formulae (e.g. *branch*), in both provable (*p*) and non-provable (*n*) forms, for each of $\mathbf{K}$, $\mathbf{KT}$, and $\mathbf{S4}$. For each type of formula, 21 examples of supposedly exponentially increasing difficulty are provided, and the result of a test is the number of the largest formula which the system was able to solve within 100 seconds of CPU time. The results of these tests with FaCT, DLP, DLP*, KSAT [4] and KRIS are summarised in Table 2. In the table, >20 indicates that

[4] The tests here used the original Lisp implementation of KSAT; a much faster C implementation is now available.

		FaCT		DLP		DLP*		KSAT		Kris	
		p	*n*	*p*	*n*	*p*	*n*	*p*	*n*	*p*	*n*
K	*branch*	6	4	18	12	10	11	8	8	3	3
	d4	>20	8	>20	>20	8	6	8	5	8	6
	dum	>20	>20	>20	>20	10	12	11	>20	15	>20
	grz	>20	>20	>20	>20	>20	>20	17	>20	13	>20
	lin	>20	>20	>20	>20	>20	>20	>20	3	6	9
	path	7	6	>20	>20	7	11	4	8	3	11
	ph	6	7	7	8	6	8	5	5	4	5
	poly	>20	>20	>20	>20	>20	>20	13	12	11	>20
	t4p	>20	>20	>20	>20	6	4	10	18	7	5
KT	*45*	>20	>20	>20	>20	9	>20	5	5	4	3
	branch	6	4	18	12	16	11	8	7	3	3
	dum	11	>20	>20	>20	9	>20	7	12	3	14
	grz	>20	>20	>20	>20	>20	>20	9	>20	0	5
	md	4	5	3	>20	3	>20	2	4	3	4
	path	5	3	8	8	2	>20	2	5	1	13
	ph	6	7	7	18	5	19	4	5	3	3
	poly	>20	7	>20	8	>20	2	1	2	2	2
	t4p	4	2	>20	>20	1	1	1	1	1	7
S4	*45*	>20	>20	>20	>20	>20	>20				
	branch	4	4	>20	12	16	12				
	grz	2	>20	>20	>20	0	>20				
	ipc	5	4	10	>20	3	10				
	md	8	4	3	>20	3	>20				
	path	2	1	6	>20	2	>20				
	ph	5	4	4	5	5	15				
	s5	>20	2	19	>20	1	>20				
	t4p	5	3	>20	>20	0	>20				

Table 2. Results for Tableaux'98 Benchmarks

the hardest problem was solved in less than 100 seconds. (Neither KSAT nor KRIS can reason with transitive roles, so they cannot be used to perform **S4** satisfiability tests.)

In these tests FaCT and DLP outperformed the other systems in this test, with DLP being a clear winner, because of its more-complete caching. Even DLP* performed better than other systems due to the optimizations retained in it. DLP also outperformed the other systems that took part in the the Tableaux'98 comparison [4].

Further analysis of the difference between DLP and DLP*, not presented here because of space limitations, shows that caching is more important than backjumping in these tests, which is more important than the heuristics. In fact the heuristics significantly degraded performance in some cases.

The optimisations in FaCT and DLP often resulted not simply in improved absolute performance but in a different qualitative behaviour. This is illustrated by Fig. 2 which shows the actual solution times for two types of formulae for DLP with backjumping and caching turned off and on. In one of these examples the qualitative improvement is

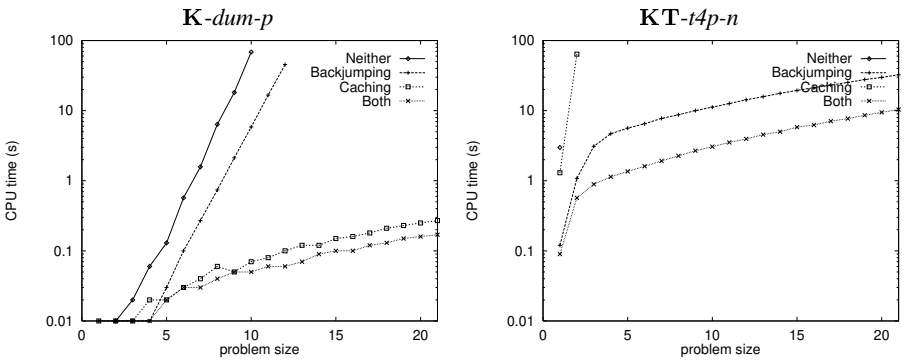


Fig. 2. Solution times for constructed satisfiability problems

due to caching (a common occurrence); in the other it is due to backjumping (a less-common occurrence).

Our second propositional modal logic test suite uses a method for testing SAT decision procedures that has been adapted for use with propositional modal **K** by Giunchiglia and Sebastiani [9], and further refined by Hustadt and Schmidt [12]. The method uses a random generator to produce formulae, with the characteristics of the formulae being controlled by a number of parameters. Each formula is a conjunction of L K-clauses, where a K-clause is a disjunction of K elements, each element being negated with a probability of 0.5. An element is either a modal atom of the form $\forall R.C$, where C is itself a K-clause, or at the maximum modal depth D, a propositional variable chosen from the N propositional variables which appear in the formula. Hustadt and Schmidt used two sets of formulae, denoted **PS12** and **PS13**, choosing $N = 4$ and $N = 6$ respectively, with $K = 3$ and $D = 1$ in both cases. The test sets are created by varying L from N to $30N$, giving formulae with a probability of satisfiability varying from $\approx$1 to $\approx$0, and generating 100 formulae for each integer value of L/N.

The median time required to test the satisfiability of the **PS12** and **PS13** formulae, with a limit of 1,000s per formula, using FaCT, DLP, DLP*, KSAT and KRIS are shown in Fig. 3. It can be seen that in these tests the performance differences between FaCT, DLP and KSAT are much less marked than was the case in the Tableaux'98 tests. This is because the purely propositional problems at depth 1 can always be solved deterministically, and so performance is dependent on the efficiency of propositional reasoning at depth 0. The optimisations which allowed FaCT and DLP to outperform KSAT, notably caching, are of little use with these formulae as there are no hard modal sub-problems.

Although the Tableaux'98 and random test suites show how our optimisations perform on propositional modal logics, neither is very good for our purposes. In particular, for the collection of random formulae most of the computational difficulties have to do with the initial non-modal component. In realistic KBs we expect to encounter problems where the hardness comes from the number of successors that have to be considered and their interaction with the non-modal component. The Tableaux'98 formulae have this form, but there are too few hard collections there to validate our optimisations, and the

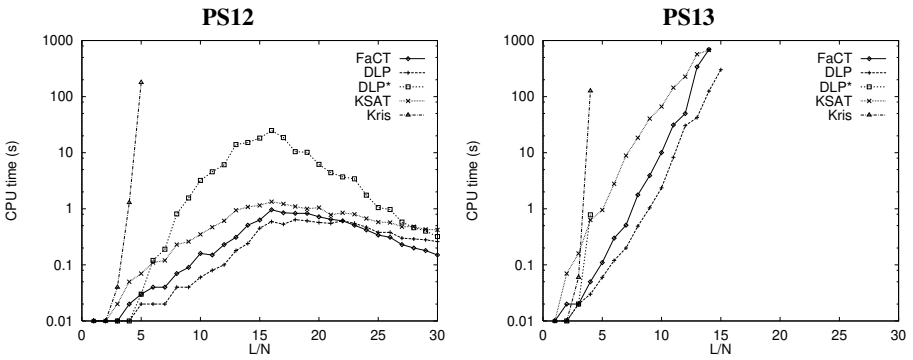


Fig. 3. Median solution times for randomly generated satisfiability problems

regular structure of the formulae tends to exaggerate the utility of the caching optimisation, particularly for satisfiable (non-provable) formulae.

5 Summary

The collection of optimizations we have described are effective in improving the speed of modal propositional logic reasoners, as shown by the results we have given above. They can also dramatically improve the speed of subsumption reasoning on description logic knowledge bases. To our knowledge some of these improvements have not been investigated in the modal propositional reasoning literature. The combination appears to be unique and, moreover, results in a powerful reasoner for the propositional modal logics $\mathbf{K}$, $\mathbf{KT}$, and $\mathbf{S4}$.

Unfortunately, the benefits of the various optimizations are not yet completely clear. Caching is best in some areas, backjumping in others. In order to better understand these effects, we continue to analyze and improve the optimisations we have incorporated into our provers. We also plan to create a test suite that emphasizes the modal nature of our description logic. Further, we are embarking on a project to create a description logic system for a description logic that corresponds to a propositional dynamic logic. This project will give us further opportunities to investigate optimisation of satisfiability reasoners.

References

1. 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. In B. Nebel, C. Rich, and W. Swartout, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference (KR'92)*, pages 270–281. Morgan-Kaufmann Publishers, San Francisco, CA, 1992. Also available as DFKI RR-93-03.

2. F. Baader and B. Hollunder. KRIS: Knowledge representation and inference system. *SIGART Bulletin*, 2(3):8–14, 1991.
3. A. B. Baker. *Intelligent Backtracking on Constraint Satisfaction Problems: Experimental and Theoretical Results*. PhD thesis, University of Oregon, 1995.
4. P. Balsiger and A. Heuerding. Comparison of theorem provers for modal logics — introduction and summary. In H. de Swart, editor, *Automated Reasoning with Analytic Tableaux and Related Methods: International Conference Tableaux'98*, number 1397 in Lecture Notes in Artificial Intelligence, pages 25–26. Springer-Verlag, May 1998.
5. P. Bresciani, E. Franconi, and S. Tessaris. Implementing and testing expressive description logics: a preliminary report. In Gerard Ellis, Robert A. Levinson, Andrew Fall, and Veronica Dahl, editors, *Knowledge Retrieval, Use and Storage for Efficiency: Proceedings of the First International KRUSE Symposium*, pages 28–39, 1995.
6. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.
7. F. Donini, G. De Giacomo, and F. Massacci. EXPTIME tableaux for $\mathcal{ALC}$. In L. Padgham, E. Franconi, M. Gehrke, D. L. McGuinness, and P. F. Patel-Schneider, editors, *Collected Papers from the International Description Logics Workshop (DL'96)*, number WS-96-05 in AAAI Technical Report, pages 107–110. AAAI Press, Menlo Park, California, 1996.
8. J. W. Freeman. Hard random 3-SAT problems and the Davis-Putnam procedure. *Artificial Intelligence*, 81:183–198, 1996.
9. F. Giunchiglia and R. Sebastiani. A SAT-based decision procedure for $\mathcal{ALC}$. In L. C. Aiello, J. Doyle, and S. Shapiro, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifth International Conference (KR'96)*, pages 304–314. Morgan Kaufmann Publishers, San Francisco, CA, November 1996.
10. A. Heuerding and S. Schwendimann. A benchmark method for the propositional modal logics k, kt, s4. Technical report IAM-96-015, University of Bern, Switzerland, October 1996.
11. I. Horrocks. *Optimising Tableaux Decision Procedures for Description Logics*. PhD thesis, University of Manchester, 1997.
12. U. Hustadt and R. A. Schmidt. On evaluating decision procedures for modal logic. Technical Report MPI-I-97-2-003, Max-Planck-Institut Für Informatik, Im Stadtwald, D 66123 Saarbrücken, Germany, February 1997.
13. F. Oppacher and E. Suen. HARP: A tableau-based theorem prover. *Journal of Automated Reasoning*, 4:69–100, 1988.
14. P. F. Patel-Schneider. System description: DLP. Bell Labs Research, Murray Hill, NJ, December 1997.
15. U. Sattler. A concept language extended with different kinds of transitive roles. In G. Görz and S. Hölldobler, editors, *20. Deutsche Jahrestagung für Künstliche Intelligenz*, number 1137 in Lecture Notes in Artificial Intelligence, pages 333–345. Springer Verlag, 1996.
16. K. Schild. A correspondence theory for terminological logics: Preliminary report. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI-91)*, pages 466–471, 1991.