Intelligent Tableau Algorithm for DL Reasoning

Ming Zuo and Volker Haarslev

Concordia University, Montreal QC H3G 1M8, Canada,
{ming_zuo|haarslev}@encs.concordia.ca

Abstract. Although state-of-the-art description logic (DL) reasoners are equipped with a comprehensive set of optimizations, reasoning performance is still a major bottleneck in both research and real world applications. In this paper, we propose a sound and complete algorithm called the intelligent tableau algorithm by incorporating comprehensive learning techniques to tackle all DL reasoning tasks. We also provide a reference implementation reasoner called LIGHT for the DL $\mathcal{ALC}$ dialect based on the algorithm we developed. Preliminary tests indicate that significant improvements can be achieved, i.e., compared to other state-of-the-art reasoners, LIGHT is up to two orders of magnitude faster for simple problems and several orders of magnitude faster for more difficult problems. Even though in this work our discussion is restricted to the $\mathcal{ALC}$ reasoning problem, our conjecture is that the algorithm developed can easily be extended to super-logics of $\mathcal{ALC}$.

Keywords: description logic, automated reasoning, learning, forgetting

1 Introduction

Most state-of-the-art DL reasoners implement tableau-based decision procedures which typically check the consistency of an ontology by constructing a so-called pre-model for the ontology. These procedures create pre-models in an often blind way which highly depends on the syntax of the input ontologies. Despite many optimization techniques studied and implemented so far, it is easy to find ontologies where one reasoner performs very well while the other is hopelessly inefficient (e.g., see [5] for combinations of nominals and qualified cardinality restrictions).

To simplify our discussion in the following sections, we restrict our research scope in this work on the DL dialect $\mathcal{ALC}$ (Attributive Concept Language with Complements) which is a subset of nearly every expressive DL [2]. Applications of the results introduced in this work to more expressive DL dialects will be addressed in our future work.

The paper is structured as follows. We first briefly introduce the syntax and semantics of $\mathcal{ALC}$ and its relationship with other logics. Then we study a sound and complete reasoning procedure based on a special DL normal form (DLNF). Afterward we discuss the integration of different types of learning into the reasoning procedure to come up with the so-called intelligent reasoning algorithm. At last, we show that any TBox can be converted into DLNF easily. The effectiveness of the algorithm proposed in this work is demonstrated by empirical results obtained from processing a number of typical test cases based on our reference implementation.

1.1 $\mathcal{ALC}$ Description Logic

Let A be a concept name (atomic concept), C and D are arbitrary concepts, and R a role name (atomic role). In $\mathcal{ALC}$, concepts are formed with the syntax as following:

$$C, D ::= \top \mid \bot \mid A \mid \neg C \mid C \sqcap D \mid C \sqcup D \mid \forall R.C \mid \exists R.C$$

where $\top$ is the abbreviation of $\neg A \sqcup A$, and $\bot$ of $\neg A \sqcap A$. An atomic concept corresponds to a unary relation in first order predicate logic (FOL), and an atomic role to a binary relation in FOL. If C and D are concepts, then $C \sqsubseteq D$ is a terminological axiom. $C \equiv D$ is the abbreviation of the two axioms $C \sqsubseteq D$ and $D \sqsubseteq C$. A finite set of terminological axioms is called a terminology or TBox. An *interpretation* $\mathcal{I} = (\Delta, \cdot^{\mathcal{I}})$ consists of a non-empty set Δ and a mapping function $\cdot^{\mathcal{I}}$. The function $\cdot^{\mathcal{I}}$ maps every role to a subset of $\Delta \times \Delta$ and every concept to a subset of Δ. If there exists an interpretation $\mathcal{I}$ which satisfies every axiom in $\mathcal{T}$, i.e., $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ must hold for every $C \sqsubseteq D$ in $\mathcal{T}$, we call the interpretation a *model* of $\mathcal{T}$. Hence $\mathcal{T}$ is called *satisfiable* if such a model exists, and *unsatisfiable* otherwise. $\mathcal{ALC}$ is a syntactic variant of the propositional modal logic $\mathrm{K}_{(\mathrm{m})}$ and can be seen as a fragment of FOL. Axioms in a TBox can be translated into FOL sentences correspondingly [2]. Let us use the function $\mathcal{F}()$ to represent such a translation. Therefore,

$$\begin{aligned}
&\mathcal{F}(A \sqsubseteq \exists R.C) = \forall x \exists y : A(x) \rightarrow (R(x,y) \wedge C(y))^{1}\\
&\mathcal{F}(A \sqsubseteq \forall R.C) = \forall x \forall y : A(x) \rightarrow (R(x,y) \rightarrow C(y))\\
&\mathcal{F}(A \sqsubseteq B) = \forall x : A(x) \rightarrow B(x) = \forall x : \neg A(x) \vee B(x)
\end{aligned}$$

In the following sections we shall only focus on the $\mathcal{ALC}$ TBox satisfiability reasoning problem for ease of illustration and evaluation. Solving other reasoning tasks is discussed in Section 6.

1.2 The contribution of this paper

In this paper we present a reasoning procedure which systematically and effectively uses a highly optimized DPLL algorithm for DL reasoning. This can be partially compared to SMT (Satisfiability Modulo Theory) based approaches but offers more advantages. In addition, we also propose a learning algorithm called unsat-learning which is proven to be very effective for reasoning optimization. Prior to our work, only approaches for unsat-caching [4, 3, 20] have been proposed, which can only prevent unsat-nodes (their number could be exponential) to be repeatedly expanded whereas our approach can prevent unsat-nodes to be repeatedly generated, which can reduce the search space exponentially in the best case. Moreover, we also integrate so-called *forgetting* techniques into our DL reasoning algorithm, which improve an algorithm's tractability in practice. At last, besides the standard negation normal form (NNF), a new DL normal form (DLNF) specifically for reasoning optimization is also investigated.

[1] In the following sections, we find it more appropriate to replace y by a skolem function $f(x)$.

2 An Intelligent TBox Reasoning Procedure

2.1 A reasoning procedure for DLNF TBoxes

Suppose we are given a TBox $\mathcal{T}$ in which the axioms can be divided into three sets $\mathcal{T}_g$, $\mathcal{T}_{ue}$, and $\mathcal{T}_{ua}$. In $\mathcal{T}_g$, each axiom is in the format $\top \sqsubseteq C$ where C is a disjunction of unary literals. In other words, if we ignore the "$\top \sqsubseteq$" part in $\mathcal{T}_g$ for all axioms, $\mathcal{T}_g$ can be considered in propositional logic conjunctive normal form (CNF). Let us call it PCNF to distinguish it from CNF in FOL that allows n-ary relations rather than only unary ones. In $\mathcal{T}_{ue}$ and $\mathcal{T}_{ua}$, all axioms are in the format such that $A \sqsubseteq \exists R.C$ and $A \sqsubseteq \forall R.C$ respectively, where A is a positive unary literal; R is an atomic role and C is a concept in the format of PCNF. In addition, all positive unary literals on the left-hand side of $\mathcal{T}_{ue}$ and $\mathcal{T}_{ua}$ are unique.

Definition 1. *A TBox $\mathcal{T}$ is in **Description Logic Normal Form (DLNF)** if all axioms in $\mathcal{T}$ can be divided into the three sets $\mathcal{T}_g$, $\mathcal{T}_{ue}$, and $\mathcal{T}_{ua}$ as described above.*

Example 1.

$$\begin{aligned}
\mathcal{T}^{'} &= \{\top \sqsubseteq \neg A_0 \sqcup \neg B_0,\ \ \neg A_0 \sqsubseteq B_0, \quad \forall R.(A_0 \sqcap \neg B_0) \sqsubseteq \neg A_0 \sqcup \forall R.\neg B_0\} \\
\mathcal{T}^{''} &= \{\top \sqsubseteq \neg A_0 \sqcup \neg B_0,\ \ \top \sqsubseteq A_0 \sqcup B_0,\ \ \top \sqsubseteq \neg A_0 \sqcup A_1 \sqcup A_2, \\
&\qquad A_1 \sqsubseteq \forall R.\neg B_0, \qquad A_2 \sqsubseteq \exists R.(\neg A_0 \sqcup B_0)\}
\end{aligned}$$

In Example 1, $\mathcal{T}^{'}$ is not in DLNF since the last two axioms do not match the definition of any of the three sets while $\mathcal{T}^{''}$ is in DLNF. If a TBox $\mathcal{T}$ is in DLNF, then it can be easily translated to sets of skolemized FOL sentences (as shown below), where $f(x)$ is a skolemization function which maps instances to instances; x and y are variables of instances of the underlying domain:

$$\begin{aligned}
\mathcal{F}(\mathcal{T}_g) &= \forall x \bigwedge_{i=1}^{m} \bigvee_{j=1}^{n} \alpha_{ij}(x) \\
\mathcal{F}(\mathcal{T}_{ua}) &= \forall x \forall y \bigwedge_{l=1}^{q} \gamma_l(x) \to (s_l(x,y) \to d_l(y)) \\
\mathcal{F}(\mathcal{T}_{ue}) &= \forall x \bigwedge_{k=1}^{p} \delta_k(x) \to (r_k(x, f_k(x)) \wedge c_k(f_k(x)))
\end{aligned} \tag{1}$$

Therefore, in Example 1, we have the FOL translation $\mathcal{F}(\mathcal{T}^{''})$ as follows:

$$\begin{aligned}
\mathcal{F}(\mathcal{T}^{''}_g) &= \forall x : (\neg A_0 \vee \neg B_0,\ \ A_0 \vee B_0,\ \ \neg A_0 \vee A_1 \vee A_2)(x) \\
\mathcal{F}(\mathcal{T}^{''}_{ua}) &= \forall x \forall y : A_1(x) \to (R(x,y) \to \neg B_0(y)) \\
\mathcal{F}(\mathcal{T}^{''}_{ue}) &= \forall x : A_2(x) \to (R(x, f(x)), (\neg A_0 \vee B_0)(f(x)))
\end{aligned}$$

where $(\neg A_0 \vee \neg B_0)(x)$ is the abbreviation of $(\neg A_0(x) \vee \neg B_0(x))$, and we replace the symbol "$\wedge$" with "," to emphasize the fit of set data structure during implementation.

Let us assume a TBox $\mathcal{T}$ is satisfiable and $\mathcal{I}^m = (\Delta, \cdot^{\mathcal{I}^m})$ is a model of $\mathcal{T}$. Therefore, Δ must be non-empty, and it should at least contain one instance. If the existence of such an instance is impossible, i.e, it is impossible to construct a mapping function $\cdot^{\mathcal{I}^m}$ to satisfy all axioms in $\mathcal{T}$, then $\mathcal{T}$ must be unsatisfiable. Without loss of generality, let us say i_0 is such an instance in Δ. We call the problem space w.r.t. only one single instance a *node*. We also use instance names to identify nodes if this does not cause any confusion. The problem space w.r.t. i_0 is called *root node* or *root*. Let us take $\mathcal{T}''$ from Example 1 to illustrate how we can solve the satisfiability problem for a TBox in DLNF effectively. As we already stated, the underlying idea of solving the satisfiability problem of $\mathcal{T}''$ is to prove that the existence of i_0 is possible.

Step (i) *Construct the root node from* $\mathcal{T}$*.*

The logical semantics of the root node is that w.r.t. i_0 all axioms in $\mathcal{T}$ map to the logical true under a mapping function. If we can prove that such a mapping function exists, $\mathcal{T}$ is satisfiable. Otherwise, $\mathcal{T}$ is unsatisfiable. Now let us consider the root node of $\mathcal{T}''$ which is the instantiation of x with i_0 in $\mathcal{F}(\mathcal{T}'')$ (for unary relations, we use A instead $A(i_0)$ for simplification purpose). Then, we get the root node of $\mathcal{T}''$:

1) $\{\neg A_0 \vee \neg B_0, A_0 \vee B_0, \neg A_0 \vee A_1 \vee A_2\}$
2) $A_1 \rightarrow \forall y : (R(i_0, y) \rightarrow \neg B_0(y))$
3) $A_2 \rightarrow \{R(i_0, f(i_0)), (\neg A_0 \vee B_0)(f(i_0))\}$

Mapping 3) and 2) to the logical true can be easily achieved if we include $(\neg A_2(i_0) : true)$ and $(\neg A_1(i_0) : true)$ in our mapping function. We call such kind of sentences *rules*. To differentiate the two types of rules, we call the sentences instantiated from $\mathcal{F}(\mathcal{T}_{ue})$ $_{\exists}$*-rules* and the ones from $\mathcal{F}(\mathcal{T}_{ua})$ $_{\forall}$*-rules*. As for 1), finding a mapping to satisfy all sentences in it is equivalent to finding a propositional model for the corresponding CNF. If there does not exist such a model, it means the mapping function $\cdot^{\mathcal{I}^m}$ can not be constructed, hence $\mathcal{T}''$ must be unsatisfiable. Let us assume there exists a propositional model for the CNF of the root node. We call the propositional model of the CNF inside a node a *path* of the node. We call an individual element in the path such as $\neg A_2(i_0)$, $A_1(i_1)$ an *item*.

Step (ii) *Find a path of the corresponding node.*

Provided that the path of the root node does exist, let us consider a specific item in the path. We have only three possibilities:

(a) The item matches to an item on the left-hand side of a $_{\forall}$-rule.
(b) The item matches to an item on the left-hand side of an $_{\exists}$-rule.
(c) The item does not match to any item on the left-hand side of any rule.

As for (c), the item occurring in the path has no impact on the sentences instantiated from $\mathcal{F}(\mathcal{T}_{ua})$ and $\mathcal{F}(\mathcal{T}_{ue})$. If all items in the path fall into this category, satisfiability of the underlying TBox is directly proven. For instance, in the above example, $\{\neg A_0, B_0, \neg A_1, \neg A_2\}$ is a path of the root node. However, none of the elements in it matches the elements on the left-hand side of 3) and 2). Therefore, $\mathcal{T}''$ is satisfiable, and one can easily construct a model with a complete mapping function for it. As for the other options (a) and (b), rule expansions are required.

Step (iii) *Perform rule expansions.*

If a $_\forall$-rule is triggered, the right-hand side of the rule needs to be added to the corresponding node, i.e., adding the sentence $\forall y : s_l(i, y) \rightarrow d_l(y)$ to node i. Similarly, if an $_\exists$-rule is triggered, a relation $r_k(i, f_k(i))$ needs to be added to $\cdot^{\mathcal{I}^m}$. Without loss of generality, let us use a new instance name i_1, which does not exist in Δ, to replace $f_k(i)$ instead of keeping the function name.[2] Therefore, we add a new instance i_1 to Δ and a binary relation $r_k(i, i_1)$ to $\cdot^{\mathcal{I}^m}$. Thus a new node w.r.t. i_1 is added to the search space and c_k is added to the new node as part of the rule.

A node which generates new nodes is called a *predecessor*, and the generated nodes its *successors*. The corresponding role r_k is called an *edge*. If a PCNF c_k is added to a node because a $_\exists$-rule is triggered by some δ_k, then δ_k is called an $_\exists$-prefix and c_k is called an $_\exists$-label of the node. Similarly, if a d_l is added to a node due to a triggered $_\forall$-rule by a γ_l, then γ_l is called a $_\forall$-prefix and d_l a $_\forall$-label of the node. The set of prefixes of a node is called a *prefix set* and the set of labels a *label set* of the node. We call two nodes *equivalent* if they contain the same label set. If there is no conflict detected in any of its successor nodes, then the node is called *satisfiable*.

To illustrate how it works, let us still use $\mathcal{T}''$ from Example 1. Let us assume that the path we found for the root node is $\{\neg A_0, B_0, A_1, A_2\}$ this time.[3] A_1 triggers a $_\forall$-rule, therefore we add a special rule $\forall y : R(i_0, y) \rightarrow \neg B_0(y)$ to the root node. A_2 triggers an $_\exists$-rule. Therefore, $\Delta = \Delta \cup \{i_1\}$ where i_1 is a new name and $\cdot^{\mathcal{I}^m} = \cdot^{\mathcal{I}^m}$ $\cup\{R(i_0, i_1)\}$. Furthermore, we also create a new node w.r.t. i_1. $\neg A_0 \vee B_0$ is added to the newly created node as part of the $_\exists$-rule. Since we have $R(i_0, i_1)$ in $\cdot^{\mathcal{I}^m}$, the rule $\forall y : R(i_0, y) \rightarrow \neg B_0(y)$ in the node i_0 is triggered in which we instantiate y with i_1. Therefore, we have the following node i_1 which contains all instantiated sentences (rules) of $\mathcal{F}(\mathcal{T}'')$ (due to the $\forall x$ restriction) by replacing x with i_1 together with $\neg B_0$ ($_\forall$-label) and $\neg A_0 \vee B_0$ ($_\exists$-label). We separate the label set from the instantiated PCNF below just for illustration purposes.

1) $\{\neg A_0 \vee \neg B_0, A_0 \vee B_0, \neg A_0 \vee A_1 \vee A_2\}, \{\neg \mathbf{B_0}, \neg \mathbf{A_0} \vee \mathbf{B_0}\}$
2) $A_1 \rightarrow \forall y : (R(i_1, y) \rightarrow \neg B_0(y))$
3) $A_2 \rightarrow \{R(i_1, f(i_1)), (\neg A_0 \vee B_0)(f(i_1))\}$

It is obvious that there does not exist a proposition model for the PCNF in 1). It means that the prefix set $\{A_1, A_2\}$ of node i_1 (which is part of the path of node i_0) leads to a contradiction. In such kind of situation, the path found in the predecessor node needs to be rolled back and recalculated. This rollback and recalculation procedure is applied recursively until no valid path can be found for the root node (the corresponding TBox is unsatisfiable) or all paths have been found for all nodes in the problem space (the corresponding TBox is satisfiable).

Step (iv) *Apply steps (ii) to (iii) recursively until either all paths have been found for all nodes or no path could be found for the root node.*

Proposition 1. *The reasoning procedure from step (i) to step (iv) is sound and complete.*

[2] Refer to [2] for details on the open world assumption in DLs.

[3] We chose a complete propositional model for illustration purposes here. A partial model containing only $\{\neg A_0, B_0\}$ is already sufficient for a satisfiability proof.

Proof. The proposition holds because a TBox $\mathcal{T}$ can straightforwardly be translated to FOL to which Herbrand's theorem applies. The above-mentioned procedure is exactly a procedure for constructing a herbrand model [10].

2.2 Integrating learning into reasoning

The underlying idea of learning w.r.t. logic reasoning is to prune unvisited search space based on the knowledge achieved from previous search steps. With the pruned search space, reasoning algorithms are supposed to find search results faster. By considering the size of the underlying search space regarding to $\mathcal{ALC}$, which can be exponential w.r.t. the size of input ontologies [2], it is almost certain that effective learning should improve the average reasoning performance significantly.

As discussed in Section 2.1, finding a path for a node is reduced to finding a propositional model for the underlying PCNF of the node. Therefore, some proven to be very effective optimization algorithms such as DPLL equipped with conflict-driven learning and back-jumping that are also employed by state-of-the-art SAT solvers [22, 15] can be directly used. We call such kind of learning inside a single node *local learning*. The discussion and improvements of *local learning* are beyond the scope of this paper and we shall focus on *global learning*, i.e., the kind of learning that affects the reasoning search space on the pre-model level, which directly affects the number of nodes to be searched. To be more specific, global learning can be categorized into three types:

1. Unsat-learning: If the status of a node has already been determined as unsatisfiable, the algorithm should learn from it and block all related unexplored search space that would definitely lead to a failed search result.
2. Sat-learning: If the status of a node has already been determined as satisfiable, the algorithm should directly mark the status of its equivalent nodes as satisfiable without performing reasoning or expansion on them.
3. Unknown-learning: When the algorithm starts, the status of visited nodes are first marked as "unknown" meaning that the sat/unsat status has not yet been determined. As the model graph is expanded during reasoning, if a newly created node is equivalent to a node already marked as "unknown", then we should avoid duplicate reasoning on the latter. In this case, we mark the latter node as "blocked" meaning its satisfiability should refer to another node. The previously visited node with an unknown status is called a *blocker*, and the blocked node is called a *blockee*.

Let us first check how an algorithm can learn from an unsat node. Without loss of generality, let us suppose the label set of an unsat node is $\{c_m, d_1, d_2, \ldots, d_n\}$.[4] Correspondingly, the prefix set is $\{\delta_m, \gamma_1, \gamma_2, \ldots, \gamma_n\}$. If the node is marked as unsat, it means that the combination of all its prefixes leads to a conflict w.r.t. $\mathcal{T}$. That is

$$\mathcal{T} \models \exists x : \delta_m(x) \wedge \gamma_1(x) \wedge \gamma_2(x) \wedge \cdots \wedge \gamma_n(x) \rightarrow \bot \qquad (2)$$

It is equivalent to:

$$\mathcal{T} \models \forall x : \neg\delta_m(x) \vee \neg\gamma_1(x) \vee \neg\gamma_2(x) \vee \cdots \vee \neg\gamma_n(x) \qquad (3)$$

[4] For $\mathcal{ALC}$, a non-root node contains exactly one existential label.

We call the right-hand side of axiom (3) a *learned sentence*. Let us add the learned sentence to $\mathcal{F}(\mathcal{T}_g)$ and populate it to all nodes with a still unknown status. We can easily prove that all nodes whose prefix set contains the set $\{\delta_m, \gamma_1, \gamma_2, \ldots, \gamma_n\}$ are pruned from the search space. To illustrate how unsat-learning works, let us again consider the example we used in Section 2.1. Node i_1 is unsat and its prefix set is $\{A_1, A_2\}$. Then the learned sentence is $\forall x : (\neg A_1 \vee \neg A_2)(x)$. First we rollback the path found for the root node and the corresponding elements related to that path which were added/created during reasoning. The second step is to add the learned sentence to $\mathcal{T}$ and populate it to the nodes with an unknown status. In our example, after the second step we get:

$$\mathcal{F}(\mathcal{T}_g^{''}) = \forall x : \{\neg A_0 \vee \neg B_0, A_0 \vee B_0, \neg A_0 \vee A_1 \vee A_2, \neg \mathbf{A_1} \vee \neg \mathbf{A_2}\}(x)$$

The changed root node:

1) $\{\neg A_0 \vee \neg B_0, A_0 \vee B_0, \neg A_0 \vee A_1 \vee A_2, \neg \mathbf{A_1} \vee \neg \mathbf{A_2}\}$
2) $A_1 \rightarrow \forall y : (R(i_0, y) \rightarrow \neg B_0(y))$
3) $A_2 \rightarrow \{R(i_0, f(i_0)), (\neg A_0 \vee B_0)(f(i_0))\}$

Now if we recalculate the path of the root node or any other node in our problem space in future search, any supersets containing $\{A_1, A_2\}$ will be automatically excluded. Unsat-learning does not affect the soundness or completeness of the algorithm and the proof is trivial.

As for sat-learning and unknown-learning, these techniques are also called sat-caching and blocking in other papers [4, 3]. They can be simply implemented as buffering, i.e., all nodes in the underlying search space are identified by their labels, and if a newly created node has a buffer hit, it can be directly marked either as 'sat' or 'blocked' without further expansion.

However, naive buffering may cause the algorithm to become unsound [8]. Solutions to fix the unsoundness are discussed in [3, 6]. The solution introduced in [6] is widely considered to be the best so far. However, it requires EXPSpace in the worst case to construct a pre-model which can easily cause the reasoning algorithm to become intractable.

Considering the procedure we discussed so far, unknown-learning is not a source of unsoundness since the status of both blockers and blockees is restricted only to "unknown" whereas sat-learning may cause unsoundness due to the problematic definition of "satisfiable" in Section 2.1. For example, in the case where all successors are "blocked" by some other "unknown" nodes, the reasoning algorithm normally marks the predecessor as "sat" due to *no conflict detected in any of its successor nodes*. In our case, such a node will be saved in the sat-buffer and might be reused by others afterwards due to a buffer hit. This is the source for the unsoundness w.r.t. sat-learning since any of the related blockers could be proven as "unsat" afterwards. Therefore, in DL we characterize node satisfiability as *relative* to blockers compared to *absolute* node unsatisfiability. In fact, there is a simple solution to ensure soundness. All we need to do is to remove the nodes from the sat-buffer which directly or indirectly depend on a blocker node whenever such a node is detected as "unsat".

Algorithm 1 Normalization to CNF

```
1: function NORMALIZE(axiom)
2:     remove $\equiv$ and non-top concept from lefthand side of $\sqsubseteq$ from axiom
3:     convert axiom to NNF
4:     if axiom matches $\top \sqsubseteq C \sqcup (D \sqcap E)$ then
5:         normalize($\top \sqsubseteq \neg\eta \sqcup D$)                ▷ $\eta$ is a new name
6:         normalize($\top \sqsubseteq \neg\eta \sqcup E$)
7:         normalize($\top \sqsubseteq C \sqcup \eta$)
```

Algorithm 2 Remove value restrictions

```
function REMOVEROLEITEM(aDLCNFClause)
    for all concept in aDLCNFClause do
        if concept matches $\exists R.C$ then
            replace concept with $\delta$                ▷ $\delta$ is a new name
            $\mathcal{T}_{ue}$.add($\delta \sqsubseteq$ concept)
        else if concept matches $\forall R.D$ then
            replace concept with $\gamma$                ▷ $\gamma$ is a new name
            $\mathcal{T}_{ua}$.add($\gamma \sqsubseteq$ concept)
```

3 Intelligent Tableau Algorithm

3.1 Normalization

Reasoning algorithms used by state-of-the-art DL reasoners usually require axioms of the input TBox to be transformed into NNF which can be done easily in linear time. However, in the above mentioned reasoning procedure we require the input TBox to be in DLNF. Therefore, before performing reasoning on a TBox, we need an algorithm to convert an arbitrary TBox into the format of DLNF which is called *normalization*.

We divide the normalization into two steps: The first step is to remove all conjunctions from all axioms in the target TBox (see Algorithm 1). In the second step, we remove all concepts with value restrictions from the resulting CNF and then add corresponding axioms to $\mathcal{T}_{ua}$ and $\mathcal{T}_{ue}$ (see Algorithm 2). Concepts C and D in Algorithm 2 can also be reduced to PCNF easily in a similar way. One can easily tell that in Example 1, $\mathcal{T}^{''}$ is the normalization result of $\mathcal{T}^{'}$.

Based on Algorithms 1 and 2, we have Proposition 2.

Proposition 2. *There exists an algorithm that converts an arbitrary TBox $\mathcal{T}$ to $\mathcal{T}'$ in polynomial time where $\mathcal{T}'$ is in DLNF and $\mathcal{T}'$ is equisatisfiable to $\mathcal{T}$.*

Proof. It is obvious that these algorithms require polynomial time. We only need to prove that $\mathcal{T}'$ is equisatisfiable to $\mathcal{T}$ after normalization. The conversion from line 4 to line 7 in Algorithm 1 is widely used in converting an arbitrary SAT problem into a 3-SAT problem, and the equisatisfiability proof needs not to be repeated here. When converting to DLNF, the only difference is the introduction of $_\exists$-rules and $_\forall$-rules. In fact, the equisatisfiability of such a conversion can be proven in exactly the same way. As for the PCNF conversion of role fillers, the proof can be easily done based on the fact that equisatisfiability is closed under conjunction.

3.2 Intelligent reasoning algorithm

As we already mentioned in previous sections, the algorithm for finding a path inside a specific node is implemented as finding a propositional model w.r.t. a CNF. A DPLL procedure with local learning is described in Algorithm 3.

Algorithm 3 findModel

```
 1: function FINDMODEL(aPCNF)
 2:     while $true$ do
 3:         while ¬unsat and ¬finish do
 4:             unfold()
 5:             propagateAndDeduce()
 6:         if unsat then
 7:             if currentLevel = 0 then
 8:                 return $false$
 9:             else
10:                 resolveConflict()
11:         else if ¬ decideNextBranch() then
12:             return $true$
```

Algorithm 4 $\forall$- and $\exists$-unfold

```
function ∀-UNFOLD(aPositiveUnaryLiteral)
    rule ← createLocalRule(aPositiveUnaryLiteral)
    for all successor ∈ successorList do
        if successor.role = rule.role then
            successor.addPrefix(rule.prefix)
            successor.cnf.add(rule.filler)
function ∃-UNFOLD(aPositiveUnaryLiteral)
    node ← createNewNode(aPositiveUnaryLiteral)
    for all rule ∈ getLocalRule(node.role) do
        node.cnf.add(rule.filler)
        node.prefix.add(rule.prefix)
    successorList.add(node)
```

Compared to SAT reasoning, we have to consider the $\exists$-rule and $\forall$-rules in DL reasoning. The function unfold() for rule expansions is called at line 4 in Algorithm 3. It is executed whenever a propositional model item has been added. For other functions, the details are similar to what is described in [22] except that the function resolveConflict() needs to additionally deal with unsat caused by global learning and the rollback() needs to do the opposite of unfold() if the rolling back item is unfoldable. $\exists$-unfold and $\forall$-unfold are described in Algorithm 4.

A recursive depth-first search (DFS) algorithm to determine satisfiability of an input TBox is shown in Algorithm 5. At line 3, the algorithm checks and updates the local CNF from the results of global learning, if applicable. This can avoid a global propagation when global learning results are applicable that may affect system performance at runtime. At line 15 we ensure the soundness of sat-learning, if applicable. Line 16 can be as simple as adding a disjunction of negated prefixes to $\mathcal{T}_g$ as described in Section 2.2.

Algorithm 5 satCheck

```
 1: external satBuffer, unknownBuffer
 2: function SATCHECK(aNode)
 3:     updateCNFFromGlobalLearning()
 4:     if ¬ findModel(aNode.pcnf) then
 5:         return $false$
 6:     for all successor ∈ successorList do
 7:         if successor ∈ satBuffer then
 8:             successor.status ← SAT
 9:         else if successor ∈ unknownBuffer then
10:             successor.status ← BLOCKED
11:         else
12:             unknownBuffer.add(successor)
13:             if ¬ satCheck(successor) then
14:                 unknownBuffer.remove(successor)
15:                 ensureSATLearningSoundness()
16:                 Learn from successor.prefix
17:                 return satCheck(aNode)
18:     if current ≠root then
19:         unknownBuffer.remove(current)
20:         satBuffer.add(current)
21:     return $true$
```

3.3 Forgetting

In Algorithm 5, a pre-model is constructed through DFS. Therefore, during reasoning we only need to keep one single branch in memory to construct the pre-model. Such kind of algorithms can be implemented in PSPACE as further studied and proved in [16]. As a result, many tableau-based decision procedures for DL reasoning can be considered as overall "practically tractable". With the presence of global buffers, worst case optimal algorithms using DFS are also studied and presented in [4] in which the analysis of "practically tractable" algorithms by using global buffers focus only on the space and time required for the construction of the pre-model whereas the space used by the global buffers is ignored. As a matter of fact, we can easily prove that, if no proper action is taken, the size of global buffers, even though only unsat-caching is involved, can be exponential with regard to the size of the input TBox. Therefore, the "practically tractable" feature might no longer hold in the presence of global buffers.

By considering the algorithm we proposed in this work, without special treatment, the size of learning buffers for both global and local learning can also be EXP size in the worst case. To achieve tractability, an intuitive solution is to remove less useful learned knowledge from learning buffers, which can be seen as an opposite operation to learning, and it is normally called *forgetting*. In our work, forgetting is applicable to local learning, unsat-learning, and sat-learning but not to unknown-learning. This scheme is good enough for practical reasoning since through DFS the number of nodes with an unknown status is normally small. A forgetting algorithm could be as simple as using a size-restricted FIFO queue, and it could be as complicated as some advanced heuristic algorithms.

4 Related Work

We named the algorithm developed in this work as "intelligent tableau" to emphasize its relationship to the traditional tableau algorithms [2] in that both algorithms construct tree-like pre-models for $\mathcal{ALC}$ reasoning. One can easily prove that both are variants for finding a herbrand model to tackle DL reasoning problems. Learning and forgetting can be considered as optimization techniques, which could also possibly be integrated into traditional DL tableau algorithms. The major difference to our work is on how to deal with disjunctions, i.e., a tightly integrated DPLL vs. standard tableau branching.

Traditional tableau algorithms are implemented by almost all state-of-the-art reasoners such as FaCT++ [21], Pellet [19], RacerPro [9] and HermiT [18]. These algorithms are widely blamed for a low efficiency in the presence of many general inclusion axioms or disjunctions [12]. Even though equipped with many optimization techniques such as boolean constraint propagation (BCP), semantic branching, back-jumping, etc., most DL reasoners still easily become intractable when dealing with ontologies containing many disjunctions. Even though the JNH test cases we used in Section 5 are considered as trivial examples for a SAT solver, no state-of-the-art DL reasoner is able to provide an efficient solution. As for more complicated CNF test cases, these reasoners easily become intractable based on our test results. Modified versions of tableau algorithms such as hypertableaux [14], which uses hyper-resolution instead of simple tableau branching, are developed and applied in reasoners such as HermiT. Even though the performance in dealing with disjunctions is improved, based on our test results, HermiT is normally performing worse than others in situations where a big amount of nodes needs to be constructed in the underlying pre-model (see Section 5).

Researchers also presented approaches for DL reasoning through SMT [17], which make it possible to solve DL reasoning problems by using efficient state-of-the-art SAT solvers. Some of the underlying ideas coincidentally overlap with our work. However, even though the SMT solutions have achieved a similar performance for some benchmark test cases compared to state-of-the-art DL reasoners, they did not provide effective ways to prune the underlying search space. In addition, the encoding algorithms used to reduce DL problems to SAT problems are still as hard as EXPTime which may cause some significant overhead when considering performance in real-world applications. Moreover, the black-box consideration of the SAT portion might cause unnecessary search if a conflict could be easily detected before a full SAT model is constructed. So far, we were unable to include a practical SMT reasoner in our comparison tests.

The research on EXPTime Tableau for $\mathcal{ALC}$ [4] (by applying global caching) has been further developed and implemented by [6, 7]. These kinds of algorithms either heavily use subset checking or require EXPSpace to construct the pre-model. Both cases can easily cause intractability in real world applications. Implementations of such kind of reasoning algorithms are still far from building a practical reasoner for real world DL applications.

5 Empirical Results

The primary goal of the algorithm developed in Section 3.2 is to conduct “fast” reasoning — the purpose of DPLL based algorithm is to improve the reasoning performance w.r.t. to a single node while comprehensive learning is to reduce the number of nodes to be searched. A good way to verify whether our goal has been achieved is through running typical benchmark test cases. In addition, designing an enable/disable switch on some specific optimization feature is the best way to verify its effectiveness. Based on such motivation, we provide a reference implementation called LIGHT in which sat-learning and unsat-learning can be switched on and off. We consider the features of our reasoning procedure such as being DPLL-based, employing DLNF ontology normalization and unknown-learning as so fundamental that they are tightly integrated into our architecture and therefore can not be disabled. We conducted our benchmark tests using four different settings of LIGHT (see Table 1): (i) both sat-learning and unsat-learning switched off (L-N); (ii) only sat-learning switched on (L-S); (iii) only unsat-learning switched on (L-U); (iv) both sat-learning and unsat-learning switched on (LIGHT). Part of the test results for the employed $\mathcal{ALC}$ benchmark test cases are shown in Table 1. The LIGHT reasoner for different platforms together with all test cases we used, complete test results and test scripts are available for download.[5]

All test results in Table 1 are based on a Ubuntu Linux 12.04 32 bit platform. The used hardware is a DELL Precison 390 with Intel Core 2 Duo processor 2.4G equipped with 4GB memory. For Java based reasoners, we used Oracle JDK v7.0.11. In Table 1, all runtimes are given in seconds. The word “cr” means the system crashed (out of memory or segment fault) during the test, and “to” means the system was aborted after a timeout ($\geq$ 2000 seconds). The suffix “s” and “u” of ontology names represents the corresponding TBox that is either satisfiable or unsatisfiable.

The JNH [11] test cases are CNF benchmarks converted to OWL syntax and are used to test the capability of DL reasoners for dealing with ontologies containing many (global) disjunctions. BCS (Basic Call System) [1] test cases are real-world examples and typical in the sense that large amount of nodes are required to construct a pre-model. GALEN test cases are used to evaluate the reasoners when dealing with simple problems. The test cases named “k_XX” are taken from Tableaux’98 [13].

As shown in Table 1, in some situations where very limited number of branches is required to build a pre-model or conflicts can be easily detected, sat and unsat learning

[5] `http://www.lightreasoner.co.nf/`

Table 1. Benchmark results for $\mathcal{ALC}$ test cases (runtimes in seconds)

	L-N	L-S	L-U	LIGHT	HermiT	Pellet	Fact++	Racer
galen1s	0.12	0.12	0.14	0.12	1.2	1.3	0.44	1.7
galen2s	0.16	0.15	0.16	0.15	1.3	1.4	0.46	1.9
JNH15u	0.02	0.02	0.02	0.02	6.2	119.1	94.7	119.5
JNH16u	0.07	0.06	0.06	0.07	237.4	452.3	cr	15384
JNH17u	0.02	0.02	0.02	0.02	1.6	21.1	9.5	1165
k_d4_12nu	to	to	44.32	44.47	to	to	1054	to
k_d4_13nu	to	to	99.77	98.70	to	to	to	to
k_dum_18nu	18.04	15.05	17.33	13.99	to	to	cr	196.29
k_dum_19nu	to	to	37.59	32.27	to	to	cr	140.88
k_ph_14pu	963.7	1001	1005	1014	cr	cr	cr	to
k_tp4_21nu	15.84	15.31	5.32	0.32	to	0.54	cr	to
k_branch_20nu	0.34	0.34	0.35	0.35	to	2.3	14.7	16.1
k_branch_21nu	0.39	0.40	0.40	0.39	to	2.4	18.2	19.2
k_path_20pu	1.5	1.53	0.19	1.53	cr	21.03	5.85	7.88
k_path_21pu	1.7	1.76	0.23	1.78	cr	25.63	7.30	9.0
k_poly_15pu	18.36	18.3	18.0	0.43	179.27	27.67	34.97	1.62
k_poly_16pu	cr	cr	cr	0.61	373.4	76.98	cr	2.03
k_poly_20ns	cr	cr	to	236.7	cr	cr	cr	149.9
k_poly_21ns	cr	cr	to	325.4	cr	cr	cr	524.6
BCS3s	to	0.03	to	0.02	1.6	20.7	cr	0.69
BCS4s	to	1.37	to	0.20	133.8	to	cr	13.8
BCS5s	to	to	to	2.14	cr	to	cr	276.2

have no significant impact on the results for GALEN and JNH. In these situations, the effectiveness of LIGHT's reasoning compared to other reasoners can be primarily attributed to the optimized DPLL algorithm used. Learning may also have negative impact in some situations such as K_PH_14P. In some cases, with only unsat learning enabled we can achieve better results than using the combination of the two, i.e., sat-learning only causes overhead for a test case such as K_PATH_20P. In many situations, unsat-learning is critical for obtaining a good performance. However, in the BCS test cases, we also see that sat-learning plays a critical role to ensure effective reasoning. Overall, the combination of both sat and unsat learning achieves very good results in most of the cases.

Compared to other DL reasoners, LIGHT is up to one order of magnitude faster for the GALEN test cases. For the BCS benchmarks, LIGHT is two orders of magnitude faster than Racer, which is the only reasoner besides LIGHT that can process all three variants. The overall performance of LIGHT is significantly improved for the benchmarks selected from Tableaux'98 (the test cases with prefix "k_"). The JNH benchmark results demonstrate the effectiveness of LIGHT by using an optimized DPLL algorithm in dealing with situations where one has only one node in the pre-model that has many disjunctions while the other reasoners are several orders of magnitude slower than LIGHT.

6 Discussion

The TBox satisfiability problem we discussed in this work can be seen as a special case of the ($\top$) concept satisfiability problem w.r.t. a non-empty TBox. From this perspective, once an algorithm solves the TBox satisfiability problem, all other DL reasoning tasks such as concept satisfiability, classification, concept subsumption, ABox satisfiability etc. can also be solved easily by using exactly the same algorithm [2].

Some may consider the normalization algorithm we presented in this work that introduces additional variables to be a source of inefficiency. After all, the worst case complexity analysis even for DPLL based algorithms is tightly related to the number of variables involved. As a matter of fact, this kind of concern is unjustified. First of all, the normalization algorithm requires only polynomial (linear) time which is normally trivial compared to the EXPtime reasoning algorithm. Another fact is that no proof or test results indicate that the introduction of variables can significantly affect the reasoning performance. Our test results have shown that the introduction of new variables such as the conversion from 5CNF to 3CNF in SAT reasoning in most of the cases interestingly improved the reasoning performance.

At last, one may wonder the necessity of the algorithm we proposed in this work. After all, the algorithm proposed in this work can be easily reduced to finding a Herbrand model in FOL which is also the case for traditional tableau algorithms. In fact, one can easily reduce the algorithm proposed in this work to traditional tableau-based algorithms for further analysis such as computational complexity and termination analysis. From our perspective, the major benefits of the algorithm proposed in this work are based on two points. First, this algorithm helps us reduce a DL-based problem to a SAT based problem so that we can delegate efficient reasoning by using proven to be efficient algorithms. The other reason is that we simplified the pre-model structure from an AND-OR graph [6] to an AND-only graph, i.e., all nodes in the discourse have to be satisfiable to make the corresponding TBox satisfiable. The "OR" portion in the graph with its reasoning algorithm is completely merged to the "AND" node. Thus, with the simplified model structure, it is easier to develop and integrate more efficient optimization algorithms such as comprehensive learning.

7 Conclusion and Future Work

In this paper we presented an efficient reasoning algorithm that incorporates learning for solving the TBox satisfiability problem. It is based on searching herbrand models, which is related to but also different from traditional DL tableau algorithms. Preliminary test results have shown that our presented algorithms are significantly more efficient than other existing ones. Besides a systematical discussion of learning on DL reasoning, our DLNF normalization form has been systematically presented and investigated, which makes it easier to incorporate effective optimization techniques into automated reasoning algorithms due to the structured format. Even though the discussion in this work is restricted to the DL $\mathcal{ALC}$, our conjecture is that the algorithm can be applied to super-logics of $\mathcal{ALC}$ or even other DL related logics with slight modifications which will be presented in our future work.

References

1. C. Areces, W. Bouma, and M. de Rijke. Description logics and feature interaction. In *Proceedings of the International Workshop on Description Logics (DL'99)*, pages 28–32, Linköping, Sweden, 1999.
2. F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook*. Cambridge University Press, 2nd edition, 2007.
3. Y. Ding and V. Haarslev. Tableau caching for description logics with inverse and transitive roles. In *Proceedings of the 2006 International Workshop on Description Logics (DL-2006)*, pages 143–149, 2006.
4. F. M. Donini and F. Massacci. EXPTIME tableaux for $\mathcal{ALC}$. *Artificial Intelligence*, 124(1):87–138, 2000.
5. J. Faddoul. *Reasoning algebraically with description logics*. PhD thesis, Department of Computer Science and Engineering, Concordia University, 2011.
6. R. Goré and L. Nguyen. Exptime tableaux for $\mathcal{ALC}$ using sound global caching. *Journal of Automated Reasoning*, pages 1–27, 2011.
7. R. Goré and L. Postniece. An experimental evaluation of global caching for $\mathcal{ALC}$ (system description). In *Proceedings of IJCAR 2008*, volume 5195, pages 299–305. Automated Reasoning, 2008.
8. V. Haarslev and R. Möller. Consistency testing: The RACE experience. In *Proceedings of International Conference on Tableaux, St Andrews, Scotland, July 4-7*, pages 57–61. Springer-Verlag, 2000.
9. V. Haarslev and R. Möller. Racer system description. In *Proceedings of International Joint Conference on Automated Reasoning*, pages 701–705. Springer-Verlag, 2001.
10. J. Herbrand. *Recherches sur la théorie de la démonstration*. PhD thesis, University of Paris, 1930.
11. J. Hooker. Satlib - benchmark problems. Website, 2011. `http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html`.
12. I. Horrocks. Using an expressive description logic: FaCT or fiction? In *Proc. of KR'1998*, pages 636–647.
13. I. Horrocks and P. Patel-Schneider. DL systems comparison. In *Proc. of the 1998 Description Logic Workshop (DL'98)*, pages 55–57. volume 11 of CEUR, 1998.
14. B. Motik, R. Shearer, and I. Horrocks. Hypertableau reasoning for description logics. *Journal of Artificial Intelligence Research*, 36:165–228, 2009.
15. L. O. Ryan. Efficient algorithms for clause learning SAT solvers. Master's thesis, Simon Fraser University, BC, Canada, 2004.
16. M. Schmidt-Schauß and G. Smolka. Attributive concept descriptions with complements. *Artificial Intelligence*, 1(48):1–26, 1991.
17. R. Sebastiani and M. Vescovi. Automated reasoning in modal and description logics via SAT encoding: the case study of K(m)/ALC-Satisfiability. *Journal of Artificial Intelligence Research*, 35, 2009.
18. R. Shearer, B. Motik, and I. Horrocks. Hermit: A highly efficient OWL reasoner. *In 5th OWL Experiences and Directions Workshop*, 2008.
19. E. Sirin, B. Parsia, B. Cuenca Grau, A. Kalyanpur, and Y. Katz. Pellet: A practical OWL-DL reasoner. *Journal of Web Semantics*, 5(2):51–53, 2007.
20. A. Steigmiller, T. Liebig, and B. Glimm. Extended caching, backjumping and merging for expressive description logics. In *Proc. of IJCAR'2012*, pages 514–529.
21. D. Tsarkov and I. Horrocks. FaCT++ description logic reasoner: System description. In *Third International Joint Conference on Automated Reasoning*, pages 292–297, 2006.
22. L. Zhang. *Searching for truth: techniques for satisfiability of boolean formulas*. PhD thesis, Departement of Electrical Engineering, Princeton University, 2003.