Optimizing Tableau Reasoning: a Prolog-based Framework

Riccardo Zese a,b,* and Giuseppe Cota a

* Dipartimento di Ingegneria, Università di Ferrara, Via Saragat 1, 44122, Ferrara, Italy
E-mails: riccardo.zese@unife.it, giuseppe.cota@unife.it

Abstract. One of the foremost reasoning services for knowledge bases is finding all the justifications for a query. This is useful for debugging purpose and for coping with uncertainty. Among Description Logics (DLs) reasoners, the tableau algorithm is one of the most used. However, in order to collect the justifications, the reasoners must manage the non-determinism of the tableau method. For these reasons, a Prolog implementation can facilitate the management of such non-determinism.

The TRILL framework contains three probabilistic reasoners written in Prolog: TRILL, TRILL P and TORNADO. Each one of them uses different approaches for probabilistic inference and handles different DLs flavours. Our previous work showed that they can achieve sometimes better results than state-of-the-art (non-)probabilistic reasoners.

In this paper we present two optimizations that improve the performances of the TRILL reasoners. In the first one, the reasoners build the hierarchy of the concepts contained in a knowledge base in order to quickly find connections among them during the expansion of the tableau. The second one modifies the order of application of tableau rules in order to reduce the number of operations. Experimental results show the effectiveness of the introduced optimizations.

All systems can be tried online in the TRILL on SWISH web application at http://trill-sw.eu/.

Keywords: Reasoner, Axiom Pinpointing, Tableau Algorithm, (Probabilistic) Description Logic, Prolog

1. Introduction

The aim of the Semantic Web is to make information available in a form that is understandable and automatically manageable by machines. In order to realize this vision, the W3C has supported the development of a family of knowledge representation formalisms of increasing complexity for defining ontologies, called OWL (Web Ontology Languages), that are based on Description Logics (DLs). Many inference systems, generally called reasoners, have been proposed to reason upon these ontologies, such as Pellet [1], HermiT [2] and FaCT++ [3].

Nonetheless, modelling real-world domains requires dealing with information that is uncertain. Therefore many semantics for combining probability theory with OWL languages, or with the underlying DLs, were conceived [4–8]. Among them, in [9, 10] the authors proposed a semantics for probabilistic DLs, called DISPONTE. This semantics borrows the distribution semantics [11] from Probabilistic Logic Programming, that has emerged as one of the most effective approaches for representing probabilistic information in Logic Programming languages.

Probabilistic systems that can perform inference under DISPONTE are BUNDLE [12, 13] and the TRILL framework. The first one is implemented in Java and it can exploit several non-probabilistic reasoners, the latter is a framework, written in Prolog, which contains
three reasoners: (i) TRILL [10, 14], able to collect the set of all the justifications and compute the probability of queries, (ii) TRILL$^P$ [10, 14], which implements in Prolog the tableau algorithm defined in [15, 16] for returning the pinpointing formula instead of the set of justifications, and (iii) TORNADO [17], which is similar to TRILL$^P$ but represents the pinpointing formula in a way that can be directly used to compute the probability of the query.

Most DL reasoners adopt the tableau algorithm [18, 19]. This algorithm applies some expansion rules on a tableau, a representation of the assertional part of the KB. In the first one, some of these rules are non-deterministic, requiring the implementation of a search strategy in an or-branching search space. The reasoners contained in the TRILL framework exploit Prolog’s backtracking facilities for performing the search. In addition, the experiments performed in [17] showed that a Prolog implementation of the tableau algorithm can achieve competitive or even better results than other state-of-the-art (non-)probabilistic reasoners.

In this paper we present two optimizations that improve the reasoning performances of the TRILL framework. In the first one, the reasoners build the hierarchy of the concepts contained in a knowledge base in order to quickly find connections among them during the expansion of the tableau. The second one modifies the application order of tableau rules in order to reduce the number of operations.

All the probabilistic reasoners in the TRILL framework are available in the TRILL on SWISH web application [20] at http://trill-sw.eu/.

Moreover, we present an extensive experimental evaluation where we compared the previous version of the TRILL framework with the latest one containing the optimizations. The experimental results show that, when the KB is relatively large and complex, the introduced extensions can significantly speed up both regular and probabilistic queries.

The paper is organized as follows: Section 2 illustrates the needed background by briefly introducing DLs, describing the tableau algorithm and presenting the probabilistic semantics DISPONTE, which motivates the need of finding all the justifications and thus the need of build highly optimized systems. Section 3 presents the former TRILL framework, followed by a detailed description of the introduced optimizations in Section 4. Section 5 discusses related work. Finally, Section 6 shows the experimental evaluation and Section 7 concludes the paper.

2. Background

2.1. Description Logics

Description Logics (DLs) [21, 22] syntax is based on individuals, representing objects of the domain, concepts, which group individuals sharing the same characteristics, and roles, connecting pairs of individuals or individuals with datatype values (i.e. integers, strings, etc.). There are many DL languages that differ by the constructs that are allowed for defining concepts and roles. We first briefly describe the DL SHI and then its extension SHIQ.

Let us consider a set of atomic concepts $C$, a set of atomic roles $R$ and a set of individuals $I$. A role could be an atomic role $R \in R$ or the inverse $R^\sim$ of an atomic role $R \in R$. We use $R^\sim$ to denote the set of all inverses of roles in $R$. Each $A \in A$, Top (also called $\top$ or $\top$) and Bottom (also called $\bot$ or $\bot$) are concepts. If $C$, $C_1$ and $C_2$ are concepts and $R \in R \cup R^\sim$, then $(C_1 \cap C_2)$, $(C_1 \cup C_2)$ and $\neg C$ are concepts, as well as $\exists R.C$ and $\forall R.C$.

A knowledge base (KB) $K = (T, R, A)$ consists of a TBox $T$, an RBox $R$ and an ABox $A$. An RBox $R$ is a finite set of transitivity axioms $\text{Trans}(R)$ and role inclusion axioms $R \sqsubseteq S$, where $R, S \in R \cup R^\sim$. A TBox $T$ is a finite set of concept inclusion axioms $C \sqsubseteq D$, where $C$ and $D$ are concepts. An ABox $A$ is a finite set of concept membership axioms $a : C$ and role membership axioms $(a, b) : R$, where $C$ is a concept, $R \in R$ and $a, b \in I$. With respect SHI, the DL SHIQ adds new constructs for the definition of qualified number restrictions, i.e., given a concept $C$ and $R \in R \cup R^\sim$, qualified number restrictions are concepts of the form $\geq n R.C$ and $\leq n R.C$. A knowledge base is usually assigned a semantics in terms of interpretations $I = (\Delta^I, \cdot^I)$, where $\Delta^I$ is a non-empty domain and $\cdot^I$ is the interpretation function, which assigns an element in $\Delta^I$ to each $a \in I$, a subset of $\Delta^I$ to each concept and a subset of $\Delta^I \times \Delta^I$ to each role (we do not consider datatype roles). A query $Q$ over a KB $K$ is usually an axiom for which we want to test the entailment from the KB, written as $K \models Q$.

Example 1. The following KB is inspired by the ontology people+pets [23]:

$$
\exists \text{hasAnimal.Pet} \sqsubseteq \text{NatureLover}
$$

$\text{Cat} \sqsubseteq \text{Pet}$

$\text{fluffy} : \text{Cat}$ (kevin,fluffy) : hasAnimal

$\text{tom} : \text{Cat}$ (kevin,tom) : hasAnimal
It states that individuals that own an animal which is a pet are nature lovers and that kevin owns the animals fluffy and tom, which are cats. Moreover, cats are pets. The KB entails the query $Q = \text{kevin : NatureLover}$. 

### 2.2. The Tableau Algorithm

In this section we discuss the tableau algorithm [21], one of the most common approaches to answer queries. In its basic definition, a tableau is an ABox represented using a tuple $G = (V, E, \mathcal{L})$ that contains a directed graph $(V, E)$ where each node of $V$ corresponds to an individual $a$ and is labelled with the set of concepts $\mathcal{L}(a)$ to which $a$ belongs. Each edge in $(a, b) \in E$ in the graph is labelled with the set of roles $\mathcal{L}((a, b))$. The binary predicate $\neq$ is used to specify inequalities between nodes. $G$ is initialized with a node for each individual $a$ of the KB, labelled with all concepts $C$ such that $a : C \in K$, and an edge $e = (a, b)$ labelled with $R$ for each assertion $(a, b) : R \in K$.

A tableau algorithm proves an axiom by refutation, starting from a tableau that contains the negation of the axiom. Then, the tableau algorithm repeatedly applies a set of consistency preserving tableau expansion rules until a clash (i.e., a contradiction) is detected or a clash-free graph is found to which no more rules are applicable. If no clashes are found, the tableau represents a model for the negation of the query, which is thus not entailed.

The tableau expansion rules can be deterministic or non-deterministic. The first type takes as input one graph and returns a single updated graph. In order to manage non-deterministic rules, a set $T$ of completion graphs is built instead of a single one. $T$ is initialized with a single completion graph $G_0$, and in the case of application of a non-deterministic rule, the tableau $G_i$ on which the rule is applied is replaced by the set of tableaux returned by the rule.

For ensuring the termination of the algorithm, a special condition known as blocking [24] is used.

**Soundness and completeness of the tableau algorithm** are proved in [21].

#### 2.2.1. Representing a Covering Set of Justification

An important problem to solve is finding the covering set of justifications for a given query. This non-standard reasoning service is also known as axiom pinpointing [25] and it is useful for tracing derivations and debugging ontologies. This problem has been investigated by various authors [24–27].

The covering set of justifications can be represented by means of either minimal axiom sets, which basically correspond with justifications, or a pinpointing formula.

**Definition 1** (Explanation). Given a KB $K$ and a query $Q$, a subset of logical axioms $E$ of a KB $K$ such that $E \models Q$ is called explanation.

**Definition 2** (Justification). A justification is an explanation such that it is minimal w.r.t. set inclusion. Formally, we say that an explanation $J \subseteq K$ is a justification if for all $J' \subseteq J$, $J' \neq Q$, i.e. $J'$ is not an explanation for $Q$.

**Definition 3** (Covering set of justifications). The set of all the justifications for the query $Q$ is the covering set of justifications for $Q$. Given a KB $K$, the covering set of justifications for $Q$ is denoted by $\text{ALL-JUST}(Q, K)$.

The covering set of justifications can be represented also by means of a pinpointing formula, as presented in [15, 16]. This formula is built using Boolean variables (one for each axiom of the KB) and the conjunction and disjunction connectives. Let’s assume that each axiom $E$ of a KB $K$ is associated with a propositional variable, indicated with $\text{var}(E)$. The set of all propositional variables is indicated with $\text{var}(K)$. A valuation $\nu$ of a monotone Boolean formula is the set of propositional variables that are true. For a valuation $\nu \subseteq \text{var}(K)$, let $K_\nu := \{ t \in K | \text{var}(t) \in \nu \}$.

**Definition 4** (Pinpointing formula). Given a query $Q$ and a KB $K$, a monotone Boolean formula $\phi$ over $\text{var}(K)$ is called a pinpointing formula for $Q$ if for every valuation $\nu \subseteq \text{var}(K)$ it holds that $K_\nu \models Q$ iff $\nu$ satisfies $\phi$.

The pinpointing formula compactly encodes the set $\text{ALL-JUST}(Q, K)$ as proved in Lemma 2.4 of [16]. This approach is correct and terminating for the DL $\text{SHIQ}$.

#### 2.2.2. Extending the Tableau to Solve the Axiom Pinpointing Problem

In order to solve the axiom pinpointing problem, the tableau algorithm has been modified so that each expansion rule updates as well a tracing function $\tau$, which associates labels of nodes and edges with justifications [28] or the pinpointing formula [16] found so far for each label.

During the initialization of the tableau, $\tau$ is initialized as empty for all the elements of its domain except for $\tau(C, a)$ and $\tau(R, (a, b))$, to which the values $\{a : C\}$ and $\{(a, b) : R\}$ are assigned if $a : C$ and $(a, b) : R$ are in the ABox. The tableau expansion rules for $\text{SHIQ}$
are shown in Figure 1, where the rules for the $SHI$ DL are marked by ($\ast$). Here, $Add(C,a)$ stands for the addition of a concept $C$ to $C(a)$ while $Add(R,(a,b))$ represents the addition of a role $R$ to $(a,b)$.

The values of the tracing function associated with the labels which causes the clash is then put together to form the covering set of justification.

The rules in Figure 1 are divided into deterministic and non-deterministic. As stated above, the first, when applied to a tableau, produce a single new tableau. The latter, when applied to a tableau, produce a set of tableaux.

Unfortunately, classical tableau algorithm returns a single justification (or a Boolean formula not representing all the justifications) using the tracing function. To solve the axiom pinpointing problem, researchers have to explore the entire search space of the possible explanations. Classical tableau-based systems, implemented using imperative languages, forces the tableau algorithm to find a new justification in several ways. The most used approach, called Hitting Set Tree (HST) [29], repeatedly removes axioms from the KB following the justifications previously found and executes the tableau algorithm to find new justifications w.r.t. the modified KB. For instance, given a KB $K$ and a query $Q$, if the justification $J = \{E_1, E_2, E_3\}$ was found, where $E_i$s are axioms, to avoid the generation of the same justification, the HST algorithm tries to find a new justification on $K' = K \setminus E_1$. If no new justification is found the HST algorithm backtracks and tries to find another justification by removing other axioms from $J$, one at a time.

**Example 2.** Consider the KB shown in Example 1. We associate Boolean variables with axioms as follows:

\[ E_1 = \exists hasAnimal.Pet \sqsubseteq NatureLover \]
\[ E_2 = \text{fluffy} : \text{Cat} \]
\[ E_3 = \text{tom} : \text{Cat} \]
\[ E_4 = \text{Cat} \sqsubseteq \text{Pet} \]
\[ E_5 = (\text{kevin,fluffy}) : \text{hasAnimal} \]
\[ E_6 = (\text{kevin,tom}) : \text{hasAnimal} \]

Let $Q = \text{kevin} : \text{NatureLover}$ be the query, then

\[ \text{ALL-JUST}(Q,K) = \{(E_5, E_2, E_1), \{E_6, E_3, E_4, E_1\}\} \]

while the pinpointing formula is

\[ ((E_5 \land E_2) \lor (E_6 \land E_3)) \land E_4 \land E_1 \]

A fruitful approach to avoid implementing such an algorithm is to rely on the backtracking facilities that are built-in in Prolog. This idea have been explored by many researchers that, for example, implemented the tableau using Prolog [30–34]. Another possibility is to apply an extension of (disjunctive) ASP [35] or an abductive proof procedure to perform ontological reasoning [36]. We proposed three systems, TRILL [10, 14], TRILL$^P$ [10, 14], and TORNADO [17], that implements the tableau algorithm in Prolog. These reasoners will be briefly presented in Section 3.

### 2.3. Probabilistic Description Logics

Axiom pinpointing problem is also important for probabilistic inference. In the following we briefly describe the DISPONTE semantics [9], which requires the set of all the justifications to compute the probability of queries.

DISPONTE [9, 10] applies the distribution semantics [11] to Probabilistic Description Logic KBs. In DISPONTE, a probabilistic knowledge base $K$ contains a set of probabilistic axioms which take the form

\[ p :: E \]  

where $p$ is a real number in $[0,1]$ and $E$ is a DL axiom. The probability $p$ can be interpreted as the degree of our belief in the truth of axiom $E$.

Following the semantics, DISPONTE associates independent Boolean random variables to the DL axioms. The set of axioms that have the random variable assigned to 1 constitutes a world, with a probability computed by multiplying the probability $p_i$ for each probabilistic axiom $E_i$ included in the world by the probability $1 - p_i$ for each probabilistic axiom $E_i$ not included in the world. The probability of the query is the sum of the probabilities of the worlds where the query is true.

The number of different worlds is exponential in the number of probabilistic axioms, so their enumeration is unfeasible. A possible approach for computing the probability of queries to KBs is to compute $\text{ALL-JUST}(Q,K)$, assign independent Boolean random variables to the axioms contained in the justifications and define the DNF Boolean formula $f_K$ as $f_K(X) = \bigvee_{i \in K} \bigwedge_{E_i \in E_{i,0}} X_i$, where $X_i = \{X_i \mid E_i \in K, K \in K\}$ is the set of Boolean random variables.

As we have seen, the set of all the justifications can be represented also by means of a pinpointing for-
Deterministic rules:
→ unfold (*): If A ∈ L(a), A atomic and (A ⊆ D) ∈ K, then
  if D ∈ L(a), then
    Add(D, a)
    τ(D, a) := (τ(A, a) ∪ {A ⊆ D})
→ CE (*): If (C ⊆ D) ∈ K, with C not atomic, a not blocked, then
  if (¬C ∪ D) /∈ L(a), then
    Add((¬C ∪ D), a)
    τ((¬C ∪ D), a) := {C ⊆ D}
→ ¬(C1 ∩ C2) ∈ L(a), a is not indirectly blocked, then
  if (C1, C2) ⊆ L(a), then
    Add((C1, C2), a)
    τ(C1, a) := τ(C1 ∩ C2), a)
→ ∃ (a): If ∃S.C ∈ L(a), a is not blocked, then
  if a has no S-neighbour b with C ∈ L(b), then
    create new node b, Add(S, (a, b)), Add(C, b)
    τ(C, b) := τ(∃S.C), (a)
    τ(S, (a, b)) := τ(∃S.C, a)
→ ∀ (∗): If ∀S.C ∈ L(a), a is not indirectly blocked and
  there is an S-neighbour b of a, then
  if C ∈ L(b), then
    Add(C, b)
    τ(C, b) := τ(∀S.C, a) ∪ τ(S, (a, b))
→ ∀ (∗): If ∀S.C ∈ L(a), a is not indirectly blocked and
  there is an R-neighbour b of a, Trans(R) and R ⊆ S, then
  if ∃R.C ∈ L(b), then
    Add(∃R.C, b)
    τ(∃R.C, b) := τ(∀S.C, a) ∪ τ(R, (a, b)) ∪ {Trans(R)} ∪ {R ⊆ S}
→ ≥ (a): If ∃nS.C ∈ L(a), a is not blocked, then
  if there are no n safe S-neighbours b1, ..., bn of a with b1 ≠ bj
    with C in L(bi) for each bi, then
    create n new nodes b1, ..., bn, Add(S, (a, bi)), Add(C, bi), (b1, bj)
    τ(S, (a, bi)) := τ(∃nS.C, a)
    τ(C, bi) := τ(∃nS.C, a) ∪ τ(S, (a, bi))
    τ(¬(b1, bi)) := τ(∃nS.C, a)

Non-deterministic rules:
→ ∪ (∗): If (C1 ∪ C2) /∈ L(a), a is not indirectly blocked, then
  if (C1, C2) ∩ L(a) = ∅, then
    Generate graphs G1 := G for each i ∈ {1, 2}
    Add(Ci, a) in Gi for each i ∈ {1, 2}
    τ(Ci, a) := τ(C1 ∩ C2), a)
→ ≤ (a): If (nS.C) /∈ L(a), a is not indirectly blocked, then
  and there are m S-neighbours b1, ..., bm of a with m > n
    with C in L(bi) for each bi, then
  For each possible pair bi, bj, 1 ≤ i, j ≤ m; i ≠ j with C ∈ L(bi) ∩ L(bj) then
    Generate a graph G'
    τ(Merge(bi, bj)) := τ(∃nS.C, a) ∪ τ(S, (a, bi)) ... ∪ τ(S, (a, bm))
→ if bi is an ancestor of bj, then Merge(bi, bj) in G'
  else Merge(bi, bj) in G'
→ if bj is merged into bi, then for each concept Ci in L(bi),
    τ(Ci, bj) := τ(Ci, bi) ∪ τ(Merge(bi, bj))
    (similarly for roles merged, and correspondingly for concepts in bj)
    if merged into bi)

Fig. 1. Tableau expansion rules for SHIQ; the subset of rules marked by (∗) is employed for SHI.
mula. Irrespective of which representation of the explainations we choose, a DNF or a general pinpointing formula, we can apply knowledge compilation and transform it into a Binary Decision Diagram (BDD), from which we can compute the probability of the query with a dynamic programming algorithm that is linear in the size of the BDD. A BDD for a function of Boolean variables is a rooted graph that has one level for each Boolean variable. A node has two children corresponding respectively to the 1 value and the 0 value of the variable associated with the level of n. When drawing BDDs, the 0-branch is distinguished from the 1-branch by drawing it with a dashed line. The leaves store either 0 or 1.

3. The TRILL Framework

The TRILL framework contains a web interface called TRILL on SWISH [20] and three inference systems, a.k.a. reasoners, called TRILL [10, 14], TRILL$^p$ [10, 14], and TORNADO [17], which use the Prolog language to implement the tableau algorithm and computing the probability of the queries under DISPONTE semantics. For the sake of clarity, in the following, if we need to consider all the three system together we will refer to “TRILL framework” or “TRILL systems”. While if we consider the single reasoners we will use TRILL, TRILL$^p$, and TORNADO.

TRILL solves the ALL-JUST($Q, K$) problem for the SHI$^Q$ DL, it can return the set of all the justifications and the probability of the queries computed by means of knowledge compilation using BDDs. TRILL$^p$ modifies TRILL in order to compute the pinpointing formula instead of the set of all the justifications. It can also compute the probability of the query by representing the formula with a BDD. It is implemented in Prolog and supports the SHI$^Q$ DL. Analogously, TORNADO builds the pinpointing formula but, differently from TRILL$^p$, it is directly represented as a BDD, avoiding some exponential blow-ups in the inference process. As TRILL$^p$, it is implemented in Prolog and supports the SHI$^Q$ DL. The TRILL framework is implemented in SWI-Prolog [37], the code of all three systems is available at https://github.com/rzese/trill.

The TRILL framework forms a layer cake, shown in Figure 2, designed to facilitate its extension. The lower layer, called “Translation Utilities”, contains a library for translating the input KB in case it is given in the RDF/XML format and loading it in the Prolog database, in order to be accessible to the upper layers. This layer contains the module utility_translation which is based on the Thea$^2$ library [38] for converting OWL DL KBs into Prolog. Thea$^2$ performs a direct translation of OWL axioms into Prolog facts. Then, there is the “TRILL Library” layer containing system-specific modules, one per system. Each module contains predicates that act differently in the three systems and settings that are specific to each reasoner or that do not share the same values. Finally, the upper layer contains predicates and settings that are in common for all the systems and defines the user interface and the queries that can be asked. In particular, TRILL, TRILL$^p$, and TORNADO can answer concept and role membership queries, subsumption queries and can test the unsatisfiability of a concept of the KB or the inconsistency of the entire KB. They can be executed by means of a SWI-Prolog console or tested online with the TRILL on SWISH web application at http://trill-sw.eu/, whose interface is shown in Figure 3.

In order to represent the tableau, a pair $\text{Tableau} = (\Lambda, T)$ is used, where $\Lambda$ is a list containing information about individuals and class assertions with the corresponding value of the tracing function. The tracing function stores a fragment of the KB in TRILL, the pinpointing formula in TRILL$^p$, and the BDD representing the pinpointing formula in TORNADO. $T$ contains the structure of the tableau and information needed during its expansion. For detailed descriptions of the implementation of the three systems we refer to the papers which presented them. In the following we recap the implementation of the tableau in order to bet-
Determine and non-deterministic tableau expansion rules are implemented following a different interface: non-deterministic rules follow the interface rule

\[ \text{rule\_name} (\text{Tab}, \text{TabList}) \]

\(\) they take as input the current tableau \(\text{Tab}\) and return the list of tableaux \(\text{TabList}\) created by the application of the rule to \(\text{Tab}\). Figure 4 shows the code of the non-deterministic rule \(\rightarrow \sqcup\). The predicate or\_rule/2 searches in the tableau \((A,T)\) for an individual to which the rule can be applied and unifies \(L\) with the list of new tableaux created by scan\_or\_list/6. find\_Class\_Assertion/4 implements the search for a class assertion in \(A\). modify\_ABox/5 checks if the class assertion axiom with the associated explanation is already present in \(A_0\), and in this case it checks the applicability of the expansion rule. If the rule can be applied it updates the list of assertions \(A_0\) creating \(A\). Calling modify\_ABox/5 avoids infinite loops in the rule application.

Deterministic rules are defined following the interface rule\_name(\(\text{Tab}, \text{Tab1}\)) that, given the current tableau \(\text{Tab}\), returns the tableau \(\text{Tab1}\) obtained by the application of the rule to \(\text{Tab}\). Figure 5 shows part of the code of the deterministic rule \(\rightarrow \text{unfold}\). The predicate unfold\_rule/2 searches in \((A,T)\) for an individual to which the rule can be applied and calls the predicate find\_sub\_sup\_class/3 in order to

```
or_rule((A,T),L):-
  findClassAssertion(unionOf(LC),
                   Ind,Expl,A),
  \+ indirectly_blocked(Ind,(A,T)),
  findall((A1,T),
     scan_or_list(LC,Ind,Expl,A,A1),L),
  dif(L,[]),!.
```

```
scan_or_list([],_Ind,_Expl,A,A).
scan_or_list([C|_T],Ind,Expl,A0,A):-
  modify_ABox(A0,C,Ind,Expl,A).
scan_or_list([_C|T],Ind,Expl,A0,A):-
  scan_or_list(T,Ind,Expl,A0,A).
```

Fig. 4. Code of the \(\rightarrow \sqcup\) rule. It unifies the list \(L\) with all the tableaux resulting by the application of the rule. \text{scan\_or\_list}/6 simply checks if the concept \(C\) can be added with the explanation \(\text{Expl}\) to the list or not.

```
  or_rule((A,T),L):-
   findClassAssertion(unionOf(LC),
                      Ind,Expl,A),
   \+ indirectly_blocked(Ind,(A,T)),
   findall((A1,T),
          scan_or_list(LC,Ind,Expl,A,A1),L),
   dif(L,[]),!.
```

```
scan_or_list([],_Ind,_Expl,A,A).
scan_or_list([C|_T],Ind,Expl,A0,A):-
  modify_ABox(A0,C,Ind,Expl,A).
scan_or_list([_C|T],Ind,Expl,A0,A):-
  scan_or_list(T,Ind,Expl,A0,A).
```

Fig. 4. Code of the \(\rightarrow \text{unfold}\) rule. It unifies the list \(L\) with all the tableaux resulting by the application of the rule. \text{scan\_or\_list}/6 simply checks if the concept \(C\) can be added with the explanation \(\text{Expl}\) to the list or not.
unfold_rule((A0,T),(A,T)):-
  findClassAssertion(C,Ind,Expl,A0),
  atomic(C),
  find_sub_sup_class(C,D,Ax),
  modify_ABox(A0,C,Ind,Expl,A).

find_sub_sup_class(C,D, subClassOf(C,D)):-
  subClassOf(C,D).

find_sub_sup_class(C,D, equivalentClasses(L)):-
  equivalentClasses(L),
  member(C,L),
  member(D,L),
  dif(C,D).

Fig. 5. Code of the \rightarrow unfold \text{ rule}. It takes an atomic class \( C \) from the input tableau and looks for a class \( D \) which is a super-class or an equivalent class of \( C \). It builds the explanation for the new class assertion found and updates the tableau where needed.

current tableau and returns a tableau obtained by the application of one of the rules. It is called as \text{apply_det_rules}(\text{RuleList}, \text{Tab}, \text{OutTab}). After the application of a deterministic rule, a cut avoids backtracking to other possible choices for the deterministic rules.

Then, non-deterministic rules are tried sequentially with the predicate \text{apply_nondet_rules}/3 (its implementation is shown in Figure 6) that is called as \text{apply_nondet_rules}(\text{RuleList}, \text{Tab}, \text{OutTab}). It takes as input the list of non-deterministic rules and the current tableau and returns a tableau obtained with the application of one of the rules. If a non-deterministic rule is applicable, the list of tableaux obtained by its application is returned by the predicate \text{corresponding to the applied rule, a cut is performed to avoid backtracking to other rule choices and a tableau from the list is non-deterministically chosen with the member/2 predicate. If no rule is applicable, the input tableau is returned and the rule application stops, otherwise a new round of rule application is performed. Finally, the \text{findall}/3 predicate is used on the set of the built tableaux for finding all the clashes contained in them in order to collect all the possible explanations.

As mentioned before, in each rule application round, the applicability of a rule is checked by calling \text{modify_ABox}/5.

apply_all_rules(Tab0,Tab):-
  apply_det_rules([...],Tab0,Tab1),
  (Tab0=Tab1 *->
    Tab=Tab1;
    apply_all_rules(Tab1,Tab)
  ).

apply_det_rules([],Tab0,Tab):-
  apply_nondet_rules([...],Tab0,Tab).

apply_det_rules([H|_],Tab0,Tab):-
  call(H,Tab0,Tab),!.

apply_det_rules([_|T],Tab0,Tab):-
  apply_det_rules(T,Tab0,Tab).

apply_nondet_rules([],Tab,Tab).

apply_nondet_rules([H|_],Tab0,Tab):-
  call(H,Tab0,Tab!),
  member(Tab,Tab),
  dif(Tab0,Tab).

apply_nondet_rules([_|T],Tab0,Tab):-
  apply_nondet_rules(T,Tab0,Tab).

Fig. 6. Application of the expansion rules by means of the predicates \text{apply_all_rules}/2, \text{apply_det_rules}/3 and \text{apply_nondet_rules}/3. The list [...\] contains the available rules and is different in \text{TRILL} and \text{TRILL}P.

4. Improving \text{TRILL}

The improvements implemented in the new version of \text{TRILL} are basically twofold: a detailed analysis of the KB during the loading phase and an optimization of the application order of the tableau expansion rules. As we will see in the following, the first extension is used for speeding up the application of some expansion rules while the second for speeding up the general expansion process. In Section 6 we compare the two versions of the \text{TRILL} framework to show that the implemented extensions can improve the performances of the reasoners. In particular, we expect a little overhead in the initial phase of the query answering process, especially in the case of simple KBs, that will be amortized the more queries are executed.

The extensions described in this paper are available in a git repository at https://github.com/rzese/trill. They are implemented in the branch called \text{trill-beta-version}, in order to maintain both versions downloadable to allow the replication of the tests presented in Section 6.
The first improvement allows TRILL to collect useful information about the KB during the loading of the KB in the Prolog database. In particular, TRILL makes use of the dictionaries of SWI-Prolog, a collection of key-value pairs, to keep in memory a structure containing information such as the number and the list of (complex) concepts, individuals and properties of the KB, together with the concept hierarchy and information about the axioms used to build the hierarchy. These pieces of information will be kept in memory so that the time to collect them is paid only once at the start-up. The management of this data structure is implemented in a new module called `utility_kb`, which is positioned on top of the layer dedicated to the translation and the loading of the axioms, as shown in Figure 7.

In turn, the module `utility_translation` has been refactored in order to quickly and recursively analyse the information contained in the KB and prepare it to be saved in the structure containing the hierarchy. This structure, called `kb`, is initialized by the `init_hierarchy/1` predicate, which is called by the `module utility_translation`. The code of the `init_hierarchy/1` predicate is shown in Figure 8.

The `kb` dictionary contains:

- `hierarchy`: an unweighted graph representing the concept hierarchy in the form of a tree. It is initialized as a tree containing two disconnected nodes called 0 and n representing the OWL built-in concepts Thing (⊤) and Nothing (⊥) respectively. The initialization is done by means of the `vertices_edges_to_ugraph/3` predicate of the `ugraphs` library of SWI-Prolog, which is used to manage the tree.
- `nClasses`: contains the number of the different concepts.
- `nIndividuals`: is the number of the named individuals of the KB.
- `disjointClasses`: a list of pairs indicating concepts that are disjoint.
- `classes`: is a dictionary mapping each node of the hierarchy tree to the concepts it represents. In particular, the Thing concept is associated to the 0 node while the Nothing concept to the n node.
- `classesName`: the list of the concepts contained in the KB. It is initialized to contain Thing and Nothing.
- `explanations`: a list containing the axioms used to create each edge of the hierarchy. This will be used during the expansion of the tableau to rapidly update the explanations.
- `individuals`: the list of individuals.
- `annotationProperties`: the list of annotation properties.
- `dataProperties`: the list of data properties. Though TRILL does not fully support datatypes management, we keep the list in order to quickly check during the expansion whether a certain property connects one individual with a scalar value.
- `datatypes`: the list of used datatypes. Note that if the KB is probabilistic at least a floating point datatype is used by the probability annotations.
- `objectProperties`: the list of object properties.

After the initialization, each atomic concept of the KB is added through the `add_class/3` predicate shown in Figure 9. Given the `kb` structure, and a concept `Class`, it looks for it in the map between nodes and concepts contained in `kb`. If the class is already present in the list it means that the hierarchy already contains such a concept and therefore nothing must be done. Otherwise, the new concept is added to the list of classes and to the map retrieving the name of the new node to insert in the hierarchy graph. This new node...
Fig. 8. Code of the `init_hierarchy/1` predicate, which is used for building the structure containing the concept hierarchy.

```prolog
init_hierarchy(kb{usermod:M,
  hierarchy:TreeH,
  nClasses:1,
  nIndividuals:0,
  disjointClasses:[0-'n'],
  classes:Classes,
  classesName:ClassesName,
  explanations:[],
  individuals:[],
  annotationProperties:[],
  dataProperties:[],
  datatypes:[],
  objectProperties:[]}):-
  vertices_edges_to_ugraph([0,'n'],[],TreeH),
  Classes=classes{'n' : 'http://www.w3.org/2002/07/owl#Nothing',
  0 : 'http://www.w3.org/2002/07/owl#Thing'},
  ClassesName=['http://www.w3.org/2002/07/owl#Nothing',
  'http://www.w3.org/2002/07/owl#Thing'],
  ..

Fig. 9. Implementation of the `add_class/3` predicate. It adds a new concept `Class` to the hierarchy by updating its structure and the list containing information about all the classes of the KB. The `find/1` predicate is used to perform the search by value in the dictionary.

```prolog
add_class(KB0,Class,KB):-
  Classes0=KB0.classes,
  ClassesN=KB0.classesName,
  \+ _=Classes0.find(Class),
  NC0=KB0.nClasses,
  NC is NC0 + 1,
  Classes=Classes0.put(NC0,Class),
  add_edges(KB0.hierarchy, [0-NC0],TreeH),
  add_subClass_expl(KB0.usermod,KB0.explanations,Class,
  'http://www.w3.org/2002/07/owl#Thing',Expls),
  KB=KB0.put([hierarchy=TreeH,nClasses=NC,classes=Classes,
  explanations=Expls,classesName=[Class|ClassesN]]).
```
add_hierarchy_link(KB0,C,C1,KB):-
Classes0=KB0.classes,
PC=Classes0.find(C),
PC1=Classes0.find(C1),!,
(dif(PC,PC1) ->
  ( add_hierarchy_link_int(KB0,PC,PC1,C,C1,KB) )
; ( add_subClass_expl(KB0.usermod,KB0.explanations,C,C1,Expls),
  KB=KB0.put(explanations,Expls))
).

add_hierarchy_link_int(KB0,PC,PC1,C,C1,KB):-
are_subClasses_int(KB0,PC,PC1),!,
merge_classes_int(KB0,PC,PC1,KB1),
add_subClass_expl(KB1.usermod,KB1.explanations,C,C1,Expls),
KB=KB1.put(explanations,Expls).

add_hierarchy_link_int(KB0,PC,PC1,C,C1,KB):-
del_edges(KB0.hierarchy,[0-PC],TreeH1),
add_edges(TreeH1,[PC1-PC],TreeH),
add_subClass_expl(KB0.usermod,KB0.explanations,C,C1,Expls),
KB=KB0.put([hierarchy=TreeH,explanations=Expls]).

Fig. 10. Implementation of the add_hierarchy_link/4 predicate. It modifies the hierarchy by creating a new link between the given concepts C and C1.

new merged node (as seen for equivalent-classes axioms). Otherwise, a new link between the nodes corresponding to C and D is added and the edge going from 0 (∅ concept) to C, if present, is removed to facilitate the management of the hierarchy graph.

Moreover, subsumption axioms are considered to find sub-class connections also regarding complex concepts, e.g., if there is an axiom C ⊑ D and the hierarchy contains two nodes representing the concepts ∃R.C and ∃R.D, the two nodes are linked. For each of these connections, the steps described above are performed to update the hierarchy. Finally, disjoint class axioms are considered in order to add links from node n to unsatisfiable classes.

The structure so built is then used to output information about the KB by calling the predicates kb_info/0, which writes the number of concepts, individuals and roles, and prints the hierarchy. Moreover, the structure is used during the expansion of the tableau in the → unfold rule that, instead of looking for a subsumption axiom C ⊑ D, searches for a concept D in the hierarchy. A possible further optimization here is to collect all the concepts directly connected to C instead of a single concept per rule call. Moreover, the structure can be exploited to quickly control whether the KB is inconsistent by checking if some individuals belong to an unsatisfiable concept, i.e., if the concept is equivalent to ⊥.

4.2. Improving the Tableau Expansion

As described in Section 2.2, the tableau algorithm can return a single justification. Therefore, a reasoner must implement a strategy to search the entire space. Classical tableau-based systems, implemented using imperative languages, forces the tableau algorithm to find a new justification by implementing ad-hoc algorithms such as the Hitting Set Algorithm. As seen in Section 3, the previous versions of TRILL, TRILLP and TORNADO leave the burden of this search to Prolog’s backtracking facilities, the rules simply pick a label from the tableau and try to expand it. However, this approach is far from being optimized.

Example 3. Suppose that the tableau contains k assertions \{a : C_1, ... , a : C_k \} and that the KB contains only the axiom C_k ⊑ C_{k+1}. Every expansion rule is called following a certain order usually defined in the specific system. In this example, the only rule that will modify the tableau is the → unfold rule. Therefore, to fully expand the tableau, all the rules that are called
before will be called twice. The \( \rightarrow \) unfold rule will also be called twice, the first to modify the tableau and the second to check whether a new modification can be done. The remaining rules, those considered after the \( \rightarrow \) unfold, will be called once. Every time a rule is called, a set of tests is performed to check whether the rule can be applied or not. For the sake of simplicity, we assume that every test is performed by an atomic instruction but, in reality, the complexity of the applicability test heavily depends on the expansion rule itself.

Let’s suppose that we try \( n \) rules and the \( \rightarrow \) unfold rule is the third in the order, suppose also that the assertions are tested in the order of their subscript. In the first round the first and second rules are tested on all the \( k \) assertions and they all fail. Then, the \( \rightarrow \) unfold rule is applied adding the new assertion \( a : C_{k+1} \) to the tableau. In the second round of application of the expansion rules, each expansion rule is tested for each of the \( k + 1 \) assertions, but no one can modify the tableau, which is fully expanded. At the end of the process \( 3k + n(k + 1) \) tests have been performed to decide whether a rule can be applied or not.

In the worst case, where the \( \rightarrow \) unfold is the last rule applied, the number of tests is \( nk + n(k + 1) = 2nk + n \)

However, as one can see from Figure 1, every rule to be applied looks for the presence of an assertion in the tableau. This means that, if an assertion has already been fully tested (every rules has already been applied and its tracing function has not been changed by now), it doesn’t lead to further expansions of the tableau and, hence, it won’t until the end of the expansion.

Following this observation, we changed the way TRILL, TRILL\(^P\) and TORNADO apply the expansion rules. In the new version, in each round of rule application, instead of trying every rule \( w.r.t. \) every assertion, they keep updated an expansion queue \( \text{ExpQueue} \).

The expansion queue \( \text{ExpQueue} \) is initialized to-gether with the initial tableau as the list of every assertion contained in the tableau. Predicate \( \text{apply\_all\_rules/2} \) of Figure 6 is replaced by the \( \text{expand\_queue/3} \) predicate, shown in Figure 11, which calls a slightly different version of \( \text{apply\_all\_rules/2} \), shown in Figure 12, which takes now 3 arguments. The \( \text{expand\_queue/3} \) predicate scans the \( \text{ExpQueue} \) list and, for each assertion in it, calls the expansion rules until no more rules are applicable to the selected assertion. To fully expand this assertion, the \( \text{apply\_all\_rules/3} \) predicate, shown in Figure 12, is called.

```prolog
apply_all_rules(Tab0, EA, Tab) :-
  apply_all_rules(Tab0, EA, Tab).
```

```prolog
expand_queue(Tab0, [EA|OtherEAs], Tab) :-
  apply_all_rules(Tab0, EA, Tab1),
  update_queue(OtherEAs, EA, NewExpQueue),
  expand_queue(Tab1, NewExpQueue, Tab).
```

**Fig. 11. Application of the expansion rules by means of the predicate \( \text{expand\_queue/3} \), which in turns calls \( \text{apply\_all\_rules/3} \) and \( \text{update\_queue/3} \).**

**Fig. 12. Application of the expansion rules by means of the predicates \( \text{apply\_all\_rules/3}, \text{apply\_det\_rules/5} \) and \( \text{apply\_nondet\_rules/5} \). The list \( \text{Rules} \) contains the available rules and is different in TRILL, TRILL\(^P\) and TORNADO.**

differs from \( \text{apply\_all\_rules/2} \) of Figure 6 because it takes as input an assertion \( EA \) to expand. This assertion is then given to the expansion rules, which now take as input also the assertion \( EA \). Figure 13 shows the code of the non-deterministic rule \( \rightarrow \) \( ⊔ \) (or_rule/3). The predicate searches the tableau \( (A, T) \) for the explanations of the given assertion and
Fig. 13. Code of the \( \rightarrow \text{⊔} \) rule implemented in the new version of
the systems. It unifies the list \( L \) with all the tableaux resulting by
the application of the rule. The assertion to expand is unified with
the concept and individual the rule looks for, in this way the search
for the explanation by means of the \texttt{findClassAssertion/4} is
performed only if the rule can be applied.

\begin{verbatim}
or_rule((A,T), (unionOf(LC), Ind), L):=
  findClassAssertion(unionOf(LC), Ind, Expl,A),
  \(+ indirectly_blocked(Ind, (A,T)),
  findall((A1,T),
  scan_or_list(LC, Ind, Expl, A,A1),L),
  dif(L,[])).
\end{verbatim}

Fig. 14. Code of the \( \rightarrow \text{unfold} \) rule implemented in the new version
of the systems. It checks whether the class \( C \) from the input assertion
is atomic and extract from the input tableau the information collected
so far. Then it looks for a class \( D \) which is a superclass or an
equivalent class of \( C \), builds the explanation \( \texttt{AxL} \) for the new class
assertion found (by using \texttt{and_f/3}) and adds it to the tableau to
update it. \texttt{and_f/3} performs a conjunction of two explanations.

\begin{verbatim}
unfold_rule((A0,T), (C,Ind), (A,T)):-
  atomic(C),
  findClassAssertion(C,Ind,Expl,A0),
  find_sub_sup_class(C,D,Ax),
  and_f(Ax,Expl,AxL),
  modify_ABox(A0,D,Ind,AxL,A).

find_sub_sup_class(C,D,Ex):-
  hierarchy(H),
  get_subclass(C,H,D),
  get_subclass_explanation(C,D,H,Ex),
\end{verbatim}

Example 4. Let’s consider the case depicted in Example 3, where we have a tableau containing \( k \) assertions \( \{ a : C_1, \ldots, a : C_k \} \), the axiom \( C_k \subseteq C_{k+1} \), and \( n \) rules
of which the \( \rightarrow \text{unfold} \) rule is applied as the third rule.

The new version of TRILL tries to apply every rule that match with the assertion and, for each matching
rule, perform the applicability test. Let assume for simplicity that every rule matches with every assertions in
the tableau, thus every rule performs the test to every assertion. Under this simplification, similarly to Example
3, in the first round TRILL tests all the assertions. The main difference in this case is in the second round, where \( \texttt{ExpQueue} \) contains only the new assertion \( a : C_{k+1} \), therefore, the number of tests in this case is
\( nk + n \), which corresponds with the worst case and
it is clearly less than \( 3k + n(k + 1) = nk + 3k + n \) or even
less than the worst case \( 2nk + n \) of the previous version.

Note that many rules match only with certain assertions, e.g., the \( \rightarrow \text{⊔} \) matches only with \texttt{unionOf}
concepts. This means that not all the rules will test their applicability, reducing even further the number of tests.

The improvement that one can achieve hugely depends on the KB itself. In the simplest case, the number of tests to perform are the same in the two versions. In this case, the expansion of the tableau imple-
Example 5. Consider a slightly different version of the tableau and KB of Examples 3 and 4 where the tableau contains only the assertion \( \{a : C_1\} \) and the KB contains a set of axioms \( C_i \sqsubseteq C_{i+1} \) with \( i \) from 1 to \( k \). With the previous version of TRILL, the number of tests to perform in the worst case is \( n(1 + \ldots + (k+1)) \) while the new version needs only \( n(k+1) \) tests. For \( k = 1 \) the difference is \( 3 : 2 \) that increases to \( 6 : 1 \) when \( k = 10 \).

5. Related Work

As far as the creation of the concepts hierarchy is concerned, many reasoners implement similar solutions to classify the KB named concepts. HermiT [39], for example, represents the hierarchy as a triple \((V, H, \rho)\), where \( V \) is the set of elements considered in the hierarchy, i.e., (set of) concepts and roles, \( H \) is a set of pairs representing the subsumption connections between elements in \( V \) and \( \rho \) is a mapping between elements in \( V \) and concepts (roles). This representation is similar to the one implemented by TRILL, where \( V \in \mathbb{N} \cup \{n\} \), \( \rho \) can be seen as the classes map while \( H \) is a representation of the hierarchy tree. HermiT starts by adding the pairs representing obvious subsumptions in \( H \) and then puts the Nothing concept as the leaf of each branch. Then, as in TRILL, it scans the graph for cycles in order to group equivalent classes and looks for unsatisfiable classes and/or disjoint concepts that are linked in the graph to update the mapping and the hierarchy.

The main difference with TRILL is that HermiT considers the construction of the hierarchy as a separate task, called classification, while TRILL uses the process to collect information about the KB that can be used during the reasoning process. TRILL only performs structural checks and simple inferences to build the hierarchy and finds possible unsatisfiable concepts without considering assertions but only subsumptions and disjunctions. On the other hand, HermiT performs many satisfiability tests on the nodes of the graph during the construction of the hierarchy. For these reasons, a direct comparison of the two methods would not be much meaningful. However, given the similarities of the two approaches, TRILL can easily be extended to perform the classification task. However, this is left as a future work. Similarly to HermiT, state-of-the-art reasoners, such as FaCT++ [3] and Pellet [1], perform the classification task as a stand-alone inference process. The main differences between the implementations of this task in the different reasoners can be found in the optimizations implemented for reducing the calls to the reasoner to perform consistency and subsumption tests. All these reasoners consider named classes and only a restricted set of inferred subclass relations, while TRILL considers a larger set of possible inferred connections and concepts to be both nominal and complex.

Similarly to TRILL, Pellet also allows to pre-compute the hierarchy to be used for answering queries, however this setting is not enabled by default and, as far as we know\(^1\), it executes the classification task and keeps in memory the information collected.

As for the optimization of the expansion queue, the closest approach is the ToDo list implemented in FaCT++ [3]. This approach creates one or more queues, depending on the priority given to expansion rules, which contain pairs (node, concept). Each time a new label is added to the tableau, the corresponding pair is added to the queues to be expanded. If there is a single queue the pair is added at the end of the queue, if there are more than one queue the pair is included in the right queues according with the expansion rules that can be applied to the new entry. The tableau algorithm takes a pair from the queues following their priority and stops when all queues are empty. The priority thus defines an order in the application of the rules that in FaCT++ can be changed by the user. This means that if there are three different queues, the rules associated with the first queue are applied to the assertions in the queue, then the rules of the second queue are applied to the pairs of the second queue and so on. If a pair is added to more than one queue, for example the first and the third, its expansion will be performed in two different moments, applying first the rules of the first queue and then those of the third, but after the application of all the other possible expansions from the first and the second queues.

It is easy to see that the pairs (node, concept) are equivalent to class assertions because a node in the tableau represents an individual. Unlike FaCT++,

TRILL systems TRILL systems consider also asser-

\(^1\)Unfortunately, there are not explicit documentation nor any publications about this option.
6. Experiments

To test the effectiveness of the optimizations introduced in the new version of the TRILL framework, we compared the system TRILL, TRILL P, and TOR NADO with their old versions. To facilitate the reader, in the following we refer to the new version using the word “hier.”, while the previous version will be indicated with “prev.”.

The experiments were done by following the tests performed in [17], where “prev.” TRILL systems have been compared with the probabilistic reasoners BORN [40], BUNDLE [13, 41], and PRONTO [42], and with the non-probabilistic reasoners Pellet [1], Konclude [43], HermiT [2], FaCT++ [3], and JFact3. The results of these tests showed that a Prolog implementation of the tableau algorithm, and in particular the “prev.” TRILL systems, can achieve better results than state-of-the-art (non-)probabilistic reasoners. We refer the reader to [17] for an in-depth description of these results.

Differently from [17], in this paper we consider only the different implementations of the TRILL framework. To carry out the comparison we performed three experiments regarding non-probabilistic inference, where we asked TRILL, TRILL P, and TOR NADO to find the explanations of queries, and three regarding probabilistic inference, where we computed the probability of the queries. We maintain the same outline used in [17], to facilitate the reader that wants to compare the results presented in the two articles.

All tests were run on the HPC System Galileo equipped with Intel Xeon E5-2697 v4 (Broadwell) @ 2.30 GHz.

6.1. Non-Probabilistic Inference

In the non-probabilistic inference setting, we performed three different tests considering KBs modelling real world domains (Test 1) and artificial KBs (Test 2, and Test 3).

Test 1 We used four real-world KBs:

- BRCA 4, which models the risk factors of breast cancer depending on many factors such as age and drugs taken;
- an extract of DBPedia ontology 5, containing structured information of Wikipedia, usually those contained in the information box on the right hand side of a page;
- BioPAX level 3 6, which models metabolic pathways;
- Vicodi 7, which contains information on European history and models historical events and important personalities.

We used a version of the DBpedia and BioPAX KBs without the ABox and a version of BRCA and Vicodi with an ABox containing 1 individual and 19 individuals respectively. We randomly created 50 subclass-of queries for DBpedia and BioPAX and 50 instance-of queries for the other two, ensuring each query had at least one explanation.

For each query, we asked:

- TRILL to answer whether the query is true or false (TRILL-1 in the table), and to find the set of all the justifications (TRILL-all in the table). TRILL can answer yes/no to a query by stopping its search when it finds the first explanation, therefore, Boolean queries answering has the same performances of finding one single explanation.
- TRILL P to build the pinpointing formula. Since TRILL P can only return the complete pinpointing formula, Boolean query answering has the same performance than finding all the justifications, therefore we do not differentiate these two cases in the results.

4http://www2.cs.man.ac.uk/~klinovp/pronto/brc/cancer_cc.owl
5http://dbpedia.org/
6http://www.biopax.org/
7http://www.vicodi.org/
Table 1 shows the average number of justifications and the average running time (in seconds) for answering Boolean queries with the reasoners TRILL, TRILL$^p$, and TORNADO (TORN.) in their two versions in Test 1, w.r.t. 4 different KBs. Each cell contains the running time. Bold values highlight the fastest reasoner for each KB.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Avg. N. Just.</th>
<th>Vers.</th>
<th>TRILL-1</th>
<th>TRILL-all</th>
<th>TRILL$^p$</th>
<th>TORN.</th>
</tr>
</thead>
<tbody>
<tr>
<td>BioPAX</td>
<td>3.92</td>
<td>prev.</td>
<td>0.079</td>
<td>0.100</td>
<td>0.081</td>
<td>0.078</td>
</tr>
<tr>
<td></td>
<td></td>
<td>hier.</td>
<td>0.160</td>
<td>0.184</td>
<td>0.160</td>
<td>0.161</td>
</tr>
<tr>
<td>DBPedia</td>
<td>16.32</td>
<td>prev.</td>
<td>0.010</td>
<td>0.010</td>
<td>0.123</td>
<td>0.007</td>
</tr>
<tr>
<td></td>
<td></td>
<td>hier.</td>
<td>0.028</td>
<td>0.028</td>
<td>0.0893</td>
<td>0.026</td>
</tr>
<tr>
<td>Vicodi</td>
<td>1.02</td>
<td>prev.</td>
<td>0.032</td>
<td>0.036</td>
<td>0.029</td>
<td>0.030</td>
</tr>
<tr>
<td></td>
<td></td>
<td>hier.</td>
<td>0.078</td>
<td>0.078</td>
<td>0.077</td>
<td>0.079</td>
</tr>
<tr>
<td>BRCA</td>
<td>6.49</td>
<td>prev.</td>
<td>0.182</td>
<td>0.183</td>
<td>1.063</td>
<td>0.060</td>
</tr>
<tr>
<td></td>
<td></td>
<td>hier.</td>
<td>0.038</td>
<td>0.039</td>
<td>0.082</td>
<td>0.021</td>
</tr>
</tbody>
</table>

- TORNADO to build the BDD representing the pinpointing formula. Analogously to TRILL$^p$, TORNADO do not stop when it finds that the query is true, but it builds the complete BDD. Boolean query answering, therefore, corresponds with building the BDD for the query and checking that its probability is not 0.

Table 1 shows the average number of explanations and the average running time in seconds to answer queries on each KB in the settings described above for the previous “prev.” version and the new “hier.” version of the systems. “TORN.” column shows the running time of TORNADO.

The results show that the “hier.” version of the systems presents as expected a little overhead due to the use of the hierarchy and the use of the expansion queue. However, when the running time increases this overhead is compensated by a faster management of the expansion, as shown for BRCA. TRILL$^p$ is the system that achieves the best speed-up from the implemented optimization.

**Test 2** In this second test we consider the artificial KB shown in the following:

\[
\begin{align*}
C_{1,1} & \sqsubseteq C_{1,2} \sqsubseteq \ldots \sqsubseteq C_{1,n} \sqsubseteq C_{n+1} \\
C_{1,1} & \sqsubseteq C_{2,2} \sqsubseteq \ldots \sqsubseteq C_{2,n} \sqsubseteq C_{n+1} \\
C_{1,1} & \sqsubseteq C_{3,2} \sqsubseteq \ldots \sqsubseteq C_{3,n} \sqsubseteq C_{n+1} \\
\vdots \\
C_{1,1} & \sqsubseteq C_{m,2} \sqsubseteq \ldots \sqsubseteq C_{m,n} \sqsubseteq C_{n+1} \\
a & : C_{1,1}
\end{align*}
\]

with $m$ and $n$ varying to increase the number of axioms. This KB forces the creation of an increasing number of choice points in order to investigate the effect of the non-determinism in the choice of rules. In fact, finding all the justifications for the query $Q = a : C_{n+1}$ forces to find $m$ justifications containing $n+1$ axioms each: $n$ subclass-of axioms and 1 assertion axiom. Moreover, during the expansion of this KB the difference between the rule calls made by “prev.” and “hier.” versions is near to 0, as only one call of the $\rightarrow$ unfold rule on each assertion succeeds. In fact, the new assertion in “prev.” version is added at the beginning of the list of assertions in the tableau of the TRILL systems and the $\rightarrow$ unfold rule is the third applied therefore the “hier.” version can cut a small number of rule tests.

We executed the query $Q$ 100 times to compute the average running time that each system in each version needs to compute all the justifications. We varied $m$ and $n$ between 1 to 7. Tables 2, 3, and 4 report the results for TRILL, TRILL$^p$, and TORNADO respectively. Columns correspond to $n$ while rows correspond to $m$. From the results, it is clear that a little overhead is present in the new version. As already said, this overhead is more visible when the KB is simple as in this case. The running time is almost always of the same order of magnitude for both the versions.

To further test how the two versions perform with larger KBs, we extended this test by increasing the value of $m$ and $n$, varying them from 10 to 100 in step of 10. We set a timeout of 30 minutes on the fastest system and a timeout of 95 minutes on the slowest system. The cells with “-” indicate that the timeout on the slowest system occurred while italic values show the
Table 2
Average time (in seconds) for computing all the explanations with the reasoner TRILL in the two versions in Test 2. Columns correspond to \( n \) while rows correspond to \( m \). \( n \) and \( m \) vary from 1 to 7 in step of 1. In bold the best time for each size.

<table>
<thead>
<tr>
<th>TRILL</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>prev.</td>
<td>0.0003</td>
<td>0.0003</td>
<td>0.0003</td>
<td>0.0004</td>
<td>0.0004</td>
<td>0.0004</td>
<td>0.0005</td>
</tr>
<tr>
<td>hier.</td>
<td>0.0003</td>
<td>0.0003</td>
<td>0.0004</td>
<td>0.0005</td>
<td>0.0005</td>
<td>0.0006</td>
<td>0.0007</td>
</tr>
<tr>
<td>prev.</td>
<td>0.0003</td>
<td>0.0004</td>
<td>0.0005</td>
<td>0.0006</td>
<td>0.0007</td>
<td>0.0008</td>
<td>0.0009</td>
</tr>
<tr>
<td>hier.</td>
<td>0.0004</td>
<td>0.0005</td>
<td>0.0007</td>
<td>0.0009</td>
<td>0.0011</td>
<td>0.0013</td>
<td>0.0017</td>
</tr>
<tr>
<td>prev.</td>
<td>0.0004</td>
<td>0.0006</td>
<td>0.0007</td>
<td>0.0008</td>
<td>0.0011</td>
<td>0.0012</td>
<td>0.0014</td>
</tr>
<tr>
<td>hier.</td>
<td>0.0005</td>
<td>0.0007</td>
<td>0.0011</td>
<td>0.0015</td>
<td>0.0019</td>
<td>0.0024</td>
<td>0.0033</td>
</tr>
<tr>
<td>prev.</td>
<td>0.0005</td>
<td>0.0007</td>
<td>0.0009</td>
<td>0.0012</td>
<td>0.0014</td>
<td>0.0018</td>
<td>0.0022</td>
</tr>
<tr>
<td>hier.</td>
<td>0.0006</td>
<td>0.0010</td>
<td>0.0016</td>
<td>0.0022</td>
<td>0.0033</td>
<td>0.0044</td>
<td>0.0054</td>
</tr>
<tr>
<td>prev.</td>
<td>0.0006</td>
<td>0.0009</td>
<td>0.0012</td>
<td>0.0015</td>
<td>0.0020</td>
<td>0.0025</td>
<td>0.0030</td>
</tr>
<tr>
<td>hier.</td>
<td>0.0008</td>
<td>0.0013</td>
<td>0.0022</td>
<td>0.0035</td>
<td>0.0047</td>
<td>0.0060</td>
<td>0.0071</td>
</tr>
<tr>
<td>prev.</td>
<td>0.0007</td>
<td>0.0011</td>
<td>0.0015</td>
<td>0.0020</td>
<td>0.0026</td>
<td>0.0033</td>
<td>0.0041</td>
</tr>
<tr>
<td>hier.</td>
<td>0.0010</td>
<td>0.0018</td>
<td>0.0031</td>
<td>0.0047</td>
<td>0.0062</td>
<td>0.0074</td>
<td>0.0090</td>
</tr>
<tr>
<td>prev.</td>
<td>0.0008</td>
<td>0.0013</td>
<td>0.0018</td>
<td>0.0025</td>
<td>0.0033</td>
<td>0.0043</td>
<td>0.0053</td>
</tr>
<tr>
<td>hier.</td>
<td>0.0012</td>
<td>0.0024</td>
<td>0.0042</td>
<td>0.0059</td>
<td>0.0074</td>
<td>0.0093</td>
<td>0.0126</td>
</tr>
</tbody>
</table>

Table 3
Average time (in seconds) for computing all the explanations with the reasoner TRILL\(^P\) in the two versions in Test 2. Columns correspond to \( n \) while rows correspond to \( m \). \( n \) and \( m \) vary from 1 to 7 in step of 1. In bold the best time for each size.

<table>
<thead>
<tr>
<th>TRILL(^P)</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>prev.</td>
<td>0.0003</td>
<td>0.0003</td>
<td>0.0003</td>
<td>0.0003</td>
<td>0.0004</td>
<td>0.0004</td>
<td>0.0004</td>
</tr>
<tr>
<td>hier.</td>
<td>0.0003</td>
<td>0.0004</td>
<td>0.0004</td>
<td>0.0005</td>
<td>0.0006</td>
<td>0.0006</td>
<td>0.0007</td>
</tr>
<tr>
<td>prev.</td>
<td>0.0012</td>
<td>0.0015</td>
<td>0.0017</td>
<td>0.0021</td>
<td>0.0025</td>
<td>0.0030</td>
<td>0.0035</td>
</tr>
<tr>
<td>hier.</td>
<td>0.0009</td>
<td>0.0012</td>
<td>0.0015</td>
<td>0.0018</td>
<td>0.0024</td>
<td>0.0029</td>
<td>0.0037</td>
</tr>
<tr>
<td>prev.</td>
<td>0.0022</td>
<td>0.0030</td>
<td>0.0039</td>
<td>0.0050</td>
<td>0.0063</td>
<td>0.0074</td>
<td>0.0089</td>
</tr>
<tr>
<td>hier.</td>
<td>0.0017</td>
<td>0.0023</td>
<td>0.0031</td>
<td>0.0040</td>
<td>0.0050</td>
<td>0.0058</td>
<td>0.0073</td>
</tr>
<tr>
<td>prev.</td>
<td>0.0038</td>
<td>0.0055</td>
<td>0.0074</td>
<td>0.0101</td>
<td>0.0121</td>
<td>0.0146</td>
<td>0.0184</td>
</tr>
<tr>
<td>hier.</td>
<td>0.0025</td>
<td>0.0037</td>
<td>0.0050</td>
<td>0.0065</td>
<td>0.0082</td>
<td>0.0106</td>
<td>0.0130</td>
</tr>
<tr>
<td>prev.</td>
<td>0.0061</td>
<td>0.0092</td>
<td>0.0131</td>
<td>0.0168</td>
<td>0.0210</td>
<td>0.0264</td>
<td>0.0316</td>
</tr>
<tr>
<td>hier.</td>
<td>0.0036</td>
<td>0.0055</td>
<td>0.0071</td>
<td>0.0099</td>
<td>0.0130</td>
<td>0.0160</td>
<td>0.0188</td>
</tr>
<tr>
<td>prev.</td>
<td>0.0093</td>
<td>0.0147</td>
<td>0.0201</td>
<td>0.0263</td>
<td>0.0344</td>
<td>0.0422</td>
<td>0.0512</td>
</tr>
<tr>
<td>hier.</td>
<td>0.0052</td>
<td>0.0073</td>
<td>0.0105</td>
<td>0.0144</td>
<td>0.0177</td>
<td>0.0220</td>
<td>0.0271</td>
</tr>
<tr>
<td>prev.</td>
<td>0.0137</td>
<td>0.0213</td>
<td>0.0298</td>
<td>0.0404</td>
<td>0.0511</td>
<td>0.0638</td>
<td>0.0794</td>
</tr>
<tr>
<td>hier.</td>
<td>0.0069</td>
<td>0.0099</td>
<td>0.0145</td>
<td>0.0186</td>
<td>0.0243</td>
<td>0.0301</td>
<td>0.0368</td>
</tr>
</tbody>
</table>

ratio when the slowest system exceeded the timeout of 30 minutes. The fastest systems never exceeded the timeout of 30 minutes. Tables 5, 6, and 7 show the ratio \"prev./\"hier.\" for TRILL, TRILL\(^P\), and TORNADO for increasing \( m \) and \( n \). As expected, the overhead in this test is significant. However, the larger the KB the higher the ratio. It is important to notice that the more difficult is the application of the rule the higher the ratio. In other words, if we consider TORNADO that is the fastest system, the application of the \( \rightarrow unfold \) rule is almost instantaneous. In this case the \"hier.\" version is always slower in this test with the highest running time around 300 seconds for the larger KB. However, from Table 7 it is possible to see that the ratio becomes higher and higher (up to 0.843) as the size of the KB increases. Regarding TRILL, which is the second fastest system, there are cases where the ratio is higher than 1. With \( n = m = 100 \) the running time was few higher than 300 seconds showing, as expected, that the overhead is more significant when the rule is simple to ap-
Results on TRILL confirm this theory. In fact, rule application in this system is more difficult than in the other systems because it relies on a SAT solver to test the applicability. In this case the reduced number of tests impacts significantly on the running time. For example, the “hier.” version reaches the 30 minutes timeout only for \( n = m = 100 \) while the “prev.” version reaches this limit with \( n = 30, m = 60 \). With this KB the “hier.” version’s average running is lower than 42 seconds. The second timeout occurs for the “prev.” version with \( n = 60, m = 90 \). With this KB the “hier.” version’s average running time is lower than 495 seconds.

Test 3 The third experiment stresses even further the management of the backtracking by considering artificial KBs of increasing size where, despite the increment of the size is linear in the number of axioms, the number of justifications increases exponentially. Such KBs are of the form

\[(C_{1,i}) B_{i-1} \sqsubseteq P_i \sqcap Q_i \quad (C_{2,i}) P_i \sqsubseteq B_i \quad (C_{3,i}) Q_i \sqsubseteq B_i\]

with \( 1 \leq i \leq n, n \) an integer, \( n \geq 1 \). The query \( Q = B_0 \sqsubseteq B_n \) has \( 2^n \) explanations, even if the KB has a size that is linear in \( n \). For \( n = 2 \) for example, we have 4 different explanations, namely

\[\{C_{1,1}, C_{2,1}, C_{1,2}, C_{2,2}\} \quad \{C_{1,1}, C_{3,1}, C_{1,2}, C_{2,2}\} \quad \{C_{1,1}, C_{2,1}, C_{1,2}, C_{3,2}\} \quad \{C_{1,1}, C_{3,1}, C_{1,2}, C_{3,2}\}\]

The corresponding pinpointing formula is

\[\bigwedge_{j \in \{1,n\}} C_{1,i} \wedge \bigwedge_{j \in \{1,n\}} \bigvee_{z \in \{2,3\}} C_{z,j}\]

whose size is linear in \( n \).

The value of \( n \) was increased from 2 to 10 in steps of 2. We performed the query \( Q \) 50 times for each KB to compute the average running time, shown in Table 8. We set a timeout of 10 minutes for each query execution, so the cells with “–” indicate that the timeout occurred.

Results show that the “hier.” version of the TRILL framework outperforms the “prev.” version. These results clearly show that the optimizations implemented in the systems can sensibly improve the performances. For the sake of completeness, TRILL’s running time with \( n = 10 \) was around 1384 seconds, while the ratio “prev./”hier.” of TRILL’s running time with \( n = 8 \) was 1.4098 with “hier.” TRILL’s running time around 183 minutes.

6.2. Probabilistic Inference

To experiment with probabilistic inference we set up three different scenarios. In particular, Test 4 and Test 5 extend with probability Test 1 and Test 3, while Test 6 works on versions of increasing size and complexity of the BRCA KB.

<table>
<thead>
<tr>
<th>TORN.</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>prev.</td>
<td>0.0005</td>
<td>0.0005</td>
<td>0.0006</td>
<td>0.0006</td>
<td>0.0006</td>
<td>0.0006</td>
<td>0.0007</td>
</tr>
<tr>
<td>hier.</td>
<td>0.0005</td>
<td>0.0005</td>
<td>0.0006</td>
<td>0.0006</td>
<td>0.0007</td>
<td>0.0008</td>
<td>0.0009</td>
</tr>
<tr>
<td>prev.</td>
<td>0.0006</td>
<td>0.0006</td>
<td>0.0007</td>
<td>0.0007</td>
<td>0.0008</td>
<td>0.0009</td>
<td>0.0010</td>
</tr>
<tr>
<td>hier.</td>
<td>0.0006</td>
<td>0.0006</td>
<td>0.0008</td>
<td>0.0010</td>
<td>0.0012</td>
<td>0.0014</td>
<td>0.0018</td>
</tr>
<tr>
<td>prev.</td>
<td>0.0006</td>
<td>0.0008</td>
<td>0.0011</td>
<td>0.0015</td>
<td>0.0020</td>
<td>0.0024</td>
<td>0.0031</td>
</tr>
<tr>
<td>hier.</td>
<td>0.0006</td>
<td>0.0010</td>
<td>0.0016</td>
<td>0.0022</td>
<td>0.0029</td>
<td>0.0042</td>
<td>0.0052</td>
</tr>
<tr>
<td>prev.</td>
<td>0.0007</td>
<td>0.0009</td>
<td>0.0013</td>
<td>0.0015</td>
<td>0.0019</td>
<td>0.0023</td>
<td>0.0027</td>
</tr>
<tr>
<td>hier.</td>
<td>0.0008</td>
<td>0.0014</td>
<td>0.0021</td>
<td>0.0030</td>
<td>0.0045</td>
<td>0.0058</td>
<td>0.0071</td>
</tr>
<tr>
<td>prev.</td>
<td>0.0008</td>
<td>0.0012</td>
<td>0.0015</td>
<td>0.0020</td>
<td>0.0024</td>
<td>0.0030</td>
<td>0.0036</td>
</tr>
<tr>
<td>hier.</td>
<td>0.0010</td>
<td>0.0017</td>
<td>0.0027</td>
<td>0.0043</td>
<td>0.0060</td>
<td>0.0073</td>
<td>0.0086</td>
</tr>
<tr>
<td>prev.</td>
<td>0.0009</td>
<td>0.0014</td>
<td>0.0018</td>
<td>0.0023</td>
<td>0.0030</td>
<td>0.0037</td>
<td>0.0046</td>
</tr>
<tr>
<td>hier.</td>
<td>0.0011</td>
<td>0.0021</td>
<td>0.0036</td>
<td>0.0056</td>
<td>0.0072</td>
<td>0.0087</td>
<td>0.0111</td>
</tr>
</tbody>
</table>

Table 4 Average time (in seconds) for computing all the explanations with the reasoner TORNADO in the two versions in Test 2. Columns correspond to \( n \) while rows correspond to \( m, n \) and \( m \) vary from 1 to 7 in step of 1. In bold the best time for each size.
Table 5
Ratio “prev.”/“hier.” for TRILL in Test 2. Columns correspond to $n$ while rows correspond to $m$. $n$ and $m$ vary from 10 to 100 in step of 10.

<table>
<thead>
<tr>
<th>TRILL</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>50</th>
<th>60</th>
<th>70</th>
<th>80</th>
<th>90</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.374</td>
<td>0.416</td>
<td>0.393</td>
<td>0.473</td>
<td>0.443</td>
<td>0.460</td>
<td>0.529</td>
<td>0.592</td>
<td>0.567</td>
<td>0.565</td>
</tr>
<tr>
<td>20</td>
<td>0.393</td>
<td>0.415</td>
<td>0.457</td>
<td>0.465</td>
<td>0.486</td>
<td>0.526</td>
<td>0.585</td>
<td>0.549</td>
<td>0.550</td>
<td>0.581</td>
</tr>
<tr>
<td>30</td>
<td>0.454</td>
<td>0.484</td>
<td>0.484</td>
<td>0.533</td>
<td>0.615</td>
<td>0.558</td>
<td>0.595</td>
<td>0.633</td>
<td>0.689</td>
<td>0.758</td>
</tr>
<tr>
<td>40</td>
<td>0.470</td>
<td>0.501</td>
<td>0.594</td>
<td>0.552</td>
<td>0.662</td>
<td>0.707</td>
<td>0.628</td>
<td>0.657</td>
<td>0.657</td>
<td>0.684</td>
</tr>
<tr>
<td>50</td>
<td>0.520</td>
<td>0.571</td>
<td>0.662</td>
<td>0.632</td>
<td>0.712</td>
<td>0.743</td>
<td>0.656</td>
<td>0.698</td>
<td>0.743</td>
<td>0.797</td>
</tr>
<tr>
<td>60</td>
<td>0.571</td>
<td>0.622</td>
<td>0.607</td>
<td>0.692</td>
<td>0.729</td>
<td>0.663</td>
<td>0.742</td>
<td>0.782</td>
<td>0.825</td>
<td>0.838</td>
</tr>
<tr>
<td>70</td>
<td>0.591</td>
<td>0.661</td>
<td>0.675</td>
<td>0.744</td>
<td>0.688</td>
<td>0.746</td>
<td>0.769</td>
<td>0.820</td>
<td>0.705</td>
<td>0.767</td>
</tr>
<tr>
<td>80</td>
<td>0.558</td>
<td>0.626</td>
<td>0.711</td>
<td>0.678</td>
<td>0.788</td>
<td>0.881</td>
<td>0.785</td>
<td>0.849</td>
<td>0.897</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>0.634</td>
<td>0.703</td>
<td>0.806</td>
<td>0.773</td>
<td>0.822</td>
<td>0.925</td>
<td>0.865</td>
<td>0.924</td>
<td>0.970</td>
<td>0.977</td>
</tr>
<tr>
<td>100</td>
<td>0.696</td>
<td>0.798</td>
<td>0.924</td>
<td>0.885</td>
<td>0.957</td>
<td>1.028</td>
<td>0.927</td>
<td>0.965</td>
<td>0.977</td>
<td>1.053</td>
</tr>
</tbody>
</table>

Table 6
Ratio “prev.”/“hier.” for TRILL in Test 2. Columns correspond to $n$ while rows correspond to $m$. $n$ and $m$ vary from 10 to 100 in step of 10.

<table>
<thead>
<tr>
<th>TRILL</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>50</th>
<th>60</th>
<th>70</th>
<th>80</th>
<th>90</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>2.881</td>
<td>2.663</td>
<td>2.718</td>
<td>2.755</td>
<td>2.784</td>
<td>2.852</td>
<td>2.860</td>
<td>2.853</td>
<td>2.803</td>
<td>2.906</td>
</tr>
<tr>
<td>80</td>
<td>17.417</td>
<td>17.187</td>
<td>17.317</td>
<td>17.103</td>
<td>– –</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>18.331</td>
<td>18.485</td>
<td>19.909</td>
<td>– –</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>21.290</td>
<td>22.037</td>
<td>23.163</td>
<td>– – – – – –</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7
Ratio “prev.”/“hier.” for TORNADO (TORN.) in Test 2. Columns correspond to $n$ while rows correspond to $m$. $n$ and $m$ vary from 10 to 100 in step of 10.

<table>
<thead>
<tr>
<th>TORN.</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>50</th>
<th>60</th>
<th>70</th>
<th>80</th>
<th>90</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.342</td>
<td>0.326</td>
<td>0.270</td>
<td>0.310</td>
<td>0.273</td>
<td>0.263</td>
<td>0.282</td>
<td>0.311</td>
<td>0.299</td>
<td>0.276</td>
</tr>
<tr>
<td>20</td>
<td>0.361</td>
<td>0.340</td>
<td>0.333</td>
<td>0.332</td>
<td>0.313</td>
<td>0.327</td>
<td>0.350</td>
<td>0.323</td>
<td>0.306</td>
<td>0.299</td>
</tr>
<tr>
<td>30</td>
<td>0.408</td>
<td>0.395</td>
<td>0.368</td>
<td>0.384</td>
<td>0.424</td>
<td>0.363</td>
<td>0.377</td>
<td>0.379</td>
<td>0.402</td>
<td>0.425</td>
</tr>
<tr>
<td>40</td>
<td>0.426</td>
<td>0.421</td>
<td>0.464</td>
<td>0.411</td>
<td>0.427</td>
<td>0.458</td>
<td>0.476</td>
<td>0.403</td>
<td>0.411</td>
<td>0.412</td>
</tr>
<tr>
<td>50</td>
<td>0.494</td>
<td>0.489</td>
<td>0.545</td>
<td>0.479</td>
<td>0.530</td>
<td>0.547</td>
<td>0.458</td>
<td>0.471</td>
<td>0.508</td>
<td>0.522</td>
</tr>
<tr>
<td>60</td>
<td>0.568</td>
<td>0.572</td>
<td>0.515</td>
<td>0.572</td>
<td>0.580</td>
<td>0.512</td>
<td>0.539</td>
<td>0.547</td>
<td>0.558</td>
<td>0.560</td>
</tr>
<tr>
<td>70</td>
<td>0.583</td>
<td>0.603</td>
<td>0.593</td>
<td>0.610</td>
<td>0.541</td>
<td>0.578</td>
<td>0.584</td>
<td>0.600</td>
<td>0.494</td>
<td>0.531</td>
</tr>
<tr>
<td>80</td>
<td>0.553</td>
<td>0.560</td>
<td>0.627</td>
<td>0.565</td>
<td>0.602</td>
<td>0.610</td>
<td>0.671</td>
<td>0.575</td>
<td>0.618</td>
<td>0.633</td>
</tr>
<tr>
<td>90</td>
<td>0.643</td>
<td>0.657</td>
<td>0.717</td>
<td>0.670</td>
<td>0.669</td>
<td>0.731</td>
<td>0.653</td>
<td>0.710</td>
<td>0.721</td>
<td>0.716</td>
</tr>
<tr>
<td>100</td>
<td>0.709</td>
<td>0.749</td>
<td>0.824</td>
<td>0.754</td>
<td>0.795</td>
<td>0.843</td>
<td>0.749</td>
<td>0.742</td>
<td>0.733</td>
<td>0.779</td>
</tr>
</tbody>
</table>
Table 8

<table>
<thead>
<tr>
<th>n</th>
<th>Vers.</th>
<th>TRILL</th>
<th>TRILL₁₀⁻ and TORNADO (TORN.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>prev.</td>
<td>0.001</td>
<td>0.005</td>
</tr>
<tr>
<td>4</td>
<td>prev.</td>
<td>0.009</td>
<td>0.197</td>
</tr>
<tr>
<td>6</td>
<td>prev.</td>
<td>0.345</td>
<td>23.741</td>
</tr>
<tr>
<td>8</td>
<td>prev.</td>
<td>21.368</td>
<td>0.094</td>
</tr>
<tr>
<td>10</td>
<td>prev.</td>
<td>–</td>
<td>0.438</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dataset</th>
<th>N. Just.</th>
<th>Vers.</th>
<th>TRILL</th>
<th>TRILL₁₀⁻</th>
<th>TORN.</th>
</tr>
</thead>
<tbody>
<tr>
<td>BioPAX</td>
<td>3.92</td>
<td>prev.</td>
<td>0.101</td>
<td>0.083</td>
<td>0.080</td>
</tr>
<tr>
<td>DBPedia</td>
<td>16.32</td>
<td>prev.</td>
<td>0.010</td>
<td>0.125</td>
<td>0.007</td>
</tr>
<tr>
<td>Vicodi</td>
<td>1.02</td>
<td>prev.</td>
<td>0.037</td>
<td>0.029</td>
<td>0.031</td>
</tr>
<tr>
<td>BRCA</td>
<td>6.49</td>
<td>prev.</td>
<td>0.184</td>
<td>1.071</td>
<td>0.062</td>
</tr>
</tbody>
</table>

Table 9

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Avg.</th>
<th>N. Just.</th>
<th>Vers.</th>
<th>TRILL</th>
<th>TRILL₁₀⁻</th>
<th>TORN.</th>
</tr>
</thead>
<tbody>
<tr>
<td>BioPAX</td>
<td>3.92</td>
<td>prev.</td>
<td>0.101</td>
<td>0.083</td>
<td>0.080</td>
<td></td>
</tr>
<tr>
<td>DBPedia</td>
<td>16.32</td>
<td>prev.</td>
<td>0.010</td>
<td>0.125</td>
<td>0.007</td>
<td></td>
</tr>
<tr>
<td>Vicodi</td>
<td>1.02</td>
<td>prev.</td>
<td>0.037</td>
<td>0.029</td>
<td>0.031</td>
<td></td>
</tr>
<tr>
<td>BRCA</td>
<td>6.49</td>
<td>prev.</td>
<td>0.184</td>
<td>1.071</td>
<td>0.062</td>
<td></td>
</tr>
</tbody>
</table>

Test 4  In this test we considered the non-probabilistic Test 1 and we annotated 50 randomly chosen axioms of each KB with a probability value generated using the learning system EDGE [44]⁸.

We ran the two versions of the TRILL systems to compute the probability of the same queries generated for Test 1 to collect the running times. Table 9 shows the average time in seconds taken by the systems for performing probabilistic inference. Differently from Test 1, there is only one column for TRILL, because in this case all the justifications must be found to compute the probability of the queries. For each dataset, the average number of justifications is also shown.

The results confirm those reported in Table 1. The extra time for the probability computation is negligible in this setting as due to a time measurement error in some cases the results of Table 9 are even better than those of Table 1. As for Test 1, TRILL₁₀⁻ achieves the best speed-up and in general when the running time is under the second, the overhead due to the management of the hierarchy and the expansion queue can make the execution of TRILL₁₀⁻ and TORNADO slower. However, as shown by BRCA KBs, it strongly depends on the complexity of the KB.

Test 5  In this test we assigned a random value of probability to every axiom of the KB of the non-probabilistic Test 3, we increased n from 2 to 10 in steps of 2 and run the query \( Q = B_0 \sqcup B_n \), 50 times for each reasoner. A timeout of 10 minutes was set. Table 10 shows, for each n, the average time in seconds taken by the systems for computing the probability of the query \( Q \). Cells with “\(-\)” indicate that the timeout occurred. Results confirm those of Test 3: the new implementation can better deal with the exponential blow-up of justifications.

Moreover, as seen in Test 4, the overhead required by the probability computation is negligible when the number of justifications is low. Also in this case, some results are better than those of Table 8 due to a time measurement error. For completeness, with \( n = 10 \) TRILL took about 1170 seconds, while the ratio “prev.”/“hier.” of TRILL₁₀⁻’s running time with \( n = 8 \)

---

⁸EDGE is an algorithm for parameter learning from a set of positive and negative examples.
was 7.6913 with “hier.” TRILL\(^P\)’s running time around 152 minutes. “hier.” version of TRILL systems again achieve the best results.

Test 6 The last test was performed by considering versions of BRCA of increasing size. To create the KBs, an increasing number of subclass-of probabilistic axioms were randomly generated and added in the initial non-probabilistic KB. The number of these axioms was varied from 9 to 26, and, for each number, 100 different consistent ontologies were created. The addition of these axioms, as shown also in [45] where the authors generated such KBs to test their probabilistic reasoner, can significantly increase the complexity of the reasoning. After the addition of the subclass-of axioms, for each KB an individual was added and randomly assigned to each simple class that appears in the probabilistic axioms with probability 0.6, where complex classes were split into their components, e.g., the complex class PostmenopausalWomanTakingTestosterone was divided into PostmenopausalWoman and WomanTakingTestosterone. Finally, for each KB we ran 100 probabilistic queries of the form \(a : C\) where \(a\) is the added individual and \(C\) is a class randomly selected among those that represent women under increased and lifetime risk such as WomanUnderLifetimeBRCRisk and WomanUnderStronglyIncreasedBRCRisk.

Tables 11, 12 and 13 show the execution time averaged over the 100 queries with the varying of the number of probabilistic axioms (“NPA” column) for the reasoners TRILL, TRILL\(^P\), and TORNADO respectively. Moreover, the ratio “prev.”/“hier.” is shown for each system and each size of the KBs. The averaged ratio ± standard deviation is also shown for each system.

Overall, the new version of the systems always outperforms the old one. In particular, the system that is the most affected by the optimizations is TRILL\(^P\), as shown also by the previous tests, followed by TORNADO, which demonstrates to be always the best system among those included in the TRILL framework.

<table>
<thead>
<tr>
<th>NPA</th>
<th>prev.</th>
<th>hier.</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>30.508</td>
<td>1.875</td>
<td>16.3</td>
</tr>
<tr>
<td>10</td>
<td>27.114</td>
<td>1.310</td>
<td>20.7</td>
</tr>
<tr>
<td>11</td>
<td>83.032</td>
<td>31.333</td>
<td>2.7</td>
</tr>
<tr>
<td>12</td>
<td>78.865</td>
<td>31.277</td>
<td>2.5</td>
</tr>
<tr>
<td>13</td>
<td>74.402</td>
<td>10.889</td>
<td>6.8</td>
</tr>
<tr>
<td>14</td>
<td>107.875</td>
<td>31.080</td>
<td>3.5</td>
</tr>
<tr>
<td>15</td>
<td>99.876</td>
<td>18.617</td>
<td>5.4</td>
</tr>
<tr>
<td>16</td>
<td>70.720</td>
<td>16.261</td>
<td>4.3</td>
</tr>
<tr>
<td>17</td>
<td>174.825</td>
<td>73.561</td>
<td>2.4</td>
</tr>
<tr>
<td>18</td>
<td>161.380</td>
<td>65.987</td>
<td>2.4</td>
</tr>
<tr>
<td>19</td>
<td>156.791</td>
<td>65.263</td>
<td>2.4</td>
</tr>
<tr>
<td>20</td>
<td>214.143</td>
<td>63.378</td>
<td>3.4</td>
</tr>
<tr>
<td>21</td>
<td>159.106</td>
<td>65.987</td>
<td>2.4</td>
</tr>
<tr>
<td>22</td>
<td>166.372</td>
<td>66.271</td>
<td>2.5</td>
</tr>
<tr>
<td>23</td>
<td>225.816</td>
<td>92.194</td>
<td>2.4</td>
</tr>
<tr>
<td>24</td>
<td>234.661</td>
<td>123.920</td>
<td>1.9</td>
</tr>
<tr>
<td>25</td>
<td>176.237</td>
<td>66.980</td>
<td>2.6</td>
</tr>
<tr>
<td>26</td>
<td>295.107</td>
<td>194.457</td>
<td>1.5</td>
</tr>
</tbody>
</table>

Avg. 4.8 ± 5.20
Table 12
Average time (in seconds) for computing the probability of queries with the reasoner TRILL in its two versions in Test 5. The column “ratio” shows the ratio “prev./hier.” for the execution time. Their average ± the standard deviation is also shown.

<table>
<thead>
<tr>
<th>NPA</th>
<th>prev.</th>
<th>hier.</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>260.925</td>
<td>2.078</td>
<td>125.5</td>
</tr>
<tr>
<td>10</td>
<td>237.148</td>
<td>2.087</td>
<td>113.7</td>
</tr>
<tr>
<td>11</td>
<td>311.544</td>
<td>7.801</td>
<td>39.9</td>
</tr>
<tr>
<td>12</td>
<td>352.997</td>
<td>6.290</td>
<td>56.1</td>
</tr>
<tr>
<td>13</td>
<td>341.674</td>
<td>5.188</td>
<td>65.9</td>
</tr>
<tr>
<td>14</td>
<td>321.559</td>
<td>8.260</td>
<td>38.9</td>
</tr>
<tr>
<td>15</td>
<td>350.364</td>
<td>7.020</td>
<td>49.9</td>
</tr>
<tr>
<td>16</td>
<td>339.781</td>
<td>6.261</td>
<td>54.3</td>
</tr>
<tr>
<td>17</td>
<td>461.894</td>
<td>27.983</td>
<td>16.5</td>
</tr>
<tr>
<td>18</td>
<td>342.495</td>
<td>24.239</td>
<td>14.1</td>
</tr>
<tr>
<td>19</td>
<td>436.081</td>
<td>36.541</td>
<td>11.9</td>
</tr>
<tr>
<td>20</td>
<td>464.981</td>
<td>31.602</td>
<td>14.7</td>
</tr>
<tr>
<td>21</td>
<td>455.583</td>
<td>27.010</td>
<td>16.9</td>
</tr>
<tr>
<td>22</td>
<td>404.276</td>
<td>24.321</td>
<td>16.6</td>
</tr>
<tr>
<td>23</td>
<td>469.687</td>
<td>56.111</td>
<td>8.4</td>
</tr>
<tr>
<td>24</td>
<td>457.938</td>
<td>91.312</td>
<td>5.0</td>
</tr>
<tr>
<td>25</td>
<td>455.783</td>
<td>48.476</td>
<td>9.4</td>
</tr>
<tr>
<td>26</td>
<td>570.789</td>
<td>128.392</td>
<td>4.4</td>
</tr>
</tbody>
</table>

Avg. 36.8 ± 35.93

Table 13
Average time (in seconds) for computing the probability of queries with the reasoner TORNADO in its two versions in Test 5. The column “ratio” shows the ratio “prev./hier.” for the execution time. Their average ± the standard deviation is also shown.

<table>
<thead>
<tr>
<th>NPA</th>
<th>prev.</th>
<th>hier.</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>1.204</td>
<td>0.071</td>
<td>17.0</td>
</tr>
<tr>
<td>10</td>
<td>1.044</td>
<td>0.067</td>
<td>15.6</td>
</tr>
<tr>
<td>11</td>
<td>1.515</td>
<td>0.085</td>
<td>17.8</td>
</tr>
<tr>
<td>12</td>
<td>1.573</td>
<td>0.086</td>
<td>18.2</td>
</tr>
<tr>
<td>13</td>
<td>1.604</td>
<td>0.087</td>
<td>18.4</td>
</tr>
<tr>
<td>14</td>
<td>1.624</td>
<td>0.084</td>
<td>19.4</td>
</tr>
<tr>
<td>15</td>
<td>1.645</td>
<td>0.088</td>
<td>18.7</td>
</tr>
<tr>
<td>16</td>
<td>1.632</td>
<td>0.085</td>
<td>19.1</td>
</tr>
<tr>
<td>17</td>
<td>2.126</td>
<td>0.109</td>
<td>19.5</td>
</tr>
<tr>
<td>18</td>
<td>1.896</td>
<td>0.101</td>
<td>18.8</td>
</tr>
<tr>
<td>19</td>
<td>2.040</td>
<td>0.103</td>
<td>19.8</td>
</tr>
<tr>
<td>20</td>
<td>2.237</td>
<td>0.113</td>
<td>19.9</td>
</tr>
<tr>
<td>21</td>
<td>2.116</td>
<td>0.107</td>
<td>19.8</td>
</tr>
<tr>
<td>22</td>
<td>1.929</td>
<td>0.100</td>
<td>19.2</td>
</tr>
<tr>
<td>23</td>
<td>2.210</td>
<td>0.114</td>
<td>19.4</td>
</tr>
<tr>
<td>24</td>
<td>2.161</td>
<td>0.111</td>
<td>19.4</td>
</tr>
<tr>
<td>25</td>
<td>2.118</td>
<td>0.108</td>
<td>19.7</td>
</tr>
<tr>
<td>26</td>
<td>2.812</td>
<td>0.136</td>
<td>20.7</td>
</tr>
</tbody>
</table>

Avg. 18.9 ± 1.19
7. Conclusions

In this paper we presented two extensions implemented in the TRILL framework, which contains three systems for reasoning on DISPONTE KBs: TRILL, able to collect the set of all the justifications and compute the probability of queries, TRILL\(_P\), which implements in Prolog the tableau algorithm defined in [15, 16] for returning the pinpointing formula instead of the set of justifications, and TORNADO, which is similar to TRILL\(_P\) but instead of building a pinpointing formula and translating it to a BDD in two different phases, it builds the BDD while building the tableau.

The first extension allows the reasoners to collect information about the KB and build the hierarchy of the concepts in order to quickly find connections between them during the expansion of the tableau. The second optimization avoids testing assertions that have not been changed since the last check during the expansion of the tableau, in order to not perform useless tests on assertions.

The extensive experimentation performed shows that the presented optimizations can significantly speed up both regular and probabilistic queries in many cases. The results show that in exchange for a small overhead, visible especially w.r.t. simpler KBs or when the time needed for the inference is under one second, the optimizations allows to achieve a speed up of up to more than 125:1.

References


[30] B. Beckert and J. Posegga, leanTAP: Lean Tableau-based De-


