A Divide and Conquer Approach for Parallel Classification of OWL Ontologies

(submitted to special issue on Web Reasoning and Rule Systems)

Kejia Wu and Volker Haarslev

Department of Computer Science and Software Engineering
Concordia University, Montreal, Canada

Abstract. Description Logic (DL) describes knowledge using entities and relationships between them, and TBox classification is a core DL reasoning service. Over more than two decades many research efforts have been devoted to optimizing TBox classification. Those classification optimization algorithms have shown their pragmatic effectiveness for sequential processing. However, as concurrent computing becomes widely available, new classification algorithms that are well suited to parallelization need to be developed. This need is further supported by the observation that most available Web Ontology Language (OWL) reasoners, which are usually based on tableau reasoning, can only utilize a single processor. Such an inadequacy often leads to frustrated users working in ontology development, especially if their ontologies are complex and require long processing times. In this paper we present a novel algorithm that uses a *divide and conquer* strategy for parallelizing OWL TBox classification, a key reasoning task. We discuss some interesting properties of our algorithm, for example, its suitability for distributed reasoning, and present an empirical study using a set of benchmark ontologies, where a speedup of up to a factor of four has been observed when using eight workers in parallel.

1 Introduction

Due to the semantic web, a multitude of OWL ontologies are emerging. Quite a few ontologies are huge and contain hundreds of thousands of concepts. Although some of these huge ontologies fit into one of OWL's three tractable profiles, e.g., the well known Snomed ontology is in the $\mathcal{EL}$ profile, there still exist a variety of other OWL ontologies that make full use of OWL DL and require long processing times, even when highly optimized OWL reasoners are employed. Moreover, although most of the huge ontologies are currently restricted to one of the tractable profiles in order to ensure fast processing, it is foreseeable that some of them will require an expressivity that is outside of the tractable OWL profiles.

Almost all well-known reasoners employ a so-called *top-search* & *bottom-search* algorithm to classify ontologies [19]. This algorithm makes use of told subsumption relationships to prune a lot of costly subsumption tests. Concepts are incrementally inserted into a subsumption hierarchy at their most specific positions. This method works efficiently in practical reasoning, and a number of variants proposed on the basis of the original version provide optimizations to some extent [2, 9]. However, only in recent years efforts appeared to investigate parallelization of top-search & bottom-search in order to gain a more scalable performance [1].

The research presented in this paper is targeted to provide better OWL reasoning scalability by making efficient use of modern hardware architectures such as multi-processor/core computers. This becomes more important in the case of ontologies that require long processing times although highly optimized OWL reasoners are already used. We consider our research an important basis for the design of next-generation OWL reasoners that can efficiently work in a parallel/concurrent or distributed context using modern hardware. One of the major obstacles that needs to be addressed in the design of corresponding algorithms and architectures is the overhead introduced by concurrent computing and its impact on scalability.

Heavily shared data as well as related communication costs always indicate an inefficient performance in parallel environments. Canonical Description Logic (DL) reasoning algorithms, which form the basis of OWL reasoning, deal with a problem domain as a whole, which generally produces monolithic data and makes it hard to parallelize employed algorithms. In order to achieve effective parallelized DL reasoning novel methods need to be developed that process data as independently as possible.

Traditional *divide and conquer* algorithms split problems into independent sub-problems before solving them under the premise that not much communication among the divisions is needed when independently solving the sub-problems, so shared data is excluded to a great extent. Therefore, divide and conquer algorithms are in principle suitable for concurrent computing, including shared-memory parallelization and non-shared-memory distributed systems.

Furthermore, recently research on *ontology partitioning* has been proposed and investigated for dealing with monolithic ontologies. Some research results, e.g. ontology modularization [10], can be used for decreasing the scale of an ontology-reasoning problem. Then, reasoning over a set of sub-ontologies can be executed in parallel. However, there is still a solution needed to reassemble sub-ontologies together. The algorithms presented in this paper can also serve as a solution for this problem.

This article is a revised and extended version of [30]. In the remaining sections, we present our divide and conquer algorithm, which uses a heuristic partitioning and a merge-based classification scheme. We report on our conducted experiments and their evaluation, and discuss related research.

2 Preliminaries

Our work is essentially about DL reasoning, TBox classification specifically, so some background knowledge is presented in this section. For a more detailed background on DLs, DL reasoning, and semantic web we refer to [3] and [13].

DL is used to represent knowledge. *Concepts* and *roles* are elements constructing DL expressions. The former conceptualize knowledge domain instances, and the latter describe binary relations between domain instances. DL axioms are constructed by associating the two essentials via a set of connectives, *concept constructors* and *role constructors*. For example, the syntax for $\mathcal{AL}$ is defined as follows [3]:

$$
\begin{array}{llll}
C, D \longrightarrow & A & \mid & \text{(* atomic concept *)} \\
& \top & \mid & \text{(* universal concept *)} \\
& \bot & \mid & \text{(* bottom concept *)} \\
& \neg A & \mid & \text{(* atomic negation *)} \\
& C \sqcap D & \mid & \text{(* intersection *)} \\
& \forall R.C & \mid & \text{(* value restriction *)} \\
& \exists R.\top & \mid & \text{(* limited existential quantification *)}
\end{array}
$$

In the productions, A corresponds to a *concept name*, C or D to either a compound concept or a concept name, and R to a role name.

Generally, a DL language's semantics is described by a model-theoretical *interpretation*. In $\mathcal{AL}$, such an interpretation, $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$, consists of a non-empty set of individuals ($\Delta^{\mathcal{I}}$) and a function ($\cdot^{\mathcal{I}}$) such that:

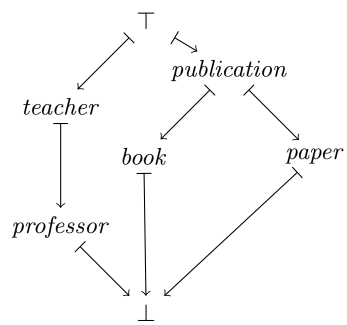
$$
\begin{aligned}
C^{\mathcal{I}} &\subseteq \Delta^{\mathcal{I}} \\
R^{\mathcal{I}} &\subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \\
\top^{\mathcal{I}} &= \Delta^{\mathcal{I}} \\
\bot^{\mathcal{I}} &= \emptyset \\
(\neg C)^{\mathcal{I}} &= \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}} \\
(C \sqcap D)^{\mathcal{I}} &= C^{\mathcal{I}} \cap D^{\mathcal{I}} \\
(\exists R.\top)^{\mathcal{I}} &= \{x \in \Delta^{\mathcal{I}} \mid \exists y(\langle x, y \rangle \in R^{\mathcal{I}})\} \\
(\forall R.C)^{\mathcal{I}} &= \{x \in \Delta^{\mathcal{I}} \mid \forall y(\langle x, y \rangle \in R^{\mathcal{I}} \implies y \in C^{\mathcal{I}})\}
\end{aligned}
$$

An interpretation $\mathcal{I}$ satisfies an axiom $C \sqsubseteq D$ iff $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$. An axiom $C \equiv D$ is considered as an abbreviations for the set of axioms $\{C \sqsubseteq D, D \sqsubseteq C\}$. An assertion $C(x)$ is satisfied by $\mathcal{I}$ if $x^{\mathcal{I}} \in C^{\mathcal{I}}$, (xRy) if $\langle x^{\mathcal{I}}, y^{\mathcal{I}} \rangle \in R^{\mathcal{I}}$, $x \doteq y$ if $x^{\mathcal{I}} = y^{\mathcal{I}}$, and $x \neq y$ if $x^{\mathcal{I}} \neq y^{\mathcal{I}}$. An overall introduction to DL syntax, semantics, notation, and extensions can be found in the appendix of [3].

A set of reasoning tasks are executed on DL knowledge bases, such as satisfiability, subsumption, and classification. Among them, TBox *classification* plays an important role. TBox classification generates hierarchical taxonomies. A TBox classification algorithm computes all subsumptions between concept names ($A \sqsubseteq ?B$) that are entailed in a TBox and inserts concepts into a hierarchical structure. A result of classification can be illustrated by a directed graph with $\top$ as the root and $\bot$ as the unique leaf, which represent the most general concept and the most specific concept respectively. Figure 1 shows a TBox *classification* example. In the graph, each node subsumes its descendant node(s), and all paths to a node from $\top$ contain its subsumer nodes. Therefore, all concept subsumptions related information can be extracted from the classified taxonomy.

$\mathcal{T}$: $professor \sqsubseteq teacher$
$book \sqsubseteq publication$
$paper \sqsubseteq publication$

(a) The TBox given.



(b) The classified terminology hierarchy.

Fig. 1. An example on *classification*.

However, it is known that TBox *classification* can be a costly computation. The naive brute-force classification method executes subsumption tests over all elements of $\{\langle A_i, A_j\rangle \mid A_i^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}, A_j^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}, 0 \leq i \leq n, 0 \leq j \leq n\}$. Although the brute-force method needs only n^2 subsumption tests for a TBox of n concepts, it is generally very expensive due to the costly subsumption testing. However, in Section 7.1 we briefly report on an earlier experiment for ontologies where we parallelized this brute-force scheme and could demonstrate excellent speedup factors.

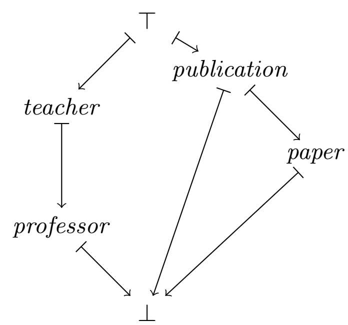
A huge computing expense lies in concept subsumption tests, so the most prominent work on classification optimization focuses on making use of the *reflexive transitive closure* of subsumptions in order to avoid costly subsumption tests—instead of checking subsumption for every pair of concepts in a brute-force way, a large number of subsumption relationships can be figured out by told subsumptions and non-subsumptions directly, and the *top-search & bottom-search* algorithm is the corner stone for such an optimization [19]. The *top-search & bottom-search* algorithm utilizes told subsumption relationships to avoid costly subsumption tests. For example, given a TBox and a partially classified terminology hierarchy shown by Figure 2, when searching for the most specific parent concept of $book$, it is unnecessary to test whether $book \sqsubseteq ?\, professor$ if $book \not\sqsubseteq teacher$ is already known. Our work shows that this technique can be extended to work in parallel.

3 A Parallelized Merge Classification Algorithm

In this section, we present an algorithm for classifying DL ontologies. Part of the algorithm is based on standard top- and bottom-search techniques to incrementally construct the classification hierarchy (e.g., see [2]). Due to the symmetry between *top-down* ($\top\_search$) and *bottom-up* ($\bot\_search$) search, we only present the first one. In the pseudo code, we use the following notational conventions: Δ_i, Δ_α, and Δ_β designate sub-domains that are divided from Δ; we consider a subsumption hierarchy as a partial order over Δ, denoted as $\leq$, a subsumption relationship where C is subsumed by D ($C \sqsubseteq D$) is expressed by $C \leq D$ or by $\langle C, D\rangle \in \leq$, and $\leq_i$, $\leq_\alpha$, and $\leq_\beta$ are subsump-

$$\mathcal{T} : \mathit{professor} \sqsubseteq \mathit{teacher}$$
$$\mathit{book} \sqsubseteq \mathit{publication}$$
$$\mathit{paper} \sqsubseteq \mathit{publication}$$

(a) The TBox given.



(b) The partially classified terminology hierarchy.

Fig. 2. An example on top- and bottom-search based classification.

tion hierarchies over Δ_i, Δ_α, and Δ_β, respectively; in a subsumption hierarchy over Δ, $C \prec D$ designates $C \sqsubseteq D$ and there does not exist a named concept E such that $C \leq E$ and $E \leq D$; $\prec_i$, $\prec_\alpha$ and $\prec_\beta$ are similar notations defined over Δ_i, Δ_α, and Δ_β, respectively.

Our merge-classification algorithm classifies a taxonomy by calculating its divided sub-domains and then by merging the classified sub-taxonomies together. The algorithm makes use of two facts: (i) If it holds that $B \leq A$, then the subsumption relationships between B's descendants and A's ancestors are determined; (ii) if it is known that $B \not\leq A$, the subsumption relationships between B's descendants and A's ancestors are undetermined. The canonical DL classification algorithm, top-search & bottom-search, is modified and integrated into the merge-classification. The algorithm consists of two stages: divide and conquering, and combining. Algorithm 1 shows the main part of our parallelized DL classification procedure. The keyword *spawn* indicates that its following calculation must be executed in parallel, either creating a new thread in a shared-memory context or generating a new process or session in a non-shared-memory context. The keyword *sync* always follows *spawn* and suspends the current calculation procedure until all calculations invoked by *spawn* have returned.

The domain Δ is divided into smaller partitions in the first stage. Then, classification computations are executed over each sub-domain Δ_i. A classified sub-terminology $\leq_i$ is inferred over Δ_i. The procedure *classify* is used by Algorithm 1 and is a general reasoning function that calls Algorithm 2. It is not shown in this paper. This divide and conquering operations can progress in parallel.

Classified sub-terminologies are to be merged in the combining stage. Told subsumption relationships are utilized in the merging process. Algorithm 2 outlines the master procedure, and the slave procedure is addressed by Algorithms 3, 4, 5, and 6.

3.1 Divide and Conquer Phase

The first task is to divide the universe, Δ, into sub-domains. Without loss of generality, Δ only focuses on *significant* concepts, i.e., concept names or atomic concepts, that are

Algorithm 1: $\kappa(\Delta_i)$

input : The sub-domain Δ_i
output : The subsumption hierarchy classified over Δ_i

begin
  if $divided\_enough?(\Delta_i)$ **then**
    return $classify(\Delta_i)$;
  else
    $\langle \Delta_\alpha, \Delta_\beta \rangle \leftarrow divide(\Delta_i)$;
    $\leq_\alpha \leftarrow$ **spawn** $\kappa(\Delta_\alpha)$;
    $\leq_\beta \leftarrow \kappa(\Delta_\beta)$;
    sync;
    return $\mu(\leq_\alpha, \leq_\beta)$;
  end if
end

Algorithm 2: $\mu(\leq_\alpha, \leq_\beta)$

input : The master subsumption hierarchy $\leq_\alpha$
The subsumption hierarchy $\leq_\beta$ to be merged into $\leq_\alpha$
output : The subsumption hierarchy resulting from merging $\leq_\alpha$ over $\leq_\beta$

begin
  $\top_\alpha \leftarrow \textit{select-top}(\leq_\alpha)$;
  $\top_\beta \leftarrow \textit{select-top}(\leq_\beta)$;
  $\bot_\alpha \leftarrow \textit{select-bottom}(\leq_\alpha)$;
  $\bot_\beta \leftarrow \textit{select-bottom}(\leq_\beta)$;
  $\leq_\alpha \leftarrow \top\_merge(\top_\alpha, \top_\beta, \leq_\alpha, \leq_\beta)$;
  $\leq_i \leftarrow \bot\_merge(\bot_\alpha, \bot_\beta, \leq_\alpha, \leq_\beta)$;
  return $\leq_i$;
end

normally declared explicitly in some ontology $\mathcal{O}$, and *intermediate* concepts, i.e., non-significant ones, only play a role in subsumption tests. Each sub-domain is classified independently. The *divide* operation can be naively implemented as an even partitioning over Δ, or by more sophisticated clustering techniques such as *heuristic partitioning* that may result in a better performance, as presented in Section 5. The conquering operation can be any standard DL classification method. We first present the most popular classification methods, top-search (Algorithm 3) and bottom-search (omitted here).

The DL classification procedure determines the most specific super- and the most general sub-concepts of each significant concept in Δ. The classified concept hierarchy is a partial order, $\leq$, over Δ. $\top\_search$ recursively calculates a concept's *intermediate* predecessors, i.e., intermediate immediate ancestors, as a relation $\prec_i$ over $\leq_i$.

3.2 Combining Phase

The independently classified sub-terminologies must be merged together in the combining phase. The original top-search (Algorithm 3) (and bottom-search) have been

Algorithm 3: $\top\_search(C, D, \leq_i)$

input : C: the new concept to be classified
D: the current concept with $\langle D, \top \rangle \in \leq_i$
$\leq_i$: the subsumption hierarchy
output : The set of parents of C: $\{p \mid \langle C, p \rangle \in \leq_i\}$.

begin
  $mark\_visited(D)$;
  $green \leftarrow \emptyset$;
  forall the $d \in \{d \mid \langle d, D \rangle \in \prec_i\}$ **do** /* collect all children of D that subsume C */
    if $\leq ?(C, d)$ **then**
      $green \leftarrow green \cup \{d\}$;
    end if
  end forall
  $box \leftarrow \emptyset$;
  if $green = \emptyset$ **then**
    $box \leftarrow \{D\}$;
  else
    forall the $g \in green$ **do**
      if $\neg marked\_visited?(g)$ **then**
        $box \leftarrow box \cup \top\_search(C, g, \leq_i)$; /* recursively test whether C is subsumed by the descendants of g */
      end if
    end forall
  end if
  return box; /* return the parents of C */
end

modified to merge two sub-terminologies $\leq_\alpha$ and $\leq_\beta$. The basic idea is to iterate over Δ_β and to use top-search (and bottom-search) to insert each element of Δ_β into $\leq_\alpha$, as shown in Algorithm 4.

However, this method does not make use of so-called told subsumption (and non-subsumption) information contained in the merged sub-terminology $\leq_\beta$. For example, it is unnecessary to test $\leq ?(B_2, A_1)$ with sophisticated reasoning algorithms when we know $B_2 \leq B_1$ and $B_1 \leq A_1$, given that A_1 occurs in Δ_α and B_1, B_2 occur in Δ_β.

Therefore, we designed a novel algorithm in order to utilize the properties addressed by Propositions 1 to 8. The calculation starts with top-merge (Algorithm 5), which uses a modified top-search algorithm (Algorithm 6). This pair of procedures finds the most specific subsumers in the master sub-terminology $\leq_\alpha$ for every concept from the sub-terminology $\leq_\beta$ that is being merged into $\leq_\alpha$.

Proposition 1. *When merging sub-terminology $\leq_\beta$ into $\leq_\alpha$, if $\langle B, A \rangle \in \prec_i$ is found in top-search, $\langle A, \top \rangle \in \leq_\alpha$ and $\langle B, \top \rangle \in \leq_\beta$, then for $\forall b_j \in \{b \mid \langle b, B \rangle \in \leq_\beta\}$ and $\forall a_k \in \{a \mid \langle A, a \rangle \in \leq_\alpha\} \cup \{A\}$ it follows that $b_j \leq a_k$.*

Proof. Figure 3 shows the case, where $\{a_1, \ldots, a_m\}$ is the set of parents of A and $\{b_1, \ldots, b_n\}$ the set of children of B. It is easy to see that $b_j \leq a_k$ due to the transitivity

Algorithm 4: $\top\_merge^{-}(A, B, \leq_\alpha, \leq_\beta)$

input : A: the current concept of the master subsumption hierarchy, i.e. $\langle A, \top\rangle \in \leq_\alpha$
B: the new concept from the merged subsumption hierarchy, i.e. $\langle B, \top\rangle \in \leq_\beta$
$\leq_\alpha$: the master subsumption hierarchy
$\leq_\beta$: the subsumption hierarchy to be merged into $\leq_\alpha$
output : The merged subsumption hierarchy $\leq_\alpha$ over $\leq_\beta$.

begin
  $parents \leftarrow \top\_search(B, A, \leq_\alpha)$;
  forall the $a \in parents$ **do**
    $\leq_\alpha \leftarrow \leq_\alpha \cup \langle B, a\rangle$; /* insert B into $\leq_\alpha$ */
    forall the $b \in \{b \mid \langle b, B\rangle \in \prec_\beta\}$ **do** /* insert children of B (in $\leq_\beta$) below parents of B (in $\leq_\alpha$) */
      $\leq_\alpha \leftarrow \top\_merge^{-}(a, b, \leq_\alpha, \leq_\beta)$;
    end forall
  end forall
  return $\leq_\alpha$;
end

Algorithm 5: $\top\_merge(A, B, \leq_\alpha, \leq_\beta)$

input : A: the current concept of the master subsumption hierarchy, i.e. $\langle A, \top\rangle \in \leq_\alpha$
B: the new concept of the merged subsumption hierarchy, i.e. $\langle B, \top\rangle \in \leq_\beta$
$\leq_\alpha$: the master subsumption hierarchy
$\leq_\beta$: the subsumption hierarchy to be merged into $\leq_\alpha$
output : the merged subsumption hierarchy $\leq_\alpha$ over $\leq_\beta$

begin
  $parents \leftarrow \top\_search^{*}(B, A, \leq_\beta, \leq_\alpha)$;
  forall the $a \in parents$ **do**
    $\leq_\alpha \leftarrow \leq_\alpha \cup \langle B, a\rangle$;
    forall the $b \in \{b \mid \langle b, B\rangle \in \prec_\beta\}$ **do**
      $\leq_\alpha \leftarrow \top\_merge(a, b, \leq_\alpha, \leq_\beta)$;
    end forall
  end forall
  return $\leq_\alpha$;
end

Algorithm 6: $\top\_search^*(C, D, \leq_\beta, \leq_\alpha)$

input : C: the new concept to be inserted into $\leq_\alpha$, and $\langle C, \top \rangle \in \leq_\beta$
D: the current concept, and $\langle D, \top \rangle \in \leq_\alpha$
$\leq_\beta$: the subsumption hierarchy to be merged into $\leq_\alpha$
$\leq_\alpha$: the master subsumption hierarchy

output : The set of parents of C: $\{p \mid \langle C, p \rangle \in \leq_\alpha\}$

```
begin
    $mark\_visited(D)$;
    $green \leftarrow \emptyset$; /* subsumers of $C$ that are from $\leq_\alpha$ */
    $red \leftarrow \emptyset$; /* non-subsumers of $C$ that are children of $D$ */
    forall the $d \in \{d \mid \langle d, D \rangle \in \prec_\alpha \wedge \langle d, \top \rangle \notin \leq_\beta\}$ do
        if $\leq ?(C, d)$ then
            $green \leftarrow green \cup \{d\}$;
        else
            $red \leftarrow red \cup \{d\}$;
        end if
    end forall
    $box \leftarrow \emptyset$;
    if $green = \emptyset$ then
        if $\leq ?(C, D)$ then
            $box \leftarrow \{D\}$;
        else
            $red \leftarrow \{D\}$;
        end if
    else
        forall the $g \in green$ do
            if $\neg marked\_visited?(g)$ then
                $box \leftarrow box \cup \top\_search^*(C, g, \leq_\beta, \leq_\alpha)$;
            end if
        end forall
    end if
    forall the $r \in red$ do
        forall the $c \in \{c \mid \langle c, C \rangle \in \prec_i\}$ do
            $\leq_\alpha \leftarrow \top\_merge(r, c, \leq_\alpha, \leq_\beta)$;
        end forall
    end forall
    return $box$;
end
```

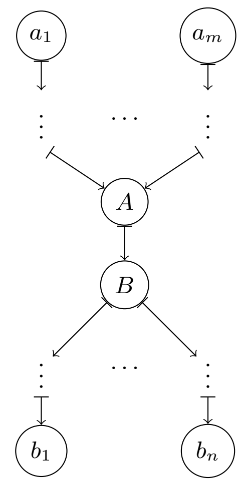


Fig. 3. $\langle B, A\rangle \in \prec_i \implies b_j \sqsubseteq a_k$.

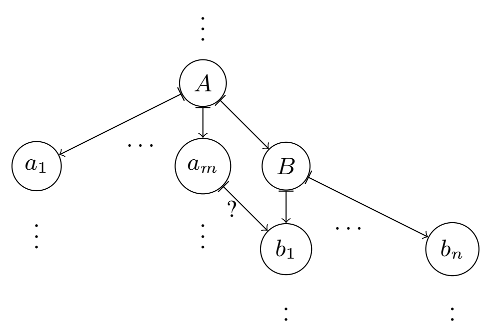


Fig. 4. $\langle B, A\rangle \in \prec_i$: $b_j \sqsubseteq ?\, a_k$.

of the subsumption relationship. From our premise we know that $b_j \leq B$, $B \leq A$ and $A \leq a_k$, therefore it holds that $b_j \leq a_k$ for all j, k. ■

Proposition 2. *When merging sub-terminology $\leq_\beta$ into $\leq_\alpha$, if $\langle B, A\rangle \in \prec_i$ is found in top-search, $\langle A, \top\rangle \in \leq_\alpha$ and $\langle B, \top\rangle \in \leq_\beta$, then for $\forall b_j \in \{b \mid \langle b, B\rangle \in \prec_\beta \wedge b \neq B\}$ and $\forall a_k \in \{a \mid \langle a, A\rangle \in \prec_\alpha \wedge a_j \neq A\}$ it is still necessary to calculate whether $b_j \leq a_k$.*

Proof. Figure 4 shows the case, where $\{a_1, \ldots, a_m\} = \{a \mid \langle a, A\rangle \in \prec_\alpha \wedge a \neq A\}$ and $\{b_1, \ldots, b_n\} = \{b \mid \langle b, B\rangle \in \prec_\beta \wedge b \neq B\}$. We know that $B^{\mathcal{I}} \subseteq A^{\mathcal{I}}$ or $B^{\mathcal{I}} \cap (\neg A)^{\mathcal{I}} = \emptyset$ and $b_j^{\mathcal{I}} \subseteq B^{\mathcal{I}}$ leads to $b_j^{\mathcal{I}} \cap (\neg A)^{\mathcal{I}} = \emptyset$ but since $(\neg a_k)^{\mathcal{I}} \supseteq (\neg A)^{\mathcal{I}}$ it is unknown for all j, k whether $b_j^{\mathcal{I}} \cap (\neg a_k)^{\mathcal{I}}$ is always empty or always not empty. ■

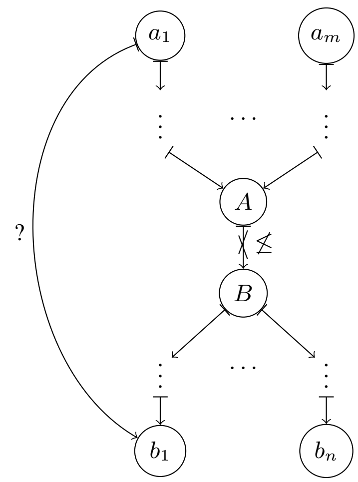


Fig. 5. $B \not\leq A : b_i \sqsubseteq ?\, a_j$.

Proposition 3. *When merging sub-terminology $\leq_\beta$ into $\leq_\alpha$, if $B \not\leq A$ is found in top-search, $\langle A, \top\rangle \in \leq_\alpha$ and $\langle B, \top\rangle \in \leq_\beta$, then for $\forall b_j \in \{b \mid \langle b, B\rangle \in \leq_\beta \wedge b \neq B\}$ and $\forall a_k \in \{a \mid \langle a, A\rangle \in \leq_\alpha\} \cup \{A\}$ it is necessary to calculate whether $b_j \leq a_k$.*

Proof. Figure 5 shows the case, where $\{b_1, \ldots, b_n\} = \{b \mid \langle b, B\rangle \in \leq_\beta \wedge b \neq B\}$ and $\{a_1, \ldots, a_m\} = \{a \mid \langle a, A\rangle \in \leq_\alpha\}$. We know that $B^\mathcal{I} \nsubseteq A^\mathcal{I}$ or $B^\mathcal{I} \cap (\neg A)^\mathcal{I} \neq \emptyset$, $b_j^\mathcal{I} \cap (\neg B)^\mathcal{I} = \emptyset$, $A^\mathcal{I} \cap (\neg a_k)^\mathcal{I} = \emptyset$. Although $B^\mathcal{I} \cap (\neg A)^\mathcal{I} \neq \emptyset$ it is unknown whether $b_j^\mathcal{I} \cap (\neg a_k)^\mathcal{I}$ is empty or not because $b_j^\mathcal{I} \subseteq B^\mathcal{I}$ and $(\neg A)^\mathcal{I} \supseteq (\neg a_k)^\mathcal{I}$ and thus neither $b_j \sqsubseteq a_k$ nor $b_j \not\sqsubseteq a_k$ is enforced for all j, k. ■

Proposition 4. *When merging sub-terminology $\leq_\beta$ into $\leq_\alpha$, if $B \not\leq A$ is found in top-search, $\langle A, \top\rangle \in \leq_\alpha$ and $\langle B, \top\rangle \in \leq_\beta$, then for $\forall b_j \in \{b \mid \langle B, b\rangle \in \leq_\beta\} \cup \{B\}$ and $\forall a_k \in \{a \mid \langle a, A\rangle \in \leq_\alpha\} \cup \{A\}$ it follows that $b_j \not\leq a_k$.*

Proof. Figure 6 illustrates the case, where $\{a_1, \ldots, a_m\} = \{a \mid \langle a, A\rangle \in \leq_\alpha\}$ and $\{b_1, \ldots, b_n\} = \{b \mid \langle B, b\rangle \in \leq_\beta\}$. We prove the contrapositive: $b_j \leq a_k \implies B \leq A$. This follows due to the transitivity of the subsumption relationship. From the premise we know that $B \leq b_j$, $b_j \leq a_k$, $a_k \leq A$; thus we have $B \leq A$. ■

Similarly, we present the following propositions for bottom-search. Due to the symmetry between top- and bottom-search the proofs for Propositions 5 to 8 are very similar to the proofs of Propositions 1 to 4 and are omitted.

Proposition 5. *When merging sub-terminology $\leq_\beta$ into $\leq_\alpha$, if $\langle A, B\rangle \in \prec_i$ is found in bottom-search, $\langle \bot, A\rangle \in \leq_\alpha$ and $\langle \bot, B\rangle \in \leq_\beta$, then for $\forall b_j \in \{b \mid \langle B, b\rangle \in \leq_\beta\}$ and $\forall a_k \in \{a \mid \langle a, A\rangle \in \leq_\alpha\} \cup \{A\}$ it follows that $a_k \leq b_j$.*

Proposition 6. *When merging sub-terminology $\leq_\beta$ into $\leq_\alpha$, if $\langle A, B\rangle \in \prec_i$ is found in bottom-search, $\langle \bot, A\rangle \in \leq_\alpha$ and $\langle \bot, B\rangle \in \leq_\beta$, then for $\forall b_j \in \{b \mid \langle B, b\rangle \in \prec_\beta \wedge b \neq B\}$ and $\forall a_k \in \{a \mid \langle A, a\rangle \in \prec_\alpha \wedge a \neq A\}$ it is necessary to calculate whether $a_k \leq b_j$.*

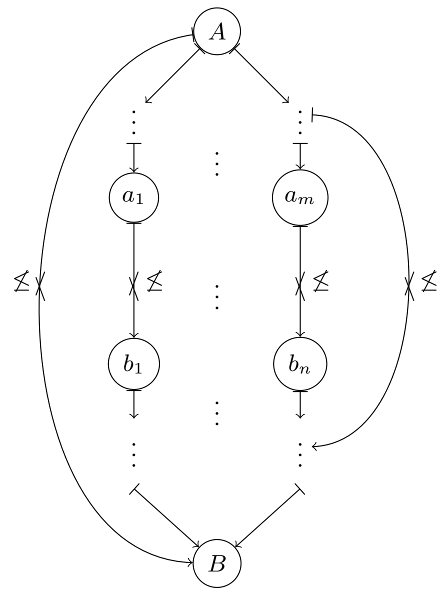


Fig. 6. $B \not\leq A \implies b_i \not\leq a_j$.

Proposition 7. *When merging sub-terminology $\leq_\beta$ into $\leq_\alpha$, if $A \not\leq B$ is found in bottom-search, $\langle \bot, A\rangle \in \leq_\alpha$ and $\langle \bot, B\rangle \in \leq_\beta$, then for $\forall b_j \in \{b \mid \langle B, b\rangle \in \leq_\beta \wedge b \neq B\}$ and $\forall a_k \in \{a \mid \langle A, a\rangle \in \leq_\alpha\} \cup \{A\}$ it is necessary to calculate whether $a_k \leq b_j$.*

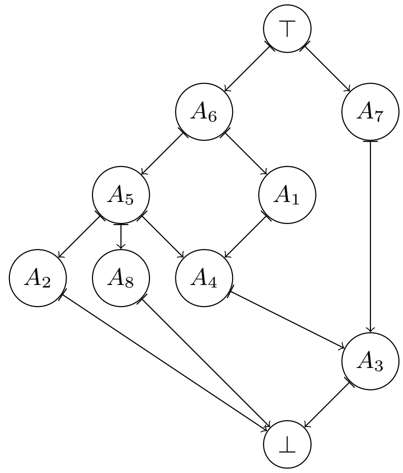
Proposition 8. *When merging sub-terminology $\leq_\beta$ into $\leq_\alpha$, if $A \not\leq B$ is found in top-search, $\langle \bot, A\rangle \in \leq_\alpha$ and $\langle \bot, B\rangle \in \leq_\beta$, then for $\forall b_j \in \{b \mid \langle b, B\rangle \in \leq_\beta\} \cup \{B\}$ and $\forall a_k \in \{a \mid \langle A, a\rangle \in \leq_\alpha\} \cup \{A\}$ it follows that $a_k \not\leq b_j$.*

When merging a concept B, $\langle B, \top\rangle \in \leq_\beta$, the top-merge algorithm first finds for B the most specific position in the master sub-terminology $\leq_\alpha$ by means of *top-down* search. When such a most specific super-concept is found, this concept and all its super-concepts are naturally super-concepts of every sub-concept of B in the sub-terminology $\leq_\beta$, as is stated by Proposition 1. However, this newly found predecessor of B may not be necessarily a predecessor of some descendant of B in $\leq_\beta$. Therefore, the algorithm continues to find the most specific positions for all sub-concepts of B in $\leq_\beta$ according to Proposition 2. Algorithm 5 addresses this procedure.

Non-subsumption information can be told in the top-merge phase. Top-down search employed by top-merge must do subsumption tests somehow. In a canonical top-search procedure, as indicated by Algorithm 3, the branch search is stopped at this point. However, the conclusion that a merged concept B, $\langle B, \top\rangle \in \leq_\beta$, is not subsumed by a concept A, $\langle A, \top\rangle \in \leq_\alpha$, does not rule out the possibility of $b_j \leq A$ with $b_j \in \{b \mid \langle b, B\rangle \in \prec_\beta\}$, which is not required in traditional top-search and may be abound in the top-merge procedure, and therefore must be followed by determining whether $b_j \leq A$. Otherwise, the algorithm is incomplete. Proposition 3 presents this

$\mathcal{T}$:

$A_6 \sqsubseteq \top$	$A_4 \sqsubseteq A_1$
$A_7 \sqsubseteq \top$	$A_3 \sqsubseteq A_7$
$A_5 \sqsubseteq A_6$	$A_3 \sqsubseteq A_4$
$A_1 \sqsubseteq A_6$	$\bot \sqsubseteq A_2$
$A_2 \sqsubseteq A_5$	$\bot \sqsubseteq A_8$
$A_8 \sqsubseteq A_5$	$\bot \sqsubseteq A_3$
$A_4 \sqsubseteq A_5$	



(a) The TBox given. (b) The classified terminology hierarchy.

Fig. 7. An example ontology.

observation. For this reason, the original top-search algorithm must be adapted to the new situation. Algorithm 6 is the updated version of the top-search procedure.

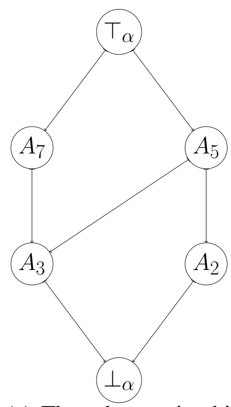
Algorithm 6 not only maintains told subsumption information by the set $green$, but also propagates told non-subsumption information by the set red for further inference.[1] As addressed by Proposition 3, when the position of a merged concept is determined, the subsumption relationships between its successors and the red set are calculated. Furthermore, the subsumption relationship for the concept C and D in Algorithm 6 must be explicitly calculated even when the set $green$ is empty. In the original top-search procedure (Algorithm 3), $C \prec_i D$ is implicitly derived if the set $green$ is empty, which does not hold in the modified top-search procedure (Algorithm 6) since it does not always start from $\top$ anymore when searching for the most specific position of a concept.

3.3 Example

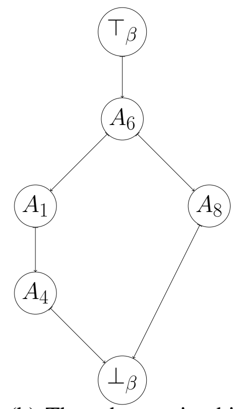
We use a small example TBox to illustrate the algorithm further. Given an ontology with a TBox defined by Figure 7(a), which only contains simple concept subsumption axioms, Figure 7(b) shows the subsumption hierarchy.

Suppose that the ontology is clustered into two groups in the divide phase: $\Delta_\alpha = \{A_2, A_3, A_5, A_7\}$ and $\Delta_\beta = \{A_1, A_4, A_6, A_8\}$. They can be classified independently, and the corresponding subsumption hierarchies are shown in Figure 8.

[1] Our implementation of Algorithm 6 treats subsumptions cycles as *synonyms*. For example, if $rat \sqsubseteq mouse$ and $mouse \sqsubseteq rat$, the two concepts are collapsed into one, $rat/mouse$. For sake of conciseness we do not show these details in Algorithm 6.



(a) The subsumption hierarchy $\leq_\alpha$.



(b) The subsumption hierarchy $\leq_\beta$.

Fig. 8. The subsumption hierarchy over divisions.

In the merge phase, the concepts from $\leq_\beta$ are merged into $\leq_\alpha$. For example, Figure 9 shows a possible computation path where $A_4 \leq A_5$ is being determined.[2] If we assume a subsumption relationship between two concepts is proven when the parent is added to the set box (see Line 15, Algorithm 6), Figure 10 shows the subsumption hierarchy after $A_4 \leq A_5$ has been determined.

4 Termination, Soundness, and Completeness

Lemma 1. *The top-merge algorithm, Algorithm 5, always terminates.*

Proof. During the process of merging two classified terminologies by using $\top\_merge$ from $\top_\alpha$ and $\top_\beta$, either $\top\_merge$ or $\top\_search^*$ is applied to the successors of one of the concerned concepts.

First of all, there can not exist a subsumption cycle between a concerned concept and its successors, because the involved concepts are collapsed and treated as synonyms once such a cycle is detected. Therefore, without an infinite execution on testing a subsumption cycle between a concerned concept and its successors, a limited number of successors are explored, the search continues until $\bot$ is taken into account, and then the algorithm terminates. Consequently, Algorithm $\top\_merge$ always terminates. ■

Similarly, we can establish the following claims:

Lemma 2. *The bottom-merge algorithm always terminates.*

Theorem 1. *Algorithm 1 always terminates.*

With Lemma 1 and 2, it is easy to prove Theorem 1.

[2] This process does not show a full calling order of computing $A_4 \leq A_5$ for sake of brevity. For instance, $\top\_merge(A_7, A_6, \leq_\alpha, \leq_\beta)$ is not shown.

$$
\begin{array}{cl}
\top\_merge(\top_\alpha, \top_\beta, \leq_\alpha, \leq_\beta) & \\
\downarrow & \{\top_\alpha\} \\
\top\_search^*(\top_\beta, \top_\alpha, \leq_\beta, \leq_\alpha) & \\
\downarrow & \leq_\alpha \leftarrow \leq_\alpha \cup \emptyset \\
\top\_merge(A_5, A_6, \leq_\alpha, \leq_\beta) & \\
\downarrow & \emptyset \\
\top\_search^*(A_6, A_5, \leq_\beta, \leq_\alpha) & \\
\downarrow & \leq_\alpha \leftarrow \leq_\alpha \cup \emptyset \\
\top\_merge(A_5, A_1, \leq_\alpha, \leq_\beta) & \\
\downarrow & \emptyset \\
\top\_search^*(A_1, A_5, \leq_\beta, \leq_\alpha) & \\
\downarrow & \leq_\alpha \leftarrow \leq_\alpha \cup \{A_4 \leq A_5\} \\
\top\_merge(A_5, A_4, \leq_\alpha, \leq_\beta) & \\
\downarrow & \{A_5\} \\
\top\_search^*(A_4, A_5, \leq_\beta, \leq_\alpha) & \\
\downarrow & \leq_\alpha \leftarrow \leq_\alpha \cup \{\bot_\beta \leq A_2\} \\
\top\_merge(A_2, \bot_\beta, \leq_\alpha, \leq_\beta) & \\
\downarrow & \{A_2\} \\
\top\_search^*(\bot_\beta, A_2, \leq_\beta, \leq_\alpha) & \\
 & \\
\vdots &
\end{array}
$$

Fig. 9. The computation path of determining $A_4 \leq_i A_5$.

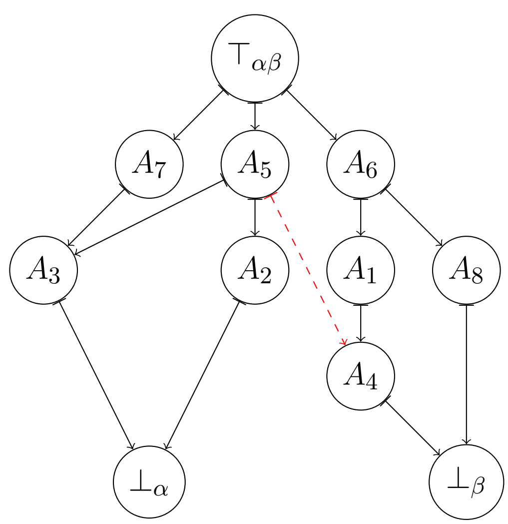


Fig. 10. The subsumption hierarchy after $A_4 \leq A_5$ has been determined.

Definition 1. *Let $S_1 = (x_0, x_1, \ldots, x_m)$ and $S_2 = (y_0, y_1, \ldots, y_n)$ be two paths, and the concatenation of $S_1 \bullet S_2 = (x_0, x_1, \ldots, x_m, y_0, y_1, \ldots, y_n)$. For the empty path λ and a path S, it holds that $S \bullet \lambda = S$, and $\lambda \bullet S = S$.*

Definition 2. *In a classified terminology $\leq$, a concept C's upper inheritance $U(C)$ is a path as follows:*

$$U(C) = \begin{cases} \lambda & C \doteq \top, \\ U(D) \bullet (D) & C \prec D, D \neq \top \end{cases} \tag{1}$$

It is obvious that the following proposition hold:

Proposition 9. *For any concept C in a classified terminology, there must exist at least one upper inheritance $U(C)$.*

Similarly, we get the following symmetric claims:

Definition 3. *In a classified terminology $\leq$, a concept C's lower inheritance $L(C)$ is a path as follows:*

$$L(C) = \begin{cases} \lambda & C \doteq \bot, \\ (D) \bullet L(D) & D \prec C, D \neq \bot \end{cases} \tag{2}$$

Proposition 10. *For any concept C in a classified terminology, there must exist at least one lower inheritance $L(C)$.*

Proposition 11. *The subsumption checking procedure $\leq ?$ is correct, i.e., it holds that $\mathcal{O} \models C \sqsubseteq D \Leftrightarrow \leq ?(C, D) \rightarrow$ true.*

Lemma 3 (Soundness of top_merge). *When merging $\leq_\beta$ into $\leq_\alpha$, for $\forall A : \langle A, \top_\alpha \rangle \in \leq_\alpha$ and $\forall B : \langle B, \top_\beta \rangle \in \leq_\beta$, if the $\top\_merge$ algorithm starting from $\top_\alpha$ and $\top_\beta$ infers that $B \leq A$, then $\mathcal{O} \models B \sqsubseteq A$.*

Proof. This proof is based on Proposition 11. We prove this lemma by contradiction. Let us assume that Algorithm 6 derives $B \leq A$ but $\mathcal{O} \models B \not\sqsubseteq A$.

When the $\top\_merge$ algorithm derives $B \leq A$, there must exist $L(A)$ and $U(B)$ such that, $\exists \underline{A} \in (A) \bullet L(A)$ and $\exists \overline{B} \in U(B) \bullet (B)$, and, as claimed in Propositions 9 and 10, the algorithm determines $\overline{B} \prec \underline{A}$. This means that $\overline{B} \leq \underline{A}$ must be the result of calling $\leq ?(\overline{B}, \underline{A})$ at line 14 of Algorithm 6. This situation is shown as Figure 11.

In the process of determining $\overline{B} \prec \underline{A}$ all children $\underline{A}_i$ of $\underline{A}$ are tested whether they subsume $\overline{B}$ and the calls of $\leq ?(\overline{B}, \underline{A}_i)$ must always have returned $false$, as shown in line 6 of Algorithm 6 and in Figure 12. Therefore, $\overline{B} \prec \underline{A}$ is derived.

We already know $\leq ?(B, \overline{B}) \rightarrow true$ and $\leq ?(\underline{A}, A) \rightarrow true$, $\leq ?(B, A) \rightarrow true$. So, due to the correctness of $\leq ?$ and the transitivity of the subsumption relationship it holds that $\mathcal{O} \models B \sqsubseteq A$, which contradicts our assumption. ■

Similarly, the following corresponding claim can be established.

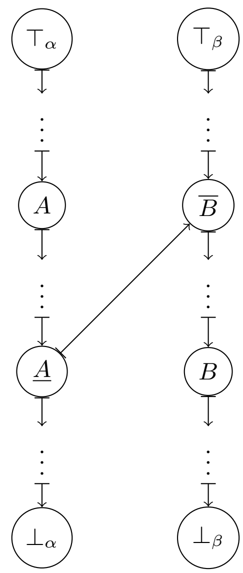


Fig. 11. $B \leq A$.

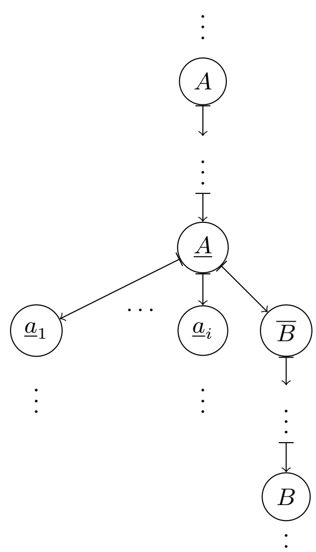


Fig. 12. $B \leq A$ is derived.

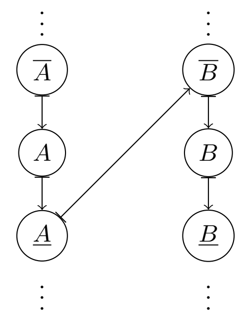


Fig. 13. $\mathcal{O} \models B \sqsubseteq A$.

Lemma 4 (Soundness of bottom_merge). *When merging $\leq_\beta$ into $\leq_\alpha$, for $\forall A : \langle \bot_\alpha, A\rangle \in \leq_\alpha$ and $\forall B : \langle \bot_\beta, B\rangle \in \leq_\beta$, if the $\bot$_merge algorithm starting from $\bot_\alpha$ and $\bot_\beta$ infers that $A \leq B$, then $\mathcal{O} \models A \sqsubseteq B$.*

Following Lemma 3 and 4, as well as Theorem 1, the *soundness* of the merge algorithm is established.

Theorem 2 (Soundness of merge algorithm). *For a merged terminology $\leq$ it holds, if $\langle C, D\rangle \in \leq$, then $\mathcal{O} \models C \sqsubseteq D$.*

Lemma 5 (Completeness of top_merge). *If $\mathcal{O} \models B \sqsubseteq A$, then for $\forall A \subseteq \Delta_\alpha$ and $\forall B \subseteq \Delta_\beta$, the top-merge algorithm infers $B \leq A$, when it merges $\leq_\beta$ into $\leq_\alpha$ starting from $\top_\alpha$ and $\top_\beta$.*

Proof. This proof is based on Proposition 11.

Let $P(A)$ be the set of all paths from $\top$ to $\bot$ that contain A, i.e. $\forall U(A), L(A) : \{U(A) \bullet L(A)\} \subseteq P(A)$. $P(A) \neq \emptyset$ by Propositions 9 and 10. Similarly, $P(B) \neq \emptyset$ is the set of all paths from $\top$ to $\bot$ that contain B. Because $\mathcal{O} \models B \sqsubseteq A$, $P(A) \cap P(B) \neq \emptyset$, i.e. $\exists U(A), L(A), U(B), L(B) : U(A) \bullet L(A) = U(B) \bullet L(B)$. Lemma 5 can be proved by structural induction: If $\mathcal{O} \models B \sqsubseteq A$, then $B \leq A$ can be derived by searching on $U(A) \bullet L(A) = U(B) \bullet L(B)$ with Algorithm 6. The proof for the base cases are trivial, so we just give the induction part.

Let $A \prec \overline{A}$, $\underline{A} \prec A$, $B \prec \overline{B}$, $\underline{B} \prec B$, and $\overline{B} \prec \underline{A}$. That is to say, $\overline{A} \in U(A)$, $\underline{A} \in L(A)$, $\overline{B} \in U(B)$, and $\underline{B} \in L(B)$, as is shown by Figure 13.

Since $\mathcal{O} \models \overline{B} \sqsubseteq \overline{A}$, we have $\leq ?(\overline{B}, \overline{A}) \rightarrow true$: Algorithm 6 puts $\overline{A}$ into $green$ at line 7. And then $\top$_$search^*$ is applied to $\overline{B}$ and every element of $green$, including $\overline{A}$, as is shown by line 21 of Algorithm 6. $\top$_$search^*(\overline{B}, \overline{A}, \leq_\beta, \leq_\alpha)$ tests the subsumption relationships between $\overline{B}$ and every child of $\overline{A}$, including A, at line 6. This process recursively continues to test $\overline{B}$ and $\underline{A}$. At this point, all children of $\underline{A}$ do not subsume $\overline{B}$ and thus are put into red, so $green = \emptyset$ and $box \leftarrow \{\underline{A}\}$, as is shown by line 22 of Algorithm 6 and Figure 14.

Now, Algorithm 6 derives $\leq ?(\overline{B}, \underline{A}) \rightarrow true$, $\leq ?(\underline{A}, A) \rightarrow true$, and $\leq ?(B, \overline{B}) \rightarrow true$, it will be determined $\leq ?(B, A) \rightarrow true$. ■

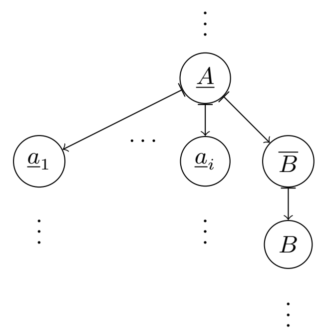


Fig. 14. $\overline{B} \prec \underline{A}$ is derived.

Correspondingly, the completeness of the bottom-merge algorithm is established by Lemma 6.

Lemma 6 (Completeness of bottom_merge). *If $\mathcal{O} \models A \sqsubseteq B$, then for $\forall A \subseteq \Delta_\alpha$ and $\forall B \subseteq \Delta_\beta$, the bottom-merge algorithm infers $A \leq B$, when it merges $\leq_\beta$ into $\leq_\alpha$ starting from $\bot_\alpha$ and $\bot_\beta$.*

From Lemma 5 and 6, we can conclude that the merge algorithm is complete.

Theorem 3 (Completeness of merge algorithm). *If $\mathcal{O} \models C \sqsubseteq D$, the merge algorithm will infer that $C \leq D$.*

5 Partitioning

Partitioning is an important part of this algorithm. It is the main task in the dividing phase. In contrast to simple problem domains such as sorting integers, where the merge phase of a standard merge-sort does not require another sorting, DL ontologies might entail numerous subsumption relationships among concepts. Building a terminology with respect to the entailed subsumption hierarchy is the primary function of DL classification. We therefore assumed that some heuristic partitioning schemes that make use of known subsumption relationships may improve reasoning efficiency by requiring a smaller number of subsumption tests, and this assumption has been proved by our experiments, which are described in Section 6.

So far, we have presented an ontology partitioning algorithm by using only told subsumption relationships that are directly derived from concept definitions and axiom declarations. Any concept that has at least one told super- and one sub-concept, can be used to construct a told subsumption hierarchy. Although such a hierarchy is usually incomplete and many entailed subsumptions are missing, it contains already known subsumptions indicating the closeness between concepts w.r.t. subsumption. Such a raw subsumption hierarchy can be represented as a directed graph with only one root, the $\top$ concept. A heuristic partitioning method can be defined by traversing the graph in a breadth-first way, starting from $\top$, and collecting traversed concepts into partitions. Algorithm 7 and 8 address this procedure.

Algorithm 7: $cluster(G)$

```
input  : G: the told subsumption graph
output : R: the concept names partitions
begin
    R ← ∅;
    visited ← ∅;
    N ← get_top_children(⊤, G);
    foreach n ∈ N do
        P ← {n};
        visited ← visited ∪ {n};
        R ← R ∪ {build_partition(n, visited, G, P)};
    end foreach
    return R;
end
```

Algorithm 8: $build\_partition(n, visited, G, P)$

```
input  : n: an concept name
         visited: a list recording visited concept names
         G: the told subsumption graph
         P: a concept names partition
output : R: a concept names partition
begin
    R ← ∅;
    N ← get_children(n, visited, G, P);
    foreach n' ∈ N do
        if n' ∉ visited then
            P ← P ∪ {n'};
            visited ← visited ∪ {n'};
            build_partition(n', visited, G, P);
        end if
    end foreach
    R ← P;
    return R;
end
```

Algorithm 9: $schedule\_merging(q)$

```
input  : q: the job queue
output : r: the updated job queue
begin
    got ← false;
    while ¬got ∧ size(q) > 0 do
        bolt ← dequeue(q);
        nut ← dequeue(q);
        if ¬null?(bolt) ∧ ¬null?(nut) then
            got ← true;
            enqueue(q, merge(bolt, nut));
        else if ¬null?(bolt) then
            enqueue(q, bolt);
            bolt ← null;
        else if ¬null?(nut) then
            enqueue(q, nut);
            nut ← null;
        end if
    end while
    r ← q;
    return r;
end
```

6 Evaluation

Our experimental results clearly show the potential of merge-classification. We could achieve speedups up to a factor of 4 by using a maximum of 8 parallel workers, depending on the particular benchmark ontology. This speedup is in the range of what we expected and comparable to other reported approaches, e.g., the experiments reported for the ELK reasoner [16, 17] also show speedups of up to a factor of 4 when using 8 workers, although a specialized polynomial procedure is used for $\mathcal{EL}$+ reasoning that seems to be more amenable to concurrent processing than standard tableau methods.

We have designed and implemented a concurrent version of the algorithm so far. Our program[3] is implemented on the basis of the well-known reasoner JFact,[4] which is open-source and implemented in Java. We modified JFact such that we can execute a set of JFact reasoning kernels in parallel in order to perform the merge-classification computation. We try to examine the effectiveness of the merge-classification algorithm by adapting such a mature DL reasoner.

6.1 Experiment

A multi-processor computer, which has 4 octa-core processors and 128G memory installed, was employed to test the program. The Linux OS and 64-bit OpenJDK 6 were

[3] http://github.com/kejia/mc
[4] http://jfact.sourceforge.net

ontology	expressivity	concept count	axiom count
adult_mouse_anatomy	$\mathcal{ALE}+$	2753	9372
amphibian_gross_anatomy	$\mathcal{ALE}+$	701	2626
c_elegans_phenotype	$\mathcal{ALEH}+$	1935	6170
cereal_plant_trait	$\mathcal{ALEH}$	1051	3349
emap	$\mathcal{ALE}$	13731	27462
environmental_entity_logical_definitions	$\mathcal{SH}$	1779	5803
envo	$\mathcal{ALEH}+$	1231	2660
fly_anatomy	$\mathcal{ALEI}+$	6222	33162
human_developmental_anatomy	$\mathcal{ALEH}$	8341	33345
medaka_anatomy_development	$\mathcal{ALE}$	4361	9081
mpath	$\mathcal{ALEH}+$	718	4315
nif-cell	$\mathcal{S}$	376	3492
sequence_types_and_features	$\mathcal{SH}$	1952	6620
teleost_anatomy	$\mathcal{ALER}+$	3036	11827
zfa	$\mathcal{ALEH}+$	2755	33024

Table 1. Metrics of the test cases.

employed in the tests. The JVM was allocated at least 16G memory initially, given that at most 64G physical memory was accessible. Most of the test cases were chosen from *OWL Reasoner Evaluation Workshop 2012 (ORE 2012)* data sets. Table 1 shows the test cases' metrics.

Each test case ontology was classified with the same setting except for an increased number of workers. Each worker is mapped to an OS thread, as indicated by the Java specification. Figures 15 and 16 show the test results.

In our initial implementation, we used an *even-partitioning* scheme. That is to say concept names are randomly assigned to a set of partitions. For the majority of the above-mentioned test cases we observed a small performance improvement below a speedup factor of 1.4, for a few an improvement of up to 4, and for others only a decrease in performance. Much overhead was shown in these test cases.

As mentioned in Section 5, we assumed that a heuristic partitioning might promote a better reasoning performance, e.g., a partitioning scheme considering subsumption axioms. This idea is addressed by Algorithm 7 and 8.

Another issue that happens when partitions are merged in a shared-memory parallel environment is *racing*. In the merge-classification case, each worker puts the classified partition to a shared queue, and then picks two out of it to merge them. Workers race with each other to get merging pairs. That is to say which and how many partitions some worker gets is indeterminate. This may become the source of deadlocks or other concurrency issues. We designed a schedule algorithm to constrain the race from such concurrency issues. Algorithm 9 ensures that each worker starts merging if and only if the worker has obtained two partitions.

We implemented Algorithms 7, 8, and 9, and tested our program. Our assumption has been proved by the test: Heuristic partitioning may improve reasoning performance where blind partitioning can not.

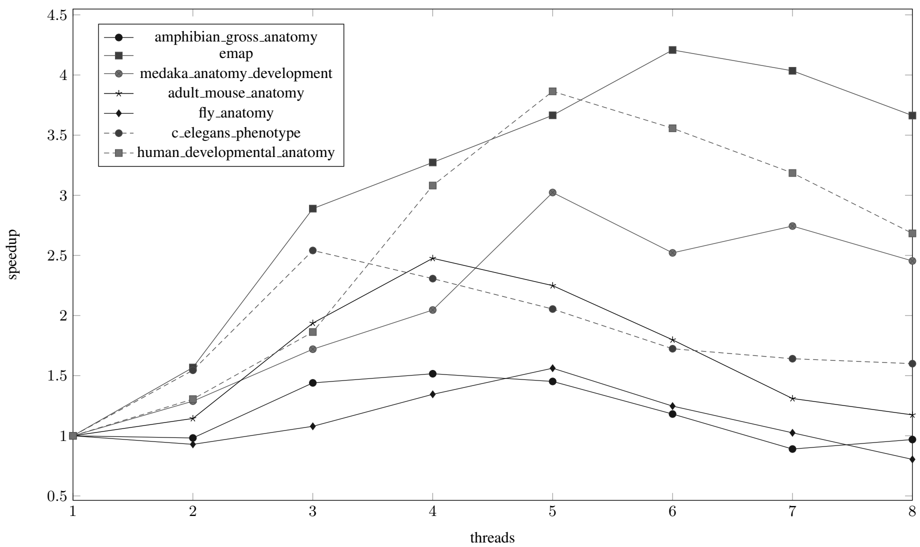


Fig. 15. The performance of parallelized merge-classification—I.

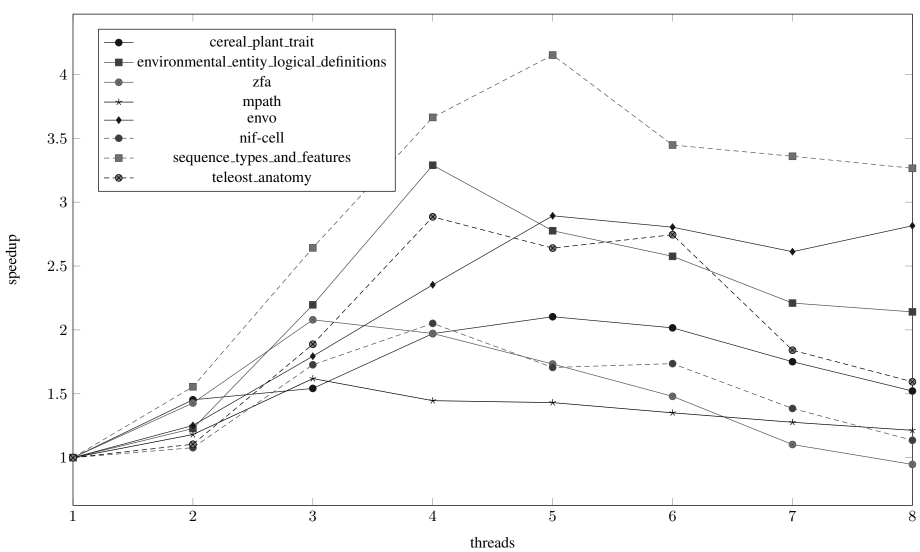


Fig. 16. The performance of parallelized merge-classification—II.

6.2 Discussion

Our experiment shows that with a heuristic divide scheme the merge-classification algorithm can increase reasoning performance. However, such performance promotion is not always tangible. In a few cases, the parallelized merge-classification merely degrades reasoning performance. The actual divide phase of our algorithm can influence the performance by creating better or worse partitions.

A heuristic divide scheme may result in a better performance than a blind one. According to our experience, when the division of the concepts from the domain is basically random, sometimes divisions contribute to promoting reasoning performance, while sometimes they do not. A promising heuristic divide scheme seems to be in grouping a family of concepts, which have potential subsumption relationships, into the same partition. Evidently, due to the presence of non-obvious subsumptions, it is hard to guess how to achieve such a good partitioning. We tried to make use of obvious subsumptions in axioms to partition closely related concepts into the same group. The tests demonstrate a clear performance improvement in a number of cases.

While in many cases merge-classification can improve reasoning performance, for some test cases its practical effectiveness is not yet convincing. We are still investigating the factors that influence the reasoning performance for these cases but cannot give a clear answer yet. The cause may be the large number of *general concept inclusion (GCI)* axioms found in some ontologies. Even with a more refined divide scheme, those GCI axioms can cause inter-dependencies between partitions, and may cause in the merge phase an increased number of subsumption tests. Also, the indeterminism of the merging schedule, i.e., the unpredictable order of merging divides, needs to be effectively solved in the implementation, and racing conditions between merging workers as well as the introduced overhead may decrease the performance. In addition, the limited performance is caused by the experimental environment: Compared with a single chip architecture, the 4-chip-distribution of the 32 processors requires extra computational overhead, and the memory and thread management of JVM may decrease the performance of our program.

7 Related Work

A key functionality of a DL reasoning system is TBox *classification*, computing all entailed subsumption relationships among named concepts. The generic top-search & bottom-search algorithm was introduced by [19] and extended by [2]. The algorithm is used as the standard technique for incrementally creating subsumption hierarchies of DL ontologies. [2] also presented some basic traversal optimizations. After that, a number of optimization techniques have been explored [26, 8, 9]. Most of the optimizations are based on making use of the partial transitivity information in searching. However, research on how to use concurrent computing for optimizing DL reasoning has started only recently.

7.1 Brute-force Parallelized Classification Scheme

TBox classification calculates subsumptions relationships between concepts. That executes subsumption tests one by one. Those subsumption tests can be processed independently. One of our previous research approaches parallelizes subsumption tests during classification, by which scalability can be gained [28].

The reasoning prototype implements a parallelized $\mathcal{ALC}$ TBox classifier. It can concurrently classify an $\mathcal{ALC}$ terminology. Its parallelized classification service computes subsumptions in a brutal way [2]. It is obvious that the algorithm is sound and complete and in a sequential context has a quadratic time complexity for subsumption computation. In order to figure out a terminology hierarchy, the algorithm calculates the subsumptions of all atomic concepts pairs. A subsumption relationship only depends on the involved concepts pair, and does not have any connections with the computation order. Therefore, the subsumptions can be computed in parallel, and the soundness and completeness are retained in a concurrent context. This naive scheme could achieve an impressive speedup factor of up to 18 when running up to 36 threads on a 16-core server [28]. This speedup is mostly due to minimal interactions and data sharing between threads independently performing subsumption tests and causing almost no overhead.

7.2 Other Research

The merge-classification algorithm is suitable for concurrent computation implementation, including both shared-memory parallelization and distributed systems. Several concurrency-oriented DL reasoning schemes have been presented recently. [18] reported on experiments with a parallel $\mathcal{SHN}$ reasoner. This reasoner could process *disjunction* and *at-most cardinality restriction* rules in parallel, as well as some primary DL tableau optimization techniques. [1] presented the first algorithms on parallelizing TBox classification using a shared global subsumption hierarchy, and the experimental results promise the feasibility of parallelized DL reasoning. [16, 17] reported on the ELK reasoner, which can classify $\mathcal{EL}$ ontologies concurrently, and its speed in reasoning about $\mathcal{EL}+$ ontologies is impressive. In [29] we explored parallelization of conjunctive branches in tableau-based DL reasoning and achieved a moderate speedup due to high overhead. The Konclude system[5] can take advantage of multiple cores within a shared memory environment and implements reasoning for $\mathcal{SROIQV}$ but it has not yet been reported what inference services have been parallelized. In [21, 20] the idea of applying a constraint programming solver has been proposed. Besides the shared-memory concurrent reasoning research mentioned above, non-shared-memory distributed concurrent reasoning has been investigated recently by [25, 22].

Merge-classification needs to divide ontologies. Ontology partitioning can be considered as a sort of *clustering* problem. These problems have been extensively investigated in networks research, such as [6, 7, 31]. Algorithms adopting more complicated heuristics in the area of ontology partitioning, have been presented in [5, 12, 10, 11, 14].

Our merge-classification approach employs the well-known *divide and conquer* strategy. There is sufficient evidence showing that this type of algorithms is well suited to be processed in parallel [27, 4, 15]. Some experimental work on parallelized merge sort are reported in [24, 23].

[5] http://www.konclude.com

8 Conclusion

The approach presented in this paper has been motivated by the observation that (i) multi-processor/core hardware is becoming ubiquitously available but standard OWL reasoners do not yet make use of these available resources; (ii) although most OWL reasoners have been highly optimized and impressive speed improvements have been reported for reasoning in the three tractable OWL profiles, there exist a multitude of OWL ontologies that are outside of the three tractable profiles and require long processing times even for highly optimized OWL reasoners. Recently, concurrent computing has emerged as a possible solution for achieving a better scalability in general and especially for such difficult ontologies, and we consider the research presented in this paper as an important step in designing adequate OWL reasoning architectures that are based on concurrent computing.

One of the most important obstacles in successfully applying concurrent computing is the management of overhead caused by concurrency. An important factor is that the load introduced by using concurrent computing in DL reasoning is usually remarkable. Concurrent algorithms that cause only a small overhead seem to be the key to successfully apply concurrent computing to DL reasoning.

Our merge-classification algorithm uses a *divide and conquer* scheme, which is potentially suitable for low overhead concurrent computing since it rarely requires communication among divisions. Although the empirical tests show that the merge-classification algorithm does not always improve reasoning performance to a great extent, they let us be confident that further research is promising. For example, investigating what factors impact the effectiveness and efficiency of the merge-classification may help us improve the performance of the algorithm further.

At present our work adopts a heuristic *partitioning* scheme at the divide phase. Different divide schemes may produce different reasoning performances. We are planning to investigate better divide methods. Furthermore, our work has only researched the performance of the *concurrent* merge-classification so far. How the number of division impacts the reasoning performance in a single thread and a multiple threads setting needs be investigated in more detail.

Acknowledgements

We are grateful to Ralf Möller at Hamburg University of Technology in giving us access to their equipment that was used to conduct the presented evaluation.

References

1. Aslani, M., Haarslev, V.: Parallel TBox classification in description logics—first experimental results. In: Proceedings of the 2010 conference on ECAI 2010: 19th European Conference on Artificial Intelligence. pp. 485–490 (2010)
2. Baader, F., Hollunder, B., Nebel, B., Profitlich, H.J., Franconi, E.: An empirical analysis of optimization techniques for terminological representation systems. Applied Intelligence 4(2), 109–132 (1994)

3. Baader, F., et al.: The description logic handbook: theory, implementation, and applications. Cambridge University Press (2003)
4. Cole, R.: Parallel merge sort. SIAM Journal on Computing 17(4), 770–785 (1988)
5. Doran, P., Tamma, V., Iannone, L.: Ontology module extraction for ontology reuse: an ontology engineering perspective. In: Proceedings of the sixteenth ACM conference on Conference on information and knowledge management. pp. 61–70 (2007)
6. Ester, M., Kriegel, H.P., Sander, J., Xu, X.: A density-based algorithm for discovering clusters in large spatial databases with noise. In: Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (1996)
7. Girvan, M., Newman, M.E.J.: Community structure in social and biological networks. Proceedings of the National Academy of Sciences of the United States of America 99(12), 7821–7826 (2002)
8. Glimm, B., Horrocks, I., Motik, B., Stoilos, G.: Optimising ontology classification. In: Proceedings of the 9th International Semantic Web Conference (ISWC 2010). LNCS, vol. 6496, pp. 225–240. Springer Verlag (2010)
9. Glimm, B., Horrocks, I., Motik, B., Shearer, R., Stoilos, G.: A novel approach to ontology classification. Web Semantics: Science, Services and Agents on the World Wide Web 14, 84–101 (2012)
10. Grau, B.C., Horrocks, I., Kazakov, Y., Sattler, U.: A logical framework for modularity of ontologies. In: Proceedings International Joint Conference on Artificial Intelligence. pp. 298–304 (2007)
11. Grau, B.C., Horrocks, I., Kazakov, Y., Sattler, U.: Modular reuse of ontologies: Theory and practice. Journal of Artificial Intelligence Research 31(1), 273–318 (2008)
12. Grau, B.C., Parsia, B., Sirin, E., Kalyanpur, A.: Modularizing OWL ontologies. In: K-CAP 2005 Workshop on Ontology Management (2005)
13. Hitzler, P., Krötzsch, M., Rudolph, S.: Foundations of semantic web technologies. Chapman & Hall/CRC (2009)
14. Hu, W., Qu, Y., Cheng, G.: Matching large ontologies: A divide-and-conquer approach. Data & Knowledge Engineering 67(1), 140–160 (2008)
15. Jeon, M., Kim, D.: Parallel merge sort with load balancing. International Journal of Parallel Programming 31(1), 21–33 (2003)
16. Kazakov, Y., Krötzsch, M., Simančík, F.: Concurrent classification of EL ontologies. In: Proceedings of the 10th International Semantic Web Conference (2011)
17. Kazakov, Y., Krötzsch, M., Simančík, F.: The incredible ELK: From polynomial procedures to efficient reasoning with EL ontologies (2013), submitted to a journal
18. Liebig, T., Müller, F.: Parallelizing tableaux-based description logic reasoning. In: Proceedings of the 2007 OTM Confederated International Conference on the Move to Meaningful Internet Systems-Volume Part II. pp. 1135–1144 (2007)
19. Lipkis, T.: A KL-ONE classifier. In: Proceedings of the 1981 KL-ONE Workshop. pp. 128–145 (1982)
20. Meissner, A.: A simple parallel reasoning system for the ALC description logic. In: Computational Collective Intelligence: Semantic Web, Social Networks and Multiagent Systems (First International Conference, ICCCI 2009, Wroclaw, Poland, October 2009). pp. 413–424 (2009)
21. Meissner, A., Brzykcy, G.: A parallel deduction for description logics with ALC language. Knowledge-Driven Computing 102, 149–164 (2008)
22. Mutharaju, R., Hitzler, P., Mateti, P.: DistEL: A distributed EL+ ontology classifier. In: Proceedings of The 9th International Workshop on Scalable Semantic Web Knowledge Base Systems (2013)

23. Radenski, A.: Shared memory, message passing, and hybrid merge sorts for standalone and clustered SMPs. In: The 2011 International Conference on Parallel and Distributed Processing Techniques and Applications. vol. 11, pp. 367–373 (2011)
24. Rolfe, T.J.: A specimen of parallel programming: parallel merge sort implementation. ACM Inroads 1(4), 72–79 (2010)
25. Schlicht, A., Stuckenschmidt, H.: Distributed resolution for ALC - first results. In: Proceedings of the Workshop on Advancing Reasoning on the Web: Scalability and Commonsense (2008)
26. Shearer, R., Horrocks, I., Motik, B.: Exploiting partial information in taxonomy construction. In: Grau, B.G., Horrocks, I., Motik, B., Sattler, U. (eds.) Proceedings of the 2009 International Workshop on Description Logics. CEUR Workshop Proceedings, vol. 477. Oxford, UK (July 27–30 2009)
27. Todd, S.: Algorithm and hardware for a merge sort using multiple processors. IBM Journal of Research and Development 22(5), 509–517 (1978)
28. Wu, K., Haarslev, V.: A parallel reasoner for the description logic ALC. In: Proceedings of the 2012 International Workshop on Description Logics (2012)
29. Wu, K., Haarslev, V.: Exploring parallelization of conjunctive branches in tableau-based description logic reasoning. In: Proceedings of the 2013 International Workshop on Description Logics (2013)
30. Wu, K., Haarslev, V.: Parallel OWL reasoning: Merge classification. In: The Third Joint International Semantic Technology Conference (2013)
31. Xu, X., Yuruk, N., Feng, Z., Schweiger, T.A.J.: SCAN: a structural clustering algorithm for networks. In: Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining. pp. 824–833 (2007)