Optimised Ontology Classification

Birte Glimm, Ian Horrocks, Boris Motik, and Giorgos Stoilos

Oxford University Computing Laboratory, UK

Abstract. The problem of computing all subclass relationships between classes and properties is one of the most important reasoning tasks for OWL ontologies. In this paper, we propose a new class classification algorithm mainly motivated by the previous work in [1]. We close several gaps that were left open in the original paper and provide average optimal strategies. We then turn our attention to (object and data) property classification and show that the commonly used algorithms for property classification are incomplete for OWL 2 ontologies. We propose an encoding of the property classification problems into class classification problems, which allows us to take advantage of all the optimisations developed for class classification. We implemented all of the proposed techniques in the OWL reasoner 1.2.4 and compare this version to Hermit 1.2.2 to evaluate the performance improvements.

1 Introduction

One of the most important reasoning tasks for OWL ontologies is computing all implied class and property subsumptions, generating a so-called “hierarchy” ($H$). Although modern OWL reasoners are highly optimised and many ontologies are quite trivial for them, the problem of efficiently computing $H$ given any ontology of arbitrary size and structure is still an open issue.

Most systems are based on the enhanced traversal (ET) algorithm [2], where an unclassified element $C$ is added in the partially constructed hierarchy by a two-phase procedure. In the first phase, the most specific superclasses are determined using a top-down breadth-first procedure, while in a second phase the most general subclasses of $C$ are determined by a bottom-up procedure. Additionally to the ET algorithm most system also include several optimisation techniques to further reduce the number of tests required to fully classify an ontology [3, 4].

Recently, an alternative strategy to the ET algorithm was proposed [1], which is likely to perform better on large ontologies where traversing $H$ with ET can be quite inefficient. In order to insert a class $C$ into the hierarchy, the new strategy traverses only the part of the hierarchy that contains its unknown subsumers and subsumees, which is expected to be quite small in practice. The algorithm does not directly work on $H$, but keeps a set of known ($K$) and possible ($P$) subsumptions for each class, which are updated as classification progresses. We call this algorithm the KP algorithm in the remainder. An interesting observation is that the transitivity of the subClassOf relation can be used to propagate (non)subsumptions over these sets, e.g., by pruning relations from $P$. 
Unfortunately, several issues have been left open for the KP algorithm. Firstly, it is not clear what the best way to initialise the relations $K$ and $P$ is, as setting $K := \emptyset$ and $P := C_O \times C_O$, for $C_O$ the set of class names occurring in the ontology, is clearly not realistic. Furthermore, only a naive pruning algorithm for $P$ was presented, which consists of several iterations over the sets $K$ and $P$, making it impractical for large ontologies. Our first goal in this paper is to close these issues and present an efficient and practical algorithm for class classification. Our contributions towards this goal can be summarised as follows:

1. We present a novel strategy for instantiating the relations $K$ and $P$. This is based on well-known graph-theoretic operations and can be implemented very efficiently. Our experience has shown that it is also optimal on average, i.e., it neither leads to redundancies nor underestimates $K$ and $P$.
2. Based on the previous strategy we present a new heuristic optimisation which is very effective on ontologies that contain many unsatisfiable classes.
3. We present an efficient pruning algorithm that is used to remove relations from $P$. We additionally show how this algorithm implements the most important parts of the pruning algorithm given in [1].
4. We further illustrate technical differences between the two algorithms that are important when classifying large ontologies.

Finally, we turn our attention to property classification. To the best of our knowledge this is the first attempt to address the issue of efficiently classifying object and data properties of an OWL 2 ontology. Traditionally, most systems implement cheap algorithms that compute the transitive reflexive closure of the asserted property hierarchy. Unfortunately, these algorithms are sufficient only for simple OWL fragments while they are incorrect already for OWL DL. Our contributions on property classification can be summarised as follows:

1. We give illustrative examples that show that in expressive formalisms such as OWL DL, a transitive reflexive closure is not enough and one has to perform actual subsumption tests in order to compute the object and the data property hierarchy.
2. We highlight the problems that one faces in computing the object and data property hierarchy and show why the standard ET and the KP algorithms do not perform well in the case of property classification.
3. We present our optimisations by proposing a way of encoding the property classification problem into a class classification problem. By doing this, our previously introduced class classification algorithm can be reused in much the same way for properties as well.

The novel class and property classification algorithms are implemented in the HermiT reasoner. In order to analyse how much improvement the novel methods achieve, we compare HermiT 1.2.2, which implements the ET classification algorithm for classes and properties, with HermiT 1.2.4, which implements the new classification methods for class, object, and data property classification.

The paper is organised as follows: in Section 2, we introduce some preliminaries including the main ideas of the classification algorithm on which we based our
new strategy [1]. In Section 3, we describe the new classification algorithm and describe several differences compared to KP. In Section 4, we highlight the problems with property classification and show how we reduce this problem to class classification. Finally, we evaluate the new classification methods in Section 5 and conclude.

2 Preliminaries

An OWL 2 DL ontology consists of a set of axioms, which describe general knowledge about a domain and facts that are generally considered to be true in that domain. The meaning of axioms in an OWL ontology is precisely defined in terms of model theory [5].

For a full set of OWL 2 modelling construct, we refer to the OWL 2 Structural Specification [6]. Examples of modelling constructs are

\[
\begin{align*}
\text{SubClassOf}(C & D) \quad (1) \\
\text{DataPropertyAssertion}(dp & a \text{“5”} \text{xsd:integer}) \quad (2) \\
\text{ObjectPropertyDomain}(op & \text{ObjectOneOf}(a \ b)) \quad (3)
\end{align*}
\]

Axiom (1) states that the class \(C\) is a subclass of the class \(D\); axiom (2) states that the individual \(a\) is related to the integer 5 with the data property \(dp\); finally, axiom (3) states that the domain of the object property \(op\) consists of the two explicitly given values \(a\) and \(b\).

Concrete values, e.g., the literal \(\text{“5”} \text{xsd:integer}\) in the above example, can come from the OWL 2 datatype map, which contains most of the XML Schema datatypes and some additional OWL specific datatypes.

The interpretation of axioms in an OWL ontology \(O\) is given by means of two-sorted interpretations over the object domain and the data domain (containing concrete values such as integers, strings, etc.). An interpretation maps classes to subsets of the object domain, object properties to pairs of elements from the object domain, data properties to a pair of an element from the object domain and an element from the data domain, individuals to elements in the object domain, a datatype to a subset of the data domain, and a literal to an element in the data domain. For an interpretation to be a model of the ontology, several conditions have to be satisfied [5]. E.g., if \(O\) contains \(\text{SubClassOf}(C \ D)\), then the interpretation of \(C\) must be a subset of the interpretation of \(D\). If the axioms of \(O\) cannot be satisfied in any interpretation, i.e., the ontology has no model, it is called inconsistent and it is called consistent otherwise. If the interpretation of a class \(C\) is necessarily a subset of the interpretation of a class \(D\) in any model of \(O\), then we say \(O\) entails \(C \subseteq D\) and we write \(O \models C \subseteq D\). If the interpretations of \(C\) and \(D\) necessarily coincide, we write \(O \models C \equiv D\). We call a class satisfiable, if there is a model of \(O\) in which the interpretation of \(C\) is non-empty and it is unsatisfiable otherwise. We use analogous notations for object and data properties. For full details of the OWL 2 Direct Semantics we refer to the OWL 2 Direct Semantics specification [5]. We use \(C_O\) to
denote the class names that occur in $O$, containing owl:Thing and owl:Nothing: $OP_O$ ($DP_O$) to denote the object (data) properties occurring in $O$, containing owl:TopObjectProperty and owl:BottomObjectProperty (owl:TopDataProperty and owl:BottomDataProperty).

The following ontology $O$ entails $C \sqsubseteq E$, although this is not directly stated:

\[
\text{SubClassOf}(C \text{ ObjectSomeValuesFrom}(opD)) \quad (4)
\]

\[
\text{ObjectPropertyDomain}(opE) \quad (5)
\]

In every model of $O$, an instance $i$ of $C$ must be related to an instance of the class $D$ with the property $op$ by axiom (4). Since $i$ has an $op$-successor, the property domain axiom (5) requires that $i$ is also an instance of the class $E$ and we have that $C \sqsubseteq E$ holds.

2.1 KP Classification Algorithm

Class classification is a prominent reasoning task, which amounts to computing all pairs of classes $\langle C, D \rangle$ such that $O | = C \sqsubseteq D$. For example, from the ontology with axioms (4) and (5), a classification algorithm should compute the hierarchy:

\[
\{ \langle \text{owl:Nothing}, C \rangle, \langle \text{owl:Nothing}, D \rangle, \langle C, E \rangle, \langle E, \text{owl:Thing} \rangle, \langle D, \text{owl:Thing} \rangle \}.
\]

Recently, a new classification algorithm [1] was proposed that revises the standard ET strategy [2]. The algorithm operates over two binary relations $K, P \subseteq C_O \times C_O$ such that if $\langle C, D \rangle \in K$ then $C \sqsubseteq D$ is known, while if $\langle C, D \rangle \in P$ then $C \sqsubseteq D$ is possible (either known or unknown). Any relation not in $P$ is interpreted as a known non-subsumption, while $P \setminus K$ contains only the unknown possible ones. The algorithm then expands $K$ and reduces $P$ until $K = P$, in which case all subsumption relations are known. Roughly speaking the algorithm works as follows: Firstly, $K$ is transitively closed to materialise all subsumptions inferred from transitivity of $\sqsubseteq$. Then, it picks an unclassified class $C$, i.e., one such that there exists $D$ with $\langle C, D \rangle \in P \setminus K$, it generates a partial hierarchy $H_C$ of all unknown possible subsumers of $C$ and applies the standard ET procedure for $C$ on $H_C$. New subsumptions are added to $K$ updating also its transitivity, while non-subsumptions are removed from $P$.

For the method to converge faster, a pruning algorithm was proposed that exploits newly discovered (non)subsumptions to further reduce $P$ without performing actual subsumption tests. For example, let $\langle C, D \rangle \in K$, $\langle D, E \rangle \in P$, $\langle E, F \rangle \in K$ and $\langle C, F \rangle \in P$. If $O \not |= C \sqsubseteq F$, then it is necessary that $O \not |= D \sqsubseteq E$, and thus $\langle D, E \rangle$ an also be removed from $P$. Similarly, for new positive subsumptions: if $\langle C, D \rangle \in P$, $\langle D, E \rangle \in P$, $\langle E, F \rangle \in K$, $\langle C, F \rangle \not \in P$ and $O |= C \sqsubseteq D$, then $\langle D, E \rangle$ can be pruned. For $P$ and $K$ as before, $V$ a set of newly discovered subsumptions and $N$ a set of newly discovered non-subsumptions the authors provide a naive pruning algorithm detailed in Algorithm 1.

A major issue of the algorithm is how one should initialise $K$ and $P$. The authors proposed to exploit the information that is generated by tableau reasoners
Algorithm 1 Prune Additional Possible Subsumptions

Algorithm: \texttt{pruneNonPossible}(P, K, V, N)

\textbf{Input:} \(P\): set of possible subsumptions to be pruned, \(K\): set of known subsumptions, \(V\): set of new positive subsumptions, \(N\): set of new non-subsumptions

\begin{algorithmic}
\begin{algorithmic}
\State \textbf{for each} \((C, D) \in N\) \textbf{do}
\State \textbf{for each} \((E, F)\) such that \((C, E) \in K\) and \((F, D) \in K\) remove \((E, F)\) from \(P\)
\State \textbf{for each} \((C, D) \in V\) \textbf{do}
\State \textbf{for each} \((D, E) \in P\) \textbf{do}
\State \hspace{1em} \textbf{if} \((E, F) \in K\) and \((C, F) \notin P\) \textbf{then} remove \((D, E)\) from \(P\)
\State \textbf{for each} \((E, C) \in P\) \textbf{do}
\State \hspace{1em} \textbf{if} \((F, E) \in K\) and \((F, D) \notin P\) \textbf{then} remove \((E, C)\) from \(P\)
\end{algorithmic}
\end{algorithmic}

During reasoning. More precisely, when testing satisfiability of a class \(A\) these algorithms usually initialise a node \(s_0\) with label \(L(s_0) = \{A\}\) and then apply expansion rules, constructing an abstraction of a model for the satisfiability of \(A\), called \textit{pre-model}. At the end of the procedure \(L(s_0)\) can be used to extract information about subsumers and non-subsumers of \(A\). More precisely, in many cases if \(L(s_0)\) does not contain a class \(B\) then the pre-model can be unravelled into a witness model for the non-subsumption \(A \not\sqsubseteq B\). Similarly, if \(B\) was deterministically added to \(L(s_0)\), i.e. no non-deterministic expansion was involved, it is usually the case that \(O \models A \sqsubseteq B\). Actually, in HermiT’s hypertableau setting these two observations always hold [7]. Consequently, one can perform satisfiability tests for atomic classes in \(O\) and use the pre-models created by HermiT to initialise \(K\) and \(P\). It is not clear however if all classes should be tested or whether there is a more optimal strategy. Another open issue is a practical and efficient implementation of the pruning algorithm, which as given consists of several loops over two potentially large relations.

3 Optimised Class Classification

Based on the KP algorithm [1] we propose a new classification algorithm that has been implemented in the HermiT (hyper)tableau reasoner. The major goal of this algorithm is to provide solutions to open issues of the KP algorithm, like an optimal strategy for initialising \(K\) and \(P\) and a practical approach to pruning non-subsumption, as well as extend it with new heuristic optimisations. In the following, we first describe the algorithm and then we also contrast it with important parts of KP.

The new algorithm is detailed in Algorithm 2. Like KP it operates over a set of known \((K)\) and possible \((P)\) subsumptions to compute a classified hierarchy. The algorithm uses an OWL reasoner to check for the satisfiability of a class (line 6) or subsumption between classes (line 25) by using the well-known reduction of class inclusion to class satisfiability. In lines 2, 16, 20, 31 and 32, it uses well-known operations over binary relations to manipulate \(K\) and \(P\). These are described next.
Algorithm 2 New Classification Algorithm

**Algorithm:** Classify($O$)

**Input:** $O$ : ontology to be classified

1. $K :=$ performStructuralSubsumption($O$)
2. $(\mathcal{H}, \rho) :=$ extractHierarchy($K$)
3. Initialise a list $ToTest := \{ C \mid \langle$owl:Nothing, $C \rangle \in \mathcal{H} \}$ and $Unsat := \emptyset$
4. **While** $ToTest \neq \emptyset$
   5. Starting removing the head $C$ from $ToTest$ until $P[C] = \emptyset$
   6. $A :=$ buildModelFor($C(s_0)$)
   7. **If** $A = \emptyset$ **then** //class was unsatisfiable
      8. **For each** $\langle C, D \rangle \in \mathcal{H}$ add $D$ to the front of $ToTest$
      9. **For each** descendant $E$ of $C$ in $\mathcal{H}$ that is not already in $Unsat$ do
         Add $\langle E, \text{owl:Nothing} \rangle$ to $K$, add $E$ to $Unsat$ and remove $E$ from $ToTest$
   10. **Else**
       11. **For each** $D \in \mathcal{L}(s_0)$ with depSet($D, A) = \emptyset$, add $\langle C, D \rangle$ to $K$
       12. **For each** $s$ in $A$ and for each $D \in \mathcal{L}(s)$ do
           13. **If** $P[D] = \emptyset$ **then** $P[D] := \mathcal{L}(s) \cap C_\emptyset$
           14. **Else** $P[D] := P[D] \cap \mathcal{L}(s)$
       15. **For each** $C$ and for each $D \in \text{reachable}(C, K)$, set $P[C] := P[C] \setminus \{D\}$
       16. $\text{UnClass} := \{ C \mid P[C] \neq \emptyset \}$
       17. **While** $\text{UnClass} \neq \emptyset$
          18. Let $B := P[C] \cup \{ \text{owl:Nothing, owl:Thing} \}$ for some $C \in \text{UnClass}$
          19. $(\mathcal{H}, \rho) :=$ extractHierarchy($\text{project}(K, B))$
          20. Initialise a queue $Q$ with $Q := \{ \text{owl:Thing} \}$
          21. **While** $Q \neq \emptyset$
             22. Remove the head $H$ from $Q$
             23. **For each** $D$ s.t. $\langle D, H \rangle \in \mathcal{H}$ and $D \in P[C]$ do
                24. $A := $ buildModelFor($\langle C \setminus \neg D \rangle(s_0)$)
                25. **If** $A \neq \emptyset$ **then** // $C \setminus \neg D$ was satisfiable, i. e., $C \not\subseteq D$
                   **For each** $s$ in $A$ and each $D \in \mathcal{L}(s)$ set $P[D] := P[D] \cap \mathcal{L}(s)$
                   26. **Else**
                      27. Add $\langle C, D \rangle$ to $K$ and $D$ to $Q$
                      28. $P[C] := \emptyset$
             29. **For each** $D \in \text{UnClass}$ and $E \in \text{reachable}(D, K)$, set $P[D] := P[D] \setminus \{E\}$
          30. **Return** extractHierarchy($K$)
Definition 1. Let $U$ be a set of elements and $R$ a binary relation over $U$, i.e., a subset of $U \times U$. The set of reachable elements $\text{reachable}(C, R)$ from $C \in U$ in $R$ contains all $D \in U$, such that there exists a path of the form $\langle C, C_1 \rangle \in R, \langle C_1, C_2 \rangle \in R, \ldots, \langle C_n, D \rangle \in R$.

A hierarchy extraction $\text{extractHierarchy}(R)$ is a process that given $R$ returns a pair $(H, \rho)$, where $H$ is a transitively reduced, strict partial-order of elements from $U$, and $\rho$ is a mapping $\rho : U \rightarrow 2^U$ that maps elements of $U$ to subsets $S \subseteq U$, such that for every $D, E \in S$, we have $D \in \text{reachable}(E, R)$ and $E \in \text{reachable}(D, R)$.

Intuitively, if $R$ is $K$, then $\text{extractHierarchy}$ firstly partitions $K$ into sets of classes $S$ such that for every $C, D \in S$, $C \equiv D$ is already known, then it picks one representative from each set, and finally constructs a transitively reduced strict partial order.

Definition 2. Consider the notation in Definition 1. A projection $\text{project}(R, S)$ of $R$ to $S \subseteq U$ is a process that returns the following relation: $\{\langle C, D \rangle \mid C, D \in S \text{ and } D \in \text{reachable}(C, R)\}$.

The restriction $R[C]$ of $R$ to $C$ returns all elements $D$ such that $\langle C, D \rangle \in R$.

The algorithm mainly consists of two parts. The first one (lines 1-15) is responsible for the initialisation of $K$ and $P$, which is done by using a new heuristic strategy, while the second one (lines 16-31) uses a mixture of the ET algorithm—as in KP—but a different way of pruning relations from $P$ that does not depend on Algorithm 1. Both phases are detailed below.

Initialisation Phase Our initial experimentations revealed that it is generally not a good strategy to perform satisfiability tests for every atomic class in $O$ in order to initialise $K$ and $P$. Even though such tests can be done quite efficiently with modern reasoners, their number can sometimes be quite large and this can affect the overall performance. It is usually better to perform some satisfiability tests for classes that will provide large pre-models and which are thus rich in (non)subsumption information. This is the case for classes that are likely to appear low in the hierarchy because when checked for satisfiability the (hyper)tableau algorithm also adds all their subsumers in $L(s_0)$ and build a model for them. Thus, non-subsumers of them are also inferred to be non-subsumers of their subsumers.

To implement this strategy the algorithm proceeds as follows: Firstly, it applies a simple structural subsumption algorithm to identify obvious subsumptions in $O$ and instantiate $K$. Then, it applies $\text{extractHierarchy}$ on $K$ to extract a class hierarchy $H$ and collects all classes $C$ such that $\langle \text{owl:Nothing}, C \rangle \in H$. Subsequently, for each such $C$ it performs a satisfiability test and if the class is satisfiable the constructed pre-model can be used to determine new known and possible subsumers as illustrated in lines 11-15. Note, however, that in order to avoid redundancies, $C$ is tested only if $P[C] = \emptyset$ (line 5). Then, in HermiT’s case, the pre-model reading process proceeds as follows: if $D$ was added to $L(s_0)$
deterministically, then $D$ is guaranteed to be a subsumer of $C$ [7] and $(C, D)$ can be added to $K$. Subsequently, possible subsumers are considered for every class that appears anywhere in the pre-model. If $D(s) \in \mathcal{A}$ and no possible subsumer for $D$ is known yet, $P[D]$ is initialised to contain all atomic classes in $\mathcal{L}(s)$. If $P[D]$ is non-empty, then $\mathcal{L}(s)$ is used to prune relations from $P$ for $D$. Note, that $P[D]$ cannot become empty again as it will always contain at-least $D$.

Consider, for example, an ontology $O$ containing axiom (4) as well as the following axioms:

\[
\text{SubClassOf}(D \text{ ObjectUnionOf}(E \ F)) \quad (6)
\]
\[
\text{SubClassOf}(G \text{ ObjectSomeValuesFrom}(op\ 2\ D)) \quad (7)
\]

Firstly, structural subsumption initialises $K$ to contain $K[X] = \{X, \text{owl:Thing}\}$ for every $X \in C_O$ and $K[\text{owl:Nothing}] = C_O$. Thus, $\text{ToTest}$ will contain the entries $C, D, E, F$ and $G$. Assume, then, that $C$ is picked first and a pre-model for $C(s_0)$ is generated. Due to axiom (4), $s_0$ must be related to an instance of $D$, say $s_1$. Since $D \in \mathcal{L}(s_1)$ a pre-model for $D$ is also generated. Due to axiom (6) and the ObjectUnionOf constructor the reasoner can pick randomly between $E$ and $F$ to add to $\mathcal{L}(s_1)$; assume it picks $E$. Then, the reasoner terminates returning $\mathcal{A}$, which can be used to infer that $P[C] = \{C\}$ and $P[E] = P[D] = \{D, E\}$. On the next iteration, $D$ is selected from the list but $P[D] \neq \emptyset$, thus the selection process continues. This avoids the re-generation of a pre-model for $D$, which has been done already in a previous iteration. At some point $G$ is picked and a model for $G(s_0)$ is built. As before, due to Axiom (7) the reasoner relates $s_0$ with some fresh $s_1$ by property $op\ 2$, such that $D \in \mathcal{L}(s_1)$. Again, axiom (6) has to be considered, but assume now that $F$ is added to $\mathcal{L}(s_1)$ instead. Since, $P[D] \neq \emptyset$ from the previous iteration $\mathcal{L}(s_1)$ can be used to prune $P[D]$. More precisely, since $E \notin \mathcal{L}(s_1)$, $E$ is removed from $P[D]$.

Note that the model reading process is performed only if the tested class is satisfiable. In an ontology with many unsatisfiable classes the initialisation process might not behave well. To avoid this situation the algorithm proceeds as follows: If the tested class was unsatisfiable, $\mathcal{H}$ is traversed “upwards” until a satisfiable class is found. During this process each time an unsatisfiable class is found this unsatisfiability is propagated to all its descendants (lines 7-10). Apart from making initialisation more robust, this approach has an additional potential benefit. In ontologies with many unsatisfiable classes it might be possible to identify unsatisfiable classes without performing actual satisfiability tests. An example of such an ontology is FMA [8] which this algorithm coupled with HermiT can classify very efficiently compared to the standard ET strategy.

**Classification Phase** After the end of the first phase all subsumers of a class might have been identified. Note that this might even be the case for classes for which a model was never explicitly generated (line 6). In our running example all possible subsumers in $P[D]$ (i.e., $D$ are known in $K[D]$). In order not to keep these classes around and to save memory, the algorithm identifies only those classes for which there are still unknown possible subsumers (lines 16-17). Note
that at after this process $P[D]$ might become empty again, but since these classes are classified the algorithm does not operate on them again.

For the unclassified classes the algorithm proceeds as follows: A class $C$ is picked and a transitively reduced strict-partial order $\mathcal{H}$ of its unknown subsumers is generated. Then the ET algorithm is applied for $C$ over $\mathcal{H}$ to identify its (non)subsumers. In contrast to KP, this part has been enriched with the following operations: For $D$ a possible unknown subsumer of $C$, if $C \sqcap \neg D$ (denoting $C$ and the negation of $D$) if satisfiable, i.e., $O \not\models C \sqsubseteq D$, then the constructed pre-model can again be used to prune non-subsumers as done in the first phase. This process is performed in place of Algorithm 1 as it provides an efficient pruning strategy. Another interesting consequence in interleaving pruning with subsumption checking is that during this process, other unknown possible subsumers of $C$, which will be tested in a subsequent iteration, might get pruned.

To exploit this before generating a model for $C \sqcap \neg D$, the algorithm checks if $D$ is still in the set of possible subsumer of $C$ (line 24).

Finally, all unknown possible subsumers have been tested and $K$ contains all subsumption relations. It can thus be used to construct the final class hierarchy.

### 3.1 Further Comparisons with the KP Algorithm

We have already illustrated major differences between Algorithm 2 and KP, like the initialisation of $K$ and $P$ and the lack of Algorithm 1 for pruning relations from $P$. In the following, we illustrate some further differences and comment further on the pruning algorithm.

- **Memory Efficiency:** There are several differences between the two algorithms motivated by memory concerns of KP. Firstly, recall that KP requires $K$ to be transitively closed. This is clearly not a good strategy in large ontologies like FMA or SNOMED, which contain thousands of classes. Secondly, it assumes that $P \supseteq K$, i.e. all known subsumptions (including those derived by the transitive closure) are contained in $P$. In contrast, Algorithm 2 uses a graph reachability algorithm to identify whether a pair $\langle C, D \rangle$ would belong to the transitive closure of $K$. Additionally, as we have presented above, information about classified classes is removed from $P$.

- **Pruning Algorithm:** Although Algorithm 2 does not directly call Algorithm 1, it does indirectly implement parts of it through the pre-model reading process. For example, for $B \in P[A]$, if $O \not\models A \sqsubseteq B$, then $L(s_0)$ contains $A$ and $\neg B$ and, in HermiT's case, all subsumers of $A$ as well. Consequently, $B$ can be readily inferred to be a non-subsumer of all the subsumers of $A$, as done in the first loop of Algorithm 1. The second loop of Algorithm 1 prunes possible subsumptions when new positive ones are inferred. Nevertheless, after the first phase has finished, most remaining unknown subsumptions are usually non-subsumptions, thus this part of Algorithm 1 rarely applies. Even if new subsumption relations are inferred, our experience has shown that this pruning pattern is quite complicated and rarely applies in practice. Consequently, the benefits from avoiding such an expensive algorithm outperform the possibilities of pruning relations using it.
- **Bottom-up procedure:** As is usually done in classification algorithms (like in ET), KP includes a bottom-up phase where the unknown possible subsumees of an unclassified class $C$ are explored. This procedure is natural for the ET algorithm as for a class $C$ one tries to determine its position into a partially constructed hierarchy. Nevertheless, since the new algorithm does not work under this principle we chose to omit such a process. We believe that this way the algorithm is much simpler and there is no need to retain doubly-linked structures for efficiently retrieving both successors and predecessors of $C$ in $K$ and $P$. Note that the algorithm is still complete even without such a step, since if for $C$ an unknown possible child $D$ exists, then $C \in P[D]$ and this subsumption would be tested when $D$ is picked.

4 **Object Property Classification**

Classification of properties has, to the best of our knowledge, not been discussed in the literature. This is most likely the case because in logics up to OWL Lite, all subsumption relations between properties can be computed by building the transitive and reflexive closure of the asserted property hierarchy. Such an operation is cheap to implement and no satisfiability tests are required in the reasoner. In the presence of nominals or property chains, however, an algorithm that builds the transitive reflexive closure of the asserted hierarchy might miss subsumptions. Consider, for example, an ontology containing the axioms:

\[
\begin{align*}
\text{ObjectPropertyDomain}(op_1 \text{ObjectOneOf}(a)) & \quad \text{ObjectPropertyRange}(op_1 \text{ObjectOneOf}(b)) \quad (8) \\
\text{ObjectPropertyDomain}(op_2 \text{ObjectOneOf}(a)) & \quad \text{ObjectPropertyRange}(op_2 \text{ObjectOneOf}(b)) \quad (9)
\end{align*}
\]

Due to the nominal semantics, each $op_1$ property must relate $a$ to $b$ and so does $op_2$ and these axioms imply $op_1 \equiv op_2$ although no property inclusion is given.

Another interesting example uses property chains, which induce a property subsumption between $op_1$ and $op_2$:

\[
\begin{align*}
\text{SubClassOf}(\text{owl:Thing} \text{ObjectSomeValuesFrom}(op \text{owl:Thing})) & \quad (10) \\
\text{SubObjectPropertyOf}(\text{ObjectPropertyChain}(op_1 \text{op} \text{ObjectInverseOf}(op))) & \quad (11)
\end{align*}
\]

In this case, whenever there is an $op_1$ successor for an individual $i_1$, say $i_2$, $i_2$ has an $op$-successor, say $i_3$, due to Axiom (10). Then, we have $op_1(i_1, i_2)$, $op(i_2, i_3)$, $\text{ObjectInverseOf}(op)(i_3, i_2)$, which implies $op_2(i_1, i_2)$ due to Axiom (11) and the subsumption $op_1 \subseteq op_2$ holds.

The initial property classification algorithm in the HermiT reasoner was based on the ET algorithm. As for classes, one could test whether $op(i_1, i_2)$ and $\neg op_2(i_1, i_2)$ is satisfiable with $i_1$, $i_2$ individual names not occurring in the ontology. This works, however, only for simple properties [6], where simple properties are roughly those that do not occur in property chains or transitivity axioms.

The problem with complex properties (i.e., non-simple ones) is that they are not necessarily made explicit in the pre-models that a (hyper-)tableau reasoner builds. This is because property chains and transitivity axioms can introduce
many additional property relations and materialising them can require a significant amount of memory. Furthermore, by adding such links explicitly, the pre-models do not have a kind of tree shape any more, which makes several parts of the implementation more complex. Instead, property chains and transitivity axioms are usually encoded into several subclass axioms that propagate classes along paths in the pre-model such that when building a model from the pre-model, it is guaranteed that adding all required property relationships does not violate any of the axioms. Roughly speaking, to eliminate a property inclusion SubObjectPropertyOf(ObjectPropertyChain(op op)), stating transitivity of op, each axiom containing a universal quantifier over op is rewritten. E.g., we replace each axiom of form (12) with axioms of the form (13) to (15)

\[
\begin{align*}
\text{SubClassOf}(C \text{ ObjectAllValuesFrom}(op D)) & \quad (12) \\
\text{SubClassOf}(C \text{ ObjectAllValuesFrom}(op D_{op})) & \quad (13) \\
\text{SubClassOf}(D_{op} D) & \quad (14) \\
\text{SubClassOf}(D_{op} \text{ ObjectAllValuesFrom}(op D_{op})) & \quad (15)
\end{align*}
\]

where \(D_{op}\) is a fresh class. In order to compute all axioms required to eliminate all property inclusions, automata are constructed for each complex property, from which we can then read off the required extra axioms [9]. In order for the elimination to work as desired, it is necessary to rewrite negative property assertions for complex properties. I.e., each assertion of form (16) is rewritten into an axiom of the form (17)

\[
\begin{align*}
\text{NegativeDataPropertyAssertion}(op a b) & \quad (16) \\
\text{ClassAssertion}(\text{ObjectAllValuesFrom}(op \text{ ObjectComplementOf(ObjectOneOf(b))) a}) & \quad (17)
\end{align*}
\]

where (17) states that \(a\) belongs to the class of individuals for which all \(op\)-successors are not \(b\). The universal quantifier then triggers the generation of further axioms in the property chain elimination as described above.

The fact that complex properties are not necessarily made explicit in the model abstractions, further means that we cannot read of non-subsumptions. I.e., when \(op_1(i_1, i_2)\) is in the pre-model constructed by the reasoner, while \(op_2(i_1, i_2)\) is not for \(op_2\) a complex property, we cannot conclude \(op_1 \not\subseteq op_2\). Thus, complex properties significantly reduce the possibility of pruning the search space, making property classification to some extent harder than class classification. It is worth pointing out that HermiT 1.2.2 incorrectly assumes \(op_1 \not\subseteq op_2\) in the above described case. The HermiT version used in the evaluation is corrected, but after the correction the performance decreased significantly.

In order to address these issues, we developed a new property classification technique, which reduces property classification to class classification. Any class classification algorithm, e.g., the optimised one described above, can then be used to classify the property hierarchy, while all the search space pruning optimisations are still applicable. In order to classify the object properties of an ontology, we proceed as follows:
Definition 3. Let \( \mathcal{O} \) be an OWL 2 DL ontology and \( \text{OPE} \) the object properties and inverse object properties occurring in \( \mathcal{O} \). An object property to class mapping w.r.t. \( \mathcal{O} \) is a total and injective function \( \tau \) from \( \text{OPE} \) to class names not occurring in \( \mathcal{O} \). Let \( C_f \) be a class name not occurring in \( \mathcal{O} \) and the range of \( \tau \). The object property hierarchy induced by \( \tau \) w.r.t. \( \mathcal{O} \), denoted \( \mathcal{H}_\mathcal{O}^\tau \), is the transitive reduction of the relation \( \{\langle \text{op}_1,\text{op}_2 \rangle \mid \text{op}_1,\text{op}_2 \in \text{OPE} \text{ and } \mathcal{O} \models \tau(\text{op}_1) \sqsubseteq \tau(\text{op}_2)\} \), where \( \mathcal{O}_\tau \) is an extension of \( \mathcal{O} \) with an axiom

\[
\text{EquivalentClasses}(\text{C}_{\text{op}} \text{ ObjectSomeValuesFrom}(\text{op} C_f))
\]

for each object property \( \text{op} \in \text{OPE} \) and \( \tau(\text{op}) = \text{C}_{\text{op}} \).

Intuitively, to test \( \text{op}_1 \sqsubseteq^? \text{op}_2 \), we test \( C_1 \sqsubseteq^? C_2 \) with \( C_1 \) and \( C_2 \) the representative concepts introduced by \( \tau \) for \( \text{op}_1 \) and \( \text{op}_2 \), respectively. The reasoner then constructs a pre-model for the additional facts \( C_1(i) \cap \neg C_2(i) \) as for class classification. The axioms in \( \mathcal{O}_\tau \) then cause the addition of an \( \text{op}_1 \)-successor for \( i \), say \( i' \) with \( C_f(i') \). If, due to other axioms in \( \mathcal{O} \), \( i' \) also must be an \( \text{op}_2 \)-successor of \( i \), the axiom in \( \mathcal{O}_\tau \) for \( \text{op}_2 \) causes the addition of \( C_2(i) \), which then causes a clash and confirms the subsumption. In order to handle complex properties, we eliminate them as described above in \( \mathcal{O}_\tau \). Reading off non-subsumptions and pruning the set of possible subsumers then works exactly as described for classes.

The following theorem shows that such an encoding of the object property classification problem into a class classification problem is indeed valid.

Theorem 1. Let \( \mathcal{O} \) be an OWL 2 DL ontology with \( \text{op}_1,\text{op}_2 \in \text{OPE} \), \( \tau \) an object property to class mapping w.r.t. \( \mathcal{O} \), and \( \mathcal{H}_\mathcal{O}^\tau \) the object property hierarchy induced by \( \tau \) w.r.t. \( \mathcal{O} \). Then \( \mathcal{O} \models \text{op}_1 \sqsubseteq \text{op}_2 \text{ iff } \langle \text{op}_1,\text{op}_2 \rangle \in \mathcal{H}_\mathcal{O}^\tau \).

Proof: For the if direction: We show the contrapositive, i.e., if \( \mathcal{O} \not\models \text{op}_1 \sqsubseteq \text{op}_2 \), then \( \langle \text{op}_1,\text{op}_2 \rangle \notin \mathcal{H}_\mathcal{O}^\tau \). Let \( I \) be a model of \( \mathcal{O} \) such that \( I \not\models \text{op}_1 \sqsubseteq \text{op}_2 \) and let \( \langle i_1,i_2 \rangle \) be in the extension of \( \text{op}_1 \), but not in the extension of \( \text{op}_2 \). Such an interpretation and pair of individuals exists by definition of non-subsumption w.r.t. \( \mathcal{O} \). Since \( \langle \text{op}_1,\text{op}_2 \rangle \notin \mathcal{H}_\mathcal{O}^\tau \text{ iff } \mathcal{O}_\tau \not\models \tau(\text{op}_1) \sqsubseteq \tau(\text{op}_2) \), we show how we can (conservatively) extend \( I \) into a model \( I' \) of \( \mathcal{O}' \) such that \( I' \not\models \tau(\text{op}_1) \sqsubseteq \tau(\text{op}_2) \). In order to interpret the new symbols in \( \mathcal{O}_\tau \), we set the extension of \( C_f \) to \( \{i_2\} \) and, for each \( \text{op} \in \text{OPE} \), we interpret \( \tau(\text{op}) \) as the set of individuals \( i \) such that \( \langle i,i_2 \rangle \) is in the extension of \( \text{op} \). It is not hard to see that \( I' \) satisfies the additional axioms in \( \mathcal{O}' \). Since \( \langle i_1,i_2 \rangle \) is in the extension of \( \text{op} \) and \( i_2 \) is in the extension of \( C_f \), we have that \( i_1 \) is in the extension of \( \tau(\text{op}_1) \). Since \( \langle i_1,i_2 \rangle \) is not in the extension of \( \text{op}_2 \) by assumption, we have that \( i_1 \) is not in the extension of \( \tau(\text{op}_2) \), which proves the claim.

For the only if direction, we again show the contrapositive, i.e., we show that if \( \mathcal{O}_\tau \not\models \tau(\text{op}_1) \sqsubseteq \tau(\text{op}_2) \), then \( \mathcal{O} \not\models \text{op}_1 \sqsubseteq \text{op}_2 \). Let \( I \) be a model of \( \mathcal{O}_\tau \) such that there is an individual \( i_1 \) in the extension of \( \tau(\text{op}_1) \) and not in the extension of \( \tau(\text{op}_2) \). Such a model \( I \) and individual \( i_1 \) exists by definition of non-entailment and subsumption. Since \( I \models \mathcal{O}_\tau \), \( i_1 \) is in the extension of \( \tau(\text{op}_1) \), the axiom for \( \text{op}_1 \) in \( \mathcal{O}_\tau \) means that there is an individual \( i_2 \) in the extension of \( C_f \) such that
⟨i₁, i₂⟩ is in the extension of op₁. Since O₂ is an extension of O we have by the monotonicity of interpretations that I |= O. Assume to the contrary of what is to be shown that ⟨i₁, i₂⟩ is in the extension of op₂. Then the axiom for op₂ in O² together with the fact that i₂ is in the extension of C_f implies that i₁ is in the extension of C₂, which is a contradiction.

4.1 Data Property Classification

Data property classification might seem easier than object property classification, since the problematic constructors such as nominals or property chains do not apply to data properties, and data properties are never complex. Thus, one could easily conclude that for data properties one can get away with just computing the transitive reflexive closure of the asserted data property subsumptions. This is, however, not the case since the DataOneOf construct in OWL behaves as the ObjectOneOf constructor and we can easily adjust axioms (8) and (9) to work with data properties and DataOneOf in the range axiom.

The next problem for classifying the data properties is how a data property subsumption test can be implemented. Since data properties are always simple, to test d₁ ⊑ d₂ one could try adding dp₁(i₁, i₂) ⊓ ¬dp₂(i₁, i₂) to the ontology, for two fresh individuals i₁, i₂, and try to build a pre-model. For data properties, however, there is nothing like a fresh data value and we cannot just pick a data value of our choice. We solve this problem by inventing a dummy datatype D, which is considered to be non-disjoint with all datatypes in the OWL 2 datatype map, i.e., its value space can be intersected or unioned with any other data range without causing a clash. The only constraint for this datatypes is that a data value cannot be in D and its complement at the same time. The datatype checker is extended accordingly. In order to test dp₁ ⊑ dp₂, the reasoner then tries to build a model for the ontology extended with

\[
\text{ClassAssertion(DataSomeValuesFrom}(d₁ \text{ D})) \quad (18)
\]
\[
\text{ClassAssertion(DataAllValuesFrom}(d₂ \text{ DataComplementOf}(D))) \quad (19)
\]

for i a fresh individual. There is, however, still a problem with this approach: The datatype checker implements the datatype checking procedure proposed for OWL 2 [10], which means that if an individual i has a data property successor n, then the datatype checker checks whether there are only finitely many values that n can take. If that is the case, the datatype checker tries to find an assignment of data values for n considering also “relevant” siblings of n, where siblings are related to the same individual i as n. A sibling n’ is relevant if it can also only have finitely many possible data values and the assignment must be different from the one for n due to an inequality between n and n’ (e.g., caused by an at-least restriction). In order to handle D properly, an inequality is generated between sibling nodes where one must be in the extension of D and the other must be in the complement of D. This guarantees that the two nodes do not get the same data value assignment from the datatype checker. Note that even if the datatype checker has to assign the same data value to two nodes, then the nodes are not
merged. E.g., if an individual is required to have the integer 1 as \( dp_1 \)-successor and the integer 1 as \( dp_2 \)-successor, then there are two different successors, say \( n_1 \) and \( n_2 \), in the pre-model that HermiT generates. Thus, reading off a non-
subsumption is not correct in this case. This was again overlooked in HermiT 1.2.2 and after fixing this, the data property classification took significantly longer.

We can, however, equally use the property to class classification encoding as described above for object properties. This encoding has the benefit that we can read off subsumptions and non-subsumptions because they materialise in the classes introduced by the encoding and our “dummy” datatype \( D \) behaves like the fresh concept \( C_f \) introduced for the object properties.

**Definition 4.** Let \( O \) be an OWL 2 DL ontology and \( D \) a dummy datatype as discussed above. A data property to class mapping w.r.t. \( O \) is a total and injective function \( \sigma \) from \( DP \) to \( C \setminus C_O \). The data property hierarchy induced by \( \sigma \) w.r.t. \( O \), denoted \( H^\sigma_O \), is the transitive reduction of the relation \( \{ \langle dp_1, dp_2 \rangle \mid dp_1, dp_2 \in DP \text{ and } O \models \sigma(dp_1) \sqsubseteq \sigma(dp_2) \} \), where \( O \sigma \) is an extension of \( O \) with an axiom

\[
\text{EquivalentClasses}(C_{dp}, DataSomeValuesFrom(dp \ D))
\]

for each data property \( dp \in DP \) and \( \sigma(dp) = C_{dp} \).

A proof that the proposed encoding is indeed valid can be given in much the same way as for Theorem 1, and we omit it here for brevity.

**Theorem 2.** Let \( O \) be an OWL 2 DL ontology with \( dp_1, dp_2 \in DP_O \), \( \sigma \) a data property to class mapping w.r.t. \( O \), and \( H^\sigma_O \) the data property hierarchy induced by \( \sigma \) w.r.t. \( O \). Then \( O \models dp_1 \sqsubseteq dp_2 \) iff \( \langle dp_1, dp_2 \rangle \in H^\sigma_O \).

## 5 Evaluation

We have implemented Algorithm 2 and the property classification encodings in the HermiT 1.2.4\(^1\) (hyper)tableau reasoner (HermiT 1.2.3 does still use the ET algorithm for data property classification). To evaluate the performance of HermiT 1.2.4, we compared it against HermiT 1.2.2a,\(^2\) which implements the ET strategy. For that purpose we selected several well-known real-world ontologies to be classified. We selected two versions of the GALEN ontology,\(^3\) several ontologies from the Open Biological Ontologies (OBO) Foundry,\(^4\) the Food and Wine ontology from the OWL Guide,\(^5\) two versions of the Foundational Model of Anatomy,\(^6\) (FMA) and ontologies from the Gardiner ontology suite.\(^7\)

---

1. [http://hermit-reasoner.com/download/1.2.4](http://hermit-reasoner.com/download/1.2.4)
2. [http://hermit-reasoner.com/download/1.2.2a](http://hermit-reasoner.com/download/1.2.2a)
5. [http://www.w3.org/TR/owl-guide/](http://www.w3.org/TR/owl-guide/)
7. [http://www.cs.man.ac.uk/~horrocks/testing/owlOntologies.zip](http://www.cs.man.ac.uk/~horrocks/testing/owlOntologies.zip)
The systems were used to classify the classes and properties of all aforementioned ontologies. For each ontology we measured the classification time (in sec) as well as the number of actual reasoning tests that are requested by the methods to the reasoner (both satisfiability and subsumption tests). All experiments were performed on a UNIX machine of an Intel x86 64bit Cluster on one node with two quad core 2.8GHz processors and Java 1.5 allowing 2GB of memory to Java. The results are summarised in Table 1. The upper part of the table contains all the deterministic ontologies, i.e., roughly speaking those that do not use disjunctive constructors, while the lower part contains all the non-deterministic ones. A "-" in the table indicates that the ontology did not contain any data properties and OoM stands for Out of Memory.

As Table 1 shows, HermiT 1.2.4 is in all cases much faster than the ET strategy of HermiT 1.2.2a, sometimes one or even two orders of a magnitude. This is especially the case in property classification where, as we have explained in the previous section, none of HermiT’s standard optimisations can be applied and one relies completely on the insertion strategy of ET to reduce the number of required subsumption tests. On the other hand, our property classification encoding brings all the benefits of the class classification optimisations achieving a very good and robust performance. These results show that it is practically feasible to perform correct property classification through reasoning, instead of the cheap but incomplete transitive closure algorithms. The picture is equally good in class classification. In most cases the new strategy has reduced the classification time by several times. Especially note the significant performance gain in the classification of both versions of FMA. This is due to the heuristic implemented in lines 7-10 and the large number of unsatisfiable classes in FMA.

The good performance results are also confirmed by the significantly reduced number of reasoning tests that HermiT 1.2.4 requires to build the class and property hierarchies.

<table>
<thead>
<tr>
<th>Ontology</th>
<th>Classes</th>
<th></th>
<th></th>
<th>Object Properties</th>
<th></th>
<th></th>
<th></th>
<th>Data Properties</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.2.2a</td>
<td>1.2.4</td>
<td></td>
<td>1.2.2a</td>
<td>1.2.4</td>
<td></td>
<td>1.2.2a</td>
<td>1.2.4</td>
<td></td>
<td>1.2.2a</td>
<td>1.2.4</td>
</tr>
<tr>
<td>Tests Time Tests Time Tests Time Tests Time Tests Time Tests Time</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GALEN-d 2744 3.5 3385 2.5 6073 439 &lt;1</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GALEN-und 2744 28.2 3793 6.2 6001 459 198 &lt;1</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GO 19260 43 14271 3.4 4 &lt;1 3 &lt;1</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GO_XP 27880 118 19971 28 9 10.4 6 4.8</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>chebi 20693 69 13505 10.8 26 59.8 12 18.1</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NCI 27652 71 21398 5 71 &lt;1 72 &lt;1</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AEO 2285 2.1 358 1.6 214 5.9 35 &lt;1</td>
<td>223 4.6 28 &lt;1</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>substance 4569 15.8 2567 11.7 962 23.5 108 &lt;1</td>
<td>957 22.5 37 &lt;1</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ProPreO 1441 7.3 1093 2.8 518 3 33 &lt;1</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OBI 12444 254 3022 169 2278 310 53 3.3</td>
<td>39 6 7 &lt;1</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Food-Wine 382 18.2 226 10.4 65 11.6 14 2</td>
<td>4 &lt;1 3 &lt;1</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FMA-CNS 49716 7973 14527 934 8281 16687 171 8</td>
<td>283 469 30 &lt;1</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FMA-noCNS 112531 12421 27181 4629 174 64 54 1.4</td>
<td>OoM OoM</td>
<td>OoM OoM</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Results of Evaluation for Class and Property Classification
Table 2. Number of Tests Performed by HermiT 1.2.4 Compared to KP

<table>
<thead>
<tr>
<th></th>
<th>GO</th>
<th>GALEN</th>
<th>NCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>KP</td>
<td>32614</td>
<td>4657</td>
<td>48389</td>
</tr>
<tr>
<td>HermiT 1.2.4</td>
<td>27239</td>
<td>4906</td>
<td>40965</td>
</tr>
</tbody>
</table>

property hierarchies. The only case where HermiT 1.2.4 performs more tests is on GALEN. This is because in deterministic ontologies HermiT 1.2.2a uses satisfiability tests and the pre-model reading technique [7] to identify the subsumers of a concept. Since the ontology is deterministic one such test identifies all its subsumers. Instead, our method does not perform satisfiability tests for every concept, thus when the first phase finishes some unknown possible subsumers need to be checked in the second phase. Especially in GALEN most of them are subsumers, thus the pruning step in lines 26-27 rarely applies. Nevertheless, such reasoning tests are usually very fast, thus overall the system still performs better than HermiT 1.2.2a. Especially in GALEN-und where satisfiability tests are expensive, one can observe the benefits of not performing satisfiability tests for every class by the favourable performance results.

As a last experiment we wanted to compare our system with the implementation of KP evaluated in [1]. We considered the three specialty constructed ontologies used in the evaluation there and we compared the number of tests performed by our method with the number of tests published in [1]. Table 2 summarises these results. We can note again that for all ontologies but GALEN our system requires much less reasoning tests. In the case of GALEN the same observations as above apply. Unfortunately, the implementation was not available to conduct a performance comparison.

6 Conclusions

This paper considers the problem of efficiently classifying OWL ontologies. In comparison to previous approaches we work both on class classification, as well as object and data property classification. To the base of our knowledge property classification has not been discussed in the literature before.

We begin by giving a new class classification algorithm, that is mainly motivated by the work in [1]. This algorithm revises several parts of the algorithm in [1] and closes several gaps that have been left open. In more detail, we provide a novel heuristic strategy for instantiating the relations $K$ and $P$, an efficient pruning strategy and a novel heuristic for pruning unsatisfiable concepts. Additionally, our new algorithm improves several memory and efficiency issues of the algorithm in [1].

Subsequently, we turn our attention to property classification. We give illustrative examples that show why traditional cheap algorithms based on the transitive reflexive closure of the asserted property hierarchy might fail to be complete. We then discuss the difficulties in reusing well-known optimisations directly on the case of properties, and then present a novel way of encoding the
property classification problem into a class classification problem. An immediate benefit of such encoding is that we can reuse our previously introduced optimised class classification.

Finally, we have implemented all classification strategies in the HermiT reasoner and evaluated its performance against the standard (ET) classification method of HermiT. The results where very encouraging and classification times have been significantly improved. Especially in the case of properties our experiments show that it is not impractical to perform exact complete property classification.

We currently work on extending the KP algorithm for realisation, i.e., the task of computing, for each individual \( i \) in an ontology, the most specific classes \( C \) such that \( i \) is an instance of \( C \), and similarly for computing property instances. Initial results show that also for these reasoning tasks significant performance benefits can be achieved.

**Acknowledgements** The presented work is funded by the EPSRC project HermiT: Reasoning with Large Ontologies.

**References**