Classifying and Querying Very Large Taxonomies with Bit-Vector Encoding

Hassan Aït-Kaci $^{\star 1}$ and Samir Amir $^{\star 2}$

* *LIRIS—Département Informatique, Université Claude Bernard Lyon 1*
Villeurbanne, France

^{1} `hassan.ait-kaci@univ-lyon1.fr`
^{2} `samir.amir@univ-lyon1.fr`

Abstract—This article addresses the question of how efficiently Semantic Web (SW) reasoners perform in processing (classifying and querying) taxonomies of enormous size. We use a bit-vector encoding technique to implement taxonomic concept classification and Boolean-query answering. We describe the technique we have used, which achieves high performance, and discuss implementation issues. We compare the performance of our implementation with those of the best existing SW reasoning systems over several very large taxonomies under the exact same conditions for so-called TBox reasoning. The results show that our system is among the best for concept classification and several orders-of-magnitude more efficient in terms of response time for query answering. We present these results in detail and comment them. We also discuss future work regarding how our method could be further optimized to handle even larger ontologies without performance degradation.

Index Terms—Scalable Taxonomic Reasoning, Bit-Vector Encoding, Implementation Techniques

◆

1 INTRODUCTION

THERE exist many and varied formal systems purporting to express and use knowledge for inference. However, despite this variety, they all share a common trait: knowledge is always organized into an "is-a" conceptual taxonomy where a concept denotes the set of all its instances. In such a taxonomy, a concept C_1 is deemed a subconcept of a concept C_2 whenever the set denoted by C_1 is a subset of the set denoted by C_2. This is what "C_1 is-a C_2" means. For example, "*employee* is-a *person*" means that all instances of the concept *employee* are also instances of the concept *person*. Hence, reasoning performance in all such systems will need to rely in a crucial way upon the performance of the underlying taxonomic reasoning. In particular, it will be necessary to identify all the maximal concepts that are subconcepts of two given concepts, those that are minimal superconcepts of two given concepts, or those that are incompatible with a given concept. These operations correspond respectively to concept conjunction (*i.e.*, intersection of the denoted sets), concept disjunction (*i.e.*, union of the denoted sets), and concept negation (*i.e.*, complement of the denoted set). Therefore, basic Propositional Algebra is central to *all* taxonomic reasoners.

The issue, then, is to enable the efficient evaluation of propositional expressions involving propositional symbols representing partially-ordered concepts making up these conceptual taxonomies. However, conceptual taxonomies can be of enormous size, making such evaluation a challenge to perform efficiently.

In this document, we address two topics in the above context: (1) robust and scalable taxonomic reasoning; and, (2) evaluation of semantic web technologies for such reasoning. Regarding the first topic, we demonstrate how a method for taxonomic reasoning based on bit-vector encoding we have implemented is both robust and scalable on very large taxonomies derived from real-life ontologies. As for the second topic, we proceed to a comparative evaluation of the best existing Semantic Web (SW) reasoners on the same taxonomies under the same conditions. This comparative study illustrates clearly the mutual stance in terms of performance for each reasoner with respect to one another. It also does so for our system, thus putting its performance in context with the state of the art.

We measured the performance of our system and compared it with those of the best existing SW reasoning systems over several very large taxonomies under the exact same conditions for so-called TBox reasoning.[1] The results show that our system is among the best for concept classification and several orders-of-magnitude more efficient in terms of response time for query answering. We present these results in detail and comment them. We also discuss future work regarding how out method could be further optimized and can be made to scale up to even larger taxonomies without performance degradation.

The rest of this paper is organized as follows. In Section 2, we explain the principle upon which our method is based and discuss its implementation and use for taxonomic reasoning.

---

[1] In Semantic Web lingo, a Knowledge Base (KB) is defined as a formal ontology consisting of two parts (or "boxes"): (1) a *Terminological Box* (abbreviated as *TBox*); and, (2) an *Assertional Box* (abbreviated as *ABox*). The TBox contains the formal axioms that define the structure and semantic properties of the actual instance data; which instance data constitute the ABox. In Database lingo, the TBox corresponds to the schema and the ABox to the actual data.

In Section 2.1, we explain the formal principles underpinning our method. In Section 2.2, we make some important remarks dealing with implementation. In Section 3, we report our experimental work in the context of state of the art in Semantic Web reasoning, which is cover in Section 3.1. In Section 3.2, we describe our experimental setup. In Section 3.3, we present the results of these experiments, and discuss these results in Section 3.4. We conclude in Section 4 with a summary of our contribution and perspectives. In the appendix, we give a detailed formal specification of a memory-efficient representation for extremely large binary codes, and an overview of the tool we have implemented to verify the claims of this paper, and web service we have set up to access it.

2 OUR METHOD

2.1 Principle

In this section, we give a self-contained summary of the method we have implemented in order to measure its performance for classification of bare taxonomies and query answering of Boolean queries.

Our method is an implementation in Java of a technique described in [1]. It consists in representing the elements of a taxonomy (*i.e.*, an arbitrary poset) as bit vectors. Thus, each element has a code (a bit vector) carrying a "`1`" in the position corresponding to the index of any other elements that it subsumes. In this manner, the three Boolean operations on sorts are readily and efficiently performed as their corresponding operations on bit-vectors. However, for this to be possible, these bit vectors must be encoded as the reflexive transitive closure of the "is-a" relation obtained from subsort declarations.

How to compute such a closure has been well-known—*e.g.*, the Warshall-Strassen method using clever matrix multiplication tricks [2]–[4]. However, for a poset of n elements, this method has quite a large time complexity—even with the best known algorithm to date, it is $\mathcal{O}(n^{2.23727})$ [5], [6].[2] In fact, in practice, the straightforward $\mathcal{O}(n^3)$ in-place multiplication method known as Warshall's Algorithm [8], [9] is used in most cases.

Figure 1 gives Java code for Warshall's algorithm performed "in-place" on the binary codes of sorts stored in an array `SORTS`. The array `SORTS` contains the set of sorts as bit-set objects. A bit-set object has a field named "`code`" which is a bit vector. The class of bit vectors is endowed with a method `get(int)` that returns the value of its bit in the specified position as a `boolean`, and a method `set(int,boolean)` that sets its bit in the specified position to the specified Boolean value.

Now, while Warshall's algorithm may be viable for relatively small posets, it simply becomes unusable for posets of the size of the taxonomies we are considering.

```
int n = SORTS.size();
for (int k = 0; k < n; k++)
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
      if (!SORTS[i].code.get(j))
        SORTS[i].code.set(j,
                          SORTS[i].code.get(k)
                          &&
                          SORTS[k].code.get(j));
```

Fig. 1: Java code for "in-place" Warshall's algorithm

Note, however, that transitive-closure methods need pay such a high performance cost only due to the fact that they are devised for arbitrary graphs. But concept taxonomies are *not* arbitrary graphs. Namely, a necessary condition for a set of partially-ordered concepts to be semantically consistent is that its graph must be *acyclic*. Thus, a consistent taxonomy must be a directed-acyclic graph (or *dag*) with a least element ($\bot$) and a highest element ($\top$). In [1], it is shown that for such a dag, an $\mathcal{O}(n)$ transitive-closure algorithm exists and is proven correct. This method is described as Algorithm 1.

Algorithm 1 Taxonomy Classification Algorithm

```
 1: procedure CLASSIFY
 2:     $L \leftarrow \mathit{Parents}(\bot)$;
 3:     while $L \neq \emptyset$ do
 4:         for all $x \in L$ do
 5:             $x.code \leftarrow 2^{x.index} \vee \bigvee_{y \in \mathit{Children}(x)} y.code$;
 6:             $x.coded \leftarrow$ true;
 7:         end for
 8:         $L \leftarrow \bigcup_{x \in L} \mathit{Parents}(x)$;
 9:         for all $x \in L$ do
10:             if $\exists y \in \mathit{Children}(x)$ and $\neg y.coded$ then
11:                 $L \leftarrow L - \{x\}$;
12:             end if
13:         end for
14:     end while
15: end procedure
```

The procedure CLASSIFY assumes that each sort s is provided with a set denoted as "*Parents(s)*" and a set denoted as "*Children(s)*" (*viz.*, from the "is-a" declarations of members of the taxonomy). Each sort is an object that has a field (called "*code*"), which is its bit vector representation (initialized to be all zeroes). A sort also has an integer field (called "*index*") which is unique per sort (*viz.*, its index in the taxonomy). Finally, a sort also has a Boolean field (called "*coded*"), which denotes whether or not this sort has been encoded yet (it is initially set to **false**).

Algorithm 1 computes the reflexive-transitive closure of a the "is-a" relation on the set of sorts comprising a taxonomy [1].[3] This algorithm can be explained as follows. It proceeds layer by layer, starting with the parents of $\bot$ (*i.e.*, the minimal sorts in the taxonomy) [Line 2], assigning a code to each element in the current layer to be the bitwise `or` of its children and also setting the bit in its index position [Line 5]. Each time an element is encoded, it is marked to be so by setting its *coded* flag to **true** [Line 6]. Then, a new layer is computed from the current one as the union of all it parents [Line 8] from which any sort that has at least one child not encoded is removed [Line 11]. Indeed, by construction, such sorts can always be reached later. This proceeds until an empty layer is obtained [Line 3]—which is when all sorts have been encoded.

---

[2] To the best of our knowledge, this is the latest best bound as of 2011. However, these algorithms are not implementable due to prohibitive size of constants. For more recent work on parallelizing Strassen's algorithm, see [7]. This, however, requires special harware (GPGPUs).

[3] *op. cit.*, pages 125–126.

This algorithm's main loop [Lines 3–14] clearly visits each sort exactly once, and is thus linear. The auxiliary computation of the next layer [Lines 8–13] has comparatively marginal cost as it can be made efficient using constant access-time data structures for the sets of parents and children, making set operations on them negligible. Also proceeding bottom up has a clear performance advantage for dags such as most concept taxonomies, where sorts tend to have many less parents than they have children.

It is this method that we have implemented, tested, and compared with the best SW reasoners we could retrieve, to run on the very large taxonomies we extracted from existing publicly accessible ontologies of enormous size. The real bonus of this method is, of course, that all three Boolean operations on sorts stay virtually $\mathcal{O}(1)$ irrespective of the size of the taxonomy nor that of the number of concepts in the query. The result of any such query is the set of sorts with codes in the set of maximal common lower-bounds of the computed code.[4] All the above claims are clearly demonstrated on all the performance graphs we are reporting in the next section.

As for incrementality, removing a sort amounts simply to erasing its index position in all codes that have it set. Adding a sort (through a new "is-a" declaration $s_1 < s_2$) is done by restarting the bottom up propagation of the loop [Lines 3–14] starting from the parents of lower of the two sorts, after having reset all its ancestors to the uncoded status. In case of several new "is-a" declarations, the same procedure is applied but starting from the union of the parents of the minimal new sorts after resetting all their ancestors to the uncoded status.

2.2 Implementation

In this section, we discuss implementation issues dealing with detecting and identifying potential cycles in a taxonomy being encoded, and decoding codes into sorts.

2.2.1 Detecting Cycles

This section adds some information concerning the detection and identification of potential cycles in a set of "is-a" declarations specifying a taxonomy.

Problem

Since it expresses a subset ordering among concepts, a taxonomy is supposed to be a directed acyclic graph (dag). However, it may be possible that mistakes occur whereby the "is-a" declarations among concepts contain inconsistencies in the form of cycles. To detect such cycles is thus useful for a classification tool to allow reporting them for correction to the user.

If the general $\mathcal{O}(n^3)$ Warshall algorithm is used to compute transitive-closure codes for all sorts in the taxonomy, then an existing maximal cycle will necessarily imply that all its elements are given equal codes. Indeed, by transitivity, the code of an element denotes the set of all its descendants. But all the elements of a cycle have the same set of descendants, and so their codes must be equal. Such a cycle is in fact an equivalence class for the least equivalence relation containing the declared "is-a" pairs.

One could eliminate all such cycles by collapsing them into a single sort (the class representative), obtaining the quotient set, which is then a dag. However, this is not desirable since such cycles are in all cases errors resulting from inconsistent declarations. In this case, they should be flagged as errors and their contents identified.

Reporting such cycles efficiently can be done by performing a topological reordering of the taxonomy according to the codes that would guarantee that sorts of equal codes are contiguous. Thus, a maximal cycle must be a maximal contiguous sequence of equal-coded sorts in this topological reordering of the taxonomy.

However, for the reasons discussed earlier in this paper, it is not feasible to use Warshall's algorithm to compute the transitive closure on very large taxonomies. In addition, reordering such a very large taxonomy using QuickSort will be on average $\mathcal{O}(n \log n)$ with a prohibitive, although very rare, $\mathcal{O}(n^2)$ worst case [10].[5]

Note that using our bottom-up encoding to compute the transitive closure of a taxonomy is correct only if it is a dag. If there are cycles in it, it will necessarily terminate with some of its elements left without code. This is because in Algorithm 1 a layer computed from a previous one [Line 8] removes any sort that has at least one child not encoded [Line 11]. This is correct for a dag since such sorts will always be reached later through a different longer path from $\perp$. But the existence of a cycle will make this assumption incorrect. For example, declaring both $s_1 \prec s_2$ and $s_2 \prec s_1$ will cause both s_1 and s_2 (and all their ancestors) to be removed from any layer to be encoded.

Therefore, the best we can expect with the bottom-up encoding method is that it always terminate in at most n iterations for a taxonomy of n elements. If the taxonomy is indeed a dag, all sorts will be correctly encoded. But if there are cycles, it will detect this to be the case (by checking that there remain non-encoded sorts upon termination). However, it does not have any possiblity to identify how many maximal cycles there are and which sorts compose them. It is because all it knows is that bottom-up encoding left some elements non-encoded—which happens if and only if there are cycles. However, it does not have specific information allowing identification of which exact (maximal) cycle(s) they are.

Solution

There is a simple and feasible way to proceed. It is enough to collect the non-encoded sorts after a bottom-up encoding in a new set to be classified using Warshall's method and topologically reordered. This is feasible then because such a set is relatively much smaller than the full declared taxonomy. In this way, all cycles can be identified as maximal contiguous elements and reported as errors.

---

[4] See Section 2.2.

[5] `en.wikipedia.org/wiki/Quicksort`

Let us now define such a topological ordering. Recall that a taxonomy of n sorts is represented as an array of size n of `Sort` objects that are characterized by three fields; the sort's:

1) `index`—its offset in the array;
2) `code`—its bit-vector encoding;
3) `name`—its name.

Using this precedence test on sorts, we can thus obtain a unique topological (total) ordering of a taxonomy whereby a sort s is said to *precede* another sort s' iff, in this order:[6]

1) $s.\texttt{code} < s'.\texttt{code}$; or,
2) $s.\texttt{code} = s'.\texttt{code}$ and $s.\texttt{index} < s'.\texttt{index}$; or,
3) s and s' are unrelated, and:
    - $|s.\texttt{code}| < |s'.\texttt{code}|$; or,
    - $|s.\texttt{code}| = |s'.\texttt{code}|$ and, $\texttt{firstDiff}(s.\texttt{code}, s'.\texttt{code})$ $<$ $\texttt{firstDiff}(s'.\texttt{code}, s.\texttt{code})$.

The expression $|c|$ for a code c denotes this code's cardinality (*i.e.*, its number of bits set to true). The expression $\texttt{firstDiff}(c, c')$ for two codes of equal cardinality c and c' denotes the lowest `1`-bit position in c that is `0` in c'. So the last condition ranks sorts according to the number of descendants, and when such are equal, according to the descendant of lowest differing index.

This ordering on sorts will keep lesser sorts and sorts of lesser cardinality at lower ranks. Same-cardinality codes (*i.e.*, sort with same number of subsorts) are ranked according to lowest index of the subsort contained in one but not the other. For example, code $c = \texttt{1000111}$ ($\{0, 1, 2, 6\}$) is toplogically less than code $c' = \texttt{0101011}$ ($\{0, 1, 3, 5\}$) because $\texttt{firstDiff}(c, c') = 2$ and $\texttt{firstDiff}(c', c) = 3$, and $2 < 3$.

It is not difficult to see that such a topological reordering will always end up with equally encoded elements being contiguous, while sorts with a greater number of lower bounds will be at higher ranks. In this manner, it is easy to identify all cycles in one single sweep of the taxonomy array as maximal sequences of contiguous equal codes of length at least 2.

On the Wikipedia taxonomy, the above method could identify 13 cycles: 12 of length 2 (*i.e.*, x *is-a* y and y *is-a* x) and 1 of length 3 (*i.e.*, x *is-a* y and y *is-a* z and z *is-a* x). There were also 49 warnings of cycles of length 1, which are harmless redundancies (*i.e.*, x *is-a* x). All these cycles were removed in the final Wikipedia taxonomy we used in our tests and measurements.

2.2.2 Decoding

We now turn to decoding—*i.e.*, relating bit-vector codes to the sorts they denote.

By construction of transitive closure, Algorithm 1, the bit vector of a sort at index i $(0 \leq i \leq n-1)$ has a `1` in position j $(0 \leq j \leq n-1)$ if and only if the sort at index j is its subsort. Therefore, a sort's bit vector has `1`s in all and only the positions of its descendant sorts. For example, the taxonomy shown as Figure 2 containing 12 sorts (other than $\top$ and $\bot$) will result in the encoding shown as Table 1. Since there are 12 sorts in this taxonomy, all codes in this taxonomy have 12 bits. The top and bottom elements ($\top$ and $\bot$) are implicit both in Figure 2 and in Table 1. So the code for bottom is all `0`s, and the code for $\top$ is all `1`s.

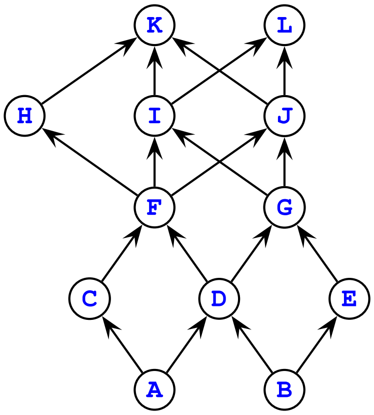


Fig. 2: Example of a small "is-a" taxonomy

Index	**Code**	**Sort**
11	`101110111111`	`L`
10	`011111111111`	`K`
9	`001010111111`	`I`
8	`000110111111`	`J`
7	`000010011011`	`G`
6	`000001101111`	`H`
5	`000000101111`	`F`
4	`000000011000`	`E`
3	`000000001000`	`B`
2	`000000000101`	`C`
1	`000000001011`	`D`
0	`000000000001`	`A`

TABLE 1: Transitive-closure codes for sorts in Figure 2

Let us first consider codes obtained without using negation. In other words, let us first restrict ourselves to decoding the result of only *positive* queries—*i.e.*, ones involving sorts of an encoded taxonomy using a Boolean expression of its sorts' bit-vector codes using bitwise `and`, `or`—but not `not`. This always results in a bit vector. In order to determine what sorts this resulting bit vector corresponds to, there are two cases: either the resulting bit vector is that of an existing sort, or it is not.

In the first case, in order to speed up determining the sort of the bit vector, all codes are stored in a hash table mapping a code to its sort. In this way, evaluating for example "`F & G`" in the taxonomy of Figure 2, which results in the bit-vector code `000000001011`, the sort can be retrieved in this hash table to be associated with the code—sort `D` in our example.

In the second case, the code resulting from a query evaluation does not correspond to an existing sort. For example

---

[6] The ordering on bit-vector codes is simply defined as $c_1 \leq c_2$ iff $c_1 = c_1 \,\&\, c_2$.

Index	Code	Sort
8	`100000000`	`poodle`
7	`010000000`	`canary`
6	`001000000`	`ostrich`
5	`100100000`	`dog`
4	`100110000`	`canid`
3	`110101000`	`pet`
2	`100110100`	`carnivore`
1	`111111111`	`animal`
0	`011000001`	`bird`

TABLE 2: Transitive-closure codes for sorts in Figure 3

evaluating "`I & J`" in the taxonomy of Figure 2 yields the code `000010111111`, which does not correspond to any specific sort in Table 1. However, semantically, this code is necessarily a minimal upper bound of the set denoted by the resulting sort if it existed. Hence, if we wish to express the resulting sort in terms of existing sorts, it is semantically the union of all the sorts whose codes are *maximal lower bounds* of the resulting code. In order to compute what sorts are in this set of maximal upper bounds, it suffices to retrieve all the sorts at index i such that there is a `1` at position i in the resulting code and keep only the maximal ones. In our example, the code `000010111111` has a `1` in positions $0, 1, 2, 3, 4, 5$ and 7. This means that its subsorts are `A`, `D`, `C`, `B`, `E`, `F`, and `G`. However, among these, only `F` and `G` are maximal. Therefore, the result of the query "`I & J`" is the *disjunctive* sort {`F ; G`}.

While the above decoding scheme is correct for positive queries, it is not so however if the query made use of negation. To see this, let us consider the taxonomy shown as Figure 3 and its encoding: shown as Table 2.

Let us now consider the (negative) query: "`!canid`." The code resulting from evaluating this query is the complement of the sort "`canid`"—namely, `011001111`. The decoding method that we used above will yield the maximal elements in index set $\{0, 1, 2, 3, 6, 7\}$—*viz.*, {`bird`, `animal`, `pet`, `ostrich`, `canary`, `poodle`}; namely, {`animal`}, which is obviously wrong. This decoding is incorrect because a negated code can no longer be interpreted as having a '`1`' in a position corresponding to a subsort. Indeed, for such a code, a '`1`' in position i means that sort of index i is either a supersort or unrelated. (This is because it comes from negating '`0`' at position i in the complement where that meant "sort of index i is *not a subsort of* this sort.")

This means that if a code to be decoded results from operations involving at least one negation, it must be ensured that all the `1`s in it are "genuine" indicators of lower bounds (which may not be the case if *such a '`1`' came from complementing a '`0`' by negation*).

In order to do that, let us consider the case of negating a sort expression `s`. If we compute the code of `!s` by switching all `1`s to `0`s and vice versa in the code of `s`, this will not work (as explained above). However, if we change a `1` in position *i* to a `0` in the code of `!s` whenever the sort of index *i* is a supersort of a sort corresponding to a `0` in the code of `!s`, then we will be left with a code having a `1` only in positions of actual subsorts of `!s`. With such a code, the decoding method for positive-query coded can then compute the correct set of maximal lower bounds of `!s`.

Given a code to be decoded, we call its set of "*undesirables*" the set of (indices of) ancestors of sorts corresponding to `0`s in this code. Clearly, by the very semantics of encoding, these indices cannot be `1`s in the code to be decoded. For if they were, so should be their descendants—but such is not the case since they are precisely defined as ancestors of sorts corresponding to a `0` in the code. So, if we switch off any `1` in the code to be decoded that corresponds to an "undesirable," we are left only with a code that can now be decoded with the method described above for codes invoving no negation—since we are now guaranteed that all the `1`s denote actual subsorts. Finally, note that for a code resulting from operations involving no negation, the set of "undesirables" is necessarily empty. Indeed, in this case, there can be no `0` in a position of a descendant of an actual ancestor.

In our example above, for the code `011001111` resulting from the evaluation of "`!canid`," the "undesirables" are the set of ancestors of sorts of indices $\{4, 5, 8\}$ (that is, the set of sorts {`canid`, `dog`, `poodle`}). This set is the set of indices $\{1, 2, 3, 4\}$ (or sorts {`animal`, `carnivore`, `pet`, `canid`}). Switching off undesirable bits gives the code: `011000001`. This set of indices ($\{0, 6, 7\}$) denotes the set of sorts {`bird`, `ostrich`, `canary`}. Keeping only maximal elements, this yield the (correct) answer: "`bird`."

For efficiency reasons, once a bit-vector code has been decoded, it is stored in a cache (a hash table) associating the code to the set of sorts whose codes are its maximal lower bounds. In this way, should the same code appear again as a result, it is first looked up in this cache to avoid the need to compute again its set of maximal lower bound sorts.

Finally, note that decoding a bit vector is only necessary for extracting the end result of a query in terms of defined sorts. All intermediate computation need not refer at all to the sorts and deal only with bit vectors, whether or not they correspond to defined sorts. There is no loss of information doing so as the encoding plunges the taxonomy in the minimal Boolean lattice containing it [1].

3 EXPERIMENTAL WORK

3.1 The State of the Art

In this section, we give a brief description of the SW reasoners that we have used for our comparative experiments. Note that we have limited our selection to systems that are full-fledged *reasoners*, and not just *classifiers*. This is because our interest goes beyond concept classification and includes Boolean query answering as well. This rules out systems such as ELK[7] [11], CEL[8] [12], CB[9] [13], *etc.*, that do not support query answering.

We retrieved and installed the following SW reasoners:

1) FaCT++;[10]
2) HermiT;[11]

---

[7] www.cs.ox.ac.uk/isg/tools/ELK/

[8] code.google.com/p/cel/

[9] code.google.com/p/cb-reasoner/

[10] owl.cs.manchester.ac.uk/fact++/

[11] www.hermit-reasoner.com/

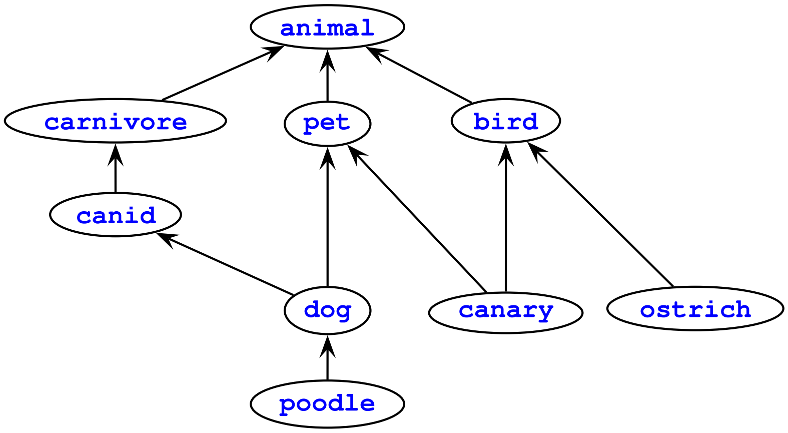


Fig. 3: Example of a small animal "is-a" taxonomy

3) Pellet;[12]
4) TrOWL;[13]
5) Racerpro;[14]
6) SnoRocket.[15]

FaCT++ (Fast Classification of Terminologies) is a reasoner developed at the University of Manchester [14]. It is based on the Description Logic fragment $\mathcal{SHOIQ}$ [15]. It is implemented in C++ as a deductive tableau [16] adapted to the specifics of this logic. It is claimed to use a wide range of heuristic optimizations. FaCT++ provides TBox reasoning (subsumption, satisfiability, classification) and partial support for ABox processing (retrieval).

HermiT is also a reasoner for a (slight extension) of the Description Logic fragment $\mathcal{SHOIQ}$ (called $\mathcal{SHOIQ}^+$) [17]. It is based upon hypertableau reasoning, an optimized version of tableau reasoning [18]. It purports to provide a faster process for classifying ontologies. The main optimization of hypertableau *vs.* tableau that it tries to minimize nondeterminism in the treatment of disjunctions and is more memory-efficient. HermiT provides TBox reasoning, with the ability of checking the consistency of an ontology and inferring implicit relationships between concepts.

Pellet is a free open-source Java-based reasoner [19]. It, too, is based on the tableau algorithm and supports the Description Logic fragment $\mathcal{SHOIN}(D)$. It provides TBox reasoning (subsumption, satisfiability, and classification) and ABox reasoning (retrieval, conjunctive query answering). It uses many optimization techniques and supports entailment checks and ABox querying through its interface.

TrOWL (Tractable reasoning infrastructure for OWL 2) was developed at the University of Aberdeen [20]. This is a system that starts by transforming an ontology from OWL-DL to OWL-QL [21] in order to classify it in polynomial time. Under this transformation, conjunctive query answering and consistency checking remain the same as for OWL-DL. In addition, TrOWL can generate a database schema for storing normalized representations of OWL-QL ontologies.

Racerpro is a commercial version of RACER (Renamed ABoxes and Concept Expression Reasoner) [22], [23]. It implements a reasoner for the description logic $\mathcal{SHIQ}$. RACER provides both TBox and ABox reasoning. It supports all the optimizations of FaCT++ as well as new techniques for dealing with number restrictions and ABoxes.

Snorocket [24] was proposed as a high-performance implementation of a polynomial-time classification algorithm for the lightweight Description Logic $\mathcal{EL}$ [25].[16] It was primarily meant to be optimized for classifying SNOMED CT. It can process only conjunctive queries.

3.2 Experiment setup

We extracted the bare concept taxonomies gathered from four very large ontologies. Here they are, listed in order of increasing sizes:

1) Wikipedia—this is an ontology derived from the Wikipedia online database (size: 111,599 sorts);[17]
2) BioModels—this is an ontology of various biological models (size: 182,651 sorts);[18]
3) MeSH—(Medical Subject Headings) this is an ontology of the National Library of Medicine (size: 286,381 sorts);[19]
4) NCBI—this is the National Center for Biotechnology Information's ontology of all known living organisms (size: 903,617 sorts).[20] This taxonomy (the largest) is a "bare" taxonomy—*i.e.*, it contains partially "is-a" ordered classes symbols (concepts), bu no properties (roles).

---

[12] `clarkparsia.com/pellet/`
[13] `trowl.eu/`
[14] `www.racer-systems.com/products/racerpro/`
[15] `research.ict.csiro.au/software/snorocket`
[16] Description Logics in the $\mathcal{EL}$-family are weaker versions that provide existential roles ($\exists r.C$) but no universal roles ($\forall r.C$).
[17] `www.h-its.org/english/research/.../wikitaxonomy.php`
[18] `bioportal.bioontology.org/ontologies/3022`
[19] `www.nlm.nih.gov/mesh/meshhome.html`
[20] `www.ncbi.nlm.nih.gov/Taxonomy/taxonomyhome.html/`

We focused only on bare conceptual taxonomic reasoning.[21] That is, we considered no roles, just sorts—a sort being defined as a monadic concept in a partially ordered "is-a" taxonomy. The "reasoning" on such sorts amounts to propositional logic. In other words, this boils down to computing Boolean expressions consisting of sorts and the three operations: `and`, `or`, `not`. Seen another (equivalent) way, these operations applied to set-denoting expressions are interpreted respectively as set intersection, union, and complementation. The topmost sort $\top$ denotes the set of all things, and the bottommost sort $\bot$ denotes the empty set—*i.e.*, the set of no thing.

Section 3.3 reports the results of our comparative experiments of our Java implementation of our method with the state-of-the-art reasoners on bare conceptual taxonomies using only propositional-logic queries.

3.3 "Just the Facts, Ma'am!"

This section reports the results of our experiments with the six reasoners that we retrieved (FaCT++, HermiT, Pellet, Racerpro, TrOWL, SnoRocket) and ours ($\mathcal{CEDAR}$). All runs for all reasoners were carried out for exactly the same queries under the exact same conditions (on an Intel Core Duo CPU, 2.20 Ghz, 64-bit processor, running Windows 64, with 250 GB SATA HDD, and 16 GB main memory).

3.3.1 Classification

Figure 4 shows the comparative classification time performances for each reasoner on each of the large taxonomies we have selected. This makes six out of the seven reasoners. We did not consider SnoRocket in these classification-performance graphs because we realized that that system does not actually perform any preliminary classification, but does so on demand at query time.

While our system's classification time ($\mathcal{CEDAR}$—the leftmost performance on all graphs) is not always the best, it is always among the three best out of six, the worst being systematically TrOWL. This latter point may be due to the fact that it involves a preliminary compilation from DL to QL. Be that as it may, one can accept a longer classification time if it means faster query answering. In this regard, as illustrated next, our system definitely keeps a huge margin over all the others. It is to be noted also that TrOWL is faster at query answering relatively to the others (although still quite worse than our system on all tested taxonomies). This, again, may be justified by the longer classification time. On the other hand, HermiT, that is among the better classifiers, is always the worst for query answering. This is shown next.

3.3.2 Querying

For each of the seven reasoners, Figure 5 shows the comparative query response time performances for two kinds of queries. If some reasoners are missing on some of these graphs, it is because they could not provide an answer before a time-out period that we set to 30 minutes.

In order not to "compare apples and bananas," we only tested the various taxonomic reasoners by submitting positive queries—*i.e.*, queries containing only conjunctions and disjunctions, but no negation. This is because the OWL-API reasoners follow a so-called "Open-World Assumption" (OWA),[22] while $\mathcal{CEDAR}$ obeys a "Closed-World Assumption" (CWA).[23] Indeed, when no negation is used in a query, in both CWA and OWA reasoners, it will evaluate to the same result—*viz.*, same set of minimal upper bounds and same set of maximal lower bounds.

The first series of graphs (leftmost column) are for mixed conjunctive and disjunctive queries, of the form: $s_1 \ \& \ \ldots \ \& \ s_{n/2} \ \& \ (s_{n/2+1} \ | \ \ldots \ | \ s_n)$, for $n = 10, 20, \ldots, 100$.[24] We made one exception for SnoRocket, for which the queries were all conjunctive since the latest system available does not support disjunctive queries.

The second series of graphs (rightmost column) are for purely disjunctive queries of the form: $s_1 \ | \ \ldots \ | \ s_n$, for $n = 10, 20, \ldots, 100$. We did not include SnoRocket in this series of tests since, again, the system we retrieved does not support disjunctive queries.

In both series of graphs, it is clear that our system ($\mathcal{CEDAR}$) systematically achieves the best performance. Moreover, it does so by several orders of magnitude (recall that the scale of time is logarithmic).

3.4 Discussion

Although the graphs reported in Figure 5 speak for themselves, it is interesting to get an appreciation of the relative performances for query answering of all the reasoners we have tested. In order to do so, Tables 3 and 4 sum up the facts displayed in the graphs by taking the average over all query sizes (*viz.*, from 10 to 100 concepts), giving the maximum of these averages the value 100, and showing all the other averages as percent values.

Empty cells mean that the reasoner was never able to provide an answer within our time-out limit (which, again, was set to 30 minutes). On the "BioModels" taxonomy, Pellet stumbles into a Java runtime error for some unknown reason.

Table 3 shows these figures for the mixed conjunctive and disjunctive queries. Table 4 shows these figures for purely disjunctive queries. Again, SnoRocket does not appear in the latter because it could not be tested on disjunctive queries (hence the N/A entries).

The reason why we limited our study to bare propositional reasoning is that this is (or ought to be) the most basic capability of any ontological reasoner. Any further capability in a more complete ontology based on a taxonomy (*e.g.*, reasoning with roles—existential and/or universal, cardinality constraints, *etc.*.) must be conjugated with the basic propositional reasoning on taxonomic sorts. In fact, in such systems, a "complex" concept is typically a sort conjoined with some additional role-related expression. Thus, one can see bare

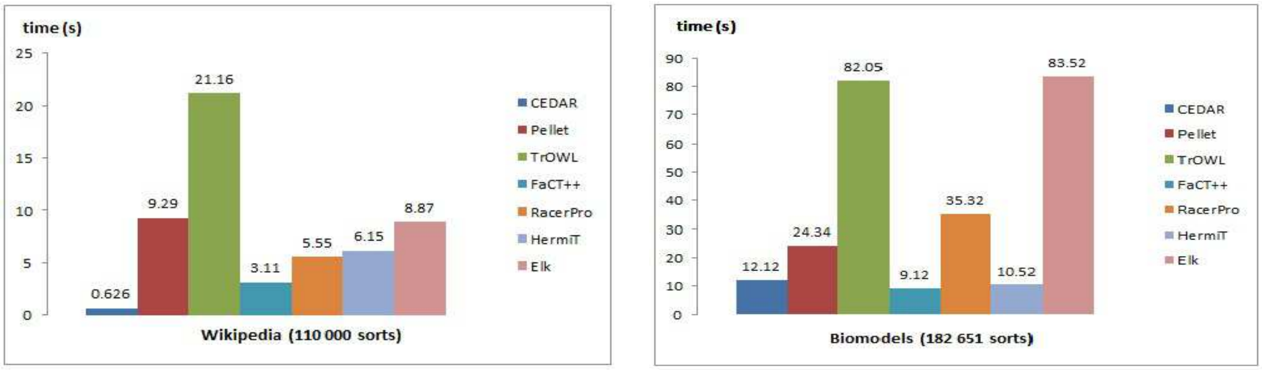
---

[21] See Section 3.4 for a discussion concerning this point.

[22] `http://c2.com/cgi/wiki?OpenWorldAssumption`

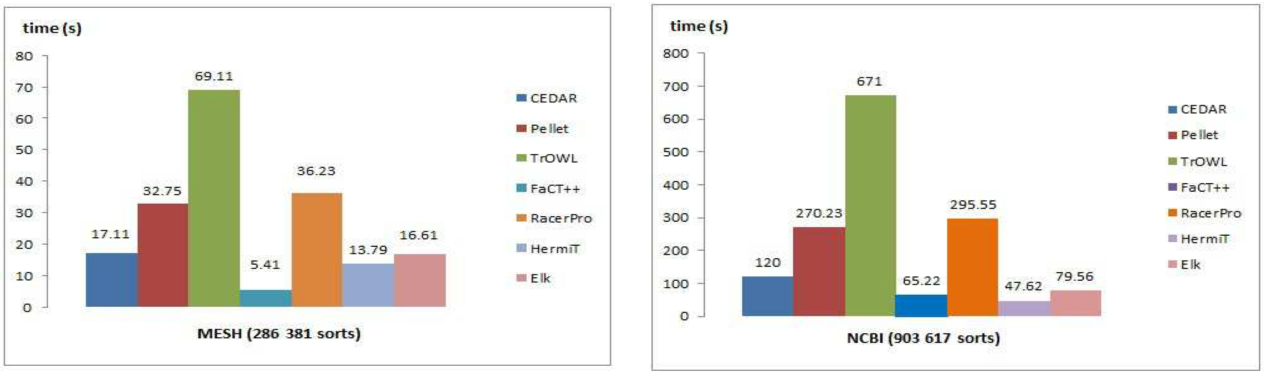
[23] `http://c2.com/cgi/wiki?ClosedWorldAssumption`

[24] We use "&" to denote "`and`," and "|" to denote "`or`."



(a) Wikipedia

(b) BioModels



(c) MeSH

(d) NCBI

Fig. 4: Classification time per reasoner per taxonomy

Boolean taxonomic sort reasoning as sheer abstract interpretation of complex-concept reasoning [26].

Now, in terms of implementing such reasoners, it comes as evidence that efficiency must start with the simpler form of reasoning since it is part of all further reasoning. In fact, taking advantage of commutativity and associativity of conjunction, it may even be a technique of optimizing the process of ontological reasoning. Indeed, taking a complex expression such as involving a Boolean combination of expressions such of the form s & s-*properties*, ignoring the s-*properties* parts as a first pass will narrow the original expression to its essential remaining maximal sorts.

In order to carry out these experiments, we also developed a tool with an easy-to-use GUI that lets a user run these tests for any listed reasoner and taxonomy. This tool is available for download from the $\mathcal{CEDAR}$ Project's website for anyone to verify our results.[25] Also, a video clip of these demos showing these experiments *in vivo*, and a web service for running these demos on line are available online.[26] In this way, our results may hopefully not have to be taken on faith, but could be verified *de visu* by anyone who might wish to check them on their own.

Finally, it is worth pointing out that once a taxonomy has been classified, it may be saved on disk to be reloaded without any penalty and reused over and over.[27] This is akin to compiling a program and not needing to recompile it for each use. In this regard, we do not understand the relative importance given in SW literature to performance of ontology classification as opposed to that of query answering. In programming-language technology, what is of prime importance is runtime performance, not compile-time performance.

4 CONCLUSION

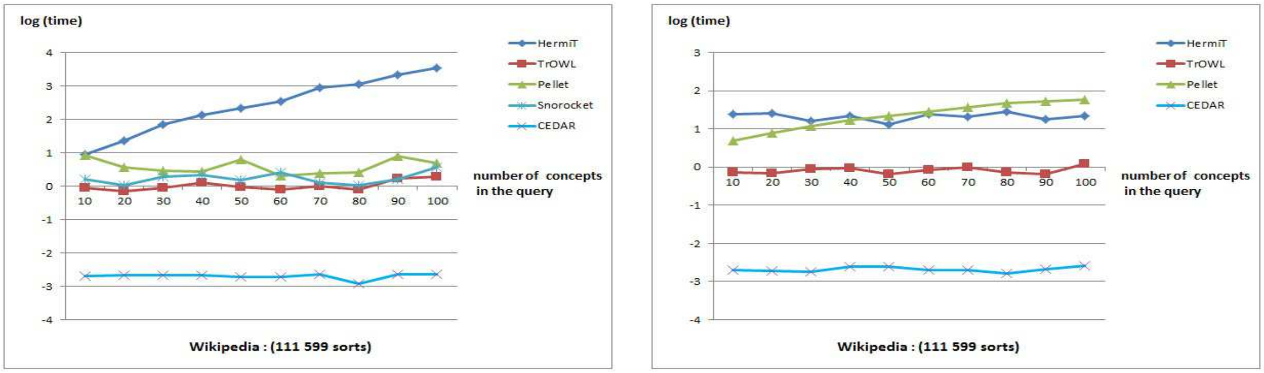
In this paper, we have presented an implementation of a Boolean Logic taxonomic reasoner based on bit-vector encoding. We have implemented such a reasoner in Java, and have compared its performances to those of six among the most renown Semantic Web reasoners on bare taxonomies. This demonstrated that the best exisiting Semantic Web reasoners fail to live up to performances that can be easily achieved using bit-vector encoding. This is true even, and especially, when applied to very large taxonomies. Focusing only on pure Boolean taxonomic reasoning—which is at the core of *every* SW reasoner—the results of our measurements show that our system achieves performances for Boolean query answering that are several orders of magnitude better than those of the state of the art.

---

[25] `cedar.liris.cnrs.fr/data/CEDAR-V1.0.zip`

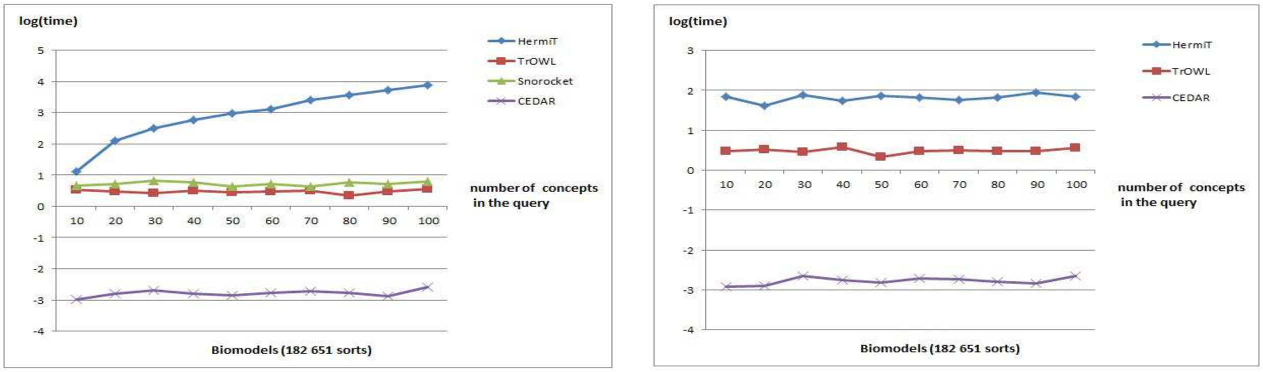
[26] `cedar.liris.cnrs.fr/demos.html`

[27] We implemented such a facility, of course—see Appendix.



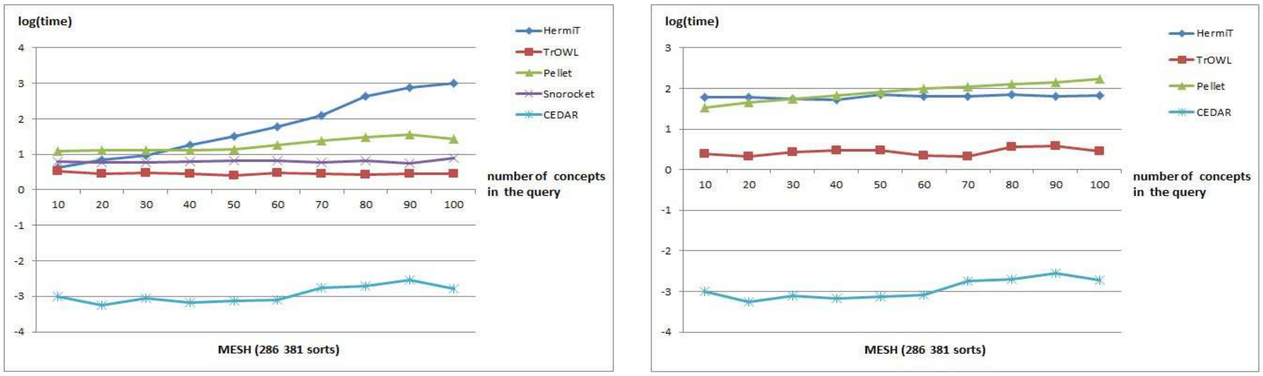
(a) Wikipedia/Mixed Queries

(b) Wikipedia/Disjunctive Queries



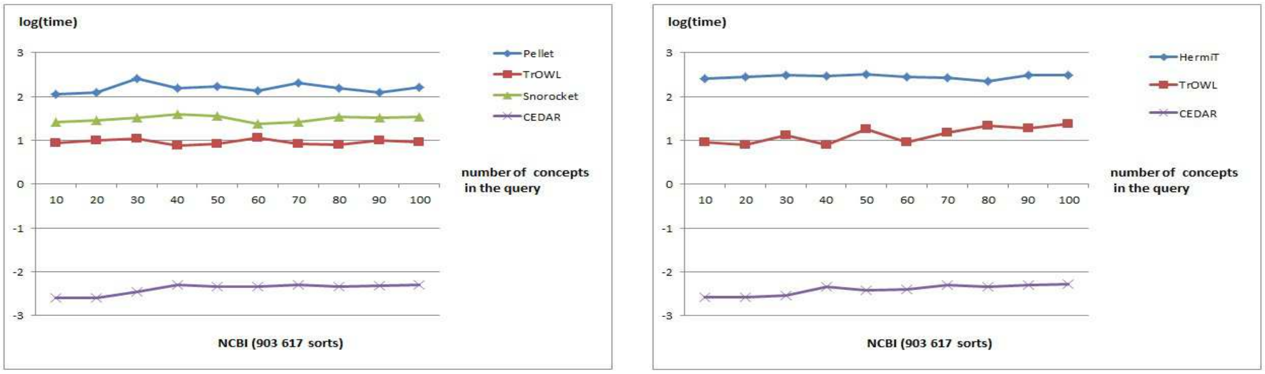
(c) BioModels/Mixed Queries

(d) BioModels/Disjunctive Queries



(e) MeSH/Mixed Queries

(f) MeSH/Disjunctive Queries



(g) NCBI/Mixed Queries

(h) NCBI/Disjunctive Queries

Fig. 5: Comparative performance graphs for query response times per taxonomy

Taxonomy	FaCT++	HermiT	TrOWL	Pellet	Racerpro	SnoRocket	$\mathcal{CEDAR}$
Wikipedia		100	0.13	0.51		0.21	0.000233
BioModels		100	0.13	Error		0.24	0.000074
MeSH		100	1.17	8.29		2.60	0.000530
NCBI			5.78	100		19.75	0.002627

TABLE 3: Relative normalized average percentiles of performance times for mixed queries

Taxonomy	FaCT++	HermiT	TrOWL	Pellet	Racerpro	SnoRocket	$\mathcal{CEDAR}$
Wikipedia		74.65	2.86	100		N/A	0.00719
BioModels		100	4.68	Error		N/A	0.00258
MeSH		67.50	3.01	100		N/A	0.00141
NCBI		100	5.10			N/A	0.00141

TABLE 4: Relative normalized average percentiles of performance times for disjunctive queries

There are several optimizations that may still be performed to our basic method. The technique known as *code modulation* (explained in detail in [1]) can take advantage of a taxonomy's specific shape to minimize drastically code space. This makes so-modulated bit-vector encoding very scalable as explained in [1]. Indeed, modulated encoding reduced code size to a logarithmic function of non-modulated encoding. Code modulation being independent of the encoding technique, it can be applied to any method. Each module can in fact use different encoding methods each adapted to its specific topology. It can thus be used recursively (*i.e.*, one can modulate a module) with the encoding technique most appropriate to the topology of the module being encoded.

Another optimization to minimize classification time could be to perform lazy encoding.[28] In other words, one could only encode the sorts relevant to a query and cache intermediate results. The price to pay would be at query time, although only the first time a subset of the concepts it involves are used.

As for further work, we are extending this work to unification-based Knowledge Representation known as Order-Sorted Feature (OSF) constraint logic [27]. While OSF logic uses functional features, we can use them to represent roles using aggregates. The advantage is that role-based reasoning is thus made simpler since it relies on Logic-Programming unification technology made possible by functional attributes [28], [29]. This is akin to compiling DL-based relational roles into aggregate-valued functional features. OSF sorts have also a "memoizing" effect whereby no property needs to be proven again once it has been established for any supersort [27].

APPENDIX

In Section A, we give an overview of an implementation specification for representing very large bit vectors, reducing memory consumption while retaining efficient operations. In our experiments, this alternative code representation was used only for saving an encoded taxonomy on disk and reloading it as a pre-encoded order. But for taxonomies of even larger size than those used in our experiments, it could be used for lattice operations as well.

APPENDIX A
COMPACT CODES

While the foregoing sections present a method for encoding elements of a partially ordered set based on transitive closure, the data structure it relies on is that of a binary word—*i.e.*, a bit vector. With such a structure, all boolean operations—*and*, *or*, *not*—are thus very efficient. This representation also eases computation of the transitive closure since setting a bit on or off is trivially accommodated.

However, while this representation is convenient and time-efficient for relatively small posets of the order of a few hundred elements, it quickly becomes space-inefficient for large posets of hundreds of thousands, or millions of elements.

In what follows, we define an alternative representation of indexed bit sets that offers the advantage of being more compact than bit vectors while retaining time-efficient boolean and bit-setting operations. It is also the format we use to save/load encoded taxonomies on/from disk. In Section A.1, the basic data structure is defined. In Section A.2, bit setting and unsetting operations are defined. In Section A.3, the three boolean operations—conjunction, disjunction, and negation—are defined. In Section A.4, some implementation considerations are discussed.

A.1 Bit Code Representation

The idea is intuitively simple. It consists of representing a bit vector as a finite array of k ($k \in \mathbb{N}$) pairs of integer indices $\langle l_i, u_i \rangle$, for $i = 0, \ldots, k-1$, such that, for all indices $i = 0, \ldots, k-2$:

$$0 \leq l_i < u_i < l_{i+1} < u_{k-1}. \tag{1}$$

We shall refer to such a sequence k pairs, $k \in \mathbb{N}$, $\{\ \langle l_i, u_i \rangle \mid i = 0, \ldots, k-1\ \}$ as a *compact code*. For $k = 0$, this is written as the empty sequence $\{\}$.

Given a compact code representation of a bit vector $\mathbb{V}$, each pair $\langle l, u \rangle$ represents a *maximal* contiguous sequence of 1's (hereafter referred to as a "*packet*") in $\mathbb{V}$. Thus, the i-th packet of a bit vector is represented as the pair of indices $\langle l_i, u_i \rangle$ such that l_i is the index of the lowest bit in the packet, and u_i is the index of the first `0`-bit following the packet.

---

[28] In the same manner as we have noticed that SnoRocket does.

For example, the compact code of the bit vector `0011111001111000000110000` corresponds to the sequence of packet pairs:[29] $\{\ \langle 4,6\rangle, \langle 12,16\rangle, \langle 18,23\rangle\ \}$.

The empty bit vector (containing all 0's) is represented as the empty sequence $\{\}$. The *length* of a bit vector represented by a compact code sequence of k pairs (or packets) is u_k. The *size* of a compact code C of k pairs (or packets) is k (*i.e.*, its number of packets).

A.2 Bit Operations

Let $C = \{\ \langle l_i, u_i\rangle \mid i = 0, \ldots, k\ \}$ be a compact code of k packets ($k > 0$). Given a number $n \in \mathbb{N}$, and a compact code C of k packets as defined above, we say that:[30]

- n *is within a packet of* C iff $\exists i \in [0, k-1]$ such that $l_i \leq n < u_i$—in which case we shall write $C.\mathbf{packet}(n) = i$;
- n *is between packets of* C iff either one of the three statements holds:
  1) $n < l_0$; or,
  2) $u_{k-1} \leq n$; or,
  3) $\exists i \in [0, k-2]$ such that $u_i \leq n < l_{i+1}$.

If a number n is between packets of a compact code C of size k, we define two functions $C.\mathbf{prev}(n)$ and $C.\mathbf{next}(n)$ for each of the three possible respective cases above as follows (where the symbol '?' means "undefined"):

1) $C.\mathbf{prev}(n) \overset{\mathit{def}}{=\!=} \ ?$ and $C.\mathbf{next}(n) \overset{\mathit{def}}{=\!=} l_0$;
2) $C.\mathbf{prev}(n) \overset{\mathit{def}}{=\!=} u_k$ and $C.\mathbf{next}(n) \overset{\mathit{def}}{=\!=} \ ?$;
3) $C.\mathbf{prev}(n) \overset{\mathit{def}}{=\!=} u_i$ and $C.\mathbf{next}(n) \overset{\mathit{def}}{=\!=} l_{i+1}$.

For such a number n, we say that:

- n is left-adjacent in C if $n = C.\mathbf{next}(n) - 1$;
- n is right-adjacent in C if $n = C.\mathbf{prev}(n)$;
- n is adjacent in C if it is both left-adjacent and right-adjacent in C.

Note that if n is between packets and adjacent, this necessarily means that the two packets on each side are only separated by a single `0`-bit (the bit in position n in the denoted bit vector).

N.B.: In all the compact code expressions to follow, we use the implicit convention that a packet with undefinable bounds is simply omitted. Thus, we will always use the notation $\{\ \langle l_0, u_0\rangle, \ldots, \langle l_{k-1}, u_{k-1}\rangle\ \}$ to denote a compact code, where $k \geq 0$ up to the above conventions regardless of the actual number of packets. For example, for $k = 0$ this will correspond to the empty code $\{\}$, and for $k = 1$, this will correspond to the single-packet code $\{\ \langle l_0, u_0\rangle\ \}$.

We define the following bit-setting operations on C. These methods operate "in place" by modifying a code C that invokes them.

- $C.\mathbf{set}(n)$, for $n \in \mathbb{N}$, which sets the n-th bit of the bit vector denoted by C to 1.
- $C.\mathbf{set}(n, m)$, for $n, m \in \mathbb{N}, n < m$, which sets to 1 all the bits from position n (*inclusive*) to position m (*exclusive*) of the bit vector denoted by C.
- $C.\mathbf{unset}(n)$, for $n \in \mathbb{N}$, which sets the n-th bit of the bit vector denoted by C to 0.
- $C.\mathbf{unset}(n, m)$, for $n, m \in \mathbb{N}, n < m$, which sets to 0 all the bits from position n (*inclusive*) to position m (*exclusive*) of the bit vector denoted by C.

For $m \leq n$, both $C.\mathbf{set}(n, m)$ and $C.\mathbf{unset}(n, m)$ are no_ops—*i.e.*, they leave C unchanged. Since $\mathbf{set}(n)$ is equivalent to $\mathbf{set}(n, n+1)$, we will just give the methods for $\mathbf{set}(n, m)$ and similarly for $\mathbf{unset}(n)$.

A.2.1 Bit setting

There are four cases to consider for which performing $C.\mathbf{set}(n, m)$ modifies C as follows.

If n is within a packet in C (say, $C.\mathbf{packet}(n) = i$) and m is within a packet in C (say, $C.\mathbf{packet}(m) = j$), then, if $i = j$, $C.\mathbf{set}(n, m)$ leaves C unchanged. Else (if $i < j$),[31] then C becomes:

$$\{\ \ldots, \langle l_i, u_j\rangle, \ldots\ \}.$$

If n is within a packet in C (say, $C.\mathbf{packet}(n) = i$) and m is between packets in C, then C becomes:

$\{\ \ldots, \langle l_i, u_j\rangle, \ldots\ \}$
if m is left-adjacent in C and $C.\mathbf{next}(m) = l_j$;

$\{\ \ldots, \langle l_i, m\rangle, \ldots\ \}$
otherwise.

If n is between packets in C and m is within a packet in C (say, $C.\mathbf{packet}(m) = j$), then C becomes:

$\{\ \ldots, \langle l_i, u_j\rangle, \ldots\ \}$
if n is right-adjacent in C and $C.\mathbf{prev}(n) = u_i$;

$\{\ \ldots, \langle n, u_j\rangle, \ldots\ \}$
otherwise.

If both n and m are between packets in C, then C becomes:

$\{\ \ldots, \langle l_i, u_j\rangle, \ldots\ \}$
if n is right-adjacent in C and $C.\mathbf{prev}(n) = u_i$, and
if m is left-adjacent in C and $C.\mathbf{next}(m) = l_j$;

$\{\ \ldots, \langle l_i, m\rangle, \ldots\ \}$
if n is right-adjacent in C and $C.\mathbf{prev}(n) = u_i$, and
if m is not left-adjacent in C;

$\{\ \ldots, \langle n, u_j\rangle, \ldots\ \}$
if n is not right-adjacent in C, and
if m is left-adjacent in C and $C.\mathbf{next}(m) = l_j$;

$\{\ \ldots, \langle n, m\rangle, \ldots\ \}$
otherwise.

A.2.2 Bit unsetting

Here again, there are four cases to consider for which performing $C.\mathbf{unset}(n, m)$ modifies C as follows.

---

[29] Recall that a bit vector is written with its lowest bit to the right.

[30] In what follows, we shall use the "dot" notation of object-oriented methods to denote all functions or operations on codes.

[31] Note that $i \neq j$ implies necessarily that $i < j$ (by Condition (1) and since $n < m$).

If both n and m are between packets in C: if $C.\mathbf{prev}(n) = C.\mathbf{prev}(m)$ (or, equivalently, if $C.\mathbf{next}(n) = C.\mathbf{next}(m)$), then $C.\mathbf{unset}(n, m)$ leaves C unchanged; else, C becomes:

$$\{\ \ldots, \langle l_i, C.\mathbf{prev}(n)\rangle, \langle C.\mathbf{next}(m), u_j\rangle, \ldots\ \}.$$

If n is within a packet in C (say, $C.\mathbf{packet}(n) = i$) and m is between packets in C, then C becomes:

$\{\ \ldots, \langle l_{i-1}, u_{i-1}\rangle, \langle C.\mathbf{next}(m), u_j\rangle, \ldots\ \}$
if $n = l_i$, where $C.\mathbf{next}(m) = l_j$;

$\{\ \ldots, \langle l_i, n\rangle, \langle C.\mathbf{next}(m), u_j\rangle, \ldots\ \}$
else (*i.e.*, if $n > l_i$), where $C.\mathbf{next}(m) = l_j$.

If n is between packets in C and m is within a packet in C (say, $C.\mathbf{packet}(m) = j$), then C becomes:

$\{\ \ldots, \langle l_i, C.\mathbf{prev}(n)\rangle, \langle l_{j+1}, u_{j+1}\rangle, \ldots\ \}$
if $m = u_j - 1$, where $C.\mathbf{prev}(m) = u_i$;

$\{\ \ldots, \langle l_i, C.\mathbf{prev}(n)\rangle, \langle m, u_j\rangle, \ldots\ \}$
else (*i.e.*, if $m < u_j - 1$), where $C.\mathbf{prev}(m) = u_i$.

If both n is within a packet (say, $C.\mathbf{packet}(n) = i$), and m is within a packet (say, $C.\mathbf{packet}(m) = i$) in C, then C becomes:

$\{\ \ldots, \langle l_{i-1}, u_{i-1}\rangle, \langle l_{j+1}, u_{j-1}\rangle, \ldots\ \}$
if $n = l_i$, and
if $m = u_{j-1}$;

$\{\ \ldots, \langle l_{i-1}, u_{i-1}\rangle, \langle m, u_j\rangle, \ldots\ \}$
if $n = l_i$, and
if $m < u_{j-1}$;

$\{\ \ldots, \langle l_i, n\rangle, \langle l_{j+1}, u_{j-1}\rangle, \ldots\ \}$
if $n > l_i$, and
if $m = u_{j-1}$;

$\{\ \ldots, \langle l_i, n\rangle, \langle m, u_j\rangle, \ldots\ \}$
if $n > l_i$, and
if $m < u_{j-1}$.

A.3 Boolean Operations

Let:

$$C = \{\ \langle l_i, u_i\rangle \mid i = 0, \ldots, k-1\ \}$$

and:

$$C' = \{\ \langle l'_i, u'_i\rangle \mid i = 0, \ldots, k'-1\ \}$$

be two compact code pair sequences, with $k \geq 0$ and $k' \geq 0$.

A.3.1 Conjunction

Invoking $C.\mathbf{and}(C')$ will modify C according to C' by unsetting all the bits in C that are between packets in C', leaving C' unchanged.

If $C = \{\}$, then C is left unchanged; else, if $C' = \{\}$, then C becomes $\{\}$.

Else (*i.e.*, if $k > 0$ and $k' > 0$), C is modified by invoking:

- $C.\mathbf{unset}(0, l'_0)$; and,
- $C.\mathbf{unset}(u'_i, l'_{i+1})$, for $i = 0$ up to $i = k' - 1$; and,
- $C.\mathbf{unset}(u'_{k'-1}, u_{k-1})$.

Note that in practice, when proceeding in the above order, as soon the first argument of any $\mathbf{unset}(\ldots)$ is greater than or equal to u_{k-1}, there is no need to perform the unsetting nor proceed any further.

A.3.2 Disjunction

Invoking $C.\mathbf{or}(C')$ will modify C according to C' by setting all the bits in C that are within packets in C', leaving C' unchanged.

If $C' = \{\}$, then C is left unchanged; else, if $C = \{\}$, then C becomes (a copy of) C'.

Else (*i.e.*, if $k > 0$ and $k' > 0$), C is modified by invoking:

- $C.\mathbf{set}(l'_i, u'_i)$, for $i = 0$ up to $i = k' - 1$.

A.3.3 Negation

Since a bit vector is open-ended, we may define its negation only up to a length at least greater than its highest `1`-bit position. This operation is denoted as $C.\mathbf{not}(n)$. Thus, $\{\}.\mathbf{not}(n)$, is undefined for any $n \geq 0$.

Otherwise, for a non-empty code $C = \{\ \langle l_0, u_0\rangle, \ldots, \langle l_{k-1}, u_{k-1}\rangle\ \}$ and $n \geq u_{k-1}$, $C.\mathbf{not}(n)$ modifies C to become:

$$\{\ \langle 0, l_0\rangle, \ldots, \langle u_i, l_{i+1}\rangle, \ldots, \langle u_{k-1}, n\rangle\ \}.$$

Again, following our convention, if $l_0 = 0$, $\langle 0, l_0\rangle$ being undefinable, the first element of $C.\mathbf{not}(n)$ is $\langle u_0, l_1\rangle$. Similarly, if $n = u_{k-1}$, then $\langle u_{k-1}, n\rangle$ is undefinable and the last element of $C.\mathbf{not}(n)$ is $\langle u_{k-2}, l_{k-1}\rangle$.

A.4 Implementation Considerations

We need to come up with a data structure for representing a compact code that would enable retaining maximal efficiency in the bit setting and unsetting operations, and hence in the boolean operations that rely on them.

Most frequently used operations on such a data structure C for an integer n are:

- $C.\mathbf{packet}(n)$—for n inside a packet in C, returning that packet number;
- $C.\mathbf{prev}(n)$—for n between packets in C, returning the upper index of the packet preceding n;
- $C.\mathbf{next}(n)$—for n between packets in C, returning the lower index of the packet following n;
- adding/removing a packet.

Because the elements of a code sequence are ranges rather than integers, one cannot expect hashed $\mathcal{O}(1)$ time access to find out whether a given integer lies within or between packets. So structures such as defined by the Java classes `java.util.HashSet` or `java.util.LinkedHashSet` cannot be used.

In order to make these operations at most $\mathcal{O}(\log(k))$ time for a compact code of size k, one way is to represent a compact code as a balanced binary tree of pairs of bit position spans $\langle l_i, u_i\rangle$, taking advantage of the ordering imposed by condition (1).

Thus, the `java.util.TreeSet` class looks like a convenient choice, since it offers the required data structure properties in addition to defining methods such as `first`, `last`, `higher`, `lower`, `add`, `remove`, *etc.*, as well as an order-respecting iterator.

On the other hand, the `java.util.TreeSet` is missing a *replace* method—which is critically needed for setting and unsetting bits. It is also missing an *insert* method that splices a new sequence of packet pairs into an existing compact code, which may also be often used. One must resort to several `add`/`remove` method invocations to replace or insert elements, which incur new searches (and possible intermediate rebalancing of the tree) each time. This is a waste since replacing and inserting can be done in $\mathcal{O}(1)$ time when having already found the required elements, and only one (final) $\mathcal{O}(\log(k))$ tree rebalancing.

Hence, rather than relying on the ready-to-use `java.util.TreeSet` class, it may be more beneficial to implement a new specific class for a compact code as a doubly linked list and a balanced binary tree adapted for the specific nature of its pair elements. This would make transparent the double links of each pair element and ease replacement and insertion.[32]

A.5 Discussion

A similar data structure was proposed by researchers in data and knowledge bases in [30]. However, the authors did not use that representation for lattice operations as we do here. Instead, they focused on using it for obtaining more compact range-sequence codes for the transitive closure of the "is-a" relation of a taxonomy. That representation is equivalent to the one we specify here and to the one in [1]. Contrary to [1], they define an element's code as the union of index ranges from the post-order arrangement of the a spanning tree of the "is-parent-of" relation of a taxonomy. Each concept in the taxonomy (*i.e.*, each element in the poset) of post-order index j is then encoded as the interval $\langle i, j\rangle$ where i is the smallest post-order index of all its descendants. Although they did not do it, it is easy to show that their representation is equivalent to bit vectors. But they did not specify lattice operations on their data structures as we do for ours in this document. What they focused on was minimizing the total number of packets in range codes. In order to do so, they suggest generating codes based on the "optimal" spanning tree for generating the most compact set of codes. The data structure and algorithm for what they call "compressed transitive closure" do not maintain dynamic interval consistency caused by potential adjacency as we do here.[33] Although they do cite [1], Agrawal *et al.* do so only in the conclusion as they had just noticed its publication. They suggest that their approach and that exposed in [1] might be combined for processing large taxonomies. As far as we know, no follow-up on this suggestion was carried out.

It is clear how the work of [30], although orthogonal to ours, could be adapted to our needs as well in order to improve its space consumption. However, it is to be noted that their code-compaction method requires a topologically ordered poset. For very large taxonomies (over 1 million elements) the price of sorting the taxonomy might be worth spending only for once-for-all prepocessing prior to query time [31]. Also, the question of incrementality is not addressed.

Finally, although this work has been motivated for obtaining a compact representation of binary codes encoding a partial order, it comes as evident that the data structure, and operations on it, specified in this appendix can represent any set of integers (or integer-indexed set) seen as a sequence of intervals. Set intersection is realized as the conjunction described in Section A.3.1; set union as the disjunction described in Section A.3.2; and, set complementation as the negation described in Section A.3.3. Therefore, it can readily be used for this purpose as well.

ACKNOWLEDGEMENTS

This work was carried out as part of the $\mathcal{CEDAR}$ Project (Constraint Event-Driven Automated Reasoning) under the *Agence Nationale de la Recherche* (ANR) Chair of Excellence grant N^o **ANR-12-CHEX-0003-01** at the *Université Claude Bernard Lyon 1* (UCBL). The authors wish to thank Prof. Mohand-Saïd Hacid and Dr. Rafiqul Haque for their constructive feedback. The authors are solely responsible for the contents of this article, including any remaining mistakes.

REFERENCES

[1] H. Aït-Kaci, R. Boyer, P. Lincoln, and R. Nasr, "Efficient implementation of lattice operations," *ACM Transactions on Programming Languages and Systems*, vol. 11, no. 1, pp. 115–146, January 1989, [Available online[34]].

[2] V. Strassen, "Gaussian elimination is not optimal," *Numerische Mathematik*, vol. 13, pp. 354–356, 1969.

[3] M. J. Fischer and A. R. Meyer, "Boolean matrix multiplication and transitive closure," in *Proceedings of the 12th Annual Symposium on Switching and Automata Theory*, ser. SWAT'71. Washington, DC, USA: IEEE Computer Society, 1971, pp. 129–131, [Available online[35]].

[4] D. Coppersmith and S. Winograd, "Matrix multiplication via arithmetic progressions," *Journal of Symbolic Computation*, vol. 9, no. 3, pp. 251–280, March 1990, [Available online[36]].

[5] A. Stothers, "On the complexity of matrix multiplication," Ph.D. dissertation, University of Edinburgh, Edinburgh, Scotland, UK, 2010, [Available online[37]].

[6] V. V. Williams, "Breaking the Coppersmith-Winograd barrier," University of California at Berkeley and Stanford University, 2011, [Available online[38]].

[7] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz, "Communication costs os Strassen's matrix multiplication," *Communications of the ACM*, vol. 57, no. 2, pp. 107–114, February 2014, [Available online[39]].

[8] S. Warshall, "A theorem on Boolean matrices," *Journal of the ACM*, vol. 9, no. 1, pp. 11–12, January 1962.

[9] H. S. Warren Jr., "A modification of Warshall's algorithm for the transitive closure of binary relations," *Communications of the ACM*, vol. 18, no. 4, pp. 218–220, April 1975, [Available online[40]].

[10] C. A. R. Hoare, "Algorithm 63: Partition, Algorithm 64: Quicksort," *Communications of the ACM*, vol. 4, no. 7, pp. 321–321, July 1961, [Available online[41]].

---

[32] Actually, the `java.util.TreeSet` does maintain a doubly-linked list for its elements in order to ensure its two ordered iterators (ascending and descending). But this structure is not made public and one cannot splice in new elements from a given found element. But it is a simple matter to modify the source code of `java.util.TreeSet.java` and adapt it to what is needed.

[33] In fact, they see that only as a possible *a posteriori* optimization, but one that would cause their optimal spanning-tree finding algorithm to be incorrect if applied incrementally while it is executed.

[34] `hassan-ait-kaci.net/pdf/encoding-toplas-89.pdf`

[35] `rjlipton.files.wordpress.com/2009/10/matrix1971.pdf`

[36] `www.sciencedirect.com/.../S0747717108800132`

[37] `www.maths.ed.ac.uk/pg/thesis/stothers.pdf`

[38] `www.cs.berkeley.edu/~virgi/matrixmult.pdf`

[39] `www.cs.berkeley.edu/~odedsc/papers/SPAA12-CAPS.pdf`

[40] `dl.acm.org/citation.cfm?id=360746`

[41] `comjnl.oxfordjournals.org/content/5/1/10.full.pdf`

[11] Y. Kazakov, M. Krötzsch, and F. Simančík, "Unchain my $\mathcal{EL}$ reasoner," in *Proceedings of the 24th International Workshop on Description Logics*, R. Rosati, S. Rudolph, and M. Zakharyaschev, Eds., DL'11. Barcelona, Spain: CEUR Workshop Proceedings, July 2011, [Available online[42]].

[12] F. Baader, C. Lutz, and B. Suntisrivaraporn, "CEL—a polynomial-time reasoner for life science ontologies," in *Proceedings of the 3rd International Joint Conference on Automated Reasoning*, U. Furbach and N. Shankar, Eds., IJCAR'06. Seattle, WA, USA: Springer-Verlag LNAI Vol. 4130, August 2006, pp. 287–291, [Available online[43]].

[13] Y. Kazakov, "Consequence-driven reasoning for horn $\mathcal{SHIQ}$ ontologies," in *Proceedings of the 21st International Conference on Artificial Intelligence*, C. Boutilier, Ed., IJCAI'09. Pasadena, CA, USA: Association for the Advancement of Artificial Intelligence, July 2009, pp. 2040–2045, [Available online[44]].

[14] D. Tsarkov and I. Horrocks, "FaCT++ description logic reasoner: System description," in *Proceedings of the 3rd International Joint conference on Automated Reasoning*, U. Furbach and N. Shankar, Eds., IJCAR'06. Seattle, WA, USA: Springer-Verlag, August 2006, pp. 292–297, [Available online[45]].

[15] I. Horrocks and U. Sattler, "A tableau decision procedure for $\mathcal{SHOIQ}$," *Journal of Automated Reasoning*, vol. 39, no. 3, pp. 249–276, July 2007, [Available online[46]].

[16] Z. Manna and R. Waldinger, "Fundamentals of deductive program synthesis," in *Combinatorial Algorithms on Words*, ser. NATO ISI Series, A. Apostolico and Z. Galil, Eds. Springer-Verlag, 1991, [Available online[47]].

[17] R. Shearer, B. Motik, and I. Horrocks, "HermiT: A highly-efficient OWL reasoner," in *Proceedings of the 5th International Workshop on OWL Experiences and Directions*, U. Sattler and C. Dolbear, Eds., OWLED'08. Karlsruhe, Germany: CEUR Workshop Proceedings, October 2008, [Available online[48]].

[18] B. Motik, R. Shearer, and I. Horrocks, "Hypertableau reasoning for description logics," *Journal of Artificial Intelligence Research*, vol. 36, no. 1, pp. 165–228, September 2009, [Available online[49]].

[19] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz, "Pellet: A practical OWL-DL reasoner," *Journal of Web Semantics*, vol. 5, no. 2, pp. 51–53, June 2007, this is a summary; full paper: [Available online[50]].

[20] E. Thomas, J. Z. Pan, and Y. Ren, "TrOWL: Tractable OWL 2 reasoning infrastructure," in *Proceedings of the 7th Extended Semantic Web Conference*, L. Aroyo, G. Antoniou, E. Hyvnen, A. ten Teije, H. Stuckenschmidt, L. Cabral, and T. Tudorache, Eds., ESWC'10. Heraklion, Greece: Springer-Verlag, May-June 2010, pp. 431–435, [Available online[51]].

[21] R. Fikes, P. Hayes, and I. Horrocks, "OWL-QL—a language for deductive query answering on the Semantic Web," *Journal of Web Semantics*, vol. 2, no. 1, pp. 19–29, December 2004, [Available online[52]].

[22] V. Haarslev and R. Möller, "RACER system description," in *Proceedings of the 1st International Joint Conference on Automated Reasoning*, R. Gore, A. Leitsch, and T. Nipkow, Eds., IJCAR'01. Siena, Italy: Springer-Verlag, June 2001, pp. 701–706, [Available online[53]].

[23] V. Haarslev, K. Hidde, R. Möller, and M. Wessel, "The RacerPro knowledge representation and reasoning system," *Semantic Web Journal*, vol. 1, pp. 1–11, March 2011, [Available online[54]].

[24] M. J. Lawley and C. Bousquet, "Fast classification in Protégé: Snorocket as an OWL 2 EL reasoner," in *Proceedings of the 2nd Australasian Ontology Workshop: Advances in Ontologies*, T. Meyer, M. A. Orgun, and K. Taylor, Eds., AOW'10. Adelaide, Australia: ACS, December 2010, pp. 45–50, [Available online[55]].

[25] F. Baader, S. Brandt, and C. Lutz, "Pushing the $\mathcal{EL}$ envelope," in *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, L. P. Kaelbling and A. Saffiotti, Eds., IJCAI'05. Edinburgh, Scotland, UK: Morgan Kaufmann Publishers, July-August 2005, pp. 364–369, [Available online[56]].

[26] P. Cousot, "Abstract interpretation," *ACM Computing Surveys—Symposium on Models of Programming Languages and Computation*, vol. 28, no. 2, pp. 324–328, June 1996, tutorial summary:[Available online[57]].

[27] H. Aït-Kaci, "Data models as constraint systems—A key to the Semantic Web," *Constraint Processsing Letters*, vol. 1, no. 1, pp. 33–88, November 2007, [Available online[58]].

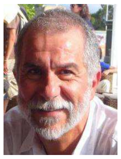
[28] H. Aït-Kaci and R. Di Cosmo, "Compiling order-sorted feature term unification," Digital Paris Research Lab, Rueil-Malmaison, France, PRL Technical Note 7, December 1993, [Available online[59]].

[29] H. Aït-Kaci, *Warren's Abstract Machine: A Tutorial Reconstruction*. Cambridge, MA, USA: The MIT Press, 1991, [Available online[60]].

[30] R. Agrawal, A. Borgida, and H. V. Jagadish, "Efficient management of transitive relationships in large data and knowledge bases," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, J. Clifford, B. G. Lindsay, and D. Maier, Eds., ACM. Portland, Oregon: SIGMOD Record 18(2), May/June 1989, pp. 253–262, [Available online[61]].

[31] H. Aït-Kaci and S. Amir, "Classifying and querying very large taxonomies—a comparative study to the best of our knowledge," $\mathcal{CEDAR}$ Project, LIRIS, Département d'Informatique, Université Claude Bernard Lyon 1, Villeurbanne, France, $\mathcal{CEDAR}$ Technical Report Number 2, May 2013, [Available online[62]].

ABOUT THE AUTHORS



Hassan Aït-Kaci holds a PhD in Computer Science from the University of Pennsylvania (1984), and a Research Director Habilitation from University of Paris 7 (1990) also in CS. He was a member of research staff at MCC in Austin, TX, and at the Digital Paris Research Lab, where he contributed with the LIFE programming language. In 1994, he was appointed full professor at SFU, in Burnaby, BC, Canada, where he held a senior NSERC Industrial Research Chair. In 2000, he joined ILOG, a French multinational company specializing in developing high-performance AI, OR, and Graphics software tools. After ILOG's acquisition by IBM in February 2009, he was a Senior Member of Technical Staff at IBM Canada. Since January 2013, he is holding an ANR Chair of Excellence at the Université Claude Bernard Lyon 1, France, heading the $\mathcal{CEDAR}$ project (Constraint Event-Driven Automated Reasoning) at the LIRIS. Dr. Aït-Kaci has contributed to the formal design and efficient implementation of very high-level and multi-paradigm programming environments (LogIn, LeFun, LIFE), and pioneered the use of constraints for representing and processing knowledge. His interests are in automated reasoning, knowledge representation, declarative computing, and language processing.

Samir Amir received his PhD degree from the University of Lille 1, France, in 2011. His doctoral research focused on finding ways to integrate heterogeneous multimedia metadata. Prior to this, he was a research engineer at CNRS where he worked in several European projects. After his doctorate, he was an Assistant Professor of Computer Science at the University of Lille 1 until 2012. Since January 2013, he has been a postdoctoral researcher in the $\mathcal{CEDAR}$ project at the LIRIS laboratory, Université Claude Bernard Lyon 1, France. His research interests are in the area of database systems and Semantic Web; specifically, knowledge-base reasoning for efficient-query compilation over Big Semantic-Web Data.

---

[42] ceur-ws.org/Vol-745/paper_54.pdf
[43] www.informatik.uni-bremen.de/.../ijcar06.pdf
[44] ijcai.org/papers09/Papers/IJCAI09-336.pdf
[45] www.cs.ox.ac.uk/Ian.Horrocks/.../2006/TsHo06a.pdf
[46] link.springer.com/article/10.1007/s10817-007-9079-9
[47] citeseer.ist.psu.edu/manna92fundamentals.html
[48] www.cs.ox.ac.uk/ian.horrocks/.../2008/ShMH08b.pdf
[49] www.jair.org/media/2811/live-2811-4689-jair.pdf
[50] pellet.owldl.com/papers/sirin05pellet.pdf
[51] homepages.abdn.ac.uk/jeff.z.pan/pages/pub/TPR2010.pdf
[52] www.sciencedirect.com/.../S1570826804000137
[53] www.racer-systems.com/technology/.../HaMo01e.pdf
[54] www.semantic-web-journal.net/.../files/swj109_3.pdf
[55] krr.meraka.org.za/~aow2010/Lawley-etal.pdf
[56] www.ijcai.org/papers/0372.pdf
[57] www.di.ens.fr/~cousot/AI/IntroAbsInt.html
[58] www.cs.brown.edu/people/pvh/CPL/Papers/v1/hak.pdf
[59] hassan-ait-kaci.net/pdf/PRL-TN-7.pdf
[60] wambook.sourceforge.net/
[61] dbs.informatik.uni-halle.de/.../p253-agrawal.pdf
[62] cedar.liris.cnrs.fr/papers/ctr2.pdf