Optimization Strategies for Instance Retrieval

Volker Haarslev
University of Hamburg
Computer Science Department
Vogt-Kölln-Str. 30
22527 Hamburg, Germany
haarslev@informatik.uni-hamburg.de

Ralf Möller
Univ. of Appl. Sciences in Wedel
Computer Science Department
Feldstr. 143
22880 Wedel, Germany
mo@fh-wedel.de

Abstract

In this paper new techniques for optimizing instance retrieval in DL systems are described. The algorithms are evaluated with application examples from a natural language processing application.

1 Motivation

Many applications of description logic (DL) systems use the instance retrieval inference problem [5] in order to properly formalize subtasks. For instance, in [1, 2] a case-study with the application of DL inference services in a natural language (NL) interpretation system is presented. In particular, the inference retrieval service of the RACER system [4] is investigated for various application-specific subtasks (e.g., resolution of referring expressions, content determination, and content realization). In this application many ABoxes are generated on the fly (see [1, 2] for details) and for each ABox a specific instance retrieval query is computed. In order to achieve good performance in the NL application, the performance of the instance retrieval procedure provided by the DL system is crucial. Furthermore, since ABoxes change quite frequently, standard techniques for optimizing instance retrieval using indexing techniques (see below for an explanation) can hardly be employed in order to improve performance because of the overhead of computing index structures in beforehand.

In this paper new techniques for optimizing instance retrieval in DL systems are described. The algorithms are evaluated with application examples from the natural language processing application described above (Racer 1-6-2, 1GHz Pentium). The TBox consists of 165 possibly cyclic "definitions" and "primitive definitions" for concepts as well as domain and range restrictions for 18 roles. In the ABox around 250 individuals are mentioned in concept and role assertions. The DL used in the knowledge base is $\mathcal{ALC}$ with inverse roles. However, the techniques investigated in this paper are also particularly suited for $\mathcal{SHIQ}$ knowledge bases. It is shown that the optimization techniques described in this paper can also be used in combination with indexing techniques. We assume the reader has basic familiarity with description logics (see, e.g., [4] for an introduction to standard inference problems supported by DL systems).

2 Technical Details

Before we start the discussion of technical details of algorithms involving ABoxes, a comment on TBoxes is appropriate. In RACER every ABox is associated with a TBox. In the discussion below, in all algorithms involving ABoxes the TBox is clear from the context and, in order to keep the presentation brief, we do not mention the TBox explicitly. In addition, we assume that all individuals in an ABox are "connected" via role assertions. If this is not the case, an obvious optimization technique uses partitions of ABox assertions in order to allow for a more efficient ABox satisfiability test. Again, details are omitted for the sake of brevity. Furthermore, we assume that the input ABox A in the algorithms below is consistent, and we omit this initial test.

2.1 Optimized Linear Instance Retrieval

Basically, the instance retrieval problem for a query concept C_q and an ABox A can be implemented as a sequence of instance tests for all individuals that are mentioned in an ABox. Hence, $instance\_retrieval(C_q, A)$ can be implemented with a call to $linear\_instance\_retrieval(C_q, contract(i, A), individuals(A))$ where $individuals(A)$ returns the set of individuals mentioned in the ABox A and the function *contract* computes a transformation of an ABox w.r.t. an individual. The idea is to transform tree-like role assertions "starting" from the individual i into equisatisfiable concept assertions with existential restrictions (see [3] for details). The reason is that in RACER, caching (see also [3]) is more effective for concepts rather than for ABox role assertions.

We assume that $ASAT$ is the standard ABox satisfiability test implemented as an optimized tableau calculus. The function $linear\_instance\_retrieval$ is implemented as follows.

Algorithm 1 $linear\_instance\_retrieval(C, A, candidates)$:

```
result := {}
for all ind ∈ candidates do
  if instance?(ind, C, A) then
    result := result ∪ {ind}
  end if
end for
return result
```

The function call $instance?(i, C, A)$ could be implemented as $\neg ASAT(A \cup \{i : \neg C\})$. However, although this implementation of *instance*? is sound and complete, it is quite inefficient. A faster variant uses sound but incomplete initial tests for detecting "obvious" non-instances: the individual model merging test (see [6]) and a subsumption test involving the negation of the query concept (see Algorithm 2).

Algorithm 2 $obvious\_non\_instance?(i, C, A)$:

```
return  individual_model_merging_possible?(i, A, negated_concept(C))
      ∨ subsumes?(negated_concept(C), individual_concept(i))
```

The main idea of the individual model merging is to extract a (pseudo) model for an individual i from a completion of the ABox A. If individual model of i and the

(pseudo) model of $\neg C$ do not "interact", i can easily be shown not to be an instance of C. If one of the "guards" returns *true*, the result of *instance*? is $false$. Otherwise, an "expensive" instance test using the tableau algorithm is performed. The function *negated_concept* returns the negation of its input concept whereas the function *individual_concept* returns the conjunction of the concepts in all ABox concept assertions for an individual i[1]. With these auxiliaries, the function *instance*? can be optimized for the average case but is still sound and complete.

Algorithm 3 *instance*?(i, C, A):

```
if obvious_non_instance?(i, C, A) then
  return false
else
  return ¬ASAT(A ∪ {i : ¬C})
end if
```

Although this variant of *instance*? is significantly faster (mainly due to the individual model merging guard), in the application discussed above, query answering times in the range of 20 seconds were unacceptable. Although for many queries the result consists of a set of only very few individuals (compared to 250 individuals mentioned in the ABox) around a hundred individuals still cause the "expensive" $ASAT$ test to be invoked, regardless of the "guard" in Algorithm 3. Thus, although each $ASAT$ test is quite fast (200 milliseconds), its number should be further reduced in order to provide adequate performance.

2.2 An Improvement: Binary Instance Retrieval

How can ABox satisfiability tests be avoided at all? The observation is that only very few additions $\{i : \neg C\}$ to A lead to an inconsistency in the function *instance*? (i.e., in very few situations i is indeed an instance of C). Therefore, in many realistic scenarios the following procedure seems to be advantageous.

Algorithm 4 *binary_instance_retrieval*$(C, A, candidates)$:

```
if candidates = ∅ then
  return ∅
else
  (partition1, partition2) := partition(candidates)
  return partition_instance_retrieval(C, A, partition1, partition2):
end if
```

We assume now that $instance\_retrieval(C_q, A)$ is implemented using the call $binary\_instance\_retrieval(C_q, contract(i, A), individuals(A))$. The function *partition* is defined in Algorithm 5, it devides a set into two partitions. Given the partitions, *binary_instance_retrieval* calls the function *partition_instance_retrieval*. The idea of *partition_instance_retrieval* (see Algorithm 7) is to first check whether *none* of the individuals in a partition is an instance of the query concept C. This is done with the function *non_instances*? (see Algorithm 6).

[1]Racer supports the unique name assumption. Role assertions for a role R with i on the lefthand side are represented by at-least terms and, depending on the number of different role assertions for i, corresponding conjuncts $(\geq n\, R)$ are generated by *individual_concept*.

Algorithm 5 *partition*(s): /* $s[i]$ refers to the i^{th} element of the set s */

if $|s| \leq 1$ **then**
  return $(s, \emptyset)$
else
  return $(\{s[1], \ldots, s[\lfloor n/2 \rfloor])\}, \{s[\lfloor n/2 \rfloor + 1], \ldots, s[n]\})$
end if

Algorithm 6 *non_instances*?$(cands, C, A)$:

return $ASAT(A \cup \{i : \neg C \mid i \in cands \wedge \neg obvious\_non\_instance?(i, C, A)\})$

The evaluation we conducted with the natural language application indicates that for instance retrieval queries which return only very few individuals a performance gain of up to a factor of 5-10 can be achieved with binary search (compared to linear instance retrieval). The reason is that the *non_instances*? test is successful in many cases. Hence, with one "expensive" ABox test a large set of candidates can be eliminated. The underlying assumption is that, in general, the computational costs of checking whether an ABox $(A \cup \{i : \neg C, j : \neg C, \ldots\})$ is consistent is largely dominated by A alone. Hence, it is assumed that the size of the set of constraints added to A has only a limited influence on the runtime. For knowledge bases with, for instance, cyclic GCIs, this may not be the case, however.

Algorithm 7 *partition_instance_retrieval*$(C, A, partition1, partition2)$:

if $|partition1| = 1$ **then**
  $\{i\} = partition1$
  if *instance*?(i, C, A) **then**
    return $\{i\} \cup binary\_instance\_retrieval(C, A, partition2)$
  else
    return $binary\_instance\_retrieval(C, A, partition2)$
  end if
else if *non_instances*?$(partition1, C, A)$ **then**
  return $binary\_instance\_retrieval(C, A, partition2)$
else if *non_instances*?$(partition2, C, A)$ **then**
  return $binary\_instance\_retrieval(C, A, partition1)$
else
  return $binary\_instance\_retrieval(C, A, partition1)$
    $\cup\ binary\_instance\_retrieval(C, A, partition2)$
end if

2.3 Another Improvement: Dependency-Based Instance Retrieval

Although *binary_instance_retrieval* is found to be faster in the average case, one can do better. If the function *non_instances*? returns $false$ then one can analyze the dependencies of the tableaux structures ("constraints") involved in the clash. If the clash is due to the constraints of only one individual, then, as a by-product of the test, this individual is known to be an instance of the query concept. The individual can be eliminated from the set of candidates to be investigated, and it is definitely part of the solution set. Eliminating candidate individuals detected by dependency analysis prevents the reasoner from detecting the same clash over and over again until

a partition of cardinality 1 is tested. In the example application, runtimes are reduced by another factor of 3 (compared to binary instance retrieval). If the solution set is large compared to the set of individuals in an ABox, there is some overhead compared to linear instance retrieval because only one individual is removed from the set of candidates at a time. In this case a combination of binary instance retrieval and dependency-based instance retrieval is required. In our investigations, dependency-based instance retrieval was always faster than binary instance retrieval. However, details are subject to further research.

2.4 Index-based Instance Retrieval

The techniques introduced in the previous section can also be exploited if indexing techniques are used for instance retrieval (see, e.g., [7, p. 108f.]). Basically, the idea is to reduce the set of candidates that have to be tested by computing the direct types of every individual. The direct types of an individual i are defined to be the most specific concept names (mentioned in a TBox) of which i is an instance. An index is constructed by deriving a function *associated_inds* defined for each concept name C mentioned in the TBox such that $i \in associated\_inds(C)$ iff $C \in direct\_types(i, A)$. Computing the direct types for each individual and the corresponding index *associated_inds* is also called *ABox realization*.

In the following we assume that CN is the set of all concept names mentioned in the TBox (including the name "top"). Furthermore, it is assumed that the function $parents(C)$ returns the most specific subsumers of C whereas $descendants(C)$ returns all subsumees of C including C. Subsumers and subsumees of a concept C are concept names from CN. The function $synonyms(C)$ returns all concept names from CN which are equivalent to C. Index-based instance retrieval is implemented as follows.

Algorithm 8 *index_based_instance_retrieval*(C, A):

if $\exists N \in CN : N \in synonyms(C)$ **then**
  return $\bigcup_{D \in descendants(C)} associated\_inds(D)$
else
  $known\_results := \bigcup_{D \in descendants(C)} associated\_inds(D)$
  $candidates := \bigcup_{P \in parents(C)} associated\_inds(P)$
  return $known\_results \cup instance\_retrieval(C, A, candidates)$
end if

It is obvious that *instance_retrieval* can be implemented by any of the techniques introduced above. However, computing the index structures (i.e., the function *associated_inds*) is rather time-consuming. The standard way to compute the index is to compute the direct types for each individual mentioned in the ABox separately (one-individual-at-a-time approach). In order to compute the direct types of individuals w.r.t. a TBox and an ABox, the TBox must be classified, i.e., for each concept name mentioned in the TBox (and the ABox) the most-specific subsumers (function *parents*) and least-specific susumees (function *children*) are precomputed. Thus, *parents* and *children* are not really queries but just functions accessing results stored in data structures. Another view is that the children (or parents) relation defines a lattice whose nodes are concept names. The root node is called *top*, the bottom node is called *bottom*. This lattice is also referred to as "taxonomy". The function $direct\_types(i, A)$ is im-

plemented as a call to $traverse1(i, top, A)$, which is given as Algorithm 9, i.e., the idea is to compute the direct types of an individual i by traversing the taxonomy of concept names CN starting from the *top* node. If i can be proven to be an instance of a node CN w.r.t. the ABox, the traversal continues at the children of CN until the node *bottom* is reached. An individual i is "sieved" into the taxonomy using a traversal process.

The traversal process is computationally expensive, and although an enormous speedup can be achieved using, for instance, the function *obvious_non_instance*? for implementing the instance test *instance*? in the same spirit as explained above, in general, many "expensive" ABox consistency checks must be performed. Using the one-individual-at-a-time approach for realization-based instance retrieval requires about 350 seconds for the investigated example application.

Algorithm 9 $traverse1(i, C, A)$:

```
if instance?(i, C, A) then
  Cs := ⋃_{D∈children(C)} traverse1(i, D, A)
  if Cs = ∅ then
    return {C}
  else
    return Cs
  end if
else
  return ∅
end if
```

Algorithm 10 $compute\_index\_one\_individual\_at\_a\_time(A)$:

```
Initialization: ∀C ∈ CN : associated_inds(C) := ∅
for all ind ∈ individuals(A) do
  for all C ∈ traverse1(ind, top, A) do
    associated_inds(C) := associated_inds(C) ∪ {ind}
  end for
end for
```

Since for many applications a runtime of up to 6 minutes for computing the index is not tolerable, new techniques had to be developed. The main problem is that for computing the index structure *associated_inds* the direct types are computed for every individual in isolation. Rather than asking for the direct types of every individual in a separate query, we investigated the idea of using *sets* of individuals which are "sieved" into the taxonomy, The idea is to use the procedure *non_instances*? to check whether all indiviudals from a set of candidates are all obviously no instances of a given concept C (w.r.t. an ABox). If *non_instances*? returns *true*, many single ABox tests can be avoided. We call the approach the sets-of-individuals-at-a-time approach.

In the natural language application we investigated, answering a specific query with realization-based instance retrieval and the set-of-individuals-at-a-time approach requires ca. 30 seconds with dependency-based instance retrieval (and 80 seconds with binary instance retrieval). Thus, for this specific application the performance gain is a factor of three. But still it holds that, if ABoxes are not static, i.e., if ABoxes are computed on the fly, and if only very few queries are posed w.r.t. the ABoxes, then

Algorithm 11 $traverse2(inds, C, A, has\_member)$:

if $inds \neq \emptyset$ **then**
  for all $D \in children(C)$ **do**
    if $has\_member(D) = unknown$ **then**
      $instances\_of\_D := instance\_retrieval(D, inds, A)$
      $has\_member(D) := instances\_of\_D$
      $traverse2(instances\_of\_D, D, A, has\_member)$
    end if
  end for
end if

the direct implementation of instance retrieval as search without exploiting indexes is much faster (and possible even without TBox classification). A detailed analysis about the structure of knowledge bases for which the techniques are most effective of for which they even fail will be given in a subsequent technical report.

Algorithm 12 $compute\_index\_sets\_of\_individuals\_at\_a\_time(A)$:

for all $C \in CN$ **do**
  $has\_member(C) := unknown$
  $associated\_inds(C) := \emptyset$
end for
$traverse2(individuals(A), top, A, has\_member)$ $has\_member(top) := individuals(A)$
for all $C \in CN$ **do**
  if $has\_member(C) \neq unknown$ **then**
    for all $ind \in has\_member(C)$ **do**
      if $\neg\exists D \in children(C) : ind \in has\_member(D)$ **then**
        $associated\_inds(C) := associated\_inds(C) \cup \{ind\}$
      end if
    end for
  end if
end for

2.5 Exploiting Query Subsumption

If a query concept is a concept name, which is the case in many queries of our example application, query answering time is reduced to zero if index-based instance retrieval is used. In addition, if a concept term rather than a concept name is used as a query, the index still reduces the set of candidates. Now, as we have discussed above, in many applications computing an index is not feasible. In order to reduce the set of candidates for instance retrieval tests also in this situation, Racer supports query subsumption. Each query is inserted as a node into the taxonomy (w.r.t. to the correct parents and children relation to existing nodes). With each query node, the instance retrieval result is associated. If a new query is to be answered, the result of previously answered queries can be exploited in a similar way as shown in the algoritm *index_based_instance_retrieval*. Hence, the number of candidates can be considerably reduced in many cases if a subsumption relationship to a previously answered query is detected. Due to space constraints we cannot explain details in this paper.

3 Conclusion

One important technique for speeding up common instance retrieval tests that return only a small set of individuals is to use a binary partitioning algorithm to eliminate half of the candidates using a single ABox satisfiability test. Furthermore, and in many cases even more effective, another technique based on an analysis of clash dependencies is investigated. We found that in the natural language application, a speedup factor of 10 to 30 can be achieved compared to linear instance retrieval. Hence, if the result sets of instance retrieval queries are only small compared to the set of initial candidates, a speedup of one order of magnitude can be achieved in the average case. In addition, the Racer system supports query subsumption in order to reuse previous retrieval results for reducing the set of candidates for a new query. At last it is indicated that binary or dependency-based instance retrieval can also be employed when index structures are available. The article also discusses new techniques for computing index structures by also exploiting fast instance retrieval functionality.

With RACER, an optimized DL system is available that supports optimizations for quickly answering instance retrieval queries w.r.t. ABoxes that often change as well as fast index-based instance retrieval for more "static" ABoxes where the initial overhead does not impose a problem. Initial applications demonstrate that ABox reasoning can indeed be used to solve important problems using the declarative means of description logics.

References

[1] Malte Gabsdil, Alexander Koller, and Kristina Striegnitz. Building a text adventure on description logic. In *International Workshop on Applications of Description Logics, Vienna, September 18*. CEUR Electronic Workshop Proceedings, `http://ceur-ws.org/Vol-44/`, 2001.

[2] Malte Gabsdil, Alexander Koller, and Kristina Striegnitz. Playing with description logic. In *Proceedings Second Workshop on Methods for Modalities M4M-02*. `http://turing.wins.uva.nl/~m4m/M4M2/program.html`, November 2001.

[3] Volker Haarslev and Ralf Möller. Consistency testing: The RACE experience. In *Proceedings International Conference Tableaux'2000*, volume 1847 of *Lecture Notes in Artificial Intelligence*, pages 57–61. Springer-Verlag, 2000.

[4] Volker Haarslev and Ralf Möller. Description of the RACER system and its applications. In *Proc. of the 2001 Description Logic Workshop (DL 2001)*, pages 132–141. CEUR Electronic Workshop Proceedings, `http://ceur-ws.org/Vol-49/`, 2001.

[5] Volker Haarslev and Ralf Möller. RACER system description. In *Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2001)*, volume 2083 of *Lecture Notes in Artificial Intelligence*, pages 701–705. Springer-Verlag, 2001.

[6] Volker Haarslev, Ralf Möller, and Anni-Yasmin Turhan. Exploiting pseudo models for tbox and abox reasoning in expressive description logics. In *Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2001)*, volume 2083 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2001.

[7] Bernhard Nebel. *Reasoning and Revision in Hybrid Representation Systems*, volume 422 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1990.