

Semantic Networks and Description Logics

Description Logics – Implementation Techniques

Knowledge Representation and Reasoning

Jan 9, 2005

Description Logics – Implementation Techniques

Motivation

Classification

Subsumption Tests

Provisional Balance

Tableau Proofs

Preprocessing

Motivation

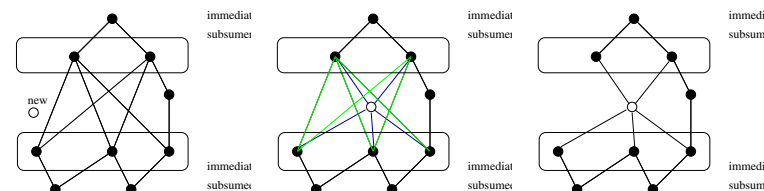
Motivation

- ▶ It is easy to write a **subsumption checker** for, e.g., *ALC*
- ▶ It is also easy to extend this to a small **TBox/Abox system**
- ▶ On top of that we need
 1. a **graphical user interface**
 2. an **application interface** with the right kind of services
 3. an **efficient implementation** of the basic reasoning services
- ▶ Possible areas of optimization:
 1. Classification
 2. Subsumption test
 3. Preprocessing the KB

Classification Partial-Order Sorting

Classification: Sorting in Partial Orders

- ▶ Compute the hierarchy induced by the subsumption relationship (similar to sorting)
- ▶ Also important in other contexts
- ▶ Incremental creation
 1. Find immediate subsumers and subsumees in hierarchy
 2. Place new concept at this place (and delete superfluous links)



Worst-Case Lower Bound for Classification

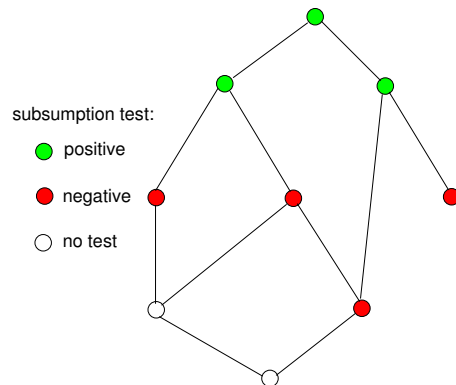
- ▶ Count number of **subsumption tests**
- ▶ What is the **worst case**?
- ↪ Worst case: All concepts are **unrelated**
- $O(n^2)$ for n being the number of concept names.
- ▶ Analytical comparison (for the **average case**) between different methods is probably difficult because it is **unknown**, how many partial orders exist for a given number of elements
- ↪ Empirical comparisons are the best we can hope for

Method 0: Brute force

- The **brute-force** method. For each new concept C :
- ▶ For each concept D in the hierarchy, check
 - ▶ whether C subsumes D and
 - ▶ whether D subsumes C .
 - ▶ Compute from those the immediate subsumers and subsumees
 - ↪ $2n$ tests for each new concept
 - $\sum_{i=0}^{n-1} 2i = n \times (n - 1)$ for the whole KB

Method 1: Simple Traversal

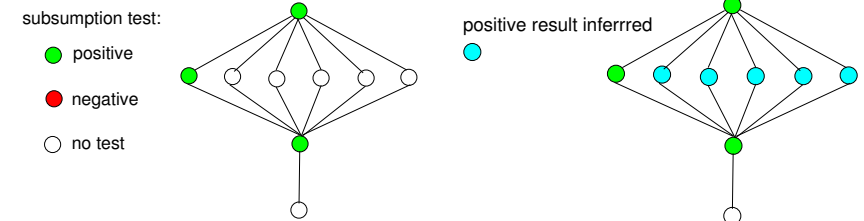
- ▶ **Top search**: Traverse hierarchy top down (e.g. depth-first), and try to identify direct subsumers by checking subsumption and mark subsumption failures.



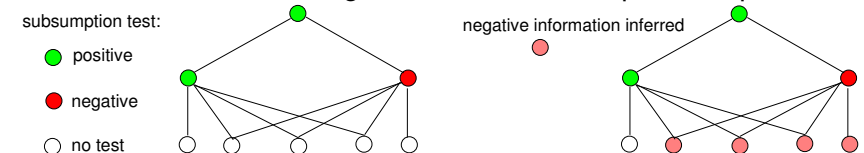
- ▶ **Bottom search**: dual, from bottom element

Method 2: Smart Traversal

- ▶ Exploit information from previous subsumption tests
- ▶ Use information about positive results in the top search path



- ▶ Use information about negative results in the top search path

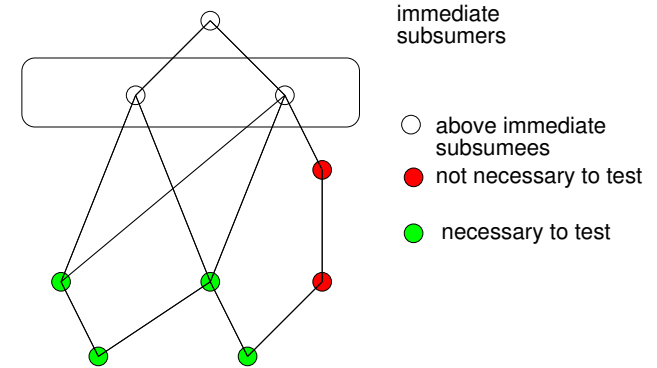


Three Alternatives

- ▶ **Note:** Smart traversal is **provably** as good as simple traversal (counting the number of subsumption tests) and the additional overhead is negligible (0.5% of the subsumption costs in a typical KB)
- ▶ **Alternatives:**
 1. Depth-first search using only positive information
 2. Breadth-first search using only negative information
 3. Depth-first search using both negative and positive information
- ↪ Our results: **Breadth-first** search is **best**. Using only positive information does not help much.

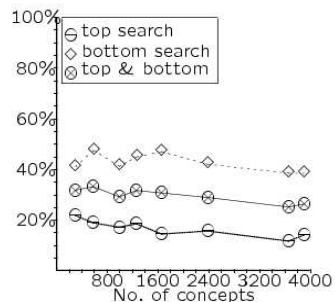
Smart Traversal: Bottom Search

- ▶ Dual optimization to top search (starting at bottom)
- ▶ In addition, it is sufficient to consider only nodes that are subsumed by all immediate subsumers!

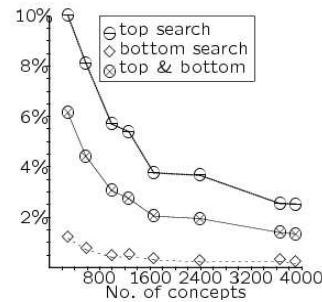


Number of Subsumption Tests Relative to “Brute Force” Method

- ▶ In 1992, we evaluated the effects of different techniques
- ▶ On real KBs, the book keeping overhead turned out to be relatively low compared with the time for subsumption checking
- ▶ We measured the effects on randomly generated KBs modeled after the structure of real KBs



Simple traversal



Smart traversal

Method 3: Insertion into Chains

- ▶ Building up a linearly ordered list incrementally, it is a good idea to use **binary search** for identifying the next insertion point
- ▶ Idea:
 - ▶ Compute a good **chain-covering** of the partial order, i.e., a set of linearly ordered chains covering all elements
 - ▶ Minimal chain covering is (of course) **NP-hard**. Therefore we used a heuristic
 - ▶ Do a **binary search** in all chains
 - ▶ and apply the same **propagation** of positive and negative information as in the smart traversal
- ↪ One gets a super-set of the set of immediate subsumers (and subsumees)
- in most typical cases a little bit worse than with smart traversal (approx. 10%)
- better only if the connectedness of the order was higher than in typical KBs

Results for Chain Insertion

Type of KB	No. of nodes	Average degree	Average no. of pred. & succ.	Breadth	Depth	Relative no. of comparisons
real KB	184	1.71	5.67	105	6	103.7%
	241	1.91	6.38	124	6	100.3%
	254	1.99	13.02	135	6	91.8%
	269	1.72	5.16	164	7	107.9%
	330	1.85	8.13	141	12	110.0%
random KB	298	2.36	8.88	142	8	115.7%
	583	2.58	12.24	330	7	114.5%
	992	2.73	16.77	478	10	111.7%
	1263	3.18	16.61	661	11	108.9%
	1659	3.19	18.86	927	10	110.3%
after real KB	2389	3.50	25.49	1188	10	111.3%
	3658	3.82	27.20	1703	8	105.3%
	3905	4.04	33.95	1858	11	99.9%
random partial orders	301	7.67	42.11	88	8	73.2%
	301	6.40	10.43	168	6	102.7%
	301	4.22	5.68	205	4	101.3%
	586	9.93	72.55	144	9	67.2%
	586	10.39	20.42	301	7	103.3%
	586	7.72	11.50	353	5	102.9%
	995	5.52	250.24	85	28	16.2%
	995	16.40	62.88	354	9	91.4%
	995	13.58	28.38	506	6	105.1%
	1266	5.78	321.19	100	30	12.9%
	1266	18.22	76.87	438	9	89.8%
	1266	17.82	41.70	592	6	101.1%

Exploiting the Concept Structure

1. If a concept is defined conjunctively, i.e.,

$$A \doteq X \sqcap Y \sqcap \dots \sqcap Z$$

all **atomic concepts** (X, Y, Z) are obviously **super-concepts**. So one can mark these concepts as subsuming (and can propagate this information)

2. Primitive concepts (introduced with \sqsubseteq) can be **ignored** in the **bottom search** phase, if we classify in definition order (i.e., before a term is used, it will be defined)
3. **Primitive components** of a concept are the undefined concepts in the unfolded form of a concept. A subsumption test between two conjunctively defined concepts can only be successful if the primitive components are in an set-inclusion relationship

Caching and Lazy Unfolding

- ▶ Instead of unfolding a concept and then testing subsumption, one can **delay the expansion** of a defined concept until no other way to prove unsatisfiability is possible
 - ▶ Instead of testing for clashes of primitive concepts, we also look for clashes on the **level of named, defined concepts**.
 - ▶ Finally, we can in addition also look for clashes implied by the subsumption hierarchy, e.g., **Man and \neg Human** form a clash.
- ↔ These optimization saved roughly 50% of the runtime on the realistic KBs

Provisional Balance

- ▶ The described methods (and a few others) were used in speeding up the KRIS system
- ▶ We achieved a speedup factor of around 400
- ▶ The difference between a first **prototype implementation** that just implements the principles and a **highly optimized system** can be huge – in particular when we are dealing with worst-case exponential time/space problems
- ▶ Interestingly, the method we believed to be superior (**chain insertion**) was not effective at all
- ▶ One has to look for what **typical cases** require – and these cases might not be easily recognizable as **polynomial special cases**
- ▶ There are techniques especially geared towards more **expressive DLs** – and some we will have a short look at

Semantic Branching in Tableau Proofs

- ▶ Consider the System: $S = \{x : (A \sqcup B), x : (A \sqcup C)\}$
- ▶ Ordinarily,
 - ▶ we would first try out A and B , nondeterministically.
 - ▶ Then we would make another nondeterministic choice between A and C .
- ▶ if A leads to clash, we would try that out twice.
- ↔ Use *semantic branching* instead (similar to what is done in the Davis-Putnam procedure)
- Branch over A and $\neg A$!
 - ▶ This avoids double work on A and leads to **tighter constraints** (when using $\neg A$ in the other branch)
 - ▶ Experience shows that the additional overhead for propagating $\neg A$ is usually negligible
- *Local simplification* becomes possible

Boolean Constraint Propagation

- ▶ When using semantic branching, we can also simplify disjunctions by propagating the concepts, we have branched on, using *Boolean constraint propagation* (aka *unit propagation*).
- ▶ Example:

$$S = \{x : (C \sqcup (D_1 \sqcap D_2)), x : (\neg D_1 \sqcup \neg D_2 \sqcup C), x : \neg C\}$$

- ▶ Because of $x : \neg C$, this can be simplified to

$$S = \{x : (D_1 \sqcap D_2), x : (\neg D_1 \sqcup \neg D_2 \sqcup C), x : \neg C\}$$

- ▶ This in turn can be simplified to

$$S = \{x : (D_1 \sqcap D_2), x : C, x : \neg C\}$$

- ↔ This is easily detected as unsatisfiable!
- Note that **no nondeterministic branching** was necessary!

More Methods for Speeding up Tableau Proofs

- ▶ *Dependency-directed backtracking*: Try to detect the **reason for a clash** and jump back (instead of exploring fruitless alternatives)
- ▶ *Heuristic guided search*: Branch on the right disjunct
- ↔ *MOMS heuristic*: Branch on the disjunct with the **maximum** number of **occurrences** in disjunctions of **minimum size**.
- ↔ Seems not to be a good heuristic in DLs.
- The best heuristic seems to be *oldest first*, which selects a disjunct from the disjunction depending on the least recent branching point
- ▶ Other techniques such as caching tableau trees and reusing them are also possible

Avoiding Internalization

- ▶ **Internalization** is a powerful technique to deal with general concept inclusion statements (we have seen in the last lecture)
- ▶ However, it introduces a lot of *non-deterministic choices*. Remember: $C_i \sqsubseteq D_i$ is translated into $D_i \sqcup \neg C_i$, which is enforced on every domain element (using the quasi-universal role)
- ↔ Each inclusion statement leads to a nondeterministic choice point for each element!
- ▶ Avoid if possible:
 - ▶ Collect multiple primitive definitions:

$$(A \sqsubseteq C_1), \dots, (A \sqsubseteq C_n) \iff (A \sqsubseteq C_1 \sqcap \dots \sqcap C_n)$$

- ▶ Partition the terminology into two parts \mathcal{T}_u and \mathcal{T}_g , where the former is **unfoldable** and the latter is the general part. Then we can apply *lazy unfolding* to all statements in \mathcal{T}_u .

Absorption

- ▶ Since general inclusion statements lead to nondeterminism, it is a good idea to reduce these as much as possible.
- ▶ Use the following equivalences to *absorb* general inclusion statements into primitive concept definitions:

$$C_1 \sqcap C_2 \sqsubseteq D \iff C_1 \sqsubseteq D \sqcap \neg C_2$$

$$C \sqsubseteq D_1 \sqcap D_2 \iff (C \sqsubseteq D_1), (C \sqsubseteq D_2)$$

- ▶ **Example Absorption.** Assume: $\text{Geometric-figure} \sqsubseteq \text{Figure}$, $\text{Geometric-figure} \sqcap \exists \text{angles.Three} \sqsubseteq \exists \text{sides.Three}$.
- ▶ The latter inclusion statement can be massaged into:

$$\text{Geometric-figure} \sqsubseteq \exists \text{sides.Three} \sqcup \neg \exists \text{angles.Three}$$

- ▶ Then it can be absorbed into:

$$\text{Geometric-figure} \sqsubseteq \text{Figure} \sqcap \exists \text{sides.Three} \sqcup \neg \exists \text{angles.Three}$$



- ▶ Using this technique can make a huge difference

Conclusion

- ▶ Implementing inference methods for DLs is easy – if we do not care about *efficiency*
- ▶ The difference between a *prototype implementation* and a carefully *optimized system* can be several orders of magnitude
- ▶ Optimizations are possible on *classification level*, the *subsumption testing level*, and the *tableau proof level*
- ▶ These optimization should be geared towards the *typical case*
- ▶ It is nearly impossible to decide analytically, which method is the right one, only *empirical comparisons* can help here
- ▶ **Note:** Although there exists now a number of efficient DL methods, in general there will remain the problem that sometimes these systems just have to give up because a subsumption query is too difficult!

Literature

Literature

-  F. Baader, B. Hollunder, B. Nebel, H.-J. Profitlich, and E. Franconi, An Empirical Analysis of Optimization Techniques for Terminological Representation Systems or “Making KRIS get a move on”, *Applied Intelligence* 4(2): 109-132, May 1994.
-  I. Horrocks, Implementation and Optimization Techniques, in: Baader, F., D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider (eds.), *The Description Logic Handbook: Theory, Implementation, Applications*, Cambridge University Press, Cambridge, UK, 2003.