Laboratoire
d'Informatique
de Robotique
et de Microélectronique
de Montpellier
LIRMM

RESEARCH REPORT

Deduction in existential conjunctive first-order logic: an algorithm and experiments

Khalil Ben Mohamed
benmohamed@lirmm.fr

Michel Leclère
leclere@lirmm.fr

Marie-Laure Mugnier
mugnier@lirmm.fr

March 2010 R.R.LIRMM 10010

161, rue Ada • 34392 Montpellier Cedex 5 • France
Tel. : 33 (0) 4 67 41 85 85 • Fax : 33 (0) 4 67 41 85 00 • www.lirmm.fr

Abstract

We consider the deduction problem in the existential conjunctive fragment of first-order logic with atomic negation. This problem can be recast in terms of other database and artificial intelligence problems, namely query containment, clause entailment and boolean query answering. We refine an algorithm scheme that was proposed for query containment, which itself improves other known algorithms in databases. To study it experimentally, we build a random generator and analyze the influence of several parameters on the problem instance difficulty. Using this methodology, we experimentally compare several heuristics. We also present preliminary results on the comparison of our algorithm, which is based on homomorphism checks, to the theorem prover Prover9, which is based on the resolution method.

Keywords: Deduction, Negation, Conjunctive Queries with Negation, Graphs, Homomorphism, Algorithm, Heuristics, Experiments

Contents

1 Introduction

We consider deduction checking in the fragment of first-order logic (FOL) composed of *existentially closed conjunctions of literals* (without functions). The deduction problem, called DEDUCTION in this paper, takes as input two formulas f and g in this fragment and asks if f can be deduced from g (noted $g \vdash f$). DEDUCTION is representative of several fundamental artificial intelligence and database problems. It can be immediately recast as a *query containment* checking problem, which is a fundamental database problem. This problem takes two queries q_1 and q_2 as input, and asks if q_1 is contained in q_2, i.e. if the set of answers to q_1 is included in the set of answers to q_2 for all databases (e.g. [AHV95]). Algorithms based on query containment can be used to solve various problems, such as query evaluation and optimization [CM77, ASU79] or rewriting queries using views [Hal01]. So-called (positive) *conjunctive queries* form a class of natural and frequently used queries and are considered as basic queries in databases [CM77, Ull89] and more recently in the semantic web. Conjunctive queries with negation extend this class with negation on atoms. Query containment checking for conjunctive queries with negation is essentially the same problem as DEDUCTION, in the sense that there are natural polynomial reductions from one problem to another, which preserve the structure of the objects. Another related problem in artificial intelligence is the *clause entailment* problem, i.e. a basic problem in inductive logic programming [MR94]: given two clauses C_1 and C_2, does C_1 entail C_2? If we consider first-order clauses, i.e. universally closed disjunctions of literals, without function symbols, by contraposition we obtain an instance of DEDUCTION. *Query answering* is a key problem in the domain of knowledge representation and reasoning. Generally speaking, it takes a knowledge base and a query as input and asks for the set of answers to the query that can be retrieved from the knowledge base. When the query is boolean, i.e. with a yes/no answer, the problem can be recast as checking whether the query can be deduced from the knowledge base. When the knowledge base is simply composed of a set of positive and negative factual assertions, i.e. existentially closed conjunctions of literals (possibly stored in a relational database), and the query is a boolean conjunctive query with negation, we again obtain DEDUCTION. Integration of an ontology in this knowledge base is discussed in the conclusion of this paper.

If the considered fragment is restricted to positive literals, deduction checking is "only" NP-complete and this has been intensively studied from an algorithm viewpoint, in particular in the form of the equivalent constraint satisfaction problem [RvBW06]. In contrast, when atomic negation is considered, deduction checking becomes Π_2^p-complete[1](e.g. [FNTU07]) and very few algorithms for solving it can be found in the literature. Several algorithms have been proposed for the database query containment problem [Ull97][WL03][LM07]. They all use homomorphism as a core notion. We have not found logical algorithms dedicated to a problem equivalent to DEDUCTION. Theorem provers in first-order logic consider more general fragments.

In this paper, we refine the algorithm scheme introduced in [LM07] for query containment checking, which itself improves other algorithms proposed in databases. To study it experimentally, we build a random generator and analyze the influence of several parameters on the problem instance difficulty. We experimentally compare several heuristics using this methodology. We also present preliminary results obtained on the comparison of our algorithm, which is based on homomorphism, to logical provers, namely the free tools Prover9 (the successor of the Otter prover [McC03b]), based on the resolution method, and its complementary tool Mace4 [McC03a].

Paper layout. Section 2 introduces the framework. In Section 3, we present our experimental methodology and choices. Section 4 is devoted to the comparison of several heuristics, which leads to refine the algorithm. First results on the comparison to logical provers are presented in Section 5. Section 6 outlines the prospects of this work.

2 Framework

We note $FOL\{\exists, \wedge, \neg_a\}$ the fragment of FOL composed of existentially closed conjunctions of literals, with constants but without other function symbols. A for-

[1]$\Pi_2^p = (co\text{-}NP)^{NP}$

mula in $FOL\{\exists, \wedge, \neg_a\}$ can also be seen as a *set* of (positive and negative) literals.

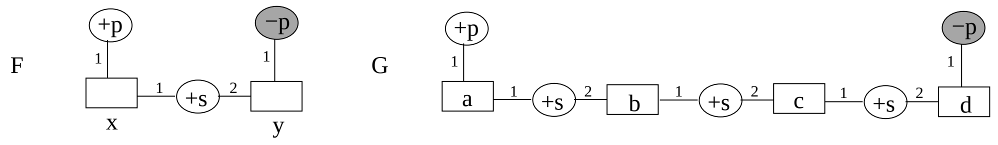
In [LM07], queries (i.e. formulas in the present paper) are seen as labeled graphs. This allows us to rely on graph notions that have no simple equivalent in logic (such as pure subgraphs, see later). More precisely, a formula f is represented as a bipartite, undirected and labeled graph F, called *polarized graph (PG)*, with two kinds of nodes: term nodes and predicate nodes. Each term of the formula becomes a term node, that is unlabeled if it is a variable, otherwise it is labeled by the constant itself. A positive (resp. negative) literal with predicate r becomes a predicate node labeled $+r$ (resp. $-r$) and it is linked to the nodes assigned to its terms. The labels on edges correspond to the position of each term in the literal (see Figure 1 for an example). For simplicity, the subgraph corresponding to a literal, i.e. induced by a predicate node and its neighbors, is also called a *literal*. We note it $+r(t_1, \ldots, t_n)$ (resp. $-r(t_1, \ldots, t_n)$) if the predicate node has label $+r$ (resp. $-r$) and list of neighbors $t_1, \ldots, t_n$. The notation $\sim r(t_1, \ldots, t_n)$ indicates that the literal with predicate r may be positive or negative. Literals $+r(t_1, \ldots, t_n)$ and $-r(u_1, \ldots, u_n)$ with the same predicate but different signs are said to be *opposite*. Literals $+r(t_1, \ldots, t_n)$ and $-r(t_1, \ldots, t_n)$ with the same list of arguments are said to be *contradictory*. Given a predicate node label (resp. literal) l, $\bar{l}$ denotes the complementary predicate label (resp. literal) of l, i.e. it is obtained from l by reversing its sign. Formulas are denoted by small letters (f and g) and the associated graphs by the corresponding capital letters (F and G). We note $G \vdash F$ *iff* $g \vdash f$. A PG is *consistent* if it does not contain two contradictory literals (i.e. the associated formula is satisfiable).

Homomorphism is a core notion in this work. A *homomorphism* h from a PG F to a PG G is a mapping from nodes of F to nodes of G, which preserves bipartition (the image of a term - resp predicate - node is a term - resp. predicate - node), preserves edges (if rt is an edge with label i in F then $h(r)h(t)$ is an edge with label i in G), preserves predicate node labels (a predicate node and its image have the same label) and can instantiate term node labels (if a term node is labeled by a constant, its image has the same label, otherwise the image can be any label). On the associated formulas f and g, it corresponds to a *substitution* h from f to g such that $h(f) \subseteq g$. When there is a homomorphism h from F to G, we say that F *maps* to G by h. F is called the *source* graph and G the *target* graph.

Definition 1 (Complete graph and completion) *Let G be a consistent PG. It is* complete *w.r.t. a set of predicates $\mathcal{P}$, if for each $p \in \mathcal{P}$ with arity k, for each k-tuple of term nodes (not necessarily distinct) $t_1, \ldots, t_k$ in G, it contains $+p(t_1, \ldots, t_k)$ or $-p(t_1, \ldots, t_k)$. A* completion *G' of G is a PG obtained from G by repeatedly adding new predicate nodes (on term nodes present in G) without yielding inconsistency of G. Each addition is a* completion step. *A completion of G is called* total *if it is a complete graph w.r.t. the set of predicates considered, otherwise it is called* partial.

If F and G have only positive literals, $G \vdash F$ *iff* F maps to G. When we consider positive and negative literals, only one side of this property remains true: if F maps to G then $G \vdash F$; the converse is false, as shown in Example 1.

Example 1 *See figure 1: F does not map to G but $g \vdash f$. Indeed, if we complete g w.r.t. predicate p, we obtain the formula g' (equivalent to g): $g' = (g \wedge p(b) \wedge p(c)) \vee (g \wedge \neg p(b) \wedge p(c)) \vee (g \wedge p(b) \wedge \neg p(c)) \vee (g \wedge \neg p(b) \wedge \neg p(c))$. Each of the four conjunctions of g' is a way to complete g w.r.t. p. F maps to each of the graphs associated with them. Thus f is deductible from g'.*



$$f = \exists x \exists y\ (p(x) \wedge s(x,y) \wedge \neg p(y))$$
$$g = p(a) \wedge s(a,b) \wedge s(b,c) \wedge s(c,d) \wedge \neg p(d)$$

Figure 1: Polarized graphs associated with f and g.

One way to solve DEDUCTION is therefore to generate all total completions obtained from G using predicates appearing in G, and then to test if F maps to each of these graphs.

Theorem 1 *[LM07] Let F and G be two PGs (with G consistent), $G \vdash F$* iff *for all G^c, total completion of G w.r.t. the set of predicates appearing in G, F maps to G^c.*

We can restrict the set of predicates considered to those appearing in opposite literals both in F and in G [LM07]. In the sequel, this set is called the *completion vocabulary* of F and G and denoted $\mathcal{V}$.

A brute-force approach, introduced in [Ull97], consists of computing the set of total completions of G and checking the existence of a homomorphism from F to each of them. However, the complexity of this algorithm is prohibitive: $\mathcal{O}(2^{(n_G)^k \times |\mathcal{V}|} \times hom(F, G^c))$, where n_G is the number of term nodes in G, k is the maximum arity of a predicate, $\mathcal{V}$ is the completion vocabulary and $hom(F, G^c)$ is the complexity of checking the existence of a homomorphism[2] from F to G^c.

[2]Homomorphism checking is NP-complete. A brute-force algorithm solves it in $\mathcal{O}(n_G^{n_F})$, where n_F is the number of term nodes in F.

Two types of improvements of this method are proposed in [LM07]. First, let us consider the space leading from G to its total completions and partially ordered by the relation “subgraph of”. This space is explored as a binary tree with G as root. The children of a node are obtained by adding, to the graph associated with this node (say G'), a predicate node in positive and negative form (each of the two new graphs is thus obtained by a completion step from G'). The aim is to find a set of partial completions covering the set of total completions of G, i.e. the question becomes: “Is there a set of partial completions $\{G_1, \ldots, G_n\}$ such that (1) F maps to each G_i for $i = 1 \ldots n$; (2) each total completion G^c of G is covered by a G_i (i.e. G_i is a subgraph of G^c) ?” After each completion step, we check whether F maps to the current partial completion: if yes, this completion is one of the sought G_i, otherwise the exploration continues.

Figure 2 illustrates this method on the very easy case of Example 1. Two graphs G_1 and G_2 are built from G, respectively by adding $+p(c)$ and $-p(c)$. F maps to G_1, thus there is no need to complete G_1. F does not map to G_2: two graphs G_3 and G_4 are built from G_2, by adding $+p(b)$ and $-p(b)$ to G_2. F maps to G_3 and to G_4, respectively. Finally, the set proving that F is deductible from G is $\{G_1, G_3, G_4\}$ (and there are four total completions of G w.r.t. p). Algorithm 1 implements this method (the numbers in the margin are relative to the refinements studied in Section 4).

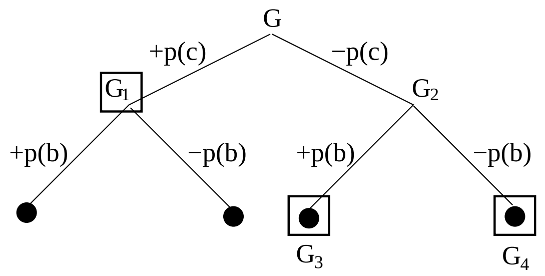


Figure 2: The search tree of Example 1. Each black dot represents a G^c and each square a G_i.

Algorithm 1: **recCheck**(G)

```
    Input: a consistent PG $G$
    Data: $F$, $\mathcal{V}$
    Result: true if $G \vdash F$, false otherwise
    begin
        if there is a homomorphism from $F$ to $G$ then return true ;
        if $G$ is complete w.r.t. $\mathcal{V}$ then return false ;
(3)     \*** Filtering step ***\
(1)     Choose $r \in \mathcal{V}$ and $t_1, \dots, t_n$ in $G$ such that $+r(t_1, \dots, t_n) \notin G$ and
        $-r(t_1, \dots, t_n) \notin G$ ;
        Let $G'$ be obtained from $G$ by adding $+r(t_1, \dots, t_n)$ ;
        Let $G''$ be obtained from $G$ by adding $-r(t_1, \dots, t_n)$ ;
(2)     return recCheck($G'$) AND recCheck($G''$) ;
    end
```

The second kind of improvement consists of identifying subgraphs of F for which there must be a homomorphism to G when $G \vdash F$. Such a subgraph, say F', can be used as a filter to detect a failure before entering the completion process: if F' does not map to G, then $G \nvdash F$. In [WL03], this property is exhibited for F^+, which is the set of all positive literals in F. This result is generalized in [LM07] with the notions of *pure subgraph* and *compatible* homomorphism.

Definition 2 (pure subgraph) *A PG is said to be* pure *if it does not contain opposite literals (i.e. each predicate appears only in one form, positive or negative). A* pure subgraph *of F is a subgraph of F that contains all term nodes in F (but not necessarily all predicate nodes)*[3] *and is pure.*

We will use the following notations for pure subgraphs of F:

- F^{max} denotes a pure subgraph that is maximal for inclusion;
- F^+ is the F^{max} with all positive predicate nodes in F;
- F^- is the F^{max} with all negative predicate nodes in F;
- F^{Max} denotes a F^{max} of maximal cardinality.

Moreover, a homomorphism from a pure subgraph of F to G has to be "compatible" with a homomorphism from F to a total completion of G. Hence, the following definition:

[3]Note that this subgraph does not necessarily correspond to a set of literals because some term nodes may be isolated.

Definition 3 (Border, Compatible homomorphism) *Let F and G be two PGs and F' be a pure subgraph of F. The predicate nodes of $F \setminus F'$ are called* border predicate nodes *of F' w.r.t. F. A homomorphism h from F' to G is said to be* compatible *w.r.t. F if, for each border predicate node inducing the literal $\sim p(t_1, \ldots, t_k)$, the opposite literal $\overline{\sim p}(h(t_1), \ldots, h(t_k))$ is* not *in G.*[4]

Theorem 2 *[LM07] If $G \vdash F$ then, for each pure subgraph F' of F, there is a compatible homomorphism from F' to G w.r.t. F.*

The following filtering step can thus be performed before the recursive algorithm:

1. select some F^{max};
2. if there is no compatible homomorphism from F^{max} to G then return *false*.

3 Experimental methodology

Due to the lack of benchmarks or real-world data available for the studied problem, we built a random generator of polarized graphs. The chosen parameters are as follows:

- the number of *term* nodes (i.e. the number of terms in the associated formula)[5];
- the number of distinct *predicates*;
- the *arity* of these predicates (set at 2 in the following experiments);
- the *density* per predicate, which is, for each predicate p, the ratio of the number of literals with predicate p in the graph to the number of literals with predicate p in a total completion of this graph w.r.t. $\{p\}$.
- the *percentage of negation* per predicate, which is, for each predicate p, the percentage of *negative* literals with predicate p among all literals with predicate p in the graph.

[4]To ensure that a compatible homomorphism from F' to G can be extended to a homomorphism from F to a total completion of G, the following condition should also be satisfied: for each pair of opposite border predicate nodes respectively on $(c_1, \ldots, c_k)$ and $(d_1, \ldots, d_k)$, $(h(c_1), \ldots, h(c_k)) \neq (h(d_1), \ldots, h(d_k))$. However, this condition is necessarily satisfied if F' is a pure subgraph that is maximal for inclusion, thus we omit it in this paper.

[5]We do not generate constants; indeed, constants tend to make the problem easier to solve because there are fewer potential homomorphisms; moreover, this parameter does not influence the studied heuristics.

An instance of DEDUCTION is obtained by generating a pair (F, G) of polarized graphs. In our first experiments, we only built connected graphs (otherwise, the generated instance can be decomposed into several instances of DEDUCTION). However, we observed that connectivity had no notable influence on the results, thus we just build random graphs without isolated nodes (i.e. each generated graph corresponds to a formula in $FOL\{\exists, \wedge, \neg_a\}$). In this paper, we chose the same number of term nodes for both graphs. The difficulty of the problem led us to restrict this number to between 5 and 8 term nodes. Beyond 9 term nodes, the running time was bigger and bigger. The program is written in Java. The experiments were performed on a Sun fire X4100 Server AMD Opteron 252, equipped with a 2.6 GHz Dual-Core CPU and 4G of RAM, under Linux.

In the sequel we adopt the following notations: *nbT* represents the number of term nodes, *nbPred* the number of distinct predicates, *SD* (resp. *TD*) the Source (resp. Target) graph Density per predicate and *neg* the percentage of negation per predicate.

In order to discriminate between different techniques on random data, it seems preferable to run them on "difficult" instances of the problem (i.e. pairs of graphs). We thus ran the `recCheck` algorithm while varying a given parameter, in order to characterize the influence of this parameter on the difficulty. The difficulty was measured in three different ways: the running time, the size of the search tree and the number of homomorphism checks (when more than one homomorphism check can be done at each node of the tree). For each value of the varying parameter, we considered 1000 instances and computed the mean search cost of the results on these instances (with a timeout set at 10 minutes).

We first studied the influence of the respective densities of both graphs with a single predicate and *50%* of negation: see Figure 3, which shows the curves for the most difficult values of *SD*.

One can expect that increasing the number of predicates occurring in graphs increases the difficulty, in terms of running time as well as the size of the searched space. Indeed, the number of completions increases exponentially (there are $(2^{n_G^2})^{nbPred}$ total completions for *nbPred* predicates). These intuitions are only partially validated by the experiments: see Table 1, which shows, for each number of predicates, the density values at the difficulty peak (the first row corresponds to the curve with *SD*=0.24 in Figure 3). We observe that the difficulty increases up to a certain number of predicates (3 here, with a CPU time of 4912 and a Tree size of 72549) and beyond this value, it continuously decreases. Moreover, the higher the number of predicates, the lower *SD* which entailed the greatest difficulty peak, and the higher the difference between *TD* and *SD* at the difficulty peak.

Concerning the negation percentage, we checked that the maximal difficulty is obtained when there are as many negative predicate nodes as positive predicate

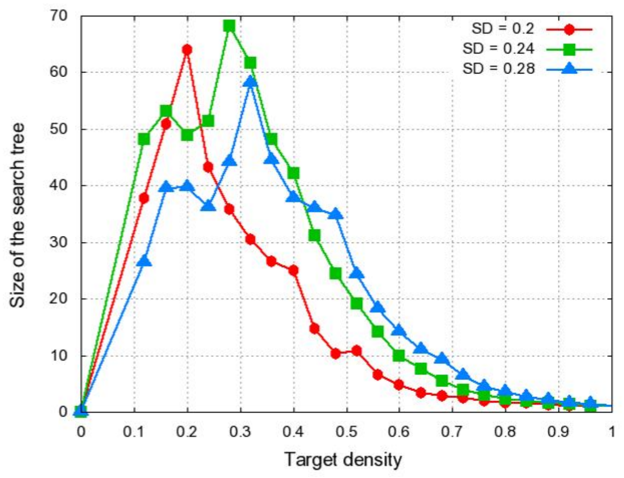


Figure 3: Influence of densities : *nbT*=5, *nbPred*=1, *neg*=50%.

NbPred	SD	TD	CPU time (ms)	Tree size
1	0.24	0.28	20	68
2	0.08	0.16	3155	53225
3	0.08	0.4	4912	72549
4	0.08	0.68	3101	44089
5	0.08	0.76	683	8483

Table 1: Influence of the number of predicates : *nbT*=5, *neg*=50%.

nodes. In the sequel we only show the CPU time when the three difficulty measures are correlated.

4 Refinement of the algorithm

In this section, we analyze three refinements of Algorithm 1, which concern the following aspects:

1. the choice of the next literal to add;
2. the choice of the child to explore first;
3. dynamic filtering at each node of the search tree.

1. Since the search space is explored in a depth-first manner, the choice of the next literal to add, i.e. $\sim r(t_1, \ldots, t_n)$ in Algorithm 1 (Point 1), is crucial. A brutal technique consists of choosing r and $t_1, \ldots, t_n$ randomly. Our proposal is to guide this choice by a compatible homomorphism, say h, from a F^{max} to the current G. More precisely, the border predicate nodes $\sim r(e_1, \ldots e_n)$ w.r.t. this F^{max} can be divided into two categories. In the first category, we have the border nodes s.t. $\sim r(h(e_1) \ldots h(e_n)) \in G$, which can be used to *extend* h; if all border nodes are in this category, h can be extended to a homomorphism from F to G. The choice of the literal to add is based on a node $\sim r(e_1, \ldots e_n)$ in the second category: r is its predicate symbol and $t_1, \ldots, t_n = h(e_1) \ldots h(e_n)$ are its neighbors (note that neither $\sim r(h(e_1) \ldots h(e_n))$ nor $\overline{\sim r}(h(e_1) \ldots h(e_n))$ is in G since $\sim r(e_1, \ldots e_n)$ is in the second category and h is compatible). Intuitively, the idea is to give priority to predicate nodes potentially able to transform this compatible homomorphism into a homomorphism from F to a (partial) completion of G, say G'. If so, all completions including G' are avoided.

Figure 4 shows the results obtained with the following choices:

- *random choice*;
- *random choice + filter*: random choice and F^+ as filter (i.e. at each recCheck step a compatible homomorphism from F^+ to G is looked for: if none exists, the false value is returned);
- *guided choice*: F^+ used both as a filter and as a guide.

Note that the guided choice comes with an implicit filter: indeed, when a compatible homomorphism from F^+ to a partial completion of G (say G') is sought, the false value is returned if none exists (since $G' \not\vDash F$). In order to only discriminate choice heuristics, we also considered a random choice with a filter.

As expected, the guided choice is always much better than the random choice (with or without filter): on the guided choice peaks (*TD*=0.15 and *TD*=0.2), it is almost 11 and 8 times better than the random choice with filter. The greatest difference is for *TD*=0.25 with the guided choice almost 116 times better than the random choice with filter.

2. Experiments have shown that the order in which the children of a node, i.e. G' and G'' in Algorithm 1 (Point 2), are explored is important. Assume that Point 1 in Algorithm 1 relies on a guiding subgraph. Consider Figure 5, where F^+ is the guiding subgraph (hence the border is composed of negative predicate nodes): we see that it is always better to explore G' before G''. If we take F^- as the guiding subgraph, then the inverse order is better. More generally, let $\sim r(e_1 \ldots e_n)$

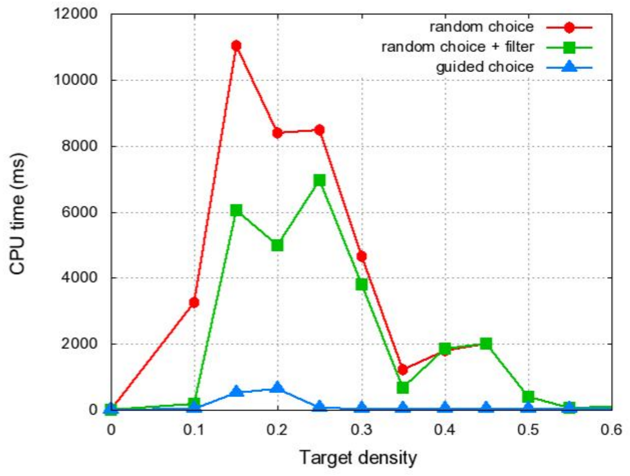


Figure 4: Influence of the completion choice : *nbT*=7, *nbPred*=1, *SD*=0.14, *neg*=50%.

be the border node that defines the literal to add. Let us call h*-extension* (resp. h*-contradiction*) the graph built from G by adding $\sim r(h(e_1)\dots h(e_n))$ (resp. $\overline{\sim r}(h(e_1)\dots h(e_n))$). See Example 2. It is better to first explore the child corresponding to the h*-contradiction*. Intuitively, by contradicting the compatible homomorphism found, this gives priority to failure detection.

Example 2 *See Figure 1.* $F^+ = \{+p(x), +s(x,y)\}$*. Let* F^+ *be the guiding subgraph. The only border node of* F^+ *w.r.t.* F *is* $-p(y)$*.* $h = \{(x,a),(y,b)\}$ *is the only compatible homomorphism from* F^+ *to* G*. The* h*-extension (resp.* h*-contradiction) is obtained by adding* $+p(b)$ *(resp.* $-p(b)$*).*

3. The last improvement consists of performing dynamic filtering at each node of the search tree. Once again, the aim is to detect a failure sooner. More precisely, we consider a set of F^{max} and check if there is a compatible homomorphism from each element in this set to the newly generated graph. Figure 6 shows the results obtained with the following configurations:

- *Max*: F^{Max} as guide and no filter;
- *Max*-$\overline{Max}$: F^{Max} as guide and $F^{\overline{Max}}$ (the subgraph on the predicate nodes in $F \setminus F^{Max}$) as filter;
- *Max-all*: F^{Max} as guide and all other F^{max} as filters.

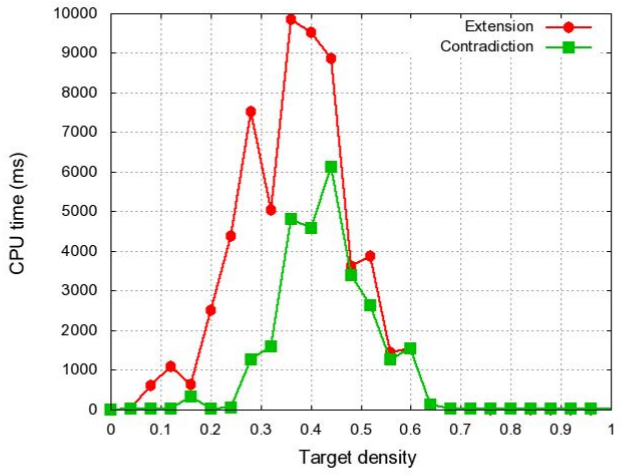


Figure 5: Influence of the exploration order : *nbT=7*, *nbPred=3*, *SD=0.06*, *neg=50%*.

Configuration	CPU time (ms)	Tree size	Hom check
Max	3939	2423	3675
Max-$\overline{Max}$	3868	2394	3858
Max-all	3570	1088	6338

Table 2: Influence of the dynamic filtering: *nbT*=8, *nbPred*=3, *SD*=0.03, *TD*=0.16, *neg*=50%.

Unsurprisingly, the stronger the dynamic filtering, the smaller the size of the search tree (Figure 6). The CPU time is almost the same for all configurations (and all TD values) though *Max-all* checks much more homomorphisms than the others (see Table 2 at the difficulty peak). Since our current algorithm for homomorphism checking can be improved, these results show that *Max-all* is the best choice.

The algorithm finally obtained is shown in Algorithm 2 and subalgorithms 3 and 4. It is initially called with $(G, \emptyset)$. The second parameter is used to memorize the compatible homomorphism found for the father of the current node, in the case where this node is an h-extension of its father (see G'' in the algorithm); otherwise, the compatible homomorphism for its father has been contradicted and a new one has to be computed, which is done in the chooseCompletionLiteral subalgorithm (Algorithm 3).

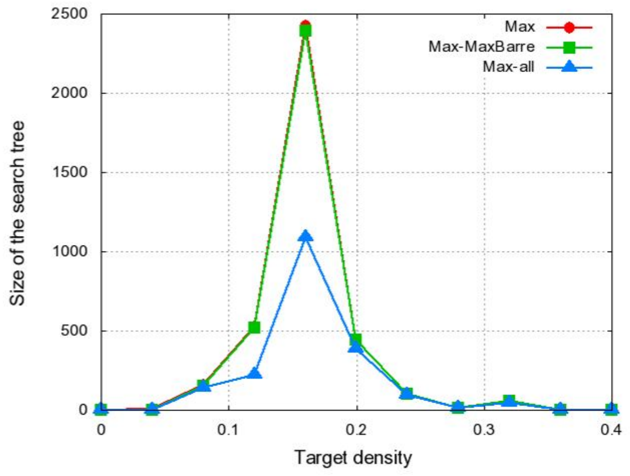


Figure 6: Dynamic filtering : *nbT*=8, *nbPred*=3, *SD*=0.03, *neg*=50%.

5 Comparison to logical approaches

We first tried to translate the problem into UNSAT (propositional unsatisfiability) in order to use SAT solvers. we rely on a translation which is not "straightforward" [BLM08]. The number of obtained clauses is equal to the number of compatible homomorphisms from a special subgraph of F (built with predicates outside the completion vocabulary), say F', to G. The size of a clause is bounded by $|F\ F'|$. Note that the size of the obtained propositional formula is exponential in the size of the initial formulas. Some experiments with our random generator have shown that the cost of the translation step is prohibitive. We thus turned our attention to general first-order theorem provers. For the moment, we have considered the free tools *Prover9* and *Mace4*. Prover9 is based on the resolution method and Mace4 enumerates models by domains of increasing size. These tools are complementary: Prover9 looks for a proof and Mace4 looks for a model that is a counterexample.

To compare both algorithms, we ran `recCheckPlus` and *Prover9-Mace4* on the same instances: we use the combination *Prover9-Mace4* because Prover9 alone was too slow (e.g. for the instances of Figure 7 there are 871 timeouts if we consider only Prover9, with timeout set to 10mn), that is why we added Mace4 which concludes faster in timeout cases. When we consider the combination Prover9-Mace4 there is no timeout for the instances of Figure 7, thus the mean CPU time decreases significantly. In other side Mace4 alone only stop when it find a counter-model (it does not stop when the formula $F \to G$ is valid). For each instance, we ran both Prover9 and Mace4 (there are used as black-boxes: we launch the program and take the time returned by the program itself) and we stopped when the fastest concluded. The first experiments show that `recCheckPlus` is faster than the combination of Prover9 and Mace4: see for example Figure 7, where `recCheckPlus` is 10 times better for *TD*=0.4. These are only preliminary results, further experiments are needed to refine the comparison.

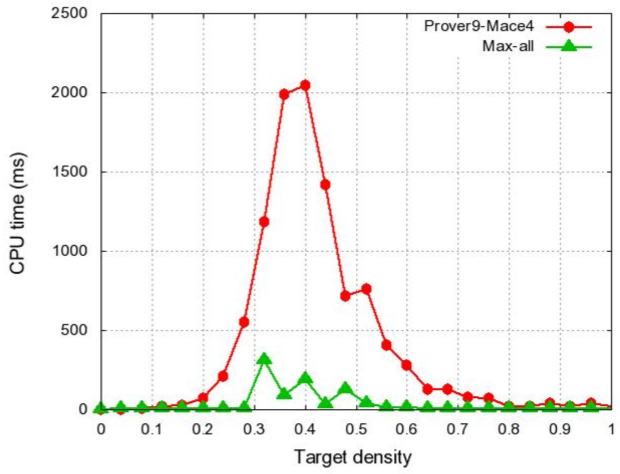


Figure 7: Comparison of recCheckPlus and Prover9-Mace4 : *nbT*=7, *nbPred*=3, *SD*=0.06, *neg*=50%.

One may argue that, even if *Prover9* is a well-known theorem prover, it is not the fastest tool today. On the other hand, these first results are very encouraging since our central homomorphism operation could be optimized. For the time being, it is a straightforward translation of the forward-checking algorithm known for the constraint satisfaction problem (CSP) and we have checked that our homomorphism check is much slower than the best known CSP solvers.

6 Conclusion

Our experiments show that the problem is really complex in practice in the difficult area. This may be an argument in favor of restricting conjunctive queries with negation (to pure queries for instance) or using alternative kinds of negation. Closed-world negation is often used. However, even in this case, good algorithms are required for comparing queries (cf. the query containment problem, which is the basis of many mechanisms in databases) and we again find the deduction problem with classical negation.

Ontologies play a central role in knowledge bases, and this role is increasing in databases. A lightweight ontology, in the form of a partial order, or more generally a preorder, on predicates (i.e. on the concepts and predicates of the ontology) can be taken into account without increasing complexity. Note that, in this case, we obtain exactly the deduction problem in a fragment of conceptual graphs [Ker01][ML07]. Homomorphism is extended in a straightforward way to take the partial order into account. The heuristics studied here still work with an extension of opposite literals: $+r(t_1, \ldots, t_n)$ and $-s(u_1, \ldots, u_n)$ are opposite if $r \geq s$ (then, a pure subgraph is defined as previously with this extended definition). How this work can be extended to more complex ontologies is an open issue. On the experimental side, further work includes precise comparison with techniques used by logical solvers.

References

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases: The Logical Level*. Addison-Wesley, 1995.

[ASU79] A. V. Aho, Y. Sagiv, and J. D. Ullman. Equivalences among relational expressions. *SIAM J. Comput.*, 8(2):218–246, 1979.

[BLM08] K. Ben Mohamed, M. Leclère, and M.-L. Mugnier. De la déduction dans le fragment $\{\exists, \wedge, \neg_a\}$ de la logique du premier ordre à sat. *Journées Nationales de lIA Fondamentale*, oct 2008.

[CM77] A.K. Chandra and P.M. Merlin. Optimal implementation of conjunctive queries in relational databases. In *9th ACM Symposium on Theory of Computing*, pages 77–90, 1977.

[FNTU07] C. Farré, W. Nutt, E. Teniente, and T. Urpí. Containment of conjunctive queries over databases with null values. In *ICDT 2007*, volume 4353 of *LNCS*, pages 389–403. Springer, 2007.

[Hal01] A. Y. Halevy. Answering queries using views: A survey. *VLDB Journal*, 10(4):270–294, 2001.

[Ker01] G. Kerdiles. *Saying it with Pictures: a logical landscape of conceptual graphs*. PhD thesis, Univ. Montpellier II / Amsterdam, Nov. 2001.

[LM07] M. Leclère and M.-L. Mugnier. Some Algorithmic Improvments for the Containment Problem of Conjunctive Queries with Negation. In

Proc. of ICDT'07, volume 4353 of *LNCS*, pages 401–418. Springer, 2007.

[McC03a] William McCune. Mace4 reference manual and guide. *CoRR*, cs.SC/0310055, 2003.

[McC03b] William McCune. Otter 3.3 reference manual. *CoRR*, cs.SC/0310056, 2003.

[ML07] M.-L. Mugnier and M. Leclère. On querying simple conceptual graphs with negation. *Data Knowl. Eng.*, 60(3):468–493, 2007.

[MR94] S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *J. Log. Program.*, 19/20:629–679, 1994.

[RvBW06] F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.

[Ull89] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume II*. Computer Science Press, 1989.

[Ull97] Jeffrey D. Ullman. Information Integration Using Logical Views. In *Proc. of ICDT'97*, volume 1186 of *LNCS*, pages 19–40. Springer, 1997.

[WL03] F. Wei and G. Lausen. Containment of Conjunctive Queries with Safe Negation. In *International Conference on Database Theory (ICDT)*, 2003.

Algorithm 2: **recCheckPlus**(G, h)

```
Input: a consistent PG $G$ and a compatible homomorphism $h$ from the
       guiding subgraph to the father of $G$ (empty for the root)
Data: $F$, $\mathcal{V}$
Result: true if $G \vdash F$, false otherwise
begin
    if there is a homomorphism from $F$ to $G$ then return true ;
    if $G$ is complete w.r.t. $\mathcal{V}$ then return false ;
(3) if dynamicFiltering(G) = failure then return false ;
(1) $l, h \leftarrow$ chooseCompletionLiteral($G, h$) ;
    if $l = failure$ then return false ;
    Let $G'$ be obtained from $G$ by adding $\bar{l}$ ;
    Let $G''$ be obtained from $G$ by adding $l$ ;
(2) return recCheckPlus($G'$, $\emptyset$) AND recCheckPlus($G''$, h) ;
end
```

Algorithm 3: chooseCompletionLiteral(G, h)

```
Input: a consistent PG $G$ and a compatible homomorphism $h$ from the
       guiding subgraph to the father of $G$ (empty for the root)
Data: $F$ and a pure subgraph $F'$ of $F$ maximal for the inclusion (i.e. a
      $F^{max}$ or a $F^{Max}$)
Result: a completion literal if there is one, and the associated guiding
        compatible homomorphism, otherwise failure
begin
    if $h = \emptyset$ then $h \leftarrow$ findCompatibleHomomorphism($F, F', G$) ;
    if $h$= failure then return failure ;
    Choose $\sim r(t_1, \ldots, t_n) \in F \setminus F'$ s.t. neither $\sim r(h(t_1), \ldots, h(t_n))$ nor
    $\overline{\sim r}(h(t_1), \ldots, h(t_n))$ is in $G$ ;
    return $\sim r(h(t_1), \ldots, h(t_n))$ ;
end
```

```
Algorithm 4: dynamicFiltering($G$)
  Input: a consistent PG $G$
  Data: $F$ and a set of pairs $L = \{(F_1^{''}, h_1^{''}), \ldots, (F_n^{''}, h_n^{''})\}$ where $F_i^{''}$ is a
        pure subgraph used to filter and $h_i^{''}$ is a compatible homomorphism
        from $F_i^{''}$ to the father of $G$ (empty if $G$ is the root)
  Result: success or failure
  begin
      foreach $(F'', h'') \in L$ do
          if $h''$ is contradicted by the last relation node added to $G$ then
              $h'' \leftarrow$ findCompatibleHomomorphism($F, F'', G$) ;
              if $h''$ = failure then return failure ;
      return success ;
  end
```