Abductive Reasoning in Three-valued Logic for Knowledge Bases

Philippe Mathieu and Jean-Paul Delahaye

Abstract

In this paper we study the case of monotonic rule-based systems which use a forward chaining algorithm to make deductions and a query algorithm to compute queries to be asked to the user in order to be closer to the expected goal (as Guru [?], IS [?] or Nexpert [?]). Unfortunately in many systems the query algorithm is not consistent with the deduction algorithm. Queries asked by these systems are often useless. We show that computing a query for those systems is equivalent to making abductive reasoning in three-valued logic and we propose a formalization of useful queries for expert systems. We give two definitions of intelligent queries and two levels to compute them: The logical level which aims at computing a list of intelligent queries, and the heuristic level which aims at choosing which query of that list it will ask the user. These are three examples:

If b then a	If b then a	If b and c then a
If c then b	If c then a	If d then a
If d then b	If d then b	If e then b
If e then not c	If b then e	not c
If f then d	If b then not e	

If we run a mixed chaining on the first example with the goal a then almost all the commercial expert systems will ask the query "Is e true?". Unfortunately, e does not help the system to be closer to the expected goal because e gives not c and then nothing is deduced. In this case, there is just one useful query which is "Is f true?".

By the same way we can see that the only useful query on the second example is c (d is useless since "yes" to this query leads to a contradiction) and on the third example only d is useful (e is useless).

1 Formalism

1.1 External representation

Definitions.

A literal is an atomic formula or a negated atomic formula. We denote by Lit the finite or infinite set of literals. A rule is a propositional formula of the form $L_1, \ldots, L_n \rightarrow L$ with $n > 0$ and $L, L_1, \ldots, L_n \in Lit$. A **knowledge base** is a set of rules. A **working memory** (WM) is a set of literals.

Let p and q be two literals, we note $p \prec q$ and we say that p strictly depends on q iff there is a rule with q in its premises and p as conclusion (for example $q \rightarrow p$). We note $<$ the transitive closure of $\prec$. $p < q$ means that p depends on q. We say that a knowledge base is **without cycle** iff there is no literal p for which $p < p$. We say that a literal is a **basic literal** iff there is no literal q for which $p < q$.

To simplify the problem in this paper, we only work on knowledge bases without cycles. The formalism used here is an elementary one, it can be easily to extended. Allowing only conjunctions in rule conditions and only one literal in rule conclusions is not restrictive. With this formalism we can easily represent the rules usually used in many expert systems. For example, conjunction in rule conclusion or the connectors 'and', 'or', 'not' in rule condition can be easily represented by transforming rule methods like distributivity laws, De Morgan's laws and double negation laws (see [?] for example).

The working memory (WM) also called extensional knowledge base, contains the literals known at the beginning of a session. Forward chaining will complete this memory by adding new facts obtained by deduction.

1.2 Internal representation

The system used here can ask queries to the user. Of course the user can answer *"I don't know"* to a query as in many expert systems. We choose to represent each literal of the working memory by a triplet (Name,Meta-value,Value). Name is the name of the atom used, its Meta-value can be Known if the atom has a value (Value at True or False) or Undefined if the user has answered *"I don't know"* to the query (Value is then empty). An Unknown atom is not explicitly present in the working memory, it is its absence which feels like the Unknown value and meta-value.

First of all, an atom is Unknown. It can become Known (True or False) if the user tells it to the system or if the system infers it. It can also become Undefined if the user answers *"I don't know"* to a query on that atom (That meta-value helps the system to ask just one time each query).

The working memory is according to the **not–contradiction principle:** Two opposite literals cannot both be in WM. If it is the case, a contradiction is detected. It is not according to the **excluded middle principle:** An atom is not always True or False. It can be Unknown. This is the reason why we will work in three–valued logic.

2 Deduction system

The general system used in this paper is the mixed chaining, often used in expert systems because it allows to center the search on a given goal and allows the system to ask queries to the user during the deduction [?]. It is the relevance of these queries which leads the user to qualify a given expert system by *'intelligent'*. Thus it is primordial to compute correctly these queries.

Mixed chaining uses a forward chaining algorithm to deduce new facts and a backward chaining algorithm to compute queries to ask to the user. To have a pleasant user interface **we consider that the expected goal is an atom** and the system tries to deduce a value for this atom. At the end of the session the expected atom is True, False or Unknown if it has not been proved neither True nor False. Of course, as we ask queries to the user, we suppose that some atoms are askable and some are not. These considerations are very classical and have been used many times (for example in [?], [?], [?], [?] or recently in [?]). To simplify the problem we will assume that only basic facts are askable in all our examples.

We can specify the working of this mixed chaining in the following algorithm:

Mixed chaining algorithm (Ato)

```
Saturate the knowledge base with forward chaining (section 3).

While Ato is unknown (neither true nor false) and
        no contradiction has been detected and
        a query can be asked to the user.
                Ask this query.
                add the answer to the working memory.
                saturate with forward chaining.
End_while

Case    - Ato is true in the working memory: write 'Ato true'
        - Ato is false in the working memory: write 'Ato false'
        - else write 'I cannot deduce any value on Ato'
end_case
```

Remark that in this algorithm, the computation of queries is always made after a step of saturation by forward chaining.

3 How Forward Chaining Works ?

Our forward chaining infers literals (classical approach). A literal L (either positive or negative) is added to the working memory iff there is a rule which have L as conclusion and all its premises in the working memory.

Forward chaining algorithm by saturation: Sat.

```
While there exist rules with their conclusions not in WM and
        all the literals of their condition part present in WM
                Search for the first of these rules.
                If the conclusion of this rule is consistent
                with all the literals in WM then
                        add the conclusion in WM.
                else
                        return 'inconsistent knowledge base'
                        Stop
                End_If
End_While
```

We can see that an atom is not always True or False. It can be Unknown. We have shown in another paper [?] that this forward chaining is sound and complete with respect to the three-valued logic defined by these connectives:

$\wedge$	T	F	U
T	T	F	U
F	F	F	F
U	U	F	U

$\vee$	T	F	U
T	T	T	T
F	T	F	U
U	T	U	U

$\rightarrow$	T	F	U
T	T	F	F
F	T	T	T
U	T	T	T

$\neg$	
T	F
F	T
U	U

The logical meaning of the implication we consider is: $A \rightarrow B$ is false when A is true and B is not, $A \rightarrow B$ is true in the other cases (as in two–valued logic). With this logic, the set of three–valued models is stable by intersection and we will see that it is very useful to characterize the set of literals computed by forward chaining. More details and justifications for this connective can be found in [?][?].

Let us note Kb a knowledge base, $Cons_B(Kb)$ (resp. $Cons_T$) the two–valued (resp. three–valued) consequence literals of Kb and $Sat(Kb)$ the output of the saturating operation by forward chaining on Kb.

Proposition 3.1. [?]

- For a set of rules without negation (Horn clauses) the set of two–valued models is stable for intersection. Thus it has a minimal two–valued model denoted $mm_B(Kb)$. We obtain also the equality

$$Cons_B(Kb) = mm_B(Kb) = Sat(Kb) \cap Ato(Kb)$$

- For a set of rules with negations, the set of three–valued models is stable for intersection. Thus it has a minimal three–valued model denoted $mm_T(Kb)$. We obtain also the equality

$$Cons_T(Kb) = mm_T(Kb) = Sat(Kb) \cap Lit(Kb)$$

We can see that forward chaining computes a minimal three-valued model (and not a two–valued one) for knowledge bases with negations. Conversely to the two–valued logic we can easily compute in three–valued logic with polynomial time algorithms as forward chaining. We have proposed in other papers a method to solve the two–valued incompleteness of forward chaining (w.r.t. classical two–valued logic) for knowledge bases with negations [?][?][?], it is not the aim of this paper.

4 Semantics of useful queries

To compute a useful query it is obvious that the query algorithm must be consistent with the deduction algorithm. In our case, it must be consistent with the forward chaining. **It is unfortunately not the case for all the commercial expert systems tested** (test the examples of the abstract with yours !).

Since we work on literals, the query algorithm will compute a literal. To have a pleasant user interface we consider that **we will only ask to the user the value of the atom used in the literal computed** by the query algorithm (i.e. we ask "is a True ?" and not "is $\neg a$ True ?").

Of course the system must deduce the expected goal in a minimum number of queries. First of all we must define what is a useful query. To do this we introduce two notions: The relevant queries and the intelligent queries.

If a formula F has for three-valued consequence a formula G we will note $F \models_T G$.

Definition 4.1. [?] Let Kb be a knowledge base, L a conjunction of basic unknown literals and x a literal, we say that L is an **intelligent query list** w.r.t. x iff:

- $Kb \cup L$ is consistent.
- $Kb \cup L \models_T x \vee \neg x$ ($L \rightarrow x$ or $L \rightarrow \neg x$ is a three–valued consequence of Kb)
- There is no intelligent query list L' w.r.t. x such that L' subsumes L.

As we have shown that forward chaining computes three-valued consequences literals we could give a more pragmatic definition: L is an intelligent query list w.r.t. x iff $x \in Sat(Kb \cup L)$ and there is no subset L' of L for which we have $x \in Sat(Kb \cup L')$.

Definition 4.2. [?] Let x and q be two literals, q is an **intelligent query** w.r.t. x iff there exists an intelligent query list w.r.t. x which contains q.

Unfortunately the computation of intelligent queries is an algorithmically complex problem (In section 7 whe show that it is an NP–complete problem). This is why we introduce a weaker definition.

Definition 4.3. [?] Let Kb be a knowledge base, L a conjunction of basic unknown literals and x a literal, we say that L is a **relevant query list** w.r.t. x iff there exist a subset R of Kb for which:

- $R \cup L$ is consistent.
- $R \cup L \models_T x \vee \neg x$ ($L \rightarrow x$ or $L \rightarrow \neg x$ is a three-valued consequence of R)
- There is no relevant query list L' w.r.t. x such that L' subsumes L.

By the same way as for intelligent queries we could give a more pragmatic definition: L is a relevant query list w.r.t. x iff there exists a subset R of Kb for which we have $x \in Sat(R \cup L)$ and there is no subset L' of L for which we have $x \in Sat(R \cup L')$.

Definition 4.4. [?] Let x and q be two literals, q is a **relevant query** w.r.t. x iff there exists a relevant query list w.r.t. x which contains q.

These definitions are in fact the 3–valued version of similar 2–valued notions of [?] called conditional answers or called minimal supports in [?]. We can see that computing a query for expert systems is equivalent to make abductive reasoning in three–valued logic (with our connectives). Thus we can see the computation of relevant queries as a weak abductive strategy and the computation of intelligent queries as a strong abductive strategy.

Intuitively, when we compute a relevant query we just work on a part of the knowledge base. This subset R of Kb could be called a **deductive cone** associated to the expected goal.

Of course, an intelligent query is a relevant one. To prove it, we just have to take $R = Kb$. Conversely it is not true that a relevant query is an intelligent one as we can see on the example 4.5.

Example 4.5.

				WM=$\emptyset$ goal: a
r1	b,c	$\rightarrow$	a	b is a relevant query (consider R={r1}) but not
r2	c	$\rightarrow$	a	an intelligent one. c is an intelligent query (thus relevant).

Example 4.6.

				WM=$\emptyset$ goal: a
r1	b	$\rightarrow$	a	b is a relevant query (consider R={r1}) but not
r2	c	$\rightarrow$	a	an intelligent one because the "yes" answer leads
r3	b	$\rightarrow$	x	to a contradiction. c is an intelligent query
r4	b	$\rightarrow$	$\neg$x	(thus relevant).

Of course, computing an intelligent query is more difficult than computing a relevant query. This is the reason why we propose the two definitions. To have a useful query we must compute a relevant or intelligent query and ask the value of this query to the user. This is what we call **the logical level**. But we have often many relevant or intelligent query lists, thus we must choose which list we will use, and in this one, which literal we will ask to the user. This is what we call **the heuristic level**.

5 The logical level

The aim of the logical level is to compute a list of literals (relevant or intelligent query list) for which we could deduce the expected goal with a forward chaining by adding the query list to the working memory.

Note that most of expert systems do not satisfy to this logical level, as can be seen in the examples proposed in the abstract.

The first idea we will explore is to use a backward chaining with backtracking in this way:

Naive_query_algorithm(Ato). Consider the expected goal as initial goal. If there exists a rule (choice point) with this atom in conclusion (either positively or negatively) then take one of the atoms used in the condition part of this rule (choice point), and consider it as the new goal.

If we obtain an atom which does not belong to WM and which is askable, ask the user about it, else go back to the last choice point. If all the choice points have been studied, the goal cannot be deduced.

Example 5.1.

r1	c	→	¬a	WM=∅ goal: a
r2	¬e	→	a	a depends on atoms c and e, c depends on
r3	¬d	→	c	the atom d and e depends on f. The atoms
r4	f	→	¬e	d and f are unknown and are basic facts so we ask the user about one of them.

This algorithm simply consists in searching a leaf in the dependence graph of the atoms of the knowledge base. It is correct for knowledge bases without negations (That is certainly the reason why it is often used) but unfortunately it is not only inefficient but also incorrect in the case of knowledge bases with negations.

5.1 Working on literals

This naive algorithm considers all the rules which have the expected atom in their conclusion (either in positive or negative form) (The idea is: if the atom appears in a rule conclusion then this rule will give interesting queries about it). Unfortunately if we apply this kind of reasoning on a knowledge base with negations we obtain useless queries as in the following example:

Example 5.2.

r1	¬b	→	a	WM=∅ goal: a
r2	c	→	b	It is obvious that the only useful query is e.
r3	d	→	¬c	The query d, found by the naive algorithm is
r4	e	→	¬b	neither intelligent nor relevant.

By definition, only the forward chaining can add literals to the working memory. The forward chaining works on literals thus it is obvious that the computation of a query cannot be made by chaining from atoms to atoms but from literals to literals. The initial goal must then be a literal. We will call that sort of algorithm a single tentative algorithm [?].

Definition 5.3. We call a **single tentative algorithm** each algorithm which works in the following way:
The initial goal is a literal and the query algorithm tries to compute a query by chaining from literals to literals. If it is not able to find a query then the goal is considered not deducible (it does not mean that this negation is True but only that this literal cannot be deduced by a query).

This sort of algorithm is in fact very natural because it is the direct extension of the algorithms used for knowledge bases without negation (for knowledge bases without negation the two–valued consequences and the three–valued ones are identical, this is why many algorithms which are incorrect with negations become correct without negation).

When we have defined the mixed-chaining algorithm we have said that it was more useful for the user to have an atom as initial goal. The system must try to deduce the value of this atom. Thus the single tentative algorithm cannot be used alone. Thus we add a new level to this algorithm to allow the user to have an atom as initial goal and to obtain its value. We will call this algorithm a double tentative algorithm [**?**].

Definition 5.4. We call a **double tentative algorithm** each algorithm which works in this way: The initial goal is an atom and the query algorithm tries to deduce the value of this atom. If it is not able to deduce that this atom is True or False then this atom is Unknown.

To make this deduction, the system can use three sorts of methods: Trying to deduce True, and if it cannot, trying to deduce False. Trying to deduce False, and if it cannot, trying to deduce True. The third method consists in trying simultaneously True and False. In that case the queries which are able to deduce True and the queries which are able to deduce False are shuffled. This is the best way to optimize the choice of the query to ask to the user when many queries are possible (it is not very interesting in fact to optimize the truth and only after to optimize the falsehood. The best query must be chosen in the queries which allow to deduce the truth and the falsehood simultaneously).

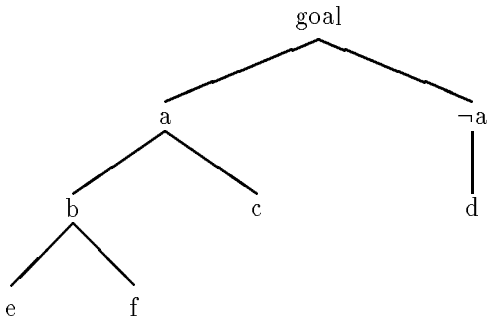
Many methods can be chosen to implement this point according to the heuristics used. We propose two of them: Let us note x the atom asked by the user as initial goal.

- Computing all the queries which allow to deduce x, computing all the queries which allow to deduce $\neg x$ and concatenate them. We have explicitly a higher level up to the *single tentative algorithm*.

- Adding two rules of the form $x \rightarrow goal$ and $\neg x \rightarrow goal$ where *goal* is a fictive goal used as initial goal. Thus we use directly a *single tentative algorithm* on a fictive goal.

These two methods are equivalent. In an implementation it is often more easy to use the first one, but when we work on examples by hand, it is more easy to use the second one which can be easily represented with a dependence tree.

b,c → a
d → ¬a
e → b
f → b



Be careful. If x is the initial goal, even if we are both interested in the rules which have x and $\neg x$ in conclusion, we do not repeat this procedure recursively. Indeed if we do not do so, we risk to compute useless or stupid queries as in the example 5.2. It it certainly because of the confusion between a *single tentative algorithm* and a *double tentative algorithm* that there are many systems which chain from atoms to atoms. The initial goal being (for practical considerations) an atom, the system must be interested in the rules which use this atom in their conclusion. Designers of such systems often think (unfortunately as we can see with the example 5.2) that this process must be recursive. This is not the case ! After the first step we must chain from literals to literals.

5.2 Consistency with the working memory

During the computation of a query we develop a literal as soon as a rule has this literal in conclusion. Unfortunately this literal can already be inconsistent with the working memory. In this case the query obtained will not be useful because forward chaining will later detect an inconsistency and the expected goal will not be deduced.

Example 5.5.

r1	b	→	a	WM={¬b} Goal: a
r2	c	→	b	d is the only intelligent query at this time. The
r3	d	→	a	query c is not relevant because b is already false.

All the problems are not so simple. In many cases we need several queries to deduce the goal expected. If we only chain from literals to literals we may compute a query which will be useless because one of the other queries needed is inconsistent with the working memory.

Example 5.6.

r1	b	→	a	WM={¬d} Goal: a
r2	c, d	→	b	e and f are intelligent queries at this time. c
r3	e, f	→	b	is not relevant because d is already false.

The only way to solve that problem is to develop not only a literal but a list of literals which are able to deduce the goal, and to verify at each step that all the literals of that list are consistent with the working memory. Our algorithm will not compute a query as in the previous algorithms but a list of queries. To be closer to our definitions of intelligent and relevant queries we will develop each list of facts with a breadth first strategy (knowledge bases are usually not very deep) to detect inconsistencies as soon as possible.

5.3 Consistency of the future deduction

An algorithm consistent with the preceding points seems to compute a relevant list of queries but in fact other problems are also present. The consistency of the future deduction is one of them. In fact, the forward chaining can add two opposite literals to the working memory during the same saturating operation as in the following examples. In that case, the query list is useless because it will not deduce the expected goal.

Example 5.7.

r1	b, c	→	a	WM=∅ goal: a
r2	d	→	b	The only intelligent query list is (x,y,z)
r3	¬b	→	c	and not (d,e) which will give a contra-
r4	e	→	¬b	diction on b by forward chaining.(d,e) is
r5	x,y,z	→	a	neither an intelligent query list nor a relevant one.

Example 5.8.

r1	b	→	a	WM=∅ goal: a
r2	c	→	b	The only intelligent query list is (x,y) and
r3	c	→	d	not (c) which will give a contradiction on
r4	c	→	¬d	d. Nevertheless (c) is a relevant query
r5	x, y	→	b	list because with R=r1,r2 our definition is satisfied.

In the example 5.7 the facts who give the contradiction are computed during the computation of the query list. In the example 5.8 the contradiction comes from rules never used during the computation of the query list.

We will then define two levels for verifying the consistency of the future deduction.

Complete verification

The only way to solve a problem like the example 5.8 is to run a fictive forward chaining by considering known all the literals of the computed query list and verifying that the expected goal will be deduced. This forward chaining is like the usual one excepted that it cannot add new facts in WM but in a temporary working memory just used for that verification. The temporary working memory is simpler than WM because it just have to treat only known facts. It does not need meta-values. A fact is then satisfied if it is either present in WM or present in the temporary working memory.

This verification, very expensive in computation time, may be used by the expert during the development of the knowledge base and then inhibited during a normal use to speed up deductions.

This verification is not necessary to compute a relevant query list but it is indispensable to compute an intelligent query list.

Partial verification of the deduction cone

As we have said before, the deductive cone is the set of rules used by the backward chaining algorithm to compute the relevant query list. We propose to verify at each development of a list that the new list is consistent both with its ancestors and itself. Thus we must store at each development all the ancestors of the current list. This verification is able to solve many problems as in the example 5.7. We will see after that this verification is really useful for knowledge bases with symbolic facts because it allows to cut useless branches very quickly. Of course, this verification is necessary for the computation of a relevant query list.

5.4 Computing minimal query lists

When we develop a list of facts with a breadth first strategy by chaining from literals to literals and verifying at each development the consistency of that list with the working memory we are sure to obtain queries which will be useful to obtain the goal by forward chaining. Unfortunately some facts of the query list computed can make the same deduction many times. In that case we do not obtain a minimal query list and thus we are not fulfilling our intentions entirely.

Example 5.9.

r1	b, c	→	a
r2	d	→	b
r3	e	→	d
r4	e	→	c
r5	x	→	e
r6	y	→	e

WM=$\emptyset$ goal: a
The query lists (x,y) and (y,x) are not relevant ones because they are subsumed by other query lists. The only two intelligent query lists on that example are (x) and (y).

To obtain a minimal query list we must verify at each development of a list that there is no literal already developed. To do that we have to verify if there is no literal already present in the ancestors of the list. If a literal is already present in the ancestor list or in the current list we delete it from the current list. This deletion is correct because our knowledge bases are without cycles. In the example 5.9 the literal e is present in the ancestor list. We are sure that all the literals which allow to deduce e are present in the current list because e has been developed before and there is no cycle. Thus e can be deleted from the current list during the development.

6 Resulting algorithms

The first proposed algorithm allows to compute a relevant query list consistent with the definition of the section 4. It takes care of all the points studied in the section 5. The second one computes an intelligent query list. It has been built over the first one.

6.1 The computation of a relevant query

The initial set of literals is built with the expected goal.

While one of the literals of the current list is unknown and not askable and there exists at least a rule with this literal in conclusion. Choose one of these rules (choice point). Replace this literal in the current list by the premises of the rule chosen while deleting all the literals already present in WM and in the current list. If one of the literals is not consistent with WM or is not consistent with the ancestor list, stop this way and go back to the last choice point. If one of the literals is already present in the ancestor list delete that literal from the current list. If there is no rule with this literal as conclusion, go back to the last choice point.

Relevant_query_list algorithm

We call x the atom asked by the user

```
Add two rules of the form x → goal and ¬x → goal
The initial list List and the ancestor list Anc are equal
to (goal)

Computing_relevant_query_list(List,Anc)
   While there exists a literal in List which is not askable
      For all the literals in List
         If there is a rule with that literal as conclusion
            Choose this rule.  (choice point)
            Substitute that literal in List by the premises of
            the rule chosen while deleting literals already
            present in the working memory.

            If one of the literals of List is not consistent
            with WM or is not consistent with Anc
               Go back to the last choice point.
            End_if

            If one of the literals of List is already present
            in Anc or in List.
               Delete this literal
            End_if
         Else
            Go back to the last choice point.
         End_if
      End_for
   End_while

   Let us call L the new obtained list.
   Add L to Anc giving A.
   computing_relevant_query_list(L,A).
   End_computing_relevant_query_list
```

Theorem 6.1. *The relevant_query_list algorithm computes a relevant query list.*

Proof. See [?]. ∎

It is important to note that our forward chaining cannot loop, because if a literal appears in its ancestor list it is deleted. Unfortunately the computed query for knowledge bases with cycles is not correct. This is why we have supposed at the beginning that our knowledge bases were without cycles. It is in fact very easy to verify during a compilation session

that a knowledge base does not contain cycles. We can see in the next example a case of incompleteness for a cyclic knowledge base.

Example 6.2.

r1	b, c	→	a
r2	b	→	b
r3	d	→	b
r4	e	→	c

WM=$\emptyset$ goal: a
There is a cycle on b (b depends on itself). One of the lists computed is (e). Unfortunately this list is not complete. The only intelligent list is of course (d,e)

6.2 The computation of an intelligent query list

To obtain an intelligent query list we must compute all the relevant ones and then verify for all of them that they will not give a contradiction on the full knowledge base with forward chaining (section 5.3). If one of the relevant query lists gives a contradiction we delete it. At the end, we delete also all the query lists included in others to be consistent with the subsumption point of the definition.

Intelligent_query_list algorithm

We call x the atom asked by the user
Add two rules of the form $x \rightarrow goal$ and $\neg x \rightarrow goal$
The initial list $List$ and the ancestor list Anc are equal to ($goal$)

Computing the set E of all the lists L for which
L=computing_relevant_query_list(List,Anc)
and for which $goal \in Sat(Kb \cup L)$
End.

Delete all the subsumed lists from E.

Theorem 6.3. *The intelligent_query_list algorithm computes an intelligent query list.*

Proof. See [?]. ∎

7 Complexity

After having defined these algorithms it is interesting to know if the problem of computing a relevant or intelligent query is a difficult (in the sense

of time complexity) problem. It is a good and a bad thing to see that the relevant query problem is a NP-complete problem. Good because it shows that this problem is not a trivial one but a fundamental one, bad because there is no longer any hope (unless P=NP) to obtain efficient algorithms to solve that problem in general case.

To prove that the relevant query problem is a NP-complete problem we must show first that a non–deterministic Turing machine solves it in polynomial time. In a second part we must show that this problem is a generic one, that means it can solve other NP-complete problems by a polynomial reduction [**?**].

Theorem 7.1. *Computing a relevant query list is a NP-complete problem.*

PROOF. 1. For a given goal it is easy to see that a non-deterministic Turing machine will solve this problem in linear time if there is a solution. The Turing machine will choose the good rule to use at each time, so the problem will be solved in n steps if n is the length of the tallest branch of the knowledge base.

2. To prove that this problem is a generic one, we will show that if we can solve the relevant query problem then we can solve the satisfiability problem by a polynomial transformation. Let us take a set of m clauses for which we want to know if it is satisfiable. Let it be of the form

$$\begin{array}{c} x_{1,1} \vee x_{1,2} \vee \ldots \vee x_{1,n_1} \\ x_{2,1} \vee x_{2,2} \vee \ldots \vee x_{2,n_2} \\ \vdots \\ x_{m,1} \vee x_{m,2} \vee \ldots \vee x_{m,n_m} \end{array}$$

For each clause of the form $x_{i,1} \vee x_{i,2} \vee \ldots \vee x_{i,n_i}$ write n_i rules of the form $x_{i,j} \rightarrow y_i$ for $j = 1 \ldots n_i$. Write then a rule of the form $y_1, \ldots, y_m \rightarrow z$. By construction, any relevant query list is a model of the set of clauses. The size of the associated knowledge base is obviously ($n * m$ rules where n is the average of the premises of each clause) polynomial in the size of the set of clauses.

The relevant query list problem is then a NP–complete problem. ∎

Example 7.2.

a∨b	will be transformed to	a	→	y_1
a∨¬b		b	→	y_1
c∨¬a		a	→	y_2
¬c∨¬a		¬b	→	y_2
		c	→	y_3
		¬a	→	y_3
		¬c	→	y_4
		¬a	→	y_4
		y_1, y_2, y_3, y_4	→	z

With the goal z we cannot compute a list of relevant queries, thus the set of clauses is inconsistent.

Example 7.3.

a∨b	will be transformed to	a	→	y_1
a∨¬c		b	→	y_1
c∨¬b		a	→	y_2
¬c∨¬a		¬c	→	y_2
		c	→	y_3
		¬b	→	y_3
		¬c	→	y_4
		¬a	→	y_4
		y_1, y_2, y_3, y_4	→	z

With the goal z we can compute the relevant query list $(a, \neg b, \neg c)$ which is also a model of the initial set of clauses.

We have seen before that an intelligent query is a relevant one thus computing an intelligent query is also a NP-complete problem.

8 The heuristic level

We are now able to compute a list of relevant or intelligent queries. We must now choose which list we will use if there are many of them and which literal of that list we will use to ask the query. We can have two different approaches. Choosing one list during the backward step or computing all the lists and then choose the list which seems more interesting.

8.1 Partial calculus of query lists

Heuristics proposed in this part do not need to compute all the lists before choosing the best one. Thus they have the advantage of working more quickly. It is also easy for the expert to follow the reasoning by hand (choices are less difficult to understand). Programs are easier to be realized. It will not be the case for the other methods. We consider by convention that we will always ask the query on the first literal of the list.

8.1.1 Priority number on rules

This is the most often used heuristic in expert systems. If many rules have the same conclusion, the expert gives a priority number at each of them. The system will examine first the rule which has the higher (or lower) priority number. This method is often used because it gives to the expert the more easy way to control deductions.

Of course, systems which examine rules in the writing order of the knowledge base or which examine first the rules with the smallest premise use the same kind of heuristic. Other variants often used are also in this part. We will just indicate them. Choosing the rules which use fact more recently obtained, choosing the rules with more conclusions, choosing the rules with more premises.

8.1.2 Rules which have less unknown premises

The heuristics presented before are static. Other variants are theoretically identical excepted that they are dynamic. The choice is not always the same for all the deductions. The leader of these heuristics is the choice of rules which have less unknown premises. It supposes that lesser unknown premises there are in a rule, many chances we have to obtain quickly the expected goal. It often gives good results but it is still an heuristic !

Example 8.1.

				WM={f} Goal: a
r1	b	→	a	Between the lists (c,d) and (e), the list (e)
r2	c,d	→	b	is chosen instead of (c,d) because it contains
r3	e,f	→	b	only one unknown literal and (c,d) contains two.

8.2 Complete calculus of query lists

In section 7, no comparison between the different possible query lists was made. Unfortunately it is often interesting to choose one list instead of another one by following precise ideas. This leads us to propose a method

which gives an order on the different lists, and gives also an order on the different literals of the chosen list.

8.2.1 Choosing the smallest query list

If we suppose that all the facts have the same probability to become True or False, by choosing the smallest list we have more chances to obtain the expected goal. Thus we can ask for example about the first literal of that list.

8.2.2 Choosing the atom more often present in all the query lists

When many lists are possible, an atom can be present in many of them. Thus it gives more information than the others because if it is false it will allow to delete many query lists in one time.

Example 8.2.

`r1`	`b`	$\rightarrow$	`a`	WM=$\emptyset$ Goal: `a`
`r2`	`c`	$\rightarrow$	`a`	The query lists obtained are `(d,x)` and `(e,x)`.
`r3`	`d,x`	$\rightarrow$	`b`	Asking `x` first seems more interesting than ask-
`r4`	`e,x`	$\rightarrow$	`c`	ing `d` or `e`

Of course this heuristic can be used both with the previous one. If an atom is present many times we choose this one else we choose the smallest query list.

8.2.3 Satisfiability coefficients

Until now we assumed that all the askable facts had the same probability to become True or False. But in many cases we have more chances to obtain one answer than another one to the query on a fact.

We have proposed a method in [?] which associates a satisfiability coefficient at each literal, which corresponds to the chance to satisfy this literal with a query on it. These coefficients are given by the expert with the knowledge base. To be consistent with the probability theory our coefficients are reals between 0 and 1 (like a percentage). If x is an atom the sum of the coefficients of x and $\neg x$ must be less than 1. The coefficient of the answer *"I don't know"* is then the complement to 1 of the sum of the coefficients of x and $\neg x$. We compute a value for each query list. We choose the list with the best value and we then choose the literal which has the biggest coefficient.

8.2.4 Cumulative method

As we can see, all these points (the smallest query list, the atom more often present and the satisfiability coefficients) are necessary to have a good heuristic. Thus we propose to generalize all these points in a cumulative method. First of all, we will compute for each literal x a weight which takes care of the length of the minimal list which contains it, the number of lists which contains it and the satisfiability coefficients. Thus we compute a value to the associated atom which takes care of the weight of the literal and its opposite form. In [?] we propose to compute this value in the following way : we note $l(X)$ the length of the shortest list which contains the literal X, $nb(X)$ the number of lists which contain X and $c(X)$ the satisfiability coefficient of X. First we compute for each literal X

$$p(X) = c(X)^{\frac{l(X)}{nb(X)}}$$

We compute then for each atom X

$$n(X) = p(X) + p(\neg X) - p(X).p(\neg X)$$

The query is then asked on the atom X which have the greatest value $n(X)$.

9 Symbolic facts

We will now discuss knowledge bases which use not only boolean facts but also symbolic facts and we will see that the previous algorithms are also necessary for these bases. Symbolic facts are often used in expert systems because they allow the expert to reduce the number of rules of the knowledge base and to obtain not only the answers True or False but also a symbolic answer. Even if this knowledge base does not contain any negation, the computation of relevant or intelligent queries is necessary to obtain a correct computation.

Example 9.1.

r1	b='3'	→	a	WM=∅ Goal: a
r2	x	→	b='1'	The first query computed by many
r3	y	→	b='2'	expert systems is x. Of course the
r4	z	→	b='3'	only interesting one is z. all the systems tested ask them !)

What is the sense of this knowledge base? We will try to come back to a boolean problem which we are now able to solve. First of all we can rewrite all the facts of the form $x =$'v' with a boolean fact of the form x_v, but it is not sufficient. We can see that if b is equal to '1' it cannot be equal

to '2' or '3'. Let us note $Def(x)$ the set of symbolic values that a fact x can take. For each symbolic fact x of the knowledge base and for each couple of values $v1$ and $v2$ of $Def(x)$ we write a rule of the form $x\_v1 \rightarrow \neg x\_v2$.

Example 9.2.

r1	b='3' →	a	will be transformed to	b_3	→	a
r2	x →	b='1'		x	→	b_1
r3	y →	b='2'		y	→	b_2
r4	z →	b='3'		z	→	b_3
				b_1	→	¬ b_2
				b_1	→	¬ b_3
				b_2	→	¬ b_1
				b_2	→	¬ b_3
				b_3	→	¬ b_1
				b_3	→	¬ b_2

On the transformed knowledge base we can see that negations have been introduced. This is the reason why classical expert systems are not able to solve these problems. The computed intelligent query list on this example must be (z) as it is on the transformed knowledge base. We can see that our three–valued logic is also able to give a good semantic to study that sort of knowledge bases.

We have proposed in [?] extensions of the two queries algorithms proposed here which are able to compute relevant and intelligent queries for knowledge bases with symbolic facts, symbolic variables as in the following examples and also meta-values as in [?] or [?].

Example 9.3.

r1	b=c →	a		b<>'1', b<>'2'	→	a
r2	x1 →	b='1'		d	→	b=c
r3	y1 →	b='2'		e	→	c='1'
r4	x2 →	c='2'		f	→	c='2'
r5	y2 →	c='3'		g	→	c='3'

In the first example with the goal `a`, the only intelligent query list is `(y1, x2)`. Neither `x1` nor `y2` should be asked to the user. In the second example with the goal `a`, the only intelligent query list is `(d, g)`. Neither `e` nor `f` should be asked to the user.

10 Conclusion

First of all we have presented a particular three-valued logic which is able to characterize exactly the set of literals computed by forward chaining algorithms. Then we have shown that actual expert systems based on mixed chaining ask queries which are not consistent with the deduction algorithm used. We have defined two sorts of queries based on our logic and two levels of computation to compute them. We have also proposed two algorithms to compute these queries. We have shown that the problem of computing a relevant query list for knowledge bases with negations is not a trivial problem. It is in fact in the class of NP-complete problems. At the end, we have extended this formalism to symbolic facts and meta-values. All these query algorithms have been implemented in the BIVOUAC system developed at the LIFL (UA CNRS 369) which contains logical tools for knowledge bases with negations.

All these researches can be easily extended to deductive databases. This provides us with more intelligent systems for knowledge bases, based on a well-defined three-valued semantic.

References

[1] F. Bancilhon and R. Ramakrishnan.– An Amateur's Introduction to Recursive Query Processing Strategies. Proc ACM SIGMOD Conference. 1986 pp16-52.

[2] S. Cook.– The Complexity of Theorem Proving Procedure. 3th ACM symposium on theory of computing, 1971.

[3] C.L.Chang and R.C.T. Lee.– Symbolic Logic and Mechanical Theorem Proving. Academic Press, inc. 1973.

[4] M.O. Cordier.– Les systèmes experts. La Recherche 151 Janvier 1984. P60-70.

[5] J.P. Delahaye.– Systèmes experts : Organisation et programmation des bases de connaissances en calcul propositionnel. Ed Eyrolles, Paris 1987.

[6] J.P. Delahaye.– Chainage avant et calculs de modèles en logique bivaluee et trivaluee. 7emes Journees Internationales sur les Systèmes Experts et leurs Applications. Avignon 1987.

[7] J.P. Delahaye and V. Thibau.– Programming in Three-Valued Logic, TCS 78, pp 189-216, 1991.

[8] R Demolombe.– A strategy for the computation of Conditional Answers. Internal report ONERA/CERT, IRIT Toulouse.

[9] R Demolombe and L.Farinas Del Cerro.– An Inference Rule for Hypothesis Generation. Internal report ONERA/CERT, IRIT Toulouse.

[10] P. Frot.– Trois systèmes experts en Turbo-Pascal. Collection Utile, Sybex 1990.

[11] Guru 3.0 MDBS Inc.– Reference manual, 1991.

[12] Intelligence service II. GSI-TECSI.– User manual, 1986.

[13] K. Inoue.– Linear Resolution for Consequence Finding. Artificial Intelligence, 56, 1992, pp301-353

[14] P. Mathieu.– The computation of useful queries for Expert Systems (in French). Internal Report IT 185 LIFL, 1990.

[15] P. Mathieu and J.P. Delahaye.– The Logical Compilation of Knowledge Bases. JELIA 90, Amsterdam. Lecture Note in Artificial Intelligence 478, pp386-398, Springer Verlag.

[16] P. Mathieu and J.P. Delahaye.– For which Bases Chaining is Sufficient. COGNITIVA 90, Madrid, pp699-702.

[17] P. Mathieu.– The Use of Three-valued Logic in Expert Systems. PhD thesis, Universite de Lille 1, Janvier 1991 (In French).

[18] Nexpert Object. Neuron Data Inc. – Reference manual, 1991.

[19] R. Reiter and J. De Kleer.– Foundations of assumption-based truth maintenance system. AAAI conference, 1987.

[20] J.D. Ullman.– Principles of Database and Knowledge-Base Systems. Vol I & II , Computer Science Press, 1989.