Available online at www.sciencedirect.com

SciVerse ScienceDirect

Fuzzy Sets and Systems 205 (2012) 1–29

FUZZY sets and systems

www.elsevier.com/locate/fss

A top-*k* query answering procedure for fuzzy logic programming

Umberto Straccia[a,*], Nicolás Madrid[b]

[a] *ISTI—CNR, Pisa, Italy*
[b] *Dept. Matemática Aplicada Universidad de Málaga, Spain*

Received 18 December 2009; received in revised form 23 January 2012; accepted 24 January 2012
Available online 2 February 2012

Abstract

We present a top-*k* query answering procedure for Fuzzy Logic Programming, in which arbitrary computable functions may appear in the rule bodies to manipulate truth values. The top-*k* ranking problem, i.e. determining the top *k* answers to a query, becomes important as soon as the set of facts becomes quite large.

Keywords: Logic programming; Deductive databases; Top-*k* query answering

1. Introduction

In this paper we address the problem of *evaluating ranked top-k queries* in a fuzzy logic programming framework.

In classical logic programming, an answer to a query is a tuple $\mathbf{c}$ that satisfies a query q, i.e. $q(\mathbf{c})$ is true in the minimal model of the program. However, very often the information need of a user involves so-called *fuzzy/vague relations*. For instance, in a logic-based e-commerce process, we may ask "find a car costing *around* \$15 000" (see [65,66]); or in ontology-mediated access to multimedia information, we may ask "find images *about* cars, which are *similar* to a given one" (see e.g. [58,79]). Unlike the classical case, a tuple $\mathbf{c}$ satisfies now the query q to a *degree* or *score* (usually in [0,1]). That is, $q(\mathbf{c})$ is not necessarily either true or false, but may be true to some degree. Therefore, in such cases, an answer to a query q is now a tuple $\langle \mathbf{c}, s \rangle$, where s is the score assigned to $q(\mathbf{c})$ in the minimal model of the program. As a consequence, answers may now be *ranked* according to their score and we are likely interested in to retrieve the top-*k* ranked answers only. This poses a new challenge when we have to deal with a huge amount of facts. Indeed, virtually every possible tuple may have a non-zero score. Of course, computing all these scores, ranking them, and then selecting the top-*k* ones is likely not feasible in practice as soon as the set of facts becomes quite large.

In this work, we address the top-*k* retrieval problem for a very expressive fuzzy variant of Datalog [82]. At the extensional level, each fact may have a score, while at the intensional level rules describe the domain of application. Queries are conjunctive queries. Furthermore, we allow arbitrary scoring functions (monotone, total and finite time computable functions) to score tuples.

* Corresponding author. Tel.: +39 050315 2894.
E-mail addresses: straccia@isti.cnr.it (U. Straccia), nmadrid@ctima.uma.es (N. Madrid).

0165-0114/$ - see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.fss.2012.01.016

The top-k retrieval problem is a novel issue for (fuzzy) logic programming and only little work exist yet on the topic (see Section 5). The basic reasoning idea stems from the database literature (see e.g. [44]) and consists in retrieving iteratively query answers and contemporarily computing a threshold δ. The threshold δ has the fundamental property that any newly retrieved tuple will have a score less or equal than δ. As a consequence, as soon as we have retrieved k tuples greater or equal to δ, we may stop.

We point out that closest to our work are [51,76]. Indeed, [76] addresses the top-k retrieval for general (recursive) fuzzy *Logic Programms* (LPs), while [51] slightly extends [76] as it tries to rely as much as possible on current top-k database technology. So, specifically, the contribution of this work is as follows:

- we recap the basic technique underlying our top-k retrieval method in fuzzy logic programming;
- more importantly, we introduce a novel threshold function. This threshold function is of fundamental importance. In fact, unfortunately both [51,76] illustrate incorrect algorithms in the sense that the top-k answers may not be the top-k ones. The reason is that in these works the computed threshold does not guarantee that any successively computed answer has score less or equal than the threshold. In this work, we correct this major flaw and show that the correction is non-trivial; and
- we also show that we can smoothly extend top-k query answering to the top-k–n problem. This latter problem has been shown to be fundamental in electronic Matchmaking [65,66], but the problem has been left unsolved so far.

We proceed as follows. In the next Section, we provide basic definitions about the language and the problems to solve. In Section 3 we address the top-k retrieval problem, while in Section 4 we address the top-k–n problem. Section 5 reviews related work, while Section 6 concludes and points to further issues. Eventually, in Appendix A we list some references related to logic programming, uncertainty and imprecision, while in Appendix B we sketch the computational complexity of our procedures.

2. Preliminaries

Truth space: To always guarantee termination of the procedures presented here, we will assume that the truth space is $\mathcal{L} = \{\bot, 0, 1/n, \ldots (n-1)/n, 1\}$, for some positive integer n. We will use the symbol $\bot$ to denote the truth "unknown" and extend the linear order $\leq$ over rational numbers by postulating $\bot < 0$. The main idea is that a statement $p(\mathbf{c})$, rather than being interpreted as either true or false, will be mapped into a truth value (or score) s in $\mathcal{L}$ (examples of other truth spaces can be found in e.g. [18,41,71]). From a practical point of view the finiteness of $\mathcal{L}$ is a limitation we can live with especially taking into account that computers have finite resources, and thus, only a finite set of truth degrees can be represented.

Knowledge base: A *knowledge base* $\mathcal{K} = \langle \mathcal{F}, \mathcal{P} \rangle$ consists of a *facts component* $\mathcal{F}$ and an *LP component* $\mathcal{P}$, which are both defined below.

Facts component: $\mathcal{F}$ is a finite set of expressions of the form

$$\langle p(c_1, \ldots, c_n), s \rangle,$$

where p is an n-ary relation, every c_i is a constant, and $s > \bot$ is a score in $\mathcal{L}$. The underlying meaning of such an expression is that the degree of truth of $p(c_1, \ldots, c_n)$ is equal to or greater than s. For each p, we represent the facts $\langle p(c_1, \ldots, c_n), s \rangle$ in $\mathcal{F}$ by means of a relational $n+1$-ary table T_p, containing the records $\langle c_1, \ldots, c_n, s \rangle$. We assume that there cannot be two records $\langle c_1, \ldots, c_n, s_1 \rangle$ and $\langle c_1, \ldots, c_n, s_2 \rangle$ in T_p with $s_1 < s_2$ (if there are, then we remove the one with the lower score). We assume each table sorted in descending order with respect to the scores. For ease, we may omit the score component and in such cases the value 1 is assumed.

Rule component: $\mathcal{P}$ is a finite set of *rules* of the form

$$p(\mathbf{x}) \leftarrow f(p_1(\mathbf{z}_1), \ldots, p_m(\mathbf{z}_m)),$$

where

1. p is an n-ary relation, every p_i is an n_i-ary relation and $p_i \neq p_j$ for $i \neq j$;
2. $\mathbf{x}$ is a tuple of variables each of which occurs on the right hand side;
3. each $\mathbf{z_i}$ is a tuple of constants or variables;

4. f is a *scoring* function $f: \mathcal{L}^m \to \mathcal{L}$, which combines the degree of truth of the m relations $p_i(\mathbf{c}'_i)$ into an overall *truth degree* to be assigned to the rule head $p(\mathbf{c})$. We assume that f is *monotone*, that is, for each $\mathbf{v}, \mathbf{v}' \in \mathcal{L}^m$ such that $\mathbf{v} \leq \mathbf{v}'$, $f(\mathbf{v}) \leq f(\mathbf{v}')$ holds, where $(v_1, \ldots, v_m) \leq (v'_1, \ldots, v'_m)$ iff $v_i \leq v'_i$ for all i. We also assume that f is computable, i.e. for any input, the value of f can be determined in finite time, and that $f(\ldots, \bot, \ldots) = \bot$.

We call $p(\mathbf{x})$ the *head* and $f(p_1(\mathbf{z}_1), \ldots, p_m(\mathbf{z}_m))$ the *body* of the rule. We assume that relations occurring in $\mathcal{F}$ do not occur in the head of rules (so, we do not allow that the fact relations occurring in $\mathcal{F}$ can be redefined by $\mathcal{P}$). As usual in deductive databases, the relations in $\mathcal{F}$ are called *extensional* relations, while the others are *intensional* relations.

Remark 1. Note that we impose relations p_i in a rule body to be distinct. This is not a limitation as we may rewrite, e.g.

$$p(x) \leftarrow f(q(x, y), q(y, z)),$$

as

$$p(x) \leftarrow f(q(x, y), p'(y, z)),$$
$$p'(y, z) \leftarrow q(y, z),$$

for a new relation symbol p'.

We note that from a practical point of view, we may have introduced *typed or sorted* constants, as in e.g. [84], and built-in relations. This is useful in practice, but from a theoretical point of view their management is easy, so we leave it out for the ease of presentation.

Semantics: From a semantics point of view, given $\mathcal{K} = \langle \mathcal{F}, \mathcal{P} \rangle$, the notions of *Herbrand universe* $H_{\mathcal{K}}$ (the set of all constants occurring in $\mathcal{K}$) and *Herbrand base* (the set of all ground atoms) $B_{\mathcal{K}}$ of $\mathcal{K}$ are as usual. Additionally, given $\mathcal{K}$, the set of ground rules $\mathcal{K}^*$ derived from the grounding of $\mathcal{P}$ is constructed as follows:

1. set $\mathcal{K}^*$ to be $\{p(\mathbf{c}) \leftarrow s | \langle p(\mathbf{c}), s \rangle \in \mathcal{F}\}$;
2. add to $\mathcal{K}^*$ the set of all ground instantiations of rules in $\mathcal{P}$.

An *interpretation* I for $\mathcal{K}$ is a function $I: B_{\mathcal{K}} \to \mathcal{L}$.

Remark 2. Note that the graph $G(I)$ of an interpretation I is set of pairs $\langle A, s \rangle \in B_{\mathcal{K}} \times \mathcal{L}$ such that for any ground atom A there is at most on pair $\langle A, s \rangle \in G(I)$. Hence, for ease we may represent an interpretation as wells as via its graph $G(I)$ by omitting the pairs $\langle A, \bot \rangle$.

We extend I in the usual way:

1. $I(s) = s$, for $s \in \mathcal{L}$; and
2. $I(f(A_1, \ldots, A_m)) = f(I(A_1), \ldots, I(A_m)))$.

Remark 3. Note that a score combination function has a fixed interpretation, i.e. does not vary from interpretation to interpretation. For ease of presentation, we identify the interpretation of f with f itself.

Furthermore, $\leq$ is extended from $\mathcal{L}$ to the set $\mathcal{I}(\mathcal{L})$ of all interpretations point-wise: (i) $I_1 \leq I_2$ iff $I_1(A) \leq I_2(A)$, for every ground atom A. With $I_\bot$ we denote the bottom interpretation ($I_\bot$ maps all atoms to $\bot$), while with $I_\top$ we denote the top interpretation ($I_\top$ maps all atoms to 1). It turns out that $\langle \mathcal{I}(\mathcal{L}), \leq \rangle$ is a complete lattice and is finite.

Now, following [30,45,46], I is a *model* of $\mathcal{K}$, denoted $I \models K$, iff for all ground rules $A \leftarrow \varphi \in \mathcal{K}^*$, $I(\varphi) \leq I(A)$ holds.

Among all the models, one model plays a special role: namely the $\leq$-least model $M_{\mathcal{K}}$ of $\mathcal{K}$. The existence, finiteness and uniqueness of the minimal model $M_{\mathcal{K}}$ is guaranteed to exists by the following argument. Consider $\mathcal{K}$, the Herbrand base $B_{\mathcal{K}} = \{A_1, \ldots, A_n\}$ and $\mathcal{K}^*$. Let us associate to each atom $A_i \in B_{\mathcal{K}}$ a variable x_i, which will take a value in $\mathcal{L}$ (sometimes, we will refer to that variable with x_A as well). An interpretation I may be seen as an assignment of truth values to the variables $x_1, \ldots, x_n$ and vice-versa. Now, for each ground fact $\langle A, s \rangle$ in $\mathcal{K}^*$, we consider the equation

$$x_A = s,$$

while for each ground rule $A \leftarrow f(A_1, \ldots, A_m)$ in $\mathcal{K}^*$ we consider the equation

$$x_A = f(x_{A_1}, \ldots, x_{A_m}),$$

if A is head of at most one rule, while we consider the equation

$$x_A = \max\{f(\ldots) | A \leftarrow f(\ldots) \in \mathcal{K}^*\},$$

if A is head of more than one rule.

Remark 4. Please note that, as $\max(\bot, a) = a \neq \bot$ (unless $a = \bot$), max cannot occur in rule bodies. We may use instead the function $\hat{\max}$ defined as follows: $\hat{\max}(a, b) = \hat{\max}(b, a)$, $\hat{\max}(\bot, a) = \bot$ and $\hat{\max}(a, b) = a$ if $a \geq b > \bot$.

Now, given $\mathcal{K}^*$, we have obtained the system of equations

$$\begin{aligned} x_1 &= f_1(x_{1_1}, \ldots, x_{1_{a_1}}), \\ &\vdots \\ x_n &= f_n(x_{n_1}, \ldots, x_{n_{a_n}}), \end{aligned} \tag{1}$$

where the variables inside the functions f_i belong to $\{x_1, \ldots, x_n\}$. Each variable x_{i_k} will take a value in $\mathcal{L}$, each (monotone) function f_i determines the value of x_i (i.e. A_i). We refer to the monotone system as in Eq. (1) as the tuple $\mathcal{S} = \langle \mathcal{L}, V, \mathbf{f} \rangle$, where $V = \{x_1, \ldots, x_n\}$ are the variables and $\mathbf{f} = \langle f_1, \ldots, f_n \rangle$ is the tuple of functions. Now, it can be verified that the minimal model of $\mathcal{K}^*$, i.e. $M_{\mathcal{K}}$, is bijectively related to the least solution of the system (1). More precisely, any model is bijectively related to a tuple $\mathbf{x}$ such that $f(\mathbf{x}) \leq \mathbf{x}$, i.e. $\mathbf{x}$ is a pre fixed-point, the set of pre fixed-points is non-empty ($\mathbf{1}$ belongs to it) and the least element is the least fixed-point. As it is well known (see e.g. [4]), a monotonic equation system as (1) (each function $f_i : \mathcal{L}^{a_i} \mapsto \mathcal{L}$ in Eq. (1) is $\leq$-monotone) has a $\leq$-least solution, $\mathrm{lfp}(\mathbf{f})$, the $\leq$-least fixed-point of $\mathbf{f}$ is given as the least upper bound of the $\leq$-monotone sequence, $\mathbf{y}_0, \ldots, \mathbf{y}_i, \ldots$, where

$$\mathbf{y}_0 = \bot, \quad \mathbf{y}_{i+1} = \mathbf{f}(\mathbf{y}_i). \tag{2}$$

It is thus immediate that, as the truth space is *finite*, the above sequence converges in a *finite* number of steps and that the $\leq$-least model $M_{\mathcal{K}}$ of $\mathcal{K}$ is finite.

We point out that we may also guarantee the existence and uniqueness of the minimal model $M_{\mathcal{K}}$ along the tradition of logic programming. In fact, the existence and uniqueness of the minimal model $M_{\mathcal{K}}$ is guaranteed by the fixed-point characterisation based on the $\leq$-monotone function $\Phi_{\mathcal{K}}$: for an interpretation I, for any ground atom A

$$\Phi_{\mathcal{K}}(I)(A) = \max(\{I(\varphi) | A \leftarrow \varphi \in \mathcal{K}^*\}).$$

Then the minimal model $M_{\mathcal{K}}$ of $\mathcal{K}$ is the least fixed-point of $\Phi_{\mathcal{K}}$ and can be computed in the usual way by iterating $\Phi_{\mathcal{K}}$ over $I_\bot$ (see also [70,71]).

Remark 5. Please, note that there is a subtle difference between the two KBs

$$\mathcal{K}_1 = \{a \leftarrow b, \quad a \leftarrow c, \quad b \leftarrow 0.7\}$$

and

$$\mathcal{K}_2 = \{a \leftarrow \hat{\max}(b, c), \quad b \leftarrow 0.7\}.$$

The system of equations is in the former case

$$x_a = \max(b, c), \quad x_b = 0.7,$$

while in the latter case the system of equations is

$$x_a = \hat{\max}(b, c), \quad x_b = 0.7.$$

Therefore, $M_{\mathcal{K}_1}(a) = 0.7$, while $M_{\mathcal{K}_2}(a) = \bot$ hold, as $\max(0.7, \bot) = 0.7$ and $\hat{\max}(0.7, \bot) = \bot$ instead.

This difference is used to avoid conclusions in a KB with unique rule such as

$$a(x) \leftarrow f(b(x, y), c(y, z)),$$

without having appropriate joining instances of b and c in the facts component (see e.g. Examples 1 and 4).

Query: A *query* is of the form $q(\mathbf{x})$, intended as a question about the truth degree of the ground instances of $q(\mathbf{x})$ in the minimal model of $\mathcal{K}$. The *answer set* of q w.r.t. $\mathcal{K}$ is defined as the set $ans(q, \mathcal{K})$ of tuples $\langle \mathbf{c}, s\rangle \in H_{\mathcal{K}} \times \cdots \times H_{\mathcal{K}} \times \mathcal{L}$ such that $M_{\mathcal{K}}(q(\mathbf{c})) = s > \bot$ (the score of $\mathbf{c}$ is $s > \bot$ in the minimal model).

Example 1. Given $\mathcal{K}$ with rule (the truth space is defined with $n = 100$)

$$q(x) \leftarrow 0.5 \cdot (p(x) + r(x)),$$

and facts

$$\langle p(a), 0.9\rangle, \quad \langle p(b), 0.2\rangle, \quad \langle r(b), 0.4\rangle,$$

then $ans(q, \mathcal{K}) = \{\langle b, 0.3\rangle\}$. Please note that e.g. $\langle a, 0.45\rangle \notin ans(q, \mathcal{K})$ since $M_{\mathcal{K}}(r(a)) = \bot$. □

Example 2. Of course, due to the expressiveness of the score combination functions, we can also easily allow queries in which we would like to retrieve only tuples above a given threshold α. For instance, in Example 1 above, for $\alpha \in (0, 1]$ consider the monotone function

$$f_\alpha(x) = \begin{cases} x & \texttt{if } x \geq \alpha \\ \bot & \texttt{otherwise}. \end{cases}$$

Then the answers to the rule

$$q(x) \leftarrow f_\alpha(0.5 \cdot (p(x) + r(x)))$$

are those tuples above the threshold α. □

As it is well-known, the following example also shows that ω steps may not be sufficient to compute the minimal model (for similar examples, see e.g. [39,84]) over a non-finite truth space.

Example 3. Consider $\mathcal{L} = [0, 1]$, the score combination function $f(x) = (x + a)/2$ $(0 < a \leq 1)$, and $\mathcal{K}_1$ containing the rules

$$p \leftarrow 0, \quad p \leftarrow f(p).$$

Then the minimal model is attained after ω steps of $\Phi_{\mathcal{K}}$ iterations starting from $I_\bot$ and is $M_{\mathcal{K}_1}(p) = a$.

Now, consider the (non-continuous) function $g(x) = 0$ if $x < a$, and $g(x) = 1$ otherwise, and $\mathcal{K}_2$ containing only the rules

$$p \leftarrow 0, \quad p \leftarrow f(p), \quad q \leftarrow g(p).$$

Then the minimal model is attained after $\omega+1$ steps of $\Phi_{\mathcal{K}}$ iterations starting from $I_\bot$ and is $M_{\mathcal{K}_2}(p) = a$, $M_{\mathcal{K}_2}(q) = 1$. However, it is not difficult to show that if all functions appearing in $\mathcal{P}$ are Scott-continuous, then at most ω steps are necessary to compute the minimal model [70,71]. □

In any case, finiteness of truth space, monotonicity of score combination functions together with the finiteness of the grounded program guarantees then that the minimal model can be computed in finite time by iterating $\Phi_{\mathcal{K}}$ over $I_\bot$.

The principal problem we address in this work is the top-k retrieval problem.

Definition 1 (*Top-k retrieval*). Given $\mathcal{K}$, retrieve top-k tuples of the answer set of q ranked in decreasing order relative to the score, denoted

$$ans_k(q, \mathcal{K}) = \mathtt{Top}_k(ans(q, \mathcal{K})).$$

We note that retrieving the top-k answers of an extensional relation p is trivial as we have just to retrieve the first k tuples in the relational table T_p associated to p. Hence, we restrict top-k retrieval to intensional relations only. Note also that (i) $ans_k(q, \mathcal{K})$ is not necessarily unique as there may be several tuples having the same score of the bottom ranked one in $ans_k(q, \mathcal{K})$ (we assume in this case that ties are broken arbitrary); and (ii) there may be less than k tuples in $ans_k(q, \mathcal{K})$ as there might not be k non-$\bot$ scored tuples in $ans(q, \mathcal{K})$.

Furthermore, please note that having introduced the degree $\bot \in \mathcal{L}$ allows us to be compatible with top-k SQL retrieval over relational databases (see e.g. [44]), where it is understood and common practice not to retrieve tuples that do not satisfy the condition of the plain SQL query (that is, the SQL query where the function computing the score is left out), and, in particular, tuples not occurring in any relational table. The following example illustrates the point.

Example 4. For instance, suppose that a motorbike relation is

`mb(id, price, comfort)`,

with tuples

$\langle 22, 9000, 0.3\rangle$ $\langle 23, 32000, 0.8\rangle$
$\langle 24, 46000, 0.9\rangle$ $\langle 25, 7500, 0.0\rangle$.

Now, consider a top-5 SQL query (see e.g. [44])

```
SELECT   *
FROM     mb
WHERE    mb.price < 15000
ORDER BY mb.comfort DESC
LIMIT    5
```

where the column `mb.comfort` contains the degree of comfort of a motorbike. Then:

- a tuple not appearing in the motorbike relation will not be retrieved (not even with degree 0).
- a tuple not satisfying the condition `mb.price` $<$ `15 000` will not be retrieved (not even with degree 0).
- a tuple satisfying the condition `mb.price` $<$ `15 000` may potentially be retrieved with degree 0.

Therefore, for the top-k SQL case, the top-5 answers will just be the two tuples

$\langle 22, 9000, 0.3\rangle$
$\langle 25, 7500, 0.0\rangle$.

Note that in our setting, using facts

$\langle \mathtt{mb}(22, 9000), 0.3\rangle$ $\langle \mathtt{mb}(23, 32000), 0.8\rangle$
$\langle \mathtt{mb}(24, 46000), 0.9\rangle$ $\langle \mathtt{mb}(25, 7500), 0.0\rangle$,

and query

$$q(x_1, x_2) \leftarrow f(\mathtt{mb}(x_1, x_2), x_2 \leq 1500),$$

where $f(x_1, x_2) = x_1$ if $x_2 \leq 1500$, otherwise $\bot$, we get the same answers. □

Example 5 (*Ragone [65,66]*). Suppose we have a car selling site, and we would like to buy a car. The properties of the cars are described in the relation *CarTable* as shown in Fig. 1. Here, the score is implicitly assumed to be 1 in each record (the truth space is $\mathcal{L}$ with $n = 100$). Now, suppose that in buying a car, preferably we would like to pay around \$11 000 and the car should have less than 15 000 km. Of course, our constraints on price and kilometers are not crisp as

ID	MODEL	PRICE	KM	DISCOUNT
455	MAZDA 3	12500	18000	0.1
34	ALFA 156	12000	17000	0.2
1812	FORD FOCUS	13000	16000	0.2
⋮	⋮	⋮	⋮	⋮

Fig. 1. The car table.

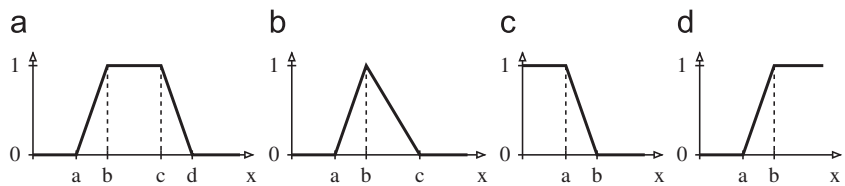

Fig. 2. (a) Trapezoidal function $trz(x; a, b, c, d)$, (b) triangular function $tri(x; a, b, c)$, (c) left shoulder function $ls(x; a, b)$, and (d) right shoulder function $rs(x; a, b)$.

$$
\begin{array}{lcll}
\texttt{Cars}(x_1) & \leftarrow & \texttt{CarTable}(x_1, x_2, x_3, x_4, x_5) & \\
\texttt{hasPrice}(x_1, x_3) & \leftarrow & \texttt{CarTable}(x_1, x_2, x_3, x_4, x_5) & \\
\texttt{hasKM}(x_1, x_4) & \leftarrow & \texttt{CarTable}(x_1, x_2, x_3, x_4, x_5) & \\
 & & & \\
\texttt{BuyPref1}(x, p) & \leftarrow & \min(\texttt{Cars}(x), \texttt{hasPrice}(x, p), \texttt{ls}(p; 9000, 13000)) & (a) \\
\texttt{BuyPref2}(x, k) & \leftarrow & \min(\texttt{Cars}(x), \texttt{hasKM}(x, k), \texttt{ls}(k; 10000, 20000)) & (b) \\
\texttt{Buy}(x, p, k) & \leftarrow & 0.8 \cdot \texttt{BuyPref1}(x, p) + 0.2 \cdot \texttt{BuyPref2}(x, k)\ . & (c)
\end{array}
$$

Fig. 3. The car buying rules.

we may still accept to some degree, e.g. a car's price of \$11, 200 and with 16 000 km. Hence, these constraints are rather *vague*. We model this by means of so-called left-shoulder functions (see Fig. 2 for some typical fuzzy membership functions), which is a well known fuzzy membership function in fuzzy set theory. We may model the vague constraint on the price with $ls(x; 9000, 13\,000)$ dictating that we are definitely satisfied if the price is less than \$9000, but can pay up to \$13 000 to a lesser degree of satisfaction. Similarly, we may model the vague constraint on the kilometers with $ls(x; 10\,000, 20\,000)$. We also set some preference (weights) on these two vague constraints, say the weight 0.8 to the price constraint and 0.2 to the kilometers constraint, indicating that we give more priority to the price rather than to the car's kilometers. The rules encoding the above conditions are represented in Fig. 3. Rule (a) in Fig. 3 encodes the preference on the price. Here, $\texttt{ls}(p; 9000, 13\,000)$ is an *extensional relation* that given a price p returns the degree of truth provided by the left-shoulder function $\texttt{ls}(\cdot; 9000, 13\,000)$ evaluated on the input p (recall that in our setting, all fuzzy membership functions provide a truth value in $\mathcal{L}$ and, thus, we may represent it by means of a finite set of facts). Similarly, for rule (b). Rule (c) encodes the combination of the preferences by taking into account the weight given to each preference.

The table below reports the top-3 answers of Buy together with their score

ID	PRICE	KM	s
34	12 000	17 000	0.26
455	12 500	18 000	0.14
1812	13 000	16 000	0.08.

□

Example 6 (*Ragone [65,66] cont.*). Consider Example 5. Now, additionally the seller may offer a discount on the car's catalogue price, as indicated in the *CarTable* relation. For instance, related to the Mazda3, the seller may consider optimal to sell above \$12 500, but can go down to \$11 250 to a lesser degree of satisfaction. Hence, we may model this as the right-shoulder function $rs(p; 11\,250, 12\,500)$. The hasPossiblePrice relation determines the prices the

$$
\begin{array}{lcll}
\texttt{CataloguePrice}(x_1, x_3) & \leftarrow & \texttt{CarTable}(x_1, x_2, x_3, x_4, x_5) & \\
\texttt{MinimalPrice}(x_1, mp) & \leftarrow & \min(\texttt{CarTable}(x_1, x_2, x_3, x_4, x_5), mp = x_3 \cdot (1 - x_5)) & \\
\texttt{hasPrice}(x, p) & \leftarrow & \texttt{hasPossiblePrice}(x, p) & \\
 & & & \\
\texttt{hasPossiblePrice}(x_1, x_3) & \leftarrow & \texttt{CarTable}(x_1, x_2, x_3, x_4, x_5) & \\
\texttt{hasPossiblePrice}(x, p) & \leftarrow & \min(\texttt{MinimalPrice}(x, mp), \texttt{hasPossiblePrice}(x, p'), & \\
 & & p = p' - 100, p \geq mp) & \\
 & & & \\
\texttt{Sell}(x, p) & \leftarrow & \min(\texttt{Cars}(x), \texttt{MinimalPrice}(x, mp), \texttt{CataloguePrice}(x, cp), & \\
 & & \texttt{hasPossiblePrice}(x, p), \texttt{rs}(p; mp, cp)) & \\
\texttt{Match}(x, p, k) & \leftarrow & \texttt{Buy}(x, p, k) \cdot \texttt{Sell}(x, p) & (d)
\end{array}
$$

Fig. 4. The car selling rules.

seller is willing to consider for a car and is a multiple of 100 between the catalogue price and the maximal discounted price. For each car the optimal price to be sold is determined as the product of the buyer's degree of satisfaction and the seller's degree of satisfaction. The rules are shown in Fig. 4. Note that there are some built-in, crisp predicates, such as $p = p' - 100$, that evaluates to 1 iff the value of p equals to the value of $p' - 100$.

Rule (d) encodes the overall satisfaction of the buyer and the seller if a car x is sold at price p and with k kilometers.

Unlike Example 5, here we are interested in computing for each car the best matching degree and then rank the top-k cars in decreasing order of matching degree:

$$ans_k(Match, \mathcal{K}) = \texttt{Top}_k\{\langle x, p, k, s\rangle | \quad \langle p, k, s\rangle \in \texttt{Top}_1\{\langle p', k', s'\rangle | s' = M_{\mathcal{K}}(Match(x, p', k'))\}\}.$$

Basically, for each car x of the database, we compute the best matching $\langle p, k, s\rangle$ for it, i.e. $\langle p, k, s\rangle \in \texttt{Top}_1\{\langle p', k', s'\rangle | s' = M_{\mathcal{K}}(Match(x, p', k')\}$, and then rank the top-$k$ cars.

It can be verified that the answer to the top-3 problem is

ID	PRICE	KM	s
34	11 500	17 000	0.29
455	12 200	18 000	0.15
1812	11 000	16 000	0.11

Hence, the buyer and seller may opt for an agreement on the `ALFA 156` at $11 500. □

Note that the main difference between Example 5 and Example 6 is that in the latter we have a *nested* top-k, top-1 problem to solve.

In the following section we will address the top-k problem, while in Section 4, we will address the problem of nested top-k, top-n problem.

3. Computing top-k answers

We describe an incremental query driven top-k query answering algorithm. Of course, theoretically we always have the possibility to compute all answers, then rank them afterwards, and eventually to select the top-k ones only. However, this requires to compute the score of all answers. We would like to avoid this in cases in which the extensional database is large and potentially too many tuples would satisfy the query.

A distinguishing feature of our query answering procedure is that we do not determine all answers, but collect, during the computation, answers incrementally together and we can stop as soon as we have gathered k answers greater or equal than a computed threshold δ.

To the ease of reading, we will proceed stepwise. In the next section, we will provide an algorithm answering ground queries. Additionally, we will see later on that the same algorithm can be used to compute the novel threshold we are going to propose in Section 3.4. In Section 3.2 we will extend the latter procedure to compute the top-k answers only.

3.1. A query driven procedure for equational systems

It is illustrative to address the following specific problem. Consider an equational system $\mathcal{S} = \langle \mathcal{L}, V, \mathbf{f}\rangle$ and a variable x_i. How can we compute the value of variable x_i in the least fixed-point of $\mathcal{S}$? The immediate way is to

Table 1
General query driven algorithm.

Procedure $Solve(\mathcal{S}, Q)$
Input: $\leq$-monotonic system $\mathcal{S} = \langle \mathcal{L}, V, \mathbf{f} \rangle$, where $Q \subseteq V$ is the set of query variables
Output: A set $B \subseteq V$, with $Q \subseteq B$ such that the mapping $\mathtt{v}$ equals $\mathrm{lfp}(\mathbf{f})$ on B
1. $\mathtt{A} := Q$, $\mathtt{dg} := Q$, $\mathtt{in} := \emptyset$
   for all $x \in V$ **do** $\mathtt{v}(x) = \bot$, $\mathtt{exp}(x) = \mathtt{false}$ **endfor**
2. **while** $\mathtt{A} \neq \emptyset$ **do**
3. **select** $x_i \in \mathtt{A}$, $\mathtt{A} := \mathtt{A} \setminus \{x_i\}$, $\mathtt{dg} := \mathtt{dg} \cup \mathtt{s}(x_i)$
4. $r := f_i(\mathtt{v}(x_{i_1}), \ldots, \mathtt{v}(x_{i_{a_i}}))$
5. **if** $r > \mathtt{v}(x_i)$ **then** $\mathtt{v}(x_i) := r$, $\mathtt{A} := \mathtt{A} \cup (\mathtt{p}(x_i) \cap \mathtt{dg})$ **fi**
6. **if not** $\mathtt{exp}(x_i)$ **then** $\mathtt{exp}(x_i) = \mathtt{true}$, $\mathtt{A} := \mathtt{A} \cup (\mathtt{s}(x_i) \setminus \mathtt{in})$, $\mathtt{in} := \mathtt{in} \cup \mathtt{s}(x_i)$ **fi**
   endwhile

compute bottom-up the least fixed-point as described in Eq. (2), and then look for the value of x_i in the least fixed-point. But, there is also a query driven method [4]. It presents a simple, yet general algorithm for computing the least fixed-point of a system of monotonic equations. The method has been used then in [70] as a basis for a query driven ground query answering method for normal logic programs and has further been extended in [51,75,76]. This is not surprising, as we have seen in the previous section that the minimal model of $\mathcal{K}^*$, i.e. $M_{\mathcal{K}}$, is bijectively related to the least solution of a system of the form (1). Hence, if we want to know the truth value of a ground atom A in $M_{\mathcal{K}}$, it suffices to look at the value of the variable x_A in the least fixed-point of the related equational system. So, if we have a query driven procedure determining the value of x_A in the least-fixed point, then we have also a query driven procedure returning the truth value of A in $M_{\mathcal{K}}$. The importance of this procedure within this paper is due to the fact that our query driven top-k procedure described in the next section will be an extension of the one we present next and, thus, may ease the comprehension of it.

Formally, let us consider an equational system $\mathcal{S} = \langle \mathcal{L}, V, \mathbf{f} \rangle$ of the form (1). The procedure described in Table 1 determines the value of a set of variables in Q in the least-fixed point, and is a slight refinement of the one presented in [4]. We next describe how it works. We use some auxiliary functions. $\mathtt{s}(x)$ denotes the set of *sons* of x, i.e. $\mathtt{s}(x_i) = \{x_{i_1}, \ldots, x_{i_{a_i}}\}$ (the set of variables appearing in the right hand side of the definition of x_i in (1)). $\mathtt{p}(x)$ denotes the set of *parents* of x, i.e. the set $\mathtt{p}(x) = \{x_i | x \in \mathtt{s}(x_i)\}$ (the set of variables depending directly on the value of x). We assume that each function $f_i : \mathcal{L}^{a_i} \mapsto \mathcal{L}$ in Eq. (1) is $\leq$-monotone. We also use f_x in place of f_i, for $x = x_i$. Informally the algorithm works as follows. Assume we are interested in the value of x_0 in the least fixed-point of the system. We associate to each variable x_i a marking $\mathtt{v}(x_i)$ denoting the current value of x_i (the mapping $\mathtt{v}$ contains the current value associated to the variables). Initially, $\mathtt{v}(x_i)$ is $\bot$.

We start with putting x_0 in the *active* set of variables $\mathtt{A}$, for which we evaluate whether the current value of the variable is identical to whatever its right-hand side evaluates to. When evaluating a right-hand side it might of course turn out that we do indeed need a better value of some sons (which are initialised with the value $\bot$) and put them on the set of active nodes to be examined. In doing so we keep track of the dependencies between variables, and whenever it turns out that a variable changes its value (actually, it can only $\leq$-increase) all variables that might depend on this variable are put in the active set to be reexamined. At some point (even if cyclic definitions are present) the active set will become empty, because of the finiteness of the truth space, and we have actually found part of the fixed-point, sufficient to determine the value of the query x_0.

The variable $\mathtt{dg}$ collects the variables that may influence the value of the query variables, the array variable $\mathtt{exp}$ traces the equations that have been "expanded" (the body variables are put into the active set), while the variable $\mathtt{in}$ keeps track of the variables that have been put into the active set so far due to an expansion (to avoid to put the same variable multiple times in the active set due to function body expansion).

In [4], it is shown that the above algorithm behaves correctly.

Proposition 1 (*Andersen [4]*). *Given a monotone system of equations* $\mathcal{S} = \langle \mathcal{L}, V, \mathbf{f} \rangle$, *then after a finite number of steps,* $Solve(\mathcal{S}, Q)$ *determines a set* $B \subseteq V$, *with* $Q \subseteq B$ *such that the mapping* $\mathtt{v}$ *equals* $\mathrm{lfp}(\mathbf{f})$ *on* B, *i.e.* $\mathtt{v}_{|B} = \mathrm{lfp}(\mathbf{f})_{|B}$.

In Appendix B we sketch the computational complexity of the *Solve* procedure.

3.2. A query driven top-k procedure

We are ready now to describe our query driven top-k algorithm. For the sake of illustrative purposes, we consider the following well-known example in which we define a path in a weighted graph.

Example 7. Consider the following rules:

$$r_1 : \mathtt{path}(x, y) \leftarrow \mathtt{edge}(x, y), \tag{3}$$

$$r_2 : \mathtt{path}(x, y) \leftarrow \min(\mathtt{path}(x, z), \mathtt{edge}(z, y)). \tag{4}$$

The knowledge base $\mathcal{K}$ contains the rules above and the extensional database of edges as shown in the relational table $T_{\mathtt{edge}}$ below:

$T_{\mathtt{edge}}$		
c	b	0.6
a	c	0.5
b	a	0.4
a	b	0.3

It can be verified that the set of answers of predicate `path` is given by

$ans(\mathtt{path}, \mathcal{K})$								
a	a	0.4	b	a	0.4	c	a	0.4
a	b	0.5	b	b	0.4	c	b	0.6
a	c	0.5	b	c	0.4	c	c	0.4

□

The procedure *TopAnswers* is detailed in Table 2.[1] Basically, we will compute answers iteratively one by one together with a threshold δ. The threshold has the important property that if we have already collected k answers with a score greater or equal to δ we can stop, as any not yet computed answer will have a score less or equal than δ. This means that we have already determined the top-k answers. Roughly, we proceed as follows. Suppose q is the query. We start with putting the predicate symbol q in the *active* set of predicate symbols `A`. At each iteration step we select a new predicate p from `A`, look for a the next highest scoring tuple for p (procedure *getNextTuple*), update the current answer set for p (`rankedList`(p)), add all predicates p', whose rule body contains p (the parents of p), to `A`, i.e. all predicate symbols that might depend on p are put in the active set to be examined, and finally we update the threshold. If we have already gathered k answers for q whose score is greater or equal than the threshold we stop. Overall, our procedure uses some auxiliary functions and data structures:

- for predicate symbol p_i, $\mathtt{s}(p_i)$ is the set of predicate symbols occurring in the rule body of a rule having p_i in its head,[2] i.e. the *sons* of p_i, from which we exclude the extensional predicates. $\mathtt{p}(p_i)$ denotes the parents of p_i, i.e. $\mathtt{p}(p_i) = \{p_j | p_i \in \mathtt{s}(p_j)\}$;
- the variable `rankedList` contains, for each intensional relation p, the current top-ranked tuples together with their score, i.e. for each predicate symbol p, the tuples $\langle \mathbf{c}, s\rangle$ in `rankedList`(p) are ranked in decreasing order with respect to the score s. We do not allow $\langle \mathbf{c}, s\rangle$ and $\langle \mathbf{c}, s'\rangle$ to occur in `rankedList`(p) with $s \leq s'$ (if so, we remove the tuple with the lower score);
- the variable `Q` is a global variable. It is used in the *getNextTuple* procedure and contains, for each intensional relation p, the next top-ranked tuples to be returned. The tuples in `Q`(p) are ranked in decreasing order with respect to the score s.

There are other variables, which play a role in the procedure *getNextTuple* (see Table 2), but are defined in the *TopAnswers* procedure, as they act as global variables.

We now describe the *getNextTuple* procedure (see Table 2). Its main purpose is, given a relation symbol p and the rules $r_i : p(\mathbf{x}) \leftarrow \varphi_i$ having $p(\mathbf{x})$ as head, to get back the next tuple (and its score) satisfying the body conditions of

[1] The condition $\mathtt{rL}' = \mathtt{rankedList}$ means that the contents do not change.

[2] Recall that there may be more than one rule having p_i in its head.

Table 2
The top-k query answering procedure.

Procedure $TopAnswers(\mathcal{L}, \mathcal{K}, q, k)$
Input: Truth space $\mathcal{L}$, KB $\mathcal{K} = \langle \mathcal{F}, \mathcal{P} \rangle$, query relation q, $k \geq 1$
Output: Mapping $\mathtt{rankedList}$ such that $\mathtt{rankedList}(q)$ contains top-k answers of q
Init: $\delta = 1$, **for all** predicates p in $\mathcal{P}$ **do**
  if p intensional **then** $\mathtt{rankedList}(p) = \emptyset$, $\mathtt{Q}(p) := \emptyset$ **fi**
  if p extensional **then** $\mathtt{rankedList}(p) = T_p$ **fi**
 endfor
1. **loop**
2. **if** $\mathtt{A} = \emptyset$ **then** $\mathtt{A} := \{q\}$, $\mathtt{dg} := \{q\}$, $\mathtt{in} := \emptyset$, $\mathtt{rL}' := \mathtt{rankedList}$, initialise all pointers ptr_i^r to 0
3. **for all** intensional predicates p **do** $\mathtt{exp}(p) = \mathtt{false}$ **endfor**
 fi
4. **select** $p \in \mathtt{A}$, $\mathtt{A} := \mathtt{A} \setminus \{p\}$, $\mathtt{dg} := \mathtt{dg} \cup \mathtt{s}(p)$
5. $\langle \mathbf{t}, s \rangle := getNextTuple(p)$
6. **if** $\langle \mathbf{t}, s \rangle \neq \mathtt{null}$ **then** $\mathtt{rankedList}(p) := \mathtt{rankedList}(p) \cup \{\langle \mathbf{t}, s \rangle\}$, $\mathtt{A} := \mathtt{A} \cup (\mathtt{p}(p) \cap \mathtt{dg})$ **fi**
7. **if not** $\mathtt{exp}(p)$ **then** $\mathtt{exp}(p) = \mathtt{true}$, $\mathtt{A} := \mathtt{A} \cup (\mathtt{s}(p) \setminus \mathtt{in})$, $\mathtt{in} := \mathtt{in} \cup \mathtt{s}(p)$ **fi**
8. Update threshold δ
9. **until** ($\mathtt{rankedList}(q)$ does contain k top-ranked tuples with score greater or equal than query rule threshold)
  or ($\mathtt{rL}' = \mathtt{rankedList}$) **and** $\mathtt{A} = \emptyset$)
10. **return** top-k ranked tuples in $\mathtt{rankedList}(q)$

Procedure $getNextTuple(p)$
Input: intensional relation symbol p. Consider set of rules $\mathcal{R} = \{r | r : p(\mathbf{x}) \leftarrow f(A_1, \ldots, A_n) \in \mathcal{P}\}$
Output: Next instance of p together with the score
Init: Let p_i be the relation symbol occurring in A_i
1. **if** $\mathtt{Q}(p) \neq \emptyset$ **then**
  $\langle \mathbf{t}, s \rangle := getTop(\mathtt{Q}(p))$, remove $\langle \mathbf{t}, s \rangle$ from $\mathtt{Q}(p)$, **return** $\{\langle \mathbf{t}, s \rangle\}$ **fi**
 loop
2. **for all** $r \in \mathcal{R}$ **do**
3. Generate the set T of all new valid join tuples $\mathbf{t}$ for rule r,
 using tuples in $\mathtt{rankedList}(p_i)$ and pointers prt_i^r
4. **for all** $\mathbf{t} \in T$ **do**
5. $s :=$ compute the score of $p(\mathbf{t})$ using f;
6. **if** neither $\langle \mathbf{t}, s' \rangle \in \mathtt{rankedList}(p)$ nor $\langle \mathbf{t}, s' \rangle \in \mathtt{Q}(p)$ with $s \leq s'$ **then**
  insert $\langle \mathbf{t}, s \rangle$ into $\mathtt{Q}(p)$ **fi**
 endfor
 endfor
 until $\mathtt{Q}(p) \neq \emptyset$ or no new valid join tuple can be generated
7. **if** $\mathtt{Q}(p) \neq \emptyset$ **then** $\langle \mathbf{t}, s \rangle := getTop(\mathtt{Q}(p))$, remove $\langle \mathbf{t}, s \rangle$ from $\mathtt{Q}(p)$, **return** $\langle \mathbf{t}, s \rangle$
  else return $\mathtt{null}$ **fi**

some of these rules using the so far retrieved tuples for the relations occurring in φ_i. Note that the procedure is quite related to the *getNext* procedure described in [32]. This is not surprising as the list of atoms in a rule body may be seen as multiple joins together with a scoring function. However, w.r.t. [32], the computation of the threshold value is more involved in our case.

Let us describe the intuition behind the procedure. For the sake of illustrative purposes assume that there is an unique rule r associated to p of the form

$$p(x) \leftarrow p_1(x, y) \cdot p_2(y, z),$$

and that the currently retrieved tuples for p_i are stored in $\mathtt{rankedList}(p_i)$. The idea is as follows:

1. We incrementally generate new join combinations $\langle a, b, c \rangle$ from the tuples in $\mathtt{rankedList}(p_1)$ and $\mathtt{rankedList}(p_2)$ using square ripple join (see [32]):[3] that is, we alternatively access first $\mathtt{rankedList}(p_1)$ and then $\mathtt{rankedList}(p_2)$. We select the next unseen tuple in $\mathtt{rankedList}(p_1)$ and then build all join combinations

[3] $\langle a, b, c \rangle$ is a join combination for p if (a, b) and (b, c) are in $\mathtt{rankedList}(p_1)$ and $\mathtt{rankedList}(p_2)$, respectively.

with the tuples seen so far in $\texttt{rankedList}$ (p_2). Then we select the next unseen tuple in $\texttt{rankedList}(p_2)$ and then build all join combinations with the tuples seen so far in $\texttt{rankedList}(p_1)$ and so on until we find some valid join tuples ($\texttt{Q}(p) \neq \emptyset$). Fig. 5 illustrates this method: the x-axis (y-axis) represents the tuples in $\texttt{rankedList}$ (p_1) ($\texttt{rankedList}(p_2)$) and a point (i, j) denotes the ith (jth) tuple of the list $\texttt{rankedList}(p_1)$ ($\texttt{rankedList}(p_2)$). Hence, we generate tuple (i, j), starting from $(1, 1)$, and then check if (i, j) is a join combination for p. Specifically, in order to explore the tuple space $\texttt{rankedList}(p_1) \times \texttt{rankedList}(p_2)$, we generate the sets $S_1 = \{(1, 1)\}$, $S_2 = \{(2, 1)\}$, $S_3 = \{(1, 2), (2, 2)\}$, $S_4 = \{(3, 1), (3, 2)\}$ and $S_5 = \{(1, 3), (2, 3), (3, 3)\}$ and stop generating S_k as soon as we generated a set S_l that contains at least one join combination for p. Then for each join combination $(i, j) \in S_l$ we compute its score s and update $\texttt{Q}(p)$. The fact that we may add more than one join combination and its score to $\texttt{Q}(p)$ becomes evident from the fact that there may be more than one join combination for p in S_l (e.g., for S_4 we may have that $(1, 3)$ and $(3, 3)$ are join combinations for p, while $(2, 3)$ is not a join combination for p). Another method, although less effective, would consist in retrieving the join combinations $\langle a, b, c\rangle$ as an SQL statement over the lists $\texttt{rankedList}(p_i)$, i.e.

SELECT p_1.x, p_1.y, p_2.z
FROM $\texttt{rankedList}(p_1)$ p_1, $\texttt{rankedList}(p_1)$ p_2
WHERE p_1.y = p_2.y AND (exclude already processed tuples for rule r)

We need the condition "exclude already processed tuples for rule r" to guarantee that we do not join tuples twice for the rule r. This can be implemented using additional flags in $\texttt{rankedList}(p_i)$.

2. The join combinations for p and their scores will be put in the queue $\texttt{Q}(p)$ and the top-1 ranked one is returned. Note that it is possible that $|\texttt{Q}(p)| \geq 2$ (i.e., $\texttt{Q}(p)$ contains more than one join combination for p) and, thus, in the next call of $getNextTuple(p)$, we may avoid to generate new join combinations for p and it suffices to get the next top-ranked tuple from $\texttt{Q}(p)$.

In summary, $getNextTuple(p)$ works as follows. In step 1, whenever we already have some join combinations for p in the queue $\texttt{Q}(p)$ (obtained by a previous call) then we just return the top-ranked one. In step 2, we take into account all rules r having p in its head. For each rule r, in step 3 we try to generate join combinations for p, involving all seen tuples of the relations occurring in the rule body of r. For each join combination we compute its score (step 5). We put the results on the queue $\texttt{Q}(p)$ (step 6) and return the top-ranked one. As $\texttt{Q}(p)$ may still contain answers for p, the next time we ask for a next tuple with respect to p, we access $\texttt{Q}(p)$ directly (step 1).

Remark 6. Note that for each rule r we compute a join that is independent from any another join computed for another rule $r' \neq r$. To do so, for every rule r and predicate p_i in the rule body of r, we will maintain a pointer ptr_i^r to the last seen tuple in the ranked list of p_i with respect to the join computation of rule r. For instance, given rules

$$r_1 : q(x) \leftarrow p_1(x), \quad r_2 : p(x) \leftarrow p_1(x) \cdot p_2(x),$$

where the facts component is

$recId$	p_1		p_2	
1	a	1.0	b	0.8
2	b	0.9	e	0.6
3	c	0.8	a	0.4
⋮	⋮	⋮	⋮	⋮

table p_1 will have pointers $ptr_1^{r_1}$ and $ptr_1^{r_2}$, while table p_2 will have pointers $ptr_2^{r_2}$. So, the first answer computed via join for r_1 will be $\langle a, 1.0\rangle$ and then $ptr_1^{r_1} = 1$, while the first answer computed via join for r_2 will be $\langle b, 0.72\rangle$ and then $ptr_1^{r_2} = 2$ and $ptr_2^{r_2} = 1$. The next answer for r_1 will be $\langle b, 0.9\rangle$ and then $ptr_1^{r_1} = 2$, while the next answer for r_2 will be $\langle a, 0.4\rangle$ and then $ptr_1^{r_2} = ptr_2^{r_2} = 3$.

Example 8. Let us illustrate the behaviour of *getNextTuple* (see Table 2) with the following simple example. Consider the rule

$$r : p(x, z) \leftarrow p_1(x, y) \cdot p_2(y, z).$$

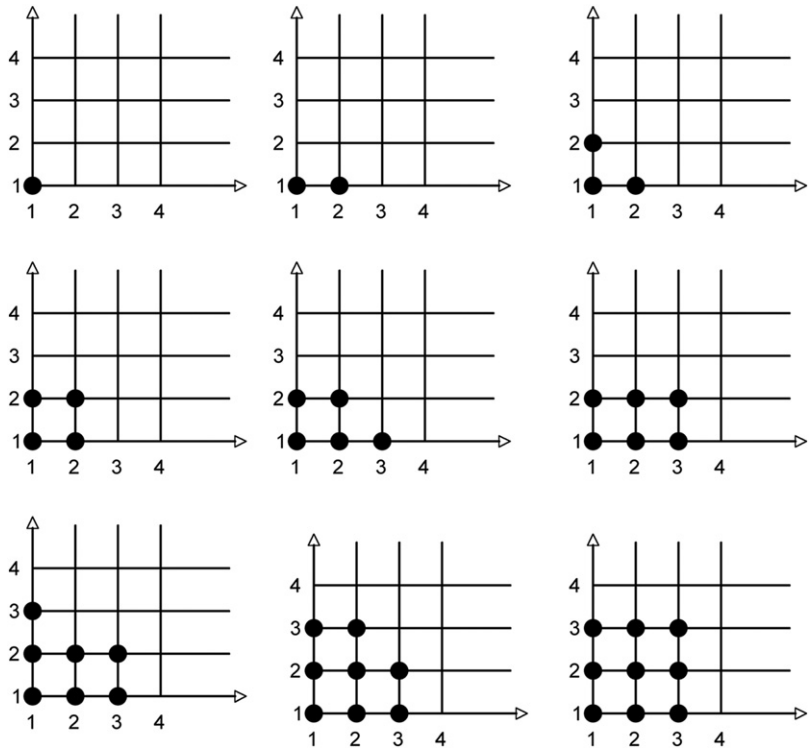

Fig. 5. Exploration of the tuple space $\texttt{rankedList}(p_1) \times \texttt{rankedList}(p_2)$: the generation of the first nine tuples selected from $\texttt{rankedList}(p_1)$ and $\texttt{rankedList}(p_2)$ to be checked for being join combinations for p.

Assume that actually $\texttt{rankedList}(p_1)$ and $\texttt{rankedList}(p_2)$ contain the following tuples:

recId	$\texttt{rankedList}(p_1)$		$\texttt{rankedList}(p_2)$	
1	*a b*	1.0	*m h*	0.95
2	*e f*	0.9	*m j*	0.85
3	*l m*	0.8	*f k*	0.75
4	*c d*	0.7	*m n*	0.65
5	*o p*	0.6	*p q*	0.55

with $\texttt{rankedList}(p_i).j$ we denote the jth tuple in $\texttt{rankedList}(p_i)$. For each list we will have a pointer ptr_i^r to the last tuple in $\texttt{rankedList}(p_i)$ seen so far with respect to rule r. Let us see how the first call to $getNextTuple(p)$ works.

At the beginning $ptr_1^r = ptr_2^r = 0$. As at step 1. $\mathtt{Q}(p) = \emptyset$, we need to generate new join tuples in step 2: we get the first tuple from $\texttt{rankedList}(p_1)$, i.e. $\texttt{rankedList}(p_1).1$, which is $\langle a, b, 1.0\rangle$. Now, we check if there is a join combination between $\langle a, b, 1.0\rangle$ and all tuples in $\texttt{rankedList}(p_2)$ seen so far, i.e. tuples in $X_2 = \{\texttt{rankedList}(p_2).j | 1 \leq j \leq ptr_2^r = 0\} = \emptyset$ and, thus, we cannot generate a new join combination for p.

We switch list, and access the next tuple from $\texttt{rankedList}(p_2)$, i.e. $\langle m, h, 0.95\rangle$ and check if there is a join combination for p between it and all tuples in $\texttt{rankedList}(p_1)$ seen so far, i.e. tuples in $X_1 = \{\texttt{rankedList}(p_1).j | 1 \leq j \leq ptr_1^r = 1\} = \{\langle a, b, 1.0\rangle\}$. Still, we do not have yet a join combination for p.

We switch list again, and access the next tuple from $\texttt{rankedList}(p_1)$, that is, $\langle e, f, 0.9\rangle$ and check if there is a join combination for p between $\langle e, f, 0.9\rangle$ and all tuples in $\texttt{rankedList}(p_2)$ seen so far, i.e. tuples in $X_2 = \{\texttt{rankedList}(p_2).j | 1 \leq j \leq ptr_2^r = 1\} = \{\langle m, h, 0.95\rangle\}$. Still, we do not have yet a join combination for p.

We switch list, and access the next tuple from $\texttt{rankedList}(p_2)$, i.e. $\langle m, j, 0.85\rangle$ and check if there is a join combination for p between it and all tuples in $\texttt{rankedList}(p_1)$ seen so far, that is, tuples in $X_1 = \{\texttt{rankedList}(p_1).j | 1 \leq j \leq ptr_1^r = 2\} = \{\langle a, b, 1.0\rangle, \langle e, f, 0.9\rangle\}$. Still, we do not have yet a join combination for p.

Now, we switch list again, and access the next tuple from $\texttt{rankedList}(p_1)$, i.e. $\langle l, m, 0.8\rangle$ and check if there is a join combination for p between it and all tuples in $\texttt{rankedList}(p_2)$ seen so far, that is, tuples in $X_2 = \{\texttt{rankedList}(p_2).j | 1 \leq j \leq ptr_2^r = 2\} = \{\langle m, h, 0.95\rangle, \langle m, j, 0.85\rangle\}$. Now, we get two join combinations for p, namely $\langle l, m\rangle \times \langle m, h\rangle$ and $\langle l, m\rangle \times \langle m, j\rangle$, and we exit step 3. In steps 4–6, we compute the score of these join combinations according to the score combination function and insert the newly generated join combinations $\langle l, h, 0.76\rangle$ and $\langle l, j, 0.68\rangle$ in $\texttt{Q}(p)$ (as no better result for $\langle l, h\rangle$ and $\langle l, j\rangle$ is in $\texttt{Q}(p)$). Finally, in step 7, we return $\langle l, h, 0.76\rangle$ as the result of $getNextTuple(p)$ and remove $\langle l, h, 0.76\rangle$ from $\texttt{Q}(p)$.

We make the following additional notes:

1. so far, $ptr_1^r = 3$ and $ptr_2^r = 2$;
2. in the second call of $getNextTuple(p)$, we will return $\langle l, j, 0.68\rangle$ as $\texttt{Q}(p)$ is still non-empty, and remove this tuple from $\texttt{Q}(p)$ (which makes it empty);
3. in the third call of $getNextTuple(p)$, we need to generate new join combinations for p, by (i) getting the next tuple from $\texttt{rankedList}(p_2)$, that is, $\langle f, k, 0.75\rangle$; (ii) checking if there is a join combination for p between $\langle f, k, 0.75\rangle$ and all tuples in $\texttt{rankedList}(p_1)$ seen so far, i.e. tuples in $X_1 = \{\texttt{rankedList}(p_1).j | 1 \leq j \leq ptr_1^r = 3\} = \{\langle a, b, 1.0\rangle, \langle e, f, 0.9\rangle, \langle l, m, 0.9\rangle\}$, generating $\langle e, k, 0.675\rangle$; (iii) inserting it into $\texttt{Q}(p)$; and (iv) returning $\langle e, f, 0.675\rangle$ and removing it from $\texttt{Q}(p)$ (which makes it empty);
4. similarly, in the fourth call of $getNextTuple(p)$, we return $\langle l, n, 0.52\rangle$, and have $ptr_1^r = ptr_2^r = 4$;
5. finally, in the fifth call of $getNextTuple(p)$, we return $\langle o, q, 0.33\rangle$, and have $ptr_1^r = ptr_2^r = 5$. □

As we have anticipated, a threshold will be used to determine when we can stop retrieving tuples. Indeed, the threshold determines when any newly retrieved tuple for q scores lower than the current top-k and, thus, cannot modify the top-k ranking (step 9). So, step 1 loops until we do not have k answers greater or equal than the threshold or, two successive loops do not modify the current set of answers and the queue $\texttt{A}$ becomes empty (step 9).[4] Steps 2 and 3 initialise the active set of relations. In step 4, we select a relation symbol to be processed. In step 5, we retrieve the next answer for p. If a new answer has been retrieved (step 6, $\langle \mathbf{t}, s\rangle \neq \texttt{null}$) then we update the current answer set $\texttt{rankedList}(p)$ and add all relations p_j, that directly depend on p, to the queue $\texttt{A}$. In step 7, we put once all intensional relation symbols appearing in rule bodies of rules having p in its head into the active set for further processing.

Remark 7. Before explaining the threshold computation mechanism, let us make the following observation. Let us assume for a moment that we do not consider the threshold mechanism in the computation of the *TopAnswers* procedure, i.e. we set initially the value to $\delta = 2$ and do not execute step 8. Hence, the threshold does not influence the stopping condition in step 9. Under this condition, please note that the *TopAnswers* will retrieve one by one *all* answers to a query.

Also note that the termination of the algorithm is guaranteed by the finiteness of the knowledge base, the finiteness of the truth set, and the monotonicity of the score combination functions: each tuple may enter a ranked list at most $h = |\mathcal{L}| - 1$ times and we stop as soon as two successive loops (steps 1–9) in *TopAnswers* do not change the ranked lists and the queue $\texttt{A}$ becomes empty.[5]

Now, as we do not want to evaluate all answers, but just to find the top-k, as already anticipated we use a threshold-based method, which we describe next. First we show how to determine the threshold if the intensional part contains the query rule only, and then extend it to the general case.

3.3. Querying an extensional knowledge base

So, let us assume that we have a knowledge base in which the rule component consists of one rule only (the query rule) of the form

$$r : q(\mathbf{x}) \leftarrow f(p_1, p_2, \ldots, p_n),$$

[4] Clearly, if we have already top-k answers greater or equal than the threshold, we can stop. The other condition is more involved and is explained in Remark 8, after Example 14.

[5] Step 9, ($\texttt{rL}' = \texttt{rankedList}$) **and** $\texttt{A} = \emptyset$.

where all p_i are extensional predicates. Example 8 is such a case. In this case, the threshold δ is determined as in [32], which we illustrate next.

Let $\mathbf{t}_i^r$ be the last tuple seen in $\texttt{rankedList}(p_i)$ so far with respect to rule r, while let $\hat{\mathbf{t}}_i$ be the top ranked one in $\texttt{rankedList}(p_i)$. With $\mathbf{t}.score$ we indicate the score of tuple $\mathbf{t}$.[6] Then we define δ^r as the maximum of the following n values:

$$\begin{aligned}
\delta_1^r &= f(\mathbf{t}_1^r.score, \hat{\mathbf{t}}_2.score, \ldots, \hat{\mathbf{t}}_n.score)\\
\delta_2^r &= f(\hat{\mathbf{t}}_1.score, \mathbf{t}_2^r.score, \ldots, \hat{\mathbf{t}}_n.score)\\
&\vdots\ \vdots\ \vdots\\
\Delta_n^r &= f(\hat{\mathbf{t}}_1.score, \hat{\mathbf{t}}_2.score, \ldots, \mathbf{t}_n^r.score).
\end{aligned}$$

Finally, we define the threshold to be used in the *TopAnswer* procedure as $\delta = \delta^r$. For instance, for

$$q(x) \leftarrow p_1(x, y) \cdot p_2(y, z),$$

we have

$$\delta_1^r = \mathbf{t}_1^r.score \cdot \hat{\mathbf{t}}_2.score, \quad \delta_2^r = \hat{\mathbf{t}}_1.score \cdot \mathbf{t}_2^r.score, \quad \delta^r = \max(\delta_1^r, \delta_2^r).$$

Example 9 (*Example 8 cont.*). After the first call of $getNextTuple(p)$ we have that $ptr_1^r = 3$ and $ptr_2^r = 2$, and $\hat{\mathbf{t}}_i = \texttt{rankedList}(p_i).1$, while $\mathbf{t}_i^r = \texttt{rankedList}(p_i).ptr_i^r$, that is

$\hat{\mathbf{t}}_1$	$\mathbf{t}_1^r$	$\hat{\mathbf{t}}_2$	$\mathbf{t}_2^r$
$\langle a, b, 1.0\rangle$	$\langle l, m, 0.8\rangle$	$\langle m, h, 0.95\rangle$	$\langle m, j, 0.85\rangle$

and, thus

$$\delta_1^r = 0.8 \cdot 0.95 = 0.76, \quad \delta_2^r = 1.0 \cdot 0.85 = 0.85, \quad \delta^r = \max(0.76, 0.85) = 0.85.$$

After the second call to $getNextTuple(p)$, the threshold does not change as we get the next tuple directly from $\texttt{Q}(p)$.

After the third call to $getNextTuple(p)$, we have $ptr_1^r = 3$, $ptr_2^r = 3$, $\hat{\mathbf{t}}_i = \texttt{rankedList}(p_i).1$, while $\mathbf{t}_i^r = \texttt{rankedList}(p_i).ptr_i^r$, that is

$\hat{\mathbf{t}}_1$	$\mathbf{t}_1^r$	$\hat{\mathbf{t}}_2$	$\mathbf{t}_2^r$
$\langle a, b, 1.0\rangle$	$\langle l, m, 0.8\rangle$	$\langle m, h, 0.95\rangle$	$\langle f, k, 0.75\rangle$

and, thus

$$\delta_1^r = 0.8 \cdot 0.95 = 0.76, \quad \delta_2^r = 1.0 \cdot 0.75 = 0.75, \quad \delta^r = \max(0.76, 0.75) = 0.76.$$

After the fourth call to $getNextTuple(p)$, we have $ptr_1^r = ptr_2^r = 4$ and, thus

$$\delta_1^r = 0.7 \cdot 0.95 = 0.665, \quad \delta_2^r = 1.0 \cdot 0.65 = 0.65, \quad \delta^r = \max(0.665, 0.65) = 0.665,$$

while after the fifth call to $getNextTuple(p)$, we have $ptr_1^r = ptr_2^r = 5$ and, thus

$$\delta_1^r = 0.6 \cdot 0.95 = 0.57, \quad \delta_2^r = 1.0 \cdot 0.55 = 0.55, \quad \delta^r = \max(0.57, 0.55) = 0.57. \quad \square$$

The important fact is now that whenever we consider a new join combination for rule r, its score will be *less or equal than* δ^r. Indeed, if we consider a new join tuple using the next unseen tuple from $\texttt{rankedList}(p_1)$ and a seen tuple in $\texttt{rankedList}(p_2)$, its score will be less or equal than δ_1^r, while if we consider a new join tuple using the next unseen

[6] If no tuple has been yet seen in p_i, then $\mathbf{t}.score = 1$ is assumed.

tuple from $\texttt{rankedList}(p_2)$ and a seen tuple in $\texttt{rankedList}(p_1)$, its score will be less or equal than δ_2^r. Therefore, overall the score will be less or equal than δ^r.

Example 10 (*Example 9 cont.*). In the second call to $getNextTuple(p)$, we get $\langle l, j, 0.68\rangle$ and $0.68 \leq 0.85 = \delta^r$, in the third call to $getNextTuple(p)$, we get $\langle e, f, 0.675\rangle$ and $0.675 \leq 0.85 = \delta^r$, in the fourth call to $getNextTuple(p)$, we get $\langle l, n, 0.52\rangle$ and $0.52 \leq 0.76 = \delta^r$ (note that after the third call, $\delta^r = 0.76$), while the fifth call to $getNextTuple(p)$, we get $\langle o, q, 0.33\rangle$ and $0.33 \leq 0.665 = \delta^r$ (after the fourth call, $\delta^r = 0.665$). □

As a consequence, whenever we have top-k answers for q with score greater or equal than δ^r, we can stop the retrieval process (see step 9 of *TopAnswers*).

Example 11 (*Example 10 cont.*). After the fourth call to $getNextTuple(p)$

$$\texttt{rankedList}(p) = [\langle l, h, 0.76\rangle, \langle l, j, 0.68\rangle, \langle e, k, 0.675\rangle, \langle l, n, 0.52\rangle],$$

and $\delta^r = 0.665$. Hence, if we are looking for top-3 answers to p then, as we have top-3 answers for p with score above δ^r, we can stop. Any newly retrieved tuple for p, e.g. tuple $\langle o, q, 0.33\rangle$, will have a score less or equal than δ^r. □

This property can be generalised to n-ary joins (see [32, Theorem 4.2.1]). We have the following result:

Proposition 2. *For a knowledge base in which the rule component consists of one rule r only* (*the query rule*) *of the form* $r : q(\mathbf{x}) \leftarrow f(p_1, p_2, \ldots, p_n)$, *where all* p_i *are extensional predicates*, *then the threshold-based method correctly reports the top-k results ordered by the score*.

Proof. For simplicity, we consider the case of score combination function f having two arguments. The proof can easily be extended to cover the n input case. So, let

$$r : q(\mathbf{x}) \leftarrow f(p_1, p_2)$$

be our query rule. As p_i are extensional, $\texttt{rankedList}(p_i)$ contains all tuples of p_i in decreasing order of score.

Assume that the algorithm halts after d sorted accesses to each $\texttt{rankedList}(p_i)$ and that $getNextTuple(q)$ reports a valid join combination $\mathbf{t} = \langle \mathbf{t}_1^i, \mathbf{t}_2^j\rangle$ for the query rule body, where $\mathbf{t}_1^i$ is the ith tuple in $\texttt{rankedList}(p_1)$ and $\mathbf{t}_2^j$ is the jth tuple in $\texttt{rankedList}(p_2)$. Since the algorithm halts (at depth d), we know that by definition $\delta = \delta^r \leq \mathbf{t}.score$, where δ^r is the maximum of [7]

$$\delta_1^r = f(\mathbf{t}_1^1.score, \mathbf{t}_2^d.score), \quad \delta_2^r = f(\mathbf{t}_1^d.score, \mathbf{t}_2^1.score).$$

Now, assume that there is yet another join combination $\bar{\mathbf{t}} = \langle \bar{\mathbf{t}}_1^l, \bar{\mathbf{t}}_2^m\rangle$ for the query rule body such that $\bar{\mathbf{t}}.score > \mathbf{t}.score$. That implies $\bar{\mathbf{t}}.score > \delta^r$, i.e.

$$f(\bar{\mathbf{t}}_1^l.score, \bar{\mathbf{t}}_2^m.score) > f(\mathbf{t}_1^1.score, \mathbf{t}_2^d.score), \tag{5}$$

$$f(\bar{\mathbf{t}}_1^l.score, \bar{\mathbf{t}}_2^m.score) > f(\mathbf{t}_1^d.score, \mathbf{t}_2^1.score). \tag{6}$$

Since each tuple in $\texttt{rankedList}(p_1)$ is ranked in decreasing order with respect to the score, $\bar{\mathbf{t}}_1^l.score \leq \mathbf{t}_1^1.score$. Therefore, $\bar{\mathbf{t}}_2^m.score$ has to be strictly greater than $\mathbf{t}_2^d.score$, as otherwise inequality (5) will not hold because of the monotonicity of the function f. We conclude that $\bar{\mathbf{t}}_2^m$ must be ranked before $\mathbf{t}_2^d$ in $\texttt{rankedList}(p_2)$, i.e.

$$m < d. \tag{7}$$

[7] Recall that $\mathbf{t}_i^1$ is the first tuple in $\texttt{rankedList}(p_i)$, while $\mathbf{t}_i^d$ is the last accessed (seen) one in $\texttt{rankedList}(p_i)$.

Using the same analogy, we have that $\bar{\mathbf{t}}_2^m.score < \mathbf{t}_2^1.score$ and, thus, $\bar{\mathbf{t}}_1^l.score$ must be strictly greater than $\mathbf{t}_1^d.score$ as otherwise inequality (6) will not hold because of the monotonicity of the function f. We conclude that $\bar{\mathbf{t}}_1^l$ must be ranked before $\mathbf{t}_1^d$ in $\texttt{rankedList}(p_1)$, i.e.

$$l < d. \tag{8}$$

From (7) and (8), the join tuple $\bar{\mathbf{t}} = \langle \bar{\mathbf{t}}_1^l, \bar{\mathbf{t}}_2^m \rangle$ must have been produced by the algorithm. Therefore there cannot be any tuple with score strictly greater than the score of any tuple reported by $TopAnswer(\mathcal{K}, q, k)$, which is not produced by the algorithm. □

Example 12. Assume that we have the following query rule:

$$r : q(x, z) \leftarrow \min(r_1(x, y), r_2(y, z)),$$

where q is the query relation and r_1, r_2 are extensional relations with tables

$recId$	r_1			r_2		
1	a	b	1.0	m	h	0.95
2	c	d	0.9	m	j	0.85
3	e	f	0.8	f	k	0.75
4	l	m	0.7	m	n	0.65
5	o	p	0.6	p	q	0.55
⋮	⋮	⋮	⋮	⋮	⋮	⋮

The table below reports a top-2 retrieval computation.

The first call of $getNextTuple(q)$ requires several alternative accesses to r_i before a tuple can be found ($\langle e, k, 0.75 \rangle$). In the second call we get immediately two candidate tuples. In the third call, as $\texttt{Q}(q) \neq \emptyset$ we get immediately the next candidate ($\langle l, j, 0.7 \rangle$). Finally, in the fourth call, we retrieve $\langle l, n, 0.65 \rangle$. As now $\texttt{rankedList}(q)$ contains two answers above the threshold of 0.7, we can stop and return $\{\langle e, k, 0.75 \rangle, \langle l, h, 0.7 \rangle\}$. Note that no new retrieved answer may have a score above 0.7. Indeed, the next one would be $\langle o, q, 0.55 \rangle$ and, thus, *not all tuples are processed* (which would be unfeasible in practice).

$TopAnswers$					
$Iter$	A	p	$\langle \mathbf{t}, s \rangle$	$\texttt{rankedList}(p)$	δ^r
1.	q	q	$\langle e, k, 0.75 \rangle$	$\langle e, k, 0.75 \rangle$	0.8
2.	q	q	$\langle l, h, 0.7 \rangle$	$\langle e, k, 0.75 \rangle, \langle l, h, 0.7 \rangle$	0.75
3.	q	q	$\langle l, j, 0.7 \rangle$	$\langle e, k, 0.75 \rangle, \langle l, h, 0.7 \rangle, \langle l, j, 0.7 \rangle$	0.75
4.	q	q	$\langle l, n, 0.65 \rangle$	$\langle e, k, 0.75 \rangle, \langle l, h, 0.7 \rangle, \langle l, j, 0.7 \rangle, \langle l, n, 0.65 \rangle$	0.7

□

So far, we have considered the case that there is one query rule only. We next show that we may easily extend our threshold-based method to the case that for a query $q(\mathbf{x})$, we have a knowledge base in which the rule component consists of $m \geq 1$ rules (the query rules) of the form

$$\begin{aligned} r_1 &: q(\mathbf{x}) \leftarrow f_1(p_1^1, p_2^1, \ldots, p_{n_1}^1) \\ &\vdots \qquad\qquad \vdots \\ r_m &: q(\mathbf{x}) \leftarrow f_m(p_1^m, p_2^m, \ldots, p_{n_m}^m), \end{aligned}$$

where all p_i^j are extensional predicates. In this case, let δ^{r_i} be the threshold for rule r_i, as computed previously. Now, let δ^q be

$$\delta^q = \max(\delta^{r_1}, \ldots, \delta^{r_m}).$$

Finally, we define $\delta = \delta^q$. Essentially, as there may be more that one rule with $q(\mathbf{x})$ as its head, we now consider the maximum among all rules' thresholds. It is easily verified that Proposition 2 holds for this more general case as well.

Proposition 3. *For a knowledge base in which the rule component consists of rules $r_1, \ldots, r_m$ having $q(\mathbf{x})$ as their head (the query rules), where the arguments of the rules are extensional predicates and $\delta = \delta^q = \max(\delta^{r_1}, \ldots, \delta^{r_m})$, then the threshold-based method correctly reports the top-k results ordered by the score.*

3.4. Querying an intensional knowledge base

We address now the general case. We recall that [51,76] address the top-k retrieval for general (recursive) fuzzy LPs and provide a threshold mechanism. Unfortunately, both of them illustrate incorrect algorithms in the sense that the top-k answers may not be the top-k ones as the following example illustrates. The reason is that the computed threshold does not guarantee that any successively computed answer has score less or equal than the threshold.

Example 13. Consider the knowledge base with rules

$$r_1 : q(x) \leftarrow b(x) \cdot c(x), \quad r_2 : b(x) \leftarrow d(x) \cdot e(x),$$

and facts

$recId$	c		d		e	
1	a	1.0	a	1.0	b	1.0
2	b	0.8	b	0.6	c	0.9
3	d	0.5	d	0.5	d	0.8
4	c	0.1	c	0.2	e	0.6
5	e	0.1	e	0.1	a	0.5

We are looking for the top-1 answer to q. As illustrated below, the algorithm returns $\langle b, 0.48\rangle$. Indeed, the threshold δ is computed in [51,76] as follows:

$$\delta_1 = \mathbf{t}_b^r.score \cdot \hat{\mathbf{t}}_c.score, \quad \delta_2 = \hat{\mathbf{t}}_b.score \cdot \mathbf{t}_c^r.score, \quad \delta = \max(\delta_1, \delta_2),$$

where $\mathbf{t}_p^r$ is the last tuple seen in $\texttt{rankedList}(p)$ and $\hat{\mathbf{t}}_p$ is the top ranked one in $\texttt{rankedList}(p)$, with $p \in \{b, c\}$. The computation is illustrated below:

	$TopAnswers$					
	$Iter$	A	p	$\langle \mathbf{t}, s\rangle$	$\texttt{rankedList}(p)$	δ
$Loop-1$						1.0
	1.	q	q	–	–	
	2.	b	b	$\langle b, 0.6\rangle$	$\langle b, 0.6\rangle$	
	3.	q	q	$\langle b, 0.48\rangle$	$\langle b, 0.48\rangle$	
$UpdateThreshold$						0.6
$Loop-2$						
	1.	q	q	–	–	
	2.	b	b	$\langle d, 0.4\rangle$	$\langle b, 0.6\rangle, \langle d, 0.4\rangle$	
	3.	q	q	$\langle b, 0.48\rangle$	$\langle d, 0.2\rangle$	
$UpdateThreshold$						0.4
$Stop, return \langle b, 0.48\rangle$						

So, the algorithm returns $\langle b, 0.48\rangle$, which is not correct. In fact, the top-1 answer should be $\langle a, 0.5\rangle$. □

The problem with the threshold computed as in the above example is that it does not guarantee that any successively retrieved answer has score less or equal than δ. Indeed, in Example 13, we stop with threshold $\delta = 0.4$, while the not yet retrieved answer a has score $0.5 > \delta$.

The solution we propose here is that we need to take into account a threshold for each intensional predicate related to the query as they may depend from each other. For instance, in Example 13, the threshold for q will depend on the threshold for b.

Specifically, let us consider a knowledge base in which an intensional predicate is head of *one rule only* (we address the case p is head of more than one rule later on, similarly as we did in Section 3.3). So, for an intensional predicate p heading exactly one rule r

$$r : p(\mathbf{x}) \leftarrow f(p_1(\mathbf{z}_1), \ldots, p_n(\mathbf{z}_n)),$$

we consider a threshold variable δ^p. With $r.\mathbf{t}_{p_i}$ ($\hat{r}.\mathbf{t}_{p_i}$) we denote the last tuple seen (the top ranked one) in $\texttt{rankedList}(p_i)$ with respect to rule r. We assume that by default $\hat{r}.\mathbf{t}_{p_i}.score = 1$ if no tuple is $\texttt{rankedList}(p_i)$.

Note that we have to record the last tuple seen and the top ranked one for each rule r, as their rule body are independent joins. For an intensional predicate p_i, we define

$$p_i^{\top} = \max(\delta^{p_i}, \hat{r}.\mathbf{t}_{p_i}.score), \quad p_i^{\perp} = \delta^{p_i},$$

while if p_i is an extensional predicate, we define

$$p_i^{\top} = \hat{r}.\mathbf{t}_{p_i}.score, \quad p_i^{\perp} = r.\mathbf{t}_{p_i}.score.$$

Now, for each rule r we consider the equation $\delta(r)$

$$\delta^p = \max(f(p_1^{\perp}, p_2^{\top}, \ldots, p_n^{\top}), f(p_1^{\top}, p_2^{\perp}, \ldots, p_n^{\top}), \ldots, f(p^{\top}, p^{\top}, \ldots, p_n^{\perp})). \tag{9}$$

For instance, for a rule

$$r : p(\mathbf{x}) \leftarrow f(p_1, p_2),$$

where p_1 is intensional and p_2 is extensional, we have

$$\delta^p = \max(f(\delta^{p_1}, \hat{r}.\mathbf{t}_{p_2}.score), f(\max(\delta^{p_1}, \hat{r}.\mathbf{t}_{p_1}.score), r.\mathbf{t}_{p_2}.score)). \tag{10}$$

Notice how now δ^p may depend on the valued of the threshold of some p_i occurring in its rule body. Essentially, $\mathbf{t}^r_{p_1}.score$ is now replaced with δ^{p_1}, while $\hat{\mathbf{t}}_{p_1}.score$ is now replaced with $\max(\delta^{p_i}, \hat{r}.\mathbf{t}_{p_i}.score)$.

For a knowledge base $\mathcal{K} = \langle \mathcal{F}, \mathcal{P} \rangle$, we consider the set Δ of all equations involving intensional predicates. Note that, if $\mathcal{K}$ has m intensional predicates, Δ consists of m equations and m variables. As all equations involve monotone functions only, as pointed out in Section 2, Δ has a minimal solution, denoted $\bar{\Delta}$. Finally, for a query $q(\mathbf{x})$, the threshold δ of the *TopAnswers* algorithm is defined as

$$\delta = \bar{\delta}^q,$$

where $\bar{\delta}^q$ is the solution to variable δ^q in the minimal solution $\bar{\Delta}$ of the set of equations Δ.

From a computational point of view, please note that $\bar{\Delta}$ and, thus, $\bar{\delta}^q$, can be computed iteratively as described in Eq. (2). Of course, if specific functions are involved only (e.g. linear functions), then better methods may be available to compute the minimal solution. Even more importantly, as ultimately we are interested in the threshold $\bar{\delta}^q$ only, we may apply the $Solve(\Delta, \{\delta^q\})$ algorithm to compute $\bar{\delta}^q$ in a more effective and query driven manner.

Example 14 (*Example 13 cont.*). Let us see how the new threshold function works on Example 13. Recall that the rules are

$$r_1 : q(x) \leftarrow b(x) \cdot c(x), \quad r_2 : b(x) \leftarrow d(x) \cdot e(x).$$

The equations for computing the thresholds are

$$\delta^q = \max(\delta^b \cdot \hat{r}_1.\mathbf{t}_c.score, \max(\delta^b, \hat{r}_1.\mathbf{t}_b.score) \cdot r_1.\mathbf{t}_c.score),$$

$$\delta^b = \max(r_2.\mathbf{t}_d.score \cdot \hat{r}_2.\mathbf{t}_e.score, \hat{r}_2.\mathbf{t}_d.score \cdot r_2.\mathbf{t}_e.score),$$

which can be rewritten as ($\hat{\mathbf{t}}_c.score = \hat{\mathbf{t}}_e.score = \hat{\mathbf{t}}_d.score = 1$, $\delta^b \geq \delta^b \cdot r.\mathbf{t}_c.score$, as the two rule bodies do not share any predicate, we may omit the prefix r_i)

$$\delta^q = \max(\delta^b, \hat{\mathbf{t}}_b.score \cdot \mathbf{t}_c.score), \quad \delta^b = \max(\mathbf{t}_d.score, \mathbf{t}_e.score).$$

The computation is illustrated below:

TopAnswers											
								scores			
	Iter	A	p	$\langle \mathbf{t}, s\rangle$	rankedList(p)	δ^q	δ^b	$\mathbf{t}_d$	$\mathbf{t}_e$	$\mathbf{t}_c$	$\hat{\mathbf{t}}_b$
Loop − 1						1.0	1.0				
	1.	q	q	–	–						
	2.	b	b	$\langle b, 0.6\rangle$	$\langle b, 0.6\rangle$						
	3.	q	q	$\langle b, 0.48\rangle$	$\langle b, 0.48\rangle$						
Threshold						1.0	1.0	0.6	0.1	0.8	0.6
Loop − 2											
	1.	q	q	–	–						
	2.	b	b	$\langle d, 0.4\rangle$	$\langle b, 0.6\rangle$, $\langle d, 0.4\rangle$						
	3.	q	q	$\langle d, 0.2\rangle$	$\langle b, 0.48\rangle$, $\langle d, 0.2\rangle$						
Threshold						0.8	0.8	0.5	0.8	0.5	0.6
Loop − 3											
	1.	q	q	–	–						
	2.	b	b	$\langle c, 0.18\rangle$	$\langle b, 0.6\rangle$, $\langle d, 0.4\rangle$, $\langle c, 0.18\rangle$						
	3.	q	q	$\langle c, 0.018\rangle$	$\langle b, 0.48\rangle$, $\langle d, 0.2\rangle$, $\langle c, 0.018\rangle$						
Threshold						0.6	0.6	0.2	0.6	0.1	0.6
Loop − 4											
	1.	q	q	–	–						
	2.	b	b	$\langle e, 0.06\rangle$	$\langle b, 0.6\rangle$, $\langle d, 0.4\rangle$, $\langle c, 0.18\rangle$, $\langle e, 0.06\rangle$						
	3.	q	q	$\langle e, 0.006\rangle$	$\langle b, 0.48\rangle$, $\langle d, 0.2\rangle$, $\langle c, 0.018\rangle$, $\langle e, 0.006\rangle$						
Threshold						0.6	0.6	0.1	0.6	0.1	0.6
Loop − 5											
	1.	q	q	–	–						
	2.	b	b	$\langle a, 0.5\rangle$	$\langle b, 0.6\rangle$, $\langle a, 0.5\rangle$, $\langle d, 0.4\rangle$, $\langle c, 0.18\rangle$, $\langle e, 0.06\rangle$						
	3.	q	q	–	$\langle b, 0.48\rangle$, $\langle d, 0.2\rangle$, $\langle c, 0.018\rangle$, $\langle e, 0.006\rangle$						
Threshold						0.5	0.5	0.1	0.5	0.1	0.6
Loop − 6											
	1.	q	q	$\langle a, 0.5\rangle$	$\langle a, 0.5\rangle$, $\langle b, 0.48\rangle$, $\langle d, 0.2\rangle$, $\langle c, 0.018\rangle$, $\langle e, 0.006\rangle$						
	2.	b	b	$\langle b, 0.6\rangle$	$\langle b, 0.6\rangle$, $\langle a, 0.5\rangle$, $\langle d, 0.4\rangle$, $\langle c, 0.18\rangle$, $\langle e, 0.06\rangle$						
	3.	q	q	$\langle a, 0.5\rangle$	$\langle a, 0.5\rangle$, $\langle b, 0.48\rangle$, $\langle d, 0.2\rangle$, $\langle c, 0.018\rangle$, $\langle e, 0.006\rangle$						
Threshold						0.5	0.5	0.1	0.6	0.1	0.6
Stop, *return* $\langle a, 0.5\rangle$											

So, the algorithm returns $\langle a, 0.5\rangle$, which is now correct.

Note that all generated answers for q at loop $i+1$ have score less or equal than the threshold δ^q, computed at the end of loop i. □

Remark 8 (*On stopping condition*). Let us point out that Example 14 also explains why we need to loop until two successive loops do not modify the current set of answers rankedList and the queue A becomes empty (step 9) of *TopAnswers*. Indeed, in Example 14, the queue A gets six times empty, while still rankedList has been changed. Each $Loop - i$ indicates that A became empty. Only when no further predicate symbol has to be elaborated ($\mathtt{A} = \emptyset$) and there has been no new answers for any predicate symbol $\mathtt{rL}' = \mathtt{rankedList}$, we can stop.

Note also that while the threshold mechanism guarantees that any successively retrieved tuple for predicate p_i has score less or equal than threshold δ^{p_i}, it does not guarantee that this tuple is ordered *below* any current pointer ptr_i^r in $\mathtt{rankedList}(p_i)$. For instance, in $Loop - 5$, tuple $\langle a, 0.5\rangle$ enters into $\mathtt{rankedList}(b)$ in second position, though the pointer $ptr_b^{r_2}$ is already at tuple $\langle e, 0.06\rangle$ and so $\langle a, 0.5\rangle$ will not be taken into account in $Loop - 5$ to get the new answer $\langle a, 0.5\rangle$ for q. So, A becomes empty but $\mathtt{rL}' \neq \mathtt{rankedList}$. We get the new answer $\langle a, 0.5\rangle$ for q in $Loop - 6$. We then stop after $Loop - 6$ as we have found top-1 answers above the threshold 0.5. If instead we wanted top-2 answers for q we had to continue as $\mathtt{A} = \emptyset$ and $\mathtt{rL}' \neq \mathtt{rankedList}$ and eventually stop after $Loop - 7$ ($\mathtt{A} = \emptyset$ and $\mathtt{rL}' = \mathtt{rankedList}$) getting the answers $\langle a, 0.5\rangle$ and $\langle b, 48\rangle$ for q.

In case an intensional relation p is in the head of more than one rule, e.g.

$$r_1 : p(\mathbf{x}) \leftarrow \varphi_1(\mathbf{x}, \mathbf{y}), \quad r_2 : p(\mathbf{x}) \leftarrow \varphi_2(\mathbf{x}, \mathbf{y}), \tag{11}$$

we proceed as at the end of Section 3.3 in which now δ^p is the maximum among the thresholds computed for rule r_1 and r_2 according to Eq. (9). That is, for two new relations p', p'', we define

$$\delta^p = \max(\delta^{p'}, \delta^{p''}), \tag{12}$$

where $\delta^{p'}$, $\delta^{p''}$ are the thresholds computed according to Eq. (9) for the two rules $p'(\mathbf{x}) \leftarrow \varphi_1(\mathbf{x}, \mathbf{y})$ and $p''(\mathbf{x}) \leftarrow \varphi_1(\mathbf{x}, \mathbf{y})$, respectively.

Example 15 (*Example 7 cont.*). Let us determine top-3 answers for $\mathtt{path}$. It can be verified that the equations for the threshold computation are

$$\delta^{\mathtt{path}} = \max(\delta^{\mathtt{path}'}, \delta^{\mathtt{path}''}), \quad \delta^{\mathtt{path}'} = r_1.\mathbf{t}_{\mathtt{edge}}.score,$$
$$\delta^{\mathtt{path}''} = \max(\min(\delta^{\mathtt{path}}, \hat{r}_2.\mathbf{t}_{\mathtt{edge}}.score), \min(\max(\delta^{\mathtt{path}}, \hat{r}_2.\mathbf{t}_{\mathtt{path}}.score), r_2.\mathbf{t}_{\mathtt{edge}}.score)).$$

The computation is shown below (we use the abbreviations $\mathtt{p}$, $\mathtt{p}'$, $\mathtt{p}''$, $\mathtt{e}$ for $\mathtt{path}$, $\mathtt{path}'$, $\mathtt{path}''$ and $\mathtt{edge}$, respectively).

TopAnswers											
								scores			
Iter	A	p	$\langle \mathbf{t}, s\rangle$	rankedList(p)	$\delta^{\mathtt{p}}$	$\delta^{\mathtt{p}'}$	$\delta^{\mathtt{p}''}$	$r_1.\mathbf{t}_{\mathtt{e}}$	$\hat{r}_2.\mathbf{t}_{\mathtt{p}}$	$r_2.\mathbf{t}_{\mathtt{e}}$	$\hat{r}_2.\mathbf{t}_{\mathtt{e}}$
Loop − 1					1.0	1.0	1.0	1.0	1.0	1.0	1.0
1.	p	p′	$\langle c, b, 0.6\rangle$	$\langle c, b, 0.6\rangle$	1.0	0.6	1.0	0.6	1.0	1.0	1.0
2.	p	p	$\langle a, c, 0.5\rangle$	$\langle c, b, 0.6\rangle$, $\langle a, c, 0.5\rangle$	0.5	0.5	0.5	0.5	0.6	0.4	0.6
3.	p	p	$\langle c, a, 0.4\rangle$	$\langle c, b, 0.6\rangle$, $\langle a, c, 0.5\rangle$, $\langle c, a, 0.4\rangle$	0.5	0.5	0.5	0.5	0.6	0.4	0.6
4.	p	p	$\langle a, b, 0.5\rangle$	$\langle c, b, 0.6\rangle$, $\langle a, c, 0.5\rangle$, $\langle a, b, 0.5\rangle$, $\langle c, a, 0.4\rangle$	0.4	0.4	0.4	0.4	0.6	0.4	0.6
Stop, return $\langle c, b, 0.6\rangle$, $\langle a, c, 0.5\rangle$, $\langle a, b, 0.5\rangle$											

Note that further answers for $\mathtt{path}$ have score not greater than the threshold $\delta^{\mathtt{path}} = 0.4$. □

Proposition 4. *Given a knowledge base, the generalised threshold-based method correctly reports the top-k results ordered by the score.*

Proof. First, let us note the following fact, which is easy to prove by induction on the number of iterations. □

Claim 1. *For any loop of* $TopAnswers(\mathcal{L}, \mathcal{K}, q, k)$, *for any predicate symbol* p, *if* $\langle \mathbf{c}, s\rangle \in \mathtt{rankedList}(p)$ *then* $s \leq M_{\mathcal{K}}(p(\mathbf{c}))$.

Essentially, it says that at any time during the computation of the top-k answers, the score of a tuple cannot be higher than the value of the tuple in the minimal model.

Now, we prove the following claim, which says that in any successive loop of the top-k retrieval procedure, the score of a retrieved tuple is less or equal than the current threshold.

Claim 2. *For any loop of* $TopAnswers(\mathcal{L}, \mathcal{K}, q, k)$, *for any intensional predicate symbol* p, *if* $\bar{\delta}^p$ *is the current threshold for* p, *then any retrieved tuple* $\langle \mathbf{c}, s\rangle$ *in a successive loop has a score less or equal than* $\bar{\delta}^p$.

Proof. Consider the current loop of the $TopAnswers(\mathcal{L}, \mathcal{K}, q, k)$ procedure and let $\bar{\delta}^p$ be the current threshold for any predicate p. Let us consider the following interpretation I: for any extensional predicate p

$$I(p(\mathbf{c})) = \begin{cases} s & \text{if } p(\langle \mathbf{c}, s\rangle) \in \mathcal{F} \\ \bot & \text{otherwise,} \end{cases}$$

while for any intensional predicate p

$$I(p(\mathbf{c})) = \begin{cases} \max\{s, \bar{\delta}^p\} & \text{if } \langle \mathbf{c}, s\rangle \in \mathtt{rankedList}(p) \\ \bar{\delta}^p & \text{otherwise.} \end{cases}$$

Note that for any predicate p, $I(p(\mathbf{c})) \leq p^{\top}$.

Now, let us show that I is a model of $\mathcal{K}$. If p is an extensional predicate symbol in $\mathcal{K}$, then by construction $I(p(\mathbf{c})) = s$ holds for all $p(\langle \mathbf{c}, s\rangle) \in \mathcal{F}$. On other words, I is a model of $\mathcal{F}$.

Otherwise, assume that p is an intensional predicate in $\mathcal{K}$. Consider a rule $r \in \mathcal{P}$ having p in its head, i.e. $p(\mathbf{x}) \leftarrow f(p_1(\mathbf{z}_1), \ldots, p_m(\mathbf{z}_m))$. We have to show that for any ground instance $p(\mathbf{c}) \leftarrow f(p_1(\mathbf{c}_1), \ldots, p_m(\mathbf{c}_m))$ of r, we have that

$$I(p(\mathbf{c})) \geq f(I(p_1(\mathbf{c}_1)), \ldots, I(p_m(\mathbf{c}_m))).$$

If for some $i \in \{1, \ldots, m\}$, $\langle \mathbf{c}_i, s_i \rangle \notin \texttt{rankedList}(p_i)$ then $I(p_i(\mathbf{c}_i)) = \bar{\delta}^{p_i}$. Then, by definition of $\bar{\delta}^{p}$, we have that

$$\begin{aligned} I(p(\mathbf{c})) \geq \bar{\delta}^{p} &\geq \max(f(p_1^{\perp}, p_2^{\top}, \ldots, p_m^{\top}), f(p_1^{\top}, p_2^{\perp}, \ldots, p_m^{\top}), \ldots, f(p^{\top}, p^{\top}, \ldots, p_m^{\perp})) \\ &\geq f(p_1^{\top}, \ldots, p_i^{\perp}, \ldots, p_m^{\top}) = f(p_1^{\top}, \ldots, \bar{\delta}^{p_i}, \ldots, p_m^{\top}) \geq f(I(p_1(\mathbf{c}_1)), \ldots, I(p_i(\mathbf{c}_i)), \ldots, I(p_m(\mathbf{c}_m))). \end{aligned}$$

Otherwise, assume that for all $i \in \{1, \ldots, m\}$, $\langle \mathbf{c}_i, s_i \rangle \in \texttt{rankedList}(p_i)$. If for some i, $s_i < \bar{\delta}^{p_i}$ then, reasoning as above we have that

$$I(p(\mathbf{c})) \geq \bar{\delta}^{p} \geq f(p_1^{\top}, \ldots, p_i^{\perp}, \ldots, p_m^{\top}) = f(p_1^{\top}, \ldots, \bar{\delta}^{p_i}, \ldots, p_m^{\top}) \geq f(I(p_1(\mathbf{c}_1)), \ldots, I(p_i(\mathbf{c}_i)), \ldots, I(p_m(\mathbf{c}_m))).$$

So, assume now that for all i, $s_i \geq \bar{\delta}^{p_i}$ and, thus, $I(p_i(\mathbf{c}_i)) = s_i$. If $\langle \mathbf{c}, s \rangle \in \texttt{rankedList}(p)$ then by definition $I(p(\mathbf{c})) \geq s$ holds. If, on the contrary, $\langle \mathbf{c}, s \rangle \notin \texttt{rankedList}(p)$, then necessarily at least for one tuple $\langle \mathbf{c}_i, s_i \rangle = p_i^{\perp}$, due to the way to retrieve tuples in $getNextTuple(p)$. Therefore

$$\begin{aligned} I(p(\mathbf{c})) \geq \bar{\delta}^{p} &\geq \max(f(p_1^{\perp}, p_2^{\top}, \ldots, p_n^{\top}), f(p_1^{\top}, p_2^{\perp}, \ldots, p_n^{\top}), \ldots, f(p^{\top}, p^{\top}, \ldots, p_n^{\perp})) \\ &\geq f(p_1^{\top}, \ldots, p_i^{\perp}, \ldots, p_n^{\top}) \geq f(s_1, \ldots, s_n) = f(I(p_1(\mathbf{c}_1), \ldots, I(p_n(\mathbf{c}_n)). \end{aligned}$$

Consequently, I is a model of $\mathcal{P}$ and, thus, of $\mathcal{K}$. It follows that $I(p(\mathbf{c})) \geq M_{\mathcal{K}}(p(\mathbf{c}))$ for any ground instance $p(\mathbf{c})$. Now, let $\langle \mathbf{c}, s \rangle$ be a tuple retrieved for an intensional predicate p in a successive loop. Then, using Claim 1, we have

$$I(p(\mathbf{c})) \geq M_{\mathcal{K}}(p(\mathbf{c})) \geq s. \tag{13}$$

If in the current loop $\langle \mathbf{c}, s' \rangle \notin \texttt{rankedList}(p)$ then $I(p(\mathbf{c})) = \bar{\delta}^{p}$. If instead, in the current loop there is $\langle \mathbf{c}, s' \rangle \in \texttt{rankedList}(p)$ then $s > s'$ (otherwise $\langle \mathbf{c}, s \rangle$ would not be retrieved for p). But, as $I(p(\mathbf{c})) = \max\{s', \bar{\delta}^{p}\} \geq s > s'$, it follows that $I(p(\mathbf{c})) = \bar{\delta}^{p}$. Therefore, in any case $I(p(\mathbf{c})) = \bar{\delta}^{p}$ and, thus, by (13), $\bar{\delta}^{p} \geq s$, which completes the proof of the claim. □

Let us now prove the proposition. At first, we show that the computed answers are answers. So, let $\langle \mathbf{c}, s \rangle \in \texttt{rankedList}(q)$ when $TopAnswers(\mathcal{L}, \mathcal{K}, q, k)$ stops. Now consider the interpretation I as in Claim 2. We have shown that I is a model of $\mathcal{K}$. Furthermore, as $TopAnswers(\mathcal{L}, \mathcal{K}, q, k)$ stops, we have also that $s \geq \bar{\delta}^{q}$ and, thus, $I(q(\mathbf{c})) = s$. By Claim 1, we have that $I(q(\mathbf{c})) = s \leq M_{\mathcal{K}}(q(\mathbf{c}))$. But, $M_{\mathcal{K}}$ is a minimal model and, thus, it cannot be $s < M_{\mathcal{K}}(q(\mathbf{c}))$. Therefore, $s = M_{\mathcal{K}}(q(\mathbf{c}))$ has to hold, i.e. $\langle \mathbf{c}, s \rangle$ is an answer for q.

Finally, suppose there is an answer $\langle \mathbf{c}', s' \rangle$ for q with score s' strictly higher than the score of some tuple $\langle \mathbf{c}, s \rangle$ returned by $TopAnswers(\mathcal{L}, \mathcal{K}, q, k)$. Therefore, $s' > s \geq \bar{\delta}^{q}$. But then, $\langle \mathbf{c}', s' \rangle$ would be retrieved in a successive loop (if the algorithm would not stop by the threshold mechanism—see also Remark 7). But this contradicts Claim 2 (as then $\bar{\delta}^{q} \geq s'$) and, thus, $TopAnswers(\mathcal{L}, \mathcal{K}, q, k)$ returns indeed the top-k answers. □

In Appendix B we sketch the computational complexity of the *TopAnswers* procedure.

4. Computing top-*k*–*n* answers

We have seen in Example 6 that it may happen that a nested top-k, top-n computation is needed, which we define as follows:

Top-k-n retrieval: Let $\mathcal{K} = \langle \mathcal{F}, \mathcal{P} \rangle$ be a logic program, let q be a query of the form

$$q(\mathbf{x}, \mathbf{y}_1, \mathbf{y}_2) \leftarrow \varphi(\mathbf{x}, \mathbf{y}_1, \mathbf{y}_2, \mathbf{y}_3),$$

and let n, k be two integers with $n \leq k$. Then the top-k–n tuples of q w.r.t. $\mathcal{K}$ is defined as the set

$$ans_{k,n}(q, \mathcal{K}) = \texttt{Top}_k\{\langle\langle \mathbf{c}, \mathbf{c}_1, \mathbf{c}_2 \rangle, s\rangle | \langle\langle \mathbf{c}_1, \mathbf{c}_2 \rangle, s\rangle \in \texttt{Top}_n\{\langle\langle \mathbf{c}_1', \mathbf{c}_2' \rangle, s'\rangle | s' = M_{\mathcal{K}}(q(\mathbf{c}, \mathbf{c}_1', \mathbf{c}_2'))\}\}.$$

Now, the question is: How do we compute the top-k–n answers? Fortunately, the procedure for top-k–n query answering is a slight extension of the top-k query answering procedure *TopAnswers*. Indeed, we can proceed exactly as for top-k, except that we always guarantee that there are no more than n tuples of the form (for a fixed $\mathbf{c}$)

$$\begin{aligned}&\langle\langle \mathbf{c}, \mathbf{c}'_1, \mathbf{c}''_1\rangle, s_1\rangle\\ &\langle\langle \mathbf{c}, \mathbf{c}'_2, \mathbf{c}''_2\rangle, s_2\rangle\\ &\qquad\vdots\\ &\langle\langle \mathbf{c}, \mathbf{c}'_l, \mathbf{c}''_l\rangle, s_l\rangle\end{aligned}$$

in the current ranked list $\texttt{rankedList}(q)$ greater or equal than the threshold δ^q. Exactly as for the top-k case, whenever we consider a new join combination of the form $\langle \mathbf{c}, \mathbf{c}''\rangle$, its score will be less or equal than δ^q. Therefore, as soon as we have n tuples of the form $\langle\langle \mathbf{c}, \mathbf{c}', \mathbf{c}''\rangle, s\rangle$ in $\texttt{rankedList}(q)$ with score greater or equal than δ^q, we do not need to generate new join tuples starting with $\mathbf{c}$ in step 2 of the $getNextTuple(q)$ procedure. Indeed any newly generated tuple starting with $\mathbf{c}$ will have a score less or equal than δ^q.

For instance, related to Example 6 (top-3-1 computation), assume that we already have

$$\langle\langle 34, 11\,500, 17\,000\rangle, 0.29\rangle$$

in $\texttt{rankedList}(\texttt{Match})$. As $0.29 \geq \delta^q = 0.29$, we do not generate any new tuple starting with 34 in the $getNextTuple(q)$ call, as e.g. tuple $\langle\langle 34, 10\,800, 17\,000\rangle, 0.25\rangle$ (even though it would be ranked better than $\langle\langle 455, 12\,200, 18\,000\rangle, 0.15\rangle$).

Finally, from a computational complexity point of view, the same results hold as for the top-k case, as checking the above condition does not affect the overall complexity.

5. Related work

While there are many works addressing the top-k problem for vague queries in databases (cf. [9,12,29,28,32,33,44, 43,54]), little is known for the corresponding problem in knowledge representation and reasoning. For instance, [85] considers non-recursive fuzzy logic programs in which the score combination function is a function of the score of the atoms in the body. The work [73] considers non-recursive fuzzy logic programs as well, though the score combination function is more expressive and may consider so-called expensive fuzzy relations (the score may depend on the value of an attribute, see [12]). However, a score combination function is allowed in the query rule only. We point out that in the case of non-recursive rules, we may rely on a query rewriting mechanism, which, given an initial query, rewrites it, using rules and/or axioms of the KB, into a set of new queries until no new query rule can be derived (this phase may require exponential time relative to the size of the KB, but is polynomial in the size of the facts). The obtained queries may then be submitted directly to a top-k retrieval database engine. The answers to each query are then merged using the disjunctive threshold algorithm given in [73]. The works [74,72] (see also [80]) address the top-k retrieval problem for the description logic *DL-Lite/DLR-Lite* [5,10], though recursion is allowed among the axioms. Again, the score combination function may consider expensive fuzzy relations. However, a score combination function is allowed in the query only. The combination with non-recursive logic programming rules, as described in this work, is straightforward. The work [79] shows an application of top-k retrieval to the case of multimedia information retrieval by relying on a fuzzy variant of *DLR-Lite*. In [76] we address the top-k retrieval for general (recursive) fuzzy LPs and is closest to this work. [51] slightly extends [76] as it allows also DLR-Lite axioms to occur and tries to rely as much as possible on current top-k database technology. However, as already pointed out, these two works exhibit incorrect algorithms, which has been corrected in this work. Here, additionally we show that we can smoothly extend the top-k–n problem. This latter problem has been shown to be fundamental in electronic Matchmaking [65,66]. [35] uses a threshold mechanism for reducing the amount of computation needed to answer a propositional query in a tabulation-based procedure for propositional multi-adjoint logic programs (strictly speaking, a language is subsumed by the one presented here) and, thus, does not address the top-k retrieval problem. [20,21,15,16] propose query answering procedures (for less expressive fuzzy logic programming languages than the one proposed here), which are based on unification to compute answers, but do not address the top-k retrieval problem. It is unclear yet whether unification-based query driven query answering procedures can be combined with a threshold mechanism in such a way to compute top-k answers.

6. Conclusions

The top-k and the top-k–n retrieval problem are important problems in logic-based languages for the Semantic Web. We have addressed this issue for a general fuzzy logic programming setting.

Other main topics for future work include: (i) can we extend our method to the non-finite truth-spaces? (ii) Can we apply similar ideas to non-monotonic LPs? (iii) How can we approach the top-k (and top-k–n) problem under a probabilistic setting, or more generally under uncertainty, possibly relying on emerging top-k retrieval systems for uncertain database management [67,69]?

Acknowledgements

Partially supported by the FEDER funds of the European Union and the Spanish Science Ministry project TIN09-14562-C05-01, and the Junta de Andalucía project FQM-5233. We are very grateful to the reviewers for their valuable and extensive comments on an early version of this paper.

Appendix A. Some references related to logic programming, uncertainty and imprecision

Below a list of references and the underlying imprecision and uncertainty theory in logic programming frameworks. The list of references is not intended by any means to be all-inclusive. The authors apologise both to the authors and with the readers for all the relevant works, which are not cited here. A much more extensive reference list can be found in [78].

Probability theory: [6,8,17,24,36,41,49,50,59,61,62,64,53];
Fuzzy set theory: [7,11,27,31,34,60,68,83,84,86,87];
Multi-valued logic: [19,18,25,30,37,35,40,38,39,42,45–48,52,55,57,56,70,71,77];
Possibilistic logic: [2,3,1,14,26,63].

Appendix B. Computational complexity

Solve: We address here the computational complexity of the *Solve* algorithm. Of course, the analysis depends on the data structures used. It is not our aim to look for the best bound, but only to provide an upper bound to illustrate roughly its behaviour. So, given a system $\mathcal{S} = \langle \mathcal{L}, V, \mathbf{f} \rangle$, let $h = |\mathcal{L}| - 1$ be the *height* of $\mathcal{L}$. We use a stack for `A` with constant insertion and extraction time. We encode variables in V as positive integers. We use an array `D` of length $|V|$ that for each variable $x \in V$ stores an object that contains

- the value $\mathtt{v}(x)$;
- a flag indicating if x is in `A`;
- a flag indicating if x is in `dg`;
- a flag indicating if x is in `in`;
- a flag indicating if x has been expanded, i.e. contains the value $\mathtt{exp}(x)$;
- the list $\mathtt{s}(x)$;
- the list $\mathtt{p}(x)$.

Let a_i be the arity of function f_i, let us assume that the c_i is the maximal cost of evaluating function f_i on its arguments and let p_i be $|\mathtt{p}(x_i)|$. Now, step 1 is $O(|V|)$. Let us determine the number of loops of step 2. As the height h of $\mathcal{L}$ is finite, observe that any variable is increasing in the $\leq$ order as it enters in the `A` list (step 5), except it enters due to step 6, which may happen only once. Therefore, each variable x_i will appear in `A` at most $a_i \cdot h + 1$ times, as a variable is only re-entered into `A` if one of its sons gets an increased value (which for each son only can happen h times), plus the additional entry due to step 6. So if l_i is the cost of steps 3–5 and k_i is the cost of step 6, then the worst-case complexity bounded by

$$\sum_{x_i \in V} (a_i h + 1) l_i + k_i .$$

Let us estimate l_i and k_i.

- Step 3 can be done in time bounded by a_i. Indeed, selecting x_i is done in constant time, while $\mathtt{dg} := \mathtt{dg} \cup \mathtt{s}(x_i)$ is implemented by updating $\mathtt{D}$ appropriately: namely, access to $\mathtt{D}(x_i)$, go through the list $\mathtt{s}(x_i)$ of successors x_j (there are a_i of them), and update the flag indicating that x_j is in $\mathtt{dg}$;
- Step 4 can be done in time bounded by c_i;
- Step 5 is bounded by p_i. Indeed, the operation $\mathtt{A} := \mathtt{A} \cup (\mathtt{p}(x_i) \cap \mathtt{dg})$ is implemented by updating $\mathtt{D}$ similarly to step 3: for any variable $x_j \in \mathtt{p}(x_i)$ (there are p_i of them), access to $\mathtt{D}(x_i)$ if the $\mathtt{dg}$ flag is 1 and the $\mathtt{A}$ flag is 0, update this latter to 1 and add x_j to $\mathtt{A}$;
- Step 6 is bounded by a_i. Indeed, the operations $\mathtt{A} := \mathtt{A} \cup (\mathtt{s}(x_i) \setminus \mathtt{in})$ and $\mathtt{in} := \mathtt{in} \cup \mathtt{s}(x_i)$ are implemented by updating $\mathtt{D}$ similarly to step 3, by going though the list $\mathtt{s}(x_i)$.

Therefore, l_i is bounded by $a_i + c_i + p_i$, while k_i is bounded by a_i and, thus, the worst-case complexity is bounded by

$$\sum_{x_i \in V} (a_i h + 1)(a_i + c_i + p_i) + a_i = \sum_{x_i \in V} a_i h(a_i + c_i + p_i) + 2a_i + c_i + p_i.$$

If the maximal values of a_i, c_i, p_i are bounded by a, c and p respectively, then the worst-case complexity is

$$O(|V|ah(a + c + p)), \tag{B.1}$$

which is

$$O(|V|ah(a + p)), \tag{B.2}$$

under the assumption that c is $O(a)$ (the functions can be evaluated in linear time w.r.t. the arguments) that in turn becomes

$$O(|V|^2 a^2 h) \tag{B.3}$$

in case p is $O(|V|)$.

We recall that, in case of non-finite truth spaces, the computation may not terminate after a finite number of steps (see Example 3). [22,70] illustrate some useful cases, where the termination is still guaranteed, e.g. for non-finite lattices. Eqs. (B.2) and (B.3) allow us also to give an immediate worst-case complexity result for answering about the truth of a ground atom A in the minimal model of $\mathcal{K} = \langle \mathcal{F}, \mathcal{P} \rangle$. Indeed, V is bijectively related to the Herbrand base $B_{\mathcal{K}}$ and, thus, it follows immediately that the worst-case complexity is $O(|B_{\mathcal{K}}|ah(a + p))$. Furthermore, by observing that $|B_{\mathcal{K}}|a$ is in $O(|\mathcal{K}^*|)$ we immediately have that the complexity is $O(|\mathcal{K}^*|h(a + p))$. As a consequence, for the classical truth set, we get a worst-case complexity of $O(|\mathcal{K}^*|(a + p)$, which may be reduced to $O(|\mathcal{K}^*|)$ as both a and p are usually negligible in size w.r.t. $|\mathcal{K}^*|$ (this is the same complexity as for classical Datalog [23]).

We are not going to look further into this as likely more efficient data structures may exists.

TopAnswers: Let us sketch the computational complexity. It parallels the one we have done for the *Solve* procedure. Let D_q be the set of intentional relation symbols that *depend* on the query relation symbol q. Of course, only relation symbols in D_q may appear in $\mathtt{A}$. We use essentially a similar data structure as for *Solve*. That is, we use an array $\mathtt{D}$ of length $|D_q|$ that for each predicate $p_i \in D_q$ stores an object that contains:

- $\mathtt{rankedList}(p_i)$;
- a flag indicating if p_i is in $\mathtt{A}$;
- a flag indicating if p_i is in $\mathtt{dg}$;
- a flag indicating if p_i is in $\mathtt{in}$;
- a flag indicating if p_i has been expanded, i.e. contains the value $\mathtt{exp}(p_i)$;
- the list $\mathtt{s}(p_i)$;
- the list $\mathtt{p}(p_i)$.

Let a_f be the arity of function f and a_p the arity of a predicate p. Let $\bar{a}_f = \max_f\{a_f\}$ and $\bar{a}_p = \max_p\{a_p\}$.

Let $\bar{s}_i = |\mathtt{s}(p_i)|$, $\bar{s} = \max\ s_i \leq |D_q|$, $\bar{p}_i = |\mathtt{p}(s_i)| \leq |D_q|$ and assume that the cost of evaluating a function f is bounded by its arity a_f adn, thus, by $\bar{a}_f$. Let r_i be the number of rules having in its head p_i.

For any rule $p(\mathbf{x}) \leftarrow f(\ldots)$, the number of instances of p is bounded by $|H_{\mathcal{K}}|^{a_p}$. So, $|\texttt{rankedList}(p)|$ is bounded by $|H_{\mathcal{K}}|^{\bar{a}_p}$. On the other hand, the number of ground instances of the rule body $f(\ldots)$ is bounded by $|H_{\mathcal{K}}|^{\bar{a}_p \cdot \bar{a}_f}$ (for any predicate occurring in f there are at most $|H_{\mathcal{K}}|^{\bar{a}_p}$ instances and there are at most $\bar{a}_f$ predicates occurring in f). So, for $\bar{g}_f = \bar{a}_p \cdot \bar{a}_f$, the number of groundings of any rule body is bounded by $|H_{\mathcal{K}}|^{\bar{g}_f}$.

Now, the cost of the initialisation phase is negligible. Steps 2 and 3 are $O(1)$. Step 4 is bounded by $\bar{s}_i$ as for *Solve*. Concerning step 6, updating the ranked list (maintaining the order) is $\log |H_{\mathcal{K}}|^{\bar{a}_p} = \bar{a}_p \log |H_{\mathcal{K}}|$, while the updating $\texttt{A}$ is as for *Solve* bounded by $\bar{p}_i$ and, thus, by $|D_q|$. Step 7 is bounded by $\bar{s}_i$, while step 9 is $O(1)$. Eventually, step 5 is bounded by $|H_{\mathcal{K}}|^{\bar{g}_f}$ (the number of ground instances of any rule body).

Let us now estimate the cost u of updating the thresholds. To this end we use the results for *Solve*. The number of variables $|V|$ is bounded by $|D_q| + \sum_{p_i \in D_q} r_i = \sum_{p_i \in D_q} (r_i + 1)$. The function defining δ^p has arity bounded by r_i (see Eq. (12)), while each function in the definition of each $\delta^{p'}$ coming from $p'(\mathbf{x}) \leftarrow f(\ldots)$ has arity a_f. Now let a be the maximum arity among all a_f, r be the maximum number of rules among all r_i and $\bar{a} = \max(a, r)$. Then $|V|$ is bounded by $|D_q|r$. From the *Solve* analysis, we have (see Eq. (B.2)) that the cost u of step 8 is $O(|V|\bar{a}h(\bar{a} + |D_q|))$

$$O(|D_q|r\bar{a}h(\bar{a} + |D_q|)).$$

Now, observe that any tuple's score is increasing as it enters in $\texttt{rankedList}$ and, thus, each p_i will appear in $\texttt{A}$ at most $|H_{\mathcal{K}}|^{\bar{g}_f} rh$ times.[8] Therefore, the number of loops is bounded by $|D_q||H_{\mathcal{K}}|^{\bar{g}_f} rh$.

By using Eq. (B.2), we get that *TopAnswers* has upper bound complexity[9]

$$O(|D_q||H_{\mathcal{K}}|^{\bar{g}_f} rh(\bar{s} + \bar{a}_p \log |H_{\mathcal{K}}| + |D_q| + u + |H_{\mathcal{K}}|^{\bar{g}_f})),$$

which is (assuming $|H_{\mathcal{K}}| > 1$)

$$O(|D_q||H_{\mathcal{K}}|^{\bar{g}_f} rh(|D_q|r\bar{a}h(\bar{a} + |D_q|) + |H_{\mathcal{K}}|^{\bar{g}_f})).$$

By observing that $|D_q||H_{\mathcal{K}}|^{\bar{g}_f} r$ has upper bound $|\mathcal{K}^*|$, $|D_q|r\bar{a}$ has upper bound $|\mathcal{K}|$, and that $|H_{\mathcal{K}}|^{\bar{g}_f}$ has upper bound $|\mathcal{K}^*|/(r \cdot |D_q|)$, we may as well rewrite the upper bound of the *TopAnswers* procedure as

$$O(|\mathcal{K}^*|h(|\mathcal{K}|h(\bar{a} + |D_q|) + |\mathcal{K}^*|/(r \cdot |D_q|))). \tag{B.4}$$

References

[1] T. Alsinet, L. Godo, A complete calculus for possibilistic logic programming with fuzzy propositional variables with fuzzy propositional variables, in: Proceedings of the 16th Conference in Uncertainty in Artificial Intelligence (UAI-00), Morgan Kaufmann, 2000, pp. 1–10.

[2] T. Alsinet, L. Godo, Towards an automated deduction system for first-order possibilistic logic programming with fuzzy constants, Int. J. Intell. Syst. 17 (9) (2002) 887–924.

[3] T. Alsinet, L. Godo, S. Sandri, On the semantics and automated deduction for PLFC, in: Proceedings of the 15th Annual Conference on Uncertainty in Artificial Intelligence (UAI-99), 1999.

[4] H.R. Andersen, Local Computation of Simultaneous Fixed-Points, Technical Report PB-420, DAIMI, 1992, see also ⟨http://www.itu.dk/people/hra/thesis.pdf⟩.

[5] in: F. Baader, D. Calvanese, D. McGuinness, D. Nardi, P.F. Patel-Schneider (Eds.), The Description Logic Handbook: Theory, Implementation, and Applications, Cambridge University Press, 2003.

[6] J.F. Baldwin, A theory of mass assignments for artificial intelligence, Lecture Notes in Computer Science, vol. 833, 1994, pp. 22–34.

[7] J.F. Baldwin, T.P. Martin, B.W. Pilsworth, Fril—Fuzzy and Evidential Reasoning in Artificial Intelligence, Research Studies Press Ltd., 1995.

[8] C. Baral, M. Gelfond, N. Rushton, Probabilistic reasoning with answer sets, in: Proceedings of the Seventh International Conference in Logic Programming and Nonmonotonic Reasoning (LPNMR-04), Lecture Notes in Artificial Intelligence, vol. 2923, Springer-Verlag, Fort Lauderdale, FL, USA, 2004, pp. 21–33.

[9] N. Bruno, S. Chaudhuri, L. Gravano, Top-k selection queries over relational databases: mapping strategies and performance evaluation, ACM Trans. Database Syst. 27 (2) (2002) 153–187.

[10] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, R. Rosati, Data complexity of query answering in description logics, in: Proceedings of the Tenth International Conference on Principles of Knowledge Representation and Reasoning (KR-06), 2006, pp. 260–270.

[11] T.H. Cao, Annotated fuzzy logic programs, Fuzzy Sets Syst. 113 (2) (2000) 277–298.

[8] There are at most $|H_{\mathcal{K}}|^{\bar{g}_f}$ instances of a rule having p_i in its head of which there are at most r rules.

[9] Note that we considered the cost $|H_{\mathcal{K}}|^{\bar{g}_f}$ of the *getNextTuple* procedure for each loop, which hardly may occur in practice. It is not clear if this even may happen in the worst case.

[12] K.C.-C. Chang, S. won Hwang, Minimal probing: supporting expensive predicates for top-*k* queries, in: SIGMOD Conference, 2002, pp. 346–357.
[14] C. Chesñevar, G. Simari, T. Alsinet, L. Godo, A logic programming framework for possibilistic argumentation with vague knowledge, in: Proceedings of the 20th Annual Conference on Uncertainty in Artificial Intelligence (UAI-04), AUAI Press, Arlington, VA, 2004, pp. 76–84.
[15] A. Chortaras, G.B. Stamou, A. Stafylopatis, Integrated query answering with weighted fuzzy rules, Ninth European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty (ECSQARU-07), Lecture Notes in Computer Science, vol. 4724, Springer-Verlag, 2007, pp. 767–778.
[16] A. Chortaras, G.B. Stamou, A. Stafylopatis, Top-down computation of the semantics of weighted fuzzy logic programs, in: Web Reasoning and Rule Systems, First International Conference (RR-07), 2007, pp. 364–366.
[17] C.V. Damásio, L.M. Pereira, Hybrid probabilistic logic programs as residuated logic programs, Studia Logica 72 (1) (2002) 113–138.
[18] C.V. Damásio, J. Medina, M. Ojeda-Aciego, Sorted multi-adjoint logic programs: termination results and applications, in: Proceedings of the Ninth European Conference on Logics in Artificial Intelligence (JELIA-04), Lecture Notes in Computer Science, vol. 3229, Springer Verlag, 2004, pp. 252–265.
[19] C.V. Damásio, L.M. Pereira, Sorted monotonic logic programs and their embeddings, in: Proceedings of the 10th International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems (IPMU-04), 2004, pp. 807–814.
[20] C.V. Damásio, M. Medina, J. Ojeda-Aciego, A tabulation procedure for first-order residuated logic programs, in: Proceedings of the IEEE World Congress on Computational Intelligence (Section Fuzzy Systems) (WCCI-06), 2006, pp. 9576–9583.
[21] C.V. Damásio, M. Medina, J. Ojeda-Aciego, A tabulation procedure for first-order residuated logic programs, in: Proceedings of the 11th International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems (IPMU-06), 2006.
[22] C.V. Damásio, M. Medina, J. Ojeda-Aciego, Termination of logic programs with imperfect information: applications and query procedure, J. Appl. Logic 7 (5) (2007) 435–458.
[23] E. Dantsin, T. Eiter, G. Gottlob, A. Voronkov, Complexity and expressive power of logic programming, ACM Comput. Surv. 33 (3) (2001) 374–425.
[24] A. Dekhtyar, M.I. Dekhtyar, Revisiting the semantics of interval probabilistic logic programs, Eighth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-05), Lecture Notes in Computer Science, vol. 3662, Springer-Verlag, 2005, pp. 330–342.
[25] M. Denecker, N. Pelov, M. Bruynooghe, Ultimate well-founded and stable semantics for logic programs with aggregates, in: P. Codognet (Ed.), Logic Programming, 17th International Conference, ICLP 2001, Paphos, Cyprus, November 26–December 1, 2001, Proceedings, Lecture Notes in Computer Science, vol. 2237, Springer, 2001, pp. 212–226.
[26] D. Dubois, J. Lang, H. Prade, Towards possibilistic logic programming, in: Proceedings of the Eighth International Conference on Logic Programming (ICLP-91), The MIT Press, 1991, pp. 581–595.
[27] R. Ebrahim, Fuzzy logic programming, Fuzzy Sets Syst. 117 (2) (2001) 215–230.
[28] R. Fagin, Combining fuzzy information: an overview, SIGMOD Rec. 31 (2) (2002) 109–118.
[29] R. Fagin, A. Lotem, M. Naor, Optimal aggregation algorithms for middleware, in: Symposium on Principles of Database Systems, 2001.
[30] M.C. Fitting, Fixpoint semantics for logic programming—a survey, Theoret. Comput. Sci. 21 (3) (2002) 25–51.
[31] D. Guller, Procedural semantics for fuzzy disjunctive programs, in: M. Baaz, A. Voronkov (Eds.), Logic for Programming, Artificial Intelligence, and Reasoning Ninth International Conference, LPAR 2002, Tbilisi, GA, October 14–18, 2002, Proceedings, Lecture Notes in Computer Science, vol. 2514, Springer, 2002, pp. 247–261.
[32] I.F. Ilyas, W.G. Aref, A.K. Elmagarmid, Supporting top-*k* join queries in relational databases, in: Proceedings of 29th International Conference on Very Large Data Bases (VLDB-03), 2003, pp. 754–765.
[33] I.F. Ilyas, W.G. Aref, A.K. Elmagarmid, H.G. Elmongui, R. Shah, J.S. Vitter, Adaptive rank-aware query optimization in relational databases, ACM Trans. Database Syst. 31 (4) (2006) 1257–1304.
[34] M. Ishizuka, N. Kanai, Prolog-ELF: incorporating fuzzy logic, in: Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI-85), Los Angeles, CA, 1985, pp. 701–703.
[35] P. Julián-Iranzo, G. Moreno, J. Medina, M. Ojeda-Aciego, Efficient thresholded tabulation for fuzzy query answering, Foundations of Reasoning Under Uncertainty, Studies in Fuzziness and Soft Computing, vol. 249, 2010, pp. 125–141.
[36] K. Kersting, L. De Raedt, Bayesian logic programming: theory and tools, in: L. Getoor, B. Taskar (Eds.), An Introduction to Statistical Relational Learning, MIT Press, 2007(Chapter 10).
[37] M.A. Khamsi, D. Misane, Fixed point theorems in logic programming, Ann. Math. Artif. Intell. 21 (1997) 231–243.
[38] M. Kifer, Ai Li, in: On the semantics of rule-based expert systems with uncertainty, in: Proceedings of the International Conference on Database Theory (ICDT-88), Lecture Notes in Computer Science, vol. 326, Springer-Verlag, 1988, pp. 102–117.
[39] M. Kifer, V.S. Subrahmanian, Theory of generalized annotated logic programming and its applications, J. Logic Prog. 12 (1992) 335–367.
[40] S. Krajči, R. Lencses, P. Vojtás, A comparison of fuzzy and annotated logic programming, Fuzzy Sets Syst. 144 (2004) 173–192.
[41] L.V.S. Lakshmanan, F. Sadri, On a theory of probabilistic deductive databases, Theory Pract. Logic Prog. 1 (1) (2001) 5–42.
[42] L.V.S. Lakshmanan, N. Shiri, A parametric approach to deductive databases with uncertainty, IEEE Trans. Knowledge Data Eng. 13 (4) (2001) 554–570.
[43] C. Li, K.C.-C. Chang, I.F. Ilyas, Supporting ad-hoc ranking aggregates, in: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD-06), ACM Press, USA, 2006, pp. 61–72.
[44] C. Li, K.C.-C. Chang, I.F. Ilyas, S. Song, RankSQL: query algebra and optimization for relational top-*k* queries, in: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD-05), ACM Press, New York, NY, USA, 2005, pp. 131–142.
[45] Y. Loyer, U. Straccia, Any-world assumptions in logic programming, Theoret. Comput. Sci. 342 (2–3) (2005) 351–381.
[46] Y. Loyer, U. Straccia, Epistemic foundation of stable model semantics, J. Theory Pract. Logic Prog. 6 (2006) 355–393.

[47] Y. Loyer, U. Straccia, Approximate well-founded semantics, query answering and generalized normal logic programs over lattices, Ann. Math. Artif. Intell. 55 (3–4) (2009) 389–417.
[48] J.J. Lu, J. Calmet, J. Schü, Computing multiple-valued logic programs, Math. Soft Comput. 2 (4) (1997) 129–153.
[49] T. Lukasiewicz, Probabilistic logic programming, in: Proceedings of the 13th European Conference on Artificial Intelligence (ECAI-98), Brighton (England), August 1998, pp. 388–392.
[50] T. Lukasiewicz, Probabilistic logic programming with conditional constraints, ACM Trans. Comput. Logic 2 (3) (2001) 289–339.
[51] T. Lukasiewicz, U. Straccia, Top-*k* retrieval in description logic programs under vagueness for the semantic web, in: Proceedings of the First International Conference on Scalable Uncertainty Management (SUM-07), Lecture Notes in Computer Science, vol. 4772, Springer-Verlag, 2007, pp. 16–30.
[52] T. Lukasiewicz, U. Straccia, Tightly coupled fuzzy description logic programs under the answer set semantics for the semantic web, Int. J. Semant. Web Inf. Syst. 4 (3) (2008) 68–89.
[53] T. Lukasiewicz, U. Straccia, Description logic programs under probabilistic uncertainty and fuzzy vagueness, Int. J. Approxi. Reason. 50 (6) (2009) 837–853.
[54] A. Marian, N. Bruno, L. Gravano, Evaluating top-*k* queries over web-accessible databases, ACM Trans. Database Syst. 29 (2) (2004) 319–362.
[55] C. Mateis, Quantitative disjunctive logic programming: semantics and computation, AI Commun. 13 (2000) 225–248.
[56] J. Medina, M. Ojeda-Aciego, Multi-adjoint logic programming, in: Proceedings of the 10th International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems (IPMU-04), 2004, pp. 823–830.
[57] J. Medina, M. Ojeda-Aciego, P. Vojtás, Multi-adjoint logic programming with continuous semantics, in: Proceedings of the Sixth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-01), Lecture Notes in Artificial Intelligence, vol. 2173, Springer-Verlag, 2001, pp. 351–364.
[58] C. Meghini, F. Sebastiani, U. Straccia, A model of multimedia information retrieval, J. ACM 48 (5) (2001) 909–970.
[59] S. Muggleton, Stochastic logic programs, in: L. De Raedt (Ed.), Proceedings of the Fifth International Workshop on Inductive Logic Programming, Department of Computer Science, Katholieke Universiteit Leuven, 1995, pp. 29.
[60] M. Mukaidono, Foundations of fuzzy logic programming, Advances in Fuzzy Systems—Application and Theory, vol. 1, World Scientific, Singapore, 1996.
[61] R. Ng, V.S. Subrahmanian, Stable model semantics for probabilistic deductive databases, in: Z.W. Ras, M. Zemenkova (Eds.), Proceedings of the Sixth International Symposium on Methodologies for Intelligent Systems (ISMIS-91), Lecture Notes in Artificial Intelligence, vol. 542, Springer-Verlag, 1991, pp. 163–171.
[62] R. Ng, V.S. Subrahmanian, Probabilistic logic programming, Inf. Comput. 101 (2) (1993) 150–201.
[63] P. Nicolas, L. Garcia, I. Stéphan, Possibilistic stable models, in: Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI-05), Morgan Kaufmann Publishers, 2005, pp. 248–253.
[64] H. Nottelmann, U. Straccia, sPLMap: a probabilistic approach to schema matching, in: Proceedings of the 27th European Conference on Information Retrieval Research (ECIR-05), Lecture Notes in Computer Science, vol. 3408, Springer-Verlag, Santiago de Conmpostela, Spain, 2005, pp. 81–95.
[65] A. Ragone, U. Straccia, T. Di Noia, E. Di Sciascio, F.M. Donini, Vague knowledge bases for matchmaking in p2p e-marketplaces, Fourth European Semantic Web Conference (ESWC-07), Lecture Notes in Computer Science, vol. 4519, Springer-Verlag, 2007, pp. 414–428.
[66] A. Ragone, U. Straccia, T. Di Noia, E. Di Sciascio, F.M. Donini, Fuzzy matchmaking in e-marketplaces of peer entities using Datalog, Fuzzy Sets Syst. 160 (2) (2009) 251–268.
[67] C. Ré, N. Dalvi, D. Suciu, Efficient top-*k* query evaluation on probabilistic data, in: Proceedings of the 23st International Conference on Data Engineering (ICDE-07), IEEE Computer Society, 2007.
[68] E.Y. Shapiro, Logic programs with uncertainties: a tool for implementing rule-based systems, in: Proceedings of the Eighth International Joint Conference on Artificial Intelligence (IJCAI-83), 1983, pp. 529–532.
[69] M.A. Soliman, I.F. Ilyas, K.C. Chang, Top-*k* query processing in uncertain databases, in: Proceedings of the 23st International Conference on Data Engineering (ICDE-07), IEEE Computer Society, 2007, pp. 896–905.
[70] U. Straccia, Query answering in normal logic programs under uncertainty, Eighth European Conferences on Symbolic and Quantitative Approaches to Reasoning with Uncertainty (ECSQARU-05), Lecture Notes in Computer Science, vol. 3571, Springer-Verlag, Barcelona, Spain, 2005, pp. 687–700.
[71] U. Straccia, Uncertainty management in logic programming: simple and effective top-down query answering, in: R. Khosla, R.J. Howlett, L.C. Jain (Eds.), Ninth International Conference on Knowledge-Based & Intelligent Information & Engineering Systems (KES-05), Part II, Lecture Notes in Computer Science, vol. 3682, Springer-Verlag, Melbourne, Australia, 2005, pp. 753–760.
[72] U. Straccia, Answering vague queries in fuzzy DL-Lite, in: Proceedings of the 11th International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems (IPMU-06), E.D.K., Paris, 2006, pp. 2238–2245.
[73] U. Straccia, Towards top-*k* query answering in deductive databases, in: Proceedings of the 2006 IEEE International Conference on Systems, Man and Cybernetics (SMC-06), IEEE, 2006, pp. 4873–4879.
[74] U. Straccia, Towards top-*k* query answering in description logics: the case of DL-Lite, in: Proceedings of the 10th European Conference on Logics in Artificial Intelligence (JELIA-06), Lecture Notes in Computer Science, vol. 4160, Springer Verlag, Liverpool, UK, 2006, pp. 439–451.
[75] U. Straccia, A top-down query answering procedure for normal logic programs under the any-world assumption, in: Proceedings of the Ninth European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty (ECSQARU-07), Lecture Notes in Computer Science, vol. 4724, Springer-Verlag, 2007, pp. 115–127.
[76] U. Straccia, Towards vague query answering in logic programming for logic-based information retrieval, World Congress of the International Fuzzy Systems Association (IFSA-07), Lecture Notes in Computer Science, vol. 4529, Springer-Verlag, Cancun, Mexico, 2007, pp. 125–134.

[77] U. Straccia, Fuzzy description logic programs, in: C. Marsala, B. Bouchon-Meunier, R.R. Yager, M. Rifqi (Eds.), Uncertainty and Intelligent Information Systems, World Scientific, 2008, pp. 405–418, (Chapter 29).
[78] U. Straccia, Managing uncertainty and vagueness in description logics, Reasoning Web, Fourth International Summer School, Tutorial Lectures, Lecture Notes in Computer Science, vol. 5224, Springer-Verlag, 2008, pp. 54–103.
[79] U. Straccia, An ontology mediated multimedia information retrieval system, in: Proceedings of the 40th International Symposium on Multiple-Valued Logic (ISMVL-10), IEEE Computer Society, 2010, pp. 319–324.
[80] U. Straccia, SoftFacts: a top-*k* retrieval engine for ontology mediated access to relational databases, in: Proceedings of the 2010 IEEE International Conference on Systems, Man and Cybernetics (SMC-10), IEEE Computer Society, 2010, pp. 4115–4122.
[82] J.D. Ullman, Principles of Database and Knowledge Base Systems, vols. 1 and 2, Computer Science Press, Potomac, MD, 1989.
[83] M.H. van Emden, Quantitative deduction and its fixpoint theory, J. Logic Prog. 4 (1) (1986) 37–53.
[84] P. Vojtás, Fuzzy logic programming, Fuzzy Sets Syst. 124 (2001) 361–370.
[85] P. Vojtás, Fuzzy logic aggregation for semantic web search for the best (top-*k*) answer, in: E. Sanchez (Ed.), Fuzzy Logic and the Semantic Web, Capturing Intelligence, Elsevier, 2006, pp. 341–359, (Chapter 17).
[86] P. Vojtás, M. Vomlelová, Transformation of deductive and inductive tasks between models of logic programming with imperfect information, in: Proceedings of the 10th International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems, (IPMU-04), 2004, pp. 839–846.
[87] G. Wagner, Negation in fuzzy and possibilistic logic programs, in: T. Martin, F. Arcelli (Eds.), Logic programming and Soft Computing, Research Studies Press, 1998.