Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 248 (2009) 115–130
www.elsevier.com/locate/entcs

Thresholded Tabulation in a Fuzzy Logic Setting

Pascual Julián[1]

University of Castilla-La Mancha
Department of Information Technologies and Systems
Pascual.Julian@uclm.es

Jesús Medina[2]

University of Cádiz
Department of Mathematics
Jesus.Medina@uca.es

Ginés Moreno[1]

University of Castilla-La Mancha
Department of Computing Systems
Gines.Moreno@uclm.es

Manuel Ojeda-Aciego[2]

University of Málaga
Department of Applied Mathematics
aciego@ctima.uma.es

Abstract

Fuzzy logic programming represents a flexible and powerful declarative paradigm amalgamating fuzzy logic and logic programming, for which there exists different promising approaches described in the literature. In this paper we propose an improved fuzzy query answering procedure for the so called multi-adjoint logic programming approach, which avoids the re-evaluation of goals and the generation of useless computations thanks to the combined use of tabulation with thresholding techniques. The general idea is that, when trying to perform a computation step by using a given program rule R, we firstly analyze if such step might contribute to reach further significant solutions (non tabulated yet). When it is the case, it is possible to avoid a useless computation step via a rule R by using thresholds and filters based on the truth degree of R, as well as a safe, accurate and dynamic estimation of the maximum truth degree associated to its body.

Keywords: Fuzzy Logic Programming, Tabulation, Thresholding

[1] Partially supported by the EU (FEDER), and the Spanish Science and Education Ministry (MEC) under grants TIN 2004-07943-C04-03 and TIN 2007-65749.

[2] Partially supported by the EU (FEDER), and the Spanish Science and Education Ministry (MEC) under grant TIN 2006-15455-C03-01 and by Junta de Andalucía under grant P06-FQM-02049.

doi:10.1016/j.entcs.2009.07.063

1 Introduction

Fuzzy Logic Programming is an interesting and still growing research area that agglutinates the efforts to introduce fuzzy logic into Logic Programming. During the last decades, several fuzzy logic programming systems have been developed [3, 8,9,15], where the classical inference mechanism of SLD–Resolution is replaced with a fuzzy variant which is able to handle partial truth and to reason with uncertainty.

This is the case of the extremely flexible framework of *Multi-adjoint logic programming* [18–20]. Given a multi-adjoint logic program, queries are evaluated in two separate computational phases. Firstly, an *operational* phase in which *admissible steps* (a generalization of the classical *modus ponens* inference rule) are systematically applied by a backward reasoning procedure, in a similar way to classical resolution steps in pure logic programming; until an expression is obtained in which all atoms have been evaluated. Then, this last expression is interpreted in the underlying lattice during an *interpretive* phase [12], providing the computed answer for the given query.

In [5] a non-deterministic tabulation goal-oriented proof procedure was introduced for residuated (a particular case of multi-adjoint) logic programs over complete lattices. The underlying idea of tabulation is, essentially, that atoms of selected tabled predicates as well as their answers are stored in a table. When an identical atom is recursively called, the selected atom is not resolved against program clauses; instead, all corresponding answers computed so far are looked up in the table and the associated answer substitutions are applied to the atom. The process is repeated for all subsequent computed answer substitutions corresponding to the atom.

In [13] a fuzzy partial evaluation framework was introduced for specializing multi-adjoint logic programs. Moreover, it was pointed out that if the proposed partial evaluation process is combined with thresholding techniques, the following benefits can be obtained:

- The *unfolding tree* (i.e., an incomplete search tree used during the partial evaluation process), consumes less computational resources by efficiently pruning unnecessary branches of the tree and, hence, drastically reducing its size.
- Those derivation sequences performed at execution time, need less computation steps to reach computed answers.

In this paper, we show how the essence of thresholding can be also embedded into a tabulation-based query answering procedure, reinforcing the benefits of both methods in a unified framework. We also provide several kinds of "thresholding filters" which largely help to avoid the generation of redundant and useless computations.

The structure of the paper is as follows. In Section 2 we summarize the main features of multi-adjoint logic programming. Section 3 adapts to the multi-adjoint logic framework the original tabulation procedure for residuated logic programs of [5]. Inspired by [13], the resulting method is refined by using thresholding techniques in Section 4. The benefits of such combination are reinforced in Section 5. Finally,

in Section 6 we draw some conclusions and discuss some lines of future work.

2 Multi-Adjoint Logic Programs

This section is a short summary of the main features of multi-adjoint languages. The reader is referred to [18,20] for a complete formulation.

We will consider a language, $\mathcal{L}$, containing propositional variables, constants, several (arbitrary) connectives to increase language expressiveness. In our fuzzy setting, we use implication connectives $(\leftarrow_1, \leftarrow_2, \ldots, \leftarrow_m)$ together with a number of aggregators, which are only required to be monotonic. They will be used to combine/propagate truth values through the rules. The general definition of aggregation operators subsumes conjunctive operators (denoted by $\&_1, \&_2, \ldots, \&_k$), disjunctive operators $(\vee_1, \vee_2, \ldots, \vee_l)$, and average and hybrid operators (usually denoted by $@_1, @_2, \ldots, @_n$).

Aggregators are useful to describe/specify user preferences: when interpreted as a truth function they may be considered, for instance, as an arithmetic mean or a weighted sum. For example, if an aggregator @ is interpreted as $[\![@]\!](x, y, z) = (3x + 2y + z)/6$, $x, y, z \in [0, 1]$, we are giving the highest preference to the first argument, then to the second, being the third argument the least significant. By definition, the truth function for an n-ary aggregator $[\![@]\!] : L^n \to L$ is required to be monotone and fulfill $[\![@]\!](\top, \ldots, \top) = \top$, $[\![@]\!](\bot, \ldots, \bot) = \bot$.

The language $\mathcal{L}$ will be interpreted on a *multi-adjoint lattice*, $\langle L, \preceq, \leftarrow_1, \&_1, \ldots, \leftarrow_n, \&_n\rangle$, which is a complete lattice equipped with a collection of adjoint pairs $\langle \leftarrow_i, \&_i\rangle$, where each $\&_i$ is a conjunctor[3] intended to provide a *modus ponens*-rule wrt $\leftarrow_i$. In general, the set of truth values L may be the carrier of any complete bounded lattice but, for simplicity, in the examples of this work we shall select L as the set of real numbers in the interval $[0, 1]$.

A *rule* is a formula $A \leftarrow_i \mathcal{B}$, where A is an propositional symbol (usually called the *head*) and $\mathcal{B}$ (which is called the *body*) is a formula built from propositional symbols $B_1, \ldots, B_n$ $(n \geq 0)$, truth values of L and conjunctions, disjunctions and aggregations. Rules with an empty body are called *facts*. A *goal* is a body submitted as a query to the system.

Roughly speaking, a multi-adjoint logic program is a set of pairs $\langle \mathcal{R}; \alpha\rangle$, where $\mathcal{R}$ is a rule and α is a value of L, which might express the confidence which the user of the system has in the truth of the rule $\mathcal{R}$. Note that the truth degrees in a given program are expected to be assigned by an expert. We will often write "$\mathcal{R}$ with α" instead of $\langle \mathcal{R}; \alpha\rangle$.

Procedural Semantics

The procedural semantics of the multi–adjoint logic language $\mathcal{L}$ can be thought as an operational phase followed by an interpretive one [12].

[3] An increasing operator satisfying boundary conditions with the top element.

In the following, $\mathcal{C}[A]$ denotes a formula where A is a sub-expression (usually a propositional symbol) which occurs in the (possibly empty) context $\mathcal{C}[]$, whereas $\mathcal{C}[A/A']$ means the replacement of A by A' in context $\mathcal{C}[]$. In the following definition, we always consider that A is the selected propositional symbol in goal $\mathcal{Q}$.

Definition 2.1 [Admissible Steps] Let $\mathcal{Q}$ be a goal, which is considered as a state, and let $\mathcal{G}$ be the set of goals. Given a program $\mathbb{P}$, an *admissible computation* is formalized as a state transition system, whose transition relation $\rightarrow_{AS} \subseteq (\mathcal{G} \times \mathcal{G})$ is the smallest relation satisfying the following *admissible rules*:

(i) $\mathcal{Q}[A] \rightarrow_{AS} \mathcal{Q}[A/v \&_i \mathcal{B}]$ if there is a rule $\langle A \leftarrow_i \mathcal{B}; v \rangle$ in $\mathbb{P}$ and $\mathcal{B}$ is not empty.

(ii) $\mathcal{Q}[A] \rightarrow_{AS} \mathcal{Q}[A/v])$ if there is a fact $\langle A \leftarrow_i ; v \rangle$ in $\mathbb{P}$.

(iii) $\mathcal{Q}[A] \rightarrow_{AS} \mathcal{Q}[A/\bot]$ if there is no rule in $\mathbb{P}$ whose head is A.

Note that the third case is introduced to cope with (possible) unsuccessful admissible derivations. We shall use the symbols $\rightarrow_{AS1}$, $\rightarrow_{AS2}$ and $\rightarrow_{AS3}$ to distinguish between computation steps performed by applying one of the specific admissible rules. Also, the application of a concrete program rule on a step will be annotated as a superscript of the $\rightarrow_{AS}$ symbol, when it was considered relevant.

Definition 2.2 Let $\mathbb{P}$ be a program and let $\mathcal{Q}$ be a goal. An *admissible derivation* is a sequence $\mathcal{Q} \rightarrow_{AS}^{*} \mathcal{Q}'$. When $\mathcal{Q}'$ is a formula not containing propositional symbols it is called an *admissible computed answer (a.c.a.)* for that derivation.

Example 2.3 Let $\mathbb{P}$ be the following program and let $([0,1], \leq)$ be the lattice where $\leq$ is the usual order on real numbers.

$$\mathcal{R}_1 : p \leftarrow_{\mathtt{P}} q \;\&_{\mathtt{G}}\; r \;\; with \; 0.8$$
$$\mathcal{R}_2 : q \leftarrow_{\mathtt{P}} s \qquad with \; 0.7$$
$$\mathcal{R}_3 : q \leftarrow_{\mathtt{L}} r \qquad with \; 0.8$$
$$\mathcal{R}_4 : r \leftarrow \qquad with \; 0.7$$
$$\mathcal{R}_5 : s \leftarrow \qquad with \; 0.9$$

where the labels P, G and L stand for *Product*, *Gödel* and *Łukasiewicz* connectives.

In the following admissible derivation for the program $\mathbb{P}$ and the goal $p \&_{\mathtt{G}} r$, we underline the selected expression in each admissible step:

$\underline{p} \&_{\mathtt{G}} r \rightarrow_{AS1}{}^{\mathcal{R}_1}$

$(0.8 \&_{\mathtt{P}} (\underline{q} \&_{\mathtt{G}} r)) \&_{\mathtt{G}} r \rightarrow_{AS1}{}^{\mathcal{R}_2}$

$(0.8 \&_{\mathtt{P}} ((0.7 \&_{\mathtt{P}} \underline{s}) \&_{\mathtt{G}} r)) \&_{\mathtt{G}} r \rightarrow_{AS2}{}^{\mathcal{R}_5}$

$(0.8 \&_{\mathtt{P}} ((0.7 \&_{\mathtt{P}} 0.9) \&_{\mathtt{G}} \underline{r})) \&_{\mathtt{G}} r \rightarrow_{AS2}{}^{\mathcal{R}_4}$

$(0.8 \&_{\mathtt{P}} ((0.7 \&_{\mathtt{P}} 0.9) \&_{\mathtt{G}} 0.7)) \&_{\mathtt{G}} \underline{r} \rightarrow_{AS2}{}^{\mathcal{R}_4}$

$(0.8 \&_{\mathtt{P}} ((0.7 \&_{\mathtt{P}} 0.9) \&_{\mathtt{G}} 0.7)) \&_{\mathtt{G}} 0.7$

The a.c.a. for this admissible derivation is: $(0.8\&_{\mathtt{P}}((0.7\&_{\mathtt{P}}0.9)\&_{\mathtt{G}}0.7))\&_{\mathtt{G}}0.7$.

If we exploit all propositional symbols of a goal, by applying admissible steps as much as needed during the operational phase, then it becomes a formula with no propositional symbols which can then be directly interpreted in the multi–adjoint lattice L. We recall from [12] the formalization of this process in terms of the following definition.

Definition 2.4 [Interpretive Step] Let $\mathbb{P}$ be a program and $\mathcal{Q}$ a goal. We formalize the notion of *interpretive computation* as a state transition system, whose transition relation $\rightarrow_{IS} \subseteq (\mathcal{G} \times \mathcal{G})$ is defined as the least one satisfying: $\mathcal{Q}[@(r_1, r_2)] \rightarrow_{IS} \mathcal{Q}[@(r_1,r_2)/[\![@]\!](r_1,r_2)]$, where $[\![@]\!]$ is the truth function of connective @ in the lattice $\langle L, \preceq \rangle$ associated to $\mathbb{P}$.

Definition 2.5 Let $\mathbb{P}$ be a program and $\mathcal{Q}$ an a.c.a., that is, $\mathcal{Q}$ is a goal not containing propositional symbols. An *interpretive derivation* is a sequence $\mathcal{Q} \rightarrow^*_{IS} \mathcal{Q}'$. When $\mathcal{Q}' = r \in L$, being $\langle L, \preceq \rangle$ the lattice associated to $\mathbb{P}$, the value r is called a *fuzzy computed answer (f.c.a.)* for that derivation.

Example 2.6 We complete the previous derivation of Example 2.3 by executing the necessary interpretive steps to obtain the final fuzzy computed answer, 0.504, with respect to lattice $([0, 1], \leq)$.

$(0.8\&_{\mathtt{P}}((0.7\&_{\mathtt{P}}0.9)\&_{\mathtt{G}}0.7))\&_{\mathtt{G}}0.7 \rightarrow_{IS}$

$(0.8\&_{\mathtt{P}}(0.63\&_{\mathtt{G}}0.7))\&_{\mathtt{G}}0.7 \rightarrow_{IS}$

$(0.8\&_{\mathtt{P}}0.63)\&_{\mathtt{G}}0.7 \rightarrow_{IS}$

$0.504\&_{\mathtt{G}}0.7 \rightarrow_{IS}$

0.504

In this section we have just seen a procedural semantics which provides a means to execute multi-adjoint logic programs. However, there exist a more efficient alternative for obtaining fuzzy computed answers for a given query as occurs with the following tabulation-based proof procedure.

3 The Tabulation Proof Procedure

In what follows, we adapt the original tabulation procedure for propositional residuated logic programs described in [5] to the general case of multi-adjoint logic programs [18]. There are two major problems to address: termination and efficiency. On the one hand, the $T_{\mathbb{P}}$ operator is bottom-up but not goal-oriented. Furthermore, the bodies of rules are all recomputed in every step. On the other hand, the usual implementations of Fuzzy Logic Programming languages (e.g. [24]) are goal-oriented, but inherit the problems of non-termination and recomputation of goals. In order to overcome these problems, the tabulation technique has been proposed in the deductive databases and logic programming communities. For instance, in [14]

it is proposed an extension of SLD for implementing generalized annotated logic programs that will be used to implement the here defined tabling procedure. Other implementation techniques have been proposed for dealing with uncertainty in logic programming, for instance translation into Disjunctive Stable Models [17], but rely on the properties of specific truth-value domains.

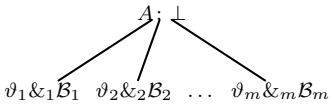
The idea of tabulation (or tabling) is simply to create a table for collecting all the answers to a given goal without repetitions. Every time a goal is invoked it is checked whether there is already a table for that goal. If so, the caller becomes a consumer of the tree, otherwise the construction of a new table is started. All answers produced are kept in the table without repetitions, and are propagated to the pending consumers. The most complete implementation of a full working tabulation system is XSB-Prolog [7] which implements SLG resolution. There is also an extension of SLG for generalized annotated logic programs [14,22] but differs from the system we present here.

In this section we present a general tabulation procedure for propositional multi-adjoint logic programs. The datatype we will use for the description of the method is that of a *forest*, that is, a finite set of trees. Each one of these trees has a root labeled with a propositional symbol together with a truth-value from the underlying lattice (called the *current value* for the *tabulated* symbol); the rest of the nodes of each of these trees are labeled with an "extended" formula in which some of the propositional symbols have been substituted by its corresponding value. For the description of the adaptation of the tabulation procedure to the framework of multi-adjoint logic programming, we will assume a program $\mathbb{P}$ consisting of a finite number of weighted rules having the form $\langle A \leftarrow_i \mathcal{B}; \vartheta \rangle$ together with a query $?A$. The purpose of the computational procedure is to give (if possible) the greatest truth-value for A that can be inferred from the information in the program $\mathbb{P}$.

3.1 Operations for Tabulation

For the sake of clarity in the presentation, we will introduce the following notation: Given a propositional symbol A, we will denote by $\mathbb{P}(A)$ the set of rules in $\mathbb{P}$ which have head A. The tabulation procedure requires four basic operations: Create New Tree, New Subgoal, Value Update, and Answer Return. The first operation creates a tree for the first invocation of a given goal. New Subgoal is applied whenever a propositional variable in the body of a rule is found without a corresponding tree in the forest, and resorts to the previous operation. Value update is used to propagate the truth-values of answers to the root of the corresponding tree. Finally, answer return substitutes a propositional variable by the current truth-value in the corresponding tree. We now describe formally the operations:

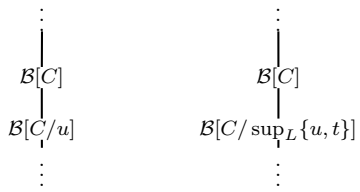
Rule 1: Create New Tree.
Given a propositional symbol A, assume $\mathbb{P}(A) = \{\langle A \leftarrow_j \mathcal{B}_j; \vartheta_j \rangle \mid j = 1, \ldots, m\}$ and construct the tree below, and append it to the current forest. If the forest did not exist, then generate a singleton list with the tree.



Rule 2: New Subgoal.
Select a non-tabulated propositional symbol C occurring in a leaf of some tree (this means that there is no tree in the forest with the root node labeled with C), then create a new tree by directly applying Rule 1, and append it to the forest.

Rule 3: Value Update.
If a tree, rooted at $C\colon r$, has a leaf $\mathcal{B}$ with no propositional symbols, and $\mathcal{B} \rightarrow_{IS}{}^{*} s$, where $s \in L$, then update the current value of the propositional symbol C by the value of $\sup_L\{r, s\}$.

Furthermore, once the tabulated truth-value of the tree rooted by C has been modified, for all the occurrences of C in a non-leaf node $\mathcal{B}[C]$ such as the one in the left of the figure below then, update the whole branch substituting the constant u by $\sup_L\{u, t\}$ (where t is the last tabulated truth-value for C—i.e., $\sup_L\{r, s\}$—) as in the right of the figure.



Rule 4: Answer Return.
Select in any leaf a propositional symbol C which is tabulated, and assume that its current value is r; then add a new successor node as shown below:

Once we have presented the rules to be applied in the tabulation procedure, it is worth to recall some facts:

(i) The only nodes with several immediate successors are root nodes; the successors correspond to the different rules whose head matches the label of the root node.

(ii) The leaf of each branch is a conjunction of the truth value of the rule which determined the branch, with an instantiation of the body of the rule.

(iii) The extension of a tree is done only by Rule 4, which applies only to leaves and extends the branch with one new node.

(iv) The only rule which changes the values of the roots of the trees in the forest is Rule 3 which, moreover, might update the nodes of existing branches.

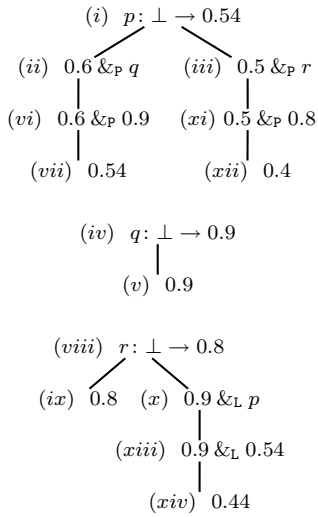


Fig. 1. Example forest for query ?p.

3.2 A non-deterministic procedure for tabulation

Now, we can state the general non-deterministic procedure for calculating the answer to a given query by using a tabulation technique in terms of the previous rules.

Initial step Create the initial forest with the *create new tree* rule, applied to the query.

Next steps Non-deterministically select a propositional symbol and apply one of the rules 2, 3, or 4.

Following the steps in [6] it is not difficult to show both that the order of application of the rules is irrelevant, and that the algorithm terminates under very general hypotheses.

Example 3.1 Consider the following program with mutual recursion and query ?p:

$\mathcal{R}_1 : p \leftarrow_{\mathtt{P}} q$ with 0.6

$\mathcal{R}_2 : p \leftarrow_{\mathtt{P}} r$ with 0.5

$\mathcal{R}_3 : q \leftarrow$ with 0.9

$\mathcal{R}_4 : r \leftarrow$ with 0.8

$\mathcal{R}_5 : r \leftarrow_{\mathtt{L}} p$ with 0.9

Firstly, the initial tree consisting of nodes $(i), (ii), (iii)$ is generated, see Figure 1. Then *New Subgoal* is applied on q, a new tree is generated with nodes (iv) and (v), and its current value is directly updated to 0.9.

By using this value, *Answer Return* extends the initial tree with node (vi). Now *Value Update* generates node (vii) and updates the current value of p to 0.54.

Then, *New Subgoal* is applied on r, and a new tree is generated with nodes $(viii), (ix)$ and (x). *Value Update* increases the current value to 0.8.

By using this value, *Answer Return* extends the initial tree with node (xi). Now *Value Update* generates node (xii). The current value is not updated since its value is greater than the newly computed one.

Finally, *Answer Return* can be applied again on propositional symbol p on node (x), generating node $(xiii)$. A further application of *Value Update* generates node (xiv) and the forest is terminated, as no rule performs any modification.

4 Combining Tabulation with Thresholding

In this section we will focus on the concept of thresholding, initially proposed in [13] for safely pruning branches when generating *unfolding trees*. The original method was firstly introduced inside the core of a fuzzy *partial evaluation* (PE) framework useful not only for specializing fuzzy programs, but also for generating *reductants* [20]. Reductancts were introduced in the context of multi-adjoint logic programming to cope with a problem of incompleteness that arises for non-linear lattices. For instance, given a, b two non-comparable elements in $\langle L, \preceq\rangle$; assume that for a goal A there are only two facts ($\langle A\leftarrow; a\rangle$ and $\langle A\leftarrow; b\rangle$) whose heads are A; both a and b are correct answers and, moreover, by definition of correct answer [20], the supremum $\sup_L\{a, b\}$, is also a correct answer which cannot be computed. The problem above can be solved by extending the original program with a special rule $\langle A\leftarrow \sup_L\{a, b\}; \top\rangle$, the so called *reductant*.

The above discussion shows that a multi-adjoint logic program, interpreted inside a partially ordered lattice, needs to contain all its reductants in order to guarantee the completeness property of a sequence of admissible computations. This obviously increases both the size and execution time of the final *"completed"* program. However, this negative effects can be highly diminished if the proposed reductants have been partially evaluated before being introduced in the target program: the computational effort done (once) at generation time is avoided (many times) at execution time.

Fortunately, if queries are evaluated following the tabulation method proposed before, reductants are not required to be included in a program (which obviously would increase both the size and execution time of the final *completed* program) because their effects are efficiently achieved by the direct use of *Rule 3: Value Update*, as the reader can easily check. Anyway, even when reductants are not mandatory in the tabulation method described in Section 3, it is important to recast some useful ideas introduced in [13], where a refined notion of reductant (called PE-reductant) was given using partial evaluation techniques with thresholding. *Partial evaluation* [1, 10, 16] is an automatic program transformation technique aiming at the optimization of a program with respect to parts of its input: hence, it is also known as *program specialization*. It is expected that the *partially evaluated* (or *residual*) program could be executed more efficiently than the original program. This is because the residual program is able to save some computations, at exe-

cution time, that were done only once at PE time. To fulfill this goal, PE uses symbolic computation as well as some techniques provided by the field of program transformation [2,4,23], specially the so called *unfolding* transformation (essentially, the replacement of a call by its definition body).

Following this path, the idea is to unfold goals, as much as possible, using the notion of unfolding rule developed in [11, 12] for multi-adjoint logic programs, in order to obtain an optimized version of the original program. In [13], the construction of such "unfolding trees" was improved by pruning some useless branches or, more exactly, by avoiding the use (during unfolding) of those program rules whose weights do not surpass a given "threshold" value. For this enhanced definition of unfolding tree we have that:

(i) Nodes contain information about an upper bound of the truth degree associated to their associated goal;

(ii) A set of threshold values is dynamically set to limit the generation of useless nodes.

This last feature provides great chances to reduce the unfolding tree shape, by stopping unfolding of those nodes whose truth degree upper bound component falls down a threshold value α.

4.1 Rules for tabulation with thresholding

In what follows, we will see that the general idea of thresholding can be combined with the tabulation technique shown in the previous section, in order to provide more efficient query answering procedures. Specifically, we will discard the previous descriptions of *Rule 1: Create New Tree* and *Rule 2: New Subgoal*, and instead of them, we propose new definitions:

Rule 1: Root Expansion.
Given a tree with root $A\colon r$ in the forest, and a program rule $\langle A \leftarrow_i \mathcal{B}; \vartheta\rangle$ not consumed before, such that $\vartheta \not\leq r$, append the new child $\vartheta \&_i \mathcal{B}$ to the root of the tree and mark the program rule as consumed.

Rule 2: New Subgoal/Tree.
Select a non-tabulated propositional symbol C occurring in a leaf of some tree (this means that there is no tree in the forest with the root node labeled with C), then create a new tree with a single node, the root $C\colon \bot$, and append it to the forest.

There are several remarks to do regarding the new definitions of Rules 1 and 2. Firstly, notice that the creation of new trees is now performed in Rule 2, instead of Rule 1, which justifies its new name. On the other hand, the new Rule 1, does not create a new tree by expanding (one level) all the possible children of the root. Instead of it, the *Root Expansion* rule has a *lazy* behaviour: each time it is fired, it expands the tree by generating at most one new leaf, if and only if this new leaf might contribute in further steps to reach greater truth degrees than the current

one heading the tree. In this sense, the truth degree attached to the root of the tree, acts as a threshold for deciding which program rules can be used for generating new nodes in the tree. Note also that this threshold is dynamically updated by rule *Value Update*: the more it grows, the less chances for *Root Expansion* to create new children of the root.

The new non-deterministic procedure for tabulation with thresholding is as follows:

Initial step Create an initial tree by using the rule *new subgoal/tree* on the query.

Next steps Non-deterministically select a propositional symbol and apply one of the rules 1, 2, 3, or 4.

In order to show the correctness of the new tabulation procedure, we have just to note that, in the *Root Expansion* rule, when we generate a leaf $\vartheta \&_i \mathcal{B}$ for a root node $A\colon r$, the value generated by the leaf will always be less than ϑ, independently of the truth degree eventually computed for the subgoal $\mathcal{B}$. So, we can safely discard at run-time the use of those program rules (or facts) whose weight ϑ falls down the threshold value r. Otherwise, we would generate useless nodes which never would increase the truth degree of the root.

4.2 A deterministic procedure for tabulation with thresholding

The main goal of thresholding is to reduce the number and size of trees in the forest. This way, although the order of application of the rules is irrelevant because they generate the same solutions, the refinements introduced by thresholding might produce different forests depending on how and when rules are applied. In this section we provide some heuristics in order to minimize as much as possible the complexity of the generated forest.

To begin with, we assume now that the procedure starts with a forest containing a single tree with root $A\colon \bot$, being A the propositional query we plan to answer.

Obviously, the *Root Expansion* rule has a crucial role in this sense: the more lazily it is applied, the less chances it has to generate new nodes. So, we assign it the lowest priority in our deterministic procedure. For a similar reason, it is also important to increase the threshold at the root of a tree as fast as possible. In order to do this, we propose:

(i) Assign maximum priority to *Value Update* and *Answer Return*.

(ii) When program rules are consumed by *Root Expansion* in a top-down way, we assume that facts textually appear before rules with body, and program rules are distributed in a descending ordering w.r.t. their weights, whenever possible.

Notice for instance, the distribution of the rules in Example 3.1, which accomplish with the ordering we have just commented. The proposed strategy applied to the example avoids the construction of a number of nodes, see Figure 2, which evidences the benefits of combining tabulation with thresholding.

The answer to the query example with this optimized procedure is as follows: the initial tree consisting of nodes $(i), (ii)$ is generated. Then *New Subgoal/Tree* is

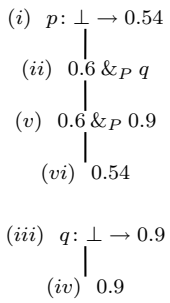


Fig. 2. Example threshold forest for p

applied on q, a new tree is generated with nodes (iii) and (iv), and its current value is directly updated to 0.9.

By using this value, *Answer Return* extends the initial tree with node (v). Now *Value Update* generates node (vi) and updates the current value of p to 0.54.

Now, *Root Expansion* prevents using the rule with body r, since its weight is smaller than the currently computed for p. Hence, the forest is terminated.

5 Reinforcing Thresholding

As we have shown in the previous section, thresholding can be seen as an improvement performed on the core of the basic tabulation proof procedure. The general idea is that all nodes whose value of the body cannot surpass the current value of the root node can be safely removed, or directly, not generated. The thresholding technique described in Section 4 was based on the truth degree of each program rule tried to expand the root of a given tree. However, there is at least two more opportunities for performing thresholding, thus avoiding the unnecessary expansion of trees, as we are going to see in this section.

A sound rule for determining the maximum value of the body of a program rule, might consist in substituting all the propositional variables occurring in it by the top element of the lattice, $\top$. It is easy to see that this second kind of *filter* can reduce the search space if it is appropriately implemented inside the *Root Expansion Rule*. This idea was initially proposed as a further refinement of the original tabulation method for propositional, residuated logic programs of [5]. In the multi-adjoint logic setting, we also find a recent precedent: the same test was used in the PE-based reductant calculus proposed in [13], when collecting leaves of residual unfolding trees. In this paper we are interested in formalizing the same idea inside our *thresholded tabulation* method for multi-adjoint logic programs.

It is easy to see that the previous *pruning rule* can be further enhanced if there is information available about completed tables in the forest, i.e. the truth degrees associated to roots of *completed* or *closed* trees (i.e., which do not admit further updates). Obviously, when the process ends, all trees in the forest are closed. However, the clever point now is how to dynamically guess when a particular tree reach this category in the middle of the process. Fortunately, we have a successful answer to this question, as we are going to explain.

Note that, when the deterministic procedure for tabulation with thresholding we have just seen at the end of the previous section, is not able to fire the root expansion rule (which has the lowest priority in the deterministic strategy explained before) on a concrete tree in the forest, then the maximum truth degree of the propositional symbol rooting such tree has been reached (even when other open trees in the forest might update –increase– the truth degrees of their roots in further steps of the tabulation process). This information collected on the so called *closed trees*, will be crucial for approximating as much as possible the *maximum truth degree of the body* of a program rule, say Up_body, as follows:

- Let $\mathcal{R} = \langle A \leftarrow_i \mathcal{B}; \vartheta \rangle$ be a program rule.
- Let $\mathcal{B}'$ an expression with no atoms, obtained from body $\mathcal{B}$ by replacing each occurrence of a propositional symbol p by $\top$ if there is not a closed tree for p in the forest, or by the corresponding truth degree of p at the root of the closed tree, otherwise.
- Let $v \in L$ be the result of interpreting (by applying the corresponding interpretive steps) $\mathcal{B}'$ under a given lattice, i.e. $\mathcal{B}' \rightarrow^*_{IS} v$.
- Then, $Up\_body(\mathcal{R}) = v$.

Apart from the truth degree ϑ of a program rule $\mathcal{R} = \langle A \leftarrow_i \mathcal{B}; \vartheta \rangle$ and the maximum truth degree of its body $Up\_body(\mathcal{R})$, in the multi-adjoint logic setting, we can consider a third kind of filter for reinforcing thresholding. The idea is to combine the two previous measures by means of the adjoint conjunction $\&_i$ of the implication $\leftarrow_i$ in rule $\mathcal{R}$. Now, we define the *maximum truth degree of a program rule*, symbolized by function Up_rule, as: $Up\_rule(\mathcal{R}) = \vartheta \&_i (Up\_body(\mathcal{R}))$.

Putting all pieces together, we propose the new improved version of the root expansion rule as follows:

Rule 1: Root Expansion.
Given a tree with root A: r in the forest, if there is at least a program rule $\mathcal{R} = \langle A \leftarrow_i \mathcal{B}; \vartheta \rangle$ not consumed before and verifying the three conditions below, append the new child $\vartheta \&_i \mathcal{B}$ to the root of the tree or otherwise, mark the tree as closed.

- Condition 1. $\vartheta \not\leq r$.
- Condition 2. $Up\_body(\mathcal{R}) \not\leq r$.
- Condition 3. $Up\_rule(\mathcal{R}) \not\leq r$.

There are some remarks to do about our definition.

(i) The more *filters* for thresholding we use, the more efficient the method becomes, since the number of nodes in trees can be drastically diminished. Think that by avoiding the generation of a single node, the method implicitly avoids too the generation of all its possible descendants.

(ii) On the other hand, the time required to properly evaluate the filters is largely compensated by the effects explained in the previous item.

(iii) Anyway, in order to perform and efficient evaluation of filters, it must be taken into account that a condition only is checked if none of the previous ones fails. In particular, the unique situation in which the three filters are completely evaluated appears only when the first two ones don't fail.

In order to illustrate the advantages of our improved method, consider that in our running example, we replace the second program rule $\mathcal{R}_2 : p \leftarrow_{\mathtt{P}} r \ \mathit{with}\ 0.5$ by $\mathcal{R}_2' : p \leftarrow_{\mathtt{P}} (r \&_{\mathtt{P}} q) \ \mathit{with}\ 0.55$. It is important to note that with the old version (previous section) of the *Root Expansion Rule*, we could not obtain thresholding benefits, due to the new truth degree 0.55 of $\mathcal{R}_2'$. Note also, that this value verifies the first condition of the new *Root Expansion Rule* when building the forest of Figure 2. So, we proceed by evaluating the second one, which is also satisfied since $Up\_body(\mathcal{R}_2') = 1 * 0.9 = 0.9 \not\leq 0.54$ (observe that q has a closed tree rooted with truth degree 0.9). Fortunately, the third condition fails, since $Up\_rule(\mathcal{R}_2') = 0.55 * 0.9 = 0.495 < 0.54$, which avoids future expansions of the tree (which is then labeled as closed) and in our case, the process finishes generating exactly the same forest of Figure 2.

6 Conclusions and Further Research

In this paper we were concerned with efficient query answering procedures for propositional multi-adjoint logic programs. We have shown that, by using a fuzzy variant of tabulation (specially tailored for the multi-adjoint logic approach) it is possible to avoid the repeated evaluation of redundant goals. Moreover, in the same fuzzy setting, we have also combined tabulation with thresholding, thus safely avoiding other kind of non-redundant, but useless computations.

- Thresholding has been naturally embedded into the core of the tabulation method by simply reformulating in a lazy way the rule which expands the root node of trees.
- By proposing a deterministic strategy which assigns priorities to each "tabulation rule", it is possible to increase the efficiency of the whole method.
- We exploit three kinds of "thresholding filters" for stopping the creation of new tree nodes and maximally reducing the search space.
- Such filters (based on the truth degree of program rules, an upper bound estimation of the truth degrees of their bodies, and a suitable combination of both values), specially the first and third one, have been specially formulated for the multi-adjoint logic approach, and can not be applied to other settings not based in weighted rules (such as pure logic programming, residuated logic programming, etc).

Nowadays, we are working in two practical extensions of our approach:

(i) In order to cover more realistic programs than the ones reported in this paper, we are enriching our technique to cope with the first order case. In this sense, we plan to take advantage from the experience acquired in [6] when lifting to

this more general case the original tabulation proof procedure for propositional residuated logic programs [5].

(ii) Regarding implementation issues, our efforts are devoted to incorporate the proposed technique inside the kernel of the FLOPER environment (see [21] and visit http://www.dsi.uclm.es/investigacion/dect/FLOPERpage.htm). Our tool offers several programming resources regarding the *multi-adjoint logic approach*, including two operational procedures for debugging/tracing and executing goals. The first way is based on a direct translation of fuzzy logic programs into Prolog code in order to safely execute these final programs (via classical SLD-resolution steps) inside any standard Prolog interpreter in a completely transparent way for the final user. The second alternative implements the notion of admissible step seen in Definition 2.1, in order to generate declarative traces based on unfolding trees with any level of depth. We think that the inclusion of a third operational semantics supporting the thresholded tabulation technique studied so far, will give us great opportunities for highlighting the practical benefits of our approach and providing experimental results.

For the future, beyond query answering procedures, we also plan to study the role that tabulation combined with thresholding might play in program transformation techniques such as partial evaluation and fold/unfold, in order to efficiently specialize and optimize multi-adjoint logic programs.

References

[1] M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Specialization of Lazy Functional Logic Programs. In *Proc. of the ACM SIGPLAN Conf. on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'97*, volume 32, 12 of *Sigplan Notices*, pages 151–162, New York, 1997. ACM Press.

[2] M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. Rules + Strategies for Transforming Lazy Functional Logic Programs. *Theoretical Computer Science, Elsevier*, 311(1-3):479–525, Jan. 2004.

[3] J. F. Baldwin, T. P. Martin, and B. W. Pilsworth. *Fril- Fuzzy and Evidential Reasoning in Artificial Intelligence*. John Wiley & Sons, Inc., 1995.

[4] R.M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, 1977.

[5] C.V. Damásio, J. Medina, and M. Ojeda-Aciego. A tabulation proof procedure for residuated logic programming. *In Proc. of the European Conference on Artificial Intelligence, Frontiers in Artificial Intelligence and Applications*, 110:808–812, 2004.

[6] C.V. Damásio, J. Medina, and M. Ojeda-Aciego. Termination of logic programs with imperfect information: applications and query procedure. *Journal of Applied Logic*, 5(3):435–458, 2007.

[7] David S. Warren *et al.* The XSB system version 3.1 volume 1: Programmer's manual. Technical Report Version released on August, 30, Stony Brook University, USA, 2007. (Available from URL: http://xsb.sourceforge.net/).

[8] S. Guadarrama, S. Muñoz, and C. Vaucheret. Fuzzy Prolog: A new approach using soft constraints propagation. *Fuzzy Sets and Systems, Elsevier*, 144(1):127–150, 2004.

[9] M. Ishizuka and N. Kanai. Prolog-ELF Incorporating Fuzzy Logic. In Aravind K. Joshi, editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence (IJCAI'85). Los Angeles, CA, August 1985.*, pages 701–703. Morgan Kaufmann, 1985.

[10] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, NJ, 1993.

[11] P. Julián, G. Moreno, and J. Penabad. On Fuzzy Unfolding. A Multi-adjoint Approach. *Fuzzy Sets and Systems, Elsevier*, 154:16–33, 2005.

[12] P. Julián, G. Moreno, and J. Penabad. Operational/Interpretive Unfolding of Multi-adjoint Logic Programs. *Journal of Universal Computer Science*, 12(11):1679–1699, 2006.

[13] P. Julián, G. Moreno, and J. Penabad. Efficient reductants calculi using partial evaluation techniques with thresholding. *Electronic Notes in Theoretical Computer Science, Elsevier Science*, 188:77–90, 2007.

[14] M. Kifer and V.S. Subrahmanian. Theory of generalized annotated logic programming and its applications. *Journal of Logic Programming*, 12:335–367, 1992.

[15] Deyi Li and Dongbo Liu. *A fuzzy Prolog database system*. John Wiley & Sons, Inc., 1990.

[16] J.W. Lloyd and J.C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11:217–242, 1991.

[17] T. Lukasiewicz. Fixpoint characterizations for many-valued disjunctive logic programs with probabilistic semantics. *Proc of Logic Programming and Non-Monotonic Reasoning, LPNMR'01, Springer-Verlag, Lecture Notes in Artificial Intelligence*, 2173:336–350, 2001.

[18] J. Medina, M. Ojeda-Aciego, and P. Vojtáš. Multi-adjoint logic programming with continuous semantics. *Proc of Logic Programming and Non-Monotonic Reasoning, LPNMR'01, Springer-Verlag, Lecture Notes in Artificial Intelligence*, 2173:351–364, 2001.

[19] J. Medina, M. Ojeda-Aciego, and P. Vojtáš. A procedural semantics for multi-adjoint logic programing. *Progress in Artificial Intelligence, EPIA'01, Springer-Verlag, Lecture Notes in Artificial Intelligence*, 2258(1):290–297, 2001.

[20] J. Medina, M. Ojeda-Aciego, and P. Vojtáš. Similarity-based Unification: a multi-adjoint approach. *Fuzzy Sets and Systems*, 146:43–62, 2004.

[21] P.J. Morcillo and G. Moreno. Programming with Fuzzy Logic Rules by using the FLOPER Tool. *Proc. of 2nd Intl Symposium on Rule Interchange and Applications, RuleML'08, Springer-Verlag, Lecture Notes in Computer Science*, 3521:119–126, 2008.

[22] T. Swift. Tabling for non-monotonic programming. *Annals of Mathematics and Artificial Intelligence*, 25(3–4):201–240, 1999.

[23] H. Tamaki and T. Sato. Unfold/Fold Transformations of Logic Programs. In S. Tärnlund, editor, *Proc. of Second Int'l Conf. on Logic Programming*, pages 127–139, 1984.

[24] P. Vojtáš and L. Paulík. Soundness and completeness of non-classical extended SLD-resolution. In R. Dyckhoff et al, editor, *Proc. ELP'96 Leipzig*, pages 289–301. LNCS 1050, Springer Verlag, 1996.