DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Relatório Técnico

RT-MAC-9413
A System for Reasoning with Fuzzy Predicates

Flávio S. Corrêa da Silva
Daniela V. Carbogim

Junho 94

A System for Reasoning with Fuzzy Predicates

Flávio S. Corrêa da Silva, Daniela V. Carbogim

Instituto de Matemática e Estatística da Universidade de São Paulo - Cid. Universitária "ASO" - PO Box 20570 - 01498-070 São Paulo (SP) - Brazil
email: {fcs, dan}@ime.usp.br

Abstract. Uncertainty is a multifaceted concept, and a system for automated reasoning with multiple representations of uncertainty was proposed in [CdSRI93]. In this work we base on that system and present an efficient language for reasoning with fuzzy predicates.
The language in [CdSRI93] is developed as a PROLOG meta-interpreter. Since the first implementation of this language is not very efficient, we propose two optimization strategies to improve its computational efficiency in time. In order to avoid redundant or unnecessary intermediate computations, we employ two classical optimization techniques, respectively *solution caching* and *α-β pruning*.
Keywords: uncertainty, uncertain reasoning, fuzzy sets, fuzzy predicates, fuzzy reasoning.

1 Introduction

Uncertain Reasoning - the general denomination given to the problems of reasoning *with* and *about* uncertainty - is an interesting and challenging issue for researchers in Artificial Intelligence. One of the multiple forms of uncertainty that can be identified is the result of pervasive *imprecision* in concepts expressed in natural language.

In 1965, Zadeh [Zad65] introduced the theory of *fuzzy sets* as a tool to deal with vaguely defined classes. The key idea behind the formalism of fuzzy sets is the extension of the concept of *characteristic functions* from binary-valued to real-valued functions: the characteristic function μ_C of a "conventional" set C takes elements of a universe U to the values 1 and 0, associated intuitively to elements *belonging* and *not belonging* to C. Hence,

- $\mu_C : U \rightarrow \{0, 1\}$
- $\mu_C(c) = \begin{cases} 0 \Leftrightarrow c \notin C \\ 1 \Leftrightarrow c \in C \end{cases}$

The characteristic function μ_F of a fuzzy set F takes elements of U to values within the real-valued interval $[0, 1]$. The intuitive meaning of $\mu_F(f) = 0$ and $\mu_F(f) = 1$ is the same as for "conventional" sets, and intermediate values represent the "degrees" to which elements of U are members of F:

- $\mu_F : U \rightarrow [0, 1]$

$$- \mu_F(f) = \begin{cases} 0 & \Leftrightarrow f \notin F \\ 1 & \Leftrightarrow f \in F \\ k, 0 < k < 1 & \Leftrightarrow f \in F \\ & \text{"to some extent"} \end{cases}$$

The "fuzzy characteristic function" μ_F is commonly referred to in the literature as the *degree-of-membership* function.

Fuzzy set theory has been used as the basis to represent various forms of uncertainty, e.g. possibilistic degrees of belief [DP87], fuzzy quantifiers [Zad88] and fuzzy predicates [Zad65, CdSRII93]. In this article we concentrate on the latter. The problem in which we are interested can be characterized by the following example: "given that tall people wear large shoes, and given that John is tall, what can be inferred about John's shoe number?"

Many implementations of automated reasoning systems that can provide an answer to this problem have been proposed (see e.g. [Hin86, IK85, Lee72, Orc89, Sha83, vE86]). Building general efficient implementations, however, has proved to be a hard task. In this article we base on the system proposed in [CdSRII93] for automated reasoning with many types of uncertainty, and present a system for reasoning with fuzzy predicates. We propose the utilization of two optimization strategies in the implementation of this system to improve its runtime performance. Both optimizations were implemented and tested, and we present the experimental results we have obtained with them.

The paper is organized as follows: in order to set the notation and to make this work self-contained, in section 2 we review the main concepts of fuzzy set theory and logic programming which are used throughout the work. In section 3 we introduce a logic programming language that works with fuzzy predicates and negation by finite failure. In section 4 we discuss the optimization strategies that we have employed, and explore some implementation issues related to runtime performance of resolution with fuzzy predicates. In section 5 we present our system for reasoning with fuzzy predicates, and the comparative performance results we have obtained with respect to the system introduced in [CdSRII93]. Finally, in section 6 we summarize and conclude this work.

2 Fuzzy Sets and Logic Programming – a Brief Review

In this section we review the concepts of fuzzy measures and logic programming that we use in the rest of the work. First we introduce the concept of *fuzzy sets and relations*, to be used in the interpretation of fuzzy sentences, then we review those concepts of logic programming that we need to define our own language, which include models for interpretation and execution of logic programs.

2.1 Fuzzy Measures

Given a countable universal set U, *fuzzy set theory* was developed to treat vaguely defined subsets by allowing degrees of membership. A fuzzy membership function

measures the degree to which an element belongs to a set or, alternatively, the degree of similarity between the class (set) to which an element belongs and a reference class (universal set). Formally, a fuzzy subset F of a referential set U is defined by an arbitrary mapping $\mu_F : U \rightarrow [0,1]$, in which, for an element $f \in U$, $\mu_F(f) = 1$ corresponds to the intuitive notion that $f \in F$ and $\mu_F(f) = 0$ to the notion that $f \notin F$.

Set-theoretic operations can be extended to fuzzy sets. In [DP89] the requirements for operations on fuzzy sets to be considered extended set operations are presented as follows: let f and g be conventional unary and binary operations on 2^U - the set of subsets of U - and let $\hat{f}$ and $\hat{g}$ be their extensions on the set $2^{\hat{U}}$ of fuzzy subsets of U. The extensions should be such that

1. they are *closed*, i.e. the results of operations on sets are also sets ($F_1, F_2 \in 2^{\hat{U}} \Rightarrow \hat{f}F_1 \in 2^{\hat{U}}, F_1 \hat{g} F_2 \in 2^{\hat{U}}$), and
2. they are *reducible to the conventional operations*, i.e. the results of the extended operations on conventional sets coincide with the ones of conventional operations ($C_1, C_2 \in 2^U \Rightarrow \hat{f}C_1 \equiv fC_1, C_1 \hat{g} C_2 \equiv C_1 g C_2$).

Triangular norms and conorms have been proved to obey these requirements as extensions to the operations of intersection and union, respectively [Kle82]. A *triangular norm* is any function $T : [0,1] \times [0,1] \rightarrow [0,1]$ such that:

- $T(x, 1) = x$ (boundary condition);
- $x_1 \leq x_2, y_1 \leq y_2 \Rightarrow T(x_1, y_1) \leq T(x_2, y_2)$ (monotonicity);
- $T(x, y) = T(y, x)$ (commutativity);
- $T(T(x, y), z) = T(x, T(y, z))$ (associativity).

The *conorm of a triangular norm* is the function $S : [0,1] \times [0,1] \rightarrow [0,1]$ defined by:

- $S(x, y) = 1 - T(1 - x, 1 - y)$.

Furthermore, following [DP89], any function $C : [0,1] \rightarrow [0,1]$ such that $C(\mu_F(d)) = 1 - \mu_F(d)$ obeys the requirements as extension of complementation.

Not all algebraic properties of set operations are shared by general triangular norms and conorms. In fact, as presented in [Kle82], the only norms and conorms that are also *distributive* and *idempotent*[1] are $T = min$ and $S = max$ - proposed in [Zad65] and known as Zadeh's triangular norms and conorms. Henceforth, in order to keep fuzzy set operations as close as possible to conventional set operations, we adopt the following functions as our extended set operations of intersection, union and complementation[2]:

[1] i.e. that obey the following rules:

1. $\left.\begin{array}{l} S(x, T(y, z)) = T(S(x, y), S(x, z)) \\ T(x, S(y, z)) = S(T(x, y), T(x, z)) \end{array}\right\}$ (distributivity);
2. $T(x, x) = x$ and $S(x, x) = x$ (idempotency).

Conventional set union and intersection are both distributive and idempotent

[2] these are the most commonly used definitions of fuzzy set operations

- intersection: $\mu_{A \cap B}(x) = min\{\mu_A(x), \mu_B(x)\}$;
- union: $\mu_{A \cup B}(x) = max\{\mu_A(x), \mu_B(x)\}$;
- complementation: $\mu_{\neg A}(x) = 1 - \mu_A(x)$.

2.2 Logic Programming with Negation

The language presented here is defined after [Kun89]. The class of logic programs supported by this language is that of *function-free, normal, non-cyclical programs* which are *strict with respect to queries* and *allowed* (see definitions below). The symbols of the language are:

- variables $x, y, ...$;
- constants $a, b, ...$;
- n-ary predicates $p, q, ...$;
- the connectives '$\leftarrow$' ("if"), '$\neg$' ("not"), ',' ("and").

A *term* is a variable or a constant, an *atom* is a predicate application on terms, and a *literal* is an atom (positive literal) or the negation of an atom (negative literal). A *normal clause* is an expression $p \leftarrow q_1, ..., q_n$ where p is an atom and $q_1, ..., q_n$ are literals, $n \geq 0$. p is called the *head* of the clause and $q_1, ..., q_n$ is called the *body* of the clause. When $n = 0$ the clause is called a *unit clause*. A *query clause* is an expression $q_1, ..., q_n$ where $n \geq 0$. A *normal program* is a finite non-empty set of normal clauses.

Let Pr_P be the set of predicates in the program P. The *immediate dependency relation* $\sqsupseteq$ is defined as follows:

- given $p, q \in Pr_P, p \sqsupseteq q$ iff there is a clause in P in which p occurs in the head and q occurs in the body.

The *dependency relation* $\geq$ is defined as the least transitive reflexive relation on Pr_P extending $\sqsupseteq$: $p \geq q$ means that p depends hereditarily on q.

Signed dependencies are defined as follows:

- $p \sqsupseteq_{+1} q$ iff there is a clause in P in which p occurs in the head and q occurs in a *positive* literal in the body;
- $p \sqsupseteq_{-1} q$ iff there is a clause in P in which p occurs in the head and q occurs in a *negative* literal in the body;
- $\geq_{+1}$ and $\geq_{-1}$ are the least pair of relations satisfying:
  - $p \geq_{+1} p$,
  - $p \sqsupseteq_i q \wedge q \geq_j r \Rightarrow p \geq_{i \times j} r$.

A program P is called *call-consistent* iff it does not have any p such that $p \geq_{-1} p$. If P also does not have any p such that $p \geq_{+1} p$ then it is called *non-cyclical*. P is called *strict with respect to a query* φ iff there are no $p \in P, q \in \varphi$ such that $q \geq_{-1} p$ and $q \geq_{+1} p$. P is called *allowed* iff every variable occurring in each clause of P occurs in at least one positive literal in the body of the clause.

An *instance* of an expression[3] ξ is the expression ξ' obtained by replacing all occurrences of a variable x in ξ by a term different from x. The operation that generates instances is called *substitution*. Essentially, a substitution is a mapping from variables to terms. A *ground instance* of an expression ξ is any variable-free instance of ξ. Given a program P, *ground*(P) stands for the set of all ground instances of the clauses in P.

A substitution σ of two expressions ξ_1 and ξ_2 is a *unifier* iff $\xi_1\sigma = \xi_2\sigma$. It is a *most general unifier* (mgu) iff for any other unifier π of ξ_1 and ξ_2, $\xi_i\pi$ are instances of $\xi_i\sigma (i = 1, 2)$.

Assuming first-order logic with equality as the underlying language, the *completion* of a program P (*Comp*(P)) is defined by the rules and axioms presented in figure 1 [Tur89].

- Rules:

  Denoting by Def_p the definition of the predicate p in the program:

  - Def_p = the set of clauses in P with p in the head.

  1. $\dfrac{Def_p = \{\}}{\forall x[\neg p(x)]}$

  2. $\dfrac{Def_p = \{p(t_i) \leftarrow \psi_i : i = 1, \ldots, k\} \neq \{\}}{\forall x[p(x) \leftarrow \bigvee_{i=1}^{k} \exists((x = t_i) \wedge \psi_i)]}$

     where

     (a) x, t_i are tuples, with the proper arity, of variables ($[x_1, \ldots, x_m]$) and terms ($[t_{i1}, \ldots, t_{mi}]$), respectively;

     (b) $x = t_i$ stands for $x_1 = t_{i1} \wedge \ldots \wedge x_m = t_{mi}$;

     (c) the scope of the existential quantifier is the variables occurring in the bodies of the clauses in Def_p;

     (d) ψ_i are (possibly empty) conjunctions of literals; and

     (e) the connective $\leftarrow$ stands for equivalence.

- Axioms:

  1. equality axioms [Men87]:

     (a) $\forall x(x = x)$ (reflexivity).

     (b) $x_1 = x_2 \leftarrow (C(x_1, x_1) \leftarrow C(x_1, x_2))$ (substitutivity)
     where x, x_1, x_2 are variables, $C(x_1, x_1)$ is a clause and $C(x_1, x_2)$ is the same clause with some (but not necessarily all) occurrences of x_1 replaced by x_2;

  2. $t(x) \neq x$ for each term in which x occurs.

Fig. 1. Completion *Comp*(P) of program P

The *semantic model* of a program P is defined in terms of its completion *Comp*(P). The *domain* of *Comp*(P) consists of the non-empty set U of constants occurring in P. The interpretation of a predicate $p \in Pr_P$ is a function $I(p)$:

[3] an expression is a term, a literal or a clause

$U^n \rightarrow \{\mathsf{T}, \bot\}$ where n is the arity of p, T stands for "true" and $\bot$ stands for "false". The interpretation of the equality and the truth tables of the connectives occurring in $Comp(P)$ are defined as usual (see, for example, [Men87]). Any interpretation that takes every expression occurring in $Comp(P)$ to the value T is a *model of* P.

Now it is possible to introduce an inference procedure for this language. The procedure is *SLDNF*[4]. First we must introduce some notation. In what follows:

- φ_i are literals;
- p_i, q_i are positive literals;
- g_i are positive ground literals;
- δ_i, ψ_i are (possibly empty) conjunctions of literals;
- σ, π are substitutions;
- R stands for "returns": $\psi R \sigma$ holds iff *SLDNF* succeeds on ψ with the substitution σ as an answer, in which case we say that ψ belongs to the *success set* R of the program;
- F stands for "fails": ψF holds iff *SLDNF* fails, in which case we say that ψ belongs to the *finite failure set* F of the program;
- *true* stands for the empty query clause;
- *yes* stands for the identity substitution.

The procedure is defined by the inductive rules presented in figure 2.

---

1. $true\ R\ yes.$
2. $\dfrac{(q,\delta), \exists[p \leftarrow \psi] : \sigma = mgu(q,p), (\psi,\delta)\sigma R \pi}{(q,\delta)\ R\ (\sigma\pi)}$
3. $\dfrac{(\neg g, \delta), gF, \delta R \sigma}{(\neg g, \delta)\ R\ \sigma}$
4. (a) $\dfrac{(q,\delta), \neg\exists[p \leftarrow \psi] : \exists mgu(q,p)}{(q,\delta)\ F}$

   (b) $\dfrac{(q,\delta), \forall[p_i \leftarrow \psi_i] : \exists\sigma = mgu(q,p_i) \Rightarrow (\psi_i,\delta)\sigma\ F}{(q,\delta)\ F}$
5. $\dfrac{(\neg g, \delta), g\ R\ yes}{(\neg g, \delta)\ F}$

Fig. 2. *SLDNF*

[4] *SLDNF* stands for *L*inear Resolution with a *S*election Rule for *D*efinite Clauses, extended with *N*egation by Finite *F*ailure. "Linear" indicates that each inference step uses the most-recently resolved clause as an input, "selection rule" indicates the use of some fixed rule to select the other inputs of each inference, "definite clauses" defines the class of clauses initially tractable by the procedure (a definite clause is a normal clause in which all literals are positive), and "negation by finite failure" indicates that these clauses are extended to accommodate negation - resulting in what we are calling normal clauses - and that negation is interpreted in the specific way presented in the following paragraphs

In [Apt87] we have that *SLDNF* is *sound*, i.e. that given a query ψ and a program P, if (using our notation) $\psi R\sigma$ then $Comp(P) \models \psi\sigma$, and if ψF then $Comp(P) \models \neg\psi$, where $Comp(P) \models \Phi$ means that Φ is a semantic consequence of $Comp(P)$. A *completeness* result can be found in [Kun89]: for the classes of programs and queries considered in our work (actually, [Kun89] treats more general classes of programs and queries, allowing e.g. cycles and functions), if $Comp(P) \models \psi\sigma$ then $\psi R\sigma$, and if $Comp(P) \models \neg\psi$ then ψF. This defines a rich subset of first-order logic with a computationally efficient inference procedure and a formally specified declarative semantics.

3 A Language to Reason with Fuzzy Predicates

The relationship between fuzzy logics and the resolution principle is well established. Since [Lee72], one of the pioneering works in the area, several proposals have been made, aiming at richer languages in respect of both the logical and the fuzzy relations supported.

In [Lee72] the language is limited to *definite clauses* [Apt87, Llog90] *allowing fuzzy predicates with truth-values always greater than 0.5*[5]. The semantics of the relevant connectives is defined according to Zadeh's triangular norms and conorms and resolution is extended to propagate truth-values in a way that is sound and complete with respect to the Herbrand interpretation of sets of clauses. Several implementations based on [Lee72] have been proposed, e.g. the ones described in [Hin86, IK85, Orc89].

More recent developments [Fit88, Fit90, KS91, Sha83, vE86] have focused on fixpoint semantics, either working with definite programs or approaching the definition of negation by means other than finite failure. We adopt negation by finite failure in this work, in order to have the more conventional languages which are based on this principle (e.g. pure PROLOG) as proper subsets of our language. This choice is corroborated by the results found in [Tur89, CL89, Fit85, Kun87, Kun89, Kun90], which determine large classes of *normal programs* with a well-defined declarative semantics.

In what follows we introduce a language to deal with fuzzy predicates. First we present the language, then its model theory and inference procedure.

Fuzzy predicates can be defined by analogy with the concept of fuzzy sets. The interpretation of predicates can be generalized to a function $I(p) : U^n \rightarrow [0, 1]$, with the extreme values corresponding to the previous values T and $\bot$ (namely, $\mathsf{T} \equiv 1$ and $\bot \equiv 0$). This function can be construed as a fuzzy membership function and the logical connectives can be interpreted as fuzzy set operators - '$\neg$' corresponding to complementation, '$\vee$' corresponding to union, '$\wedge$' corresponding to intersection, and '$\leftrightarrow$' corresponding to set-equivalence. Intuitively, the semantics of a closed formula becomes a "degree of truth", rather than simply one value out of $\{\mathsf{T}, \bot\}$. Let τ denote this value and $\mathcal{T}(\psi, \tau)$ state that "the

[5] the limitations on the types of clauses and range of truth-values are conditions imposed to obtain soundness for the specific resolution procedure employed in [Lee72]

truth-degree of ψ is τ". This evaluation can be made operational using an *extended SLDNF (e-SLDNF)* procedure, to be related to the model of an *extended completion* of a program P (*e-Comp*(P)). We assume that the *unit clauses* (and only them) in the program express truth-degrees, that is, unit clauses are of the form $\mathcal{T}(p, \tau)$, where $\tau > 0$.[6]

The *extended completion* of a program P (*e-Comp* (P)) is defined as presented in figure 3.

Two classes of formulae can be identified in *e-Comp*(P):

- *unit formulae*, generated by rule 1 or from the unit clauses occurring in P; and
- *equivalence formulae*, i.e. the remaining ones, all of them containing the connective $\leftrightarrow$.

The connectives occurring in *e-Comp*(P) are interpreted according to the truth-functions defined below:

Assuming that:

- $\mathcal{T}(\delta, \tau_\delta)$, and
- $\mathcal{T}(\psi, \tau_\psi)$

We have that:

- $\mathcal{T}((\delta \wedge \psi), \tau) \Rightarrow \tau = min\{\tau_\delta, \tau_\psi\}$
- $\mathcal{T}((\delta \vee \psi), \tau) \Rightarrow \tau = max\{\tau_\delta, \tau_\psi\}$
- $\mathcal{T}((\neg\delta), \tau) \Rightarrow \tau = 1 - \tau_\delta$
- $\begin{cases} \mathcal{T}((\delta \leftrightarrow \psi), 1) \Rightarrow \tau_\delta = \tau_\psi \\ \mathcal{T}((\delta \leftrightarrow \psi), 0) \Rightarrow \tau_\delta \neq \tau_\psi \end{cases}$

The completion of a conventional program defines a unique model for the program. For the extended completion to do the same, a necessary condition is to fix the truth-values for the unit clauses occurring in P as values greater than 0. This condition is also sufficient, as all the other formulae in *e-Comp*(P) - i.e. the equivalence formulae and the unit formulae generated by rule 1 - must have truth-values equal to 1 in the model of the program.

A model for a program containing fuzzy predicates is any interpretation for which every expression φ occurring in *e-Comp*(P) has a truth-value $\tau > 0$.

Our notation for logic programs and the *e-SLDNF* procedure is basically the notation used in figure 2, with the following alterations:

- R^e stands for "returns with a truth-value greater than 0": $\psi\ R^e\ (\sigma, \tau)$ holds iff *e-SLDNF* succeeds, assigning a truth-value τ to ψ, with the substitution σ as an answer;

[6] the restrictions on how to declare truth-degrees are imposed to avoid ambiguity, redundancy and conflicting declarations. The language presented here is monotonic and does not contain mechanisms to resolve truth-degrees if they are declared for unit clauses as well as larger constructs (e.g. general normal clauses)

- Rules:
  1. $\dfrac{Def_p = \{\}}{\forall \mathbf{x}[\mathcal{T}(\neg p(\mathbf{x}), 1)]}$
  2. $\dfrac{Def_p = \{p(\mathbf{t}_i) \leftarrow \psi_i : i = 1, \ldots, k\} \neq \{\}}{\forall \mathbf{x}[\mathcal{T}(p(\mathbf{x}), \tau) \leftrightarrow max\{\tau_i : (\mathbf{x} = \mathbf{t}_i) \wedge [(\psi_i \neq \{\} \wedge \mathcal{T}(\psi_i, \tau_i)) \vee (\psi_i = \{\} \wedge \mathcal{T}(p(\mathbf{t}_i), \tau_i))]\} = \tau]}$

     where
     (a) Def_p is the set of clauses in P with p in the head;
     (b) $\mathbf{x}, \mathbf{t}_i$ are tuples of variables ($[x_1, \ldots, x_m]$) and terms ($[t_{i1}, \ldots, t_{mi}]$), respectively;
     (c) $\mathbf{x} = \mathbf{t}_i$ stands for $x_1 = t_{i1} \wedge \ldots \wedge x_m = t_{mi}$;
     (d) ψ_i are (possibly empty) conjunctions of literals;
     (e) the connective $\leftrightarrow$ stands for equivalence.
- Axioms:
  - same as in figure 1.

Fig. 3. Extended completion of P

- F^e stands for "fails": ψF^e holds iff *e-SLDNF* fails, implying the assignment of a truth-value $\tau = 0$ to ψ.

e-SLDNF is defined inductively as presented in figure 4.

The intuition underlying the definitions of *e-Comp*(P) and *e-SLDNF* is that we need the truth-degrees declared in the unit clauses in a program P "transferred" to the heads of the clauses in P in a consistent way. Central to these definitions is the notion of completed database, which makes it possible to define the truth-degrees to be "transferred" as unique.

One feature of the way the language is formalized is that it can be programmed

as a PROLOG meta-interpreter almost literally, i.e. we can identify a term, predicate and clause in the PROLOG implementation with each term, predicate and clause in the language in an almost immediate way. Unfortunately, this implementation is not very efficient, as it requires the exhaustive search of all possible ways of achieving all solutions for any query to select the appropriate truth-value.

The implementation can be presented as follows (adapted from [CdS92]):
Given a program P and a query $\mathcal{T}(\psi, \tau)$:

1. assume that $\psi = (\varphi, \delta)$, where φ is a literal and δ is a query clause (notice that $\varphi = (\varphi, true)$).
2. solve $\mathcal{T}(\varphi, \tau_\varphi)$, resulting either $\varphi\ R^e\ (\sigma_\varphi, \tau_\varphi)$ or $\varphi\ F^e$.
   if $\varphi\ F^e$ then return $\psi\ F^e$.
3. if $\varphi\ R^e\ (\sigma_\varphi, \tau_\varphi)$ then solve $\mathcal{T}(\delta\sigma_\varphi, \tau_\delta)$, resulting either $\delta\ R^e\ (\sigma_\varphi\sigma_\delta, \tau_\delta)$ or $\delta\sigma_\varphi\ F^e$.
   if $\delta\sigma_\varphi\ F^e$ then return $\psi\ F^e$.
4. if $\delta\ R^e\ (\sigma_\varphi\sigma_\delta, \tau_\delta)$ then return $\psi\ R^e\ (\sigma_\varphi\sigma_\delta, min\{\tau_\varphi, \tau_\delta\})$.

1. $true\ R^e\ (yes, 1)$

2. (a) $$\frac{(q,\delta), max\left\{\begin{array}{l}\tau_i : [p_i \leftarrow \psi_i], \sigma_i = mgu(q, p_i),\\ (\psi_i, \delta)\sigma_i\ R^e\ (\pi_i, \tau_i) \vee \sigma_i = mgu(q, p_i),\\ (\delta)\sigma_i\ R^e\ (\pi_i, \tau_i'), \mathcal{T}((p_i)\sigma_i, \tau_i''),\\ min\{\tau_i', \tau_i''\} = \tau_i\end{array}\right\} = \tau}{(q,\delta)\ R^e\ (\sigma\pi, \tau)}$$

where $\sigma\pi$ is the substitution that generates τ and the ψ_i are non-empty conjunctions.

(b) $$\frac{(\neg g, \delta), g\ R^e\ (yes, \tau'), \tau' < 1, \delta\ R^e\ (\sigma, \tau''), min\{(1 - \tau'), \tau''\} = \tau}{(\neg g, \delta)\ R^e\ (\sigma, \tau)}$$

3. $$\frac{(\neg g, \delta), g F^e, \delta\ R^e\ (\sigma, \tau)}{(\neg g, \delta)\ R^e\ (\sigma, \tau)}$$

4. (a) $$\frac{(q, \delta), \neg\exists[p \leftarrow \psi] : \exists mgu(q, p)}{(q, \delta)\ F^e}$$

(b) $$\frac{(q, \delta), \forall[p_i \leftarrow \psi_i] : \exists\sigma = mgu(q, p_i) \Rightarrow (\psi_i, \delta)\sigma\ F^e}{(q, \delta)\ F^e}$$

5. $$\frac{(\neg g, \delta), g\ R^e\ (yes, 1)}{(\neg g, \delta)\ F^e}$$

Fig. 4. *e-SLDNF*

To solve $\mathcal{T}(\varphi, \tau_\varphi)$:

1. if $\varphi = true$ then return $true\ R^e\ (yes, 1)$.
2. if $\varphi =$ positive literal q then find the collection C of all expressions E in P such that

   (a) $E = p_i \leftarrow \psi_i$ and $\exists\sigma_i = mgu(q, p_i)$ and $\mathcal{T}(\psi_i\sigma_i, \tau_i)$, or
   (b) $E = \mathcal{T}(p_i, \tau_i)$ and $\exists\sigma_i = mgu(q, p_i)$.

   if $C = \{\}$ then return $q\ F^e$. Otherwise
   select from C the index j such that $\tau_j = max_C\{\tau_i\}$ and return $q\ R^e\ (\sigma_j, \tau_j)$.
3. if $\varphi =$ negative ground literal $\neg g$ then solve $\mathcal{T}(g, \tau_g)$.
   if $g\ F^e$ then return $\neg g\ R^e\ (yes, 1)$.
   if $g\ R^e\ (\sigma_g, 1)$ then return $\neg g F^e$.
   if $g\ R^e\ (\sigma_g, \tau_g), \tau_g < 1$ then return $\neg g\ R^e\ (\sigma_g, 1 - \tau_g)$.

One possible way to improve the runtime performance of this system is by avoiding the exhaustive search of all paths to all solutions for a query. To do this, the system must ignore intermediate computations that no longer will affect the selection of the appropriate truth-value.

In order to avoid these redundant or unnecessary calculations, we apply two classical program optimization techniques, respectively *solution caching* and α-β *pruning*.

In the following section, we explore the applicability of these two techniques to the meta-interpreter in order to improve its efficiency in time.

4 Optimization Strategies for Fuzzy Reasoning

As mentioned above, the two optimization techniques we explore are *solution caching* and α-β *pruning*.

4.1 Solution Caching

The *solution caching* strategy can be described as "storing conclusions found during an evaluation process in a way that allows the system to reuse these conclusions to answer subsequent queries".

When the evaluation of a clause is required in different points of a program, a subtree is generated repeatedly to resolve a query. If this generation is computationally costly, it can be worthwhile to store the result of its evaluation the first time it is generated in order to avoid its recalculation.

Unfortunately, storing and recalling values also imply in computational costs. The trade-off is advantageous only when these costs are smaller than the costs of generating repeatedly the subtrees which evaluation is being stored.

So, the improvements on efficiency by applying *solution caching* depend on the characteristics of specific programs. It may not be useful also when queries are not repeated a considerably large number of times during resolution.

4.2 α-β Pruning

The α-β *pruning* strategy was originally proposed to increase efficiency of search processes in game trees without loss of information [Pea84]. Here, it is employed to guide query evaluation.

The key idea is that some operations can be simplified in the evaluation procedure whenever a value is selected from a partially ordered set of alternatives and the choice is made based on this partial order.

When computing a collection of expressions, some values are generated and immediately discarded, since only the maximum value in this collection is used for further calculations. Similarly, when computing a conjuctive query, only the minimum value obtained is used.

The idea is to avoid unnecessary computations. We do not have to generate the truth-values for all elements of a collection of expressions if we can generate one value and verify that no other value can be greater than it. Analogously, we do not have to generate the truth-values for each conjunct in a query if we can generate one value and verify that no other conjunct can produce a smaller one. The α-β *pruning* technique tries to verify these conditions.

However, this strategy can be totally ineffective depending on the order in which clauses are selected for unification. So, the improvements on efficiency obtained by applying it also depend on the characteristics of specific programs.

Generally speaking, we can expect an improvement on the average runtime behavior of the system after we employ these optimization strategies, although not much can be guaranteed about their effect on the behavior of the system for specific programs.

In the following section, we detail some experimental results we have obtained from using *solution caching* and α-β *pruning* to evaluate fuzzy query clauses.

5 An Efficient Language to Reason with Fuzzy Predicates

In order to obtain an efficent language to reason with fuzzy predicates, we base on the PROLOG meta-interpreter described in section 3 and apply the explored optimization strategies to this implementation.

We have built three PROLOG programs, implementing:

- the"non-optimized" meta-interpreter;
- the interpreter employing the *solution caching* strategy;
- the interpreter employing both α-β *pruning* and *solution caching* strategies.

In what follows we present an efficient language to deal with fuzzy predicates including both optimization strategies. We also present some experimental results based on representative examples of its use.

5.1 The Optimized System

Here we present the language obtained by employing both optimization strategies to the meta-interpreter. Its implementation can be described as follows (adapted from [CdS92]):

Given a program P and a query $\mathcal{T}(\psi, r)$:

1. assume that $\psi = (\varphi, \delta)$, where φ is a literal and δ is a query clause.
2. solve $\mathcal{T}(\varphi, r_\varphi)$, resulting either $\varphi\ R^e\ (\sigma_\varphi, r_\varphi)$ or $\varphi\ F^e$.
   if $\varphi\ F^e$ then return $\psi\ F^e$.
3. if $\varphi\ R^e\ (\sigma_\varphi, r_\varphi)$ then solve $\alpha\mathcal{T}(\delta\sigma_\varphi, \dot{r}_\delta)$.
   if $\dot{r}_\delta \geq r_\varphi$ then return $\psi\ R^e\ (\sigma_\varphi, r_\varphi)$. Otherwise
4. solve $\mathcal{T}(\delta\sigma_\varphi, r_\delta)$, resulting either $\delta\ R^e\ (\sigma_\varphi\sigma_\delta, r_\delta)$ or $\delta\sigma_\varphi\ F^e$.
   if $\delta\sigma_\varphi\ F^e$ then return $\psi\ F^e$.
5. if $\delta\ R^e\ (\sigma_\varphi\sigma_\delta, r_\delta)$ then return $\psi\ R^e\ (\sigma_\varphi\sigma_\delta, min\{r_\varphi, r_\delta\})$.

To solve $\mathcal{T}(\varphi, r_\varphi)$:

1. if $\varphi = true$ then return $true\ R^e\ (yes, 1)$.
2. if $\varphi =$ positive literal q then
   (a) if $\varphi =$ ground literal and there is one previously stored expression E in P of the form $\mathcal{T}m(q, r_q)$ then return $\varphi\ R^e\ (yes, r_q)$. Otherwise
   (b) find one expression E in P such that
      i. $E = p_i \leftarrow \psi_i$ and $\exists\sigma_i = mgu(q, p_i)$ and $\mathcal{T}(\psi_i\sigma_i, r_i)$, or
      ii. $E = \mathcal{T}(p_i, r_i)$ and $\exists\sigma_i = mgu(q, p_i)$.

   if there is no such E then return $q\ F^e$. Otherwise solve $\beta\mathcal{T}(\varphi, \dot{r}_\varphi)$.
   if $\dot{r}_\varphi \leq r_i$ then store in P the expression $\mathcal{T}m(\varphi\sigma_i, r_i)$ and return $\varphi\ R^e\ (\sigma_i, r_i)$. Otherwise find another expression E with the same conditions and iterate the process.

3. if φ = negative ground literal $\neg g$ then solve $\mathcal{T}(g, \tau_g)$.
   if $g\ F^e$ then return $\neg g\ R^e\ (yes, 1)$.
   if $g\ R^e\ (\sigma_g, 1)$ then return $\neg g F^e$.
   if $g\ R^e\ (\sigma_g, \tau_g), \tau_g < 1$ then return $\neg g\ R^e\ (\sigma_g, 1 - \tau_g)$.

To solve $\alpha\mathcal{T}(\psi, \tau)$: similar to $\mathcal{T}(\psi, \tau)$, replacing $\mathcal{T}(\varphi, \tau_\varphi)$ by $\alpha\mathcal{T}(\varphi, \tau_\varphi)$.

To solve $\alpha\mathcal{T}(\varphi, \tau_\varphi)$:

1. if φ = *true* then return *true* R^e $(yes, 1)$.
2. if φ = positive literal q then find one expression E in P such that
   (a) $E = p_i \leftarrow \psi_i$ and $\exists\sigma_i = mgu(q, p_i)$ and $\mathcal{T}(\psi_i\sigma_i, \tau_i)$, or
   (b) $E = \mathcal{T}(p_i, \tau_i)$ and $\exists\sigma_i = mgu(q, p_i)$.
   if there is no such E then return $q\ F^e$. Otherwise return τ_i.
3. if φ = negative ground literal $\neg g$ then solve $\mathcal{T}(g, \tau_g)$.
   if $g\ F^e$ then return $\tau_\varphi = 1$.
   if $g\ R^e\ (\sigma_g, 1)$ then return $\neg g F^e$.
   if $g\ R^e\ (\sigma_g, \tau_g), \tau_g < 1$ then return $\tau_\varphi = 1 - \tau_g$.

To solve $\beta\mathcal{T}(\psi, \tau)$:

1. assume that $\psi = (\varphi, \delta)$, where φ is a literal and δ is a query clause.
2. solve $\mathcal{T}(\varphi, \tau_\varphi)$, resulting either $\varphi\ R^e\ (\sigma_\varphi, \tau_\varphi)$ or $\varphi\ F^e$.
   if $\varphi\ F^e$ then return $\psi\ F^e$. Otherwise return τ_φ.

To solve $\beta\mathcal{T}(\varphi, \tau_\varphi)$: similar to $\mathcal{T}(\varphi, \tau_\varphi)$, replacing $\mathcal{T}(...)$ by $\beta\mathcal{T}(...)$

5.2 Experimental Results

In order to verify the effectiveness of the optimization strategies applied to the meta-interpreter, we develop an empirical comparative analysis of the system runtime efficiency.

For this analysis we consider the available PROLOG programs. $P1$ corresponds to the "non-optimized" language, $P2$ corresponds to the version including *solution caching*, and $P3$ corresponds to the version combining both *solution caching* and *α-β pruning*.

To compare the efficiency in time of the above implementations, we use representative examples of two types, henceforth called Ak and Bk, $k = 1, ..., 10$, in which respectively *$\alpha - \beta$ pruning is* and *is not* effective.

We assume that the and/or resolution trees are generated depth-first and left-to-right.

The Ak/Bk-examples are presented as follows:

Examples Ak:

$A1$:

$a_0(X) : -a_1(X), a_2(X), a_3(X).$ $\mathcal{T}(a_1(a), 0.1).$
$a_3(X) : -a_2(X).$ $\mathcal{T}(a_4(a), 0.9).$
$a_2(X) : -a_4(X).$ $\mathcal{T}(a_5(a), 0.2).$
$a_2(X) : -a_5(X).$
$a_2(X) : -a_6(X).$
$a_6(X) : -a_5(X).$

(...)

$A10$:

$a_0(X) : -a_1(X), a_2(X), a_3(X).$ $\mathcal{T}(a_1(a), 0.1).$
$a_3(X) : -a_2(X).$ $\mathcal{T}(a_4(a), 0.9).$
$a_2(X) : -a_4(X).$ $\mathcal{T}(a_7(a), 0.1).$

$a_2(X) : -a_5(X).$ $\vdots$
$a_2(X) : -a_6(X).$ $\mathcal{T}(a_{55}(a), 0.1).$
$a_6(X) : -a_8(X).$ $\mathcal{T}(a_{58}(a), 0.9).$
$a_5(X) : -a_7(X), a_8(X), a_9(X).$ $\mathcal{T}(a_{59}(a), 0.2).$
$a_9(X) : -a_8(X).$
$\vdots$
$a_{56}(X) : -a_{58}(X).$
$a_{56}(X) : -a_{59}(X).$
$a_{56}(X) : -a_{60}(X).$
$a_{60}(X) : -a_{59}(X).$

Examples Bk:

$B1$:

$a_0(X) : -a_1(X), a_2(X), a_3(X).$ $\mathcal{T}(a_3(a), 0.1).$
$a_2(X) : -a_1(X).$ $\mathcal{T}(a_6(a), 0.9).$
$a_1(X) : -a_4(X).$ $\mathcal{T}(a_4(a), 0.2).$
$a_1(X) : -a_5(X).$
$a_1(X) : -a_6(X).$
$a_5(X) : -a_4(X).$

(...)

$B10$:

$a_0(X) : -a_1(X), a_2(X), a_3(X).$ $\mathcal{T}(a_3(a), 0.1).$
$a_2(X) : -a_1(X).$ $\mathcal{T}(a_6(a), 0.9).$
$a_1(X) : -a_4(X).$ $\mathcal{T}(a_9(a), 0.1).$

$a_1(X) : -a_5(X).$ $\vdots$
$a_1(X) : -a_6(X).$ $\mathcal{T}(a_{57}(a), 0.1).$
$a_5(X) : -a_4(X).$ $\mathcal{T}(a_{60}(a), 0.9).$
$a_4(X) : -a_7(X), a_8(X), a_9(X).$ $\mathcal{T}(a_{58}(a), 0.2).$
$a_8(X) : -a_7(X).$
$\vdots$
$a_{55}(X) : -a_{58}(X).$
$a_{55}(X) : -a_{59}(X).$
$a_{55}(X) : -a_{60}(X).$
$a_{59}(X) : -a_{58}(X).$

Based on these examples, we obtain the execution times for programs $P1$, $P2$ and $P3$ to solve query $a_0(a)$[7], as presented in tables 1 and 2 and figures 5

[7] the experiments were run using SICStus PROLOG on SPARC workstations. The execution times are presented in milliseconds.

and 6.

	P1	*P2*	*P3*
A1	70	40	30
A2	2300	70	60
A3	84720	110	90
A4	-	140	120
A5	-	180	150
A6	-	220	180
A7	-	250	210
A8	-	280	240
A9	-	320	260
A10	-	360	290

Table 1. Experimental Results - Execution Times for Resolving *Ak*-examples (msecs)

	P1	*P2*	*P3*
B1	110	50	60
B2	7050	80	110
B3	529709	120	150
B4	-	160	190
B5	-	200	250
B6	-	230	290
B7	-	270	340
B8	-	310	370
B9	-	340	410
B10	-	370	470

Table 2. Experimental Results - Execution Times for Resolving *Bk*-examples (msecs)

Both *Ak* and *Bk*-examples show positive results for the implementation employing *solution caching* (*P2*). The program still makes an exhaustive search of all paths to all solutions, but it does not have to recalculate the truth-value of a ***ground query*** once it is already stored.

The *Ak*-examples are also positive with respect to the implementation employing both strategies (*P3*). The order in which clauses are selected for unification is suitable for the *α-β pruning* strategy. This does not occur for the *B*-examples, for which this strategy is totally ineffective.

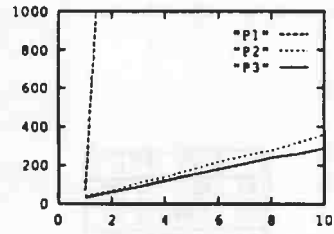

Fig. 5. Experimental Results - Execution Times for Resolving Ak-examples (msecs)

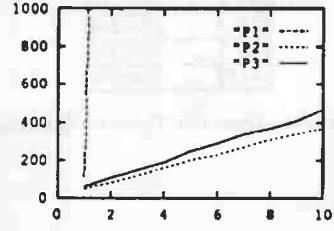

Fig. 6. Experimental Results - Execution Times for Resolving Bk-examples (msecs)

6 Discussion and Conclusions

In this paper we have discussed the problem of efficient reasoning with fuzzy predicates. As the experimental results presented have shown, this is a very relevant problem if we are interested in building practical applications of fuzzy reasoning, since great improvements in performance were achieved employing rather simple optimization techniques.

"Software engineering for logic programming" is a research area at its infancy as yet (see e.g. [KMN93] for a recent reference in this subject), what to say "software engineering for *fuzzy* logic programming". We expect this paper to be the first of many to come exploring the issue of complexity of automated deduction for logic programs with fuzzy predicates.

In our future work, we plan to derive analytic bounds for the complexity of *e-SLDNF* resolution strategies, and verify the realtions between our experimental results and those bounds.

Acknowledgements: the first author is partially supported by FAPESP grant 93/0603-01, and CNPq grant 300041/93-4. The second author is partially supported by CNPq grant 112425/92-5.

References

[Apt87] K. F. Apt. Introduction to Logic Programming. Technical Report CS-R8741, Centre for Mathematics and Computer Science, 1987.

[CdS92] F. S. Correa da Silva. *Automated Reasoning with Uncertainties*. PhD thesis, University of Edinburgh, Department of Artificial Intelligence, 1992.

[CdSRH93] F. S. Correa da Silva, D. S. Robertson, and J. Hesketh. Automated Reasoning with Uncertainties. In M. Masuch and L. Pawlos, editors, *Knowledge Representation and Uncertainty - ch. 5 (forthcoming) - a preliminary version was presented at the APLOC - Applied Logic Conference (Logic at Work), 1992*. Springer Verlag, 1993.

[CL89] L. Cavedon and J. W. Lloyd. A Completeness Theorem for SLDNF Resolution. *Journal of Logic Programming*, 7:177–191, 1989.

[DP87] D. Dubois and H. Prade. Necessity Measures and the Resolution Principle. Technical Report 267, LSI - Universite Paul Sabatier, 1987.

[DP89] D. Dubois and H. Prade. Fuzzy Sets, Probability and Measurement. *European Journal of Operational Research*, 40:135–154, 1989.

[Fit85] M. Fitting. A Kripke-Kleene Semantics for Logic Programs. *Journal of Logic Programming*, 4:295–312, 1985.

[Fit88] M. Fitting. Logic Programming on a Topological Bilattice. *Fundamenta Informaticae*, XI:209–218, 1988.

[Fit90] M. Fitting. Bilattices in Logic Programming. In *Proceedings of the 20^{th} International Symposium on Multiple-valued Logic*, 1990.

[Hin86] C. J. Hinde. Fuzzy Prolog. *International Journal of Man-Machine Studies*, 24:569–595, 1986.

[Hog90] C. J. Hogger. *Essentials of Logic Programming*. Oxford Univ. Press, 1990.

[IK85] M. Ishizuka and K. Kanai. Prolog-ELF Incorporating Fuzzy Logic. In *IJCAI'85 - Proceedings of the 9^{th} International Joint Conference on Artificial Intelligence*, 1985.

[Kle82] E. P. Klement. Construction of Fuzzy σ-algebras Using Triangular Norms. *Journal of Mathematical Analysis and Applications*, 85:543–565, 1982.

[KMN93] H. Ko, D. McAllester, and M. E. Nadel. Lower Bounds for the Lengths of Refutations. *Journal of Logic Programming*, 17:31–58, 1993.

[KS91] M. Kifer and V. S. Subrahmanian. Theory of Generalized Annotated Logic Programs and its Applications. *Journal of Logic Programming (forthcoming)*, 1991.

[Kun87] K. Kunen. Negation in Logic Programming. *Journal of Logic Programming*, 4:289–308, 1987.

[Kun89] K. Kunen. Signed Data Dependencies in Logic Programs. *Journal of Logic Programming*, 7:231–245, 1989.

[Kun90] K. Kunen. Some Remarks on the Completed Database. *Fundamenta Informaticae*, XIII:35–49, 1990.

[Lee72] R. C. T. Lee. Fuzzy Logic and the Resolution Principle. *Journal of the ACM*, 19:109–119, 1972.

[Men87] E. Mendelson. *Introduction to Mathematical Logic (3rd ed)*. Wadsworth & Brooks/Cole, 1987.

[Orc89] I. P. Orci. Programming in Possibilistic Logic. *International Journal of Expert Systems*, 2:79–96, 1989.

[Pea84] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley Publishing Company, 1984.

[Sha83] E. Y. Shapiro. Logic Programming with Uncertainties - a Tool for Implementing Rule-based Systems. In *IJCAI'83 - Proceedings of the 8th International Joint Conference on Artificial Intelligence*, 1983.

[Tur89] D. Turi. Logic Programs with Negation: Classes, Models, Interpreters. Technical Report CS-R8943, Centre for Mathematics and Computer Science, 1989.

[vE86] M. H. van Emden. Quantitative Deduction and its Fixpoint Theory. *Journal of Logic Programming*, 1:37–53, 1986.

[Zad65] L. Zadeh. Fuzzy Sets. *Information and Control*, 8:338–353, 1965.

[Zad88] L. Zadeh. Fuzzy Logic. Technical Report CSLI-88-116, Center for the Study of Language and Information, 1988.

This article was processed using the LaTeX macro package with LLNCS style

RELATÓRIOS TÉCNICOS

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Instituto de Matemática e Estatística da USP

A listagem contendo os relatórios técnicos anteriores a 1992 poderá ser consultada ou solicitada à Secretaria do Departamento, pessoalmente, por carta ou e-mail(mac@ime.usp.br).

J.Z. Gonçalves, Arnaldo Mandel
COMMUTATIVITY THEOREMS FOR DIVISION RINGS AND DOMAINS
RT-MAC-9201, Janeiro 1992, 12 pp.

J. Sakarovitch
THE "LAST" DECISION PROBLEM FOR RATIONAL TRACE LANGUAGES
RT-MAC 9202, Abril 1992, 20 pp.

Valdemar W. Setzer, Fábio Henrique Carvalheiro
ALGOTRITMOS E SUA ANÁLISE (UMA INTRODUÇÃO DIDÁTICA)
RT-MAC 9203, Agosto 1992, 19 pp.

Claudio Santos Pinhanez
UM SIMULADOR DE SUBSUMPTION ARCHITECTURES
RT-MAC-9204, Outubro 1992, 18 pp.

Julio M. Stern
REGIONALIZAÇÃO DA MATRIZ PARA O ESTADO DE SÃO PAULO
RT-MAC-9205, Julho 1992, 14 pp.

Imre Simon
THE PRODUCT OF RATIONAL LANGUAGES
RT-MAC-9301, Maio 1993, 18 pp.

Flávio Soares C. da Silva
AUTOMATED REASONING WITH UNCERTAINTIES
RT-MAC-9302, Maio 1993, 25 pp.

Flávio Soares C. da Silva
ON PROOF-AND MODEL-BASED TECHNIQUES FOR REASONING WITH UNCERTAINTY
RT-MAC-9303, Maio 1993, 11 pp.

Carlos Humes Jr.,Leônidas de O.Brandão,Manuel Pera Garcia
A MIXED DYNAMICS APPROACH FOR LINEAR CORRIDOR POLICIES
(A REVISITATION OF DYNAMIC SETUP SCHEDULING AND FLOW CONTROL IN MANUFACTURING SYSTEMS)
RT-MAC-9304, Junho 1993, 25 pp.

Ana Flora P.C.Humes e Carlos Humes Jr.
STABILITY OF CLEARING OPEN LOOP POLICIES IN MANUFACTURING SYSTEMS (Revised Version)
RT-MAC-9305, Julho 1993, 31 pp.

Maria Angela M.C. Gurgel e Yoshiko Wakabayashi
THE COMPLETE PRE-ORDER POLYTOPE: FACETS AND SEPARATION PROBLEM
RT-MAC-9306, Julho 1993, 29 pp.

Tito Homem de Mello e Carlos Humes Jr.
SOME STABILITY CONDITIONS FOR FLEXIBLE MANUFACTURING SYSTEMS WITH SET-UP TIMES
RT-MAC-9307, Julho de 1993, 26 pp.

Carlos Humes Jr. e Tito Homem de Mello
A NECESSARY AND SUFFICIENT CONDITION FOR THE EXISTENCE OF ANALY. CENTERS IN PATH FOLLOWING METHODS FOR LINEAR PROGRAMMING
RT-MAC-9308, Agosto de 1993

Flavio S. Corrêa da Silva
AN ALGEBRAIC VIEW OF COMBINATION RULES
RT-MAC-9401, Janeiro de 1994, 10 pp.

Flavio S. Corrêa da Silva e Junior Barrera
AUTOMATING THE GENERATION OF PROCEDURES TO ANALYSE BINARY IMAGES
RT-MAC-9402, Janeiro de 1994, 13 pp.

Junior Barrera, Gerald Jean Francis Banon e Roberto de Alencar Lotufo
A MATHEMATICAL MORPHOLOGY TOOLBOX FOR THE KHOROS SYSTEM
RT-MAC-9403, Janeiro de 1994, 28 pp.

Flavio S. Corrêa da Silva
ON THE RELATIONS BETWEEN INCIDENCE CALCULUS AND FAGIN-HALPERN STRUCTURES
RT-MAC-9404, abril de 1994, 11 pp.

Junior Barrera; Flávio Soares Corrêa da Silva e Gerald Jean Francis Banon
AUTOMATIC PROGRAMMING OF BINARY MORPHOLOGICAL MACHINES
RT-MAC-9405, abril de 1994, 15 pp.

Valdemar W. Setzer; Cristina G. Fernandes; Wania Gomes Pedrosa e Flavio Hirata
UM GERADOR DE ANALISADORES SINTÁTICOS PARA GRAFOS SINTÁTICOS SIMPLES
RT-MAC-9406, abril de 1994, 16 pp.

Siang W. Song
TOWARDS A SIMPLE CONSTRUCTION METHOD FOR HAMILTONIAN DECOMPOSITION OF THE HYPERCUBE
RT-MAC-9407, maio de 1994, 13 pp.

Julio M. Stern
MODELOS MATEMATICOS PARA FORMAÇÃO DE PORTFÓLIOS
RT-MAC-9408, maio de 1994, 50 pp.

Imre Simon
STRING MATCHING ALGORITHMS AND AUTOMATA
RT-MAC-9409, maio de 1994, 14 pp.

Valdemar W. Setzer e Andrea Zisman
*CONCURRENCY CONTROL FOR ACCESSING AND COMPACTING B-TREES**
RT-MAC-9410, junho de 1994, 21 pp.

Renata Wassermann e Flávio S. Corrêa da Silva
TOWARDS EFFICIENT MODELLING OF DISTRIBUTED KNOWLEDGE USING EQUATIONAL AND ORDER-SORTED LOGIC
RT-MAC-9411, junho de 1994, 15 pp.

Jair M. Abe, Flávio S. Corrêa da Silva e Marcio Rillo
PARACONSISTENT LOGICS IN ARTIFICIAL INTELLIGENCE AND ROBOTICS.
RT-MAC-9412, junho de 1994, 14 pp.

Flávio S. Corrêa da Silva, Daniela V. Carbogim
A SYSTEM FOR REASONING WITH FUZZY PREDICATES
RT-MAC-9413, junho de 1994, 22 pp.