IEEE Copyright Notice

© 2018 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

This work has been accepted to be published in the proceedings of the 2018 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE 2018) held in Rio de Janeiro (Brazil).
An Efficient Proximity-based Unification Algorithm

Pascual Julián-Iranzo  
Dept. of Information Technologies and Systems,  
University of Castilla-La Mancha.  
Email: Pascual.Julian@uclm.es

Fernando Sáenz-Pérez  
Dept. of Software Engineering and Artificial Intelligence,  
Universidad Complutense de Madrid.  
Email: fernan@sip.ucm.es

Abstract—Unification is a central concept in deductive systems based on the resolution principle. Recently, we introduced a new weak unification algorithm based on proximity relations (i.e., reflexive, symmetric, fuzzy binary relations). Proximity relations are able to manage vague or imprecise information and, in combination with the unification algorithm, allow certain forms of approximate reasoning in a logic programming framework. In this paper, we present a reformulation of the weak unification algorithm and an elaborated method to implement it efficiently.

Index Terms—Fuzzy Logic Programming, Fuzzy Prolog, Weak Unification, Proximity Relations.

I. INTRODUCTION

Fuzzy Logic Programming integrates fuzzy logic and pure logic programming in order to deal with the essential vagueness of some problems by using declarative techniques. Similarity-based Logic Programming is a branch of Fuzzy Logic Programming where the classical SLD resolution Selection-function driven Linear resolution for Definite clauses 1 principle is extended by modifying its classical unification algorithm and replacing it with a fuzzy unification algorithm supported by similarity relations (i.e., reflexive, symmetric, transitive, fuzzy binary relations). It started with the pioneering papers [5]–[7] and [4], where the concept of unification by similarity was first developed. Afterwards, in [17], Maria Sessa defined an extension of the SLD-resolution principle, incorporating a similarity-based unification procedure which is a reformulation of Martelli and Montanari’s unification algorithm [14] where symbols can match if they are similar (instead of syntactically equal). Roughly speaking, the similarity-based unification algorithm states that two terms $f(t_1, \ldots, t_n)$ and $g(s_1, \ldots, s_n)$ weakly unify if the root symbols $f$ and $g$ are similar, with a certain degree, and each of their arguments $t_i$ and $s_i$ pairwise weakly unify. Therefore, the similarity-based unification algorithm does not produce a failure if there is a clash of two syntactically distinct symbols, whenever they are close, but a success with a certain approximation degree, which is computed under a given t-norm (as, e.g., minimum). 2

Bousi-Prolog [10], [16] was the first fuzzy logic programming language that proposed the use of proximity relations (that are reflexive and symmetric, but not necessarily transitive, fuzzy binary relations) as a generalization of similarity relations [11]. In order to deal with proximity relations, Bousi-Prolog has needed to develop new theoretical [9] and conceptual [16] bases. The idea is to generalize the operational mechanism of [17], increasing the expressive power of the resulting language, thanks to the removal of the transitivity restriction that provides users with more freedom to express (vague) knowledge.

In [9], we referred to at least three motivations for using proximity relations:

1) The exclusive use of similarity relations may cause wrong modeling of vague information.
2) The transitivity constraints imposed by similarity relations may produce conflicts with user specifications.
3) Proximity relations are necessary to define semantic unification in terms of a weak unification algorithm [12].

A naïve treatment of proximity relations may cause unexpected severe problems for the operational mechanism of these languages. Also, in [9] we commented some of them (see that reference for details). The first one is that Sessa’s weak unification algorithm cannot be used with proximity relations because it becomes incomplete (i.e., there exist unifiers of two terms which are not computed by the weak unification algorithm). In addition, incompleteness of the weak unification algorithm leads to the incompleteness of the weak SLD resolution procedure, when Sessa’s unification algorithm is used in combination with proximity relations.

In order to take advantage of proximity relations, but avoiding their problems, in [9] we gave a notion of proximity between expressions (terms or atomic formulas) of a syntactic domain and a weak unification algorithm, based on the new notion of approximation, which is able to manage proximity relations properly. Although this definition allowed us to reason about its formal properties, it is not suitable for an efficient implementation. This work is devoted to provide a new reformulation of the proximity-based unification algorithm with the aim of achieving a method for an efficient implementation that solves the problems of the previous one.

1See http://www.fdi.ucm.es/profesor/fernan/sp/JS18c.pdf as a reference for usual logic programming terms.

2In this context, an approximation degree can be thought of as a truth degree, a confidence factor or a degree of belief in the delivered answer. See [17] for the exact description of this algorithm.
implementation, and, thus, to be able to soundly deal with proximity relations, present in many applications (e.g., social networks and GIS-related systems). Also, we make a study to measure the execution cost of the new algorithm compared with both the proximity-based unification algorithm originally defined and the similarity-based unification algorithm.

In the following, we assume familiarity with fuzzy logic [15] and the theory and practice of logic programming [1, 13]. However, before introducing the main contributions of this work, in the next section we recall some concepts and definitions developed in [9].

II. PROXIMITY RELATIONS AND WEAK UNIFICATION

A. Proximity Relations

A binary fuzzy relation on a set which is reflexive and symmetric. A proximity relation is characterized by a set Λ = {λ1, ..., λn} of approximation levels. We say that a value λ ∈ Λ is a cut value. A special, and well-known, type of proximity relations are similarity relations (i.e., transitive proximity relations).

For a proximity relation on a set U, the notion of equivalence class splits into two different concepts: the blocks and the classes of proximity. Given a proximity relation R on a set U, a proximity block of level λ (or λ-block), is a subset of U such that the restriction of R to this subset is a total relation, and maximal with this property. On the other hand, the proximity class of level λ (λ-class) of an element x ∈ U is the subset of those elements of U that are close to x (at a certain level λ). The blocks and the classes of a proximity relation on a set U form coverings of U, but not necessarily partitions.

Finally note that, corresponding to any fuzzy binary relation R on U, there is a labeled digraph (or digraph) G whose nodes (or vertices) are the members of the domain of R and whose labeled arcs are the triples a1a2a3 for which R(a1, a2, a3) is a cut norm.

We are interested in proximity relations on a syntactic domain. In this context, the proximity relation among non-variable symbols is fixed axiomatically, whilst syntactically equal variables are close with degree 1 and, otherwise, with degree 0. The extension of a proximity relation on a syntactic domain to syntactic expressions like terms or atoms is based on the notion of proximity block and uses the notion of “position of a term”.

Positions of an expression e (also called occurrences) are represented by sequences of natural numbers used to address subterms of e. Pos(e) denotes the set of positions of an expression e. If p ∈ Pos(e), e[p] denotes the symbol of e at position p.

Given a proximity relation R on a syntactic domain, two expressions e1 and e2 of a first-order language L are approximate at level λ (or λ-approximate) when they have the same set of positions (i.e. Pos(e1) = Pos(e2)), their symbols, in their corresponding positions, belong to the same λ-block (i.e. for all u ∈ Pos(e1), e1[u] ∈ Bλ iff e2[u] ∈ Bλ) and a certain symbol is always assigned to the same λ-block. When two expressions e1 and e2 are λ-approximate we denote this as: e1 ≈R,λ e2. On the other hand, the proximity degree R(e1, e2) of two expressions of L which are λ-approximate is defined by structural induction, starting from R, as usual. See [9] for a formal definition of these concepts.

B. Unification by proximity

In the context of a proximity relation R on a cut value λ, a substitution θ is a weak unifier of level λ for two expressions e1 and e2 with respect to R (or λ-unifier) if and only if e1θ ≈R,λ e2θ. We say that R(e1, e2) ≥ λ is the degree of unification of e1 and e2 w.r.t. θ and R.

Following [14], in this subsection we define an algorithm to solve weak unification problems, which is based on a transition system characterized by a notion of unification state and a set of transition rules. A weak unification problem seeks for the (weak) most general unifier of two expressions t and s. Formally, it is represented by the equation t ≈ s.

A weak unification state is a tuple (P, S, C, σ) where: P is a (multi-)set of weak unification problems or a failure; S is a set of equations in solved form; C is a set of proximity constraints of level λ and σ is a degree of unification. A set of equations S = {x1 ≈ t1, ..., xn ≈ tn} is in solved form if each xi occurs only once in the set of equations. For any set of equations S in solved form, the corresponding idempotent substitution {x1/t1, ..., xn/tn} is denoted by σ.S.

A proximity constraint a → b is a non-ordered pair of non-variable symbols of L. The set of proximity constraints C defines a crisp binary relation on symbols of L that may be visually represented by a (non-directed) graph G whose nodes (or vertices) are symbols of L and whose edges are proximity constraints. A proximity constraint a → b points out that a and b are in the same block. We say that a path is divergent if it connects two nodes a and b such that R(a, b) < λ. A divergent path is pointing out the existence of nodes (with symbols) assigned to different blocks. A set of proximity constraints is used to detect inconsistencies in “block assignments” for an alphabet symbol. In our framework, we only consider admissible that a certain symbol is assigned to a single block.

III. PROXIMITY-BASED UNIFICATION ALGORITHMS

A. The Proximity-Based Unification Algorithm

The proximity-based unification algorithm is an algorithm to solve weak unification problems, which is based on a transition system characterized by a notion of unification state and a set of transition rules. A weak unification problem seeks for the (weak) most general unifier of two expressions t and s. Formally, it is represented by the equation t ≈ s.

A weak unification state is a tuple (P, S, C, σ) where: P is a (multi-)set of weak unification problems or a failure; S is a set of equations in solved form; C is a set of proximity constraints of level λ and σ is a degree of unification. A set of equations S = {x1 ≈ t1, ..., xn ≈ tn} is in solved form if each xi occurs only once in the set of equations. For any set of equations S in solved form, the corresponding idempotent substitution {x1/t1, ..., xn/tn} is denoted by σ.S.

A proximity constraint a → b is a non-ordered pair of non-variable symbols of L. The set of proximity constraints C defines a crisp binary relation on symbols of L that may be visually represented by a (non-directed) graph G whose nodes (or vertices) are symbols of L and whose edges are proximity constraints. A proximity constraint a → b points out that a and b are in the same block. We say that a path is divergent if it connects two nodes a and b such that R(a, b) < λ. A divergent path is pointing out the existence of nodes (with symbols) assigned to different blocks. A set of proximity constraints is used to detect inconsistencies in “block assignments” for an alphabet symbol. In our framework, we only consider admissible that a certain symbol is assigned to a single block.
during a computation. So, a set of proximity constraints is satisfiable (or successful) if it is not possible to find a path in the proximity constraint graph whose nodes are assigned to different blocks. Otherwise, we say that the set of proximity constraints is unsatisfiable (or fails). In [9] it is defined a function Sat for proximity constraint satisfaction.

The proximity-based unification relation ⇒ acting on weak unification states is the smallest relation defined by the following set of transition rules:

1) Term decomposition:
   (a) \( \{ \{ f(t_n) \approx f(s_m) \} \cup E, S, C, \alpha \} \)
   \( \Rightarrow \{ \{ t_n \approx s_m \} \cup E, S, C, \alpha \} \)
   (b) \( \{ \{ f(t_n) \approx g(s_m) \} \cup E, S, C, \alpha \} \)
   \( \Rightarrow \{ \{ t_n \approx s_m \} \cup E, S, \{ f-g \} \cup C, \alpha \} \),
   if \( R(f, g) = \beta \geq \lambda \) and \( \text{Sat}(\{ f-g \} \cup C) \neq \text{failure} \).

2) Removal of trivial equations:
   \( \{ x \approx x \} \cup E, S, C, \alpha \) \( \Rightarrow \) \( (E, S, C, \alpha) \).

3) Swap:
   \( \{ t \approx t \} \cup E, S, C, \alpha \) \( \Rightarrow \) \( \{ x \approx t \} \cup E, S, C, \alpha \),
   if \( t \notin X \).

4) Var. elimination:
   \( \{ x \approx t \} \cup E, S, C, \alpha \) \( \Rightarrow \) \( \{ E[x/t], S[x/t] \cup \{ x \approx t \}, C, \alpha \} \),
   if \( x \notin \text{Var}(t) \).

5) Failure rule:
   \( \{ f(t_n) \approx g(s_m) \} \cup E, S, C, \alpha \) \( \Rightarrow \) \( \text{failure, S, C, } \alpha \),
   if either \( n \neq m \) or \( R(f, g) \) \( < \lambda \) or \( \text{Sat}(\{ f-g \} \cup C) = \text{failure} \).

6) Occur check:
   \( \{ x \approx t \} \cup E, S, C, \alpha \) \( \Rightarrow \) \( \text{failure, S, C, } \alpha \),
   if \( x \in \text{Var}(t) \).

In the rules above, \( E \) denotes a set of (remaining) weak unification problems.

A weak unification process is formalized as a sequence of transition steps performed using the proximity-based unification relation \( \Rightarrow \). Given a proximity relation \( R \) and a cut value \( \lambda \), the unification of a set of expressions \( P = \{ e_1 \approx e_1', \ldots, e_n \approx e_n' \} \) is obtained by a state transformation sequence starting from an initial state \( (P, \emptyset, \emptyset, 1) \):

\[ (P, \emptyset, \emptyset, 1) \Rightarrow (P_1, S_1, C_1, \alpha_1) \Rightarrow \ldots \Rightarrow (P_n, S_n, C_n, \alpha_n). \]

When the final state \( (P_n, S_n, C_n, \alpha_n) \), with \( P_n = \emptyset \) is reached (i.e., all unification problems have been solved and proximity constraints are compatible), the output of the unification process is a weak most general unifier (wmgu) of level \( \lambda \) \( \sigma_{S_n} \) and unification degree \( \alpha_n \). We say that the expressions in \( P \) are unifiable by proximity with weak most general unifier (wmgu) (of level \( \lambda \)) \( \sigma_{S_n} \) and unification degree \( \alpha_n \). Therefore, the final state \( (\emptyset, S_n, C_n, \alpha_n) \) signals out the unification success. On the other hand, when \( P_n = \text{failure} \), the state transformation sequence ends with failure and the set of expressions is not unifiable.

III. AN EFFICIENT IMPLEMENTATION OF THE WEAK UNIFICATION ALGORITHM

The proximity-based unification algorithm presented in Subsection II-B was defined to reason about its formal properties, but it needs to live with a set of proximity constraints and to check continuously its satisfaction, what is a great source of inefficiencies. Moreover, the definition of the function of satisfaction Sat is too complex and need to compute the transitive closure of the relation represented by the proximity constraints of the terms involved in a query. Also, this process is done at run time, what makes this mechanism inviable for its implementation in a programming language with reasonable times of response.

To remedy the above problems, we propose to act on the following points, which mainly aim to improve the performance of the satisfaction function Sat:

1) At compile time, we analyze the proximity relation \( R \) extracting the set of proximity blocks and assigning to each symbol involved in the relation its corresponding block.

2) Using the information provided by the last analysis, also at compile time, we extend the proximity relation \( R \) into a new relation \( R_B \), enhancing \( R \) with information about the specific \( \lambda \)-block corresponding to the elements involved in the relation entry. That is, given a cut value \( \lambda \), an entry \( R(a, b) = \alpha \) is converted into \( R_B(a, b, B_\lambda) = \alpha \), being \( B_\lambda \) the \( \lambda \)-block to which the related elements belong.

3) Thanks to the explicit use of information about \( \lambda \)-blocks, we can replace the set of proximity constraints by an association list whose elements are pairs formed by the elements in the domain of the relation \( R_B \) and the \( \lambda \)-block assigned to them by \( R_B \). Hence, at run time, if we are testing whether two expressions are unifiable, when contrasting two symbols, e.g., \( a \) and \( b \), instead of generating the proximity constraint \( a \sim b \), we will generate the pairs \( \langle a, B_\lambda \rangle \) and \( \langle b, B_\lambda \rangle \) if there exists an entry \( R_B(a, b, B_\lambda) = \alpha \) in the relation \( R_B \).

4) After all that, we can simplify the definition of the function Sat, converting it in a function which essentially performs a membership test on an association list, what can be done efficiently at run time.

In the following we detail some of the points above sketched.

A. Analysis of the Proximity Relation: \( \Lambda \)-Blocks and Extended Relation

The central point of this analysis is, given a cut value \( \lambda \), the generation of the set of \( \lambda \)-blocks associated to the proximity relation \( R \). The effective computations of \( \lambda \)-blocks is linked with the problem of finding all maximal cliques on an undirected graph \( G \) corresponding to \( R \). The Bron-Kerbosch algorithm [2] is a widely used algorithm for this purpose. It is a branch and bound algorithm which is easy to implement and has been shown to work well in practice. Also, theoretical evidence for the fast performance of the
Bron-Kerbosch algorithm has been presented in [3]. By these reasons, we adapt a variant of the Bron-Kerbosch algorithm with pivoting, which limits the number of recursive calls made by their algorithm [2], [3].

In order to formalize our adapted algorithm we use the following notations: $C$ is the set of vertices belonging to the current (growing) clique of $G$; $P$ is set of candidate vertices which can be added to $C$ (these are prospective nodes which are connected to all nodes in $C$ and using them $C$ can be expanded); $X$ is the set of vertices which are not allowed to be added to $C$ (it contains nodes already processed, i.e., nodes which were previously in $P$ and hence all maximal cliques containing them have already been reported) and $\Gamma_\lambda(u) = \mathcal{K}_\lambda(u) \setminus \{u\}$ is the set of neighbours of the vertex $u$ in $G$. Also, given a set $S$, $|S|$ denotes its cardinality and we use the customary operators on sets.

Algorithm 1:
Input: The set of entries defining the proximity relation $\mathcal{R}$ and a cut value $\lambda$
Output: The set $B$ of labeled $\lambda$-blocks for $\mathcal{R}$
begin
1. $B \leftarrow \emptyset$; $i \leftarrow 1$;
2. Obtain the set of vertices $V$ of graph $G$ corresponding to $\mathcal{R}$;
3. $C \leftarrow \emptyset$; $P \leftarrow V$; $X \leftarrow \emptyset$;
4. call maxCliques($C, P, X$);
5. return $B$;
end

Procedure maxCliques($C, P, X$):
1. if $P = \emptyset$ and $X = \emptyset$ then
   Report $C$ as a maximal clique ($\lambda$-block) and label it as $\mathcal{B}_{\mathcal{R}}^\lambda$;
   $B \leftarrow B \cup \{(\mathcal{B}_{\mathcal{R}}^\lambda, C)\}$; $i \leftarrow i + 1$;
2. endif;
3. Choose a pivot $u \in P \cup X$ in order to maximize $|P \cap \Gamma_\lambda(u)|$; (Tomita et al. [18])
4. for each vertex $v \in P \setminus \Gamma_\lambda(u)$ do
   5. $P \leftarrow P \setminus \{v\}$;
   6. maxCliques($C \cup \{v\}, P \cap \Gamma_\lambda(v), X \cap \Gamma_\lambda(v)$);
7. $X \leftarrow X \cup \{v\}$;
8. endfor
end

Given a proximity relation $\mathcal{R}$, after this point we have a labeled set of $\lambda$-blocks and we can generate the extended proximity relation $\mathcal{R}_\mathcal{B}$. For simplicity of exposition, we omit the precise steps that allow us to carry out this process, which is only minute work. Note that, because a pair of elements $a$ and $b$ in $\mathcal{R}$ can belong two or more $\lambda$-blocks, for each entry in $\mathcal{R}$ it is needed to generate as many entries in $\mathcal{R}_\mathcal{B}$ as $\lambda$-blocks to which these elements belong.

Example 1: Let $\mathcal{R}$ be a proximity relation with entries $\mathcal{R}(a, b) = 0.8$, $\mathcal{R}(a, c) = 0.5$, $\mathcal{R}(b, c) = 0.7$, $\mathcal{R}(b, e) = 0.9$, and $\mathcal{R}(c, e) = 0.8$. This proximity relation is characterized by the set of approximation levels $\Lambda = \{0.5, 0.7, 0.8, 0.9, 1\}$. In particular, for the cut value $\lambda=0.5$, there exist two blocks of level $0.5$: $\mathcal{B}_1^{0.5} = \{a, b, c\}$ and $\mathcal{B}_2^{0.5} = \{b, c, e\}$.

Then for the entry $\mathcal{R}(a, b) = 0.8$ we generate the extended entry $\mathcal{R}_\mathcal{B}(a, b, \mathcal{B}_1^{0.5}) = 0.8$ and so on. But for the particular case $\mathcal{R}(b, c) = 0.7$, we generate two extended entries: $\mathcal{R}_\mathcal{B}(b, c, \mathcal{B}_1^{0.5}) = 0.7$ and $\mathcal{R}_\mathcal{B}(b, c, \mathcal{B}_2^{0.5}) = 0.7$, since the pair $b$ and $c$ belongs to two different $0.5$-blocks.

Ending this subsection, we recall that all these previous steps are performed at compile time.

B. The Weak Unification Algorithm and the Satisfaction Function

In order to adapt the weak unification algorithm introduced in Section II, at the time we maintain the concept of weak unification state we have to replace the old concept of proximity constraint for a new one. If before now a proximity constraint linked a pair of symbols confronted inside a proximity-based unification process, after now a constraint links a symbol with a proximity $\lambda$-block label. We denote these new constraints as "$(\lambda_{\text{block label}})$" and we call them block constraints (of level $\lambda$) from here on.

The proximity-based unification relation on weak unification states, "$\Rightarrow$", is modified adapting rules 1 and 5 as follows:

1. Term decomposition:
   a. $\langle\{f(t_a) \approx f(t_b)\} \cup E, S, C, \alpha\rangle$ ⇒ $\langle\{t_a \approx s_t\} \cup E, S, C, \alpha\rangle$,
   b. $\langle\{f(t_a) \approx g(t_b)\} \cup E, S, C, \alpha\rangle$ ⇒ $\langle\{t_a \approx s_t\} \cup E, S, \{(f: B_{\mathcal{R}}^\lambda), (g: B_{\mathcal{R}}^\lambda)\} \cup C, \alpha\beta\rangle$,
   if $\mathcal{R}_\mathcal{B}(f, g, \mathcal{B}_{\mathcal{R}}^\lambda) = \beta \geq \lambda$ or $\mathcal{Sat}(\{(f: B_{\mathcal{R}}^\lambda), (g: B_{\mathcal{R}}^\lambda)\}, C) \neq \text{failure}$.

5. Failure rule:
   $\langle\{f(t_a) \approx g(t_b)\} \cup E, S, C, \alpha\rangle$ ⇒ $\langle\text{failure}, S, C, \alpha\rangle$,
   if $n \neq m$, $\mathcal{R}_\mathcal{B}(f, g, \mathcal{B}_{\mathcal{R}}^\lambda) < \lambda$ or $\mathcal{Sat}(\{(f: B_{\mathcal{R}}^\lambda), (g: B_{\mathcal{R}}^\lambda)\}, C) = \text{failure}$.

where the function $\mathcal{Sat}$ checks the compatibility or satisfaction of block constraints. Note that, by construction, the sets of block constraints can be understood as association lists, that is, collections of unique keys that are associated to values. Then, the new satisfaction function $\mathcal{Sat}$ can be defined as follows:

Algorithm 2:
Input: Two sets $C_1$ and $C_2$ of block constraints of level $\lambda$
Output: Success or Failure
begin
1. for each $(a: B_{\mathcal{R}}^\lambda) \in C_1$
   1.1. if $(a: B_{\mathcal{R}}^\lambda) = (c: B_{\mathcal{R}}^\lambda)$ or $B_{\mathcal{R}}^\lambda \neq B_{\mathcal{R}}^\lambda$ then return failure
   2. else success endif
2. endfor
end

A notable fact to mention is that many Prolog systems (e.g., SICStus Prolog and SWI-Prolog) provide libraries with predicates to manage association lists efficiently. There, association lists are implemented by using AVL trees\(^7\) what allows predicates for inserting a key, changing an association or fetching a single element which are $\mathcal{O}(\log(n))$ worst-case time operations, where $n$ denotes the number of elements in the association list. The logarithmic overhead is a very acceptable behaviour in practice and considerably better than

---

\(^7\)AVL trees, i.e., they are subject to the Adelson-Velskii-Landis balance criterion.
the constant time complexity, \(O(n)\), obtained by predicates for manipulating standard lists.

Finishing this subsection we note the slight notational variation that we will use: given two expressions \(e_1\) and \(e_2\), if there is a successful transition sequence, \(\{e_1 \approx e_2\}, id, T\) \(\Rightarrow^*\) \((\theta, \alpha, C)\), then we write that \(\text{wmgu}_\alpha^\mu(e_1, e_2) = (\theta, \alpha, C)\), being \(\theta\) the \(\lambda\)-wmgu of \(e_1\) and \(e_2\), \(\alpha \geq \lambda\) is their unification degree, and \(C\) the set of (compatible) block constrains.

In order to illustrate the proposed weak unification algorithm and some of the mentioned problems it solves, we present the following example:

**Example 2:**

Let \(\mathcal{R}\) be a proximity relation defined by the entries \(\mathcal{R}(p, q) = 0.9\), \(\mathcal{R}(a, b) = 0.8\) and \(\mathcal{R}(b, c) = 0.75\), whose associated t-norm is the minimum t-norm and the cut value \(\lambda = 0.75\). This proximity relation has three 0.75-blocks: \(B_{1}^{0.75} = \{p, q\}\), \(B_{2}^{0.75} = \{a, b\}\) and \(B_{3}^{0.75} = \{b, c\}\). Therefore it can be extended into the relation \(\mathcal{R}\mathcal{B}\) with entries: \(\mathcal{R}\mathcal{B}(p, q, B_{1}^{0.75}) = 0.9\), \(\mathcal{R}\mathcal{B}(a, b, B_{2}^{0.75}) = 0.8\) and \(\mathcal{R}\mathcal{B}(b, c, B_{3}^{0.75}) = 0.75\).

For the expressions \(p(x, x)\) and \(p(b, c)\), in order to determine whether they are unifiable by proximity, we build the initial state configuration:

\[
\langle \{p(x, x) \equiv p(b, c)\}, id, \emptyset, 1 \rangle
\]

and then we apply the unification rules, above specified, until a configuration of success or failure is reached:

\[
\Rightarrow_1 \langle \{x \cong b, x \cong c\}, id, \emptyset, 1 \rangle
\]

\[
\Rightarrow_2 \langle \{b \cong c\}, \{x / b\}, \emptyset, 1 \rangle
\]

\[
\Rightarrow_3 \langle \emptyset, \{x / b\}, \{\langle b : B_{1}^{0.75}\rangle, \langle c : B_{3}^{0.75}\rangle\}, 0.75 \rangle
\]

Since \(\mathcal{R}\mathcal{B}(b, c, B_{1}^{0.75}) = 0.75 \geq \lambda\) and \(\text{Sat}(\{\langle b : B_{1}^{0.75}\rangle, \langle c : B_{3}^{0.75}\rangle\}, \emptyset) \neq \text{failure}\), the final state is a successful configuration and the output of the unification process is: \(\langle x / b\rangle, \{\langle b : B_{3}^{0.75}\rangle, \langle c : B_{3}^{0.75}\rangle\}, 0.75\rangle\). We say that the expressions \(p(x, x)\) and \(p(b, c)\) are unifiable with 0.75-wmgu \(x / b\), unification degree 0.75 and a set of compatible block constraints \(\{\langle b : B_{3}^{0.75}\rangle, \langle c : B_{3}^{0.75}\rangle\}\).

Now consider the expressions \(p(x, x)\) and \(q(a, c)\). In this case, the unification process leads to the sequence of steps:

\[
\Rightarrow_1 \langle \{x \cong a, x \cong c\}, id, \emptyset, 1 \rangle
\]

\[
\Rightarrow_2 \langle \{a \cong c\}, \{x / a\}, \{\langle p : B_{1}^{0.75}\rangle, \langle q : B_{1}^{0.75}\rangle\}, 0.9 \wedge 1 \rangle
\]

\[
\Rightarrow_3 \langle \text{failure}, \{x / a\}, \{\langle p : B_{1}^{0.75}\rangle, \langle q : B_{1}^{0.75}\rangle\}, 0.9 \rangle
\]

Because \(\mathcal{R}(a, c) = 0 < \lambda\) the final state is a failure configuration and the expressions \(p(x, x)\) and \(q(a, c)\) are not 0.75-unifiable.

Finally, consider the expressions \(p(b, b)\) and \(q(a, c)\). In this case, the unification process leads to the sequence of steps:

\[
\Rightarrow_1 \langle \{b \cong a, b \cong c\}, id, \emptyset, 1 \rangle
\]

\[
\Rightarrow_2 \langle \{b \cong c\}, id, \{\langle p : B_{1}^{0.75}\rangle, \langle q : B_{1}^{0.75}\rangle\}, 0.9 \wedge 1 \rangle
\]

\[
\Rightarrow_3 \langle \text{failure}, id, \{\langle b : B_{1}^{0.75}\rangle, \langle a : B_{1}^{0.75}\rangle, \langle p : B_{1}^{0.75}\rangle, \langle q : B_{1}^{0.75}\rangle\}, 0.8 \wedge 0.9 \rangle
\]

Note that, in the examples, we are directly using bindings instead of equations in solved form in the second component of the unification states, which ends in a failure configuration since \(\text{Sat}(\{\langle b : B_{1}^{0.75}\rangle, \langle c : B_{3}^{0.75}\rangle\}, \{\langle b : B_{2}^{0.75}\rangle, \langle a : B_{2}^{0.75}\rangle, \langle p : B_{1}^{0.75}\rangle, \langle q : B_{1}^{0.75}\rangle\}) = \text{failure}\). This is because the symbol \(b\) is bound to different 0.75-blocks along the unification process. Therefore the expressions \(p(b, b)\) and \(q(a, c)\) are not 0.75-unifiable.

It is noteworthy that Sessa’s unification algorithm [17] cannot be used with proximity relations because it becomes incomplete (although it remains complete for similarity relations, the class of fuzzy relations for which it was conceived). In fact, if we apply Sessa’s unification algorithm on this example, for the expressions \(p(x, x)\) and \(p(a, c)\), and using a naive concept of proximity, there exist unifiers of these two expressions which are not computed by Sessa’s unification algorithm (see Example 2 in [9] for a detailed explanation). However, our definitions prevent this problem, as it was demonstrated in [9] and Example 2 illustrates.

IV. IMPLEMENTING THE PROXIMITY-BASED WEAK UNIFICATION ALGORITHM

The proximity-based weak unification algorithm specified in Subsection III-B is implemented by the predicate `weak_unify/6` listed below. It is thought as a predicate which receives two expressions, `Term1` and `Term2`, a cut value `Lambda` and a set of block constraints `Cin` (in general from a former weak unification), and returns an approximation degree `Degree` and a set of compatible block constraints `Cout` if they are unifiable by proximity:

```prolog
weak_unify(?Term1, ?Term2, +Lambda, +Cin,-Cout,-Degree) :-
  % Term decomposition
  compound(Term1), compound(Term2), !,
  Term1 =.. [Functor1|Args1],
  Term2 =.. [Functor2|Args2],
  length(Args1, A arity),
  length(Args2, A arity),
  sim(Functor1, Functor2, Block, DegreeFunct),
  DegreeFunct => Lambda,
  sat([Functor1:Block, Functor2:Block], Cin, Cin1),
  weak_unify_args(Args1,Args2,Lambda,
  Cin1,Cout,DegreeArgs),
  Degree is min(DegreeFunct, DegreeArgs).
```

```prolog
weak_unify(Term, Variable, _Lambda, Cin, Cin, 1) :-
  % Term/variable swap + Variable removal
  nonvar(Term), var(Variable), !,
  % occur_check(Variable, Term),
  Variable = Term.
```

The first clause corresponds to the transition rule 1 (Term decomposition) in Subsection III-B, which traverses the tree representation of both terms to check its compatibility: First, root symbols are checked to verify that they are close according to the relation \(RB\) (which is internally represented by the predicate `sim(7Symbole1,7Symbole2,7Block,7Degree)`), and then with a satisfiability predicate `sat/3`, which essentially implements the function defined by Algorithm 2.
This predicate takes in its first argument Ctrs a list of proximity constraints, each one relating a symbol with a block (Symbol:Block); in its second argument Cin the input block constraints; and in its third one Cout it returns the output block constraints. If Ctrs are satisfiable with respect to Cin, it returns the union in Cout; otherwise, it fails. The predicate sat/3 is implemented as follows:

\[
\begin{align*}
\% \text{sat}(+	ext{Ctrs}, +\text{Cin}, -\text{Cout}) & \rightarrow \\
\text{sat}([], \text{Cin}, \text{Cin}). \\
\text{sat}([\text{Ctr} | \text{Ctrs}], \text{Cin}, \text{Cout}) & \leftarrow \\
\text{sat}_\text{ctr}(\text{Ctr}, \text{Cin}, \text{Cin}), \\
\text{sat}(_\text{Ctrs}, \text{Cin1}, \text{Cout}). \\
\end{align*}
\]

\[
\text{sat}_\text{ctr}(\text{Symbol:Block}, \text{Cin}, \text{Cout}) :- \\
\text{get_assoc}(\text{Symbol, CIn, Block}), !, \\
\text{Block} = \text{Block1}. \\
\]

\[
\text{sat}(\text{Ctrs}, \text{Cin1}, \text{Cout}). \\
\]

This implementation uses the SWI-Prolog association list library assoc, which implements efficient AVL trees for search (get_assoc/3) and updating (put_assoc/3).

Following with the predicate weak_unify/6, there is a call to weak_unify_args to continue with the unification of the arguments of the term, checking if the terms in the lists Args1 and Args2 can unify pairwise, and returns the minimum (\(t\)-norm) approximation degree Degree of the unifications:

\[
\begin{align*}
\% \text{weak_unify}_\text{args}(\?\text{Args1}, \?\text{Args2}, +\text{Lambda}, \\
\% +\text{Cin}, -\text{Cout}, ?\text{Degree}) & \rightarrow \\
\text{weak_unify}(_\text{Args1}, _\text{Args2}, \text{Lambda}, \text{Cin}, \text{Cout}, \text{Degree}) :- \\
\text{weak_unify}(\text{Args1}, \text{Args2}, \text{Lambda}, \text{Cin}, \text{Cout}, \text{DegreeArg}), \\
\text{weak_unify}_\text{args}(\text{Args1}, \text{Args2}, \text{Lambda}, \text{Cin}, \text{Cout}, \text{DegreeArg}), \\
\text{Degree} = \text{min}(\text{DegreeArg}, \text{DegreeArgs}). \\
\end{align*}
\]

The transition rules 2, 3, 4, and 6 are implemented in the second and third clauses of the predicate weak_unify/6. The failure rule (5) is included in the first clause of this predicate: Either the terms have different arities (tested with the built-in length/2, the resulting approximation degree is below the \(\lambda\)-cut, or the symbols belong to different blocks (satisfiability fails). The occur check rule (6) is commented in the code above because most implementations omit them for performance purposes. However, we also provide its implementation in case of need, in the predicate:

\[
\begin{align*}
\text{occur_check}\text{(Variable, Term)} & :- \\
\text{term_variables}(\text{Term}, \text{Variables}), \% \text{Needs} \\
\text{\(\backslash+\) memberchk_eq(Variable, Variables)}. \% \text{hprolog.pl}
\end{align*}
\]

A. Differences with Other Algorithms

The original algorithm proposed by Sessa can be implemented in a similar way to our proposal by eliding the \(\lambda\)-cut and the satisfiability predicate. Thus, the prototype of the main weak unification predicate would simply be weak_unify(?Term1,?Term2,?Degree), and the calls to the predicate sat/3 and the comparison of the \(\lambda\)-cut with the approximation degree would be removed. Notably, Sessa’s algorithm works directly with the relation \(\mathcal{R}\) which is internally represented by means of a predicate sim(?Symbol1,?Symbol2,?Degree). From here on, we will call this algorithm A1.

As well, implementing the proposal in [9], which we call A2, follows a similar track to ours, which we call A3. The prototype for the predicate weak_unify/6 is the same, and there are two main differences w.r.t. A3. First, A3 works directly with the relation \(\mathcal{R}\) (as in the case of A1). And, second, the satisfiability predicate takes a single proximity constraint Symbol1--Symbol2 as its first argument (stating that Symbol1 has been weakly unified with Symbol2), and input and output proximity constraint stores for the two remaining arguments. This predicate checks if a set of proximity constraints \(\mathcal{C}\) is consistent, i.e., if there are no divergent paths in the graph of proximity constraints. A path between vertexes X and Y is divergent if X is proximate to Y with approximation degree \(D(\text{sim}(X,Y,D))\). For two elements to belong to the same \(\lambda\)-block, it is necessary that \(\text{sim}(X,Y,D)\) exists. Then, if \(\text{sim}(X,Y,D)\) exists is because X belongs to a block and Y to another. This is signalling out that an inner element in the divergent path is playing two roles. This can be implemented as follows:

\[
\begin{align*}
\text{sat}(\text{E1}\text{--}\text{E2}, \text{Cin}, \text{Cout}) & :- \\
\text{sat}_\text{ctr}(\text{Cin1}, \text{Cout}), !, \\
\text{clear}_\text{ctrs}, \text{false} \\
\text{\(\backslash\) clear}_\text{ctrs}. \\
\end{align*}
\]

Note that if there are no divergent paths, the condition \((\text{ctrs}_\text{path}(X,Y), \lambda+ \text{sim}(X,Y,D))\) generates the transitive closure of the relation \(\mathcal{R}\) by backtracking. Also, it is not necessary to compute the whole transitive closure, it suffices to detect the first divergent path and fail.

Another point worth to mention in this implementation is that the Prolog dynamic database has been used for dealing with the set of constraints to check for satisfiability. The predicate load_ctrs loads the constraints \(\mathcal{C}\) in the dynamic database (with the built-in assertz), and the predicate ctrs_path(X,Y) find possible paths by referring to the entries X--Y as stored in such database. Notably, the cost of using this database is less than passing a list of entries and traversing it, even with ordered and (nested) association lists. SWI-Prolog builds indexes for stored predicates so that the cost of accessing individual entries is faster than maintaining in the heap the necessary data structures for passing the lists.

V. Performance

This section includes an experimental analysis for the performance of the three different unification algorithms to test whether our proposal would fit an efficient implementation of a weak unification algorithm. We chose an experiment including a maximal clique, parametric in the number of nodes \(n\), in order to test in particular scalability. Each node in this clique is an integer (for the sake of easy instance generation) related
with each other with an approximation degree of 1 so that the unification succeeds for any \( \lambda \)-cut. As a unification goal, we selected to unify two lists with length \( n - 1 \): \([1, \ldots, n - 1]\), and \([2, \ldots, n]\), therefore compelling to traversing all nodes.

As a test platform, we used a Windows 10 64-bit OS on an Intel Xeon CPU E3-1505M v5 (4 physical cores) running at 2.8 GHz at its peak, with 16GB RAM. For the Prolog system, we used SWI-Prolog 64 bits, version 7.4.2. Running times are in seconds, and each test has been run 10 times, discarding the minimum and maximum values, and computing the average. Times have been got with the built-in predicate \texttt{time/3}, which in particular returns the wall time for the execution of a goal.

### A. Comparing A1 to A3

Algorithm A1, though incomplete for proximity relations, should be the fastest one because it only relies on the relation \( \mathcal{R} \) for testing how two symbols are related. Thus, we test how much overhead we add in the execution of A3 with respect to A1 to find out the cost of implementing a unification algorithm (as is the case of A3) that does not lose completeness when working with proximity relations.

Table I includes several instances of the experiment ranging from 500 to 1500 nodes in the maximal clique (2.5K to 2.25M arcs, including those for symmetry and excluding those for reflexivity; this generates a file for the relation \( \mathcal{RB} \) of up to 34MB).

<table>
<thead>
<tr>
<th>Nodes</th>
<th>A1</th>
<th>A3</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>0.150</td>
<td>0.183</td>
<td>21.75</td>
</tr>
<tr>
<td>600</td>
<td>0.208</td>
<td>0.225</td>
<td>8.13</td>
</tr>
<tr>
<td>700</td>
<td>0.258</td>
<td>0.292</td>
<td>12.86</td>
</tr>
<tr>
<td>800</td>
<td>0.369</td>
<td>0.399</td>
<td>7.97</td>
</tr>
<tr>
<td>900</td>
<td>0.461</td>
<td>0.527</td>
<td>14.33</td>
</tr>
<tr>
<td>1000</td>
<td>0.618</td>
<td>0.751</td>
<td>21.45</td>
</tr>
<tr>
<td>1100</td>
<td>0.746</td>
<td>0.788</td>
<td>5.68</td>
</tr>
<tr>
<td>1200</td>
<td>0.855</td>
<td>0.939</td>
<td>9.76</td>
</tr>
<tr>
<td>1300</td>
<td>0.935</td>
<td>0.980</td>
<td>4.85</td>
</tr>
<tr>
<td>1400</td>
<td>1.072</td>
<td>1.137</td>
<td>6.09</td>
</tr>
<tr>
<td>1500</td>
<td>1.214</td>
<td>1.316</td>
<td>8.48</td>
</tr>
</tbody>
</table>

On average, there is an overhead of about 11%, which is a small price to pay for completeness. Note that for small numbers for timings as these, the underlying multi-processing OS may interfere the results. This, coupled with the granularity of the clock, explains the different overheads that are computed for different instances. Figure 1 illustrates this graphically.

### B. Comparing A2 to A3

Despite the above encouraging results, recall that A3 requires the relation \( \mathcal{RB} \), which is computed at compile-time. So, we are interested in determining how much time is employed in building such relation for big cliques as those which will be considered in this section. In addition, we compare the execution time of A2 to A3, but also including its compilation time. This would be useful to know the performance of A3, compared to A2, for a single run (obviously, for practical uses, many runs are expected for a single compilation, making negligible the compilation time).

Table II includes times for compilation and execution for both algorithms, showing the speed-up of A3 with respect to A2 (as the ratio of column values A2/A3). By contrast to the rest of measures, the column A2 includes the time for a single run of the instance, due to its large numbers.

<table>
<thead>
<tr>
<th>Nodes</th>
<th>A2</th>
<th>A3</th>
<th>Speed-Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>840</td>
<td>41.15</td>
<td>20.42</td>
</tr>
<tr>
<td>600</td>
<td>1800</td>
<td>69.90</td>
<td>25.75</td>
</tr>
<tr>
<td>700</td>
<td>3472</td>
<td>106.10</td>
<td>32.69</td>
</tr>
<tr>
<td>800</td>
<td>5644</td>
<td>154.52</td>
<td>36.50</td>
</tr>
<tr>
<td>900</td>
<td>8967</td>
<td>216.34</td>
<td>41.42</td>
</tr>
<tr>
<td>1000</td>
<td>13,687</td>
<td>292.78</td>
<td>46.69</td>
</tr>
<tr>
<td>1100</td>
<td>19,863</td>
<td>377.05</td>
<td>52.65</td>
</tr>
<tr>
<td>1200</td>
<td>27,885</td>
<td>482.93</td>
<td>57.72</td>
</tr>
<tr>
<td>1300</td>
<td>43,033</td>
<td>614.35</td>
<td>69.18</td>
</tr>
<tr>
<td>1400</td>
<td>&gt;&gt;</td>
<td>761.17</td>
<td>0.00</td>
</tr>
<tr>
<td>1500</td>
<td>&gt;&gt;</td>
<td>933.14</td>
<td>0.00</td>
</tr>
</tbody>
</table>

From the numbers in the table we can see that, even considering the compilation stage, A3 clearly outperforms A2. For big cliques (1400 and beyond – though hardly expected to be found in practice) times are manageable for A3, but represents roughly a computation day for 1400 nodes in A2 (marked in the text with \( >> \)). Therefore the claimed assumption about the efficiency of A3 is confirmed.

Despite this, the slope of A3 is way below A2 as Figure 2 illustrates by comparing both curves in a single graph. An exponential estimation of the numbers in Table II (\( n \) for the number of nodes, and \( t \) for the elapsed time) provides: \( t = 718.32 \cdot e^{0.4723 \cdot n} \) for A2, and \( t = 40.72 \cdot e^{0.3024 \cdot n} \) for A3.

### VI. Conclusions and Future Work

In this paper, we introduced an elaborated method to efficiently implement the weak unification algorithm proposed in [9]. This method consists of transferring the greatest number of useful tasks to achieve the unification of two expressions to the compilation phase of a program and to define a function for
checking the satisfaction of proximity constraints that operates efficiently at run-time.

We claim that this method for implementing the weak unification algorithm is able to fix some incompleteness problems as found in early versions of the WSLD resolution procedure.

To the best of our knowledge, this is the first efficient implementation of an algorithm which combines syntactic unification and proximity relations safely. Performance results support the effectiveness of our proposal.

As a matter of future work, we want to integrate this new weak unification algorithm, based on proximity relations, into fuzzy logic programming systems like Bousi\textendash Prolog [10] and FuzzyDES [8].

Finally, we envision that proximity $\lambda$-blocks can be useful to model the notion of context inside lexical ontologies, and our overall proposal can thus be useful to perform contextual reasoning in that frameworks. We would like to explore this path in a near future.

REFERENCES