Multiconditional Approximate Reasoning with Continuous Piecewise Linear Membership Functions

P.M. van den Broek
Department of Computer Science, University of Twente,
P.O.Box 217, 7500 AE Enschede, the Netherlands
email : pimvdb@cs.utwente.nl

Abstract. It is shown that, for some intersection and implication functions, an exact and efficient algorithm exists for the computation of inference results in multiconditional approximate reasoning on domains which are finite intervals of the real numbers, when membership functions are restricted to functions which are continuous and piecewise linear. An implementation of the algorithm is given in the functional programming language Miranda.

1. Introduction

In a previous paper [1], we have considered the problem of computing inference results for fuzzy reasoning in the case of a single rule. We have given algorithms to compute exact results when the fuzzy sets have membership functions which are continuous and piecewise linear on domains which are finite intervals of the real numbers. These algorithms are efficient, since they do not require any discretization. These results can however not be generalised to the case of multiconditional reasoning, since this would imply the use of piecewise linear functions of more than one variable, which is prohibitively complex.

Instead, in this paper we present a completely different approach. Apart from the extension to multiple rules, a further advantage of the new approach is that it is generic, in the sense that all intersections and implications are treated in the same way, whereas in [1] in all cases a different algorithm was needed. The price to be paid, however, lies in the efficiency of the algorithm; when restricted to the case of a single rule, the approach of this paper is less efficient than the approach in [1]. However, also the new approach has the advantage that no discretization is necessary.

In multiconditional approximate reasoning with generalised modus ponens:

Rule 1	If	$X=A_1$ then	$Y=B_1$
Rule 2	If	$X=A_2$ then	$Y=B_2$
..	..		..
Rule n	If	$X=A_n$ then	$Y=B_n$
Fact		$X=A'$	
--			
Conclusion			$Y=B'$

one calculates the fuzzy set B’ from the rules, the fact, some intersection function I and some implication function J in one of four possible ways (Klir and Yuan [2], page 320):

$$B'(y) = \max_i \{\sup_x (I\ (A'(x), J(A_i(x), B_i(y))))\} \tag{1}$$
$$B'(y) = \min_i \{\sup_x (I\ (A'(x), J(A_i(x), B_i(y))))\} \tag{2}$$
$$B'(y) = \sup_x (I\ (A'(x), \max_i \{J(A_i(x), B_i(y))\})) \tag{3}$$
$$B'(y) = \sup_x (I\ (A'(x),\min_i \{J(A_i(x), B_i(y))\})) \tag{4}$$

When the domains of the fuzzy sets are finite, the computation of B’ presents no difficulties. Here we are interested in the case where the domains are finite intervals of the real numbers. As in [1], we only consider membership functions which are continuous and piecewise linear. A function is said to be piecewise linear if it is linear in all but a finite number of points. The idea of restricting membership functions in order to simplify the calculations is not new; for instance Dubois, Martin-Clouaire and Prade [3] used trapezoidal functions, and De Baets and Kerre [4] used triangular functions.

In the equations (1) and (2), the result is obtained as an aggregation of the results for separate rules. So in these cases the algorithms of [1] may be used, and the aggregation of the results for the separate rules is straightforward. Equation (3) can be rewritten to equation (1), using the fact that I is non-decreasing in its second argument. Therefore, the only case which needs special attention, and which is treated in this paper, is the case of equation (4).

Our approach is only feasable for those intersection functions and implication functions which have the property that when A’, A_i, and B_i are continuous and piecewise linear, also B’ will be continuous and piecewise linear. Therefore we restrict, as has been done in [1], the intersection function to be either the standard intersection or the bounded difference, and the implication function to be either the Lukasiewicz implication, the Kleene-Dienes implication, the Early-Zadeh implication or the Willmott implication. The Mamdani implication function (which is the minimum function, not a real implication function) need not be treated here, since in this case equation (4) can be written as

$$B'(y) = \sup_x (I\ (A'(x),J(\min_i\ \{A_i(x)\},\min_i\ \{B_i(y)\}))) \tag{5}$$

which shows that B’ can be computed as in the single rule case.

Our approach to compute the piecewise linear function B’ from equation (4) consists of three steps. In the first step, in section 2, the problem is reduced to the problem of solving equation (4) for the special case where all functions B_i are linear. In the second step, in section 3, the problem is further reduced to the problem of solving equation (4) for the special case where all functions A’, A_i and B_i are linear. This special case is solved in the third step, in section 4. In this section, the case of standard intersection and Lukasiewicz implication is elaborated. In section 5 we show how the results of section 4 should be accommodated in order to apply to the other intersection and implication functions. We conclude in section 6 with an example, taken from Mizumoto [6].

We will derive our results in a rather informal way; rigorous proofs of our results are however straightforward, and therefore not given in this paper. On the other hand, complete implementations of the algorithms will be given in the functional programming language Miranda (Turner [5]), since Miranda provides an excellent formalism for the notation of algorithms, and this notation is executable.

2. Preliminaries

A continuous piecewise linear function (plf, for short) will be represented as a list of tuples of numbers. The Miranda type definition is

```
plf == [(num,num)]
```

The list $[(x_0,y_0),(x_1,y_1),...(x_n,y_n)]$ with $x_0<x_1<...<x_n$ will represent the plf f on the closed interval $[x_0,x_n]$ which is linear on all intervals $[x_i,x_{i+1}]$ and whose values are determined by $f(x_i) = y_i$. A rule of the form “If $X=A_1$ then $Y=B_1$” will be represented by the tuple (A_1,B_1). The Miranda type definition is

```
rule == (plf,plf)
```

Further, there are types for the identification of intersections and implications, defined respectively by

```
intersection ::= Min | BD
implication ::= Willmott | Lukasiewicz | Kleene_Dienes | Early_Zadeh
```

The aim of this paper is to define the function *mult_app_reas* with type definition

```
mult_app_reas :: intersection -> implication -> [rule] -> plf -> plf
```

which, when given an intersection I, an implication J, a list of rules (A_i,B_i) and a plf A’, returns the plf B’ defined by equation (4).

3. First reduction of the problem

As the first step, we will define the function *mult_app_reas* in terms of the function *mult_app_reas2* which has the same type definition:

```
mult_app_reas2 :: intersection -> implication -> [rule] -> plf -> plf
```

and which solves the same problem in the special case where all plfs B_i are linear. The interval on which the plfs B_i are defined is partitioned into a set of subintervals such that each B_i is linear on each subinterval. The result of the function *mult_app_reas* for the whole interval is obtained as a combination of the results for the subintervals, obtained from the function *mult_app_reas2*. This leads to the following specification of *mult_app_reas* in Miranda:

```
mult_app_reas int impl rules fact =
  (normalize.combine.map result) intervals
  where
  result interval = mult_app_reas2 int impl (newrules interval) fact
  ys = (sort.mkset.concat.map(map fst).map snd) rules
  intervals = zip (ys, tl ys)
  combine [x] = x
  combine (x:xs) = x ++ tl (combine xs)
  newrules (y,y’) = [(a,[(y,apply b y),(y’,apply b y’)])|(a,b)<-rules]
```

Note that this first step is independent of the intersection and the implication. The function `normalize` takes a plf A as argument, and returns the same A with a “normalised” representation: points where A happens to be linear are removed. The function `apply` takes a plf A and a number x as arguments, and returns A(x).

4. Second reduction of the problem

As the second step, we will define the function *mult_app_reas2* in terms of the function *mult_app_reas3* which has the same type definition:

```
mult_app_reas3 :: intersection -> implication -> [rule] -> plf -> plf
```

and which solves the same problem in the special case where all plfs A', A_i and B_i are linear. We proceed as in the previous section. The interval om which the functions A' and A_i are defined is partitioned into subintervals where the functions A' and A_i are linear; the problem is solved on each of these intervals by means of the function *mult_app_reas3* and these results are combined. From equation (4) it follows that combination here means taking the maximum.

This leads to the following specification of *mult_app_reas2* in Miranda:

```
mult_app_reas2 int impl rules fact
  = (combine.map result) intervals
    where
    result interval
      = mult_app_reas3 int impl (newrules interval) (newfact interval)
    xs = (sort.mkset.concat.map(map fst).(fact:).map fst) rules
    intervals = zip (xs, tl xs)
    combine [x] = x
    combine (x:xs) = maxplf x (combine xs)
    newrules (x,x') = [([(x,apply a x),(x',apply a x')],b)|(a,b)<-rules]
    newfact (x,x') = [(x,apply fact x),(x',apply fact x')]
```

Note that also this second step is independent of the intersection and the implication. The function `maxplf` takes two plfs A and B as arguments, and returns the plf C with
C(x) = max (A(x),B(x)).

5. Solution of the reduced problem

In this section we will solve the original problem for the special case where all plfs A', A_i and B_i are linear. Here we consider the case were the intersection is the standard intersection, defined by:

$$I(x,y) = \min(x,y) \tag{6}$$

and the implication is the Lukasiewicz implication, defined by:

$$J(x,y) = \min(1, 1-x+y) \tag{7}$$

Equation (4) in this case reads

$$B(y) = \sup_x (\min(A'(x), \min_i \{1-A_i(x)+B_i(y)\})) \tag{8}$$

It is straightforward to compute B'(y) for any given y in the domain [y,y']. So, if we would know the set of values for y where B' is not linear, say $\{y_1, y_2, .., y_n\}$ with $y<y_1<y_2<$

$<y_n<y'$ then the plf B' could be computed to be
$[(y,B'(y)),(y_1,B'(y_1)),(y_2,B'(y_2)),....,(y_n,B'(y_n)),(y',B'(y'))]$. The problem is to compute the set $\{y_1,y_2,..,y_n\}$. This is the main problem of this paper. At this point, it is crucial to observe that it is sufficient if we compute a finite set which contains the set $\{y_1,y_2,..,y_n\}$. We will calculate such a superset by means of a function with type definition

```
superset :: intersection -> implication -> [rule] -> plf -> [num]
```

Then the implementation of the function `mult_app_reas3` is straightforward:

```
mult_app_reas3 int impl rules [(x0,a0'),(x1,a1')]
  = (y,b int impl y) :
    [(y2,b int impl y2)|y2<-superset int impl rules [(x0,a0'),(x1,a1')]]
    ++ [(y',b int impl y')]
    where
    y = (fst.hd.snd.hd) rules
    y' = (fst.hd.tl.snd.hd) rules
    b Min impl y
       = sup (minplf [(x0,a0'),(x1,a1')]
         (foldl1 minplf [f Min impl a (apply b y) | (a,b) <- rules] ))
    f Min Lukasiewicz [(x0,a0),(x1,a1)] by
       =[(x0,1-a0+by),(x1,1-a1+by)]
```

The function `sup` takes a plf A as argument and returns $\sup_x A(x)$. The function `minplf` takes two plfs A and B as arguments, and returns the plf C with C(x) = min (A(x),B(x)). Note that most of the coding above is in fact a straightforward translation of the equations (6), (7) and (8).

We will now take a closer look at equation (8). We see that it has the form $B'(y) = \sup_x C(x,y)$, where C(x,y), for fixed y, is a function of x which is defined by applying the minimum operation on a number of linear functions. In the case of equations (6) and (7) these functions are A'(x) and $1+A_i(x)-B_i(y)$. In the other cases both the mimimum and the maximum operations are applied to a number of linear functions. As a consequence, C(x,y) is, for fixed y, a plf which coincides everywhere on its domain with at least one of the set of linear functions. So, its supremum is reached at an intersection point, which is a point where either two lines intersect or a line intersects the boundary. The value of B'(y) always is the value of one of the functions at an intersection point.

We have to consider what happens when y varies. Lines corresponding to functions like A'(x) do not vary with y; they remain fixed when y varies. Lines corresponding to functions like $A_i(x)-B_i(y)$ do vary with y; they will move upwards or downwards, linear with respect to y. So, the vertical position of all lines is linear with respect to y. It follows that also the positions of the intersection points are linear with respect to y. Now, B'(y) is the vertical component of an intersection point. If B'(y) would correspond to the same intersection point for all values of y, then B'(y) would be linear. However, B'(y) may correspond to different intersection points for different values of y. If this is the case, then there are "transitions" of B'(y) from one intersection point to another. These transitions are the cause for nonlinearity of B'(y). A transition can occur for those values of y for which two different intersection points have the same vertical value. Those values of y are the members of our desired superset, which is a finite set containing all values of y where B' possibly is nonlinear.

To compute the superset we proceed as follows : for all pairs of two lines and for all pairs consisting of a line and a boundary, we compute the vertical component of the corresponding intersection point, as a linear function of y. For all pairs of these linear functions, we include the value of y for which they intersect in the superset.
This leads to the following Miranda implementation of the function `superset`:

```
superset int impl rules [(x0,a0'),(x1,a1')]
 = (sort . mkset . concat . map cross2 . pairs . mkset)
     intersection_points
   where
   y = (fst.hd.snd.hd) rules
   y' = (fst.hd.tl.snd.hd) rules
   lines Min j
     = mkset ((a0',a1',0):(concat (map (lines_from_rule Min j) rules)))
   lines_from_rule Min Lukasiewicz ([(x0,a0),(x1,a1)],[(y0,b0),(y1,b1)])
      = [(1,1,0),(1-a0+b0,1-a1+b0,b1-b0)]
   intersection_points
     = [(min[p,q],min[p,q]+v)|(p,q,v)<-ls] ++
       [(max[p,q],max[p,q]+v)|(p,q,v)<-ls] ++
       (concat.map cross. pairs) ls
       where ls = lines int impl
   cross2 ((a0,a1),(b0,b1))
     = [], greater_or_equal ((b0-a0)*(b1-a1)) 0
     = [y + (y'-y)*(b0-a0)/(b0-a0-b1+a1)], otherwise
   cross ((p1,q1,v1),(p2,q2,v2))
     = [] , if equal n 0
     = [((q1*p2-p1*q2)/n, (q1*p2-p1*q2+v1*(p2-q2)+v2*(q1-p1))/n)],
                                                                otherwise
       where
       n = q1-p1-q2+p2
   pairs [] = []
   pairs (x:xs) = [(x,y)|y<-xs] ++ pairs xs
```

5. The general case

In the previous sections we have considered the standard intersection and the Lukasiewicz implication. In this section we will show how the results of the previous sections can be adapted in order to deal with the other cases, which are mentioned in the introduction, as well.

Except the standard intersection, defined in equation (6), we will consider the bounded difference intersection, which is defined by

$$I(x,y) = \max(0,x+y-1) \quad (9)$$

Except the Lukasiewicz implication, defined in equation (7), we will consider the Kleene-Dienes implication, which is defined by

$$J(x,y) = \max(1-x,y) \quad (10)$$

the Early-Zadeh implication, which is defined by

$$J(x,y) = \max(1-x,\min(x,y)) \quad (11)$$

and the Willmott implication, which is defined by

$$J(x,y) = \min(\max(1-x,y),\max(x,1-x),\max(y,1-y)) \quad (12)$$

We have to adapt the functions *b* and *f*, which are local functions within the function *mult_app_reas3*, and the functions *lines* and *lines_from_rule*, which are local functions within the function *superset*. These adaptations are a straightforward coding of the intersections and implications, and are given below.

```
b Min impl y
```

```
   = sup (minplf [(x0,a0’),(x1,a1’)]
      (foldl1 minplf [f Min impl a (apply b y) | (a,b) <- rules] ))
b BD impl y
   = sup (maxplf [(x0,0),(x1,0)]
      (foldl1 minplf [f BD impl a (apply b y) | (a,b) <- rules] ))
f Min Lukasiewicz [(x0,a0),(x1,a1)] by
   = [(1,1,0),(x0,1-a0+by),(x1,1-a1+by)]
f Min Early_Zadeh [(x0,a0),(x1,a1)] by
   = maxplf [(x0,1-a0),(x1,1-a1)]
      (minplf [(x0,a0),(x1,a1)] [(x0,by),(x1,by)])
f Min Kleene_Dienes [(x0,a0),(x1,a1)] by
   = maxplf [(x0,1-a0),(x1,1-a1)] [(x0,by),(x1,by)]
f Min Willmott [(x0,a0),(x1,a1)] by
   = minplf (maxplf [(x0,1-a0),(x1,1-a1)] [(x0,by),(x1,by)] )
      (minplf (maxplf [(x0,a0),(x1,a1)] [(x0,1-a0),(x1,1-a1)] )
      [(x0,max[by,1-by]),(x1,max[by,1-by])])
f BD Lukasiewicz [(x0,a0),(x1,a1)] by
   = minplf [(x0,a0’),(x1,a1’)] [(x0,a0’-a0+by),(x1,a1’-a1+by)]
f BD Kleene_Dienes [(x0,a0),(x1,a1)] by
   = maxplf [(x0,a0’-a0),(x1,a1’-a1)] [(x0,a0’-1+by),(x1,a1’-1+by)]
f BD Early_Zadeh [(x0,a0),(x1,a1)] by
   = maxplf [(x0,a0’-a0),(x1,a1’-a1)]
      (minplf [(x0,a0),(x1,a1)] [(x0,a0’-1+by),(x1,a1’-1+by)])
f BD Willmott [(x0,a0),(x1,a1)] by
   = minplf (maxplf [(x0,a0’-a0),(x1,a1’-a1)] [(x0,a0’-1+by),(x1,by)] )
      (minplf (maxplf [(x0,a0’-1+a0),(x1,a1’-1+a1)]
      [(x0,a0’-a0),(x1,a1’-a1)])
      [(x0,max[a0’-1+by,a0’-by]),(x1,max[a1’-1+by,a1’-by])])

lines Min impl
   = mkset ((a0’,a1’,0):(concat (map (lines_from_rule Min impl) rules)))
lines BD impl
   = mkset ((0,0,0):(concat (map (lines_from_rule BD impl) rules)))

lines_from_rule Min Lukasiewicz ([(x0,a0),(x1,a1)],[(y0,b0),(y1,b1)])
   = [(1,1,0),(1-a0+b0,1-a1+b0,b1-b0)]
lines_from_rule Min Kleene_Dienes ([(x0,a0),(x1,a1)],[(y0,b0),(y1,b1)])
   = [(1-a0,1-a1,0),(b0,b0,b1-b0)]
lines_from_rule Min Early_Zadeh ([(x0,a0),(x1,a1)],[(y0,b0),(y1,b1)])
   = [(a0,a1,0),(1-a0,1-a1,0),(b0,b0,b1-b0)]
lines_from_rule Min Willmott ([(x0,a0),(x1,a1)],[(y0,b0),(y1,b1)])
   = [(a0,a1,0),(1-a0,1-a1,0),(b0,b0,b1-b0),(1-b0,1-b0,b0-b1)]
lines_from_rule BD Lukasiewicz ([(x0,a0),(x1,a1)],[(y0,b0),(y1,b1)])
   = [(a0’,a1’,0),(a0’-a0+b0,a1’-a1+b0,b1-b0)]
lines_from_rule BD Kleene_Dienes ([(x0,a0),(x1,a1)],[(y0,b0),(y1,b1)])
   = [(a0’-a0,a1’-a1,0),(a0’-1+b0,a1’-1+b0,b1-b0)]
lines_from_rule BD Early_Zadeh ([(x0,a0),(x1,a1)],[(y0,b0),(y1,b1)])
   = [(a0’-1+a0,a1’-1+a1,0),(a0’-a0,a1’-a1,0),(a0’-1+b0,a1’-1+b0,b1-b0)]
lines_from_rule BD Willmott ([(x0,a0),(x1,a1)],[(y0,b0),(y1,b1)])
   = [(a0’-1+a0,a1’-1+a1,0),(a0’-a0,a1’-a1,0),
      (a0’-1+b0,a1’-1+b0,b1-b0),(a0’-b0,a1’-b0,b0-b1)]
```

6. Example

Mizumoto [6] has calculated inference results with multiconditional approximate reasoning. He used both the standard and the bounded difference intersections, and 7 implications, among which the Lukasiewicz, the Kleene-Dienes and the Early-Zadeh implications. We show here how, for these intersections and these three implications, we can reproduce his results.
He used three rules:

If X=a1 then Y=b1
If X=a2 then Y=b2
If X=a3 then Y=b3

with

```
a1 = [(0,0),(1,0),(3,1),(5,0),(10,0)]
b1 = [(0,0),(1,0),(3,1),(5,0),(10,0)]
a2 = [(0,0),(3,0),(5,1),(7,0),(10,0)]
b2 = [(0,0),(3,0),(5,1),(7,0),(10,0)]
a3 = [(0,0),(5,0),(7,1),(9,0),(10,0)]
b3 = [(0,0),(5,0),(7,1),(9,0),(10,0)]
```

With the fact

```
a′ = [(0,0),(2,0),(4,1),(6,0),(10,0)]
```

the six expressions

```
mult_app_reas Min Lukasiewicz [(a1,b1),(a2,b2),(a3,b3)] a′
mult_app_reas Min Kleene_Dienes [(a1,b1),(a2,b2),(a3,b3)] a′
mult_app_reas Min Early_Zadeh [(a1,b1),(a2,b2),(a3,b3)] a′
mult_app_reas BD Lukasiewicz [(a1,b1),(a2,b2),(a3,b3)] a′
mult_app_reas BD Kleene_Dienes [(a1,b1),(a2,b2),(a3,b3)] a′
mult_app_reas BD Early_Zadeh [(a1,b1),(a2,b2),(a3,b3)] a′
```

evaluate to, respectively, the plfs

[(0,0.5),(1,0.5),(2.0,0.75),(3,0.75),(4.0,1.0),(5,0.75),(6.0,0.75),(7,0.5),(10,0.5)]
[(0,0.5),(2.0,0.5),(2.5,0.75),(3.5,0.75),(4.0,0.5),(4.5,0.75),(5.5,0.75),(6.0,0.5),(10,0.5)]
[(0,0.5),(2.0,0.5),(2.5,0.75),(3.5,0.75),(4.0,0.5),(4.5,0.75),(5.5,0.75),(6.0,0.5),(10,0.5)]
[(0,0.5),(3,0.5),(4.0,1.0),(5,0.5),(10,0.5)]
[(0,0.5),(10,0.5)]
[(0,0.5),(10,0.5)]

which is equal to the result obtained by Mizumoto.

References

[1] P.M. van den Broek, Fuzzy Reasoning with Continuous Piecewise Linear Membership Functions, in: NAFIPS’97, Proceedings of the 1997 Annual Meeting of the North American Fuzzy Information Processing Society (eds. C.Isik and V.Cross), Syracuse, New York, U.S.A. (1997) pp. 371-376

[2] G.J. Klir and B. Yuan, Fuzzy sets and fuzzy logic, theory and applications (Prentice-Hall) 1995

[3] D. Dubois, R. Martin-Clouaire and H. Prade, Practical Computing in Fuzzy Logic, in: Fuzzy Computing (eds. M.M. Gupta and T. Yamakawa), Elsevier Science Publishers B.V. (North-Holland) 1988, 11-34

[4] B. De Baets and E.E. Kerre, The Generalised Modus Ponens and the Triangular Fuzzy Data Model, Fuzzy Sets and Systems 59 (1993) 305-317

[5] D. Turner, Miranda: a non-strict functional language with polymorphic types, in: Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science Vol. 201, ed. J.-P. Jouannaud, (Springer-Verlag) 1-16, 1985

[6] M. Mizumoto, Multifold Fuzzy Reasoning as Interpolative Reasoning, in: Fuzzy Sets, Neural Networks, and Soft Computing (eds. R.R. Yager and L.A. Zadeh), Van Nostrand Reinhold (New York) 1994, 188-193