A Preliminary Study of Probabilistic Argumentation

Thomas C. Henderson, Amar Mitiche, Robert Simmons and Xiuyi Fan
University of Utah

UUCS-17-001

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

14 February 2017

Abstract

We investigate methods to define a probabilistic logic and their application to probabilistic argumentation[1]. We begin with a discussion of augmenting propositional calculus with probabilities. We start with a set of sentences, S, each with a known probability, and then the problem is to determine the probability of a query sentence that is a disjunction of literals appearing in S. First, we examine Nilsson's [15] solution based on the semantic models of the sentences; we develop two different approaches to solving the problem as posed: (1) using a linear solver, and (2) geometrically finding the intersection of a line with the probability convex hull (see below). However, Nilsson's approach only provides lower and upper bounds on the solution. We then propose a new approach which finds probabilities for the atoms found in the sentences, and then uses these probabilities to compute the probability of the query sentence. Finally, we describe how this probability representation method can form the basis for a probabilistic argumentation system.

[1]This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-17-1-0077.

1 Introduction

Given the uncertainty associated with statements about the world, or with sensor data, or hypothesis formation in general, it is important to be able to represent such uncertainty appropriately for each source of information, and to be able to combine those disparate types of uncertainty in a consistent manner. For example, sensor data may have associated Gaussian or other types of noise models, while computational processes may have algorithmic uncertainty due to truncation errors, roundoff errors, etc. Logical sentences have commonly been used to represent knowledge, and it is useful to associate an uncertainty with such sentences.

Here we address the problem of finding a suitable representation for uncertainty associated with logical sentences. Although several approaches have been proposed in the past (see [1, 7, 12, 14, 16, 18]), they generally have some significant drawbacks. Usually, these have to do with the computational complexity of the semantics of the sentences (i.e., finding the set of consistent truth assignments is exponential in the number of sentences). There have been some attempts to address these issues, but even those generally have algorithmic issues. For example, consider the probabilistic theorem proving (PTP) method of Gogate and Domingos [11]. They solve PTP:

> by reducing it to lifted weight model counting. *Model counting* is the problem of determining the number of worlds that satisfy a KB (knowledge base of sentences). *Weighted* model counting can be defined as follows. Assign a weight to each literal, and let the weight of a world be the product of the weights of the literals that are true in it. Then weighted model counting is the problem of determining the sum of the weights of the worlds that satisfy a KB.

This approach is based on Nilsson's early work on probabilistic logic [15] (for more details on Nilsson's approach, see Section 2). In particular, they use Nilsson's framework in which the probability of a query formula is equal to the sum of the probabilities of the worlds that satisfy it, and give a formula:

$$P(T \mid K) = \frac{\sum_x 1_T(x) \prod_i \Phi_i(x)}{Z(K)},$$

where $P(T \mid K)$ is the probability of the query, T, given the knowledge base of sentences, $1_T(x)$ is the characteristic function for when the query is true in a possible world, and $\Phi_i(x)$ results from using a set of potential functions (see [11] for details on this) to estimate the probability of the possible world (i.e., $P(x)$), and $Z(K)$ is a normalizing factor found

by a weighted model counting method. This method also requires exploring the semantic models for the $KB \cup \{T\}$, or using Monte Carlo to sample that space.

We describe here an alternative approach which avoids the computation of the semantic models, and provides a solution for $Prob(T \mid K)$. Basically, this is done by exploiting the probability of a disjunctive clause, and developing a set of equations from the sentences and their probabilities, and then solving those equations (where the number of equations is equal to the number of sentences). We show that this can be employed in an argumentation framework (for more on argumentation, see [2, 3, 4, 5, 6, 8, 9, 10, 17, 19]).

2 Nilsson's Proposed Method for Probabilistic Logic

Our approach to probabilistic logic starts with an analysis of Nilsson's method [15]. Given a set of n sentences, $S = \{S_1, S_2, \ldots, S_n\}$, in the propositional calculus, where $\{S_1, \ldots, S_{n-1}\}$ is the KB and S_n is the query, he first finds the set of models of the sentences (i.e., the set of truth value assignments to the sentences that are consistent). Figure 1 shows the general semantic tree [13] for a set of sentences (Matlab function BR_find_worlds in Appendix A computes the consistent assignments of truth values to a set of sentences). A simple ex-

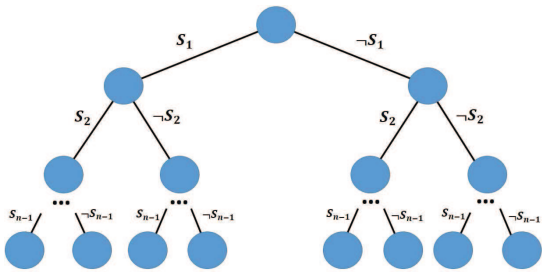

Figure 1: The General Semantic Tree for a Set of Sentences.

ample given by Nilsson is shown in Figure 2 which is the semantic tree for the sentences $S = \{P, \neg P \vee Q, Q\}$; we will call this the **Modus Ponens Problem**. That is, show that $KB = \{S_1, S_2\} \models S_3$. This set of models is formed (as columns) into a matrix, V:

$$V = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix},$$

where V is determined by expanding the semantic tree of all possible combinations of truth assignments to the sentences; note that the determination of the models of a set of sentences

has computational cost exponential in the number of sentences or $O(2^{|S|})$, where $\mid S \mid = n$. Each row, i, of V gives the truth assignment of sentence S_i in the models.

Nilsson exploits the relation between the probability of the sentences, Π, and the probabilities of the models of the sentences, P, by means of the truth value models themselves, V. Π is an $n \times 1$ vector, P is an $m \times 1$ vector, and V is an $n \times m$ array such that:

$$\Pi = VP.$$

Note that each column of V is a distinct possible world (model) for the sentences.

In order to determine $\Pi(n)$, the probability of the last sentence. Nilsson proposes to solve:

$$\Pi' = V'P,$$

where Π' is the first $n-1$ elements of Π, and V' is the first $n-1$ rows of V. In addition, to impose the constraint that $\sum_{i=1}^{m} P(i) = 1$, a new first element, 1, is added to Π', and an all 1 first row is added to V' (this specifies that *True* is true in all models). We can now solve for P. However, this system is generally underdetermined and can be severely so for large n. Nilsson provides some ways to overcome this, but the computational complexity of the approach.

2.1 Linear Solvers for $\Pi' = V'P$

A method is now given which can find the lower and upper bounds for the probability of the query sentence using standard linear solvers (in this case *lsqlin* in Matlab). Given the matrix V, and a set of probabilities, Π, for the n sentences (i.e., $n-1$ from the knowledge base and the n^{th} sentence which is the query, do the following:

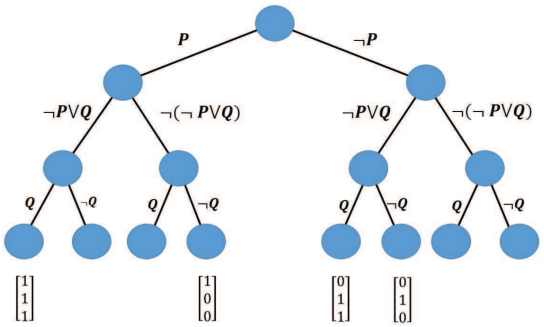

Figure 2: The Semantic Tree for $S = \{P, \neg P \vee Q, Q\}$.

1. Add a row of 1's as the first row of V; call this new matrix V_{aug}.

2. Add a 1 as the first element of Π; call this vector Π_{aug}. This means that $\Pi_{aug}(k)$ is $\Pi(k-1)$ for $k = 2 \ldots n+1$.

3. Set $\Pi_{aug}(n+1)$ to 0.

4. For $y = 0$ to 1 in steps of 0.01, set $\Pi_{aug}(n+1) \leftarrow y$, and solve:

$$\Pi_{aug} = V_{aug} P.$$

The first two steps ensure that the probabilities of the possible worlds sum to 1. The last step produces a solution for each guess for the value of the probability of the query sentence. Here we use *lsqlin* which returns a zero probability for one of the world models when the guess value, y, is not a valid solution. For example, in the **Modus Ponens Problem**, we find $[0.4, 0.69]$ as the bounds for the probability of the query sentence, $\Pi(S_{n+1})$. The upper bound should be 0.7, and the difference is due to the step size for y. The Matlab function for this is given in Appendix A as BR_Nilsson_method_all. Alternatively, constraints can also be added to the Matlab *lsqlin* call in order to find the bounds; Matlab function BR_Nilsson_calculate_prob_bounds performs (see Appendix A) this computation and returns $[0.4, 0.7]$ as the upper and lower bounds.

Summary: The basic method proposed by Nilsson does not provide a unique solution for the probability of the query sentence since the system is underdetermined. Moreover, the computation of the semantic tree is exponential in the number of sentences.

2.2 A Geometric Approach to Find Lower and Upper Bounds

Nilsson also points out that consistent probability assignments lie within the convex hull of the semantic vectors (columns of V). Figure 3 shows the convex hull for the set of vectors in the **Modus Ponens Problem**. The axes are the probabilities of the sentences; the figure shows that the bounds on the probability of the query (i.e., $\Pi(S_n)$) can be found by intersecting the line parallel to the probability of the $Prob(Q)$ axis (and in general, the axis for the last sentence or query) that goes through the lower dimensional point in the space of the probabilities of the sentences whose probabilities are known, and the convex hull of the V column points. The figure indicates that these bounds are 0.4 for the lower probability and 0.7 for the upper bound. The results found here can be produced using the Matlab function BR_problem_interval_inhull (see Appendix A).

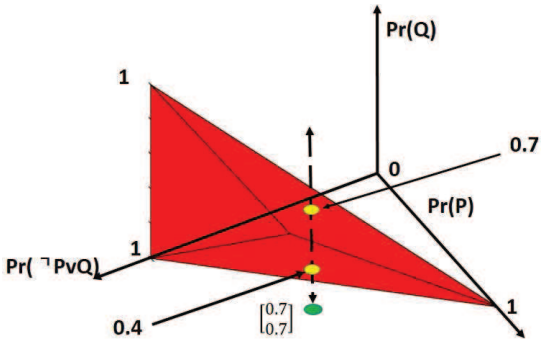

Figure 3: The Convex Hull for the Modus Ponens Semantic Tree.

3 A New Method for Probabilistic Logic

We propose an alternative approach that overcomes the complexity issue. First, we assume that the sentences are given in conjunctive normal form. This means that each sentence is a disjunct of literals (an atom or its negation). Our second assumption is that $Pr(P \wedge Q) = Pr(P)Pr(Q)$; note that if this assumption is violated, our methods also allow the bounds on the probability to be determined. Next, we determine the set of logical atoms (i.e., variables) in S; let $A = \{A_1, A_2, \ldots, A_k\}$ be this set. In this case the probability of a sentence can be computed from the probability of the literals as follows:

$$Prob(L_1 \vee L_2 \vee \ldots \vee L_p) =$$

$$Prob(L_1) + Prob(L_2 \vee \ldots \vee L_p)$$
$$-Prob(L_1)Prob(L_2 \vee \ldots \vee L_p),$$

where the probabilities of clauses on the right hand side are computed recursively.

Assuming that the logical (random) variables are independent, each sentence gives rise to a (usually) nonlinear equation defined by the recursive probability of the disjunctive clause as defined above. The resulting set of equations can be solved using standard nonlinear solvers (e.g., *fsolve* in Matlab), and a set of consistent values for the probabilities of the atoms determined.

Consider Nilsson's example: $S_1 = P$, $S_2 = \neg P \vee Q$, and $S_3 = Q$ with $\Pi(S_1) = \Pi(S_2) = 0.7$. The resulting equations for *fsolve* are:

$$F(1) = -0.7 + x_1$$

$$F(2) = -0.7 + (1 - x_1) + x_2 - (1 - x_1)x_2$$

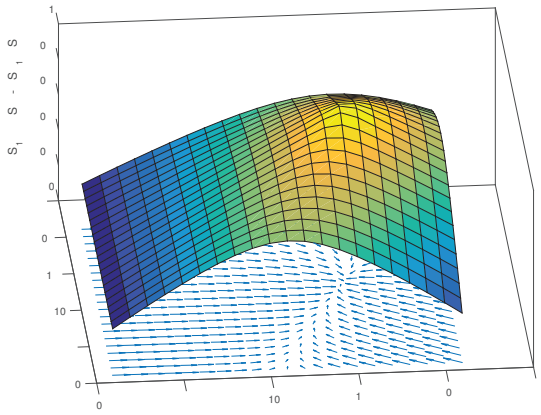

Figure 4: Error in Known Sentence Probabilities and Sentence Probabilities based on Atom Probabilities.

where $x_1 = Prob(P)$ and $x_2 = Prob(Q)$. These are derived as follows:

$$\Pi(S_1) = Prob(P)$$

$$\Pi(S_2) = Prob(\neg P \vee Q)$$

$$\Rightarrow 0.7 = Prob(P)$$

$$\Rightarrow 0.7 = (1 - Prob(P)) + Prob(Q) - (1 - Prob(P))Prob(Q)$$

$$\Rightarrow 0 = x_1 - 0.7$$

$$\Rightarrow 0 = (1 - x_1) + x_2 - (1 - x_1)x_2 - 0.7$$

$$\Rightarrow F(1) = x_1 - 0.7$$

$$\Rightarrow F(2) = (1 - x_1) + x_2 - (1 - x_1)x_2 - 0.7$$

A solution found by the nonlinear solver (*fsolve* in Matlab) is $x_1 = 0.7$ and $x_2 = 0.571$. The equations for F can be generated automatically from the sentences; this is done by function BR_KB2nonlinear_function (see Appendix A). Figure 4 shows the distance function between the known sentence probabilities and the atom determined sentence probabilities over all possible atom probability combinations. The gradient vectors are shown in the x-y plane, and as can be seen, they all lead to the optimal assignment of atom probabilities. (Note that although the surface is shown as a max, *fsolve* finds the minimum of 1 minus this distance.)

Consider a second example with 16 sentences and 9 atoms:

```
 1. A
 2. ~A v C v D
 3. ~B v C v D
 4. ~A v E v F
 5. ~B v E v F
 6. ~C v G v H
 7. ~D v G v H
 8. ~E v G v H
 9. ~F v G v H
10. ~A v G v H
11. ~C v ~G
12. ~C v ~H
13. ~E v ~G
14. ~E v ~H
15. ~D v I
16. ~F v I
17. I          -- query
```

The solution to this has $Prob(A) = 1$, $Prob(B) \in \{0, 1\}$, $Prob(C) = 0$, $Prob(D) = 1$, $Prob(E) = 0$, $Prob(F) = 1$, $(Prob(G), Prob(H)) \in \{(0, 1), (1, 0), 1, 1)\}$, $Prob(I) = 1$. Assigning sentence probabilities all 1 (i.e., $Prob(S_i) = 1, i = 1 \ldots 16$), the nonlinear solver finds the following 3 solutions from starting at initial estimate all 0's, all 0.5's and all 1's:

```
1 0 -0 1 0 1 0.9998 0.9998 1
1 0.0644 -0 1 -0 1 0.9998 0.9998 1
1 1 -0 1 -0 1 0.9997 0.9997 1
```

Of course, one problem with the nonlinear solver approach is that it may not find a solution, even when one or more exist. For this reason and the fact that high-dimensional spaces (in the number of atoms or sentences) may also pose problems for such solvers, we also propose to use Monte Carlo techniques to estimate the probability of the query sentence. A simple Monte Carlo approach, called *Monte Carlo - Modified Nilsson* (MCMN), can now be used to estimate the value of $\Pi(n)$ (or what is the same thing, $Pr(S_n)$):

```
Algorithm MCMN

  e <-- max desired error
```

```
N <-- max number of iterations
err <-- infinity
P* <-- 0
i <-- 0
while (err>e)&(i<N)
  P_a ~ U([0,1]^k)
  Pi_a <-- sentence probs from atoms
  if |Pi - Pi_a|<err
    P* <-- P_a
    err <-- |Pi - Pi_a|
  end
end
```

This method avoids the calculation of the semantic tree and provides a set of probabilities, P*, for the atoms where these probabilities produce the known sentence probabilities. P* can now be used to compute the probability of the query since it is a disjunction in the literals of the atoms.

Algorithm MCMN can also be posed as a Markov Chain MC method by using the error (between the actual sentence probabilities and the predicted sentence probabilities based on the atom probabilities) to compute a fitness function, using the ratio of the current and previous sample in the sample retention test. For example, we use the Euclidean distance between sentence probabilities derived from sample atom probabilities to known sentence probabilities. Then the statistic computed over the set of samples is just the atom probability sample that maximizes the correctness of the sentence probabilities (i.e., minimizes the Euclidean distance). Applying this method to the **Modus Ponens Problem** with 500 samples results in an estimate of $Prob(P) = 0.6972$ and $Prob(Q) = 0.5821$. For the second example above, $[0.8472, 0.2337, 0.1069, 0.8053, 0.2837, 0.7182, 0.4427, 0.6567, 0.8574]$ is the MCMN result for the probability of the atoms A through I.

In some cases, the probability of a sentence may be a range of values; for example, consider the **Modus Ponens Problem** with $Pr(P) = 0$ and the $Pr(\neg P \vee Q) = 1$. Then $Pr(Q)$ can range from 0 to 1. Running the Markov Chain Monte Carlo method on this problem results in a set of samples. If a subset of those samples is selected such that the $Pr(P) \leq 0.1$, then we find that $0 \leq Pr(Q) \leq 1$ (Figure 5 shows an example; the max of these samples is 0.9996, the min is 0.0069, and the mean is 0.514). Thus, the Monte Carlo approach also allows for the determination of a range of the probability of a sentence.

Summary: This method avoids the exponential cost of the semantic tree expansion, and finds the unique solution to the set of sentence probability equations. Matlab functions

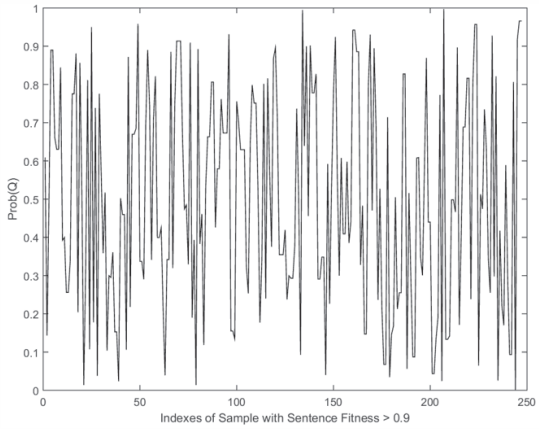

Figure 5: The Set of Sample Values for $Prob(Q)$ for the Modus Ponens Problem when $Prob(P) = 0$ and $Prob(\neg P \vee Q) = 1$.

BR_KB2nonlinear_function and BR_MC_AT compute the nonlinear solver solution and the Monte Carlo solution, respectively. The success of the nonlinear solver in finding a solution is dependent on a good initial starting point, but the Monte Carlo method is guaranteed to find a solution if one exists if enough samples are taken.

4 Probabilistic Argumentation

So far, we have provided examples of probabilistic logic for a propositional knowledge base; this assumes that the KB is consistent. An argumentation knowledge base may contain a set of inconsistent sentences. Given a query, a minimal set of sentences in the KB are sought such that they are consistent and they entail the query, and they have no counter-arguments to their component sentences. We have implemented a set of Matlab functions that provide the probabilistic analysis for a query in the argumentation setting. The Matlab function BR_PIKB_query given in Appendix A demonstrates this functionality.

5 Conclusions

We have examined Nilsson's probabilistic logic and discussed its shortcomings; moreover, we have provided new ways to solve this problem and obtain the unique solution (assuming

the atoms are independent random variables). We believe that this new approach offers an effective and efficient approach to providing a probabilistic logic for argumentation. We are currently extending this approach to First Order Logic, as well as testing it on a set of applications.

A Matlab Functions

A.1 BR_Nilsson_method_all(KB,thm,probs)

```
function p_range = BR_Nilsson_method_all(KB,thm,probs)
% BR_Nilsson_method_all - determine lower and upper bounds for the
%    probability of the thm in terms of the probabilities of the KB
% On input:
%     KB (1x(n-1)KB struct vector): each element has the following
%     field:
%       (i).clauses (1xki vector): represents a disjunctive clause;
%       each element is a +/- a for an integer representing a logical
%       atom
%     thm (KB struct vector): only has one element
%       (i).clauses (1xp vector): represents a disjunctive clause;
%       each element is a +/- a for an integer representing a logical
%       atom
%     probs (1x(n-1) vector): probabilities of clauses in KB
% On output:
%     p_range (1x2 vector): lower and upper bounds for probability of
%     query
% Call:
%     KB(1).clauses = [1];
%     KB(2).clauses = [-1,2];
%     thm(1).clauses = [2];
%     probs - [0.7,0.7];
%     pr = BR_Nilsson_method_all(KB,thm,probs);
% Author:
%     T. Henderson
%     UU
%     Spring 2017
%

p_range = [];

KB_all = KB;
KB_all(end+1).clauses = thm(1).clauses;

V = BR_find_worlds(KB_all);
```

```
[m,n] = size(V);
Vp = [ones(1,n);V];

PI = reshape(probs,length(probs),1);
PIp = [1;PI;0];

for e = 1:101
    v = (e-1)*0.01;
    PIp(end) = v;
    P = lsqlin(Vp,PIp,[],[],[],[]);
    if min(P)>=0
        p_range = [p_range,v];
    end
end
p_range = [min(p_range),max(p_range)];
```

A.2 BR_find_worlds

```
function worlds = BR_find_worlds(KB)
% BR_find_worlds - find possible consistent truth assignments to KB
% clauses
% On input:
%     KB (KB struct vector)
%       (i).clauses (1xm vector): disjunct clause
% On output:
%     worlds (mxn array): m is KB length; n is number of consistent
%     worlds;
% Call:
%     KB(1).clauses = [1];  -- Nilsson example
%     KB(2).clauses = [-1,2];
%     KB(3).clauses = [2];
%     V = BR_find_worlds(KB);
% Author:
%     Tom Henderson
%     UU
%     Spring 2017
%

worlds = [];
len_KB = length(KB);
if len_KB==0
    return
end

V = [];
for s = 1:2^len_KB
    s_bits = BR_int2bits(s-1,len_KB);
    index = 0;
    clear KB_s
    for n = 1:len_KB
        clause = KB(n).clauses;
        if s_bits(n)==0
            num_disjuncts = length(clause);
            for d = 1:num_disjuncts
                index = index + 1;
                KB_s(index).clauses = -clause(d);
```

```
            end
        else
            index = index + 1;
            KB_s(index).clauses = clause;
        end
    end
    vars = BR_vars(KB_s,[]);
    alpha = max(vars) + 1;
    Sr = BR_RTP(KB_s,alpha,[vars,alpha]);
    if ~isempty(Sr)
        V = [s_bits',V];
    end
end
worlds = V;
tch = 0;
```

A.3 BR_int2bits

```
function v_bits = BR_int2bits(v,n)
% BR_int2bits - convert an integer to an n-bit binary number
% On input:
%     v (int): integer value
%     n (int): number of bits
% On output:
%     v_bits (1xn vector): binary representation of v
% Call:
%     v = BR_int2bits(5,3);
% Author:
%     T. Henderson
%     UU
%     Fall 2014
%

v_bits = zeros(1,n);

for b = 1:n
    v_bits(b) = rem(v,2);
    v = floor(v/2);
end

v_bits = v_bits(end:-1:1);
```

A.4 BR_RTP

```
function Sip = BR_RTP(sentences,thm,vars)
% BR_RTP - resolution theorem prover
% On input:
%     sentences (CNF data structure): array of conjunctive clauses
%       (i).clauses
%           each clause is a list of integers (- for negated literal)
%     thm (1xk vector): disjunctive clause to be tested
%     vars (1xn vector): list of variables (positive integers)
% On output:
%     Sip (CNF data structure): results of resolution
%        []: proved sentence |- thm
%        not []: thm does not follow from sentences
% Method:
%  Let S1 = S.
% Let i = 1.
% LOOP until i = n + 1.
% Discard members of Si in which a literal and its
%    complement appear, to obtain Sip.
% Let Ti be the set of parent clauses in Sip in which Pi or
%    -Pi appears.
% Let Ui be the set of resolvent clauses obtained by
%     resolving (over Pi ) every pair of clauses C U {Pi} and
%     D U {-Pi} in Ti.
% Set Si+1 equal to (Sip\Ti) U Ui . (Eliminate Pi ).
% Let i be increased by 1.
% ENDLOOP.
% Output Sn+1.
% Call:  (example from Russell & Norvig, p. 252)
%     DP(1).clauses = [-1,2,3,4];
%     DP(2).clauses = [-2];
%     DP(3).clauses = [-3];
%     DP(4).clauses = [1];
%     thm = [4];
%     vars = [1,2,3,4];
%     Sr = BR_RTP(DP,thm,vars);
% Author:
%     T. Henderson
%     UU
```

```
%     Summer 2014; modified Summer 2016
%

num_sentences = length(sentences);
len_thm = length(thm);
not_thm = -thm;
for ind = 1:len_thm
    num_sentences = num_sentences + 1;
    sentences(num_sentences).clauses = [not_thm(ind)];
end

n = length(vars);
Sipn = sentences;
for i = 1:n
    Sip = BR_elim_L_nL(Sipn);
    Ti = BR_parent_clauses(Sip,vars(i));
    Ui = BR_resolvent_clauses(Ti,vars(i));
    if BR_empty_clause(Ui)
        Sip = [];
        return
    end
    Sipn = BR_update_S(Sip,Ti,Ui);
end
Sip = Sipn;
```

A.5 BR_elim_L_nL

```
function Sip = BR_elim_L_nL(S)
% BR_elim_L_nL - remove a disjunction with both a literal
%   and its negation
% On input:
%     S (CNF data structure): array of conjunctive clauses
%       (i).clauses
%           each clause is a list of integers (- for negated literal)
% On output:
%     Sip (CNF data structure): results of disjunction elimination
% Call:
%     Se = BR_elim_L_nL(Sip);
% Author:
%     T. Henderson
%     UU
%     Summer 2014; modified Summer 2016
%

len_S = length(S);
num_Sip = 0;
for s = 1:len_S
    clauses = S(s).clauses;
    if ~isempty(clauses)
        len_c = length(clauses);
        found = 0;
        for c = 1:len_c
            if ~isempty(find(-clauses(c)==clauses))
                found = 1;
            end
        end
        if found==0
            num_Sip = num_Sip + 1;
            Sip(num_Sip).clauses = clauses;
        end
    end
end
```

A.6 BR_parent_clauses

```
function Ti = BR_parent_clauses(Sip,P)
% BR_parent_clauses - find all clauses with literal P or its negation
% On input:
%     Sip (CNF data structure): array of conjunctive clauses
%       (i).clauses
%           each clause is a list of integers (- for negated literal)
%     P (int): positive literal
% On output:
%     Ti (CNF data structure): subset of Sip with +/- P in them
% Call:
%     Ti = BR_parent_clauses(Sip,2);
% Author:
%     T. Henderson
%     UU
%     Summer 2014; modified summer 2016
%

Ti = [];
if isempty(Sip)
    return
end

len_Sip = length(Sip);
len_Ti = 0;
for s = 1:len_Sip
    clauses = Sip(s).clauses;
    if ~isempty(find(clauses==P|clauses==(-P)))
        len_Ti = len_Ti + 1;
        Ti(len_Ti).clauses = clauses;
    end
end
```

A.7 BR_resolvent_clauses

```
function Ui = BR_resolvent_clauses(Ti,P)
% BR_resolvent_clauses - results of resolving clauses with P
%   or its negation
% On input:
%     Ti (CNF data structure): array of conjunctive clauses
%       (i).clauses
%           each clause is a list of integers (- for negated literal)
%     P (int): literal
% On output:
%     Ui (CNF data structure): results of disjunction elimination
% Call:
%     Ui = BR_resolvent_clauses(Ti,4);
% Author:
%     T. Henderson
%     UU
%     Summer 2014; modified Summer 2016
%

len_Ti = length(Ti);
sorted_clauses = zeros(len_Ti,2);

for ind = 1:len_Ti
    pind = find(Ti(ind).clauses==P);
    nind = find(Ti(ind).clauses==(-P));
    if ~isempty(pind)
        sorted_clauses(ind,1) = 1;
        sorted_clauses(ind,2) = pind;
    else
        sorted_clauses(ind,1) = -1;
        sorted_clauses(ind,2) = nind;
    end
end
indexes_pos = find(sorted_clauses(:,1)>0);
num_pos = length(indexes_pos);
indexes_neg = find(sorted_clauses(:,1)<0);
num_neg = length(indexes_neg);
if num_pos*num_neg==0
    Ui = [];
```

```
    return
end

len_Ui = 0;
for p = 1:num_pos
    p_clause = Ti(indexes_pos(p)).clauses;
    p_index = sorted_clauses(indexes_pos(p),2);
    for n = 1:num_neg
        n_clause = Ti(indexes_neg(n)).clauses;
        len_Ui = len_Ui + 1;
        n_index = sorted_clauses(indexes_neg(n),2);
        Ui(len_Ui).clauses = ...
            [p_clause(1:p_index-1),p_clause(p_index+1:end), ...
             n_clause(1:n_index-1),n_clause(n_index+1:end)];
    end
end
```

A.8 BR_update_S

```
function Sn = BR_update_S(Sip,Ti,Ui)
% BR_update_S - remove Ti from S and add Ui
% On input:
%     Sip (CNF data structure): current conjunctive clauses
%       (i).clauses
%           each clause is a list of integers (- for negated literal)
%     Ti (CNF data structure): parent clauses
%     Ui (CNF data structure): resolvent clauses
% On output:
%     Sn (CNF data structure): results of set operations
% Call:
%     Sn = BR_update_S(Sip,Ti,Ui);
% Author:
%     T. Henderson
%     UU
%     Summer 2014; modified Summer 2016
%

Sn = [];

len_Sip = length(Sip);
len_Ti = length(Ti);
len_Ui = length(Ui);

len_Sn = 0;

found = zeros(len_Sip,1);
for ind_Ti = 1:len_Ti
    clause_Ti = Ti(ind_Ti).clauses;
    s_Ti = sort(clause_Ti);
    for ind_Sip = 1:len_Sip
        clause_Sip = Sip(ind_Sip).clauses;
        s_Sip = sort(clause_Sip);
        if (length(s_Ti)==length(s_Sip))&prod(s_Ti==s_Sip)==1
            found(ind_Sip) = 1;
        end
    end
end
```

```
indexes = find(found==0);
if ~isempty(indexes)
    len_Sn = length(indexes);
    for n = 1:len_Sn
        Sn(n).clauses = Sip(indexes(n)).clauses;
    end
end

if isempty(Ui)
    Sn = Sip;
    return
end
for ind_Ui = 1:len_Ui
    clause_Ui = Ui(ind_Ui).clauses;
    len_Sn = len_Sn + 1;
    Sn(len_Sn).clauses = clause_Ui;
end
```

A.9 BR_calculate_prob_bounds

```
function [ prob_min, prob_max ] = Nilsson_calculate_prob_bounds(...
    sentences, thm, pi_prime, vars )
% NILSSON_CALCULATE_PROBS Calculate the probability of thm given
% sentences
% with the probabilities pi_prime.
% Input:
%   sentences (CNF data structure): Sentences with known probabilities
%   thm (CNF data structure): Disjunctive clause whose probability we
%   want
%       to solve for
%   pi_prime (nx1 double vector): Probabilities for sentences. To
%   match the
%       Nilsson's notation, the first element of pi_prime should be 1,
%       the
%       second should be the probability of the first element of
%       sentences
%       etc.
%   vars (1xm int vector): Enumeration of variables
% Output:
%   prob_min (double): lower bound
%   prob_max (double): higher bound
% Sample call:
%   sentences(1).clauses = [1];
%   sentences(2).clauses = [-1, 2];
%   thm(1).clauses = [2];
%   pi_prime = [1; .5; 1];
%   vars = [1, 2];
%   [pmin, pmax] = Nilsson_calculate_prob_bounds ( sentences, thm,...
%      pi_prime, vars);
%
% Author:
%   Robert Simmons
%   UU
%   Winter 2016

iters = 10;
fp_tol = .000001;
```

```
V = Nilsson_tree([sentences, thm], vars);

V_prime = double([ones(1, size(V, 1)); V(:,1: size(sentences, 2))']);
%size(V,2) - 1)'];

%P = pinv(V_prime)*pi_prime;
%?
P = lsqlin(V_prime, pi_prime, [], [], [], [], zeros(1,
length(V_prime)),...
    []);
%P = lsqnonneg(V_prime, pi_prime);

%opt = optimoptions('lsqlin', 'Algorithm', 'interior-point');
opt = optimoptions('lsqlin');

VP_L = length(V_prime);

P_start = V'*P;
P_start = P_start(size(sentences, 2) + 1);

P_min = P;
lb_min = 0;
lb_range = P_start;
prob_min = -1;

P_exists = abs(V_prime*P_min - pi_prime) < fp_tol;

if P_exists > 0
    prob_min = (V'*P_min);
    prob_min = prob_min(size(sentences, 2) + 1);
%    lb_range = lb_range/2;
else
    prob_max = -1;
    return;
end

for lb_i = 1:iters
    target = [ones(size(sentences, 2), 1); lb_min + lb_range/2];
    P_min = lsqlin(V_prime, pi_prime, V', target , [], [], zeros(1,...
        VP_L), [], P_min, opt);
```

```
    P_exists = abs(V_prime*P_min - pi_prime) < fp_tol;

    if P_exists
        prob_min = (V'*P_min);
        prob_min = prob_min(size(sentences, 2) + 1);
        lb_range = lb_range/2;
    else
        lb_min = lb_min + lb_range/2;
        lb_range = lb_range/2;
    end
end

P_max = P;
ub_max = 1;
ub_range = 1 - P_start;
prob_max = -1;

P_exists = abs(V_prime*P_max - pi_prime) < fp_tol;

if P_exists
    prob_max = V'*P_max;
    prob_max = prob_max(size(sentences, 2) + 1);
%    ub_range = ub_range/2;
end

for ub_i = 1:iters
    target = [zeros(size(sentences, 2), 1); ub_max - ub_range/2];

    P_max = lsqlin(V_prime, pi_prime, -V', -target, [], [],
    zeros(1,...
        VP_L), [], P_max, opt);

    P_exists = abs(V_prime*P_max - pi_prime) < fp_tol;

    if P_exists
        prob_max = V'*P_max;
        prob_max = prob_max(size(sentences, 2) + 1);
        ub_range = ub_range/2;
    else
        ub_max = ub_max - ub_range/2;
        ub_range = ub_range/2;
```

```
    end
end
end
```

A.10 BR_problem_interval_inhull

```
function prob_interval = BR_prob_interval_inhull(pts,probs)
% BR_prob_interval_inhull - find the query probability bounds using
% the
%    convex hull of the semantic tree points
% On input:
%     pts (nxm array): sentence semantic tree points
%     probs (1x(n-1) vector): probabilities of KB sentences
% On output:
%     prob_interval (1x2 vector): lower and upper bounds for query
%     prob.
% Call:
%     p = BR_prob_interval_inhull(V',probs);
% Author:
%     T. Henderson
%     UU
%     Spring 2017
%

EPS = 0.000001;
step = 0.001;

prob_interval = [0,1];

if isempty(pts)
    return
end

z = [0:step:1]';
len_z = length(z);
for p = 1:len_z
    testpts(p,:) = [probs,z(p)];
end
res = inhull(testpts,pts);
indexes = find(res);
if ~isempty(indexes)
    prob_interval(1) = z(indexes(1));
    prob_interval(2) = z(indexes(end));
end
```

A.11 BR_KB2nonlinear_function

```
function BR_KB2nonlinear_function(KB,probs,f_name)
% BR_KB2nonlinear_function - convert logical sentences with associated
%   probabilities to a set of nonlinear equations
% On input:
%     KB (KB struct vector): knowledge base CNF
%       (i).clauses (1xk_i vector): disjunction of literals
%     probs (1x(n-1) vector): probabilities of the (n-1) sentences in
%     KB
%     f_name (string): name of .m file function for nonlinear
%     equations
% On output:
%     N/A: writes file f_name.m
% Call:
%     BR_KB2nonlinear_function(KB,probs,'KB4');
% Author:
%     T. Henderson
%     UU
%     Spring 2017
%

if isempty(KB)
    return
end

fd = fopen([f_name,'.m'],'w');
num_sentences = length(KB);
fprintf(fd,'function F = %s(x)\n',f_name);
fprintf(fd,' \n');

for s = 1:num_sentences
    f = BR_sentence2formula(KB(s).clauses,probs(s));
    f = ['F(',int2str(s),') = ',f,';'];
    fprintf(fd,'%s\n',f);
end
fclose(fd);
```

A.12 BR_sentence2formula

```
function F = BR_sentence2formula(s,p)
% BR_sentence2formula - convert logical sentence to nonlinear equation
% On input:
%     s (1xk vector): disjunctive clause
%     p (float): probability s
% On output:
%     F (string): equation for s with probability p
% Call:
%     Fs = BR_sentence2formula([-1,2],0.7);
% Author:
%     T. Henderson
%     UU
%     Spring 2017
%

len_s = length(s);
F = ['-',num2str(p),'+'];

for k = 1:len_s
    combos = nchoosek([1:len_s],k);
    num_combos = length(combos(:,1));
    if rem(k,2)==0
        c_sign = '-';
    else
        c_sign = '+';
    end
    for c = 1:num_combos
        if ~(c==1&k==1)
            F = [F,c_sign];
        end
        for m = 1:k
            atom = s(abs(combos(c,m)));
            if atom>0
                term = ['x(',int2str(abs(atom)),')'];
            else
                term = ['(1-x(',int2str(abs(atom)),'))'];
            end
            F = [F,term];
```

```
            if m<k
                F = [F,'*'];
            end
        end
    end
end
```

A.13 BR_MC_AT

```
function [samples,S_probs_trace] = BR_MC_AT(KB,thm,probs,num_trials)
% BR_MC_AT - use Monte Carlo to determine probability of query
% On input:
%     KB (KB struct vector): CNF knowledge base
%       (i).clauses (1xk_i vector): disjunction of literals
%     thm (KB struct vector): CNF knowledge base (one disjunction)
%       (1).clauses (1xk vector): disjunction of literals
%     probs (1x(n-1) vector): probabilities of the (n-1) KB sentences
%     num_trials (int): number of times to try to generate samples
% On output:
%     samples (mxp array): m samples kept using MCMC
%       columns 1:p-1 are the probabilities of the atoms of the
%       sentences
%       column p is the fitness of the sample (Euclidean distance of
%       known
%       sentence probabilities from the inferred sentence
%       probabilities
%     S_probs_trace (mxn array): sentence probabilities computed from
%     the
%       atom probabilities (columns 1:(n-1)); last column is error
% Call:
%     [sa,St] = BR_MC_AT(KB,thm,probs,1000);
% Author:
%     T. Henderson
%     UU
%     Spring 2017
%

%rng default   % reset seed to default value

vars = BR_vars(KB,thm);
len_KB = length(KB);
num_vars = length(vars);
samples = [rand(1,num_vars),0];
A_probs = samples(1:num_vars);
S_probs_trace = [];

Fp = .01;
```

```
wb = waitbar(0,'MCMC');
for t = 1:num_trials
    waitbar(t/num_trials);
    A_probs(randi(num_vars)) = rand;
    S_probs = zeros(1,len_KB);
    for s = 1:len_KB
        L_probs = A_probs;
        clause = KB(s).clauses;
        len_clause = length(clause);
        for l = 1:len_clause
            if clause(l)<0
                L_probs(abs(clause(l))) = 1 - A_probs(abs(clause(l)));
            end
        end
        S_probs(s) = BR_prob_or(clause,L_probs);
    end
    E = max(abs(probs-S_probs));
    E = sum((probs-S_probs).^2);
    F = 1 - E;
    if Fp==0
        F_ratio = 0;
    else
        F_ratio = F/Fp;
    end
    alpha = min(1,F_ratio);
    if alpha>=1|rand<alpha
        samples = [samples;A_probs,F];
        S_probs_trace = [S_probs_trace;S_probs,E];
        Fp = F;
    end
end
close(wb);
```

A.14 BR_PIKB_query

```
function [Sip,args,prob_intervals] = BR_PIKB_query1(KB,probs,query)
% BR_PIKB_query - query a possibly inconsistent knowledge base
% On input:
%     KB (nx1 conjunctive normal form vector): conjunctive clauses
%       (i).clauses (1xm vector): disjunctive clause
%     probs (1xn vector): KB sentence probabilities
%     query (1x1 conjunctive normal form vector): 1 conjunctive clause
%       (1).clauses (1xp vector): disjunctive clause
% On output:
%     Sip (empty or not): empty indicates proof found (else not empty)
%     args (kxn Boolean array): selects sentences in args (==1)
%     prob_intervals (kx2 array): gives probability of query for each
%     arg
% Call:
%     KB0(1).clauses = [1];
%     thm0(1).clauses = [2];
%     [Sip0,args0,prob0] = BR_PIKB_query(KB0,[0.9999],thm0);
%       Sip0 = 1
%       args0 = []
%       prob0 = 0   1
% Author:
%     T. Henderson
%     UU
%     Spring 2017
%

Sip = 1;
prob_intervals = [0,1];

num_sentences = length(KB);
args = BR_find_args(KB,query);
if isempty(args)
    return
end

num_args = length(args(:,1));
prob_intervals = zeros(num_args,2);
for a = 1:num_args
```

```
    index = 0;
    KB_indexes = [];
    for s = 1:num_sentences
        if args(a,s)==1
            index = index + 1;
            KB_a(index).clauses = KB(s).clauses;
            KB_indexes(index) = s;
        end
    end
    KB_a(index+1).clauses = query(1).clauses;
    pts = BR_find_worlds(KB_a)';
    prob_a = BR_prob_interval_inhull(pts,probs(KB_indexes));
    prob_intervals(a,:) = prob_a;
end
```

References

[1] T. Alsinet, C.I. Chesnevar, L. Godo, and G.R. Simari. A Logic Programming Framework for Possibilistic Argumentation. *Fuzzy Sets and Systems*, 159(10):1208–1228, 2008.

[2] T. Bench-Capon and P.E. Dunne. Argumentation in Artificial Intelligence. *Artificial Intelligence*, 171(10-15):619–641, 2007.

[3] T. Bench-Capon, H. Prakken, and G. Sartor. Argumentation in Legal Reasoning. In I. Rahwan and G.R. Simari, editors, *Argumentation in Artificial Intelligence*, pages 363–382, New York, NY, 2009. Springer Verlag.

[4] P. Besnard, A.J. Garcia, A. Hunter, S. Modgil, H.Prakken, and G.R. Simari. Introduction to Structured Argumentation. *Argument and Computation*, 5(1):1–4, 2014.

[5] P. Besnard and A. Hunter. *Elements of Argumentation*. MIT Press, Cambridge, MA, 2008.

[6] M. Caminada and D. Gabbay. A Logical Account of Formal Argumentation. *Studia Logica*, 93(2):109–145, 2009.

[7] P. Domingos and D. Lowd. *Markov Logic: An Interface Layer for Artificial Intelligence*. Morgan and Claypool, San Rafael, CA, 2009.

[8] P.M. Dung. On the Acceptability of Arguments and its Fundamental Role in Nonmonotonic Reasoning, Logic Programmin and n-Person Games. *Artificial Intelligence*, 77(2):321–357, 1995.

[9] P.M. Dung, R.A. Kowalski, and F. Toni. Assumption-Based Argumentation. In I. Rahwan and G.R. Simari, editors, *Argumentation in Artificial Intelligence*, pages 25–44, New York, NY, 2009. Springer Verlag.

[10] X. Fan and F. Toni. On Computing Argumentative Explanation for Abstract Argumentation. In *Proceedings of 21st European Conference on Artificial Intelligence*, Prague, Czech Republic, august 2014.

[11] V. Gogate and P. Domingos. Probabilistic Theorem Proving. *Communications of the ACM*, 59(7):107–115, 2016.

[12] A. Hunter. A Probabilistic Approach to Modelling Uncertan Logical Arguments. *International Journal of Approximate Reasoning*, 54(1):47–81, January 2013.

[13] R. Kowalsi and P.J. Hayes. Semantic Tress in Automatic Theorem Proving. In J.J. Siekmann and G. Wrightson, editors, *Automation of Reasoning*, pages 217–232, Berlin, 1983.

[14] H. Li, N. Oren, and T. Norman. Probabilistic Argumentation Frameworks. In *Proc. 1st International Workshop on the Theory and Applications of Formal Argumentation*, Beijing, China, August 2011.

[15] N. Nilsson. Probabilistic Logic. *Artificial Intelligence Journal*, 28:71–87, 1986.

[16] L. De Raedt, A. Kimmig, and H. Toivonen. HProbLog: A Probabilistic Prolog and its Application in Link Discovery. pages 2462–2467, 2007.

[17] I. Rahwan and G.R. Simari. *Argumentation in Artificial Intelligence*. Springer Verlag, New York, NY, 2009.

[18] M. Thimm. A Probabilistic Semantics for Abstract Argumentation. In *Proc. 20th European Conference on Artificial Intelligence*, Montpellier, France, August 2012.

[19] F. Toni. A Generalized Framework for Dispute Derivations in Assumption-based Argumentation. *Artificial Intelligence*, 195:1–43, 2013.