Using *NILS* to Solve Probabilistic Satisfiability for CNF Knowledge Bases[a]

Thomas C. Henderson and Michael Cline
University of Utah

UUCS-18-006

[a]This research supported in part by Dynamic Data Driven Application Systems AFOSR grant FA9550-17-1-0077.

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

4 December 2018

Abstract

This report describes the set of functions developed for the Nonlinear Logic Solver (NILS) system (see [1, 2, 3] for details of the method). This allows any logical sentence to be converted to a Conjunctive Normal Form sentence, and then to a standard knowledge base representation wherein each conjunct may be given a probability. Functions are provided which then allow for the solution of PSAT (probabilistic satisfiability) using several alternative algorithms.

Contents

1 Major Capabilities

The major capabilities include:

- Produce Conjunctive Normal Form knowledge base from an arbitrary logical sentence (expressed in fully parenthesized form using the logical operators: equivalence ('='), implies ('>'), and ('^'), or ('v'), and not ('~')).
- Solve PSAT for an arbitrary probabilistic CNF knowledge base using Nilsson's method.
- Solve PSAT for a probabilistic CNF knowledge base with independent variables.
- Solve SSAT for a general probabilistic CNF knowledge base using the nonlinear systems approach.

The functions used for these are described in Appendix A.

2 Logical Sentence to CNF Knowledge Base

The knowledge base representation consists of a set of clauses (disjunctions) with associated probabilities. In Matlab, this is a vector struct with two fields per vector element: (1) clauses, and (2) prob. For example, to represent the Modus Ponens conditions (1. A, and 2. $A \rightarrow B$) with probabilities 0.7 each, we have:

```
KBMP(1).clauses = [1];

KBMP(1).prob = 0.7;

KBMP(2).clauses = [-1,2];

KBMP(2).prob = 0.7;
```

where A is represented by the integer 1, and B is represented by 2. Note that '$A \rightarrow B$' is converted to a disjunction '~AvB'. We show here how to convert a logical sentence to such a knowledge base. An arbitrary logical sentence using integers as logical variables and with connectives {=,>,^,v,~} and in fully parenthesized form may be converted to CNF using *BR_s2CNF*. For example, given a sentence:

```
s = '((((1)^(2))v((3)^(4)))v((5)^(6)))';
```

Then s can be converted:

```
s_CNF = BR_s2CNF(s);
```

which produces:

```
s_CNF = '((((((-7)v((-8)v(1)))^((-7)v((-8)v(2))))^((-7)v
    ((8)v(3))))^((-7)v((8)v(4))))^(((7)v(5))^((7)v(6))))'
```

Now, given such a logical CNF sentence, s, it can be converted to a knowledge base as follows:

```
KB = BR_s2KB(s);
```

Continuing, s_CNF produces the following KB:

```
>> KB(:).clauses

ans = -7  8  4

ans =  7  5

ans =  7  6

ans = -7  8  3

ans = -7 -8  1

ans = -7 -8  2

 >>
```

3 Nilsson's Method for PSAT

Nilsson proposed a method for solving PSAT (i.e., finding a set of consistent probabilities for the complete conjunction set of a set of logical variables). This is done as follows using the Modus Ponens knowledge base:

```
MP_cc_probs = BR_Nilsson(KBMP)

MP_cc_probs =

   0.1500

   0.1500

   0.3000

   0.4000

   >>
```

Queries can then be made to get the probability of new sentences using *BR_Nilsson_query*:

```
query.clauses = [2];

query.prob = 0;

P_query = BR_Nilsson_query (query, P_cc_probs)

P_query = 0.5500

  >>
```

4 Solving PSAT with Independent Variables

Given a KB with independent logical variables (i.e., $P(A \wedge B) = P(A)P(B)$), this can be solved if the set of equations generated from the KB can be solved. Consider Modus Ponens. First, the function to compute the error in the sentence probabilities from a probability assignment to the logical variables is generated:

```
>> BR_KB2F_ind(KBMP, 'KBMPi');
```

This produces files called KBMPi.m and KBMPis.m as follows:

```
>> type KBMPi.m

function F = KBMPi(x)

%

F(1) = -0.7+x(1);

F(2) = -0.7+(1-x(1))+x(2)-(1-x(1))*x(2);

>> type KBMPis.m

   function err = KBMPis(x)

   %

   F(1) = -0.7+x(1);

   F(2) = -0.7+(1-x(1))+x(2)-(1-x(1))*x(2);

   err = norm(F);

   >>
```

where the first produces a vector of the sentence errors, and the latter produces the norm of that vector. The solver is then called:

```
>> [x,e,xt,et] = BR_KB_Jacobian(KBMP,'KBMPi'
   ,0.0001,0.0001,5000,rand(2,1));
```

where the start point is set randomly. The result is:

```
>> x

x = 0.7000

0.5713

>> e

e = 6.1151e-05

>>
```

The solution is then $P(A) = 0.7, P(B) = 0.5713$, and the error in the sentence probabilities (vector norm) is 6.1151×10^{-5}. This can then be used to find the probability of a query:

```
>> BR_prob_or_atom_probs(query.clauses,x) % finding P(B)
   in Modus Ponens

ans = 0.5713

>>
```

If the logical variables are not independent, then this method will not find a solution with low sentence error.

5 Solving SSAT with General KB

SSAT is the Sentence Satisfiability problem which means finding an assignment of probabilities to a set of variables representing the atom probabilities and either conditional or and'ed variables. This is done as follows. First, in addition to creating a sentence error function, it is necessary to create a table which describes all the variables used in the solver. New variables arise when atoms appear together in a KB clause. For example, in Modus Ponens, the clause '~AvB' has probability:

$$P(\sim A \text{v} B) = P(\sim A) + P(B)\check{}P(\sim A|B)P(B)$$

Thus, in addition to the variables for $P(A)$ and $P(B)$ (note that $P(\sim A) = 1\check{}P(A)$), there is a variable for $P(\sim A|B)$. In order to ensure consistency, three other equations are added to the system representing the constraints between $P(A|B), P(A|\sim B)$ and $P(\sim A|\sim B)$. These require three additional variables.

The sentence error function is set up by:

```
>> [vtableMP,FMP] = BR_KB2F_gen(KBMP,'KBMPg');
```

where KBMPg is the sentence error function:

```
>> type KBMPg.m

function F = KBMPg(x)

%

F(1) = -0.7 + (x(1));

F(2) = -0.7 + (1-x(1)) + x(2) - x(3)*x(2);

F(3) = -x(6) + 1 - x(3);

F(4) = -x(5) + 1 - x(4);

F(5) = -x(4) + (1-(x(1) + x(2) - x(6)*x(2)))/(1-x(2));

>>
```

The outputs of the call are vtableMP:

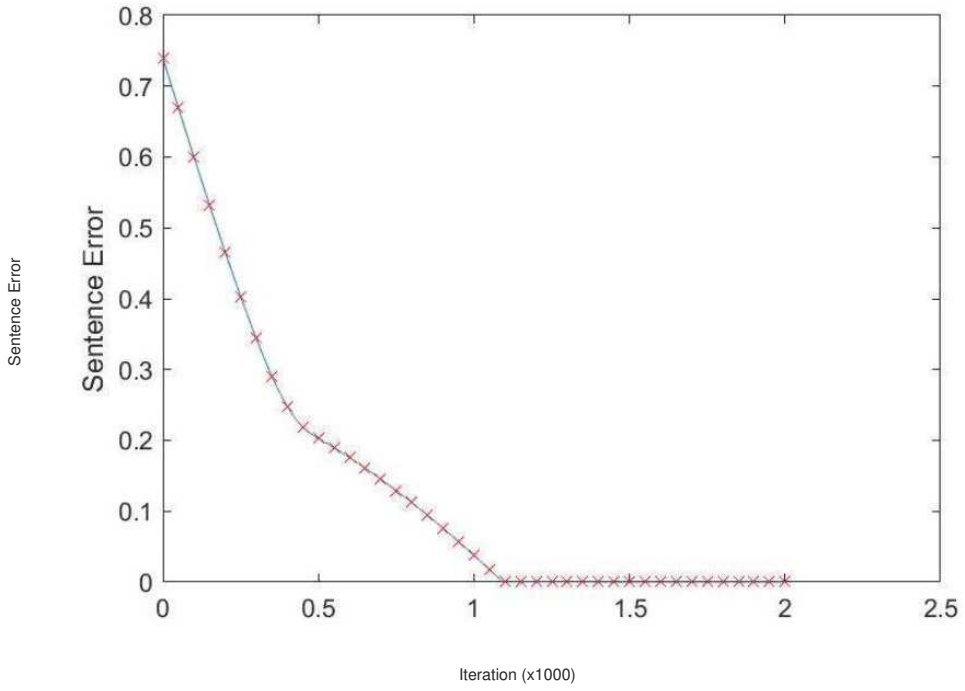
```
>> vtableMP =
 1   0   0   0   1   1   0

 2   0   0   0   1   2   0

-1   2   0   2   1   3   0

-1  -2   0   2   0   4   0

 1  -2   0   2   2   5   0

 1   2   0   2   3   6   0

>>
```

The vector of unknowns has the size of the number of rows in the vtable, and a solution is found as:

```
>> [x,e,xt,et] = BR_KB_Jacobian(KBMP,'KBMPg',...
0.0001,0.0001,20000,rand(6,1));

>> e = 1.7592e-04

>> x =
```

```
0.7000

0.6291

0.3641

0.1912

0.8088

0.6359

>>
```

where the first two values represent P(A) and P(B). It is possible to see how the search converges by plotting it:



References

[1] T.C. Henderson, A. Mitiche, R. Simmons, and X. Fan. A preliminary study of probabilistic argumentation. Technical Report UUCS-17-001, University of Utah, Salt Lake City, UT, February, 2017.

[2] T.C. Henderson, R. Simmons, A. Mitiche, X. Fan, and D. Sacharny. A probabilistic logic for multi-source heterogenous information fusion. In *Proceedings of 2017 IEEE Conference on Multisensor Fusion and Integration (MFI)*, Daegu, South Korea, 15-18 November, 2017.

[3] T.C. Henderson, R. Simmons, B. Serbinowski, X. Fan, A. Mitiche, and M. Cline. Probabilistic logic for intelligent systems. In *Proceedings of 2018 International Conference on Intelligent Autonomous Systems*, Baden-Baden, Germany, 11-15 June, 2018.

Appendix

BR_3PSAT.m

```
function pp = BR_3PSAT(C)
% BR_2PSAT - converts a CNF clause of length k to a set of clauses
% each of
%     which has at most 3 terms (disjuncts)
% On input:
%     C (KB structure element): one element of a KB
%       C.clauses (vector): disjunction
%       C.prob (float): probability of clause
% On output:
%     pp (float): probability of new clauses
% Call:
%     pp = BR_3PSAT(KB(1));
% Author:
%     T. Henderson
%     UU
%     Fall 2018
%

k = length(C.clauses);
p = C.prob;
kp = k - 2;
Nv = 2*k - 3;
Ncc = 2^Nv;
Nt = 2^(2*k-4) + 2^(2*k-5) + 2^(2*k-6);
Nf = Ncc - Nt;
No = (kp-1)*(kp-2)*2^(kp-3);
Nand = Ncc - (kp*Nf-No);
pa = p/Nand;
pn = (1-p)/(Ncc-Nand);
pp = p + (Nt-Nand)*pn;
res = [k,kp,Nv,Ncc,Nt,Nf,No,Nand]';
resp = [p,pa,pn,pp]';
```

BR_KB2F_gen.m

```
function [vtable,F] = BR_KB2F_gen(KB,fname)
% BR_KB2F_gen - produce error function for gradient descent
% On input:
%     KB (KB struct): CNF KB
%     fname (string): name of error function
% On output:
%     vtable (kxw array): variable table; identifies conditional
```

```
%      variables
%      F (string): text of error function
% Call:
%      [vt,F] = BR_KB2F_gen(KB,'MP');
% Author:
%      T. Henderson
%      UU
%      Fall 2017
%

OP_AND = 1;
OP_OR = 2;

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

len_vtable = 0;
n = length(BR_vars(KB,[]));
num_x = n;

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

vtable = BR_create_vtable(n);

for c = 1:len_KB
    clause = KB(c).clauses;
    p = KB(c).prob;
    switch length(clause)
        case 1  % [L1]
            L1 = clause(1);
            if L1<1
                F(c).f = ['F(',num2str(c),') = -',num2str(p),...
                    ' + (1-x(',num2str(abs(L1)),'));'];
            else
                F(c).f = ['F(',num2str(c),') = -',num2str(p),...
                    ' + (x(',num2str(abs(L1)),'));'];
            end
        case 2  % [L1,L2]
            L1 = clause(1);
            L2 = clause(2);
            parity = BR_parity(L1,L2,0);
            index = BR_get_vtable(vtable,L1,L2,0,0,parity,0);
            if isempty(index)
```

```
            [vtable,index,ne] = BR_put_vtable(vtable,L1,L2,0,c,0);
        end
        if L1<1
            s1 = ['(1-x(',num2str(abs(L1)),'))'];
        else
            s1 = ['x(',num2str(abs(L1)),')'];
        end
        if L2<1
            s2 = ['(1-x(',num2str(abs(L2)),'))'];
        else
            s2 = ['x(',num2str(abs(L2)),')'];
        end
        F(c).f = ['F(',num2str(c),') = -',num2str(p),...
            ' + ',s1,' + ',s2,' - x(',num2str(index),')*',s2,';'];
    case 3  % [L1,L2,L3]
        L1 = clause(1);
        L2 = clause(2);
        L3 = clause(3);
        p12 = BR_parity(L1,L2,0);
        index12 = BR_get_vtable(vtable,L1,L2,0,0,p12,0);
        if isempty(index12)
            [vtable,index12,ne] =
            BR_put_vtable(vtable,L1,L2,0,c,0);
        end
        p13 = BR_parity(L1,L3,0);
        index13 = BR_get_vtable(vtable,L1,L3,0,0,p13,0);
        if isempty(index13)
            [vtable,index13,ne] =
            BR_put_vtable(vtable,L1,L3,0,c,0);
        end
        p23 = BR_parity(L2,L3,0);
        index23 = BR_get_vtable(vtable,L2,L3,0,0,p23,0);
        if isempty(index23)
            [vtable,index23,ne] =
            BR_put_vtable(vtable,L2,L3,0,c,0);
        end
        p123 = BR_parity(L1,L2,L3);
        index123 = BR_get_vtable(vtable,L1,L2,L3,0,p123,OP_AND);
        if isempty(index123)
            [vtable,index123,ne] =
            BR_put_vtable(vtable,L1,L2,L3,c,...
                OP_AND);
        end
        if L1<1
            s1 = ['(1-x(',num2str(abs(L1)),'))'];
        else
            s1 = ['x(',num2str(abs(L1)),')'];
        end
        if L2<1
```

```
                s2 = ['(1-x(',num2str(abs(L2)),'))'];
            else
                s2 = ['x(',num2str(abs(L2)),')'];
            end
            if L3<1
                s3 = ['(1-x(',num2str(abs(L3)),'))'];
            else
                s3 = ['x(',num2str(abs(L3)),')'];
            end
            F(c).f = ['F(',num2str(c),') = -',num2str(p),...
                ' + ',s1,' + ',s2,' + ',s3,' -
                x(',num2str(index12),...
                ')*',s2,' - x(',num2str(index13),')*',s3,...
                ' - x(',num2str(index23),')*',s3,' + x(',...
                num2str(index123),')*x(',num2str(index23),')*',s3,';'];
    end
end

num_eqn = len_KB;
%-------    Add Constraints ----------------
num_x = length(vtable(:,1));
ca = zeros(num_x,1);
ca(1:n) = 1;
for k = n+1:num_x
    if ca(k)==0
        if vtable(k,3)==0&vtable(2)~=0
            L1 = vtable(k,1);
            L2 = vtable(k,2);
            c = vtable(k,4);
            indexes = zeros(4,1);
            pp = BR_parity(-abs(L1),-abs(L2),0);
            index = BR_get_vtable(vtable,-abs(L1),-abs(L2),0,0,pp,0);
            if isempty(index)
                [vtable,index,ne] = BR_put_vtable(vtable,-abs(L1),...
                    -abs(L2),0,vtable(k,4),0);
            end
            indexes(1) = index;
            pp = BR_parity(-abs(L1),abs(L2),0);
            index = BR_get_vtable(vtable,-abs(L1),abs(L2),0,0,pp,0);
            if isempty(index)
                [vtable,index,ne] = BR_put_vtable(vtable,-abs(L1),...
                    abs(L2),0,vtable(k,4),0);
            end
            indexes(2) = index;
            pp = BR_parity(abs(L1),-abs(L2),0);
            index = BR_get_vtable(vtable,abs(L1),-abs(L2),0,0,pp,0);
            if isempty(index)
                [vtable,index,ne] = BR_put_vtable(vtable,abs(L1),...
                    -abs(L2),0,vtable(k,4),0);
```

```
end
indexes(3) = index;
pp = BR_parity(abs(L1),abs(L2),0);
index = BR_get_vtable(vtable,abs(L1),abs(L2),0,0,pp,0);
if isempty(index)
    [vtable,index,ne] = BR_put_vtable(vtable,abs(L1),...
        abs(L2),0,vtable(k,4),0);
end
indexes(4) = index;
indexa1 = abs(L1);
indexa2 = abs(L2);
indexp0 = indexes(1);
indexp1 = indexes(2);
indexp2 = indexes(3);
indexp3 = indexes(4);
if vtable(k,5)==0
    % Equation 1
    num_eqn = num_eqn + 1;
    F(num_eqn).f = ['F(',num2str(num_eqn),') = x(',...
        num2str(indexp0),')*(1-x(',num2str(indexa2),...
        '))-1+(1-x(',num2str(indexa1),'))+(1-x(',...
        num2str(indexa2),'))-x(',num2str(indexp3),...
        ')*(1-x(',num2str(indexa2),'));'];
    % Equation 2
    num_eqn = num_eqn + 1;
    F(num_eqn).f = ['F(',num2str(num_eqn),') = -x(',...
        num2str(indexp2),') + 1 -
        x(',num2str(indexp0),');'];
    % Equation 3
    num_eqn = num_eqn + 1;
    F(num_eqn).f = ['F(',num2str(num_eqn),') = -x(',...
        num2str(indexp1),') + 1 -
        x(',num2str(indexp3),');'];
end
if vtable(k,5)==1
    % Equation 1
    num_eqn = num_eqn + 1;
    F(num_eqn).f = ['F(',num2str(num_eqn),') = -x(',...
        num2str(indexp3),') + 1 -
        x(',num2str(indexp1),');'];
    % Equation 2
    num_eqn = num_eqn + 1;
    F(num_eqn).f = ['F(',num2str(num_eqn),') = -x(',...
        num2str(indexp2),') + 1 -
        x(',num2str(indexp0),');'];
    % Equation 3
    num_eqn = num_eqn + 1;
    F(num_eqn).f = ['F(',num2str(num_eqn),') = -x(',...
        num2str(indexp0),') + (1-(x(',num2str(indexa1),...
```

```
            ') + x(',num2str(indexa2),') -
            x(',num2str(indexp3),...
            ')*x(',num2str(indexa2),')))/(1-x(',num2str(indexa2),...
            '));'];
    end
    if vtable(k,5)==2
        % Equation 1
        num_eqn = num_eqn + 1;
        F(num_eqn).f = ['F(',num2str(num_eqn),') = -x(',...
            num2str(indexp2),') + 1 -
            x(',num2str(indexp0),');'];
        % Equation 2
        num_eqn = num_eqn + 1;
        F(num_eqn).f = ['F(',num2str(num_eqn),') = -x(',...
            num2str(indexp3),') + (x(',num2str(indexp0),...
            ')*(-x(',num2str(indexa2),') - 1 + x(',...
            num2str(indexa1),') +
            x(',num2str(indexa2),')))/x('...
            num2str(indexa2),');'];
        % Equation 3
        num_eqn = num_eqn + 1;
        F(num_eqn).f = ['F(',num2str(num_eqn),') = -x(',...
            num2str(indexp1),') + 1 -
            x(',num2str(indexp3),');'];
    end
    if vtable(k,5)==3
        % Equation 1
        num_eqn = num_eqn + 1;
        F(num_eqn).f = ['F(',num2str(num_eqn),') = -x(',...
            num2str(indexp1),') + 1 -
            x(',num2str(indexp3),');'];
        % Equation 2
        num_eqn = num_eqn + 1;
        F(num_eqn).f = ['F(',num2str(num_eqn),') = -x(',...
            num2str(indexp0),') + (1-(x(',num2str(indexa1),...
            ') + x(',num2str(indexa2),') -
            x(',num2str(indexp3),...
            ')*x(',num2str(indexa2),')))/(1-x(',num2str(indexa2),...
            '));'];
        % Equation 3
        num_eqn = num_eqn + 1;
        F(num_eqn).f = ['F(',num2str(num_eqn),') = -x(',...
            num2str(indexp2),') + 1 -
            x(',num2str(indexp0),');'];
    end
elseif vtable(k,3)~=0
    L1 = vtable(k,1);
    L2 = vtable(k,2);
    L3 = vtable(k,3);
```

```
c = vtable(k,4);
indexes = zeros(4,1);
pp = BR_parity(-L1,-L2,-L3);
index = BR_get_vtable(vtable,-L1,-L2,-L3,0,pp,OP_OR);
if isempty(index)
    [vtable,index,ne] =
    BR_put_vtable(vtable,-L1,-L2,-L3,...
        vtable(k,4),OP_OR);
end
indexes(1) = index;
pp = BR_parity(-L1,L2,L3);
index = BR_get_vtable(vtable,-L1,L2,L3,0,pp,OP_AND);
if isempty(index)
    [vtable,index,ne] =
    BR_put_vtable(vtable,-L1,L2,L3,...
        vtable(k,4),OP_AND);
end
indexes(2) = index;
pp = BR_parity(L1,-L2,-L3);
index = BR_get_vtable(vtable,L1,-L2,-L3,0,pp,OP_OR);
if isempty(index)
    [vtable,index,ne] =
    BR_put_vtable(vtable,L1,-L2,-L3,...
        vtable(k,4),OP_OR);
end
indexes(3) = index;
pp = BR_parity(L1,L2,L3);
index = BR_get_vtable(vtable,L1,L2,L3,0,pp,OP_AND);
if isempty(index)
    [vtable,index,ne] = BR_put_vtable(vtable,L1,L2,L3,...
        vtable(k,4),OP_AND);
end
indexes(4) = index;
indexa1 = abs(L1);
indexa2 = abs(L2);
indexa3 = abs(L3);
[vv,index23,ne] = BR_vtable(vtable,L2,L3,0,-1,0);
indexp0 = indexes(1);
indexp1 = indexes(2);
indexp2 = indexes(3);
indexp3 = indexes(4);
if vtable(k,5)==0
    % Equation 1
    num_eqn = num_eqn + 1;
    F(num_eqn).f = ['F(',num2str(num_eqn),') = x(',...
        num2str(indexp0),')*(1-x(',num2str(index23),...
        ')*(1-x(',num2str(indexa3),')))-1+(1-x(',...
        num2str(indexa1),'))+x(',num2str(index23),')*(1-x(',...
        num2str(indexa3),'))-x(',num2str(indexp3),')*x(',...
```

```
            num2str(index23),')*(1-x(',num2str(indexa3),'));'];
        % Equation 2
        num_eqn = num_eqn + 1;
        F(num_eqn).f = ['F(',num2str(num_eqn),') = -x(',...
            num2str(indexp2),') + 1 -
            x(',num2str(indexp0),');'];
        % Equation 3
        num_eqn = num_eqn + 1;
        F(num_eqn).f = ['F(',num2str(num_eqn),') = -x(',...
            num2str(indexp1),') + 1 -
            x(',num2str(indexp3),');'];
    end
    if vtable(k,5)==3
        % Equation 1
        num_eqn = num_eqn + 1;
        F(num_eqn).f = ['F(',num2str(num_eqn),') = x(',...
            num2str(indexp1),') - 1 +
            x(',num2str(indexp3),');'];
        % Equation 2
        num_eqn = num_eqn + 1;
        F(num_eqn).f = ['F(',num2str(num_eqn),') = x(',...
            num2str(indexp0),')*(1-x(',num2str(index23),...
            ')*x(',num2str(indexa3),'))-1+(1-x(',...
            num2str(indexa1),'))+x(',num2str(index23),')*x(',...
            num2str(indexa3),')-x(',num2str(indexp3),')*x(',...
            num2str(index23),')*x(',num2str(indexa3),');'];
        % Equation 3
        num_eqn = num_eqn + 1;
        F(num_eqn).f = ['F(',num2str(num_eqn),') = -x(',...
            num2str(indexp2),') + 1 -
            x(',num2str(indexp0),');'];
    end
    if vtable(k,5)==4  % P(A|~B)
        % Equation 1
        num_eqn = num_eqn + 1;
        F(num_eqn).f = ['F(',num2str(num_eqn),') = x(',...
            num2str(indexp0),') - 1 +
            x(',num2str(indexp2),');'];
        % Equation 2
        num_eqn = num_eqn + 1;
        F(num_eqn).f = ['F(',num2str(num_eqn),') = x(',...
            num2str(indexp0),')*(1-x(',num2str(index23),...
            ')*x(',num2str(indexa3),'))-1+x(',...
            num2str(indexa1),')+x(',num2str(index23),')*x(',...
            num2str(indexa3),')+x(',num2str(indexp3),')*x(',...
            num2str(index23),')*x(',num2str(indexa3),');'];
        % Equation 3
        num_eqn = num_eqn + 1;
        F(num_eqn).f = ['F(',num2str(num_eqn),') = x(',...
```

```
                    num2str(indexp1),') - 1 +
                    x(',num2str(indexp3),');'];
            end
            if vtable(k,5)==7  % P(A|B)
                % Equation 1
                num_eqn = num_eqn + 1;
                F(num_eqn).f = ['F(',num2str(num_eqn),') = x(',...
                    num2str(indexp1),') - 1 +
                    x(',num2str(indexp3),');'];
                % Equation 2
                num_eqn = num_eqn + 1;
                F(num_eqn).f = ['F(',num2str(num_eqn),') = x(',...
                    num2str(indexp0),')*(1-x(',num2str(index23),...
                    ')*x(',num2str(indexa3),'))-1+x(',...
                    num2str(indexa1),')+x(',num2str(index23),')*x(',...
                    num2str(indexa3),')+x(',num2str(indexp3),')*x(',...
                    num2str(index23),')*x(',num2str(indexa3),');'];
                % Equation 3
                num_eqn = num_eqn + 1;
                F(num_eqn).f = ['F(',num2str(num_eqn),') = x(',...
                    num2str(indexp2),') - 1 +
                    x(',num2str(indexp0),');'];
            end
            tch = 0;
        end
    end
end

%-------------------------------------------
for c = 1:num_eqn
    fprintf(fd,'%s\n',F(c).f);
end
fclose(fd);

return

%-------------------------------------------------------------------------
function vars = BR_vars(KB,sentence)
% BR_vars - find list of variables in logical sentences
% On input:
%     KB (nx1 conjunctive normal form vector): conjunctive clauses
%       (i).clauses (1xm vector): disjunctive clause
%     sentence (1x1 conjunctive normal form vector): conjunctive
%     clause
%       (1).clauses (1xk vector): disjunctive clause
% On output:
%     vars (1xp vector): list of variables in KB and sentence
% Call:
%     vars = BR_vars(KB,thm);
```

```
% Author:
%      T. Henderson
%      UU
%      Spring 2017
%

vars = [];

for s = 1:length(KB)
    vars = unique([vars,abs(KB(s).clauses)]);
end
for s = 1:length(sentence)
    vars = unique([vars,abs(sentence(s).clauses)]);
end
vars = sort(vars);
```

BR_KB2F_ind.m

```
function BR_KB2F_ind(KB,fname)
% BR_KB2F_ind - convert KB to .m function file
% On input:
%      KB (KB struct): CNF KB
%      fname (string): root name of file
% On output:
%      N/A - side effect is creation of 2 files:
%        fname.m: has vector function for sentence probabilities
%        fnames.m: scalar output function (norm(F) from fname.m)
% Call:
%      BR_KB2F_ind(KB,'IJ');
% Author:
%      T. Henderson
%      UU
%      Fall 2017
%

num_clauses = length(KB);

% fsolve function (vector output)
file_name = [fname,'.m'];

fd = fopen(file_name,'w');

fprintf(fd,'function F = %s(x)\n',fname);
fprintf(fd,'%%%\n\n');

for c = 1:num_clauses
    F = BR_sentence2formula(KB(c).clauses,KB(c).prob);
    F = ['F(',num2str(c),') = ',F];
```

```
    fprintf(fd,'%s;\n',F);
end
fclose(fd);

% fmincon function (scalar - error output)
file_name = [fname,'s.m'];

fd = fopen(file_name,'w');

fprintf(fd,'function err = %s(x)\n',[fname,'s']);
fprintf(fd,'%%\n\n');

for c = 1:num_clauses
    F = BR_sentence2formula(KB(c).clauses,KB(c).prob);
    F = ['F(',num2str(c),') = ',F];
    fprintf(fd,'%s;\n',F);
end
fprintf(fd,'err = norm(F);\n');
fclose(fd);

%-------------------------------------------------------------------------
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
```

BR_KB2s.m

```
function s = BR_KB2s(KB)
% BR_KB2s - convert KB to string
% On input:
%     KB (KB structure): knowledge base
%       (k).clauses (vector): disjunction of literals
%       (k).prob (float): probability of clause
% On output:
%     s (string): conjunction of disjunction
%       not is pushed into the atoms as -<int> (no negations left)
% Call:
%     s1 = BR_KB2s(KB);
% Author:
%     T. Henderson
%     UU
%     Fall 2018
%

num_clauses = length(KB);

s = [];

for c = 1:num_clauses
    clause = KB(c).clauses;
    len_clause = length(clause);
    cs = [];
    if len_clause==1
        if clause(1)>0
            cs = ['(',num2str(clause(1)),')'];
        else
```

```
            cs = ['(~(',num2str(abs(clause(1))),'))'];
        end
    else
        if clause(1)>0
            cs = ['(',num2str(clause(1))];
        else
            cs = ['(','~(',num2str(abs(clause(1))),')'];
        end
        for d = 2:len_clause-1
            if clause(d)>0
                cs = ['(',cs,')','v','(',num2str(clause(d)),')'];
            else
                cs =
                ['(',cs,')','v','(~(',num2str(abs(clause(d))),'))'];
            end
        end
        if clause(end)>0
            cs = ['(',cs,')','v','(',num2str(clause(end)),')',')'];
        else
            cs = ['(',cs,')','v(~(',num2str(abs(clause(end))),')))'];
        end
    end
    if c<2
        s = [s,cs];
    else
        s = ['(',s,'^',cs,')'];
    end
end
```

BR_KB_GD.m

```
function [x,e,xt,et] = BR_KB_GD(KB,KBF,alpha,tol,max_iter,x0)
% BR_KB_Jacobian - gradient descent to solve PSAT
% On input:
%     KB (KB struct): CNF KB
%     KBF (string): name of error function
%     alpha (float): step size in gradient direction
%     tol (float): quitting tolerance
%     max_iter (int): maximum number of steps to take
%     x0 (kx1 vector): initial starting value
%       * for independent variable, k is n (number of atoms)
%       * if not, k is number of rows in vtable from BR_KB2F_gen call
% On output:
%     x (kx1 vector): final gradient descent vector
%     e (float): sentence probability error
%     xt (mxk array): trace of x values during descent
%     et (mx1 vector): trace of sentence error values
% Call:
```

```
%     % indepenent variables example
%     KB(1).clauses = [1];
%     KB(1).prob = 0.7;
%     KB(2).clauses = [-1,2];
%     KB(2).prob = 0.7;
%     BR_KB2F_ind(KB,'KB');
%     [x,e,xt,et] = BR_KB_GD(KB,KBF,0.001,0.01,10000,0.5*ones(2,1));
%     % indepenent variables example
%     KB(1).clauses = [1];
%     KB(1).prob = 0.7;
%     KB(2).clauses = [-1,2];
%     KB(2).prob = 0.7;
%     [vt,F] = BR_KB2F_gen(KB,'KB');
%     v_len = length(vt(:,1));
%     [x,e,xt,et] =
%     BR_KB_GD(KB,KBF,0.001,0.01,10000,0.5*ones(v_len,1));
% Author:
%     T. Henderson
%     UU
%     Fall 2017
%

n = length(x0);
done = 0;
iter = 0;
x = x0;
xt = x0';
vt = feval(KBF,x0);
et = norm(vt);
e = et(end);
wb = waitbar(0,'Jacobian');
done = et(end)<tol;
while done==0
    waitbar(iter/max_iter);
    iter = iter + 1;
    J = zeros(n,1);
    for d = 1:n
        xm = x;
        xm(d) = x(d) - alpha;
        vm = norm(feval(KBF,xm));
        xp = x;
        xp(d) = x(d) + alpha;
        vp = norm(feval(KBF,xp));
        J(d) = vm - vp;
    end
    if e>tol&norm(J)==0
        J = 0.5 - rand(n,1);
    end
    J = J/norm(J);
```

```
    x = abs(x + alpha*J);
    x = max(0,min(x,1));
    xt = [xt; x'];
    et = [et;norm(feval(KBF,x))];
    e = et(end);
    if et(end)<tol||(iter==max_iter)
        done = 1;
    end
end
close(wb);
e = et(end);
```

BR_KB_Jacobian.m

```
function [x,e,xt,et] = BR_KB_Jacobian(KB,KBF,alpha,tol,max_iter,x0)
% BR_KB_Jacobian - gradient descent to solve PSAT
% On input:
%     KB (KB structure): knowledge base
%     KBF (string): KB function name
%     alpha (float): step size
%     tol (float): termination tolerance
%     max_iter (int): max number of iterations to try
%     x0 (vector): initial guess
% On output:
%     x (nx1 vector): solution
%     e (float): sentence error generated by solution
%     xt (kxn array): trace of descent
%     et (kx1 vector): trace of sentence error
% Call:
%     [x,e,xt,et] =
%     BR_KB_Jacobian(KB1,'KB1i',0.001,0.001,1000,rand(20,1));
% Author:
%     T. Henderson
%     UU
%      Spring 2018
%

n = length(x0);
done = 0;
iter = 0;
x = x0;
xt = x0';
vt = feval(KBF,x0);
et = norm(vt);
e = et(end);
wb = waitbar(0,'Jacobian');
while done==0
    waitbar(iter/max_iter);
```

```
    iter = iter + 1;
    J = zeros(n,1);
    for d = 1:n
        xm = x;
        xm(d) = x(d) - alpha;
        vm = norm(feval(KBF,xm));
        xp = x;
        xp(d) = x(d) + alpha;
        vp = norm(feval(KBF,xp));
        J(d) = vm - vp;
    end
    if e>tol&norm(J)==0
        J = 0.5 - rand(n,1);
    end
    J = J/norm(J);
    x = abs(x + alpha*J);
    x = max(0,min(x,1));
    xt = [xt; x'];
    et = [et;norm(feval(KBF,x))];
    e = et(end);
    if et(end)<tol||(iter==max_iter)
        done = 1;
    end
end
close(wb);
e = et(end);
```

BR_KB_query_prob.m

```
function query_prob = BR_KB_query_prob(KB,vtable,x,query)
% BR_KB_query_prob - determine probability of KB query
% On input:
%     KB (KB struct): knowledge base CNF
%     vtable (kx7 array): variable table
%     x (kx1 vector): variable probabilities
%     query (1xp vector): p is 1, 2 or 3 (a disjunction)
% On output:
%     query_prob (float): probability of query
% Call:
%     p = BR_KB_query_prob(KB1,vtable1,x1,[1,2,-3]);
% Author:
%     T. Henderson
%     UU
%     Summer 2018
%

query_prob = 0;
F = [];
```

```
if isempty(query)
    return
end

if length(query)==1
    atom1 = abs(query);
    if query>0
        query_prob = x(atom1);
    else
        query_prob = 1 - x(atom1);
    end
end

if length(query)==2
    L1 = query(1);
    L2 = query(2);
    atom1 = abs(L1);
    atom2 = abs(L2);
    if L1>0
        p1 = x(atom1);
    else
        p1 = 1 - x(atom1);
    end
    if L2>0
        p2 = x(atom2);
    else
        p2 = 1 - x(atom2);
    end
    index = BR_get_vtable(vtable,L1,L2,0,0,BR_parity(L1,L2,0),0);
    if ~isempty(index)
        query_prob = p1 + p2 - x(index)*p2;
    else
        F = ['F(1) = ',num2str(p1),' + ',num2str(p2),' - x(2)*',...
            num2str(p2),' - x(1);'];
    end
end

if ~isempty(F)&length(query)==2
    fd = fopen('tjqwa.m','w');
    fprintf(fd,'function F = %s(x)\n','tjqwa');
    fprintf(fd,'%%%\n\n');
    fprintf(fd,'%s\n',F);
    fclose(fd);
    [xg,eg,xt,et] = BR_KB_Jacobian(KB,'tjqwa',0.0001,0.0001,20000,...
        0.5*ones(2,1));
    query_prob = xg(1);
end
```

```
if length(query)==3
    L1 = query(1);
    L2 = query(2);
    L3 = query(3);
    atom1 = abs(L1);
    atom2 = abs(L2);
    atom3 = abs(L3);
    if L1>0
        p1 = x(atom1);
    else
        p1 = 1 - x(atom1);
    end
    if L2>0
        p2 = x(atom2);
    else
        p2 = 1 - x(atom2);
    end
    if L3>0
        p3 = x(atom3);
    else
        p3 = 1 - x(atom3);
    end
    num_unknowns = 1;
    index12 = BR_get_vtable(vtable,L1,L2,0,0,BR_parity(L1,L2,0),0);
    index13 = BR_get_vtable(vtable,L1,L3,0,0,BR_parity(L1,L3,0),0);
    index23 = BR_get_vtable(vtable,L2,L3,0,0,BR_parity(L2,L3,0),0);
    index123 = BR_get_vtable(vtable,L1,L2,L3,0,BR_parity(L1,L2,L3),1);
    if ~isempty(index12)
        s12 = num2str(x(index12));
    else
        num_unknowns = num_unknowns + 1;
        s12 = ['x(',num2str(num_unknowns),')'];
    end
    if ~isempty(index13)
        s13 = num2str(x(index13));
    else
        num_unknowns = num_unknowns + 1;
        s13 = ['x(',num2str(num_unknowns),')'];
    end
    if ~isempty(index23)
        s23 = num2str(x(index23));
    else
        num_unknowns = num_unknowns + 1;
        s23 = ['x(',num2str(num_unknowns),')'];
    end
    if ~isempty(index123)
        s123 = num2str(x(index123));
    else
        num_unknowns = num_unknowns + 1;
```

```
        s123 = ['x(',num2str(num_unknowns),')'];
    end
    F = ['F(1) = -x(1) + ',num2str(p1),' + ',num2str(p2),' + ',...
        num2str(p3),' - ',s12,'*',num2str(p2),' -
        ',s13,'*',num2str(p3),...
        ' - ',s23,'*',num2str(p3),' +
        ',s123,'*',s23,'*',num2str(p3),';'];
end

if ~isempty(F)&length(query)==3
    fd = fopen('tjqwa.m','w');
    fprintf(fd,'function F = %s(x)\n','tjqwa');
    fprintf(fd,'%%\n\n');
    fprintf(fd,'%s\n',F);
    fclose(fd);
    [xg,eg,xt,et] = BR_KB_Jacobian(KB,'tjqwa',0.0001,0.0001,20000,...
        0.5*ones(num_unknowns,1));
    query_prob = xg(1);
end

tch = 0;
```

BR_Nilsson_method_all.m

```
function [P_cc,p_query] = BR_Nilsson_method_all(KB,query)
% BR_Nilsson_method - Nilsson's probabilistic logic method (max value)
% On input:
%     KB (KB struct): KB clauses
%     query (KB struct): query
% On output:
%     P_cc (kx1 vector): model probabilities
%     p_query (float): probability of query
% Call:
%     KB(1).clauses = [1];
%     KB(1).prob = 0.7;
%     KB(2).clauses = [-1,2];
%     KB(2).prob = 0.7;
%     query.clauses = [2];
%     query.prob = 0;
%     [P,pt] = BR_Nilsson_method(KB,query);
% Author:
%     T. Henderson
%     UU
%     Summer 2018
%

p_query = 0;
```

```
num_sentences = length(KB);
vars = BR_vars(KB,query);
num_vars = length(vars);
num_cc = 2^num_vars;
query_sat = ones(1,num_cc);
V = zeros(num_sentences+1,num_cc);
P_s = zeros(num_sentences+1,1);
V(end,:) = ones(1,num_cc);
P_s(end) = 1;

for s = 1:num_sentences
    P_s(s) = KB(s).prob;
    for cc = 0:num_cc-1
        bits = BR_int2bits(cc,num_vars);
        V(s,cc+1) = BR_sat(KB(s).clauses,bits);
        query_sat(cc+1) = query_sat(cc+1)&BR_sat(query.clauses,bits);
    end
end

%P_cc = V\P_s;
P_cc = lsqlin(V,P_s,[],[],[],[],zeros(num_cc,1),ones(num_cc,1));
p_query = query_sat*P_cc;

%-------------------------------------------------------------------------
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);

%-------------------------------------------------------------------------
```

```
function b = BR_sat(sentence,bits)
% BR_sat - check if bits satisfy the sentence
% On input:
%     sentence (1xk vector): disjunction of k literals
%     bits (1xn vector): n truth assignments to logical variables
% On output:
%     b (Boolean): 1 if bits satisfy sentence; else 0
% Call:
%     b = BR_sat([1,-2],[0,1]);
% Author:
%     T. Henderson
%     UU
%     Summer 2018
%

b = 0;

len_sentence = length(sentence);
for e = 1:len_sentence
    if sentence(e)>0==bits(abs(sentence(e)))
        b = 1;
        return
    end
end

%-------------------------------------------------------------------------
function vars = BR_vars(KB,sentence)
% BR_vars - find list of variables in logical sentences
% On input:
%     KB (nx1 conjunctive normal form vector): conjunctive clauses
%       (i).clauses (1xm vector): disjunctive clause
%     sentence (1x1 conjunctive normal form vector): conjunctive
%     clause
%       (1).clauses (1xk vector): disjunctive clause
% On output:
%     vars (1xp vector): list of variables in KB and sentence
% Call:
%     vars = BR_vars(KB,thm);
% Author:
%     T. Henderson
%     UU
%     Spring 2017
%

vars = [];

for s = 1:length(KB)
    vars = unique([vars,abs(KB(s).clauses)]);
end
```

```
for s = 1:length(sentence)
    vars = unique([vars,abs(sentence(s).clauses)]);
end
vars = sort(vars);
```

BR_PSAT2PSAT3.m

```
function KB3PSAT = BR_PSAT2PSAT3(KB)
% BR_PSAT2PSAT3 - convert a probabilistic CNF KB to a 3CNF
% probabilistic KB
% On input:
%     KB (KB structure): CNF probabilistic knowledge base
% On output:
%     KB3PSAT (KB structure): 3CNF probabilistic knowledge base
% Call:
%     KB3 = BR_PSAT23PSAT(KB);
% Author:
%     T. Henderson
%     UU
%     Summer 2018
%

KB3PSAT = [];
len_KB3PSAT = 0;
num_clauses = length(KB);
vars = BR_vars(KB,[]);
num_vars = max(vars);

for c = 1:num_clauses
    clause = KB(c).clauses;
    p = KB(c).prob;
    len_clause = length(clause);
    if len_clause<4
        len_KB3PSAT = len_KB3PSAT + 1;
        KB3PSAT(len_KB3PSAT).clauses = clause;
        KB3PSAT(len_KB3PSAT).prob = p;
    else
        num_new_clauses = len_clause - 2;
        len_KB3PSAT = len_KB3PSAT + 1;
        new_vars = [num_vars+1:num_vars+(len_clause-3)];
        num_new_vars = length(new_vars);
        num_times = len_clause - 4;
        pp = BR_3PSAT(KB(c));
        KB3PSAT(len_KB3PSAT).clauses = [clause(1:2),new_vars(1)];
        KB3PSAT(len_KB3PSAT).prob = pp;
        for cc = 1:num_times
            len_KB3PSAT = len_KB3PSAT + 1;
            KB3PSAT(len_KB3PSAT).clauses = [clause(cc+2),...
```

```
                -new_vars(cc),new_vars(cc+1)];
            KB3PSAT(len_KB3PSAT).prob = pp;
        end
        len_KB3PSAT = len_KB3PSAT + 1;
        KB3PSAT(len_KB3PSAT).clauses =
        [clause(end-1:end),-new_vars(end)];
        KB3PSAT(len_KB3PSAT).prob = pp;
        num_vars = num_vars + num_new_vars;
    end
end

%-------------------------------------------------------------------------
function vars = BR_vars(KB,sentence)
% BR_vars - find list of variables in logical sentences
% On input:
%     KB (nx1 conjunctive normal form vector): conjunctive clauses
%       (i).clauses (1xm vector): disjunctive clause
%     sentence (1x1 conjunctive normal form vector): conjunctive
%     clause
%       (1).clauses (1xk vector): disjunctive clause
% On output:
%     vars (1xp vector): list of variables in KB and sentence
% Call:
%     vars = BR_vars(KB,thm);
% Author:
%     T. Henderson
%     UU
%     Spring 2017
%

vars = [];

for s = 1:length(KB)
    vars = unique([vars,abs(KB(s).clauses)]);
end
for s = 1:length(sentence)
    vars = unique([vars,abs(sentence(s).clauses)]);
end
vars = sort(vars);
```

BR_all_and_args.m

```
function args = BR_all_and_args(s)
% BR_all_and_args - get all sub-clauses for and clause
% On input:
%     s (string): logical sentence (fully parenthesize)
% On output:
%     args (struct vector): args of s
```

```
%     (k).s (string): k_th clause in s
% Call:
%     args = BR_all_and_args('((1)^(2))');
% Author:
%     T. Henderson
%     UU
%     Fall 2018
%

args = [];
num_args = 0;

if isempty(s)
    return
end

num_clauses = 1;
clauses(1).s = s;
queue = [1];

while ~isempty(queue)
    p = queue(1);
    queue = queue(2:end);
    clause = clauses(p).s;
    if BR_isANDop(clause)
        [OP1,OP2] = BR_getANDargs(clause);
        num_clauses = num_clauses + 1;
        clauses(num_clauses).s = OP1;
        queue = [queue; num_clauses];
        num_clauses = num_clauses + 1;
        clauses(num_clauses).s = OP2;
        queue = [queue; num_clauses];
    else
        num_args = num_args + 1;
        args(num_args).s = clause(2:end-1);
    end
end
```

BR_create_vtable.m

```
function vtable = BR_create_vtable(n)
% BR_create_vtable - create a variable table for general KBs
% On input:
%     n (int): number of variables
% On output:
%     vtable (nx7 array): variable characterization
%      col 1: literal 1
%      col 2: literal 2
```

```
%       col 3: literal 3
%       col 4: clause number giving rise to variable
%       col 5: parity of clause
%          for 2-literal clause: 0, 1, 2, 3
%          for 3-literal clause: 0, 2, 3, 7
%       col 6: variable number in equations
%       col 7: operator for 3-literal clause
%         1: P(A|B^C), i.e., 'and'; 2: P(A|BvC), i.e., 'or'
% Call:
%     vtable = BR_create_vtable(6);
% Author:
%     T. Henderson
%     UU
%     Summer 2018
%

vtable = zeros(n,7);
vtable(:,1) = [1:n]';
vtable(:,6) = [1:n]';
vtable(:,5) = ones(n,1);
```

BR_dist_or.m

```
function s = BR_dist_or(s_in)
% BR_dist_or - distributes v in logical expression (exponential time)
% On input:
%     s_in (string): logical string
% On output:
%     s (string): s_in with (a^b)vc replaced with (avc)^(bvc)
% Call:
%     s1 = BR_dist_or(s)
% Author:
%     T. Henderson
%     UU
%     Fall 2018
%

OR_OP = 'v';
AND_OP = '^';
LEFT_PAR = '(';
RIGHT_PAR = ')';

s = s_in;
index = 0;
change = 1;

while change==1
    len_s = length(s);
```

```
if index>=len_s
    change = 0;
end
index = index + 1;
while index<=len_s&s(index)~=OR_OP
    index = index + 1;
end
if index<len_s
    count = 1;
    index1 = index - 1;
    while count>0
        index1 = index1 - 1;
        if s(index1)==LEFT_PAR
            count = count - 1;
        elseif s(index1)==RIGHT_PAR
            count = count + 1;
        end
    end
    S1 = s(index1:index-1);
    count = 1;
    index2 = index + 1;
    while count>0
        index2 = index2 + 1;
        if s(index2)==LEFT_PAR
            count = count + 1;
        elseif s(index2)==RIGHT_PAR
            count = count - 1;
        end
    end
    S2 = s(index+1:index2);
    if BR_isANDop(S1)
        [S11,S12] = BR_getANDargs(S1);
        prefix = s(1:index1-1);
        postfix = s(index2+1:end);
        middle = ['(',S11,OR_OP,S2,')^(',S12,OR_OP,S2,')'];
        s = [prefix,middle,postfix];
        change = 1;
        index = 0;
    elseif BR_isANDop(S2)
        [S21,S22] = BR_getANDargs(S2);
        prefix = s(1:index1-1);
        postfix = s(index2+1:end);
        middle =
        [LEFT_PAR,S1,OR_OP,S21,')^(',S1,OR_OP,S22,RIGHT_PAR];
        s = [prefix,middle,postfix];
        change = 1;
        index = 0;
    end
end
```

```
end
```

BR_gen_test_KB_gen.m

```
function [KB,cc_probs] = BR_gen_test_KB_gen(n,max_sentences,...
    max_sentence_len)
% BR_gen_test_KB_gen - create a random KB with correct probabilities
% On input:
%     n (int): number of atoms
%     max_sentences (int): maximum number of sentences
%     max_sentence_length (int): maximum length of any sentence
% On output:
%     KB (KB data structure): KB
%     cc_probs (1x2^n vector): complete conjunction probabilities
% Call:
%     [KB2,cc_probs2] = BR_gen_test_KB_gen(3,6,3);
% Author:
%     T. Henderson
%     UU
%     Fall 2017
%

MAX_FAILS = 100;

m = randi(max_sentences);
cc_probs = [];
max_sentence_len = min(n,max_sentence_len);
OK = 0;
count = 0;
thm1(1).clauses = [1];
thm2(1).clauses = [-1];
while OK==0
    OK = 1;
    KB = [];
    num_sentences = 0;
    while num_sentences<m
        clause_length = randi(max_sentence_len);
        clause = randperm(n);
        clause = clause(1:clause_length);
        lit_sign = rand(1,clause_length)<0.5;
        lit_sign = 3.^lit_sign - 2;
        clause = clause.*lit_sign;
        clause = sort(unique(clause));
        KB = BR_KB_insert_clause(KB,clause);
        num_sentences = length(KB);
    end
    B1 = CS4300_Ask(KB,thm1);
    B2 = CS4300_Ask(KB,thm2);
```

```
    if isempty(KB)|(B1*B2==1)|(max(BR_vars(KB,[]))<n)
        count = count + 1;
%        ['Failed: ',num2str(count)]
        OK = 0;
    end
end
%display('Success');
num_cc = 2^n;
cc_probs = rand(1,num_cc);
cc_probs = cc_probs/sum(cc_probs);
for s = 1:num_sentences
    KB(s).prob = BR_cc2clauseprob(KB(s).clauses,cc_probs);
end

%-------------------------------------------------------------------------
function p = BR_cc2clauseprob(clause,cc_probs)
% BR_cc2clauseprob - compute clause probability from cc probs
% On input:
%     clause (1xk vector): disjunctive clause
%     cc_probs (1x2^n vector): cc probabilities
% On output:
%     p (float): probability of clause
% Author:
%     T. Henderson
%     UU
%     Fall 2017
%

p = 0;

len_clause = length(clause);
clause_sign = zeros(1,len_clause);
for d = 1:len_clause
    clause_sign(d) = clause(d)>0;
end
num_cc = length(cc_probs);
n = log2(num_cc);
indexes = sort(abs(clause));

for b = 0:num_cc-1
    bits = BR_int2bits(b,n);
    cc_sign = bits(indexes);
    if max(cc_sign==clause_sign)==1
        p = p + cc_probs(b+1);
    end
end

%-------------------------------------------------------------------------
function KB_out = BR_KB_insert_clause(KB,clause)
```

```
% BR_KB_insert_clause - insert a new clause into KB
% On input:
%     KB (KB data structure): knowledge base of disjunctions
%     clause (1xn vector): clause of literals
% On output:
%     KB_out (KB data structure): KB with new clause
% Call:
%     KB1 = BR_KB_insert_clause(KB,[2]);
% Author:
%     T. Henderson
%     UU
%     Fall 2017
%

KB_out = KB;

num_clauses = length(KB);
clause = sort(unique(clause));
len_clause = length(clause);
for s = 1:num_clauses
    KB_clause = KB(s).clauses;
    if (length(KB_clause)==len_clause)&(min(KB_clause==clause)==1)
        return
    end
end
num_clauses = num_clauses + 1;
KB_out(num_clauses).clauses = clause;
KB_out(num_clauses).prob = 0;

%-----------------------------------------------------------------------
function b = CS4300_Ask(KB,sentence)
% CS4300_Ask - Ask function for logic KB
% On input:
%     KB (KB struct): Knowledge base (CNF)
%       (k).clauses (1xp vector): disjunction clause
%     sentence (KB struct): query theorem (CNF)
%       (k).clauses (1xq vector): disjunction
% On output:
%     b (Boolean): 1 if KB entails sentence, else 0
% Call:
%     KB(1).clauses = [1];
%     KB(2).clauses = [-1,2];
%     sentence(1).clauses = [2];
%     b = CS4300_Ask(KB,sentence);
% Author:
%     T. Henderson
%     UU
%     Fall 2017
%
```

```
b = 0;

if isempty(sentence)
    return
end

vars = CS4300_vars(KB,sentence);
num_sentences = length(KB);
len_sentence = length(sentence);
for s = 1:len_sentence
    KB(num_sentences+1).clauses = -sentence(s).clauses;
    CS4300_create_SAT_prob(KB,'HYBKB');
    system('sat.py < HYBKB >popo');
    fd = fopen('popo','r');
    t = fscanf(fd,'%s');
    if ~isempty(t)
        return
    end
    clear t
    fclose(fd);
end

b = 1;

%-------------------------------------------------------------------------
function CS4300_create_SAT_prob(KB,fn)
% CS4300_create_SAT_prob - setup file to call Python SAT solver
% On input:
%     KB (KB data struct): CNF KB
%     fn (string): filename for SAT problem
% On output:
%   N/A (writes a file with name fn)
% Call:
%     CS4300_create_SAT_prob(KB,'HYBKB');
% Author:
%     T. Henderson
%     UU
%     Fall 2017
%

MINUS = '~';
BLANK = ' ';
x = 'x';

fd = fopen(fn,'w');
num_clauses = length(KB);

for c = 1:num_clauses
```

```
    clause = KB(c).clauses;
    len_clause = length(clause);
    for d = 1:len_clause
        if clause(d)<0
            fprintf(fd,'%s',MINUS);
        end
        fprintf(fd,'%s',x);
        fprintf(fd,'%d',abs(clause(d)));
        fprintf(fd,'%s',BLANK);
    end
    fprintf(fd,'\n');
end
fclose(fd);

%-------------------------------------------------------------------------
function vars = CS4300_vars(KB,sentence)
% CS4300_vars - determine variable set of KB
% On input:
%     KB (KB struct): CNF KB
%     sentence (1xk vector): disjunction
% On output:
%     vars (1xn vector): atom variable values
% Call:
%     v = CS4300_vars(KB,[1]);
% Author:
%     T. Henderson
%     UU
%     Fall 2017
%

vars = [];

for s = 1:length(KB)
    vars = unique([vars,abs(KB(s).clauses)]);
end
for s = 1:length(sentence)
    vars = unique([vars,abs(sentence(s).clauses)]);
end
vars = sort(vars);

%-------------------------------------------------------------------------
function vars = BR_vars(KB,sentence)
% BR_vars - find list of variables in logical sentences
% On input:
%     KB (nx1 conjunctive normal form vector): conjunctive clauses
%       (i).clauses (1xm vector): disjunctive clause
%     sentence (1x1 conjunctive normal form vector): conjunctive
%     clause
%       (1).clauses (1xk vector): disjunctive clause
```

```
% On output:
%      vars (1xp vector): list of variables in KB and sentence
% Call:
%      vars = BR_vars(KB,thm);
% Author:
%      T. Henderson
%      UU
%      Spring 2017
%

vars = [];

for s = 1:length(KB)
    vars = unique([vars,abs(KB(s).clauses)]);
end
for s = 1:length(sentence)
    vars = unique([vars,abs(sentence(s).clauses)]);
end
vars = sort(vars);

%-------------------------------------------------------------------------
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);
```

BR_gen_test_KB_ind.m

```
function [KB,cc_probs,ap] = BR_gen_test_KB_ind(n,max_sentences,...
```

```
    max_sentence_len)
% BR_gen_test_KB_ind - create a random KB with correct probabilities
%   where variables are independent
% On input:
%     n (int): number of atoms
%     max_sentences (int): maximum number of sentences
%     max_sentence_length (int): maximum length of any sentence
% On output:
%     KB (KB data structure): KB
%     cc_probs (1x2^n vector): complete conjunction probabilities
%     ap (1xn vector): atom probabilities
% Call:
%     [KB2,cc_probs2] = BR_gen_test_KB_ind(3,6,3);
% Author:
%     T. Henderson
%     UU
%     Fall 2017
%

MAX_FAILS = 100;

KB = [];
cc_probs = [];
ap = [];

m = randi(max_sentences);
max_sentence_len = min(n,max_sentence_len);
OK = 0;
count = 0;
thm1(1).clauses = [1];
thm2(1).clauses = [-1];
while OK==0
    OK = 1;
    KB = [];
    num_sentences = 0;
    while num_sentences<m
        clause_length = randi(max_sentence_len);
        clause = randperm(n);
        clause = clause(1:clause_length);
        lit_sign = rand(1,clause_length)<0.5;
        lit_sign = 3.^lit_sign - 2;
        clause = clause.*lit_sign;
        clause = sort(unique(clause));
        KB = BR_KB_insert_clause(KB,clause);
        num_sentences = length(KB);
    end
    B1 = CS4300_Ask(KB,thm1);
    B2 = CS4300_Ask(KB,thm2);
    if ~isempty(KB)&(B1*B2==1)
```

```
        count = count + 1;
        if count>MAX_FAILS
            return
        end
        OK = 0;
    end
end

ap = rand(1,n);
cc_probs = BR_ap2ccp(ap);
for s = 1:num_sentences
    KB(s).prob = BR_prob_or_atom_probs(KB(s).clauses,ap);
end

%-------------------------------------------------------------------------
function KB_out = BR_KB_insert_clause(KB,clause)
% BR_KB_insert_clause - insert a new clause into KB
% On input:
%     KB (KB data structure): knowledge base of disjunctions
%     clause (1xn vector): clause of literals
% On output:
%     KB_out (KB data structure): KB with new clause
% Call:
%     KB1 = BR_KB_insert_clause(KB,[2]);
% Author:
%     T. Henderson
%     UU
%     Fall 2017
%

KB_out = KB;

num_clauses = length(KB);
clause = sort(unique(clause));
len_clause = length(clause);
for s = 1:num_clauses
    KB_clause = KB(s).clauses;
    if (length(KB_clause)==len_clause)&(min(KB_clause==clause)==1)
        return
    end
end
num_clauses = num_clauses + 1;
KB_out(num_clauses).clauses = clause;
KB_out(num_clauses).prob = 0;

%-------------------------------------------------------------------------
function b = CS4300_Ask(KB,sentence)
% CS4300_Ask - Ask function for logic KB
% On input:
```

```
%     KB (KB struct): Knowledge base (CNF)
%       (k).clauses (1xp vector): disjunction clause
%     sentence (KB struct): query theorem (CNF)
%       (k).clauses (1xq vector): disjunction
% On output:
%     b (Boolean): 1 if KB entails sentence, else 0
% Call:
%     KB(1).clauses = [1];
%     KB(2).clauses = [-1,2];
%     sentence(1).clauses = [2];
%     b = CS4300_Ask(KB,sentence);
% Author:
%     T. Henderson
%     UU
%     Fall 2017
%

b = 0;

if isempty(sentence)
    return
end

vars = CS4300_vars(KB,sentence);
num_sentences = length(KB);
len_sentence = length(sentence);
for s = 1:len_sentence
    KB(num_sentences+1).clauses = -sentence(s).clauses;
    CS4300_create_SAT_prob(KB,'HYBKB');
    system('sat.py < HYBKB >popo');
    fd = fopen('popo','r');
    t = fscanf(fd,'%s');
    if ~isempty(t)
        return
    end
    clear t
    fclose(fd);
end

b = 1;

%-----------------------------------------------------------------
function CS4300_create_SAT_prob(KB,fn)
% CS4300_create_SAT_prob - setup file to call Python SAT solver
% On input:
%     KB (KB data struct): CNF KB
%     fn (string): filename for SAT problem
% On output:
%   N/A (writes a file with name fn)
```

```
% Call:
%     CS4300_create_SAT_prob(KB,'HYBKB');
% Author:
%     T. Henderson
%     UU
%     Fall 2017
%

MINUS = '~';
BLANK = ' ';
x = 'x';

fd = fopen(fn,'w');
num_clauses = length(KB);

for c = 1:num_clauses
    clause = KB(c).clauses;
    len_clause = length(clause);
    for d = 1:len_clause
        if clause(d)<0
            fprintf(fd,'%s',MINUS);
        end
        fprintf(fd,'%s',x);
        fprintf(fd,'%d',abs(clause(d)));
        fprintf(fd,'%s',BLANK);
    end
    fprintf(fd,'\n');
end
fclose(fd);

%-------------------------------------------------------------------------
function vars = CS4300_vars(KB,sentence)
% CS4300_vars - determine variable set of KB
% On input:
%     KB (KB struct): CNF KB
%     sentence (1xk vector): disjunction
% On output:
%     vars (1xn vector): atom variable values
% Call:
%     v = CS4300_vars(KB,[1]);
% Author:
%     T. Henderson
%     UU
%     Fall 2017
%

vars = [];

for s = 1:length(KB)
```

```
    vars = unique([vars,abs(KB(s).clauses)]);
end
for s = 1:length(sentence)
    vars = unique([vars,abs(sentence(s).clauses)]);
end
vars = sort(vars);

%-------------------------------------------------------------------------
function prob = BR_prob_or_atom_probs(clause,A_probs)
% BR_prob_or_atom_probs -= compute probability of or clause from atom
% probs
% On input:
%     clause (1xk vector): disjunction of literals
%     A_probs (1xn vector): atom probabilities
% On output:
%     prob (float): probability of clause
% Call:
%     p = BR_prob_or_atom_probs(clause,A_probs);
% Author:
%     T. Henderson
%     UU
%     Spring 2017
%

prob = 0;
if isempty(clause)
    return
end
len_clause = length(clause);
if len_clause==1
    if clause(1)<0
        prob = 1 - A_probs(abs(clause(1)));
    else
        prob = A_probs(abs(clause(1)));
    end
    return
end
L_probs = A_probs;
for e = 1:len_clause
    if clause(e)<0
        L_probs(abs(clause(e))) = 1 - A_probs(abs(clause(e)));
    end
end
p1 = L_probs(abs(clause(1)));
p2 = BR_prob_or(clause(2:end),L_probs);
prob = p1 + p2 - p1*p2;

%-------------------------------------------------------------------------
function prob = BR_prob_or(clause,L_probs)
```

```
% BR_prob_or - compute probability of or clause
% On input:
%     clause (1xk vector): disjuction of literals
%     L_probs (1xn vector): probabilities of literals
% On output:
%     prob (float): probability of disjunction
% Call:
%     p = BR_prob_or(clause,L_probs);
% Author:
%     T. Henderson
%     UU
%     Spring 2017
%

prob = 0;
if isempty(clause)
    return
end
len_clause = length(clause);
if len_clause==1
    prob = L_probs(abs(clause(1)));
    return
end
p1 = L_probs(abs(clause(1)));
p2 = BR_prob_or(clause(2:end),L_probs);
prob = p1 + p2 - p1*p2;

%-----------------------------------------------------------------------
function cc_probs = BR_ap2ccp(ap)
% BR_ap2ccp - compute complete conjunction probs from atom probs
% On input:
%     ap (1xn vector): atom probabilities
% On output:
%     cc_probs (1x2^n vector): complete conjunction probabilities
% Call:
%     cc = BR_ap2ccp(ap);
% Author:
%     T. Henderson
%     UU
%     Fall 2017
%

n = length(ap);
num_cc = 2^n;
cc_probs = zeros(1,num_cc);

for w = 0:num_cc-1
    bits = BR_int2bits(w,n);
    p = 1;
```

```
    for e = 1:n
        if bits(e)==0
            p = p*(1-ap(e));
        else
            p = p*ap(e);
        end
    end
    cc_probs(w+1) = p;
end

%-------------------------------------------------------------------------
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);
```

BR_getANDargs.m

```
function [S1,S2] = BR_getANDargs(s)
% BR_getANDargs - returns the arguments to top-level ^ operator in s
% On input:
%     s (string): logical string
% On output:
%     S1 (string): first arg of (S1^S2)
%     S2 (string): second arg of (S1^S2)
% Call:
%     [S1,S2] = BR_getANDargs(s)
% Author:
%     T. Henderson
```

```
%     UU
%     Fall 2018
%

LEFT_PAREN = '(';
RIGHT_PAREN = ')';
AND_OP = '^';

S1 = [];
S2 = [];

len_s = length(s);
count = 1;
for c = 2:len_s
    if s(c)==LEFT_PAREN
        count = count + 1;
    elseif s(c)==RIGHT_PAREN
        count = count - 1;
    end
    if count==1&s(c+1)==AND_OP
        S1 = s(2:c);
        S2 = s(c+2:len_s-1);
        return
    end
end
```

BR_getEQargs.m

```
function [S1,S2] = BR_getEQargs(s)
% BR_getEQargs - returns the arguments to top-level = operator in s
% On input:
%     s (string): logical string
% On output:
%     S1 (string): first arg of (S1=S2)
%     S2 (string): second arg of (S1=S2)
% Call:
%     [S1,S2] = BR_getEQargs(s)
% Author:
%     T. Henderson
%     UU
%     Fall 2018
%

LEFT_PAREN = '(';
RIGHT_PAREN = ')';
EQ_OP = '=';

S1 = [];
```

```
S2 = [];

len_s = length(s);
count = 1;
for c = 2:len_s
    if s(c)==LEFT_PAREN
        count = count + 1;
    elseif s(c)==RIGHT_PAREN
        count = count - 1;
    end
    if count==1&s(c+1)==EQ_OP
        S1 = s(2:c);
        S2 = s(c+2:len_s-1);
        return
    end
end
```

BR_getIMPLIESargs.m

```
function [S1,S2] = BR_getIMPLIESargs(s)
% BR_getIMPLIESargs - returns the arguments to top-level > operator in
% s
% On input:
%     s (string): logical string
% On output:
%     S1 (string): first arg of (S1>S2)
%     S2 (string): second arg of (S1>S2)
% Call:
%     [S1,S2] = BR_getIMPLIESargs(s)
% Author:
%     T. Henderson
%     UU
%     Fall 2018
%

LEFT_PAREN = '(';
RIGHT_PAREN = ')';
IMPLIES_OP = '>';

S1 = [];
S2 = [];

len_s = length(s);
count = 1;
for c = 2:len_s
    if s(c)==LEFT_PAREN
        count = count + 1;
    elseif s(c)==RIGHT_PAREN
```

```
        count = count - 1;
    end
    if count==1&s(c+1)==IMPLIES_OP
        S1 = s(2:c);
        S2 = s(c+2:len_s-1);
        return
    end
end
```

BR_getNEGargs.m

```
function [S1,S2,next_op] = BR_getNEGargs(s)
% BR_getNEGargs - returns the arguments to top-level ~ operator in s
% On input:
%     s (string): logical string
% On output:
%     S1 (string): first arg of (S1=S2)
%     S2 (string): second arg of (S1=S2)
% Call:
%     [S1,S2] = BR_getNEGargs(s)
% Author:
%     T. Henderson
%     UU
%     Fall 2018
%

NEG_OP = '~';
AND_OP = '^';
OR_OP = 'v';

S1 = [];
S2 = [];
next_op = [];
sp = s(3:end-1);

if BR_isNEGop(sp)
    S1 = sp;
    next_op = NEG_OP;
    return
end

if BR_isANDop(sp)
    [S1,S2] = BR_getANDargs(sp);
    next_op = AND_OP;
    return
end

if BR_isORop(sp)
```

```
    [S1,S2] = BR_getORargs(sp);
    next_op = OR_OP;
    return
end
```

BR_getORargs.m

```
function [S1,S2] = BR_getORargs(s)
% BR_getORargs - returns the arguments to top-level v operator in s
% On input:
%     s (string): logical string
% On output:
%     S1 (string): first arg of (S1vS2)
%     S2 (string): second arg of (S1vS2)
% Call:
%     [S1,S2] = BR_getORargs(s)
% Author:
%     T. Henderson
%     UU
%     Fall 2018
%

LEFT_PAREN = '(';
RIGHT_PAREN = ')';
OR_OP = 'v';

S1 = [];
S2 = [];

len_s = length(s);
count = 1;
for c = 2:len_s
    if s(c)==LEFT_PAREN
        count = count + 1;
    elseif s(c)==RIGHT_PAREN
        count = count - 1;
    end
    if count==1&s(c+1)==OR_OP
        S1 = s(2:c);
        S2 = s(c+2:len_s-1);
        return
    end
end
```

BR_get_literals.m

```
function literals = BR_get_literals(s)
% BR_get_literals - returns vector of literals in s
% On input:
%     s (string): fully parenthesized logical sentence
% On output:
%     literals (vector): literal values (signed) in s
% Call:
%     b = BR_get_literals('((1)v(2))');
% Author:
%     T. Henderson
%     UU
%     Fall 2018
%

LEFT_PAREN = '(';
MINUS = '-';

literals = [];

if isempty(s)
    return
end

len_s = length(s);
index = 0;
while index<len_s
    index = index + 1;
    if (s(index)==LEFT_PAREN&BR_isdigit(s(index+1)))
        index = index + 1;
        v = str2num(s(index));
        index = index + 1;
        while BR_isdigit(s(index))
            v = 10*v + str2num(s(index));
        end
        literals = [literals,v];
    elseif (s(index)==LEFT_PAREN&s(index+1)==MINUS)
        index = index + 2;
        v = str2num(s(index));
        index = index + 1;
        while BR_isdigit(s(index))
            v = 10*v + str2num(s(index));
        end
        literals = [literals,-v];
    end
end
```

BR_get_vtable.m

```
function indexes = BR_get_vtable(vtable,L1,L2,L3,c,parity,op)
% BR_get_table - find entries in vtable matching query
% On input:
%     vtable (kx7 array): variable table
%     L1 (int): if 0, no query, else literal value
%     L2 (int): if 0, no query, else literal value
%     L3 (int): if 0, no query, else literal value
%     c (int): if <1, no query, else clause number
%     parity (int): if <0, no query, else clause number
%     op (int): if <0, no query, else operator value
% On output:
%     indexes (px1 vector): indexes of matching entries
% Call:
%     BR_get_vtable(vtable,1,2,3,-1,-1,-1)
% Author:
%     T. Henderson
%     UU
%     Summer 2018
%

all_indexes = vtable(:,6);
if L1~=0
    indexes = intersect(all_indexes,find(vtable(:,1)==L1));
end
if L2~=0
    indexes = intersect(indexes,find(vtable(:,2)==L2));
end
if L3~=0
    indexes = intersect(indexes,find(vtable(:,3)==L3));
end
if c>0
    indexes = intersect(indexes,find(vtable(:,4)==c));
end
if parity>=0
    indexes = intersect(indexes,find(vtable(:,5)==parity));
end
if op>=0
    indexes = intersect(indexes,find(vtable(:,7)==op));
end
```

BR_int2bits.m

```
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);
```

BR_isANDop.m

```
function b = BR_isANDop(s)
% BR_isANDop - determines if s is an AND statement
% On input:
%     s (string): logical string
% On output:
%     b (Boolean): true if AND statement, else false
% Call:
%     b = BR_isANDop(s);
% Author:
%     T. Henderson
%     UU
%     Fall 2018
%

LEFT_PAREN = '(';
RIGHT_PAREN = ')';
AND_OP = '^';

b = 0;

len_s = length(s);
count = 1;
for c = 2:len_s
    if s(c)==LEFT_PAREN
        count = count + 1;
    elseif s(c)==RIGHT_PAREN
        count = count - 1;
    end
```

```
    if count==1&s(c+1)==AND_OP
        b = 1;
        return
    end
end
```

BR_isEQop.m

```
function b = BR_isEQop(s)
% BR_isEQop - determines if s is an = statement
% On input:
%     s (string): logical string
% On output:
%     b (Boolean): true if = statement, else false
% Call:
%     b = BR_isEQop(s);
% Author:
%     T. Henderson
%     UU
%     Fall 2018
%

LEFT_PAREN = '(';
RIGHT_PAREN = ')';
EQ_OP = '=';

b = 0;

len_s = length(s);
count = 1;
for c = 2:len_s
    if s(c)==LEFT_PAREN
        count = count + 1;
    elseif s(c)==RIGHT_PAREN
        count = count - 1;
    end
    if count==1&s(c+1)==EQ_OP
        b = 1;
        return
    end
end
```

BR_isNEGop.m

```
function b = BR_isNEGop(s)
% BR_isNEGop - determines if s is an NOT statement
```

```
% On input:
%     s (string): logical string
% On output:
%     b (Boolean): true if NOT statement, else false
% Call:
%     b = BR_isNOTop(s);
% Author:
%     T. Henderson
%     UU
%     Fall 2018
%

b = 0;

NEG_OP = '~';

if s(2)==NEG_OP
    b = 1;
end
```

BR_isORop.m

```
function b = BR_isORop(s)
% BR_isORop - determines if s is an OR statement
% On input:
%     s (string): logical string
% On output:
%     b (Boolean): true if OR statement, else false
% Call:
%     b = BR_isORop(s);
% Author:
%     T. Henderson
%     UU
%     Fall 2018
%

LEFT_PAREN = '(';
RIGHT_PAREN = ')';
OR_OP = 'v';

b = 0;

len_s = length(s);
count = 1;
for c = 2:len_s
    if s(c)==LEFT_PAREN
        count = count + 1;
    elseif s(c)==RIGHT_PAREN
```

```
        count = count - 1;
    end
    if count==1&s(c+1)==OR_OP
        b = 1;
        return
    end
end
```

BR_isdigit.m

```
function b = BR_isdigit(c)
% BR_isdigit - determines if c is a digit
% On input:
%     c (character): character from logical sentence
% On output:
%     b (Boolean): true if c is digit, else false
% Call:
%     b = BR_isdigit('4');
% Author:
%     T. Henderson
%     UU
%     Fall 2018
%

b = 0;
if c=='0'
    b = 1;
    return
end
if c=='1'
    b = 1;
    return
end
if c=='2'
    b = 1;
    return
end
if c=='3'
    b = 1;
    return
end
if c=='4'
    b = 1;
    return
end
if c=='5'
    b = 1;
    return
```

```
end
if c=='6'
    b = 1;
    return
end
if c=='7'
    b = 1;
    return
end
if c=='8'
    b = 1;
    return
end
if c=='9'
    b = 1;
    return
end
```

BR_isliteral.m

```
function b = BR_isliteral(s)
% BR_isliteral - determines if s is a literal: (<int>)/(-<int>)
% On input:
%     s (string): logical string
% On output:
%     b (Boolean): true if literal, else false
% Call:
%     b = BR_isliteral(s);
% Author:
%     T. Henderson
%     UU
%     Fall 2018
%

LEFT_PAREN = '(';
RIGHT_PAREN = ')';
MINUS = '-';

trans = [2,5,5,5,5; 5,3,2,4,5; 5,5,3,4,5];

b = 0;

state = 1;
if s(1)==LEFT_PAREN
    c = 1;
elseif s(1)==MINUS
    c = 2;
elseif BR_isdigit(s(1))
```

```
    c = 3;
elseif s(1)==RIGHT_PAREN
    c = 4;
else
    c = 5;
end

if c~=1
    return
end

index = 1;
state = 2;
while state<4
    index = index + 1;
    if s(index)==LEFT_PAREN
        c = 1;
    elseif s(index)==MINUS
        c = 2;
    elseif BR_isdigit(s(index))
        c = 3;
    elseif s(index)==RIGHT_PAREN
        c = 4;
    else
        c = 5;
    end
    state = trans(state,c);
end
if state==4&index==length(s)
    b = 1;
end
```

BR_isvariable.m

```
function b = BR_isvariable(s)
% BR_isvariable - determines if s is a variable statement: (<int>)
% On input:
%     s (string): logical string
% On output:
%     b (Boolean): true if variable statement, else false
% Call:
%     b = BR_isvariable(s);
% Author:
%     T. Henderson
%     UU
%     Fall 2018
%
```

```
LEFT_PAREN = '(';
RIGHT_PAREN = ')';
b = 1;

if s(1)==LEFT_PAREN&s(end)==RIGHT_PAREN
    len_s = length(s);
    for c = 2:len_s-1
        if ~BR_isdigit(s(c))
            b = 0;
            return
        end
    end
end
```

BR_max_var_in_s.m

```
function max_val = BR_max_var_in_s(s)
% BR_max_var_in_s - determines maximum variable value in s
% On input:
%     s (string): fully parenthesized logical sentence
% On output:
%     max_val (int): max value of abs of literals in s
% Call:
%     b = BR_max_var_in_s('((1)v(2))');
% Author:
%     T. Henderson
%     UU
%     Fall 2018
%

if isempty(s)
    max_val = 0;
    return
end

d = isstrprop(s,'digit');

d_len = length(d);

max_val = 0;
k = 0;
while k<d_len
    k = k + 1;
    if d(k)==1
        k1 = k + 1;
        val = str2num(s(k));
        while k1<d_len&d(k1)==1
            val = val*10 + str2num(s(k1));
```

```
            k1 = k1 + 1;
        end
        if val>max_val
            max_val = val;
        end
    end
end
```

BR_move_not_in.m

```
function s = BR_move_not_in(s_in)
% BR_move_not_in - moves not to atoms in logical expression
% On input:
%     s_in (string): logical string
% On output:
%     s (string): s_in with ~ moved all the way to -<int>
% Call:
%     s1 = BR_move_not_in(s)
% Author:
%     T. Henderson
%     UU
%     Fall 2018
%

NEG = '~';
LEFT_PAR = '(';
RIGHT_PAR = ')';

s = s_in;
change = 1;
index = 0;

while change==1
    len_s = length(s);
    change = 0;
    index = index + 1;
    while index<=len_s & s(index)~=NEG
        index = index + 1;
    end
    if index<len_s
        prefix = s(1:index-1);
        count = 1;
        n_index = index + 2;
        while n_index<len_s&count>0
            if s(n_index)==LEFT_PAR
                count = count + 1;
            elseif s(n_index)==RIGHT_PAR
                count = count - 1;
```

```
            end
            n_index = n_index + 1;
        end
        n_index = n_index + 1;
        postfix = s(n_index-1:end);
        SN = s(index+1:n_index-2);
        if BR_isvariable(SN)
            s = [prefix,num2str(-str2num(SN(2:end-1))),postfix];
            change = 1;
            index = 0;
        elseif BR_isNEGop(SN)
            s = [prefix,s(index+4:n_index-4),postfix];
            change = 1;
            index = 0;
        elseif BR_isANDop(SN)
            [S1,S2] = BR_getANDargs(SN);
            s = [prefix,'(~',S1,')v(~',S2,')',postfix];
            change = 1;
            index = 0;
        elseif BR_isORop(SN)
            [S1,S2] = BR_getORargs(SN);
            s = [prefix,'(~',S1,')^(~',S2,')',postfix];
            change = 1;
            index = 0;
        end
    end
end
```

BR_Nilsson.m

```
function P_cc = BR_Nilsson(KB)
% BR_Nilsson - Nilsson's PSAT solution
% On input:
%     KB (KB struct): KB clauses
% On output:
%     P_cc (kx1 vector): probabilities of complete conjunctions
% Call:
%     KB(1).clauses = [1];
%     KB(1).prob = 0.7;
%     KB(2).clauses = [-1,2];
%     KB(2).prob = 0.7;
%     P = BR_Nilsson(KB);
% Author:
%     T. Henderson
%     UU
%     Fall 2018
%
```

```
num_sentences = length(KB);
vars = BR_vars(KB,[]);
num_vars = length(vars);
num_cc = 2^num_vars;
V = zeros(num_sentences+1,num_cc);
P_s = zeros(num_sentences+1,1);
V(end,:) = ones(1,num_cc);
P_s(end) = 1;

for s = 1:num_sentences
    P_s(s) = KB(s).prob;
    for cc = 0:num_cc-1
        bits = BR_int2bits(cc,num_vars);
        V(s,cc+1) = BR_sat(KB(s).clauses,bits);
    end
end

P_cc = lsqlin(V,P_s,[],[],[],[],zeros(num_cc,1),ones(num_cc,1));
```

BR_Nilsson_query.m

```
function p = BR_Nilsson_query(query,cc_probs)
% BR_Nilsson_query - find probability of sentence using probabilities
% of
%     complete conjunction set
% On input:
%     query (KB structure): query
%     cc_probs (kx1 vector): complete conjunction probabilities
% On output:
%     p (float): probability of query
% Call:
%     p = BR_Nilsson_query(query,probs);
% Author:
%     T. Henderson
%     UU
%     Fall 2018
%

num_cc = length(cc_probs);
num_vars = log2(num_cc);
query_sat = ones(1,num_cc);

for cc = 0:num_cc-1
    bits = BR_int2bits(cc,num_vars);
    query_sat(cc+1) = query_sat(cc+1)&BR_sat(query.clauses,bits);
end

p = query_sat*cc_probs;
```

BR_only_AND.m

```
function b = BR_only_AND(s)
% BR_only_AND - determines if only ^ operator in string
% On input:
%     s (string): fully parenthesized logical sentence
% On output:
%     b (Boolean): 0 if v in s, else 1
% Call:
%     b = BR_only_AND('((1)v(2))');
% Author:
%     T. Henderson
%     UU
%     Fall 2018
%

NEG_OP = '~';
OR_OP = 'v';

b = 1;

len_s = length(s);

for c = 1:len_s
    if s(c)==NEG_OP|s(c)==OR_OP
        b = 0;
        return
    end
end
```

BR_only_OR.m

```
function b = BR_only_OR(s)
% BR_only_OR - determines if only v operator in string
% On input:
%     s (string): fully parenthesized logical sentence
% On output:
%     b (Boolean): 0 if ^ in s, else 1
% Call:
%     b = BR_only_AND('((1)v(2))');
% Author:
%     T. Henderson
%     UU
%     Fall 2018
%

NEG_OP = '~';
AND_OP = '^';
```

```
b = 1;

len_s = length(s);

for c = 1:len_s
    if s(c)==NEG_OP|s(c)==AND_OP
        b = 0;
        return
    end
end
```

BR_parity.m

```
function parity = BR_parity(L1,L2,L3)
% BR_parity - parity of literals
% On input:
%     L1 (int): literal 1
%     L2 (int): literal 2
%     L3 (int): literal 3
% On output:
%     parity (int): parity of literals
% Call:
%     p = BR_parity(1,2,3);
% Author:
%     T. Henderson
%     UU
%     Summer 2018
%

if L3==0
    parity = 2*(L1>0) + (L2>0);
else
    parity = 4*(L1>0) + 2*(L2>0) + (L3>0);
end
```

BR_prob_or.m

```
function prob = BR_prob_or(clause,L_probs)
% BR_prob_or - compute probability of or clause
% On input:
%     clause (1xk vector): disjuction of literals
%     L_probs (1xn vector): probabilities of literals
% On output:
%     prob (float): probability of disjunction
% Call:
```

```
%     p = BR_prob_or(clause,L_probs);
% Author:
%     T. Henderson
%     UU
%     Spring 2017
%

prob = 0;
if isempty(clause)
    return
end
len_clause = length(clause);
if len_clause==1
    prob = L_probs(abs(clause(1)));
    return
end
p1 = L_probs(abs(clause(1)));
p2 = BR_prob_or(clause(2:end),L_probs);
prob = p1 + p2 - p1*p2;
```

BR_prob_or_atom_probs.m

```
function prob = BR_prob_or_atom_probs(clause,A_probs)
% BR_prob_or_atom_probs -= compute probability of
%   or clause from atom probs
% On input:
%     clause (1xk vector): disjunction of literals
%     A_probs (1xn vector): atom probabilities
% On output:
%     prob (float): probability of clause
% Call:
%     p = BR_prob_or_atom_probs(clause,A_probs);
% Author:
%     T. Henderson
%     UU
%     Spring 2017
%

prob = 0;
if isempty(clause)
    return
end
len_clause = length(clause);
if len_clause==1
    if clause(1)<0
        prob = 1 - A_probs(abs(clause(1)));
```

```
    else
        prob = A_probs(abs(clause(1)));
    end
    return
end
L_probs = A_probs;
for e = 1:len_clause
    if clause(e)<0
        L_probs(abs(clause(e))) = 1 - A_probs(abs(clause(e)));
    end
end
p1 = L_probs(abs(clause(1)));
p2 = BR_prob_or(clause(2:end),L_probs);
prob = p1 + p2 - p1*p2;
```

BR_put_vtable.m

```
function [vtable_out,index,new_entry] =
BR_put_vtable(vtable,L1,L2,L3,c,op)
% BR_put_table - add entry to vtable
% On input:
%     vtable (kx7 array): variable table
%     L1 (int): first literal value
%     L2 (int): second literal value
%     L3 (int): third literal value (or 0 if none)
%     c (int): clause number
%     op (int): operator value (1 is 'AND'; 2 is 'OR')
% On output:
%     vtable_out (px1 array): updated vtable
%     index (int): index of added (or existing) variable
%     new_entry (Boolean): 1 if new entry, esle 0
% Call:
%     [vtable,index23,ne] = BR_put_vtable(vtable,1,2,3,3,1)
% Author:
%     T. Henderson
%     UU
%     Summer 2018
%

vtable_out = vtable;
new_entry = 0;
p = BR_parity(L1,L2,L3);
index = BR_get_vtable(vtable,L1,L2,L3,0,p,op);
if isempty(index)
    len_vtable = length(vtable(:,1));
    vtable_out(len_vtable+1,1:7) = [L1,L2,L3,c,p,len_vtable+1,op];
    index = len_vtable + 1;
    new_entry = 1;
```

```
end
```

BR_remove_eq.m

```
function s = BR_remove_eq(s_in)
% BR_remove_eq - remove equivalence from logical expression
% On input:
%     s_in (string): logical string
% On output:
%     s (string): s_in with a=b replaced with (a>b)^(b>a)
% Call:
%     s1 = BR_remove_eq(s)
% Author:
%     T. Henderson
%     UU
%     Fall 2018
%

OR_OP = 'v';
AND_OP = '^';
EQ_OP = '=';
IMPLIES_OP = '>';

LEFT_PAR = '(';
RIGHT_PAR = ')';

s = s_in;
index = 0;
change = 1;

while change==1
    len_s = length(s);
    if index>=len_s
        change = 0;
    end
    index = index + 1;
    while index<=len_s&s(index)~=EQ_OP
        index = index + 1;
    end
    if index<len_s
        count = 1;
        index1 = index - 1;
        while count>0
            index1 = index1 - 1;
            if s(index1)==LEFT_PAR
                count = count - 1;
            elseif s(index1)==RIGHT_PAR
                count = count + 1;
```

```
            end
        end
        S1 = s(index1:index-1);
        count = 1;
        index2 = index + 1;
        while count>0
            index2 = index2 + 1;
            if s(index2)==LEFT_PAR
                count = count + 1;
            elseif s(index2)==RIGHT_PAR
                count = count - 1;
            end
        end
        S2 = s(index+1:index2);
        S = s(index1-1:index2+1);
        [S11,S12] = BR_getEQargs(S);
        prefix = s(1:index1-1);
        postfix = s(index2+1:end);
        middle = ['(',S1,IMPLIES_OP,S2,')^(',S2,IMPLIES_OP,S1,')'];
        s = [prefix,middle,postfix];
        change = 1;
        index = 0;
    end
end
```

BR_remove_implies.m

```
function s = BR_remove_implies(s_in)
% BR_remove_implies - remove implies from logical expression
% On input:
%     s_in (string): logical string
% On output:
%     s (string): s_in with a>b replaced with ~avb
% Call:
%     s1 = BR_remove_implies(s)
% Author:
%     T. Henderson
%     UU
%     Fall 2018
%

NEG_OP = '~';
OR_OP = 'v';
AND_OP = '^';
EQ_OP = '=';
IMPLIES_OP = '>';

LEFT_PAR = '(';
```

```
RIGHT_PAR = ')';

s = s_in;
index = 0;
change = 1;

while change==1
    len_s = length(s);
    if index>=len_s
        change = 0;
    end
    index = index + 1;
    while index<=len_s&s(index)~=IMPLIES_OP
        index = index + 1;
    end
    if index<len_s
        count = 1;
        index1 = index - 1;
        while count>0
            index1 = index1 - 1;
            if s(index1)==LEFT_PAR
                count = count - 1;
            elseif s(index1)==RIGHT_PAR
                count = count + 1;
            end
        end
        S1 = s(index1:index-1);
        count = 1;
        index2 = index + 1;
        while count>0
            index2 = index2 + 1;
            if s(index2)==LEFT_PAR
                count = count + 1;
            elseif s(index2)==RIGHT_PAR
                count = count - 1;
            end
        end
        S2 = s(index+1:index2);
        S = s(index1-1:index2+1);
        [S11,S12] = BR_getIMPLIESargs(S);
        prefix = s(1:index1-1);
        postfix = s(index2+1:end);
        middle = ['(',NEG_OP,S1,')',OR_OP,S2];
        s = [prefix,middle,postfix];
        change = 1;
        index = 0;
    end
end
```

BR_s2CNF.m

```
function CNF = BR_s2CNF(s)
% BR_s2CNF - convert general logical expression to CNF
%   fully parenthesized using ~,v,^,>,=
% On input:
%     s (string): logical string
% On output:
%     CNF (string): conjunction of disjunctions
%       not is pushed into the atoms as -<int> (no negations left)
% Call:
%     s1 = '((((2)v(3))>(1))=(4)))';
%     C1 = BR_s2CNF(s1)
%        =
%   '(((((2)v(3))v(4))^((-1)v(4)))^(((-4)v((-2)v(1)))^((-4)v((-3)v(1))))))'
% Author:
%     T. Henderson
%     UU
%     Fall 2018
%

s1 = BR_remove_eq(s);
s2 = BR_remove_implies(s1);
s3 = BR_move_not_in(s2);
v = BR_max_var_in_s(s3);
CNF = BR_switch_vars(s3,v);
```

BR_s2KB.m

```
function KB = BR_s2KB(s)
% BR_s2KB - convert logical sentence to KB
% On input:
%     s (string): logical sentence (fully parenthesized)
% On output:
%     KB (KB data structure): knowledge base
%       (k).clauses (vector): or clause
%       (k).prob (float): probability of clause
% Call:
%     s7 = '((((1)^(2))v((3)^(4)))v((5)^(6)))';
%     s7a = BR_switch_vars(s7,6);
%      KB = BR_s2KB(s7a);
% Author:
%     T. Henderson
%     UU
%     Fal 2018
%

s = BR_s2CNF(s);
```

```
frontier = [1];
sentences(1).s = s;
num_sentences = 1;
num_clauses = 0;

while ~isempty(frontier)
    s = sentences(frontier(1)).s;
    frontier = frontier(2:end);
    if BR_only_OR(s)
        num_clauses = num_clauses + 1;
        clauses(num_clauses).clause = s;
    else
        [S1,S2] = BR_getANDargs(s);
        num_sentences = num_sentences + 1;
        sentences(num_sentences).s = S1;
        num_sentences = num_sentences + 1;
        sentences(num_sentences).s = S2;
        frontier = [frontier,num_sentences-1,num_sentences];
    end
end
n = 0;
for s = 1:num_sentences
    if BR_only_OR(sentences(s).s)
        n = n + 1;
        KB(n).clauses = BR_s2clause(sentences(s).s);
        KB(n).prob = 0;
    end
end

tch = 0;
```

BR_s2clause.m

```
function clause = BR_s2clause(s)
% BR_s2clause - convert logical string to clause
% On input:
%     s (string): logical sentence (dijunction; fully parenthesized)
% On output:
%     clause (vector): or clause (represented as integers)
% Call:
%     cl = BR_s2clause('((1)v(2))';
% Author:
%     T. Henderson
%     UU
%     Fall 2018
%
```

```
MINUS = '-';

clause = [];

len_s = length(s);
d = isstrprop(s,'digit');

for c = 1:len_s
    if d(c)
        index = c;
        val = 0;
        while d(index)
            val = 10*val + str2num(s(index));
            index = index + 1;
        end
        if s(c-1)==MINUS
            val = -val;
        end
        clause = [clause,val];
    end
end
```

BR_sat.m

```
function b = BR_sat(sentence,bits)
% BR_sat - check if bits satisfy the sentence
% On input:
%     sentence (1xk vector): disjunction of k literals
%     bits (1xn vector): n truth assignments to logical variables
% On output:
%     b (Boolean): 1 if bits satisfy sentence; else 0
% Call:
%     b = BR_sat([1,-2],[0,1]);
% Author:
%     T. Henderson
%     UU
%     Summer 2018
%

b = 0;

len_sentence = length(sentence);
for e = 1:len_sentence
    if sentence(e)>0==bits(abs(sentence(e)))
        b = 1;
        return
    end
end
```

BR_switch_vars.m

```
function s_out = BR_switch_vars(s,v)
% BR_switch_vars - polynomial time distribution of v over ^
% On input:
%     s (string): logical sentence (fully parenthesize)
%     v (int): max variable value (as integer)
% On output:
%     s_out (string): s with v distributed over ^
% Call:
%     ss = BR_switch_vars('(1)',1);
% Author:
%     T. Henderson
%     UU
%     Fall 2018
%

s_out = s;

if isempty(s)|BR_isvariable(s)|BR_only_OR(s)|BR_only_AND(s)
    return
end

if BR_isORop(s)
    [OP1,OP2] = BR_getORargs(s);
    if BR_isliteral(OP1)&BR_only_AND(OP2)
        literals = BR_get_literals(OP2);
        num_literals = length(literals);
        C = ['(',OP1,'v','(',num2str(literals(1)),'))'];
        for k = 2:num_literals
            C = ['(',C,'^(',OP1,'v(',num2str(literals(k)),')))'];
        end
        s_out = C;
        return
    elseif BR_isliteral(OP1)
        ss = BR_switch_vars(OP2,v);
        args = BR_all_and_args(ss);
        num_args = length(args);
        C = ['(',OP1,'v(',args(1).s,'))'];
        for a = 2:num_args
            C = ['(',C,'^(',OP1,'v(',args(a).s,')))'];
        end
        s_out = C;
        return
    end
end

if BR_isANDop(s)
    [OP1,OP2] = BR_getANDargs(s);
```

```
    s_out = ['(',BR_switch_vars(OP1,v),'^',BR_switch_vars(OP2,v),')'];
else
    [OP1,OP2] = BR_getORargs(s);
    v = v + 1;
    LHS = ['((-',num2str(v),')v',OP1,')'];
    RHS = ['((',num2str(v),')v',OP2,')'];
    s_out = ['(',BR_switch_vars(LHS,v),'^',BR_switch_vars(RHS,v),')'];
end
```

BR_test_NILS.m

```
function BR_test_NILS
% BR_test_NILS - test PSAT functions
% On input:
%     N/A
% On output:
%     echoes variable values after calls
% Call:
%     BR_test_NILS
% Author:
%     T. Henderson
%     UU
%     Summer 2018
%

% Set up a KB
KB(1).clauses = [1];
KB(1).prob = 0.7;
KB(2).clauses = [-1,2];
KB(2).prob = 0.7;

% Set up a query (to test Nilsson method)
query(1).clauses = [2];
query(1).prob = 0;

% Run Nilsson
[P,pq] = BR_Nilsson_method_all(KB,query)

% Run BR_gen_test_KB_ind
[KB_ind,CC,ap] = BR_gen_test_KB_ind(3,7,3)

% Run BR_gen_test_KB_gen
[KB_gen,CC] = BR_gen_test_KB_gen(3,10,3)

% Run BR_KB2F_ind
BR_KB2F_ind(KB,'KB')

% Run BR_KB2F_gen
```

```
[vtable,F] = BR_KB2F_gen(KB,'KBg')

% Run gradient descent on ind variables
[x,e,xt,et] = BR_KB_GD(KB,'KB',0.001,0.01,10000,0.5*ones(2,1));
x
e


% Run gradient descent on general variables
[x,e,xt,et] = BR_KB_GD(KB,'KB',0.001,0.01,10000,0.5*ones(3,1));
x
e
```

BR_vars.m

```
function vars = BR_vars(KB,sentence)
% BR_vars - find list of variables in logical sentences
% On input:
%     KB (nx1 conjunctive normal form vector): conjunctive clauses
%       (i).clauses (1xm vector): disjunctive clause
%     sentence (1x1 conjunctive normal form vector): conjunctive
%     clause
%       (1).clauses (1xk vector): disjunctive clause
% On output:
%     vars (1xp vector): list of variables in KB and sentence
% Call:
%     vars = BR_vars(KB,thm);
% Author:
%     T. Henderson
%     UU
%     Spring 2017
%

vars = [];

for s = 1:length(KB)
    vars = unique([vars,abs(KB(s).clauses)]);
end
for s = 1:length(sentence)
    vars = unique([vars,abs(sentence(s).clauses)]);
end
vars = sort(vars);
```