Genifer in Lisp

editor: YKY (general.intelligence@Gmail.com)

Contents

1 Preface

1.1 Literate programming

The source code is contained in these files:

Genifer.tex (contains Latex header etc)
memory.lisp
deduction-Z.tex
unification.lisp
induction.tex
pretty-printing.lisp
unser-interface.lisp
main.lisp

This file is Genifer.pdf. It is compiled from Genifer.tex by:
(The command may need to be run twice to generate the table of contents and references.)

```
pdflatex Genifer.tex
```

The Lisp source code can be extracted from *.tex by the *nix utility sed:

```
sed '/^\\end/,/language=Lisp\]\$/ d' >FILENAME.lisp <FILENAME.tex
```

2 Z deduction

A set of introductory slides about deduction is here: Slides.

Abbreviations:

sub = substitution
TV = truth value

2.1 To do

1. factor graph calculations are wrong
   1.1. allow multiple-head rules – OK
   1.2. store the way to calculate subgoal in the factor
2. using rules in the abductive direction
3. abduction
4. junction nodes (for cycle resolution)

2.2 Introduction

All truth values are formated as (Z . confidence)
All rules are all of the form:

((head) (body))

where

(head) = (predicate arg1 arg2 ...)

note: the # of args can be 0
and

(body) = (θ c_1 c_2 ... X_1 X_2 ...)

where θ is the operator, represented as a number in [0,1] where

θ = 0 or 1 corresponds to AND or OR
$X_1, X_2, ...$ are the literals
$c_1, c_2, ...$ are parameters associated with each literal

Each literal can invoke other operators recursively.

For example:

smart $\leftarrow$ creative $\overset{Z}{\vee}$ humorous
smart(?X) $\leftarrow$ creative(?X) $\overset{Z}{\vee}$ humorous(?X)

can be expressed as a Z rule as:

((smart ?X) ('Z-OR (creative ?X) (humorous ?X)))

The best-frist search algorithm can be implemented using call-by-continuations which allows remembering the search state, but we don't use that here. We simply maintain the priority queue.

2.3 Parameters

```
(defparameter *max-depth*      7)     ; maximum search depth
(defparameter *max-rule-nodes* 150)   ; maximum number of rule nodes in proof tree
(defparameter max-rule-length  20)    ; rule length = \# of arguments (literals) it has
(defparameter W0 50.0)                ; constant used in calculating confidence from w
(defparameter abduction-penalty 0.2)  ; < 1.0;  used to reduce the scores of abductive steps
(defparameter facts-boost       1.5)  ; > 1.0;  increases the scores of facts
(defparameter args-decay        0.8)  ; < 1.0;  decreases the scores of consecutive args in a rule
(defparameter testing-decay     0.98) ; < 1.0;  decreases scores of successive literal tests
```

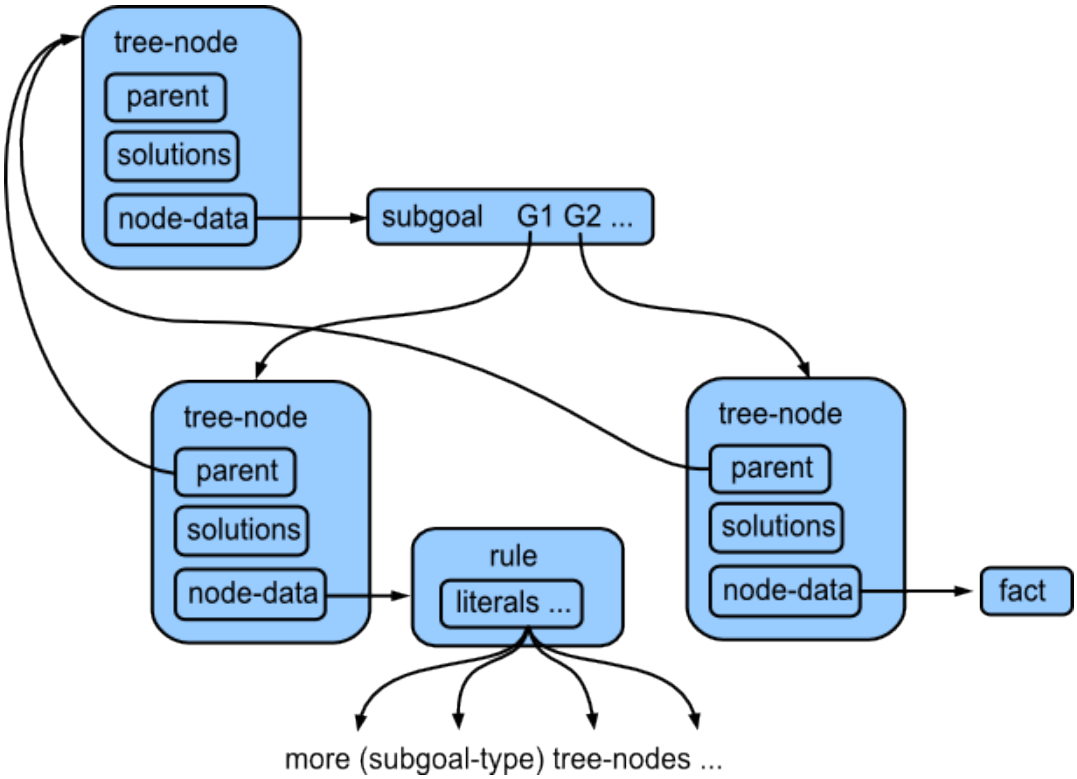


Figure 2.1: Data structures

2.4 Data structures

Proof tree is a tree with back-links so we can go from a child node to its parent node. This is to facilitate the bottom-up evaluation of expressions, with the goal at the root. A tree node can be either a sub-goal or a fact/rule. Each tree node has a pointer to its parent.

A sub-goal node contains a list in its "node-data" slot:
(sub-goal (child node1) (child node2) ...)

A rule node contains a rule class item (see below) in the "node-data" slot;
It has the following elements:
operator confidence literal1 [literal2...] [params...]
where literal1,2,... are proof tree nodes and each has a pointer pointing to *this* node. The parameters of the rule create a CPT (conditional probability table) which is then represented as a factor (as in factor graph).

A fact node contains the fact (as a list) in its "data" slot;

solutions = fuzzy messages that pass from node to node in the factor graph algorithm
The idea originated from Judea Pearl's message-passing algorithm for Bayes nets. Each solution is also associated with a substitution (because each substitution instantiates a distinct propositional sub-tree in the factor graph, and we use only one sub-tree to represent all those instances).
So we can potentially have a list of solutions for each node, not just one solution per node (In a more advanced version, such a list can be implemented as a lazy sequence).

node-type can be:
For subgoal nodes:
#\H = head literal, or
#\B = body literal, or
#\X = unknown (all else)
For rule nodes:
top's type = #\H or #\B or #\X (same as above)

```
(defclass tree-node () (
  (parent     :initarg :parent     :accessor parent                    :type symbol)
  (solutions  :initarg :solutions  :accessor solutions  :initform nil  :type (list solution))
```

```
  (node-data  :initarg :node-data  :accessor node-data  :initform nil  :type list)
  (node-type  :initarg :node-type  :accessor node-type  :initform #\X  :type char)
))

;;; Each solution is a pair:  (sub, message)
(defclass solution () (
  (sub     :initarg :sub     :accessor sub     :initform nil  :type list)
  (message :initarg :message :accessor message :initform nil)
))

;;; Class for a rule data item within a proof tree node
;;; rule-type is either #\H or #\B depending if the top node is a head literal or body literal
(defclass rule () (
; (op          :initarg :op          :accessor op          :type single-float)
  (literals   :initarg :literals   :accessor literals   :type list           :initform nil)
  (parameters :initarg :parameters :accessor parameters :type (list single-float))
  (rule-type  :initarg :rule-type  :accessor rule-type  :type char           :initform #\X)
  (confidence :initarg :confidence :accessor confidence :type single-float)
  (factor     :initarg :factor     :accessor factor     :type single-float)
))

;;; An item of the priority list.
;;;      list-data = either a sub-goal or a fact/rule
;;;      ptr       = pointer to proof-tree node
;;;      data-type = rule-type, see above
(defclass p-list-item () (
  (score     :initarg :score      :accessor score      :type single-float  :initform 0.0)
  (depth     :initarg :depth      :accessor depth      :type fixnum        :initform 0)
  (list-data :initarg :list-data  :accessor list-data  :type symbol)
  (data-type :initarg :data-type  :accessor data-type  :type char          :initform #\X)
  (ptr       :initarg :ptr        :accessor ptr        :type symbol)
))

(defvar proof-tree      nil)          ; The initial proof-tree
(defvar priority-list   nil)
(defvar new-states-list nil)
(defvar *explanation*   nil)          ; the result of abduction
(defvar *abduct-mode*   nil)          ; true if abduction mode is ON

(defvar timer             0)
(defvar *goal-nodes*      0)
(defvar *rule-nodes*      0)
(defvar *fact-nodes*      0)

(setf *print-circle* t)               ; Lisp key that allows printing of circular objects
```

2.5 Backward-chaining – best-first search

NOTE: the abduction algorithm is embedded in this code.

**** Best-first search algorithm:
==========
0. Loop forever:
1. If priority queue is empty, return fail
2. Remove first item from priority list (the item points to a node in the proof tree)
3. Go to the proof tree and process the node;
This results in some new child nodes of the proof tree node
4. Put the new nodes into priority queue (via ranking-based merging)
==========
About the scoring of the priority list:
There are 3 types of items in the priority list: subgoals, rules, and facts;
Each has some factors that can be taken into consideration when calculating the score:
A. rule – length, confidence, depth
B. fact – confidence, depth
C. subgoal – order within its rule, assuming most-significant first

```
(defun backward-chain (query)
  ;; 0. Initialize
  (setf proof-tree (make-instance 'tree-node :parent nil))
  (setf *goal-nodes*  1
        *rule-nodes*  0
        *fact-nodes*  0)
```

```
  ;; On entry, priority list contains the goal query
  (setf priority-list (list (make-instance 'p-list-item
                              :score      0.0
                              :depth      0
                              :list-data  query
                              :ptr        proof-tree)))
  (loop
    ;; Has found result?
    (if (or (not (null (solutions proof-tree)))
            (equal 'fail (solutions proof-tree)))
      (return))
    ;(if (> (- (get-internal-run-time) timer) 10000000)
    ;   (return))
    ;; 1. Remove first item from priority list
    (if (null priority-list)
      (return))
    (setf best (car priority-list))               ; "best" = top item on priority list
    (setf priority-list (cdr priority-list))
    ;; The current tree node being processed:
    (setf node (ptr best))
    ;; Maximum search depth reached?
    (setf depth (depth best))
    (if (> depth *max-depth*)
      (progn (****DEBUG 3 "backward-chain: FAIL -- past max tree depth")
        (return 'fail)))
    (if (> *rule-nodes* *max-rule-nodes*)
      (progn (****DEBUG 3 "backward-chain: FAIL -- past max num of rules")
        (return 'fail)))
    (if (> *goal-nodes* *max-rule-nodes*)
      (progn (****DEBUG 3 "backward-chain: FAIL -- past max num of goals")
        (return 'fail)))
    (****DEBUG 1 "~%")
    (****DEBUG 1 "backward-chain: proof-tree node = ~a" (node-data node))
    ;; 2. Process the current node, which can be a sub-goal, a rule, or a fact:
    (setf new-states-list (list nil))
    ;; Is it a sub-goal?
    (if (listp (list-data best))
      (process-subgoal node best)
      ;; Else... it is either a fact/rule
      (let ((formula (list-data best)))
        ;; Is it a fact?
        (if (null (body formula))
          (process-fact node formula best)
          ;; It's a rule:
          (process-rule node formula best))))
    ;; 3. Merge results with priority queue
    ;;    Results are stored in "new-states-list"
    (setf new-states-list (cdr new-states-list))   ; remove first element which is a dummy nil
    (sort new-states-list #'compare-scores)
    (****DEBUG 1 "backward-chain: new-states-list has ~a element(s)" (length new-states-list))
    (setf priority-list (merge 'list new-states-list priority-list #'compare-scores))
    (print-priority-list)
    (print-proof-tree)))

;;; compare the scores of 2 priority-list items
(defun compare-scores (new old)
  (> (score new) (score old)))
```

2.6 Building the proof tree

2.6.1 Processing a sub-goal

On entry, "node" is the proof tree node being processed

1. Fetch facts / rules that match the subgoal
2. For each applicable rule / fact:
3. Create a corresponding node in proof tree, as a child of the current node
4. Prepare the new nodes to be pushed to priority list

```
(defun process-subgoal (node best)
  (setf best-data (list-data best))
  (****DEBUG 1 "process-subgoal: sub-goal ~a" best-data)
  ;; store the sub-goal in the proof tree node
  (setf (node-data node) (list best-data))
  (setf (node-type node) (data-type best))
  ;; Fetch rules that match the subgoal:
  ;; 'fetch-rules' is defined in memory.lisp
  (multiple-value-bind (facts-list rules-list) (fetch (car best-data))
```

```
    (dolist (formula facts-list)
      ;; standardize apart head-of-subgoal and formula-to-be-added
      (setf subs (standardize-apart (head formula) nil)
                 (head formula) (do-subst (head formula) subs)
                 (body formula) nil)
      (****DEBUG 1 "process-subgoal: adding fact... ~a" (head formula))
      ;; create proof tree node for the fact, with initial tv = nil
      (setf new-node (make-instance 'tree-node :parent node))
      (incf *fact-nodes*)
      ;; add node as current node's child
      (nconc (node-data node) (list new-node))
      ;; prepare priority list item:
      ;; score := confidence * (1 - partial_length / max_length)
      (setf depth (depth best))
      (setf score (* facts-boost (confidence formula) (- 1.0 (/ depth *max-depth*))))
      (setf new-state (make-instance 'p-list-item
                        :score     score
                        :depth     (+ 1 depth)
                        :list-data formula
                        :ptr       new-node))
      (nconc new-states-list (list new-state)))
    ;; Do the same for rules:
    (dolist (formula rules-list)
      ;; standardize apart head-of-subgoal and formula-to-be-added
      (setf subs (standardize-apart (head formula) (body formula))
                 (head formula) (do-subst (head formula) subs)
                 (body formula) (do-subst (body formula) subs))
      (setf body (body formula))
      (****DEBUG 1 "process-subgoal: adding rule... ~a <- ~a" (head formula) body)
      ;; create proof tree node and store rule-formula in it
      (incf *rule-nodes*)
      (setf new-node (make-instance 'tree-node
                       :parent     node
                       :node-data  (make-instance 'rule
                                     :parameters (if (numberp body)    ; bodyless rule?
                                                     nil
                                                     (list (car body)))  ; first element = theta
                                     :confidence (confidence formula)
                                     :literals   nil)))
      ;; Add node as current node's child
      (nconc (node-data node) (list new-node))
      ;; Prepare priority list item:
      (setf depth (depth best))
      (if (numberp body)
        (setf score  (* (confidence formula) (- 1.0 (/ depth *max-depth*))))
        (setf score  (* (confidence formula) (- 1.0 (/ depth *max-depth*))
                                             (- 1.0 (/ (length body) max-rule-length))
                                             (if (> (top-index formula) 0)
                                               abduction-penalty
                                               1.0))))
      (setf new-state (make-instance 'p-list-item
                        :score     score
                        :depth     (+ 1 depth)
                        :list-data formula
                        :ptr       new-node))
      (nconc new-states-list (list new-state)))))
```

2.6.2 Processing a fact node

On entry, node points to the fact node in proof tree
0. Try to unify the fact with its parent node
1. IF unify() succeeds:
2. add the sub to the node;
3. send the resulting sub up to parent node
5. update the entire proof tree in a bottom-up manner (by calling propagate)
6. ELSE ; unify() fails
7. no substitution is added
8. the node can be deleted

```
(defun process-fact (node formula best)
  (****DEBUG 1 "process-fact: processing fact formula: ~a" (head formula))
  ;; Try to unify the fact with the parent node (the latter is a sub-goal):
  (****DEBUG 1 "process-fact: to unify with parent: ~a" (car (node-data (parent node))))
  (setf sub (unify (head formula) (car (node-data (parent node)))))
  ;; If unify fails, retract the proof tree node
  (if (eq sub 'fail)
    (progn
```

```
    (decf *fact-nodes*)
    (retract node best)
    (return-from process-fact)))
;; If unify succeeds:
(****DEBUG 1 "process-fact: substitution = ~a" sub)
;; Store the (sub, message) pair in the list of solutions in the current node
;; Here we pass the first message from a leaf node
;; Note that there is an implicit factor node below this leaf, but it simply passes the
;;    fuzzy value of the node to its parent.  (first TV) = fuzzy value
(setf current-solution (make-instance 'solution
                         :sub sub
                         :message (first (tv formula))))
;; The current node is filled with the literal (this is useful for abduction; see below)
(setf (node-data node) (list 'fact (head formula)))
;; Record the node for abduction
;(setf *current-explanation* (list 'fact (head formula)))
;; Propagate the TV bottom-up, recursively
(propagate node (list current-solution)))
```

2.6.3 Processing a rule node

INPUT: formula is a mem-item

1. Try to match parent node with head of rule.
2. IF unify succeeds:
3. Set up the proof tree sub-nodes (= literals of the rule)
4. Return the subgoals (= arguments = antecedents) to be pushed to priority list
5. ELSE:
6. retract the node

```
(defun process-rule (node formula best)
  (****DEBUG 1 "process-rule: expanding rule formula ~a <- ~a" (head formula) (body formula))
  (setf head       (head formula)
        body       (body formula)
        top-index  (top-index formula))
  ;; At this point, a designated part of the rule (let's call it 'top') is already (partly)
  ;;   matched with the parent.  The parent is a subgoal.  We need to perform unify on them.
  (****DEBUG 1 "process-rule: to unify with parent: ~a" (car (node-data (parent node))))
  ;; top = the rule's actual "head" per this instantiation
  ;; if top-index > 0, top is a body literal; otherwise top is a head literal
  (if (> top-index 0)
    (setf top (nth (+ top-index (/ (- (length body) 1) 2)) body))
    (setf top (nth (- top-index)   head)))
  (****DEBUG 1 "process-rule: to unify with index: ~a, top: ~a" top-index top)
  (setf sub (unify top (car (node-data (parent node)))))
  (****DEBUG 1 "process-rule: substitution = ~a" sub)
  ;; If unify fails, retract proof tree node
  (if (eq sub 'fail)
    (progn
      (decf *rule-nodes*)
      (retract node best)
      (return-from process-rule)))
  ;; If unify succeeds:
  ;; We don't have to apply sub to the rule's top b/c the top is identical with the parent
  ;;    and is not stored in the rule.
  ;; If the rule is body-less, we immediately have a solution that should be propagated up
  ;;    to the parent:
  (if (numberp body)
    (progn
      (setf current-solution (make-instance 'solution
                               :sub sub
                               :message  body))                    ; the TV is the body
      (propagate node (list current-solution))
      (return-from process-rule)))
  (if (> top-index 0)
    (setf (node-type node) #\B)
    (setf (node-type node) #\H))
  ;; For each parameter c or body literal...
  (setf top-index2 1)
  (dolist (arg (cdr body))
    ;; Auxiliary parameters?
    (if (floatp arg)
      ;; Yes, add them to parameter list
      ;; The original order should be preserved regardless of where top-index is.
      (setf (parameters (node-data node)) (append (parameters (node-data node)) (list arg)))
      ;; For each literal:
      (progn
        (if (not (equal top-index top-index2))
          (progn
```

```
            ;; Apply substitution to literal:
            (setf lit (do-subst arg sub))
        ;; Create new nodes for the proof-tree, corresponding to arguments of the rule:
        ;; The operator and confidence are already in the proof-tree node
            (incf *goal-nodes*)
        (setf new-lit (make-instance 'tree-node :parent node))
        ;; Add new lit to current node's literals list
        (setf (literals (node-data node)) (append (literals (node-data node)) (list new-lit)))
        ;; Prepare item for priority list:
        ;; note:  depth need not increase
        ;;        score is inherited from the rule's, multiplied by a consecutive decay factor
        (nconc new-states-list (list (make-instance 'p-list-item
                                    :score     (* args-decay (score best))
                                    :depth     (depth best)
                                    :list-data lit
                                        :data-type #\B               ; #\B = body literal
                                        :ptr       new-lit)))))
        (incf top-index2))))
  ;; For each head literal...
  (setf top-index2 0)              ; head index counts from 0 downwards
  (dolist (arg head)
    (if (not (equal top-index top-index2))
      (progn
        ;; Apply substitution to literal
        (setf lit (do-subst arg sub))
        ;; Create new node
        (incf *goal-nodes*)
        (setf new-lit (make-instance 'tree-node :parent node))
        ;; Add new lit to current node's literals list
        (setf (literals (node-data node)) (append (literals (node-data node)) (list new-lit)))
        ;; Prepare item for priority list
        (nconc new-states-list (list (make-instance 'p-list-item
                                    :score     (* args-decay (score best))
                                    :depth     (depth best)
                                    :list-data lit
                                    :data-type #\H               ; #\H = head literal
                                    :ptr       new-lit)))))
    (decf top-index2)))
```

2.6.4 Retracting a failed node

On entry: current node has failed

1. remove the node
2. IF parent is a goal:
3. parent has no child?
4. if so, recurse to remove parent
5. IF parent is a rule:
6. the entire rule needs to be removed

```
(defun retract (node best)
  (****DEBUG 1 "retracting node...")
  (setf parent (parent node))
  ;; The root node failed?
  (if (null parent)
    (progn
      (setf (solutions proof-tree) 'fail)
      (return-from retract)))
  ;; Is parent a rule or subgoal?
  (if (listp (node-data parent))
    ;; If parent is a sub-goal:  delete the node itself (ie the parent's child);
    ;; If the parent becomes empty then recurse
    (progn
      ;; Delete the node:  destructively modify the parent
      ;(decf *rule-nodes*)
      (setf (node-data parent) (delete node (node-data parent) :count 1))
      ;; Is parent empty now?  If so, delete it too
      (if (null (third (node-data parent)))       ; third item is the subgoal's first argument
        ;; recurse
        (retract parent best))
    )
    ;; If parent is a rule: delete parent
    (retract parent best)
  ))
```

2.7 Belief propagation

2.7.1 Introduction

This part concerns the propagation of truth values. For example, we have an inference step:

$$\begin{array}{ll} A \twoheadrightarrow B & \langle TV_1 \rangle \\ A & \langle TV_2 \rangle \\ \hline B & \langle TV_3 \rangle \end{array}$$

So TV_3 would be "some function" of TV_1 and TV_2. Each TV is some number(s).

In **NARS** (one of the early AGI logics that influenced both OpenCog and Genifer), Pei Wang uses 2 numbers as TV. One is somewhat like probability, the other is confidence.

In **OpenCog's PLN**, Ben Goertzel uses up to 4 numbers for a TV.

In **Genifer**, we use 1 number for probability, another number if fuzzy-probabilistic, and another number for confidence.

In Genifer, the propagation of probabilistic TVs follows strictly the belief propagation algorithm in Bayesian networks, which is why it is more complicated.

In addition, this part of the AGI also solves the problem of **defeasible reasoning**. Two examples are:

1. *Tweety is a penguin, which is a bird. All birds can fly. But penguins are an exception that cannot fly. Can Tweety fly?*

2. *John promises to go to the airport to meet Mary. John is very conscientious so he will probably arrive at the airport. But, John is killed en route in a car accident. Will John arrive at the airport?*

Both examples require one line of reasoning to "defeat" the other. Pei Wang provides a solution with the **weighted average**:

$$f_3 = \frac{f_1 c_1 + f_2 c_2}{c_1 + c_2}$$

where the f's are probabilities and c's are confidences (ie, how many examples support each hypothesis). This is a neat solution, but Abram Demski formulated an alternative solution that more closely conforms with Bayesian theory.

This is all you need to know if you want to treat this part of the algorithm as a black box. If you want to know more details, read on. The AGI book explains the theory in detail.

2.7.2 Pearl's algorithm

The algorithm is explained in detail in Chapter 4 of [Pearl, 1988]. We're using the belief propagation algorithm for polytrees (singly-connected networks), reproduced below:

A typical node (figure 2.2) updates its belief according to:

$$Bel(X) \triangleq (X|K^{\uparrow}, K^{\downarrow}) = \frac{(K^{\uparrow})}{(K^{\uparrow}, K^{\downarrow})}(K^{\downarrow}|X)(X|K^{\uparrow}) = \alpha\lambda(X)\pi(X) \tag{2.1}$$

where $K^{\uparrow}$ and $K^{\downarrow}$ are the evidence (or background knowledge, ie, nodes that are "clamped") upstream and downstream of X, α is the normalizing constant, and:

$$\lambda(X) \triangleq (K^{\downarrow}|X) = \prod_j \lambda_{Y_j \to X} \tag{2.2}$$

$$\pi(X) \triangleq (X|K^{\uparrow}) = \sum_{U_i}(X|U_{1\ldots n}) \prod_i \pi_{U_i \to X} \tag{2.3}$$

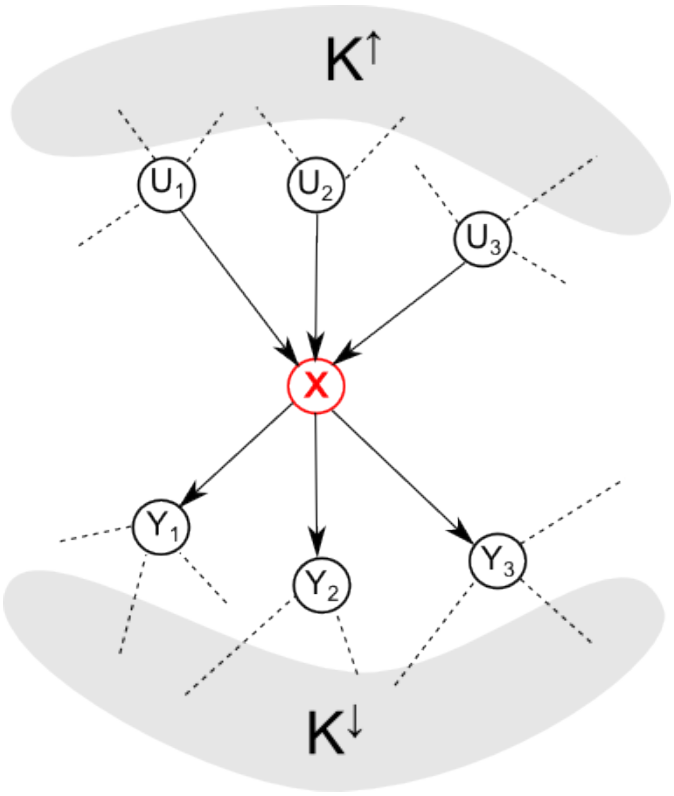


Figure 2.2: A typical polytree node X and its parents and children

where $\lambda_{Y_j \to X}$ and $\pi_{U_i \to X}$ are the messages sent to X from its children and parents, respectively. In turn, node X sends messages to its parents and children, given by[1]:

$$\lambda_{X \to U_i} = \beta \sum_X \lambda(X) \sum_{\{\setminus U_i\}} (X|U_{1\ldots n}) \prod_{k \neq i} \pi_{U_k \to X} \tag{2.4}$$

$$\pi_{X \to Y_j} = \alpha \pi(X) \prod_{k \neq j} \lambda_{Y_k \to X} \tag{2.5}$$

where α is same as above ($= \frac{(K^\uparrow)}{(K^\uparrow, K^\downarrow)}$) and $\beta = (K^\uparrow_{U_{k \neq i}})$.

2.7.3 Factor graph algorithm

A well-written introduction to factor graphs is [Kschischang, 2001]. A Bayes net can be easily translated into a factor graph by adding factor nodes. Let's look at a typical variable node X (figure 2.3).

1. From the top, each U_i sends to G:

$$\pi_{U_i \to G} = (U_i | K^\uparrow_{U_i}) \tag{2.6}$$

G sends to X:

$$\begin{aligned} \pi_{G \to X} &= \sum G \prod \pi \\ &= \sum_{U_i} (X|U_{1\ldots n}) \prod_i \pi_{U_i \to G} \end{aligned} \tag{2.7}$$

2. From the bottom, each Y_i sends to F_i:

$$\lambda_{Y_i \to F_i} = (K^\downarrow_{Y_i} | X) \tag{2.8}$$

[1] About the notation $\sum_{\{\setminus U_i\}}$: the probability $(X|U_{1\ldots n})$ is conditioned on all the U_k's including U_i, but the sum is over the permutation of all U_k's excluding U_i. This is a special kind of summation operation peculiar to probability calculations, also known as the **summary operation**. What it does is to perform the marginalization of a joint probability, by summing over all variables except the variable in question. Summary is special in that its order can be interchanged with multiplication. Thus the distributive law of multiplication can be exploited. For more about this, see [Kschischang, 2001]

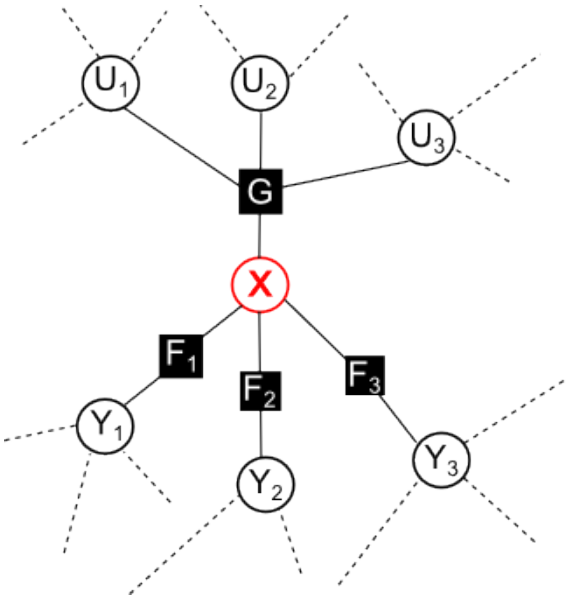


Figure 2.3: The previous Bayesian network node X translated to factor graph form

Each F_i sends to X:

$$\lambda_{F_i \to X} = \lambda_{Y_i \to F_i} \tag{2.9}$$

3. At this point $Bel(X)$ can be computed by multiplying all incoming messages at X. Then X sends to each node the product of *other* incoming messages. For example, it sends the product of $G, F_2, F_3, ...$'s messages to F_1, and so on.

4. Upwards, G sends to each U_i:

$$\begin{aligned} \lambda_{G \to U_i} &= \sum \lambda(X) \sum G \prod \pi \\ &= \sum_X \lambda(X) \sum_{\{\setminus U_i\}} (X|U_{1...n}) \prod_{k \neq i} \pi_{U_k \to G} \end{aligned} \tag{2.10}$$

5. Downwards, each F_i sends to each Y_i:

$$\begin{aligned} \pi_{F_i \to Y_i} &= \pi(X) \prod \lambda \\ &= \pi(X) \lambda_{X \to F_i} \end{aligned} \tag{2.11}$$

2.7.4 Evaluating a single rule

The rule is of the form:

$$H_1, H_2, ... \leftarrow B_1, B_2, B_3, ...$$

where H means "head" and B means "body".

The multi-head form is needed because *substitutions* are shared across the entire rule, but probabilistically the rule is equivalent to:

$$\begin{aligned} H_1 &\leftarrow B_1, B_2, ... \\ H_2 &\leftarrow B_1, B_2, ... \\ &.... \end{aligned}$$

Abram's method is so nice that we can use it to construct $(H|B_1, B_2...)$ out of the individual $(H|B_i)$'s, with the (optional) help of the prior (H).

There are 3 cases:

1A. We're seeking H_j. Forward application (= sum-product).

1B. We're seeking H_j but one of the B_i's is missing. Apply Case 2 and then apply Case 1A.
2. We're seeking B_j. Backward application (= Bayes law).

In the following assume K = background knowledge.

Case 1A

Simple sum-product:

$$(H_j|K) = \sum_{B_i}(H_j|B_i)(B_i|K)$$

Case 1B

One of the B_i's is missing, but we have one $H_{k\neq j}$. Apply Case 2 to get B_j.
Then apply Case 1A to get H_j.

Case 2

Simple application of Bayes law:

$$(B_i|K) = \sum_{H_j}\frac{(H_j,K)}{(H_j)}\sum_{B_{j\neq i}}(H_j|B_{1\ldots n})\prod_k(B_k|B_i) \tag{2.12}$$

2.7.5 Combining multiple rules – Abram's method

If all rules are aligned in the "forward" direction, as in:

$$\begin{aligned} H &\Leftarrow A_1, A_2, A_3, \ldots \\ H &\Leftarrow B_1, B_2, B_3, \ldots \\ &\ldots. \end{aligned}$$

Abram's method is a trick to construct a joint CPT (which would otherwise not be given) using marginal conditionals and priors:

$$\begin{aligned} (H|A_i,B_i,\ldots) &= \frac{(A_i,B_i,\ldots|H)(H)}{(A_i,B_i,\ldots)} && \text{Bayes law} \\ &= \frac{(A_i|H)(B_i|H)\ldots(H)}{(A_i,B_i,\ldots)} && \text{Naive Bayes assumption} \\ &= \frac{(H|A_i)(A_i)(H|B_i)(B_i)\ldots}{(A_i,B_i,\ldots)(H)^{n-1}} && \text{Bayes law again} \end{aligned}$$

where n is the number of rules.

Before applying the naive Bayes assumption, the joint CPT is underconstained by the given probabilities; Abram believes that one can always choose the priors judiciously within some bounds such that the resulting Bayes net remains probabilistically consistent.

Then we can apply the forward rule:

$$(H|K) = \sum_{A_i,B_i,\ldots}(H|A_i,B_i,\ldots)\prod(A_i|K)\prod(B_i|K)\ldots$$

If any of the rules is malaligned, we need to reverse that rule first, by Bayes law (see eqn 2.12).

Case $n = 2$

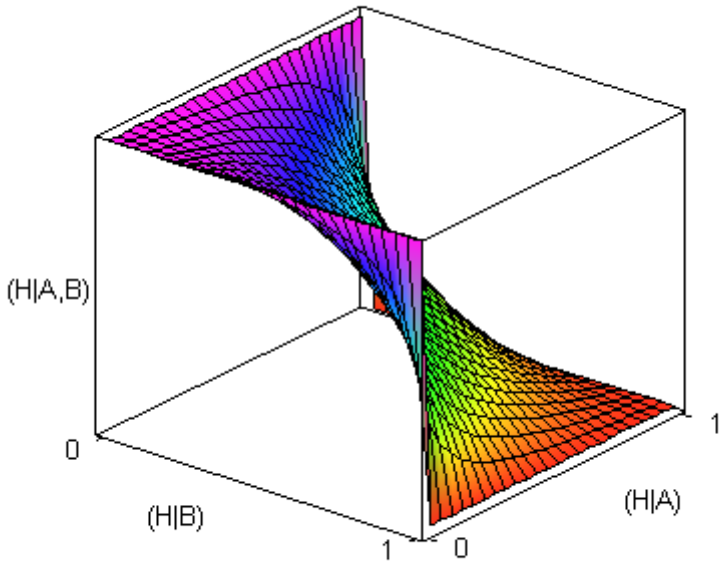
$$
\begin{aligned}
H &\leftarrow A \\
H &\leftarrow B
\end{aligned}
$$

$$
\begin{aligned}
(H|A,B) &= \frac{(A,B|H)(H)}{(A,B)} && \text{Bayes law} \\
&= \frac{(A|H)(B|H)(H)}{(A,B)} && \text{Naive Bayes assumption} \\
&= \frac{(H|A)(A)(H|B)(B)}{(A,B)(H)^{2-1}} && \text{Bayes law again} \\
&= \frac{(H|A)(H|B)(A)}{(A|B)(H)} \\
&= \frac{(H|A)(H|B)(A)}{[(A|H)(H|B)+(A|\neg H)(\neg H|B)](H)} \\
&= \frac{(H|A)(H|B)}{(H|A)(H|B)+\frac{(\neg H|A)(\neg H|B)(H)}{(\neg H)}}
\end{aligned}
$$

If we don't have the prior (H), we may assume it is 0.5, thus $(H) = (\neg H)$ and:

$$(H|A,B) = \frac{(H|A)(H|B)}{(H|A)(H|B)+(\neg H|A)(\neg H|B)}$$

and the plot is:



Case $n = 3$

$$
\begin{aligned}
H &\leftarrow A \\
H &\leftarrow B \\
H &\leftarrow C
\end{aligned}
$$

$$
\begin{aligned}
(H|A,B,C) &= \frac{(A,B,C|H)(H)}{(A,B,C)} && \text{Bayes law} \\
&= \frac{(A|H)(B|H)(C|H)(H)}{(A,B,C)} && \text{Naive Bayes assumption} \\
&= \frac{(H|A)(A)(H|B)(B)(H|C)(C)}{(A,B,C)(H)^2} && \text{Bayes law again} \\
&= \frac{(H|A)(H|B)(H|C)(A)(B)(C)}{(A|B,C)(B,C)(H)^2} \\
&= \frac{(H|A)(H|B)(H|C)(A)(B)(C)}{[(A|H)(H|B,C)+(A|\neg H)(\neg H|B,C)](B,C)(H)^2} \\
&= \frac{(H|A)(H|B)(H|C)(B)(C)}{[\frac{(H|A)(H|B,C)}{(H)}+\frac{(\neg H|A)(\neg H|B,C)}{(\neg H)}](B,C)(H)^2} \\
&= \frac{(H|A)(H|B)(H|C)(B)}{[\frac{(H|A)(H|B,C)}{(H)}+\frac{(\neg H|A)(\neg H|B,C)}{(\neg H)}](B|C)(H)^2} \\
&= \frac{(H|A)(H|B)(H|C)(B)}{[\frac{(H|A)(H|B,C)}{(H)}+\frac{(\neg H|A)(\neg H|B,C)}{(\neg H)}][(B|H)(H|C)+(B|\neg H)(\neg H|C)](H)^2} \\
&= \frac{(H|A)(H|B)(H|C)}{[\frac{(H|A)(H|B,C)}{(H)}+\frac{(\neg H|A)(\neg H|B,C)}{(\neg H)}][\frac{(H|B)(H|C)}{(H)}+\frac{(\neg H|B)(\neg H|C)}{(\neg H)}](H)^2} \\
&= \frac{(H|A)(H|B,C)}{(H|A)(H|B,C)+\frac{(\neg H|A)(\neg H|B,C)(H)}{(\neg H)}}
\end{aligned}
$$

which suggests a recursive pattern.

2.7.6 Code

**** Propagate messages up the proof tree
INPUT: a new list of solutions arrives at the current node
RETURN: nothing (update solutions in proof tree)
Algorithm:
0. IF we have reached root of the proof tree:
   return a list of new solutions
1. IF parent is a sub-goal:
   here we potentially have multiple answers to the sub-goal...
3. First, determine if any new solution is potentially competing with existing ones
   (ie, referring to the same instance)
4. IF a solution is not competing with others, send it up to parent
5. ELSE apply mixture rule, such as NARS' rule or Abram's rule; send result to parent
**** Note: In the current code, we don't do #3-5.
**** We simply send multiple solutions to the parent, even if they are conflicting.
6. IF parent is a rule:
   try to apply the rule if all required arguments in the conjunction are available;
   ie, match the newly added sub against the subs of sibling nodes
7. IF a consistent sub is found:
8. calculate: message = local factor * all incoming messages
   send the result message to parent node
   (it is impossible for the parent to be a fact)
9. recurse to parent

```
(defun propagate (node new-solutions)
  (****DEBUG 1 "propagate: evaluating node: data = ~a" (node-data node))
  (setf parent    (parent node))
  ;; Add the new solutions to the current node
```

```
(setf (solutions node) (append (solutions node) new-solutions))
;; Reached root node?
(if (null parent)
  (progn
    (****DEBUG 1 "propagate: got solution = ~a" (print-solutions new-solutions))
    (return-from propagate)))
(setf parent-data (node-data parent))
;; Is parent a sub-goal?
(if (listp parent-data)
  ;; 3-6. Simply send all solutions to parent
  ;;      It seems that only the new solutions need to be sent up
  (propagate parent new-solutions)
  ;; ELSE parent is a rule:  try to apply the rule
  (let* ((rule     parent-data)
         (cR       (confidence rule))
         (theta    (car (parameters rule)))
         (literals (literals   rule)))
    (****DEBUG 1 "propagate: evaluating rule... op = ~a" theta)
    (****DEBUG 1 "propagate: rule type = ~a" (node-type parent))
    (if (equal (node-type parent) #\H)
      (tagbody
      case1A
        ;; case 1A: all Bi's are available
        ;; The strategy is to apply the AND/OR *pairwise*, sequentially.
        (setf c_list (cdr (parameters rule)))          ; List of parameters c1,c2,...
        (setf c1 (car c_list))                         ; Set c1
        (setf c_list (cdr c_list))                     ; Point to next element
        ;; First set up the left operand, X1
        (setf X1 (car literals))                       ; X1 = the first literal
        (if (eql X1 node)                              ; Is it the node with new solutions?
            (setf X1-solutions new-solutions)          ; If so, process new-solutions *only*
          (setf X1-solutions (solutions X1)))          ; If not, get the set of old solutions
        (prepare-left-operand c1 X1-solutions)         ; Prepare X1
        (****DEBUG 1 "propagate: literals = ~a" literals)
        ;; For each of the rest of siblings, X2:
        (dolist (X2 (cdr literals))
          ;; If some silbing has null TV, this means case 1A is broken, try case 1B
          (if (null (solutions x2))
            (go case1B))
          ;; If this is a "head" literal, it means the end of body literals, signals success
          (if (equal (node-type X2) #\H)
            (return))
          (if (eql X2 node)                            ; Is it the node with new solutions?
              (setf X2-solutions new-solutions)        ; If so, process new-solutions *only*
            (setf X2-solutions (solutions X2)))        ; If not, get the set of old solutions
          (setf c2 (car c_list))                       ; Set c2
          (setf c_list (cdr c_list))                   ; Point to next element
          ;; Merge with sibling's sequence:
          (setf X1-solutions
            (merge-solutions c2 X1-solutions X2-solutions))
          ;; Abduction mode?  **** TO-DO: abduction is unfinished
          (if *abduct-mode*
            ;; record the 2 arguments as parts of a potential explanation
            (setf *current-explanation* (list op *current-explanation* arg))))
    ;; If new-solutions is not empty
        (if (not (null X1-solutions))
          (progn
            ;; apply "theta":  result = (1 - theta) * AND-factor + theta * OR-factor
            (apply-theta theta X1-solutions)
            (****DEBUG 1 "propagate: got first TV = ~a" (message (car X1-solutions)))
            ;; send new-solutions to parent node;  recurse
            (propagate parent X1-solutions)))
        (return-from propagate)
      case1B
        ;; Case 1B:  apply Bayes rule (as in case 2) and then do forward (as in case 1A)
        (****DEBUG 1 "propagate: case 1B")
      )
      ;; case 2:  reverse direction; apply Bayes rule
      (progn
        (****DEBUG 1 "propagate: case 2")
        (setf c_list (cdr (parameters rule)))          ; List of parameters c1,c2,...
        (setf c1 (car c_list))                         ; Set c1
        (setf c_list (cdr c_list))                     ; Point to next element
        ;; First set up the left operand, X1
        (setf X1 (car literals))                       ; X1 = the first literal
        (if (eql X1 node)                              ; Is it the node with new solutions?
            (setf X1-solutions new-solutions)          ; If so, process new-solutions *only*
          (setf X1-solutions (solutions X1)))          ; If not, get the set of old solutions
        (prepare-left-operand c1 X1-solutions)         ; Prepare X1
        ;; For each of the rest of siblings, X2:
        (dolist (X2 (cdr literals))
          ;; If some silbing has null TV, this means case 2 is broken
          (if (null (solutions X2))
            (return-from propagate))
```

```
            ;; break out of loop if "head" literal is reached
            (if (equal (node-type X2) #\H)
              (return))
            (if (eql X2 node)                          ; Is it the node with new solutions?
                (setf X2-solutions new-solutions)      ; If so, process new-solutions *only*
              (setf X2-solutions (solutions X2)))      ; If not, get the set of old solutions
            (setf c2 (car c_list))                     ; Set c2
            (setf c_list (cdr c_list))                 ; Point to next element
            ;; Merge with sibling's sequence:
            (setf X1-solutions
              (merge-solutions c2 X1-solutions X2-solutions)))
          ;; Now find a head with solution

          ;; If new-solutions is not empty
          (if (not (null X1-solutions))
            (progn
              ;; apply "theta":  result = (1 - theta) * AND-factor + theta * OR-factor
              (apply-theta theta X1-solutions)
              (****DEBUG 1 "propagate: got first TV = ~a" (message (car X1-solutions)))
          ;; send new-solutions to parent node;  recurse
              (propagate parent X1-solutions))))
))))
```

```
;;; Merge two lists of solutions
;;; INPUT:   two lists of solutions
;;;          if merging is OK, perform sum-product operation
;;; OUTPUT:  new list of merged solutions
(defun merge-solutions (c2 soln-list1 soln-list2)
  (setf result-solution-list nil)
  ;; Pick an element from list1
  (dolist (solution1 soln-list1)
    ;; Add the following results to the total results
    (setf result-solution-list (nconc result-solution-list
      ;; Try to merge element1 with list2 elements
      (merge-single-solution c2 solution1 soln-list2))))
  ;; return the results
  result-solution-list)

;;; Merge a single solution against a list of solutions
(defun merge-single-solution (c2 solution1 soln-list2)
  (setf soln-list0 nil)
  (dolist (solution2 soln-list2)
    (block outer-loop
      (setf sub1 (sub solution1)
            sub2 (sub solution2))
      ;; Merge sub1 and sub2 to give sub0, avoiding duplicates
      ;;   First add to sub0 elements in sub1 that are not in sub2
      (setf sub0 nil)
      (dolist (pair1 sub1)
        (dolist (pair2 sub2)
          ;; var1 and var2 are different
          (if (not (equal (car pair1) (car pair2)))
            ;; add the pair to the result (sub0)
            (push pair1 sub0)
            ;; var1 and var2 are same:
            ;; sub1 and sub2 NOT equal?
            (if (not (equal (cdr pair1) (cdr pair2)))
              ;; break from the loop
              (return-from outer-loop)
              ;; otherwise omit this pair, do nothing
              ))))
      ;; Then add the remaining elements in sub2 to sub0
      (setf sub0 (nconc sub0 sub2))
      ;; now sub0 has the new merged sub
      ;; calculate:  X0 = sum-product X1 X2
      (setf x0 (sum-product c2 (message solution1) (message solution2)))
      ;; Create solution0 with the new X0
      (setf solution0 (make-instance 'solution
                                     :sub sub0
                                     :message x0))
      ;; add solution0 to soln-list0
      (push solution0 soln-list0)))
      ;; return the new list
      soln-list0)
```

3 First-order logic

3.1 Unification

3.2 Substitution management

4 Inductive learning

Bibliography

Kschischang. Factor graphs and the sum-product algorithm. 2001.

Pearl. *Probabilistic Reasoning in Intelligent Systems - Networks of Plausible Inference*. Morgan Kaufmann, California, 1988.