BAYESIAN REASONING TOOL

Deliverable No. A002

Prepared by:
JAYCOR

Prepared for:
Naval Research Laboratory
Washington, DC 20375-5000

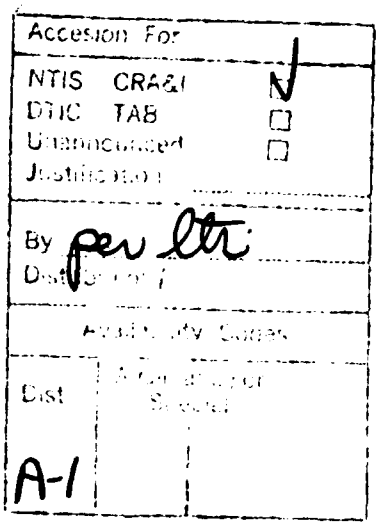
In Response to:
Contract #N00014-86-C-2352

26 May 1988

Bayesian Reasoning Tool

The work performed to meet the requirement of this task is a continuing effort, evolving toward a general purpose reasoning tool. A general purpose Bayesian reasoning tools has been implemented and all sources and documentation are attached. The documentation gives an in-depth description of the inference engine and how to use it.

The tool is written in Common-Lisp. The core functions can be used on any machine with a Common-Lisp. User interface is written for SYMBOLICS and SUN work-stations. The sources reside in the directory /usr/prj/bart/version1.0/ on NRL-SUN7 and in the directory Sy1:>bart>version1-0> on the NRL-SYM1.



BaRT Manual
Preliminary Version 1.0

Naveen Hota, Connie Loggia Ramsey and Lashon B. Booker

Abstract

An inference engine has been developed to aid in classification problem solving. This tool is the belief maintenance component of an expert system shell under development. The inference engine uses Bayesian reasoning and can handle problems associated with incomplete and uncertain evidence. It has successfully been used to perform ship classification. This manual describes this inference engine, and provides some of the theoretical background for this work.

1. Introduction

Many real world problems are associated with uncertainty; the evidence people observe which helps them to reason about some goal event is almost always uncertain and incomplete. Still, people make judgements based on this uncertain and incomplete evidence. These uncertain evidences can be combined in various ways to find the validity or strength of a hypothesis [7,9], and Bayesian probability theory is a *normative* theory that allows one to reason about and combine uncertainties. Pearl has devised a way to represent, reason about and combine uncertain evidences in a way that conforms to the tenets of probability theory, but avoids the disadvantages usually associated with probabilistic computations of belief [7]. BaRT is a Bayesian Reasoning Tool which implements Pearl's ideas. It has been implemented as an AI programming environment which can perform classification problem solving, and it has been used to classify ships [2]. Section 2 provides an overview of the theoretical background for this work. Section 3 explains how to use BaRT and provides an example. Sections 4 and 5 provide details concerning the implementation of this system.

This manual describes a preliminary version of a system which is under development. Later versions of BaRT will have greater capabilities, so any of the functions and capabilities described here are subject to change.

2. Belief Network

Pearl's framework provides a method for hierarchical probabilistic reasoning in directed, acyclic graphs called belief networks. Each node in the network represents a discrete-valued propositional variable which describes an aspect of the domain, and each node contains information about both the current belief of each value of the proposition and the most probable instantiation of the proposition given the evidence available, called the belief* distribution. Each link between two nodes represents a direct causal dependence between two of the propositions, and the directionality of the link is from cause to manifestation. Each link contains a matrix of probabilities conditioned on the states of the causal variable. It is important to note that numbers used to quantify the links do not have to be probabilities. All that is required is that the matrix entries are correct relative to each other.

The belief updating scheme keeps track of two sources of support for belief at each node: the diagnostic support derived from the evidence gathered by descendants of the node and the causal support derived from evidence gathered by parents of the node.

Diagnostic support (λ) provides the kind of information summarized in a likelihood ratio for binary variables. Causal support (π) is the analogue of a prior probability, summarizing the background knowledge lending support to a belief. These two kinds of evidential support are combined to compute the belief at a node with a computation that generalizes the odds/likelihood version of Bayes' rule. Each source of support is summarized by a separate local parameter, which makes it possible to perform diagnostic and causal inferences at the same time. These two local parameters (λ and π), together with the matrix of numbers quantifying the relationship between the node and its parents, are all that is required to update beliefs. Incoming evidence perturbates one or both of the support parameters for a node. This serves as an activation signal, causing belief at that node to be recomputed and support for neighboring nodes to be revised. The revised support is transmitted to the neighboring nodes, thereby propagating the impact of the evidence. Propagation continues until the network reaches equilibrium. The overall computation assigns a belief to each node that is consistent with probability theory. Using a similar computation, similar supporting factors (π^* and λ^*) are used to find the belief* distribution. Section 4 provides details concerning the implementation of this belief network, and the equations for belief and belief* updating are presented in Appendix A. The reader is referred to Pearl [7] for more details about the theoretical framework of this belief maintenance system.

3. Using BaRT

The system is implemented in Portable Common Loops (PCL) on top of Common LISP. A graphic interface is provided on Symbolics and Suns showing the network (the nodes and their relations), the beliefs, the support parameters, and the dynamic propagation of beliefs as new evidence is obtained. Each node is represented as a rectangle in the graph with a histogram in it representing the belief in the individual values of the proposition.

To build the network for a particular problem, the user must provide the specific information about the nodes and the links. This information is declared in a data file which is presented to the program. The nodes and links should be defined in the data file as shown below.

Declare each node and link in the network using the macro *mmake*. Mmake takes two arguments and some keyword arguments. The first argument is the name to be given to the object being created. The second argument is the type of the object to be created; possible values are *node* and *link*. Keyword arguments differ depending on the type of object to be created, and all possible keyword arguments are given below for both types node and link. Text in italics is to be replaced by the user. To create an object of type node, use mmake as follows:

(mmake '*nodename* 'node
:doc *doc-string*
:i-name *display-name*
:node-values '(*val1 val2 val3 .. valn*)
:prior '*prior-probability*)

where

doc-string is a documentation string explaining the proposition. The default value for this is "noname".

display-name is a string of not more than 9 characters representing the name of the node. This is used while displaying the node on the screen graphically. The default value is "noname".

val1, val2..valn are the possible values the proposition can take. The default value for this is (true false).

prior-probability is a list of the prior probabilities of the values of the proposition. This is only needed for the top nodes.

Besides the keywords mentioned above, the object *node* has another keyword *ext-evid* representing the external evidence for that proposition. Initially the value of ext-evid is nil, indicating no external evidence for the proposition. Whenever new evidence concerning a particular node is observed, this information can be added to the ext-evid slot of that node so the impact of this new evidence can be propagated in the network.

Similarly mmake is used to create an object of type link:

(mmake '*link-name* 'link
:i-name '*display-name*
:tnode '*top-node-name*
:bnode '*bottom-node-name*
:indpro '*conditional-probability-matrix*)

where

link-name is the name of the link from the top node A to the bottom node B represented by A -> B.

display-name is a string of not more than 9 characters representing the name of the link. This string represents the link while describing the link coefficients λ and π pictorially. The default value is "noname".

top-node-name is the top node or the causal node of the link: node A.

bottom-node-name is the bottom node or the manifestation node of the link: node B.

conditional-probability-matrix is the matrix quantifying the relationship between the causal node (A) and manifestation node (B). This is a list of lists of the form

$$((P[B_1 \mid A_1] \;\; P[B_1 \mid A_2] \;\; .. \;\; P[B_1 \mid A_n])$$
$$(P[B_2 \mid A_1] \;\; P[B_2 \mid A_2] \;\; .. \;\; P[B_2 \mid A_n])$$
$$.....$$
$$(P[B_m \mid A_1] \;\; P[B_m \mid A_2] \;\; .. \;\; P[B_m \mid A_n]))).$$

Each element $P[B_i \mid A_j]$ is the probability that the proposition B is equal to the value B_i given that the proposition A is equal to the value A_j.

An example which shows how to create a network is provided at the end of this section.

3.1. Running the Program

A graphic interface has been developed for Symbolics and Suns. In order to run the program without the graphic interface, the reader should see Appendix B.

1. Invoking PCL and Loading the System Definitions and the System:

Symbolics: Get into a Common LISP environment which supports PCL. From the LISP listener, load the system definitions by loading the file *bart-defsys* which is in the *src* subdirectory of the *bart* directory. Then load the system with the command *(bart-util::load-bart)*. Change to package *bart-frame* with the command *(in-package 'bart-frame)*. Now, invoke the program by first pressing the *Select* key and then pressing the *Symbol, Shift* and *B* keys simultaneously.

Sun: Invoke suntools, and go to the *src* subdirectory of the *bart* directory. Then type *run-bart* from the shell; this loads the system definitions and the system. Change to package *bart-frame* with the command *(in-package 'bart-frame)*. Now, invoke the program with the command *(bart-command-loop)*.

At this state, the whole screen consists of six windows: the title pane, the belief network window, the global system parameters pane, the command menu pane, the node-link information display pane, and the interaction window. Figure 1 provides a sample screen display.

The title window consists of the heading *Bayesian Reasoning Tool (BaRT)* in boldface.

The belief network display is on the left hand side of the screen occupying a large portion of the screen. This pane is used to display the network which consists of

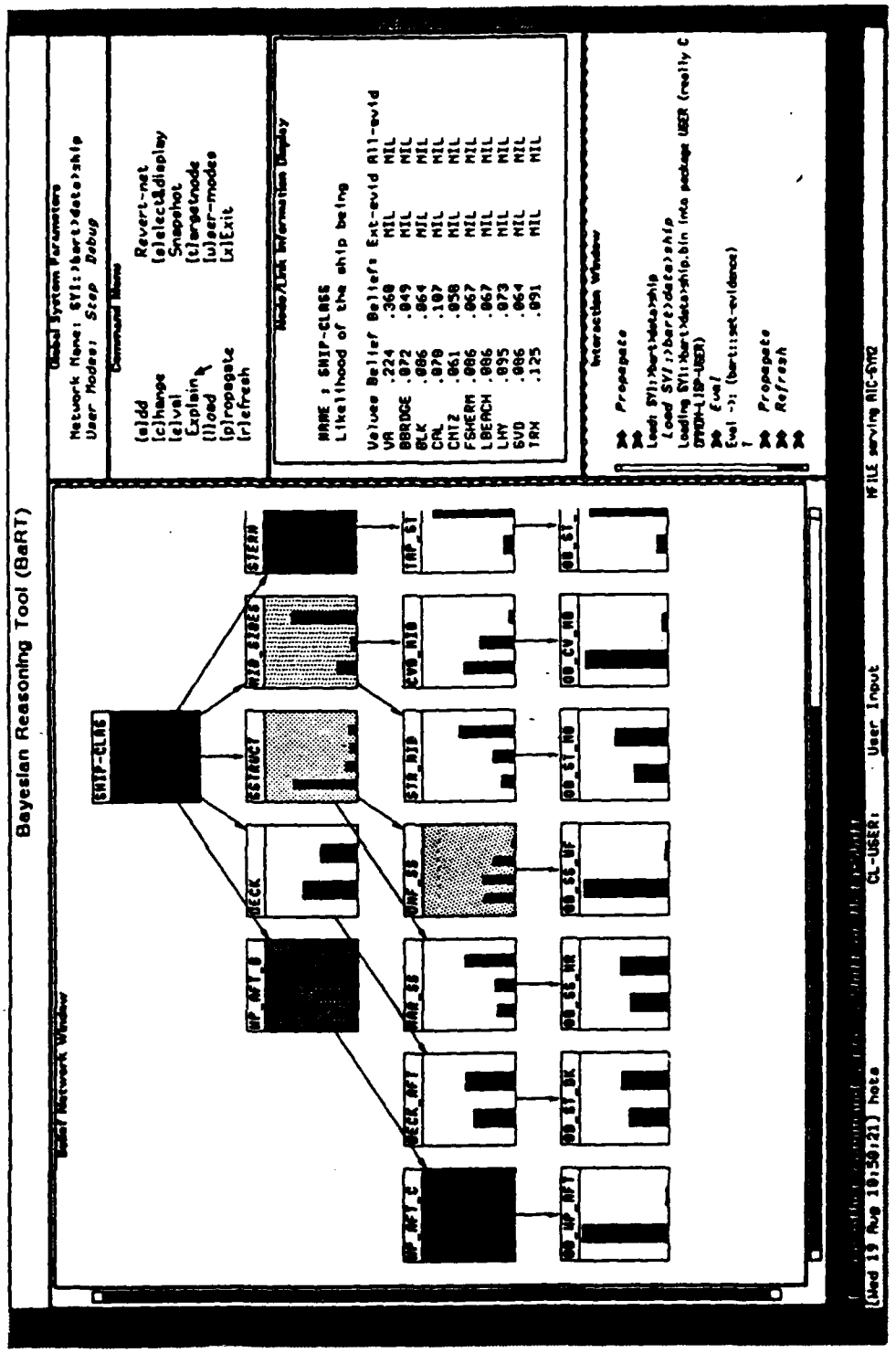


Figure 1-sample screen display

nodes and links, their connectivity, and a histogram in each node indicating the belief of that node. Some of the nodes in the network are grayed; the intensity depends on that node's influence on the target node (selected hypothesis).

The global system parameters pane is on the top right hand side of the screen and consists of two lines. The first line has the data file name. Initially, this slot will be empty before selecting the data file. If the selected data file name is in boldface, then the network is in equilibrium; otherwise the network is not in equilibrium. This distinction is useful when propagating the effect of new evidence in the network in step mode, i.e., updating one node at a time. The second line consists of two flags: *step* and *debug*. These are boolean flags. If the string is in bold letters, then it indicates that the flag is set on; otherwise it is off. Step mode allows the user to see the network update one node at a time. Debug mode is not yet implemented.

The command menu pane is right below the global system parameters pane and consists of the commands in the top level command loop. These are mouse sensitive and can be invoked by mouse-clicking left on them. All mouse-sensitive objects are highlighted when the mouse pointer points to them.

The node-link information display pane is below the command menu pane and is used to present information about nodes/links in the network.

In the bottom right hand corner of the screen is the interaction window for normal interaction. Expressions in this pane can be evaluated by first clicking on *eval* in the command menu pane and then typing the expression to the interaction window.

2. Using the Command Menu:

Choosing Nodes and Links

In order to select a node or a link for certain commands, the user must mouse-click on the node/link. To select a node, the user must **click left on a node.** To select a link on the Symbolics, the user must **click left on a link**. To select a link on the Sun, the user must **click middle on each node** which is connected to the link.

Commands

The available commands are activated by **mouse-clicking left** on them. (On the Symbolics, the user can also activate a command whose first letter is in brackets by typing the first letter of that command.) In addition, brief documentation for each command is provided on the Sun by **mouse-clicking right** on it. The possible choices are:

load :

Prompts for a data file and loads it. It performs all the necessary internal

calculations, brings the network into equilibrium, and displays the network.

change :

Adds new external evidence to a node. This is done by clicking on *change* first and then clicking on the node to which the user wants to add external evidence. Now, a menu appears which has the heading *New External Evidence* followed by the node's name. Next to each value of the proposition is a number summarizing the relative impact of the evidence. These numbers are to be interpreted as components of a likelihood vector having the standard semantics: numbers greater than 1.0 indicate values supported by the evidence, numbers less than 1.0 (but non-negative) indicate values argued against by the evidence, and 1.0 indicates no evidential impact one way or the other. The default is always 1.0. The numeric fields are mouse sensitive and the user can give new external evidence by changing these fields. This is done by mouse-clicking left on that value and then entering a new value followed by a carriage-return. This should be done for each value affected by the evidence. Now, mouse-click left on either *done* to enter the new evidence into the system or *abort* to ignore the change. Once the new evidence is entered, the effect of that evidence can be propagated by clicking on *propagate*.

propagate :

Updates the network and redisplays the information.

select&display :

After choosing this, clicking on any node or link in the network displays the information about that object in the node-link display pane.

targetnode :

After choosing this, clicking on any node in the network indicates the influence of the other nodes in the network on this selected node. The node clicked on, called the target node, is always grayed maximally. The rest of the nodes with decreasing gray levels indicate the strength of the influence of those nodes on the target node. This guides the user to seek evidence of those propositions that contribute most to confirm/reject the hypothesis (target node).

user-modes :

Allows the user to set available options. After clicking on this, a temporary menu of all the user setable options appears. The user can change any of these values. Presently three global options, *step mode, debug mode,* and *clear node link window each time*, appear with their present values. The user can change the values of any of these by clicking on them. This can be terminated by clicking either on *done* to process the request or *abort* to ignore the request. *Step mode* shows the propagation in steps, i.e., the system propagates one node at a time. This is useful if the user wants to see the results after each update of a node. *Debug mode* is not

implemented at present. *Clear node link window each time* determines whether the system will clear the node-link display pane before presenting new information or append the new information. The default is to clear the window each time. On the Sun, if the option is set to append and the buffer becomes full, then this system will automatically clear the old information and present the new information on a fresh window.

revert-net :
Resets the network back to the initial equilibrium state so the user can try a new run with new observations without loading and reinitializing.

refresh :
Refreshes the display.

eval :
Clicking on this makes the LISP reader (displayed in the interaction window) read an expression and evaluate.

exit :
Exits from the program.

add :
Adds nodes and links to the network. Not yet implemented.

explain :
Explains the reasoning. Not yet implemented.

snapshot :
Saves the present environment in a file. Not yet implemented.

3.2. Example

This example was presented in Kim[5]. Say an alarm in a house rings when there is an intrusion or when there is an earthquake. Also, earthquakes are reported on the radio. The nodes and links and their prior probabilities and conditional probabilities for this problem are given below, and the network is shown pictorially in Figure 2. (The information for this network is in the data file called alarm.lisp.)

```
;;;-=-=-=-=-=-=-=-=-= Beginning of data file -=-=-=-=-=-=-=
;;;-=-=-=-=-=-= node information -=-=-=-=
(mmake-instance
  'ALARM 'node
  ':i-name                    "ALARM"
  ':doc                       "Probability of ALARM ringing ")


(mmake-instance
  'BURGLARY 'node
  ':i-name                    "BURGLARY"
  ':doc                       "Probability of BURGLARY "
  ':prior                     '(0.1 0.9))


(mmake-instance
  'EARTH-QUAKE 'node
  ':i-name                    "EARTH-QUAKE"
  ':doc                       "Probability of EARTH-QUAKE "
  ':prior                     '(0.2 0.8))


(mmake-instance
  'RADIO-BROADCAST 'node
  ':i-name                    "RADIO-MSG"
  ':doc     "Probability of EARTH-QUAKE indicated by RADIO-BROADCAST ")


;;; -=-=-=-=-=-= link information -=-=-=-=-=-=

(mmake-instance
  'lk-BURGLARY->ALARM 'link
  ':tnode                     'BURGLARY
  ':bnode                     'ALARM
  ':indpro                    '((0.7 0.1) (0.3 0.9)))
;;; where
;;; Probability[ ALARM=true | BURGLARY=true] = 0.7
;;; Probability[ ALARM=true | BURGLARY=false] = 0.1
;;; Probability[ ALARM=false | BURGLARY=true] = 0.3
;;; Probability[ ALARM=false | BURGLARY=false] = 0.9
```

```
(mmake-instance
  'lk-EARTH-QUAKE->ALARM 'link
  ':tnode                    'EARTH-QUAKE
  ':bnode                    'ALARM
  ':indpro                   '((0.2 0.1) (0.8 0.9)))
;;; where
;;; Probability[ ALARM=true | EARTH-QUAKE=true] = 0.2
;;; Probability[ ALARM=true | EARTH-QUAKE=false] = 0.1
;;; Probability[ ALARM=false | EARTH-QUAKE=true] = 0.8
;;; Probability[ ALARM=false | EARTH-QUAKE=false] = 0.9


(mmake-instance
  'lk-EARTH-QUAKE->RADIO-BROADCAST 'link
  ':tnode                    'EARTH-QUAKE
  ':bnode                    'RADIO-BROADCAST
  ':indpro                   '((0.8 0.001)
                             (0.2 0.999)))
;;; where
;;; Probability[ RADIO-BROADCAST=true | EARTH-QUAKE=true] = 0.8
;;; Probability[ RADIO-BROADCAST=true | EARTH-QUAKE=false] = 0.001
;;; Probability[ RADIO-BROADCAST=false | EARTH-QUAKE=true] = 0.2
;;; Probability[ RADIO-BROADCAST=false | EARTH-QUAKE=false] = 0.999

;;;-=-=-=-=-=-=-=- End of file -=-=-=-=-=-=-=-
```

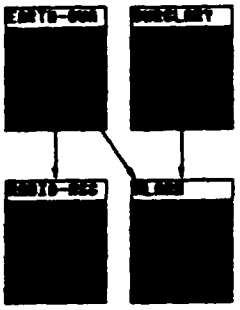


Figure 2 - Network

To run BaRT with this data file, do the following:

Invoke BaRT as described in Section 3.1. Click on *load*. This prompts for the data file name. Type the data file name. When the program brings the network into equilibrium initially, it picks an arbitrary node and makes that the target node. At

this stage we do not have any external evidence, so information on any node displays nil in the external evidence field. Now click on *change* and then click on the node *alarm*. Change the external evidence of that node by replacing the "1.0 fields" with "1" (to show that it is true that the alarm is ringing) and "0" (to indicate that it is false that the alarm is not ringing) in the pop up window that appears after clicking on the alarm node. Once the evidence is given, click on *propagate*. This propagates the new evidence in the network, changing the beliefs of the nodes, and brings the network into equilibrium. Click on *select&display* and then click on a node or a link to see information about that node or link in the node/link display window. The evidence that the alarm is ringing would actually increase the belief of both burglary and earthquake. Now change the external evidence fields of radio broadcast to "1" and "0". This change would result in a further increase in the belief of earthquake and reduces the belief of burglary.

To run the program again with the initial settings, click on *revert-net* and proceed.

4. Implementation

BaRT is implemented in PCL on top of Common LISP. A graphic interface is provided on Symbolics and Suns showing the network (the nodes and their relations), the beliefs, the coefficients, and the dynamic propagation of beliefs as new evidence s obtained. Each node is represented as a rectangle in the graph with a histogram in it representing the belief of that node.

Two generic classes *node* and *link* are defined. Instances of node represent propositions of the domain and instances of link represent the relationship between propositions. The complete definitions of class *node* and class *link* are presented below.

```
(defclass node ()
  ((i-name "noname")
   (doc nil)
   (node-values '(True False))      ;values node can take, by default binary
   (rank nil)                       ;number of values for this node
   (ent 0)                          ;entropy of the node
   (imp 0)                          ;importance of the node
   (relative-x 0)                   ;relative x-axis position
                                    ;of the node in the network
   (relative-y 0)                   ;relative y-axis position
                                    ;of the node in the network
   (rel-ben 0)                      ;relative importance of the node.
   (tlinks nil)                     ;top links – list
```

```
(blinks nil)                ;bottom links – list
(unit-vec nil)              ;vector of 1's for internal calculation
(prior nil)                 ;prior probabilities – list
(ext-evid nil)              ;external evidence – list of lists
(init-prior nil)            ;initial priors – list
(parranks nil)              ;parent's ranks – list of integers
(parprobs nil)              ;parent's probabilities – list of lists
(condpro1 nil)              ;conditional probabilities – list of lists
(condpro2 nil)              ;transpose of condpro1
(belief nil)                ;belief – list
(bel* nil)                  ;belief* for explanations – list
(init-belief nil)           ;initial belief – list
(init-bel* nil)             ;initial belief* – list
(parents nil)               ;parents – list
(children nil))             ;children – list
(:accessor-prefix nil))
```

Following is a brief description of each slot within class *node*. Each description has External/Internal, type of value, and other information. External means the user can give a value to that slot. Internal slots are those which are used by the program internally to cache some values, etc.

i-name

External. String.
The name which appears above the node in the graphic representation of the network. Default value is "noname".

doc

External. String.
Description of the proposition attached to the node. This string would appear when describing the belief and the belief*. Default value is "noname".

node-values

External. List.
The values the proposition can take. Default value of this slot is (true false).

rank

Internal. Integer.
Cardinality of the node, i.e., the number of values that the proposition can take.

relative-x, relative-y

Internal. Integer.
The relative x and y positions of the node in the network.

ent, imp

Internal. Real number.

Entropy and importance of a node. Used to find the relative benefit factors described below. Refer to Kim [5] for the formulas on how to calculate these values.

rel-ben

Internal. Real number.

Node's relative benefit factor: the product of the importance and entropy. This is a measure of the influence of the individual nodes in the network on the selected target node. This is indicated visually by the gray level.

tlinks, blinks

Internal. List of link names.

The node's top and bottom links.

parents, children

Internal. List.

Lists of the parents and the children of the node.

prior

External. List of real numbers.

The prior probabilities of the values of the proposition. Valid only for top nodes. Default value is a list whose elements are 1/rank of the node. The length of this list is the rank of the node.

ext-evid

External. List of sublists.

All of the external evidence for a node.

unit-vec

Internal. List.

A unit vector of length equal to the rank of the node.

parranks

Internal. List of integers.

A list of the ranks of the node's parents.

parprobs

Internal. A list of sublists.

A list of the individual conditional probabilities of all of the top links.

condpro1

Internal. A list of sublists.

This a tensor representing the joint conditional probabilities of the node with respect to its parents. This value is calculated internally from the individual conditional probabilities of the node's top links assuming conditional independence between parents.

condpro2

Internal. A list of sublists.

This is the matrix transpose of condpro1.

belief

Internal. A list of real numbers.

Belief distribution of the proposition.

*bel**

Internal. A list of real numbers.

This is the belief distribution for asserting the categorical value of the proposition.

*init-prior, init-belief, init-bel**

Internal. List of real numbers.

The values of the prior/belief/belief* slots at the beginning of the program so they can be copied into the slots prior/belief/belief* to reset the program for a fresh run.

```
(defclass link ()
 ((i-name "noname")
  (tnode nil)                    ; node above the link
  (bnode nil)                    ; node below the link
  (indpro nil)                   ; conditional prob for the link
                                 ;  – list of lists
  (link-lambda nil)              ; lambda's – list
  (init-lambda nil)              ; initial lambda's – list
  (link-lambda* nil)             ; lambda*'s – list
  (init-lambda* nil)             ; initial lambda*'s – list
  (link-pi nil)                  ; pi's – list
  (init-pi nil)                  ; initial pi's – list
  (link-pi* nil)                 ; pi*'s – list
  (init-pi* nil)                 ; initial pi*'s – list
  (mu-info 0))                   ; mutual information
  (:accessor-prefix nil))
```

i-name

External. String.

This string represents the link while describing the link coefficients λ and π pictorially.

tnode, bnode

External. Atom.

Top/Bottom node's name which the link is connected to. The user must provide this.

indpro

External. List of sublists.

This the conditional probability matrix representing the relation between the top and bottom node of the link. The user must provide this.

link-lambda, link-pi, link-lambda, link-pi**

Internal. List of real numbers.

These are the diagnostic and causal coefficients supporting the propositions that the link is connected to.

init-lambda, init-pi, init-lambda, init-pi**

Internal. List of real numbers. These are the values of λ, π, λ^*, and π^* at the beginning of the program. These are copied into the λ, π, λ^*, and π^* slots to reset the program for another run.

mu-info

Internal. Real number.

Represents the mutual information of the link. This is used in calculating the importance of a node with respect to a selected target node. For the formula, refer to Kim[5].

Each node carries the following information: a vector containing the possible values which the proposition can hold, a vector which stores the belief distribution of that node, a vector which stores the belief* distribution used for the categorical assertion of that node, a vector containing the the prior probabilities of the values of the proposition, and a tensor containing the joint conditional probabilities which represent the relationship between the node and all of its parents (see Appendix A for a definition of *tensor*). The number of elements in the belief, belief*, and prior probability vectors is equal to the number of values that the proposition can take (the rank), and each element represents the belief/belief*/prior probability of the corresponding element in the values vector. The prior probability vector is only needed for the top most nodes where there will not be any parents. The joint conditional probability tensor is needed for all the nodes except for the top nodes, and it is constructed from the information in the individual

conditional probability matrices associated with each parent link of a node. This tensor is of order (n + 1) with indices ijk..l where i is the rank of the node, j,k,..l are the ranks of its parents, and n is the total number of parents. The computation of the tensor, which assumes conditional independence, uses the following formula:

$$P[X_i \mid U_{1_j}, U_{2_k}, ..., U_{n_l}] = P[X_i \mid U_{1_j}] * P[X_i \mid U_{2_k}] * ... * P[X_i \mid U_{n_l}]$$

where X is a node and U_1, U_2, ..., U_n are all of its parents

$P[X_i \mid U_{1_j}, U_{2_k}, ..., U_{n_l}]$ is the probability of $X = X_i$ given that $U_1 = U_{1_j}$, $U_2 = U_{2_k}$, ..., $U_n = U_{n_l}$,

$P[X_i \mid U_{1_j}]$ is the probability of $X = X_i$ given that $U_1 = U_{1_j}$,

$P[X_i \mid U_{2_k}]$ is the probability of $X = X_i$ given that $U_2 = U_{2_k}$, ...

$P[X_i \mid U_{n_l}]$ is the probability of $X = X_i$ given that $U_n = U_{n_l}$,

X_1, X_2, ..., X_m are the possible values of X,

U_{1_1}, U_{1_2}, ..., U_{1_p}, are the possible values of U_1,

U_{2_1}, U_{2_2}, ..., U_{2_q}, are the possible values of U_2, ...,

and U_{n_1}, U_{n_2}, ..., U_{n_r}, are the possible values of U_n.

Each link connects two nodes (a causal or top node and a manifestation or bottom node) and carries the following information: a vector λ which indicates the diagnostic support from the bottom node, a vector π which indicates the causal support from the top node, a vector λ^* which represents the diagnostic support for the belief* distribution, a vector π^* which represents the causal support for the belief* distribution, and a conditional probability matrix which quantifies the relationship between the cause and manifestation nodes present on a link. λ, λ^*, π and π^* are all of degree n where n is the rank of the causal node. The conditional probability matrix is made up of i rows and j columns where j is the number of values the causal node connected to this link would take and i is the number of values the manifestation node would take. Each element of the matrix Mij represents the probability of the manifestation node being the ith value given that the causal node is the jth value.

Each node has procedures attached to it. *Update* is one such procedure that updates the belief and belief* of a node. When new evidence is obtained for a node, it can be propagated in the network by changing the incoming λ (or λ^*) or π (or π^*) at the node [λ (or λ^*) in the case of a manifestation, π (or π^*) in the case of a cause]. The system immediately detects the introduced inconsistency (i.e., the difference between the old coefficients and the current ones) and updates the node by calculating the new belief (or belief*) vector using all the incoming λs (or λ^*s), πs (or π^*s) and the combined conditional probability matrix. The new coefficients which will be sent to neighboring nodes are also calculated. Now, if the new coefficients sent to neighboring nodes are different from the old coefficients, then these nodes are updated also. This propagation continues until there is no further change in the coefficients, and the network reaches equilibrium.

This procedure is described in more detail in the following paragraphs.

In each update of the belief of a node, two variables, effective λ and effective π of the node, are calculated using all incoming coefficients (the λs and πs of all the links connected to the node). Effective λ is the term product of all the incoming λs. Effective π is the tensor product of the combined conditional probability tensor and the outer product of all incoming πs. Then the ratio of the belief is calculated as the term product of the effective λ and the effective π. Absolute belief is obtained by normalizing this ratio with respect to 1. After calculating the belief, updating involves calculating the new coefficients (πs and λs) for all the links of the node. The new coefficient of a link is the belief of the node supported by all the incoming πs and λs except that particular λ or π of the link for which the new coefficient is being calculated. New πs of a link are calculated by taking the term quotient of the new belief of the node and the incoming λ of that link. The new λ of a link is the matrix product of the term product of all λs and the tensor product of the combined conditional probability tensor and the outer product of all the πs except that particular π that is associated with the link for which the new λ is being calculated. Once these new coefficients are calculated, the updating procedure involves comparing these new coefficients against the old ones and finding out which link's coefficients are changed. If a change is detected, then the neighboring nodes at the other end of these links must be updated. The new nodes are updated, and this propagation continues until there is no further change in the coefficients, and equilibrium is reached. The equations for updating belief are shown in Appendix A.

The computation of belief* is slightly different from belief updating. In belief updating, individual support from all of the node's neighbors is added whereas in belief* updating, these individual supports are maximized. As shown in the equations in Appendix A, this can be achieved by replacing the first operator + used in the inner product by the operator *max* and replacing all πs with π*s and λs with λ*s. A fuller discussion of the belief and belief* updating equations used in BaRT is given in Booker [3]. For more details refer to Pearl[4,6,7,8].

All the basic procedures that have been introduced so far allow one to update the network and bring it into equilibrium. All of these core functions are defined in a package called *bart* which is in the file named *bart*. This does not have any interface and can be used on any machine. The user interface (machine dependent) is developed for two machines: Symbolics and Suns. The interface code for the Symbolics is in the file *bart-frame-3600* and for the Sun it is in *bart-frame-sun*. Anything a particular programmer wants to add can be added here in *bart-frame-XXX*. The name of the package in these files is *bart-frame*. All the general utilities are in the package *bart-util* which resides in the file *bart-util*. To use the system currently, the user must provide all of the information about the nodes and links in a data file. A knowledge acquisition module to accept this information dynamically would be more elegant and will be implemented in the future. Presently, four sample data files are being used and they are in the directory

called *data* in the bart directory.

5. Interesting Implementation Details

The following paragraphs discuss some of the implementation details which affect the system's efficiency.

• The update procedure can be invoked recursively to update all of the nodes. However, this is inefficient because of the large stack spaces that it would have to maintain. Instead the global procedure *updateall* first updates a node and then places the nodes that are returned on a list so it can later update these new nodes. This way the recursive overhead is avoided. Choosing which node to update or, alternatively, where to place the freshly returned to-be-updated nodes in the global list can also affect the program's efficiency. When placing a node in the global list, it is moved to the end of the list if it was already in there. Since the updating procedure takes one node at a time from the beginning of this global list, the updating of an affected node is postponed as long as possible, resulting in fewer updates to reach equilibrium when multiple evidence is available at the same time.

• While constructing the network initially, joint conditional probabilities at a node are approximated by assuming conditional independence between the parents. This is calculated by using the formula:

$$P[X_i \mid U_{1_j}, U_{2_k}, ..., U_{n_l}] = P[X_i \mid U_{1_j}] * P[X_i \mid U_{2_k}] * ... * P[X_i \mid U_{n_l}]$$

for a node X with all of its parents U_1, U_2, ..., U_n. (This formula was described in Section 4.) Since the calculations for the joint conditional probabilities are large and are the same throughout the run as long as the connectivity is fixed, the joint conditional probabilities are saved at each node after approximating. Even though this requires more memory, this saves in run time.

• The coefficients λ and π represent their support as a ratio. After finding the new values of each coefficient during the updating of a node, these coefficients are normalized before they are stored at the appropriate links. Since the old and new values of the coefficients are compared to determine whether there has been a change, comparing the normalized values allows the system to avoid duplicate updates, and the system can attain equilibrium more quickly.

• The coefficients are real numbers and are represented as floating numbers. Comparing these real numbers using the built-in "equal" function compares them to the last decimal digit (16th). Instead, assuming a small error, and comparing the numbers to some nth (4,5...) digit reduces the computation greatly without any significant affect on the accuracy. The present implementation compares numbers up to the 4th digit. This can be changed by changing the value of the global variable *precision* to any desired accuracy.

The default value of this variable is .0001.

• Calculating new πs at a node involves two inner products, one outer product and one term product for each π for each iteration. Inner products are costly. Instead the product

$$\left(\mathbf{P} \bullet \mathbf{\Lambda}\right) * \mathbf{\Pi}$$

where
$*$ is the term product
$\bullet$ is the inner product
$\mathbf{P}$ is the joint conditional probability tensor
$\mathbf{\Lambda}$ is the term product of incoming λs
$\mathbf{\Pi}$ is the outer product of incoming πs (these terms are described more fully in Appendix A)

is calculated for each iteration and then πs are calculated by summing over different indices of the above product. This saves a great deal of execution time by reducing multiplications and divisions to additions.

Appendix A

Tensor Product Computation

A *tensor* is a mathematical object that is a generalization of a vector to higher orders. The order of a tensor is the number of indices needed to specify an element. A vector is therefore a tensor of order one and a matrix is a tensor of order two. Three standard operations defined on tensors are relevant to this discussion:

Term Product The term product is defined between two tensors **A** and **B** having the same indices. Each element in the resulting tensor **C** is simply the product of the elements with the corresponding indices from **A** and **B**.

$$\mathbf{C} = \mathbf{A} * \mathbf{B} \quad \text{where} \quad c_{i_1 \cdots i_n} = a_{i_1 \cdots i_n} \times b_{i_1 \cdots i_n}$$

Outer Product The outer product of two tensors **A** and **B** having order m and n respectively is a tensor **C** of order $m+n$. Each element of **C** is the product of the elements of **A** and **B** whose aggregate indices correspond to its own indices.

$$\mathbf{C} = \mathbf{A} \circ \mathbf{B} \quad \text{where} \quad c_{i_1 \cdots i_m j_1 \cdots j_n} = a_{i_1 \cdots i_m} \times b_{j_1 \cdots j_n}$$

Inner Product The inner product of two tensors **A** and **B** is a tensor formed by taking the outer product of **A** and **B** and then summing up over common indices that appear both in **A** and **B**. If **A** is of order m, **B** is of order n and they have k common indices then the inner product **C** is a tensor of order $(m-k)+(n-k)$.

$$\mathbf{C} = \mathbf{A} \bullet \mathbf{B} \quad \text{where} \quad c_{i_1 \cdots i_{m-k} j_1 \cdots j_{n-k}} = \sum_{l_1, \cdots, l_k} a_{i_1 \cdots i_{m-k} l_1 \cdots l_k} \times b_{l_1 \cdots l_k j_1 \cdots j_{n-k}}$$

Equations

Let λ_{Y_j}, $\lambda^*_{Y_j}$, π_{U_i}, and $\pi^*_{U_i}$ be vectors (or, equivalently, tensors of order 1) whose elements are the messages a node X receives from its children and its parents respectively:

$$\lambda_{Y_j} = \left(\lambda_{Y_j}(x_1), \cdots, \lambda_{Y_j}(x_r)\right) \quad \text{where } r \text{ is the number of possible values for } X$$

$$\lambda^*_{Y_j} = \left(\lambda^*_{Y_j}(x_1), \cdots, \lambda^*_{Y_j}(x_r)\right) \quad \text{where } r \text{ is the number of possible values for } X$$

$$\pi_{U_i} = \left(\pi_X(u_{i_1}), \cdots, \pi_X(u_{i_{r(i)}})\right) \quad \text{where } r(i) \text{ is the number of possible values for } U_i$$

$$\pi^*_{U_i} = \left(\pi^*_X(u_{i_1}), \cdots, \pi^*_X(u_{i_{r(i)}})\right) \quad \text{where } r(i) \text{ is the number of possible values for } U_i$$

The term product of all λ_{Y_j} vectors is another vector Λ of length r given by

$$\Lambda = \lambda_{Y_1} * \cdots * \lambda_{Y_m} = \left(\prod_{j=1}^{m} \lambda_{Y_j}(x_1), \cdots, \prod_{j=1}^{m} \lambda_{Y_j}(x_r)\right)$$

The term product of all $\lambda^*_{Y_j}$ vectors is another vector Λ^* of length r given by

$$\Lambda^* = \lambda^*_{Y_1} * \cdots * \lambda^*_{Y_m} = \left(\prod_{j=1}^{m} \lambda^*_{Y_j}(x_1), \cdots, \prod_{j=1}^{m} \lambda^*_{Y_j}(x_r)\right)$$

The outer product of all π_{U_i} vectors is a tensor Π of order n given by

$$\Pi = \pi_{U_1} \bigcirc \cdots \bigcirc \pi_{U_n} \text{ where } \pi_{k_1 \cdots k_n} = \prod_{i=1}^{n} \pi_X(u_{i_i})$$

The outer product of all $\pi^*_{U_i}$ vectors is a tensor Π^* of order n given by

$$\Pi^* = \pi^*_{U_1} \bigcirc \cdots \bigcirc \pi^*_{U_n} \text{ where } \pi^*_{k_1 \cdots k_n} = \prod_{i=1}^{n} \pi^*_X(u_{i_i})$$

We can consider the set of fixed probabilities $P(x \mid u_1, \cdots, u_n)$ as elements of a tensor $\mathbf{P}$ of order $n+1$. Now if we compute the inner product of $\mathbf{P}$ with Π we obtain a tensor of order 1 (the indices for the U_i are common to both tensors):

$$\mathbf{P} \bullet \Pi = \left(\sum_{i_1, \cdots, i_n} P(x_1 \mid u_{i_1}, \cdots, u_{i_n}) \prod_{k=1}^{n} \pi_X(u_{i_k}) \, , \cdots , \sum_{i_1, \cdots, i_n} P(x_r \mid u_{i_1}, \cdots, u_{i_n}) \prod_{k=1}^{n} \pi_X(u_{i_k}) \right)$$

If we make the summation operator explicit, we can rewrite the formula as

$$\mathbf{P} \bullet_+ \Pi = \left(\sum_{i_1, \cdots, i_n} P(x_1 \mid u_{i_1}, \cdots, u_{i_n}) \prod_{k=1}^{n} \pi_X(u_{i_k}) \, , \cdots , \sum_{i_1, \cdots, i_n} P(x_r \mid u_{i_1}, \cdots, u_{i_n}) \prod_{k=1}^{n} \pi_X(u_{i_k}) \right)$$

We can now denote the formula for BEL as

$$\mathbf{BEL} = \alpha \, \Lambda \ast (\mathbf{P} \bullet_+ \Pi)$$

If we compute the inner product of $\mathbf{P}$ with Π^* we obtain a tensor of order 1:

$$\mathbf{P} \bullet \Pi^* = \left(\sum_{i_1, \cdots, i_n} P(x_1 \mid u_{i_1}, \cdots, u_{i_n}) \prod_{k=1}^{n} \pi^*_X(u_{i_k}) \, , \cdots , \sum_{i_1, \cdots, i_n} P(x_r \mid u_{i_1}, \cdots, u_{i_n}) \prod_{k=1}^{n} \pi^*_X(u_{i_k}) \right)$$

The *BEL** computation requires us to maximize over all elements u_k rather than taking a sum, so we can redefine the inner product operator as a maximize operator, and we can denote this new inner product with the symbol $\bullet_{\max}$.

$$\mathbf{P} \bullet_{\max} \Pi^* = \left(\max_{i_1, \cdots, i_n} P(x_1 \mid u_{i_1}, \cdots, u_{i_n}) \prod_{k=1}^{n} \pi^*_X(u_{i_k}) \, , \cdots , \max_{i_1, \cdots, i_n} P(x_r \mid u_{i_1}, \cdots, u_{i_n}) \prod_{k=1}^{n} \pi^*_X(u_{i_k}) \right)$$

Now the $BEL^*(x)$ computation can be written in tensor notation as

$$\mathbf{BEL}^* = \alpha \, \Lambda^* \ast (\mathbf{P} \bullet_{\max} \Pi^*)$$

Moreover, it is clear that we can use similar methods to compute the messages that node X will send to its neighbors. The vector π_{Y_j} destined for child Y_j can be computed by term-by-term division of the elements of **BEL** by the elements of λ_{Y_j}, and the vector $\pi^*_{Y_j}$ can be computed by term-by-term division of the elements of **BEL*** by the elements of $\lambda^*_{Y_j}$. The vector λ_X destined for parent U_i can be computed just like **BEL** except that we replace the vector π_{U_i} with a unit vector $(1, \cdots, 1)$ of equal length when computing the outer product Π, and the vector λ^*_X can be computed just like **BEL*** except that we replace the vector $\pi^*_{U_i}$ with a unit vector $(1, \cdots, 1)$ of equal length when computing the outer product Π^*.

The beliefs and belief commitments can be computed in one uniform scheme as shown in Figure 3.

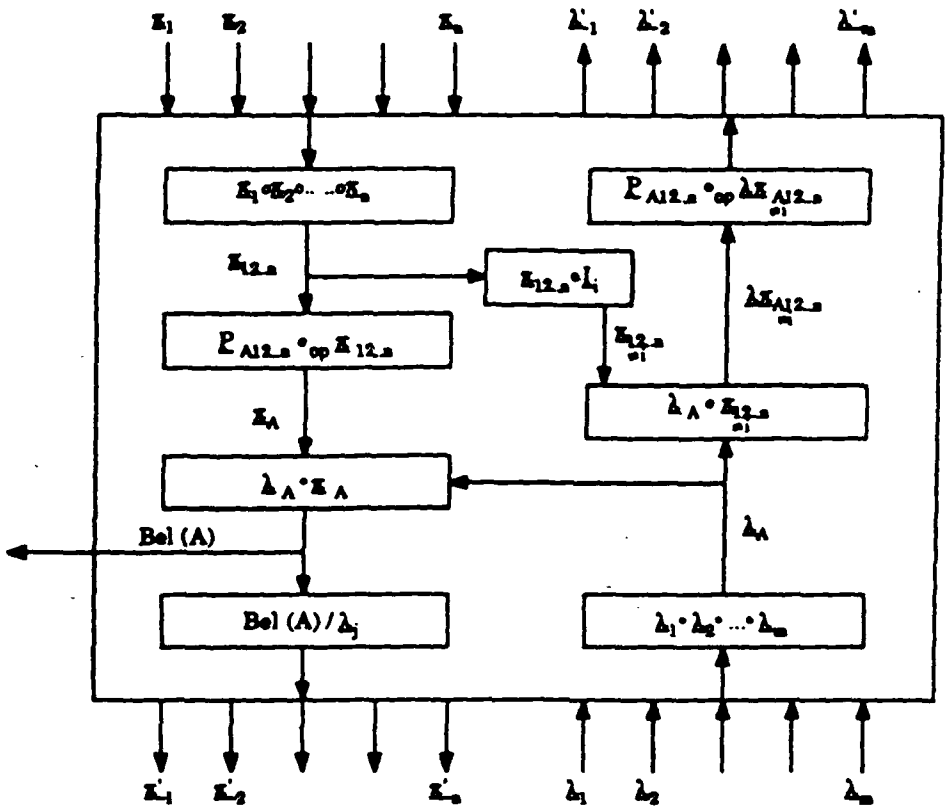


Figure 3. Combined updating of belief and belief commitment (the operator $*_{op}$ represents the standard or modified inner product depending on whether op is + or max). In belief commitment all the π, λs, and Bels should be changed to π*s, λ*s, and Bel*s.

Appendix B - Using the System without the Graphic Interface

BaRT can be run without the graphic interface. The user must get into a Common LISP environment which supports PCL. Load the file *generic.lisp* which is in the *src* subdirectory of the *bart* directory. Then, type the command *(in-package 'bart-frame).*

Now, the following LISP Functions can be invoked:

gen-load :

Loads a data file. This function takes a file name as an optional argument and loads the file. If a file name is not specified, then this function will prompt for one. It performs the necessary internal calculations and then brings the network into equilibrium. An example of the gen-load command on the Symbolics might be *(gen-load "local:>bart>data>ship.lisp")* and an example of this command on the Sun might be *(gen-load "/usr/prj/bart/data/ship.lisp").*

gen-change :

Adds new external evidence to a node. This function takes a node name as an optional argument. If a node name is not specified, this function will prompt for one. If an invalid node name is given, a list of the valid node names will appear, and the function will again prompt for a node name. The function will then prompt for a list of numbers (of length equal to the rank of that node) representing the new evidence; each number in the list gives the impact of the evidence on the belief in the corresponding value (in order) of the proposition. If new evidence for a particular value is unknown, that can be given by including a nil in the list corresponding to that value. The effect of the new evidence can be propagated by calling *gen-propagate.*

gen-propagate :

Updates the network. This function does not take any arguments.

gen-select&display :

Displays the information about a node or a link. This function takes a node or a link name as an optional argument. If a node or a link name is not specified, this function will prompt for one. If an invalid name is given, a list of the valid node and link names will appear, and the function will again prompt for a name.

display-nodes :

Displays the information about all of the nodes. This function does not take any arguments.

display-links :

Displays the information about all of the links. This function does not take any arguments.

display-all :

Displays the information about all of the nodes and links. This function does not take any arguments.

gen-revert-net :

Resets the network back to the initial equilibrium state so the user can try a new run with new observations without loading and reinitializing. This function does not take any arguments.

gen-refresh :

This is only applicable when using the graphic interface, and it is presented here for consistency.

gen-targetnode :

This is only applicable when using the graphic interface, and it is presented here for consistency.

gen-user-modes :

This is only applicable when using the graphic interface, and it is presented here for consistency.

gen-exit :

This is only applicable when using the graphic interface, and it is presented here for consistency.

gen-explain :

Explains the reasoning. Not yet implemented.

gen-snapshot :

Saves the present environment in a file. Not yet implemented.

gen-add :

Adds nodes and links to the network. Not yet implemented.

References

1. Booker, L. B., An Artificial Intelligence (AI) Approach to Ship Classification. In *Intelligent Systems: Their Development and Application.* Proceedings of the 24th Annual Technical Symposium, Washington, D.C. Chapter of the ACM. Gaithersburg, MD., June, 1985, p.29-35.
2. Booker, L. B. and Hota, N., Probabilistic Reasoning About Ship Images. Proceedings of the 2nd AAAI Workshop on Uncertainty in Artificial Intelligence, Philadelphia, PA, August 8-10, 1986, p.29-36; also to appear in *Uncertainty in Artificial Intelligence,* Lemmer and Kanal (Eds.), North Holland, 1987.
3. Booker, L. B., Hota, N. and Hemphill, G., Implementing a Bayesian Scheme for Revising Belief Commitments. Proceedings of the 3rd AAAI Workshop on Uncertainty in Artificial Intelligence, Seattle, WA, July 10-12, 1987, p.348-354.
4. Geffner, H. and Pearl, J., Distributed Diagnosis of Systems with Multiple Faults. Technical Report CSD-860023, Computer Science Department, University of California, Los Angeles, CA, December 1986.
5. Kim, J., CONVINCE: A **CONV**ersational **IN**ference **C**onsolidation **E**ngine. Ph.D. Dissertation, University of California, Los Angeles, 1983.
6. Kim, J. and Pearl, J., A Computation Model for Combined Causal and Diagnostic Reasoning in Inference Systems, *Proceedings of IJCAI-83*, Los Angeles, CA, August 1983, p.190-193.
7. Pearl, J., Fusion, Propagation, and Structuring in Belief Networks, *Artificial Intelligence*, Vol 9, p.241-288, 1986.
8. Pearl, J., Distributed Revision of Belief Commitment in Multi-Hypotheses Interpretation. Proceedings of the 2nd AAAI Workshop on Uncertainty in Artificial Intelligence, Philadelphia, PA, August 8-10, 1986, p.201-209.
9. Shafer, G., Tversky, A., Languages and Designs for Probability Judgement. Cognitive Science, Vol 9, p.309-339, 1985.

```
;;;-*-Mode:LISP; Package:CL-USER; Base:10; Syntax:Common-lisp -*-
;;;
;;; ******************************************************************************
;;; Copyright (c) 1985, 1986, 1987 Xerox Corporation.  All rights reserved.
;;;
;;; Use and copying of this software and preparation of derivative works
;;; based upon this software are permitted.  Any distribution of this
;;; software or derivative works must comply with all applicable United
;;; States export control laws.
;;;
;;; This software is made available AS IS, and Xerox Corporation makes no
;;; warranty about the software, its performance or its conformity to any
;;; specification.
;;;
;;; Any person obtaining a copy of this software is requested to send their
;;; name and post office or electronic mail address to:
;;;   CommonLoops Coordinator
;;;   Xerox Artifical Intelligence Systems
;;;   2400 Hanover St.
;;;   Palo Alto, CA 94303
;;; (or send Arpanet mail to CommonLoops-Coordinator.pa@Xerox.arpa)
;;;
;;; Suggestions, comments and requests for improvements are also welcome.
;;; ******************************************************************************
;;;
;;; This is a hack on the pcl defsys file to use it to generate the bart
;;; system in a portable manner.

;;; Set up the packages we will use.
(or (find-package 'bart-util)
#+Symbolics (in-package 'bart-util :use '(scl lisp))
#-Symbolics (in-package 'bart-util :use (package-use-list (find-package 'user))))

(or (find-package 'bart) (make-package 'bart :use '(pcl lisp)))

(or (find-package 'bart-frame)
#+Symbolics (make-package 'bart-frame :use '(scl lisp))
#-Symbolics (make-package 'bart-frame :use (package-use-list (find-package 'user))))

(defvar *BaRT-system-date* "5/15/87 May 15, 1987")

;;;
;;; Various hacks to get people's *features* into better shape.
;;;
(eval-when (compile load eval)

  #+Symbolics
  (si:inhibit-style-warnings
    (let ((major (si:get-system-version)))
      (cond ((= major 271) (pushnew ':symbolics-release-6 *features*))
            ((= major 349) (pushnew ':symbolics-release-7 *features*))
            (t (error "don't know this system version")))))

  (dolist (feature *features*)
    (when (and (symbolp feature)                ;3600!!
               (equal (symbol-name feature) "CMU"))
      (pushnew :cmu *features*)))

  #+TI
  (if (eq (si:local-binary-file-type) :xld)
      (pushnew 'ti-release-3 *features*)
      (pushnew 'ti-release-2 *features*))

  )

;;;
;;; When installing PCL at your site, edit this defvar to give the directory
;;; in which the PCL files are stored.  The values given below are EXAMPLES
;;; of correct values for *pcl-pathname-defaults*.
;;;
(defvar *pcl-pathname-defaults*
        #+Symbolics                 (pathname "local:>bart>src>")
        #+SUN                       (pathname "/usr/prj/bart/src/")
        #+ExCL                      (pathname "/usr/prj/bart/src/")
        #+KCL                       (pathname "/usr/prj/bart/src/")
        )

;;;
```

```
;;; Note: Something people installing BaRT on a machine running Unix
;;;       might find useful.  If you want to change the extensions
;;;       of the source files from ".lisp" to ".lsp", *all* you have
;;;       to do is the following:
;;;
;;;       % foreach i (*.lisp)
;;;       ? mv $i $i:r.lsp
;;;       ? end
;;;       %
;;;
;;;       I am sure that a lot of people already know that, and some
;;;       Unix hackers may say, "jeez who doesn't know that".  Those
;;;       same Unix hackers are invited to fix mv so that I can type
;;;       "mv *.lisp *.lsp".
;;;

(defvar *pathname-extensions*
        (let ((files-renamed-p t)
              (proper-extensions
                (car '(#+Symbolics                          ("lisp"  . "bin")
                       #+(and dec common vax (not ultrix)) ("LSP"   . "FAS")
                       #+(and dec common vax ultrix)       ("lsp"   . "fas")
                       #+KCL                               ("lsp"   . "o")
                       #+Xerox                             ("lisp"  . "dfasl")
                       #+(and Lucid MC68000)               ("lisp"  . "lbin")
                       #+(and Lucid VAX VMS)               ("lisp"  . "vbin")
                       #+(and Lucid Prime)                 ("lisp"  . "pbin")
                       #+excl                              ("cl"    . "fasl")
;                      #+system::cmu                       ("slisp" . "sfasl")
                       #+HP                                ("l"     . "b")
                       #+TI                                ("lisp"  . "xfasl")
                       #+:gclisp                           ("LSP"   . "FAS")
                       ))))
          (cond ((null proper-extensions) '("l" . "lbin"))
                ((null files-renamed-p) (cons "lisp" (cdr proper-extensions)))
                (t proper-extensions))))

(defun make-source-pathname (name)
  (make-pathname
    :name #-VMS (string-downcase (string name))
          #+VMS (string-downcase (substitute #\_ #\- (string name)))
    :type (car *pathname-extensions*)
    :defaults *pcl-pathname-defaults*))

(defun make-binary-pathname (name)
  (make-pathname
    :name #-VMS (string-downcase (string name))
          #+VMS (string-downcase (substitute #\_ #\- (string name)))
    :type (cdr *pathname-extensions*)
    :defaults *pcl-pathname-defaults*))



;;;
;;; *BART-FILES* is a kind of "defsystem" for BaRT.  A new port of BaRT should
;;; add an entry for that port's xxx-low file.
;;;

(defparameter
  *BaRT-files*

  ;; file         load           compile         files which force
  ;;              environment    environment     recompilations of
  ;;                                             this file
  (let ((x-bart-frame (or #+Symbolics 'bart-frame-3600
                          #+Lucid     'bart-frame-sun
                          'bart-frame-gen)))
    '(
      (bart-util      ()                ()                    ())
      (bart           (bart-util)       (bart-util bart)      (bart-util))
      (,x-bart-frame  (bart-util bart)  (bart-util
                                          bart
                                          ,x-bart-frame)      (bart-util bart))
      (bart-evid      t                 (bart-util
                                          bart
                                          ,x-bart-frame
                                          bart-evid)          ())
      )))
```

```
  ;;
;;;;;; operate-on-system
  ;;
;;; Yet Another Sort Of General System Facility and friends.
;;;

(defstruct (module (:constructor make-module (name))
                   (:print-function
                     (lambda (m s d)
                       (declare (ignore d))
                       (format s "#<Module ~A>" (module-name m)))))
  name
  load-env
  comp-env
  recomp-reasons)

(defun make-modules (system-description)
  (let ((modules ()))
    (labels ((get-module (name)
               (or (find name modules :key #'module-name)
                   (progn (setq modules (cons (make-module name) modules))
                          (car modules))))
             (parse-spec (spec)
               (if (eq spec 't)
                   (reverse (cdr modules))
                   (mapcar #'get-module spec))))
      (dolist (file system-description)
        (let* ((name (car file))
               (module (get-module name)))
          (setf (module-load-env module) (parse-spec (cadr file))
                (module-comp-env module) (parse-spec (caddr file))
                (module-recomp-reasons module) (parse-spec (cadddr file))))))
    (reverse modules)))

(defun make-transformations (modules filter make-transform)
  (let ((transforms (list nil)))
    (dolist (m modules)
      (when (funcall filter m transforms)
        (funcall make-transform m transforms)))
    (reverse (cdr transforms))))

(defun make-compile-transformation (module transforms)
  (unless (dolist (trans transforms)
            (and (eq (car trans) ':compile)
                 (eq (cadr trans) module)
                 (return trans)))
    (dolist (c (module-comp-env module))
      (make-load-transformation c transforms))
    #+symbolics-release-6 (make-load-transformation module transforms)
    (push `(:compile ,module) (cdr transforms))))

(defun make-load-transformation (module transforms)
  (unless (dolist (trans transforms)
            (when (eq (cadr trans) module)
              (cond ((eq (car trans) ':compile) (return nil))
                    ((eq (car trans) ':load)    (return trans)))))
    (dolist (l (module-load-env module))
      (make-load-transformation l transforms))
    (push `(:load ,module) (cdr transforms))))

(defun make-load-without-dependencies-transformation (module transforms)
  (unless (dolist (trans transforms)
            (and (eq (car trans) ':load)
                 (eq (cadr trans) module)
                 (return trans)))
    (push `(:load ,module) (cdr transforms))))

(defun compile-filter (module transforms)
  (or (dolist (r (module-recomp-reasons module))
        (when (dolist (transform transforms)
                (when (and (eq (car transform) ':compile)
                           (eq (cadr transform) r))
                  (return t)))
          (return t)))
      (null (probe-file (make-binary-pathname (module-name module))))
      (> (file-write-date (make-source-pathname (module-name module)))
         (file-write-date (make-binary-pathname (module-name module))))))
```

```
(defun operate-on-system (system mode &optional arg print-only)
  (let ((modules (make-modules system))
        (transformations ()))
    (flet ((load-module (m)
             (let ((name (module-name m))
                   (*load-verbose* nil))
               (if (dolist (trans transformations)
                     (and (eq (car trans) :compile)
                          (eq (cadr trans) m)
                          (return trans)))
                   (progn (format t "~&Loading source of ~A..." name)
                          (or print-only
                              (load (make-source-pathname name))))
                   (progn (format t "~&Loading binary of ~A..." name)
                          (or print-only
                              (load (make-binary-pathname name)))))))
           (compile-module (m)
             (format t "~&Compiling ~A..." (module-name m))
             (or print-only
                 (compile-file (make-source-pathname (module-name m))))))
      (setq transformations
        (ecase mode
          (:compile
            (make-transformations modules
                                  #'compile-filter
                                  #'make-compile-transformation))
          (:recompile
            (make-transformations modules
                                  #'(lambda (&rest ignore) ignore t)
                                  #'make-compile-transformation))
          (:query-compile
            (make-transformations modules
                                  #'(lambda (m transforms)
                                      (or (compile-filter m transforms)
                                          (y-or-n-p "Compile ~A?"
                                                    (module-name m))))
                                  #'make-compile-transformation))
          (:compile-from
            (make-transformations modules
                                  #'(lambda (m transforms)
                                      (or (member (module-name m) arg)
                                          (compile-filter m transforms)))
                                  #'make-compile-transformation))
          (:load
            (make-transformations modules
                                  #'(lambda (&rest ignore) ignore t)
                                  #'make-load-transformation))
          (:query-load
            (make-transformations modules
              #'(lambda (m transforms)
                  (y-or-n-p "Load ~A?" (module-name m)))
              #'make-load-without-dependencies-transformation))))

      (#+Symbolics compiler:compiler-warnings-context-bind
       #-Symbolics progn
       (loop (when (null transformations) (return t))
             (let ((transform (pop transformations)))
               (ecase (car transform)
                 (:compile (compile-module (cadr transform)))
                 (:load (load-module (cadr transform)))))))))))



(defun compile-BaRT (&optional m)
  (cond ((null m)      (operate-on-system *BaRT-files* :compile))
        ((eq m 't)     (operate-on-system *BaRT-files* :recompile))
        ((eq m :print) (operate-on-system *BaRT-files* :compile () t))
        ((eq m :query) (operate-on-system *BaRT-files* :query-compile))
        ((symbolp m)   (operate-on-system *BaRT-files* :compile-from (list m)))
        ((listp m)     (operate-on-system *BaRT-files* :compile-from m))))

(defun load-BaRT (&optional m)
  (cond ((null m)      (operate-on-system *BaRT-files* :load))
        ((eq m :query) (operate-on-system *BaRT-files* :query-load))))

(defun rename-BaRT ()
  (dolist (f *BaRT-files*)
```

```
      (let ((old nil)
            (new nil))
        (let ((*BaRT-pathname-defaults* *default-pathname-defaults*))
          (setq old (make-source-pathname (car f))))
        (setq new  (make-source-pathname (car f)))
        (rename-file old new))))

#+Symbolics
(defun edit-BaRT ()
  (dolist (f *BaRT-files*)
    (zwei:find-file (make-source-pathname (car f)))))

#+Symbolics
(defun hardcopy-BaRT ()
  (dolist (f *BaRT-files*)
    (multiple-value-bind (ignore b)
        (zwei:find-file (make-source-pathname (car f)))
      (zwei:hardcopy-buffer b))))
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: BART-UTIL; Lowercase: Yes; Base: 10; -*-

(provide 'bart-util)                              ;the module name
#+Symbolics (in-package 'bart-util :use '(scl lisp))    ;our package name
#-Symbolics (in-package 'bart-util :use (package-use-list (find-package 'user)))
;(shadow 'whatever)                               ;any things to be shadowed
(export '(arrange-and-find-lambda*s               ;the list of advertized functions
          arrange-and-find-lambdas
          findcp-and
          findcp-or
          lisdiv
          lisdivrec
          maklis
          mydiff
          myequal
          outerpro
          scale-and-fix
          termpro
          transpose
          unscale-and-float
          normalize
          checking-divide ))
;(require 'junk)                                  ;modules to be loaded with this one
;(use-package 'whatever)                          ;we want to use the user package
;(import '(whatever))                             ;any symbols we want to import from random
                                                  ;packages
;;; Create the packages we will use here since this is the first file loaded
;;; this is also done in the file bart-defsys
(or (find-package 'bart) (make-package 'bart :use '(pcl lisp)))
(or (find-package 'bart-frame) (make-package 'bart-frame :use '(scl lisp)))

;;; mat.lisp : general mathematical routines.

;;;(eval-when (load compile eval) (load 'init.bin))
(defvar *precision* 0.0001 "precision")

(defun scale-and-fix (lis)
  "Scale the elements of lis by *precision* and fix them"
  (do ((ans nil (cons (floor (/ (car tmp) *precision*)) ans))
       (tmp lis (cdr tmp)))
      ((null tmp) (nreverse ans))))

(defun unscale-and-float (lis)
  (do ((ans nil (cons (* (car tmp) *precision*) ans))
       (tmp lis (cdr tmp)))
      ((null tmp) (nreverse ans))))

;;; normalising if wrt-length-p do it with respect to length of list
(defun normalize(lst &optional (wrt-length-p nil))
  " normalise elements of a list with respect to 1
    (normalize '(.2 .3)) => (.4 .6)"
  (let ((sum (apply #'+ lst)))
    (declare (float sum))
    (cond ((zerop sum)
           ;;(listsomany (length lst) (/ 1.0 (length lst)))
           (make-sequence 'list (length lst)
                          :initial-element (/ 1.0 (length lst))))
          (t (if wrt-length-p (setf sum (/ sum (length lst))))
             (do ((ans1 nil
                        (cons (/ (car lst1) sum) ans1))
                  (lst1 lst (cdr lst1)))
                 ((null lst1) (nreverse ans1)))))))

(defun checking-divide (x y)
  (if (zerop x) 0 (/ x y)))

(defun outerpro2(l1 l2)                           ;outerproduct of two lists
  "returns a vector with the elements of the outerproduct of two vectors.
   (outerpro2 '(1 2) '(.2 .3)) => (.2 .3 .4 .6)"
  (do ((ans nil
            (append
              (do ((xelt tmp2 (cdr xelt))
                   (tmplcar (car tmp1))
                   (ans1 nil (cons (* tmplcar (car xelt)) ans1)))
                  ((null xelt) ans1))
              ans))
       (tmp1 (reverse l1) (cdr tmp1))
       (tmp2 (reverse l2))))
```

```
      ((null tmp1) ans)))

(defun outerpro(l)
  ;; outerproduct of any number of lists of numbers
  "returns a vector representing the outerproduct of each of the elements of l.
   (outerpro '((1 2) (.2 .3))) => (.2 .3 .4 .6)"
  (do ((ans (car (last l))
            (do ((xelt (car nav) (cdr xelt))
                 (ans1 nil))
                ((null xelt) (nreverse ans1))
              (do ((yelt ans (cdr yelt))
                   (xeltcar (car xelt)))
                  ((null yelt))
                (setf ans1 (cons (* xeltcar (car yelt)) ans1)))))
       (nav (cdr (reverse l)) (cdr nav)))
      ((null nav) ans)))

;;; use mapcar outerpro of lis instead of outerpromany
(defun outerpromany (lis)
  "returns the outerproduct of each pair of vectors in the list.
   (outerpromany '(((.2 .3) (.4 .5)) ((.1 .2) (.3 .4))))
        => ((.08 .1 .12 .15) (.03 .04 .06 .08))"
  (do ((ans nil (cons (outerpro (car tmp)) ans))
       (tmp lis (cdr tmp)))
      ((null tmp) (nreverse ans))))

(defvar *maklis-tmp* (make-array 10))
(setf (aref *maklis-tmp* 0) nil)
(dolist (i '(1 2 3 4 5 6 7 8 9))
  (setf (aref *maklis-tmp* i)
        (append (aref *maklis-tmp* (1- i))
                (list i))))

(defun maklis(n)
  "returns a list from 1 to n:  (maklis 3) => (1 2 3)"
  (declare (fixnum n))
  (cond ((< n 10) (aref *maklis-tmp* n))
        (t (append (maklis (1- n)) (list n)))))

;;; use (make-sequence 'list l :initial-element e)
;(defun listsomany (l e)
;  "retuns a list of length l whose elements are e
;   (listsomany 4 A) => (A A A A)"
;  (declare (fixnum l))
;  (cond ((< l 1) nil)
;        ((= l 1) (list e))
;        (t (cons e (listsomany (- l 1) e)))))

(defun lisdiv (lis n1)
  "retuns a list of two sublists having the first n1 elements of lis
   as the first sublist and the rest as the second sublist
  (lisdiv '(1 2 3 4 5) 3) => ((1 2 3) (4 5))"
  (declare (fixnum n1))
  (do ((ans nil (cons (car tmp) ans))
       (x1 0 (1+ x1))
       (tmp lis (cdr tmp)))
      ((= x1 n1) (list (nreverse ans) tmp))))

(defun lisdivrec (lis n1)
  "retuns a list of sublists each having n1 elements from lis
   (lisdivrec '(1 2 3 4 5 6 7 8) '2) => ((1 2) (3 4) (5 6) (7 8))"
  (do ((ans nil
            (cons (do ((ans1 nil (cons (car tmp) ans1))
                       (junk nil (setf tmp (cdr tmp)))
                       (n2 n1 (1- n2)))
                      ((zerop n2) (nreverse ans1)))
                  ans))
       (tmp lis))
      ((null tmp) (nreverse ans))))

;;; use  (mapcar #'* l1 l2) instead of termpro2
;(defun termpro2(l1 l2)
;  ;; term product of two lists of equal length
;  "retuns the term product of two list of numbers of equal length
;   (termpro2 '(1 2) '(3 4)) => (3 8)"
;
;  (do ((ans nil (cons (* (car tmp1) (car tmp2)) ans))
;       (tmp1 l1 (cdr tmp1))
```

```
;       (tmp2 l2 (cdr tmp2)))
;      ((null tmp1) (nreverse ans))))

(defun termpro(l)
  ;; term product of a list of several sublists of equal length
  "retuns the term product of a list of several sublist of numbers
   of equal length
   (termpro '((1 2) (3 4) (5 6)) => (15 48)"
  (do ((ans (car l)
            (do ((ans1 nil (cons (* (car tmp1) (car tmp2)) ans1))
                 (tmp1 ans (cdr tmp1))
                 (tmp2 (car nav) (cdr tmp2)))
                ((null tmp1) (nreverse ans1))))
       (nav (cdr l) (cdr nav)))
      ((null nav) ans)))

(defun transpose(ppro)                          ; matrix transposeion
  "transposes elements of a list of sublists of equal length.
   like the rows and columns exchange in a matrix
   (transpose '((1 2 3) (4 5 6))) => ((1 4) (2 5) (3 6))"
  (let ((rppro (reverse ppro)))
    (do ((ans (mapcar 'list (car rppro))
              (do ((ans1 nil
                         (cons (cons (car tmp1) (car tmp2))
                               ans1))
                   (tmp1 (car nav) (cdr tmp1))
                   (tmp2 ans (cdr tmp2)))
                  ((null tmp1) (nreverse ans1))))
         (nav (cdr rppro) (cdr nav)))
        ((null nav) ans))))

;;; find the outgoing Lambdas by deducting the incomming contribution of the link "which"
;;; from the total.
(defun arrange-and-find-lambdas (ls ord which thispi)
  "arranges LS of order ORD over subscript WHICH into sublists and
   then transpose the resultant and add each sublist. This would be of
   of the same order as THISPI. Then it divides each element of the
   resultant list by THISPI.
   (arrange-and-find-lambdas '(1 2 3 4 5 6) '(3 2) 2 '(7 8)
     => ((/ (+ 1 3 5) 7) (/ (+ 2 4 6) 8))
     => (1.2857143 1.5)"
  (setf ord (lisdiv ord (1- which)))
  (do ((ans nil (cons (apply #'+ (car tmp)) ans))
       (tmp (let ((pr (apply #'*  (car ord)))
                  (tr (caadr ord))
                  (nr (apply #'* (cdadr ord))))
              (cond ((= nr 1) (transpose (lisdivrec ls tr)))
                    ((= pr 1) (lisdivrec ls nr))
                    (t (transpose1 (lisdivrec (lisdivrec ls nr) tr)))))
              (cdr tmp)))
      ((null tmp) (mapcar #'/ (nreverse ans) thispi))))
;;; for star
(defun arrange-and-find-lambda*s (ls ord which) ;
  "arranges LS of order ORD over subscript WHICH into sublists and
   then transpose the resultant and maximizes each sublist. This would be of
   of the same order as of the index which.
   (arrange-and-find-lambda*s '(1 2 3 4 5 6) '(3 2) 2)
     => ( (max 1 3 5) (max 2 4 6))
     => (5 6)"
  (setf ord (lisdiv ord (1- which)))
  (do ((ans nil (cons (apply #'max (car tmp))  ans))
       (tmp (let ((pr (apply #'*  (car ord)))
                  (tr (caadr ord))
                  (nr (apply #'* (cdadr ord))))
              (cond ((= nr 1) (transpose (lisdivrec ls tr)))
                    ((= pr 1) (lisdivrec ls nr))
                    (t (transpose1 (lisdivrec (lisdivrec ls nr) tr)))))
              (cdr tmp)))
      ((null tmp) (nreverse ans))))

(defun transpose1(ppro)
  "transposes elements of a list of sublists of equal length and
   appends each row.
   like the rows and columns exchange in a matrix + append each row
   (transpose  '((1 2 3) (4 5 6))              => ((1 4) (2 5) (3 6))
   (transpose1 '(((1) (2) (3)) ((4) (5) (6)))) => ((1 4) (2 5) (3 6))"
  (let ((rppro (reverse ppro)))
    (do ((ans (car rppro)
```

```
            (do ((ans1 nil
                       (cons (append (car tmp1) (car tmp2))
                             ans1))
                 (tmp1 (car nav) (cdr tmp1))
                 (tmp2 ans (cdr tmp2)))
                ((null tmp1) (nreverse ans1))))
          (nav (cdr rppro) (cdr nav)))
         ((null nav) ans))))

;;; use (mapcar #'/ bel y)
;(defun matdiv (bel y)
;  " this is actually a mapcar over divide. may be more efficient
;    (matdiv '(1 2 3 4) '(.5 1.0 2.0 3.0)) => (2.0 2.0 1.5 1.333)"
;  (declare (float y))
;  (do ((ans nil
;            (cons (cond ((= (car tmp1) 0) 0)
;                        (t (/ (car tmp1) (car tmp2))))
;                  ans))
;       (tmp1 bel (cdr tmp1))
;       (tmp2 y (cdr tmp2)))
;      ((null tmp1) (nreverse ans))))

;;; for AND gate

(defun findcp-and(ppro)
  "retuns the conditional probability matrix normalized with
   respect to  1

            B1  B2       C1  C2  C3            B     C
       A1   .2   .1       .1  .2  .3             \   /
       A2   .8   .9       .9  .8  .7               A

   now conditional probability of A is --
   (findcp '(((.2 .1) (.8 .9)) ((.1 .2 .3) (.9 .8 .7))))
   => ((.0270 .9729) (.0588 .9411) (.0967 .9032) (.0121 .9878) (.0270 .9729) (.0454 .954))."
  (do ((ans nil (cons (normalize (car tmp1)) ans))
       (tmp1 (do ((ans1 (car ppro)
                        (do ((ans2 nil
                                   (cons (outerpro2 (car tmp3)
                                                    (car tmp4)) ans2))
                             (tmp3 ans1 (cdr tmp3))
                             (tmp4 (car tmp2) (cdr tmp4)))
                            ((null tmp3) (nreverse ans2))))
                 (tmp2 (cdr ppro) (cdr tmp2)))
                ((null tmp2) (transpose ans1)))
             (cdr tmp1)))
      ((null tmp1) (nreverse ans))))

;;;    for OR gate
;;; P(Ai | Bk,Cl) = a sumover q>=i [P(Ai | Bk) P(Aq | Cl)]
;;;                 + sumover q>i  [P(Aq | Bk) P(Ai | Cl)]
;;;  (findcp-or '(((.7 .1) (.3 .9)) ((.2 .1) (.8 .9))))
;;;    => ((.76 .24) (.28 .72) (.73 .27) (.19 .81)).
(defun findcp-or(ppro)                          ;(((.7 .1) (.3 .9)) ((.2 .1) (.8 .9)))
  (labels ((findcp-or1(pro1i pro2i)             ;((.7 .3) (.1 .9)) ((.2 .8) (.1 .9))
            (apply 'append
                   (do ((ans nil
                             (cons (do ((ans1 nil
                                              (cons (findcp-or2 (car temp1)
                                                                (car temp2)) ans1))
                                        (temp2 temp3 (cdr temp2)))
                                       ((null temp2) ans1))
                                   ans))
                        (temp1 (reverse pro1i) (cdr temp1))
                        (temp3 (reverse pro2i)))
                       ((null temp1) ans))))
           (findcp-or2(l1 l2)                   ; (.1 .9) (.1 .9)
             (do ((anss nil (cons (+ (* (car tmp1) (apply #'+ tmp2))
                                     (* (car tmp2) (apply #'+ (cdr tmp1))))
                                  anss))
                  (tmp1 l1 (cdr tmp1))
                  (tmp2 l2 (cdr tmp2)))
                 ((null tmp1) (normalize (nreverse anss))))))
    (do ((fans (findcp-or1 (transpose (car ppro)) (transpose (cadr ppro)))
               (findcp-or1 fans (transpose (car ttmp))))
         (ttmp (cddr ppro) (cdr ttmp)))
        ((null ttmp) fans))))
```

```
(defun myequal (ls1 ls2)
  " checks if two lists containing numbers are equal or not by comparing
    corresponding numbers from each list
    (myequal '(1.23459 2.3456) '(1.23451 2.3456)) => T
    two numbers are equal if the absolute difference between those is
    less than the value of the global *PRECISION*"
  (do ((tmp1 ls1 (cdr tmp1))
       (tmp2 ls2 (cdr tmp2)))
      ((or (null tmp1)
           (null tmp2)
           (not (mydiff (car tmp1) (car tmp2))))
       (and (null tmp1) (null tmp2)))))

(defun mydiff(x y)
  " two numbers are equal if the absolute difference between those is
    less than the value of the global *PRECISION*.
    (mydiff 1.23451 1.23459) => t"
  (declare (float x y))
  (< (abs (- y x)) *precision*))

;;; ------------------ end of Util package -------------------------
```

bart.lisp

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: BART; Lowercase: Yes; Base: 10; -*-
(provide 'bart)                                 ;the module name
(in-package 'bart :use '(pcl lisp))             ;our package name
;(shadow 'whatever)                             ;any things to be shadowed
(export '(*targetnode*
          *to-be-updated*
          *to-be-updated**
          *all-nodes*
          *all-nodesh*
          *all-links*
          *all-linksh*
          *selected-node-or-link*
          *condpro-changed-p*
          *step-p*
          *first-pass-p*
          *debug-mode*
          *equilibrium-p*
          *clear-each-time-p*
          do-reset
          init-net
          re-init-net
          updateall-b
          copy-network
          find-importance
          find-entropy
          find-benefit-factors
          find-xys
          revert-net))
; nodes w.r.t the *targetnode*

(require 'bart-util)                            ;modules to be loaded with this one
;(use-package 'pcl)                             ;we want to use the user package
;(import 'whatever)                             ;any symbols we want to import from random
;packages


(defvar *targetnode*)
(setf *targetnode* nil)
(defvar *to-be-updated*)
(setf *to-be-updated* nil)
(defvar *to-be-updated**)
(setf *to-be-updated** nil)
(defvar *selected-node-or-link*)
(setf *selected-node-or-link* nil)
(defvar *condpro-changed-p*)
(setf *condpro-changed-p* nil)
(defvar *step-p* nil)                           ;temporary
(setf *step-p* nil)                             ;temporary
(defvar *first-pass-p*)
(setf *first-pass-p* nil)
(defvar *debug-mode*)
(setf *debug-mode* nil)
(defvar *equilibrium-p*)
(setf *equilibrium-p* nil)
(defvar *clear-each-time-p*)
(setf *clear-each-time-p* t)
(defvar *snapped-input-file-p*)
(setf *snapped-input-file-p* nil)

(defvar *all-nodes* nil "to record instances of nodes")
(defvar *all-links* nil "to record instances of links")
(defvar *all-nodesh* nil "to record the internal names of all instances of nodes")
(defvar *all-linksh* nil "to record the internal names of all instances of links")
(defvar *m-x* nil)
(defvar *m-y* nil)
(defvar *ma-x* nil)
(setf *all-nodes* nil *all-nodesh* nil *all-links* nil *all-linksh* nil
      *m-x* nil *m-y* nil *ma-x* nil)

;;; define objects node and link
(defclass node ()
  ((i-name "noname")
   (doc nil)
   (node-values '(True False))                  ;values node can take, by default binary
   (rank nil)                                   ;number of values for this node
   (gate 'and)                                  ;kind of gate.. or, and , ask, keep
   (relative-x 0)
   (relative-y 0)
```

```
    (ent 0)                                ;entropy of the node
    (imp 0)                                ;importance of the node
    (rel-ben 0)                            ;relative importance of the node.
    (tlinks nil)                           ;top links -- list
    (blinks nil)                           ;bottom links -- list
    (parents nil)
    (children nil)                         ;initial belief -- list
    (unit-vec nil)
    (prior nil)                            ;prior probabilities -- list
    (ext-evid nil)                         ;external evidence
    (init-prior nil)                       ;initial priors -- list
    (parranks nil)                         ;parent's ranks -- list of integers
    (parprobs nil)                         ;parent's probabilities -- list of lists
    (condpro1 nil)                         ;conditional probs -- list of lists
    (condpro2 nil)                         ;transpose of condpro1
    (belief nil)                           ;belief -- list
    (bel* nil)                             ;belief star for explanations
    (init-belief nil)
    (init-bel* nil))
  (:accessor-prefix ||))

(defclass link ()
  ((i-name "noname")
   (tnode nil)                             ; link to node above
   (bnode nil)                             ; link to node below
   (indpro nil)                            ; conditional prob for the link
                                           ;   -- list of list
   (link-lambda nil)                       ; lambda's -- list
   (init-lambda nil)                       ; initial lambda's -- list
   (link-lambda* nil)                      ; lambda star -- list
   (init-lambda* nil)                      ; initial lambda star -- list
   (link-pi nil)                           ; pi's -- list
   (init-pi nil)                           ; initial pi's -- list
   (link-pi* nil)                          ; initial pi star -- list
   (init-pi* nil)                          ; initial pi star -- list
   (mu-info 0))                            ; mutual information
  (:accessor-prefix ||))


(defmacro cnet-while (wh%test &rest wh%body)     ; cnet-while macro
  (let ((%lp (gensym)))
    `(prog nil ,%lp
           (or ,wh%test (return nil))
           ,@wh%body
           (go ,%lp))))

;;; hash table to keep the instance names and their internal names
(defvar *myhash* (make-hash-table :test 'equal))

(defmacro mmake (rin1 rin2 &rest rin)
  "makes an instance and keeps the external and internal instance names
  in a hash table *myhash*."
  `(cond ((equal ,rin2 'node)
          (setf (gethash ,rin1 *myhash*) (make-instance ,rin2 ,@rin)
                *all-nodes* (cons ,rin1 *all-nodes*)
                *all-nodesh* (cons (gethash ,rin1 *myhash*) *all-nodesh*)) t)
         ((equal ,rin2 'link)
          (setf (gethash ,rin1 *myhash*) (make-instance ,rin2 ,@rin)
                *all-links* (cons ,rin1 *all-links*)
                *all-linksh* (cons (gethash ,rin1 *myhash*) *all-linksh*)) t)))


(defmacro msendal(namh slot val)
  " adds VAL to the value of SLOTGET of NAMH at the end"
  `(setf (slot-value ,namh ,slot)
         (nreverse (cons ,val (nreverse (slot-value ,namh ,slot))))))

(defun mspsend (lnkh slot val)
  " checks if VAL is equal to the value of the slot SLOT of LNKH
    and retuns t if equal. Otherwise it returns tnode or bnode
    of LNKH depending if SLOT is :link-lambda or :link-pi respectively."
  (cond ((bart-util:myequal (slot-value lnkh slot) val)
         (setf (slot-value lnkh slot) val) nil)
        (t (setf (slot-value lnkh slot) val)
           (cond ((or (equal slot 'link-pi) (equal slot 'link-pi*))
                  (bnode lnkh))
                 ((or (equal slot 'link-lambda) (equal slot 'link-lambda*))
                  (tnode lnkh))))))
```

```
;; for pi* use fnarg link-pi*
(defmethod getallpis ((slf node) &optional (fnarg #'link-pi))
  "  retuns a list of all incomming PIs or PI*s of a node"
  (cond ((tlinks slf)
         (do ((ans nil
                   (cons (funcall fnarg (gethash (car tmp1) *myhash*))
                         ans))
              (tmp1 (tlinks slf) (cdr tmp1)))
             ((null tmp1) (nreverse ans))))
        (t (list (prior slf)))))                ;top node case.

;; use link-lambda*  as fnarg for getalllambda*s
(defmethod getalllambdas ((slf node) &optional (fnarg #'link-lambda))
  "  retuns a list of all incomming LAMBDAs or LAMBDA*s of a node"
  (cond ((blinks slf)
         (do ((ans nil
                   (cons (funcall fnarg (gethash (car tmp1) *myhash*))
                         ans))
              (tmp1 (blinks slf) (cdr tmp1)))
             ((null tmp1) (nreverse ans))))
        (t (list (unit-vec slf)))))

;;;-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
;;; following are for finding importance and entropy of a node
(defmethod cal-mu-info ((slf link))             ;for a link
  "  finds the mutual-information of a link "
  (setf (mu-info slf)
        (do ((ans 0 (+ ans
                       (do ((ans1 0 (+ ans1
                                       (cond ((= (car tmp4) 0) 0)
                                             ((or (= (car tmp1) 0)
                                                  (= (car tmp2) 0)) 25)
                                             (t
                                              (* (car tmp4)
                                                 (log (/ (car tmp4)
                                                         (car tmp1)
                                                         (car tmp2)))))))))
                            (tmp2 tmp5 (cdr tmp2))
                            (tmp4 (car tmp3) (cdr tmp4)))
                           ((null tmp4) ans1))))
             (tmp1 (belief (gethash (bnode slf) *myhash*))
                   (cdr tmp1))
             (tmp3 (indpro slf)
                   (cdr tmp3))
             (tmp5 (belief (gethash (tnode slf) *myhash*))))
            ((null tmp1) ans))))

(defmethod cal-imp ((slf node))
  (let ((sumtop (apply #'+ (mapcar #'(lambda(x) (mu-info (gethash x *myhash*)))
                                   (tlinks slf))))
        (sumbot (apply #'+ (mapcar #'(lambda(x) (mu-info (gethash x *myhash*)))
                                   (blinks slf))))
        newnodesh tmph)
    (dolist
      (x (mapcar #'(lambda(k) (gethash k *myhash*)) (tlinks slf)))
      (setf tmph (gethash (tnode x) *myhash*))
      (cond ((imp tmph) nil)
            (t (setf (imp tmph)
                     (/ (* (imp slf) (mu-info x)) sumtop))
               (push tmph newnodesh))))
    (dolist
      (x (mapcar #'(lambda(j) (gethash j *myhash*)) (blinks slf)))
      (setf tmph (gethash (bnode x) *myhash*))
      (cond ((imp tmph) nil)
            (t (setf (imp tmph)
                     (/ (* (imp slf) (mu-info x)) sumbot))
               (push tmph newnodesh))))
    (dolist (y newnodesh) (cal-imp y))))

(defun find-importance()
  "  finds the mutual information of each link and then finds the
     importance factors of each node in the network."
  ;; first find mutual information of each link
  (dolist (x *all-linksh*) (cal-mu-info x))
  ;; then find the importance factors of each node
  ;;(or (gethash *targetnode* *myhash*)
  ;;(set-target-node 1))
```

```
  (dolist (x *all-nodesh*) (setf (imp x) nil))
  (setf (imp (gethash *targetnode* *myhash*)) 1)
  (cal-imp (gethash *targetnode* *myhash*)))


(defmethod cal-ent ((slf node))
  "  finds the entropy of a node and saves it in ENT of the node."
  (setf (ent slf)
        (do ((ans 0 (+ ans
                       (cond ((or (= (car tmp1) 0)
                                  (= (car tmp1) 1)) 0)
                             (t (* (car tmp1)
                                   (log (car tmp1)))))))))
            (tmp1  (belief slf) (cdr tmp1)))
           ((null tmp1) ans))))

(defun find-entropy()
  "  finds the entropy of all nodes and saves them in the ENT slot
    of each node."
  (dolist (x *all-nodesh*) (cal-ent x)))

(defun find-rel-benefit-factors()               ; this is w.r.t benefit factor
  "  finds the relative importance of each node in the network with respect to
     the *targetnode* and ranks them . This rank is kept in the slot rel-imp"
  (let* ((avg-imp 0)
         (temp (mapcar #'(lambda(x)
                           (incf avg-imp (abs (* (imp x) (ent x))))
                           (list (abs (* (imp x) (ent x))) x))
                       (set-difference *all-nodesh*
                                       (list (gethash *targetnode* *myhash*))))))
    (setf avg-imp (/ avg-imp (length temp))
          temp (sort temp #'> :key #'car)
          (rel-ben (gethash *targetnode* *myhash*)) 1)  ;targetnode's rel-ben
    (do ((tmp1 temp (cdr tmp1))
         (tmp2 (cdr (bart-util::maklis(1+ (length temp))))
               (cdr tmp2)))
        ((null tmp1) t)
      (setf (rel-ben (cadar tmp1)) (car tmp2)))))

(defun find-benefit-factors()
  "finds the importance, entropy and relative importance of each node in the net"
  (find-importance)
  (find-entropy)
  (find-rel-benefit-factors))
;;; ----------------------------------------------------------------------------
;;; ============================================================================
;;; following are for finding the relative position of each node in the network
;;; ============================================================================
(defmethod find-ys ((slf node))
  "  finds the y-co-ordinate (relative), the depth co-ord, of the node in the network"
  (cond ((relative-y slf))
        (t (setf (relative-y slf)
                 (cond ((null (parents slf)) 1)
                       (t (+ 1 (apply 'max
                                      (mapcar #'(lambda (y)
                                                  (find-ys (gethash y *myhash*)))
                                              (parents slf)))))))))))


(defun find-xys()
  "  finds x,y positional co-ordinates (relative) of all nodes
     in the network and save them as relative-x and relative-y in the node. "
  ;;reinitialize everything to nil
  (setf *m-y* 0 *m-x* 0)
  (dolist (j *all-nodesh*)
    (setf (relative-x j) nil) (setf (relative-y j) nil))
  ;;set up the relative y position for each node and find the max depth(y)
  (dolist (j *all-nodesh*)
    (setq *m-y* (max *m-y* (find-ys j))))
  ;;set up a dummy array to store the number of nodes in each row
  (setf *ma-x* (make-array (list *m-y* 2)))
  (let ((pre (do ((ans nil (cond ((= *m-y* (relative-y  (gethash (car tmp) *myhash*)))
                                  (cons (car tmp) ans))
                                 (t ans)))
                  (tmp *all-nodes* (cdr tmp)))
                 ((null tmp) (nreverse ans))))
        (count 0))
    (dotimes (jk *m-y*)
```

```
      (setf *m-x* (max *m-x* (setf (aref *ma-x* (- *m-y* 1 jk) 0) (length pre))))
      (setf count -1)
      (dolist
        (node-name pre)
        (incf count)
        (setf (relative-x (gethash node-name *myhash*)) count)
        (setf (relative-y (gethash node-name *myhash*)) (- *m-y* 1 jk)))
      (setf pre (remove-duplicates
                 (apply 'append
                        (mapcar #'(lambda(node-name)
                                    (parents (gethash node-name *myhash*)))
                                pre))
                 :from-end t)))))
;;; ---------------------------------------------------------------------------

(defmethod get-lkr((slf link))                  ; don't need this. used in
                                                ; explanations.
  "  returns the contribution of the top part of a link towards the bottom node as
     a likelyhood ratio"
  (do ((ans nil
            (cons (apply #'+ (mapcar #'* (car tmp) tmppi)) ans))
       (tmp (indpro slf) (cdr tmp))
       (tmppi (link-pi slf)))
      ((null tmp) (bart-util:normalize (nreverse ans)))))

(defmethod give-exp((slf node) whichval)        ;should be rewritten
  " gives explanations (very primitive) "
  (let ((th-nd-name (i-name slf))
        (th-value (nth whichval (node-values slf)))
        (ttlks (tlinks slf))
        (bblks (blinks slf)))
    (cond ((null ttlks)                         ; top node
           (format t " ~% ~a is a top node (cause) " th-nd-name)
           (cond ((equal (nth whichval (prior slf)) 1.0)
                  (format t " with no prior knowledge for the value ~a." th-value ))
                 (t (format t " with a prior likelyhood ratio of ~5,3f."
                            (nth whichval (prior slf))))))
          ((null bblks)                         ; botoom node
           (format t " ~% ~a is a leaf node (manifestation) " th-nd-name)
           (cond ((equal (nth whichval (prior slf)) 1.0)
                  (format t "with no prior knowledge for the value ~a." th-value))
                 (t (format t "with an observed likelyhood ratio of ~5,3f."
                            (nth whichval (prior slf))))))
          (t (format t "~% ~a is an intermediate node." th-nd-name)))   ; intermediate node
    (format t "~% Present belief of the value ~a is ~5,3f" th-value
            (nth whichval (belief slf)))

    (cond (ttlks                                ; support from top links
           (format
             t "~%  the following nodes are contributing causal support for the value ~a"
             th-value)
           (mapcar #'(lambda(x)
                       (format t "~%    ~a ==> ~4,0,2f" (tnode (gethash x *myhash*))
                               (nth whichval (get-lkr (gethash x *myhash*)))))
                   ttlks)))
    (cond (bblks
           (format
             t "~% the following nodes are contributing evidential support for the value ~a"
             th-value)
           (mapcar #'(lambda(x)
                       (format t "~%    ~a ==> ~4,0,2f" (bnode (gethash x *myhash*))
                               (nth whichval (link-lambda (gethash x *myhash*)))))
                   bblks)))))

(defun init-net()
  "  Initializes the network"
  ;; sets prior, ext-evid, belief, top links, bottom links and rank of each node
  (dolist
    (xh *all-nodesh*)
    (setf (blinks xh) nil                       ;remove previous vals if any
          (tlinks xh) nil                       ;remove previous vals if any
          (rank xh) (length (node-values xh))
          (unit-vec xh) (make-sequence 'list (rank xh) :initial-element 1.0)
          (belief xh) (bart-util::normalize (unit-vec xh))))

  ;; sets the top node and bottom node of each link and parents and children for each node
  (dolist
    (x *all-links*)
```

```
      (let ((xh (gethash x *myhash*)))
        (msendal (gethash (tnode xh) *myhash*)
                 'blinks x)
        (msendal (gethash (bnode xh) *myhash*)
                 'tlinks x)
        (msendal (gethash (tnode xh) *myhash*)
                 'children (bnode xh))
        (msendal (gethash (bnode xh) *myhash*)
                 'parents (tnode xh))))

    ;;if prior is not given for top node
    (dolist (xh *all-nodesh*)
        (if (and (not (tlinks xh)) (not (prior xh)))
            (setf (prior xh) (unit-vec xh))))

    ;; sets parent probabilities, parent ranks, condpro1, condpro2 for each
    ;; link. condpro2 is (bart-util:transpose condpro1).
    (dolist
      (xh *all-nodesh*)
      (let ((tlk (tlinks xh)))
        (cond (tlk (do ((parpro nil
                                (cons (indpro (gethash (car tlks) *myhash*))
                                      parpro))
                        (parrnk nil
                                (cons (rank (gethash (tnode (gethash (car tlks) *myhash*))
                                                     *myhash*))
                                      parrnk))
                        (tlks (reverse tlk) (cdr tlks)))
                       ((null tlks)
                        (setf (parprobs xh) parpro)
                        (setf (parranks xh) parrnk)
                        (let ((tgate (gate xh)))
                          (cond ((equal tgate 'or)
                                 (setf (condpro1 xh) (bart-util:findcp-or parpro)))
                                ;;((equal tgate 'ask)
                                ;;(setf (condpro1 xh) (ask-condpro xh)))
                                ((and (equal tgate 'keep) (condpro1 xh)))
                                (t (setf (condpro1 xh) (bart-util:findcp-and parpro)))))
                        (setf (condpro2 xh) (bart-util:transpose (condpro1 xh)))))))))
    ;; sets initial values for PIs and LAMBDAs for links.
    (dolist
      (xh *all-linksh*)
      (setf (link-pi xh) (bart-util:normalize (unit-vec (gethash (tnode xh) *myhash*)))
            (link-pi* xh) (link-pi xh))
      (setf (link-lambda xh) (unit-vec (gethash (tnode xh) *myhash*))
            (link-lambda* xh) (link-lambda xh)))

    ;; find positional co-ordinates of nodes
    ;;(find-xys);; should be in main before before calling find-pos
    ;; find absolute co-ordinates of nodes
    ;;(find-pos);??????
    t)

  ;;; *******
  (defun re-init-net(changed-links)
    " what the heck- set these for all the nodes.."
    ;; sets parent probabilities, parent ranks, condpro1, condpro2 for each
    ;; link. condpro2 is (bart-util:transpose condpro1).
    (dolist
      (xh *all-nodesh*)
      (let ((tlk (tlinks xh)))
        (cond (tlk (do ((parpro nil
                                (cons (indpro (gethash (car tlks) *myhash*))
                                      parpro))
                        (tlks (reverse tlk) (cdr tlks)))
                       ((null tlks)
                        (setf (parprobs xh) parpro)
                        (let ((tgate (gate xh)))
                          (cond ((equal tgate 'or)
                                 (setf (condpro1 xh) (bart-util:findcp-or parpro)))
                                ;;((equal tgate 'ask)
                                ;;(setf (condpro1 xh) (ask-condpro xh)))
                                ((and (equal tgate 'keep) (condpro1 xh)))
                                (t (setf (condpro1 xh) (bart-util:findcp-and parpro)))))
                        (setf (condpro2 xh) (bart-util:transpose (condpro1 xh)))))))))
    t)

  (defmethod update ((slf node))
```

```
"  updates the beliefs, finds new PIs or LAMBDAS for the out
   going links and pushes the names of the neighbours whose new updateing
   factors (PIs & LAMBDAs) are different from the old updating factors
   on to a global list (*TO-BE-UPDATED*) so messages would be sent to them
   later. It can be done more elegently by recursion, but doing it this
   way is more efficient."
;; change back ground color for present node
(let* ((ptlinksln (length (tlinks slf)))
       (allpis (getallpis slf))
       (alllambdas (getalllambdas slf))
       (piout (bart-util:outerpro allpis))
       (prelambda (if (ext-evid slf)
                      (mapcar #'*
                              (bart-util::termpro (ext-evid slf))
                              (bart-util:termpro alllambdas))
                      (bart-util:termpro alllambdas)))
      bel contlam prebel)

  (cond ((= ptlinksln 0)                          ; no top links
         (setf bel (mapcar #'* prelambda piout)))
        (t
         (setf bel                                ; new belief
               (mapcar #'*
                       (do ((ans nil
                                 (cons (do ((ans1 0
                                                  (+ ans1
                                                     (* (car mt1)
                                                        (car mt2))))
                                            (mt1 (car temp1)
                                                 (cdr mt1))
                                            (mt2 temp2 (cdr mt2)))
                                           ((null mt1) ans1))
                                       ans))
                            (temp1 (condpro2 slf) (cdr temp1))
                            (temp2 piout))
                           ((null temp1) (nreverse ans)))
                       prelambda)

               ;; matrix mult of condpro * (bart-util:termpro of incomming LAMBDAS)
               contlam
               (do ((ans nil
                         (cons (do ((ans1 0 (+ ans1
                                               (* (car mt1)
                                                  (car mt2))))
                                    (mt1 (car temp1) (cdr mt1))
                                    (mt2 temp2 (cdr mt2)))
                                   ((null mt1) ans1))
                               ans))
                    (temp1 (condpro1 slf) (cdr temp1))
                    (temp2 prelambda))
                   ((null temp1) (nreverse ans))))))

  (setf prebel (belief slf))                      ; note previous belief
  (setf (belief slf) (bart-util:normalize bel))       ; update belief


  (cond ((= ptlinksln 1)                          ; update LAMBDAs
         (let ((temp (mspsend (gethash (car (tlinks slf)) *myhash*)
                              'link-lambda (bart-util:normalize contlam))))
           (cond (temp (push temp *to-be-updated*)))))
        ((> ptlinksln 1)
         (do ((temp1 (tlinks slf) (cdr temp1))
              (temp2 (bart-util:maklis (length (parranks slf))) (cdr temp2))
              (temp3 (mapcar #'* contlam piout)))
             ((null temp1))
           (let* ((tc1h (gethash (car temp1) *myhash*))
                  (temp (mspsend
                          tc1h 'link-lambda
                          (bart-util:normalize
                            (bart-util:arrange-and-find-lambdas
                              temp3
                              (parranks slf) (car temp2)
                              (link-pi tc1h))))))
             ;; save affacted neighbouring nodes
             (cond (temp (push temp *to-be-updated*)))))))


  (do ((temp1 (blinks slf) (cdr temp1))       ; update PIs
```

```
          (temp2 alllambdas (cdr temp2)))
         ((null temp1))
       (let ((temp3 (mspsend (gethash (car temp1) *myhash*)
                             'link-pi (bart-util:normalize
                                        (mapcar #'bart-util:checking-divide
                                                bel (car temp2))))))
         ;; save affacted neighbouring nodes
         (cond (temp3 (push temp3 *to-be-updated*)))))))

;;; updating for explanations *** explanations ***
(defmethod update* ((slf node))
  "  updates the belief*s, finds new PI*s or LAMBDA*S for the out
     going links and pushes the names of the neighbours whose new updateing
     factors (PI*s & LAMBDA*s) are different from the old updating factors
     on to a global list (*TO-BE-UPDATED*) so messages would be sent to them
     later. It can be done more elegently by recursion, but doing it this
     way is more efficient."

  (let* ((ptlinksln (length (tlinks slf)))
         (allpis (getallpis slf #'link-pi*))
         (alllambdas (getalllambdas slf #'link-lambda*))
         (piout (bart-util:outerpro allpis))
         (prelambda (if (ext-evid slf)
                        (mapcar #'*
                                (bart-util::termpro (ext-evid slf))
                                (bart-util:termpro alllambdas))
                        (bart-util:termpro alllambdas)))
         bel contlam prebel)

    (cond ((= ptlinksln 0)                          ; no top links
           (setf bel (mapcar #'* prelambda piout)))
          (t
           (setf bel                                ; new belief*
                 (mapcar #'*
                         (do ((ans nil
                                   (cons (do ((ans1 0
                                                    (max ans1
                                                         (* (car mt1)
                                                            (car mt2))))
                                              (mt1 (car temp1)
                                                   (cdr mt1))
                                              (mt2 temp2 (cdr mt2)))
                                             ((null mt1) ans1))
                                         ans))
                              (temp1 (condpro2 slf) (cdr temp1))
                              (temp2 piout))
                             ((null temp1) (nreverse ans)))
                         prelambda)

                 ;; matrix mult of condpro * (bart-util:termpro of incomming LAMBDAS)
                 contlam
                 (do ((ans nil
                           (cons (do ((ans1 0 (max ans1
                                                   (* (car mt1)
                                                      (car mt2))))
                                      (mt1 (car temp1) (cdr mt1))
                                      (mt2 temp2 (cdr mt2)))
                                     ((null mt1) ans1))
                                 ans))
                      (temp1 (condpro1 slf) (cdr temp1))
                      (temp2 prelambda))
                     ((null temp1) (nreverse ans))))))

    (setf prebel (bel* slf))                        ; note previous belief
    (setf (bel* slf) (bart-util:normalize bel)) ; update belief

    (cond ((= ptlinksln 1)                          ; update LAMBDA*s
           (let ((temp (mspsend (gethash (car (tlinks slf)) *myhash*)
                                'link-lambda* (bart-util:normalize contlam))))
             (cond (temp (push temp *to-be-updated**)))))
          ((> ptlinksln 1)
           (do ((temp1 (tlinks slf) (cdr temp1))
                (temp2 (bart-util:maklis (length (parranks slf))) (cdr temp2)))
               ((null temp1))
             (let ((tc1h (gethash (car temp1) *myhash*))
                   (temp-pis (copy-seq allpis))
                   temp)
               (setf (elt temp-pis (1- (car temp2)))
```

```
                    (make-sequence 'list  (length (link-pi* tclh))
                                   :initial-element 1.0)
                    temp (mspsend
                           tclh 'link-lambda*
                           (bart-util:normalize
                             (bart-util:arrange-and-find-lambda*s
                               (mapcar #'* contlam (bart-util:outerpro temp-pis))
                               (parranks slf) (car temp2)))))
                ;; save affacted neighbouring nodes
                (cond (temp (push temp *to-be-updated**)))))))


    (do ((templ (blinks slf) (cdr templ))       ; update PI*s
         (temp2 alllambdas (cdr temp2)))
        ((null templ))
      (let ((temp3 (mspsend (gethash (car templ) *myhash*)
                            'link-pi* (bart-util:normalize
                                        (mapcar #'bart-util:checking-divide
                                                bel (car temp2))))))
        ;; save affacted neighbouring nodes
        (cond (temp3 (push temp3 *to-be-updated**)))))
    ))

;;; -=*-=*-=*
(defun revert-net()
  " resets the network to the initial equilibrium state right after
    loading the data file and updating."
  (dolist (x *all-nodesh*)
    (setf (prior x) (init-prior x)
          ;;(ext-evid x) (make-sequence 'list (rank x) :initial-element 1.0)
          (belief x) (init-belief x)
          (bel* x) (init-bel* x)
          (ext-evid x) nil))
  (dolist (x *all-linksh*)
    (setf (link-lambda x) (init-lambda x)
          (link-lambda* x) (init-lambda* x)
          (link-pi x) (init-pi x)
          (link-pi* x) (init-pi* x)))
  (find-benefit-factors))

(defun copy-network()
  "  saves the equilibrium information."
  (dolist (x *all-nodesh*)
    (setf (init-prior x) (prior x)
          (init-belief x) (belief x)
          (init-bel* x) (bel* x)))
  (dolist (x *all-linksh*)
    (setf (init-lambda x) (link-lambda x)
          (init-lambda* x) (link-lambda* x)
          (init-pi x) (link-pi x)
          (init-pi* x) (link-pi* x))))


(defun updateall-b(&optional (one-node-only nil))
  "  breadth first upating removing duplicate elements from the begining.
     with an argument it sends an update message to that node.
     Otherwise it sends an update message to every node in the network.
     Then it finds the importance and entropy of each node and draws the
     appropriate shading to reflect the importance of each node to a
     given target node in the network"
  (prog ()
     loop1
        (or *to-be-updated* (return nil))
        (setf *to-be-updated* (remove-duplicates *to-be-updated*))
        (update (gethash (car (last *to-be-updated*)) *myhash*))
        (setf *to-be-updated* (reverse (cdr (reverse *to-be-updated*))))
        (if *step-p* (return nil) (go loop1)))
  ;; now update for explanations (a different updating)
  (prog ()
     loop2
        (or *to-be-updated** (return nil))
        (setf *to-be-updated** (remove-duplicates *to-be-updated**))
        (update* (gethash (car (last *to-be-updated**)) *myhash*))
        (setf *to-be-updated** (reverse (cdr (reverse *to-be-updated**))))
        (if *step-p* (return nil) (go loop2)))

  ;; see if it is in equilibrium.
  (if (or *to-be-updated* *to-be-updated**) nil (setf *equilibrium-p* t))
```

```
  ;; find relative importance,entropy and relative benefit factors of all
  ;; the nodes w.r.t the *targetnode*
  (find-benefit-factors)
  (cond ((and (not *to-be-updated*)
              (not *to-be-updated**)
              *first-pass-p*)
         (copy-network)                          ; save the initial net for later use
         (setf *first-pass-p* nil))))

(defun do-reset()
  "  resets the hash table, removes all pointers to instances of
     nodes and links"
  (clrhash *myhash*)                             ; clear the hash table
  (setf *all-nodes* nil
        *all-links* nil
        *all-nodesh* nil
        *all-linksh* nil
        *to-be-updated* nil
        *to-be-updated** nil))

;(defun set-target-node(dummy) (setf *targetnode* dummy))
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: BART-FRAME; Lowercase:Yes;Base: 10; -*-

;;; ********************************************************
;;;                 package definitions
;;; ********************************************************
(provide 'bart-frame)
(in-package 'bart-frame)
;(shadow 'whatever)                              ;any things to be shadowed
;(export '(clear-bart-window))
(require 'bart-util)                            ;modules to be loaded with this one
(require 'bart)                                 ;
;(use-package 'bart)                            ;we want to use the pcl package
;(import '(bart:*all-nodes*)

;;; ********************************************************
;;;                 Global definitions
;;; ********************************************************

                                                ;packages
(defvar *all-nodes-pos* nil)
(defvar *all-links-pos* nil)
(defvar *input-file*)
(defvar *program* nil)
(defvar *node-w*)
(defvar *node-h*)
(defvar *text-height*)
(defvar *text-attributes*)
(defvar *horizontal-spacing*)
(defvar *vertical-spacing*)
(defvar *screen-x-offset*)
(defvar *screen-y-offset*)
(defvar *active-region-locations*)
(defvar *node-extent*)
(defvar *choice-extent*)
(defvar *choice-list*)
(defvar *title-pane*)
(defvar *lisp-pane*)                            ; lisp interaction pane
(defvar *root-pane*)                            ; root pane
(defvar *display-pane*)                         ; graphic display pane
(defvar *choice-pane*)
(defvar *global-parm-pane*)                     ; bart global-parm pane
(defvar *nd-lk-pane*)                           ; node link info. display pane
(defvar *user-modes-pane*)
(defvar *doc-pane*)
(defvar *evidence-pane*)
(defvar *all-panes*)                            ; a list of all panes???
(defvar 75%-gray)                               ; gray scales
(defvar 50%-gray)
(defvar 33%-gray)
(defvar 25%-gray)
(defvar hes-gray)
(defvar 12%-gray)
(defvar 10%-gray)
(defvar 8%-gray)
(defvar 6%-gray)
(defvar black-gray)
(defvar white-gray)


(setf *input-file* nil
      *node-w* 80
      *node-h* 96
      *text-height* 16
      *text-attributes* nil
      *horizontal-spacing* (+ *node-w* 20)
      *vertical-spacing* (+ *node-h* 40)
      *screen-x-offset* 20
      *screen-y-offset* 20
      *active-region-locations* nil
      *node-extent* (make-extent *node-w* *node-h*)
      *choice-extent* (make-extent 123 20)
      *choice-list* '("add" "change"  "explain" "load" "propagate" "refresh"
                      "revert-bet" "select&display" "snapshot" "targetnode"
                      "user-modes" "exit")
      75%-gray (load-bitmap "/usr/prj/bart/src/bitmaps/75%-gray.bitmap")
      50%-gray (load-bitmap "/usr/prj/bart/src/bitmaps/50%-gray.bitmap")
      33%-gray (load-bitmap "/usr/prj/bart/src/bitmaps/33%-gray.bitmap")
      25%-gray (load-bitmap "/usr/prj/bart/src/bitmaps/25%-gray.bitmap")
```

```
      hes-gray (load-bitmap "/usr/prj/bart/src/bitmaps/hes-gray.bitmap")
      12%-gray (load-bitmap "/usr/prj/bart/src/bitmaps/12%-gray.bitmap")
      10%-gray (load-bitmap "/usr/prj/bart/src/bitmaps/10%-gray.bitmap")
      8%-gray  (load-bitmap "/usr/prj/bart/src/bitmaps/8%-gray.bitmap")
      6%-gray  (load-bitmap "/usr/prj/bart/src/bitmaps/6%-gray.bitmap")
      black-gray (load-bitmap "/usr/prj/bart/src/bitmaps/black-gray.bitmap")
      white-gray (load-bitmap "/usr/prj/bart/src/bitmaps/white-gray.bitmap"))

;;; *******************************************************
;;;                 Functions
;;; *******************************************************
(defun draw-rect (vprt x1 y1 wd ht &key (alu boole-xor))
  (let ((x2 (+ x1 wd))
        (y2 (+ y1 ht)))
    (draw-line vprt (make-position x1 y1) (make-position (1- x2) y1)
               :operation alu)
    (draw-line vprt (make-position x2 y1) (make-position x2 (1- y2))
               :operation alu)
    (draw-line vprt (make-position x2 y2) (make-position (1+ x1) y2)
               :operation alu)
    (draw-line vprt (make-position x1 y2) (make-position x1 (1+ y1))
               :operation alu)))

(defun draw-pattern (source-bitmap dest-bitmap dest-x dest-y wd ht
                     &key (alu boole-xor))
  (let ((s-wd (bitmap-width source-bitmap))
        (s-ht (bitmap-height source-bitmap)))
    (dotimes (temp-y ht)
      (dotimes (temp-x wd)
        (bitblt source-bitmap 0 0
                dest-bitmap
                (+ dest-x temp-x)
                (+ dest-y temp-y)
                (min s-wd (- wd temp-x))
                (min s-ht (- ht temp-y))
                alu)
        (incf temp-x (1- s-wd)))
      (incf temp-y (1- s-ht)))))

(defun get-gray (gray)
  (cond ((equal gray 1) 75%-gray)
        ((equal gray 2) 50%-gray)
        ((equal gray 3) 33%-gray)
        ((equal gray 4) 25%-gray)
        ((equal gray 5) hes-gray)
        ((equal gray 6) 12%-gray)
        ((equal gray 7) 8%-gray)
        ((equal gray 8) 6%-gray)))

(defun find-pos()
  "  finds the absolute co-ordinates of each node in the network
     (for the display window) and saves them in *all-nodes-pos* as a
     property list.
     Also finds the absolute co-ordinates of each link in the network
     and saves them in *all-links-pos* as a property list"
  ;;set the x-offset to center the nodes in each row
  (dotimes (jk bart::*m-y*)
    (setf (aref bart::*ma-x* jk 1)
          (floor (* (- bart::*m-x* (aref bart::*ma-x* jk 0))
                    *horizontal-spacing*) 2)))

  (dolist (x bart::*all-nodes*)                    ; node co-ordinates
            (let ((node-internal-name (gethash x bart::*myhash*)))
              (setf (get '*all-nodes-pos* x)
                    (list (+ (* (bart::relative-x node-internal-name) *horizontal-spacing*)
                             (aref bart::*ma-x* (bart::relative-y node-internal-name) 1)
                             *screen-x-offset*)
                          (+ (* (bart::relative-y node-internal-name) *vertical-spacing*)
                             *screen-y-offset*))))))

  (dolist (x bart::*all-links*)                    ; link co-ordinates
            (let* ((node-name-internal (gethash x bart::*myhash*))
                   (y1 (get '*all-nodes-pos* (bart::tnode node-name-internal)))
                   (y2 (get '*all-nodes-pos* (bart::bnode node-name-internal)))
                   (x1 (car y1))
                   (x2 (car y2))
                   (node-h2 (floor *node-h* 2))
                   (node-w2 (floor *node-w* 2))
```

```
                slope a1 b1 a2 b2)

            (setf x1 (+ node-w2 x1) x2 (+ node-w2 x2)
                  y1 (+ node-h2 (cadr y1)) y2 (+ node-h2 (cadr y2)))
            (cond ((= x1 x2)
                   (setf a1 x1 a2 x1 b1 (+ y1 node-h2) b2 (- y2 node-h2)))
                  ((> x1 x2)
                   (setf slope (/ (- y2 y1) (- x2 x1))
                         a1 (floor (max (- x1 node-w2)
                                        (+ x1 (/ node-h2 slope))))
                         b1 (ceiling (min (+ y1 node-h2)
                                          (- y1 (* node-w2 slope))))
                         a2 (ceiling (min (- x2 (/ node-h2 slope))
                                          (+ x2 node-w2)))
                         b2 (floor (max (- y2 node-h2)
                                        (+ y2 (* node-w2 slope))))))
                  (t (setf slope (/ (- y2 y1) (- x2 x1))
                           a1 (ceiling (min (+ x1 (/ node-h2 slope))
                                            (+ x1 node-w2)))
                           b1 (ceiling (min (+ y1 node-h2)
                                            (+ y1 (* node-w2 slope))))
                           a2 (floor (max (- x2 node-w2)
                                          (- x2 (/ node-h2 slope))))
                           b2 (floor (max (- y2 node-h2)
                                          (- y2 (* node-w2 slope)))))))
            (setf (get '*all-links-pos* x)  (list a1 b1 a2 b2)))))

;;; inverse video a region
(defun inverse-region1(viewport active-region mouse-event x y)
  (declare (ignore mouse-event x y))
  (bitblt-region (viewport-bitmap viewport) active-region
                 (viewport-bitmap viewport) active-region
                 boole-c1))

(defun inverse-once1(viewport active-region)
  (bitblt-region (viewport-bitmap viewport) active-region
                 (viewport-bitmap viewport) active-region
                 boole-c1))

(defun inverse-region (viewport active-region mouse-event x y)
  (declare (ignore mouse-event x y))
    (draw-rect viewport (- (region-origin-x active-region) 1)
               (- (region-origin-y active-region) 1)
               (+ (region-width active-region) 2)
               (+ (region-height active-region) 2)))

(defun inverse-once(viewport active-region)
  (declare (ignore mouse-event x y))
    (draw-rect viewport (- (region-origin-x active-region) 1)
               (- (region-origin-y active-region) 1)
               (+ (region-width active-region) 2)
               (+ (region-height active-region) 2)))

(defun make-nodes-sensitive()
  (clear-bart-window *display-pane* t)
  (dolist (node-name bart::*all-nodes*)
    (let* ((view-xys (get '*all-nodes-pos* node-name))
           (x1 (car view-xys))
           (y1 (cadr view-xys)))
      (push (list node-name x1 (+ x1 *node-w*) y1 (+ y1 *node-h*))
            *active-region-locations*)
      (make-active-region
        (make-region :x x1 :y y1 :extent *node-extent*)
        :bitmap (viewport-bitmap *display-pane*)
        :mouse-enter-region
        #'inverse-region
        :mouse-exit-region
        #'inverse-region
        :mouse-left-down
        #'(lambda (viewport active-region mouse-event x y)
            (declare (ignore viewport active-region mouse-event))
            (throw 'top-level
              (list (find-node-name-at-xy x y)
                    'left-click)))
        :mouse-middle-down
        #'(lambda (viewport active-region mouse-event x y)
            (declare (ignore viewport active-region mouse-event))
            (throw 'top-level
```

```
                (list (find-node-name-at-xy x y)
                      'middle-click)))
          :mouse-right-down
          #'(lambda (viewport active-region mouse-event x y)
              (declare (ignore viewport active-region mouse-event))
              (throw 'top-level
                (list (find-node-name-at-xy x y)
                      'right-click)))))))

(defun find-node-name-at-xy (x y)
  (labels ((in-bounds (x1 y1 list1)
             (if (and (>= x1 (second list1))
                      (<= x1 (third list1))
                      (>= y1 (fourth list1))
                      (<= y1 (fifth list1)))
                 (first list1))))
    (do ((all-regions *active-region-locations* (cdr all-regions)))
        ((or (null all-regions)
             (in-bounds x y (car all-regions)))
         (caar all-regions)))))

(defun accept-a-node-or-link(&optional (node-only nil))
  (let ((stream *display-pane*)
        first-click second-click result)
    (setf first-click (catch 'top-level (read-any (mouse-input))))
    (setf result
          (cond ((and (member (car first-click) bart::*all-nodes*)
                      (equal (cadr first-click) 'left-click))
                 (car first-click))
                ((and (member (car first-click) bart::*all-nodes*)
                      (equal (cadr first-click) 'middle-click)
                      (not node-only))
                 (format t "~% choose the second node to specify the link ~%")
                 (setf second-click (catch 'top-level (read-any (mouse-input))))
                 (if (member (car second-click) bart::*all-nodes*)
                     (let ((1st-node (gethash (car first-click) bart::*myhash*))
                           (2nd-node (gethash (car second-click) bart::*myhash*)))
                       (car (intersection
                              (union (bart::tlinks 1st-node)
                                     (bart::blinks 1st-node))
                              (union (bart::tlinks 2nd-node)
                                     (bart::blinks 2nd-node))))))))))
    (cond ((and node-only (member (car first-click) bart::*all-nodes*))
           (car first-click))
          (result result)
          (t (format t "~% The object selected ~a is neither a node nor a link.~@
                      Try again." result)
             (accept-a-node-or-link)))))

(defun draw-pic (node-name &optional (stream *display-pane*)
                 (view-xys (get '*all-nodes-pos* node-name)))
  "  draws the node and its beliefs as a histogram in the display
     at the proper place (ie. at the value of POS of that node."
  (let* ((node-name-internal (gethash node-name bart::*myhash*))
         (node-rank (bart::rank node-name-internal))
         (wh-gray (get-gray (bart::rel-ben node-name-internal)))
         (ng-h (- *node-h* 16))                  ;bar graph height
         (ng-w (floor (/ *node-w* (+ node-rank node-rank 1))))  ;bar graph width
         (view-x (car view-xys))
         (view-y (cadr view-xys)))

    ;; clear the space first and draw a rectangle with no fill
    (clear-bitmap stream
                  (make-region :x view-x :y view-y
                               :width *node-w* :height *node-h*))
    (draw-rect stream view-x view-y *node-w* *node-h*)
    ;; graying the nodes
    (if wh-gray
        (draw-pattern wh-gray stream view-x (+ view-y 16)
                      *node-w* (-  *node-h* 16)))

    ;; draw node name. should be done at view-x and view-y
    (draw-line stream (make-position view-x (+ view-y 16))
               (make-position (+ view-x *node-w*) (+ view-y 16)))
    (stringblt stream (make-position (+ view-x 3) (+ view-y 14))
               (find-font 'bold-roman)
               (subseq (bart::i-name node-name-internal)
                       0 (min 9 (length (bart::i-name node-name-internal)))))
```

```
                :operation boole-xor)

      ;; draw belief histogram
      (do ((tmpl (bart::belief node-name-internal) (cdr tmpl))
           (d-xl (+ view-x
                    (floor (/ (- *node-w* (* ng-w (+ node-rank node-rank -1))) 2)))
                 (+ d-xl ng-w ng-w)))
          ((null tmpl) t)
        (draw-pattern black-gray stream
                      d-xl (+ view-y (- *node-h* (floor (* ng-h (car tmpl)))))
                      ng-w (floor (* ng-h (car tmpl))))))))

(defun display-net ()
  (clear-bart-window *display-pane*)            ;clear network window
  (with-fast-drawing-environment
    (let ((stream *display-pane*))
      (labels ((draw-link-arrow(link-name)
                 (let ((xys (get '*all-links-pos* link-name)))
                   (draw-line stream
                              (make-position (car xys) (cadr xys))
                              (make-position (caddr xys) (cadddr xys))))))

        (mapcar #'draw-pic bart::*all-nodes*)
        (mapcar #'draw-link-arrow bart::*all-links*)))
    (display-a-node-or-link)))

(defun scroll-to-end-of-window(pane)
  (let ((present-scroll-ratio
         (/ (stream-y-position pane) (bitmap-height (viewport-bitmap pane)))))
    (cond ((> present-scroll-ratio .9)
           (clear-bart-window pane)
           (funcall (window-vertical-scroll pane) pane .001)
           (setf (window-vertical-scroll-ratio pane) 0))
          (t ;;scroll the window
             (funcall (window-vertical-scroll pane) pane present-scroll-ratio)
             ;; update the bubble
             (setf (window-vertical-scroll-ratio pane) present-scroll-ratio)))))

(defun display-a-node-or-link
       (&optional (nd-lk-nm bart::*selected-node-or-link*)
        (flag  bart::*clear-each-time-p*))
  (with-fast-drawing-environment
    (let ((stream *nd-lk-pane*)
          (nd-lk-nmh (gethash nd-lk-nm bart::*myhash*)))
      (if flag (clear-bart-window stream))
      (scroll-to-end-of-window stream)
      (format stream "~% NAME : ~a~%" (symbol-name nd-lk-nm))

      (cond ((member nd-lk-nm bart::*all-nodes*)
             (format stream "~%~5tValues~17tBelief~25tBelief*~33tExt-evid~41tAll-evid")
             (do ((t-vals (bart::node-values nd-lk-nmh) (cdr t-vals))
                  (t-bels (bart::belief nd-lk-nmh) (cdr t-bels))
                  (t-bels* (bart::bel* nd-lk-nmh) (cdr t-bels*))
                  (t-evids (bart-util::termpro (bart::ext-evid nd-lk-nmh))
                           (cdr t-evids))
                  (all-evids (if (equal (car (bart::ext-evid nd-lk-nmh))
                                        (bart::unit-vec nd-lk-nmh))
                                 (bart-util::transpose (cdr (bart::ext-evid nd-lk-nmh)))
                                 (bart-util::transpose (bart::ext-evid nd-lk-nmh)))
                             (cdr all-evids)))
                 ((null t-vals))
                 (format stream "~%~a~18t~4,3f~25t~4,3f~34t~4,3f~42t~a"
                         (subseq (symbol-name (car t-vals))
                                 0 (min 15 (length (symbol-name (car t-vals)))))
                         (car t-bels) (car t-bels*) (car t-evids) (car all-evids))))
            ((member nd-lk-nm bart::*all-links*)
             (format stream "~%~3tPis~12tPi*~19tLambda~28tLambda*")
             (do ((t-pis (bart::link-pi nd-lk-nmh) (cdr t-pis))
                  (t-lams (bart::link-lambda nd-lk-nmh) (cdr t-lams))
                  (t-pis* (bart::link-pi* nd-lk-nmh) (cdr t-pis*))
                  (t-lams* (bart::link-lambda* nd-lk-nmh) (cdr t-lams*)))
                 ((null t-pis))
                 (format stream "~%~2t~5,3f~11t~5,3f~20t~5,3f~29t~5,3f"
                         (car t-pis) (car t-lams) (car t-pis*) (car t-lams*)))
             (format stream "~%~% Conditional Probability Matrix:~%")
             (format stream "~% ~{~10,12t~a~}"
                     (mapcar #'(lambda(x)
                                 (setf x (symbol-name x))
```

```
                                (subseq x 0 (min 10 (length x))))
                              (bart::node-values
                                (gethash (bart::tnode nd-lk-nmh) bart::*myhash*))))
                    (do ((child-vals (bart::node-values
                                       (gethash (bart::bnode nd-lk-nmh) bart::*myhash*))
                                     (cdr child-vals))
                         (indpro-matrix (bart::indpro nd-lk-nmh) (cdr indpro-matrix)))
                        ((null child-vals))
                      (format stream "~%~a~{~12,12t~5,3f~}"
                              (subseq (symbol-name (car child-vals))
                                      0 (min 10 (length (symbol-name (car child-vals)))))
                              (car indpro-matrix)))))

    (format stream "~%"))))
;;; ------------------------------

(defun get-new-evidence (prompt-string val-list default-val)
  (let (ans max-limit res-list)
    (reshape-viewport
     *evidence-pane* :x 200 :y 100
     :width (+ 30 (string-width (format nil " New External Evidence: ~a" prompt-string)
                                (find-font 'bold-roman)))
     :height (+ 55 (* 20 (length val-list))))
    ;; clearout all previous active regions in this window
    (clear-bart-window  *evidence-pane* t)
    (activate-viewport *evidence-pane*)
    (expose-viewport *evidence-pane*)
    ;; print the heading
    (stringblt *evidence-pane* (make-position 10 20)
               (find-font 'bold-roman)
               (format nil " New External Evidence: ~a" prompt-string))
    ;; now set up individual regions
    (setup-regions val-list default-val)
    (setf res-list (copy-list default-val))
    (setf max-limit (+ 1 (* 20 (1+ (length val-list)))))
    (loop (setf ans (catch 'evid-tag (sleep 1000000)))
          (cond ((equal 'abort (car ans)) (return nil))
                ((equal 'done (car ans)) (return res-list))
                ((and (numberp (car ans))
                      (> (car ans) 19)
                      (< (car ans) max-limit))
                 (setf (nth (floor (- (car ans) 21) 20) res-list)
                       (cadr ans)))))))

;;; now for each possible value in val-list print the name and a default value.
;;; Then make the value region mouse sensitive so a left click on them
;;; erases the present value and takes a new value followed by a carriage return.
;;; Also change this new value read in to a temporary list containing all the
;;; values. Finally provide 2 more mouse sensitive regions aboot and done.
;;; return the temporary list of values if user clicks on done.
;;; Otherwise return nil.
(defun setup-regions (val-list default-val)
  (let (i temp-list)
    (setf temp-list
          (mapcar #'(lambda(x)
                      (format nil " ~a : " x))
                  val-list))
    (do ((tlist temp-list (cdr tlist))
         (vlist default-val (cdr vlist))
         (i 40 (+ i 20))
         (j 1 (+ j 1)))
        ((null tlist) (create-done-and-abort i))
      ;;print the value name and the default value
      (stringblt *evidence-pane* (make-position 10 i)
                 *default-font*
                 (car tlist))
      (stringblt *evidence-pane*
                 (make-position (+ 15 (string-width (car tlist) *default-font*)) i)
                 *default-font* (format nil " ~a" (car vlist)))
      ;;create active region
      (make-active-region
        (make-region
          :x (+ 15 (string-width (car tlist) *default-font*))  :y (- i 15)
          :extent (make-extent 75 18))
        :bitmap (viewport-bitmap *evidence-pane*)
        :mouse-enter-region
        #'inverse-region
        :mouse-exit-region
```

```
        #'inverse-region
        :mouse-left-down
        ;; rubout the present val and read a new one.
        #'crete-individual-regions))))

(defun crete-individual-regions
       (viewport active-region mouse-event x y)
  (declare (ignore mouse-event x))
  (let (origin-x origin-y region-w region-h ans-read)
    (setf origin-x (region-origin-x active-region)
          origin-y (region-origin-y active-region)
          region-w (region-width active-region)
          region-h (region-height active-region))
    ;; clear what ever is there first.
    (clear-bitmap
      *evidence-pane*
      (make-region :x origin-x :y origin-y
                   :width region-w :height region-h))
    ;; bring the mouse here
    (setf (stream-x-position *evidence-pane*) origin-x
          (stream-y-position *evidence-pane*) (+ origin-y 15))
    ;; read a new value
    (setf ans-read (get-a-number *evidence-pane*))
    ;; print that val here and change the back ground video
    (stringblt *evidence-pane*
               (make-position origin-x (+ origin-y 15))
               *default-font*
               (format nil " ~a" ans-read))
    ;;(inverse-once viewport active-region)
    (throw 'evid-tag (list y ans-read))))

(defun get-a-number (pane)
  (let (ans-read)
    (format t "~% Type a number : ")
    (with-asynchronous-method-invocation-allowed
     (let ((old-stream (mouse-input)))
       (unwind-protect
         (progn (setf (mouse-input)
                      (make-mouse-input-stream
                       :viewport pane))
                (setf ans-read (read)))
         (setf (mouse-input) old-stream))))
  (cond ((numberp ans-read) ans-read)
        (t (format t "~% Illegal input. Try again.")
           (get-a-number pane)))))

(defun create-done-and-abort (i)
  (stringblt *evidence-pane* (make-position 100 i)
             (find-font 'bold-roman)
             "Done")
  (make-active-region
    (make-region
      :x 100  :y (- i 16)
      :extent (make-extent 40 18))
    :bitmap (viewport-bitmap *evidence-pane*)
    :mouse-enter-region
    #'inverse-region
    :mouse-exit-region
    #'inverse-region
    :mouse-left-down
    ;; rubout the present val and read a new one.
    #'(lambda(viewport active-region mouse-event x y)
        (declare (ignore viewport active-region mouse-event x y))
        (format t "~% done~%")
        (deactivate-viewport *evidence-pane*)
        (hide-viewport *evidence-pane*)
        (throw 'evid-tag '(done))
        ))
  (stringblt *evidence-pane* (make-position 200 i)
             (find-font 'bold-roman)
             "Abort")
  (make-active-region
    (make-region
      :x 200  :y (- i 16)
      :extent (make-extent 50 18))
    :bitmap (viewport-bitmap *evidence-pane*)
    :mouse-enter-region
    #'inverse-region
```

```
        :mouse-exit-region
        #'inverse-region
        :mouse-left-down
        ;; rubout the present val and read a new one.
        #'(lambda(viewport active-region mouse-event x y)
            (declare (ignore viewport active-region mouse-event x y))
            (format t "~% abort~%")
            (deactivate-viewport *evidence-pane*)
            (hide-viewport *evidence-pane*)
            (throw 'evid-tag '(abort))
            )))

(defun display-and-change(&optional (nd-lk-nm bart::*selected-node-or-link*))
  (let ((nd-lk-nmh (gethash nd-lk-nm bart::*myhash*))
        new-evid-supplied)
       (cond ((typep nd-lk-nmh 'bart::node)
              (setf new-evid-supplied
                    (get-new-evidence
                      (symbol-name nd-lk-nm)
                      (bart::node-values nd-lk-nmh)
                      (bart::unit-vec nd-lk-nmh)))
              (cond (new-evid-supplied
                     (push nd-lk-nm bart::*to-be-updated*)
                     (push nd-lk-nm bart::*to-be-updated**)
                     (setf bart::*equilibrium-p* nil)
                     (setf (bart::ext-evid nd-lk-nmh)
                           (cons new-evid-supplied
                                 (bart::ext-evid nd-lk-nmh)))))))))


;;; ****************************************************************************
;;;                          user interface
;;; ****************************************************************************

;;; to open the window toosl
(defun start-window()
  (ed "/usr/prj/bart/src/junk.lisp"
      :windows t
      :title "   Bayesian Reasoning Tool (BaRT)"
      :x 0 :y 0 :width 1174 :height 862
      :viewport-x 750 :viewport-y 500
      :viewport-height 324 :viewport-width 400 :scroll t))

;;; create appropriate windows
(defun start-bart()
  (change-memory-management :growth-limit 1200 :expand-p t)
  (setf *root-pane* (root-viewport))
  (setf *lisp-pane* (car (viewport-children *root-pane*)))
  (reshape-viewport *lisp-pane* :x 760 :y 564 :width 396 :height 243)
  ;; reshape lisp pane
  ;;(let ((editor-lisp-pane (editor::window-dpy-window *lisp-pane*)))
    ;;(reshape-viewport editor-lisp-pane :x 760 :y 564 :width 396 :height 243)
    ;;(editor::update-modified-editor-window editor-lisp-pane *lisp-pane*))

  (setf *title-pane*
        (make-window :x 0 :y 0 :width 1150 :height 20))
  (stringblt *title-pane* (make-position 440 16)
             (find-font 'bold-roman)
             "Bayesian Reasoning Tool (BaRT)")
  (setf *display-pane*
        (make-window :x 0 :y 0 :width 1500 :height 2000
                     :viewport-x 0 :viewport-y 42
                     :viewport-width 745 :viewport-height 758
                     :scroll t
                     :title "   Belief Network Display"))
  (setf *global-parm-pane*
        (make-window :x 760 :y 25 :width 390 :height 39
                     :title "   Global System Parameters" ))
  (setf *choice-pane*
        (make-window :x 760 :y 88 :width 390 :height 133
                     :title "   Command Menu"))
  (setf *nd-lk-pane*
        (make-window :width 800 :height 1350
                     :viewport-x 763 :viewport-y 265
                     :viewport-width 382 :viewport-height 285
                     :scroll t
                     :title "   Node/Link information display pane"))
  (setf *user-modes-pane*
```

```
        (make-window :x 300 :y 300 :width 400 :height 200))
  (present-user-modes-pane)
  (hide-viewport *user-modes-pane*)
  (deactivate-viewport *user-modes-pane*)
  (setf *doc-pane*
        (make-window :x 300 :y 300 :width 400 :height 400))
  (hide-viewport *doc-pane*)
  (deactivate-viewport *doc-pane*)
  (setf *evidence-pane*
        (make-window :x 200 :y 100 :width 400 :height 450))
  (hide-viewport *evidence-pane*)
  (deactivate-viewport *evidence-pane*)
  (expose-all)
  (present-global-parm-pane)
  (present-choice-pane)
  (bart-help)
  (bart-command-loop)
  )

(defun bart-command-loop()
    (loop (catch 'top-level (sleep 100000))))


(defun back()
  (throw 'top-level '(back to bart top level)))

(defun bart-help()
  (format t "~%   You are typing a command to the Bayesian")
  (format t "~%   Reasoning Tool.")
  (format t "~%   Click on one of the mouse choices.")
  (format t "~%   Note: You must issue a LOAD command before")
  (format t "~%   you do anything else with the network.~%~%"))

(defun expose-all()
  (expose-viewport *title-pane*)
  (expose-viewport *global-parm-pane*)
  (expose-viewport *choice-pane*)
  (expose-viewport *nd-lk-pane*)
  (expose-viewport *display-pane*)
  (expose-viewport *lisp-pane*))

;;; set the documentatin pane
(defun hide-doc-pane (viewport active-region mouse-event x y)
  (declare (ignore viewport active-region mouse-event x y))
  (clear-bart-window *doc-pane*)                ; clear the window
  (hide-viewport *doc-pane*)
  (deactivate-viewport *doc-pane*))

(defun update-doc-pane (list-of-strings)
    (expose-viewport *doc-pane*)
    (activate-viewport *doc-pane*)
    (format *doc-pane* "~%~{~% ~a ~}" list-of-strings))

;;; all this for a silly multiple choose menu
(defun present-user-modes-pane()
  (stringblt *user-modes-pane*
             (make-position 10 20) (find-font 'bold-roman)
             "Select User Modes")
  (stringblt *user-modes-pane*
             (make-position 10 50) *default-font*
             "Step Mode: ")
  (stringblt *user-modes-pane*
             (make-position 100 50) (find-font 'bold-roman)
             (if bart::*step-p* "yes" "no"))
  (stringblt *user-modes-pane*
             (make-position 10 100) *default-font*
             "Debug Mode: ")
  (stringblt *user-modes-pane* (make-position 108 100)
             (find-font 'bold-roman)
             (if bart::*debug-mode* "yes" "no"))
  (stringblt *user-modes-pane*
             (make-position 10 150) *default-font*
             "Clear node/link window each time: ")
  (stringblt *user-modes-pane* (make-position 324 150)
             (find-font 'bold-roman)
             (if bart::*clear-each-time-p* "yes" "no"))
  (stringblt *user-modes-pane* (make-position 50 190)
             (find-font 'bold-roman) "Done")
```

```
(stringblt *user-modes-pane* (make-position 150 190)
           (find-font 'bold-roman) "Abort")
(make-active-region                                ;step mode
  (make-region :x 99 :y 35 :extent (make-extent 30 20))
  :bitmap (viewport-bitmap *user-modes-pane*)
  :mouse-enter-region
  #'inverse-region
  :mouse-exit-region
  #'inverse-region
  :mouse-left-down
  #'(lambda(viewport active-region mouse-event x y)
      (declare (ignore viewport active-region mouse-event x y))
      (setf bart::*step-p* (not bart::*step-p*))
      (update-global-parm-pane)
      (clear-bitmap *user-modes-pane*
                    (make-region :x 99 :y 35
                                 :extent (make-extent 30 20)))
      (stringblt *user-modes-pane* (make-position 100 50)
                 (find-font 'bold-roman)
                 (if bart::*step-p* "yes" "no"))
      (throw 'user-mode-tag (list 'step bart::*step-p*))))

(make-active-region                                ;debug mode
  (make-region :x 107 :y 85 :extent (make-extent 30 20))
  :bitmap (viewport-bitmap *user-modes-pane*)
  :mouse-enter-region
  #'inverse-region
  :mouse-exit-region
  #'inverse-region
  :mouse-left-down
  #'(lambda(viewport active-region mouse-event x y)
      (declare (ignore viewport active-region mouse-event x y))
      (setf bart::*debug-mode* (not bart::*debug-mode*))
      (update-global-parm-pane)
      (clear-bitmap *user-modes-pane*
                    (make-region :x 107 :y 85
                                 :extent (make-extent 30 20)))
      (stringblt *user-modes-pane* (make-position 108 100)
                 (find-font 'bold-roman)
                 (if bart::*debug-mode* "yes" "no"))
      (throw 'user-mode-tag (list 'debug bart::*debug-mode*))))

(make-active-region                                ;clear-window mode
  (make-region :x 323 :y 135 :extent (make-extent 30 20))
  :bitmap (viewport-bitmap *user-modes-pane*)
  :mouse-enter-region
  #'inverse-region
  :mouse-exit-region
  #'inverse-region
  :mouse-left-down
  #'(lambda(viewport active-region mouse-event x y)
      (declare (ignore viewport active-region mouse-event x y))
      (setf bart::*clear-each-time-p* (not bart::*clear-each-time-p*))
      (clear-bitmap *user-modes-pane*
                    (make-region :x 323 :y 135
                                 :extent (make-extent 30 20)))
      (stringblt *user-modes-pane* (make-position 324 150)
                 (find-font 'bold-roman)
                 (if bart::*clear-each-time-p* "yes" "no"))
      (throw 'user-mode-tag (list 'clear bart::*clear-each-time-p*))))

(make-active-region                                ;done
  (make-region :x 50 :y 173 :extent (make-extent 40 18))
  :bitmap (viewport-bitmap *user-modes-pane*)
  :mouse-enter-region
  #'inverse-region
  :mouse-exit-region
  #'inverse-region
  :mouse-left-down
  #'(lambda(viewport active-region mouse-event x y)
      (declare (ignore viewport active-region mouse-event x y))
      (hide-viewport *user-modes-pane*)
      (deactivate-viewport *user-modes-pane*)
      (throw 'user-mode-tag '(done))))

(make-active-region                                ;abort
  (make-region :x 150 :y 173 :extent (make-extent 50 18))
  :bitmap (viewport-bitmap *user-modes-pane*)
```

```
      :mouse-enter-region
      #'inverse-region
      :mouse-exit-region
      #'inverse-region
      :mouse-left-down
      #'(lambda(viewport active-region mouse-event x y)
          (declare (ignore viewport active-region mouse-event x y))
          (hide-viewport *user-modes-pane*)
          (deactivate-viewport *user-modes-pane*)
          (throw 'user-mode-tag '(abort))))
    )
  ;;''''
  (defun update-user-modes-pane()
    (clear-bitmap *user-modes-pane*
                        (make-region :x 99 :y 34
                                     :extent (make-extent 30 18)))
    (stringblt *user-modes-pane*
               (make-position 100 50) (find-font 'bold-roman)
               (if bart::*step-p* "yes" "no"))
    (clear-bitmap *user-modes-pane*
                        (make-region :x 107 :y 84
                                     :extent (make-extent 30 18)))
    (stringblt *user-modes-pane* (make-position 108 100)
               (find-font 'bold-roman)
               (if bart::*debug-mode* "yes" "no"))
    (clear-bitmap *user-modes-pane*
                        (make-region :x 323 :y 134
                                     :extent (make-extent 30 18)))
    (stringblt *user-modes-pane* (make-position 324 150)
               (find-font 'bold-roman)
               (if bart::*clear-each-time-p* "yes" "no"))
    )
  (defun get-and-set-user-modes()
    (let ((a bart::*step-p*)
          (b bart::*debug-mode*)
          (c bart::*clear-each-time-p*) ans)
    (with-asynchronous-method-invocation-allowed
      (activate-viewport *user-modes-pane*)
      (expose-viewport *user-modes-pane*)
      (update-user-modes-pane)
      (loop (setf ans (catch 'user-mode-tag (sleep 1000000)))
            (cond ((equal 'abort (car ans))
                   (setf bart::*step-p* a
                         bart::*debug-mode* b
                         bart::*clear-each-time-p* c)
                   (update-global-parm-pane)
                   (return nil))
                  ((equal 'done (car ans)) (return nil)))))))


  (defun present-global-parm-pane()
    (stringblt *global-parm-pane*
               (make-position 10 17) *default-font*
               " Network Name: ")
    (stringblt *global-parm-pane* (make-position 125 17)
               (find-font 'italic)
               (pathname-name *input-file*))
    (stringblt *global-parm-pane* (make-position 10 35)
               *default-font* " User Modes: ")
    (stringblt *global-parm-pane* (make-position 145 35)
               (find-font 'italic) "*Step-p*")
    (stringblt *global-parm-pane* (make-position 235 35)
               (find-font 'italic) "*Debug-mode*")
    (make-active-region
      (make-region :x 140 :y 20 :extent (make-extent 90 18))
      :bitmap (viewport-bitmap *global-parm-pane*)
      :mouse-enter-region
      #'inverse-region
      :mouse-exit-region
      #'inverse-region
      :mouse-left-down
      #'(lambda(viewport active-region mouse-event x y)
          (declare (ignore mouse-event x y))
          (setf bart::*step-p* (not bart::*step-p*))
          (update-user-modes-pane)
          (clear-bitmap *global-parm-pane*
                        (make-region :x 140 :y 20
                                     :extent (make-extent 90 18)))
```

```
          (stringblt *global-parm-pane* (make-position 145 35)
                     (find-font (if bart::*step-p* 'bold-roman 'italic))
                     "*Step-p*")
          ;;(inverse-once viewport active-region)
          ))
    (make-active-region
      (make-region :x 235 :y 20 :extent (make-extent 120 18))
      :bitmap (viewport-bitmap *global-parm-pane*)
      :mouse-enter-region
      #'inverse-region
      :mouse-exit-region
      #'inverse-region
      :mouse-left-down
      #'(lambda(viewport active-region mouse-event x y)
          (declare (ignore mouse-event x y))
          (setf bart::*debug-mode* (not bart::*debug-mode*))
          (update-user-modes-pane)
          (clear-bitmap *global-parm-pane*
                        (make-region :x 235 :y 20
                                     :extent (make-extent 120 18)))
          (stringblt *global-parm-pane* (make-position 235 35)
                     (find-font (if bart::*debug-mode* 'bold-roman 'italic))
                     "*Debug-mode*")
          ;;(inverse-once viewport active-region)
          )))

;;; method for setting the global-parm
;;; need to fix window size and add the other data file name for the network
(defun update-global-parm-pane()
  (clear-bitmap *global-parm-pane* (make-region :x 120 :y 1 :width 250 :height 17))
  (clear-bitmap *global-parm-pane* (make-region :x 140 :y 17 :width 230 :height 17))
  (stringblt *global-parm-pane* (make-position 125 17)
             (find-font (if bart::*equilibrium-p* 'bold-roman 'italic))
             (pathname-name *input-file*))
  (stringblt *global-parm-pane* (make-position 145 35)
             (find-font (if bart::*step-p* 'bold-roman 'italic))
             "*Step-p*")
  (stringblt *global-parm-pane* (make-position 235 35)
             (find-font (if bart::*debug-mode* 'bold-roman 'italic))
             "*Debug-mode*"))

(defun clear-bart-window(pane &optional (active-regions-too nil))
  "Clear the window in question and its history if the optinal arg is t
   eg. (clear-bart-window *display-pane t) "
  (clear-bitmap pane)
  (if active-regions-too (clear-bitmap-active-regions pane))
  (setf (stream-x-position pane) 0)
  (setf (stream-y-position pane) (font-baseline *default-font*)))

(defun present-choice-pane()
  (let ((x 11)
        (y 8)
        (x1 257)
        (inc 20))
    (make-region-generic
      x y "Add" #'m-l-d-add #'m-r-d-add)
    (make-region-generic
      x (+ y inc) "Change" #'m-l-d-change #'m-r-d-change)
    (make-region-generic
      x (+ y (* 2 inc)) "Explain" #'m-l-d-explain #'m-r-d-explain)
    (make-region-generic
      x (+ y (* 3 inc)) "Load" #'m-l-d-load #'m-r-d-load)
    (make-region-generic
      x (+ y (* 4 inc)) "Propagate" #'m-l-d-propagate #'m-r-d-propagate)
    (make-region-generic
      x (+ y (* 5 inc)) "Refresh" #'m-l-d-refresh #'m-r-d-refresh)
    (make-region-generic
      134 70 "Eval" #'m-l-d-eval #'m-r-d-eval)
    (make-region-generic
      x1 y "Revert-net" #'m-l-d-revert-net #'m-r-d-revert-net)
    (make-region-generic
      x1 (+ y inc) "Select&display" #'m-l-d-select&display #'m-r-d-select&display)
    (make-region-generic
      x1 (+ y (* 2 inc)) "Snapshot" #'m-l-d-snapshot #'m-r-d-snapshot)
    (make-region-generic
      x1 (+ y (* 3 inc)) "Targetnode" #'m-l-d-targetnode #'m-r-d-targetnode)
    (make-region-generic
      x1 (+ y (* 4 inc)) "User-modes" #'m-l-d-user-modes #'m-r-d-user-modes)
```

```
      (make-region-generic
        xl (+ y (* 5 inc)) "Exit" #'m-l-d-exit #'m-r-d-exit)))

(defun get-a-file-name()
  (let (tempfile)
    (format t "~%Enter file name : ")
    (setf tempfile (pathname (string-downcase (read))))
    (or (probe-file tempfile)
        (setf tempfile
              (make-pathname
                :directory
                (cond ((equal :relative (car (pathname-directory tempfile)))
                       (list "usr" "prj" "bart" "data"))
                      (t (pathname-directory tempfile)))
                :name (pathname-name tempfile)
                :type
                (cond ((pathname-type tempfile))
                      (t "lbin")))))
    (or (probe-file tempfile)
        (setf tempfile
              (make-pathname
                :directory
                (cond ((equal :relative (car (pathname-directory tempfile)))
                       (list "usr" "prj" "bart" "data"))
                      (t (pathname-directory tempfile)))
                :name (pathname-name tempfile)
                :type "lisp")))
    tempfile))

;;; ****************************************************************************
;;;                      Command definitions
;;; ****************************************************************************
(defun m-l-d-add (viewport active-region mouse-event x y)
  (declare (ignore viewport active-region mouse-event x y))
  (format t " Add -- not yet implemented~%")
  (throw 'top-level 'Add))

(defun m-r-d-add (viewport active-region mouse-event x y)
  (declare (ignore viewport active-region mouse-event x y))
  (with-asynchronous-method-invocation-allowed
    (update-doc-pane
      '(" -- to add a node or a link to the"
        "    network"
        "   Not yet implemented.")))
  (throw 'top-level " "))

(defun m-l-d-change  (viewport active-region mouse-event x y)
  (declare (ignore viewport active-region mouse-event x y))
  (with-asynchronous-method-invocation-allowed
    (display-and-change (accept-a-node-or-link t)))
  (throw 'top-level 'Change))

(defun m-r-d-change (viewport active-region mouse-event x y)
  (declare (ignore viewport active-region mouse-event x y))
  (with-asynchronous-method-invocation-allowed
    (update-doc-pane
      '(" -- to give new external evidence to a"
        "    node or to change the conditional"
        "    probability matrix of a link."
        "  Choose a node or link to do this"
        "    You can choose a node by clicking left"
        "    on a node. A link can be chosen by clicking"
        "    right on both the top and bottom nodes of a"
        "    link.")))
  (throw 'top-level " "))

(defun m-l-d-eval (viewport active-region mouse-event x y)
  (declare (ignore viewport active-region mouse-event x y))
  (with-asynchronous-method-invocation-allowed
    (format t "~% Eval ->")
    (format t "~% ~{~a ~}~%"
            (multiple-value-list
              (eval (read)))))
  (throw 'top-level 'Change))

(defun m-r-d-eval (viewport active-region mouse-event x y)
  (declare (ignore viewport active-region mouse-event x y))
  (with-asynchronous-method-invocation-allowed
```

```
      (update-doc-pane
        '(" -- reads a lisp expression and evaluates it")))
    (throw 'top-level " "))

(defun m-l-d-explain (viewport active-region mouse-event x y)
  (declare (ignore viewport active-region mouse-event x y))
  (format t " Expalin -- not yet implemented~%")
  (throw 'top-level 'Explain))

(defun m-r-d-explain (viewport active-region mouse-event x y)
  (declare (ignore viewport active-region mouse-event x y))
  (with-asynchronous-method-invocation-allowed
    (update-doc-pane
      '(" -- to explain the reasoning!!"
        "    Not yet implemented")))
  (throw 'top-level " "))

(defun m-l-d-load (viewport active-region mouse-event x y)
  (declare (ignore viewport active-region mouse-event x y))
  (let ((tempfile
         (with-asynchronous-method-invocation-allowed
           (get-a-file-name))))
    (cond ((probe-file tempfile)
           (setf *input-file* tempfile)
           (bart::do-reset)                    ; remove all old information
           (clear-bart-window *display-pane* t) ;clear network window
           (clear-bart-window *nd-lk-pane* t)   ;clear node/link window
           (setf *active-region-locations* nil)
           (load *input-file* :verbose t)
           (cond (bart::*snapped-input-file-p*)
                 (t (bart:init-net)            ;; initialize the network
                    (bart:find-xys)            ; find relative positions of each node
                    (find-pos)                 ; find absolute co-ordinates of each node
                    (setf bart::*selected-node-or-link*
                          (setf bart:*targetnode* (car bart::*all-nodes*))
                          bart::*first-pass-p* t
                          bart::*to-be-updated* bart::*all-nodes*
                          bart::*to-be-updated** bart::*all-nodes*
                          bart::*equilibrium-p* nil)
                    (bart:updateall-b)))       ; to bring it into equilibrium
           (update-global-parm-pane)
           ;; now display the net and shade the nodes depending on the the importance.
           (make-nodes-sensitive)
           (display-net))
          (t (format t "~% can't find file ~s.~%" tempfile)))
    (throw 'top-level 'Load)))

(defun m-r-d-load (viewport active-region mouse-event x y)
  (declare (ignore viewport active-region mouse-event x y))
  (with-asynchronous-method-invocation-allowed
    (update-doc-pane
      '(" -- prompts for and loads a data file."
        "    It then brings the network into "
        "    equilibrium initially.")))
  (throw 'top-level " "))

(defun m-l-d-propagate (viewport active-region mouse-event x y)
  (declare (ignore viewport active-region mouse-event x y))
  ;; see if *condpro-changed-p* is set. if so reset the joint conditional
  ;; probability and set the flag to nil and then update.
  (cond (bart::*condpro-changed-p*
         (let (changed-links-bottom-nodes)
           (bart::re-init-net bart::*condpro-changed-p*)
           (setf changed-links-bottom-nodes
                 (mapcar #'(lambda(x)
                             (bart::bnode (gethash x bart::*myhash*)))
                         bart::*condpro-changed-p*)
                 bart::*to-be-updated*
                 (append changed-links-bottom-nodes bart::*to-be-updated*)
                 bart::*to-be-updated**
                 (append changed-links-bottom-nodes bart::*to-be-updated**)
                 bart::*condpro-changed-p*
                 nil))))
  (bart::updateall-b)
  (update-global-parm-pane)
  (display-net)
  (throw 'top-level 'Propagate))
```

```
(defun m-r-d-propagate (viewport active-region mouse-event x y)
  (declare (ignore viewport active-region mouse-event x y))
  (with-asynchronous-method-invocation-allowed
    (update-doc-pane
      '(" -- to update the network."
        "   Used after giving new evidence or"
        "    changing the conditional probability"
        "    matrix of a link to propagate the"
        "    affect in the network.")))
  (throw 'top-level " "))

;;; Refresh -- refreshes the screen
(defun m-l-d-refresh (viewport active-region mouse-event x y)
  (declare (ignore viewport active-region mouse-event x y))
  (display-net)
  (throw 'top-level 'Refresh))

(defun m-r-d-refresh (viewport active-region mouse-event x y)
  (declare (ignore viewport active-region mouse-event x y))
  (with-asynchronous-method-invocation-allowed
    (update-doc-pane '(" -- to refresh the display pane.")))
  (throw 'top-level " "))

;;; Revert-net -- keeps the network in the initial equilibrium state.
(defun m-l-d-revert-net (viewport active-region mouse-event x y)
  (declare (ignore viewport active-region mouse-event x y))
  (cond (bart::*first-pass-p*
         (format t "~% Initial equilibrium has not been reached yet. "))
        (t (bart::revert-net)
           (update-global-parm-pane)
           (display-net)))
  (throw 'top-level 'Revert-net))

(defun m-r-d-revert-net (viewport active-region mouse-event x y)
  (declare (ignore viewport active-region mouse-event x y))
  (with-asynchronous-method-invocation-allowed
    (update-doc-pane
      '(" -- to bring the network  into the initial"
        "    equilibrium state. ")))
  (throw 'top-level " "))

;;; Select&display -- selects a node or link and displays it in node/link window.
(defun m-l-d-select&display (viewport active-region mouse-event x y)
  (declare (ignore viewport active-region mouse-event x y))
  (with-asynchronous-method-invocation-allowed
    (setf bart::*selected-node-or-link*
          (accept-a-node-or-link)))
  (display-a-node-or-link)
  (throw 'top-level 'Select&display))

(defun m-r-d-select&display (viewport active-region mouse-event x y)
  (declare (ignore viewport active-region mouse-event x y))
  (with-asynchronous-method-invocation-allowed
    (update-doc-pane
      '(" -- to display information about a selected"
        "    node/link in the display pane."
        "  Choose a node or a link."
        "    You can choose a node by clicking left"
        "    on a node. A link can be chosen by clicking"
        "    right on both the top and bottom nodes of a"
        "    link.")))
  (throw 'top-level " "))

;;; Snapshot -- saves the results/network in a file.
(defun m-l-d-snapshot (viewport active-region mouse-event x y)
  (declare (ignore viewport active-region mouse-event x y))
  ;;call the save routine here
  (format t "Snapshot -- not yet implemented. ~%")
  (throw 'top-level 'Snapshot))

(defun m-r-d-snapshot (viewport active-region mouse-event x y)
  (declare (ignore viewport active-region mouse-event x y))
  (with-asynchronous-method-invocation-allowed
    (update-doc-pane
      '(" -- to save the present state of the"
        "    network in a file."
        "   Not yet implemented.")))
  (throw 'top-level " "))
```

```
;;; Targetnode -- sets the targetnode and updates the dependency relations and
;;; display.
(defun m-l-d-targetnode (viewport active-region mouse-event x y)
  (declare (ignore viewport active-region mouse-event x y))
  (let (view-xys)
    (with-asynchronous-method-invocation-allowed
      (setf bart:*targetnode* (accept-a-node-or-link t)))
    (bart:find-benefit-factors)
    (display-net)
    (setf view-xys (get '*all-nodes-pos* bart::*targetnode*))
    (draw-pattern black-gray *display-pane* (car view-xys)
                  (cadr view-xys) *node-w* *node-h*)
    (throw 'top-level 'Targetnode)))

(defun m-r-d-targetnode (viewport active-region mouse-event x y)
  (declare (ignore viewport active-region mouse-event x y))
  (with-asynchronous-method-invocation-allowed
    (update-doc-pane
      '(" -- prompts for a node and sets it to be the"
        "    targetnode. Recomputes the importance and"
        "    entropy of all the nodes in the network"
        "    with respect to this selected targetnode"
        "    and displays the network."
        "   The chosen targetnode is grayed maximally."
        "    Other nodes which can affect this "
        "    targetnode are grayed according to the"
        "    weight of their influence on reducing the"
        "    uncertainity associated with the beliefs"
        "    of the targetnode")))
  (throw 'top-level " "))


;;; User-modes -- to set the user modes
(defun m-l-d-user-modes (viewport active-region mouse-event x y)
  (declare (ignore viewport active-region mouse-event x y))
  (get-and-set-user-modes)
  (throw 'top-level 'User-modes))

(defun m-r-d-user-modes (viewport active-region mouse-event x y)
  (declare (ignore viewport active-region mouse-event x y))
  (with-asynchronous-method-invocation-allowed
    (update-doc-pane
      '(" -- pops up a window to allow the user to"
        "    change the values of some global system"
        "    parameters."
        "    A brief description of these follows:"
        "  step mode: If t then updating is done"
        "    one node at a time. Otherwise updating"
        "    is done until the network reaches "
        "    equilibrium"
        "  debug mode: not implemented presently"
        "  clear node/link window each time: If t "
        "   then it clears the node/link information"
        "   display each time new information about"
        "   a node/link is displayed. Otherwise it"
        "   appends the new information.")))
  (throw 'top-level " "))


;;; Exit --
(defun m-l-d-exit (viewport active-region mouse-event x y)
        (declare (ignore viewport active-region mouse-event x y))
        ;;(hide-viewport *display-pane*)
        ;;(hide-viewport *global-parm-pane*)
        ;;(hide-viewport *choice-pane*)
        ;;(hide-viewport *nd-lk-pane*)
        ;;(hide-viewport *lisp-pane*)
        (abort)
        (throw 'top-level 'Exit))

(defun m-r-d-exit (viewport active-region mouse-event x y)
        (declare (ignore viewport active-region mouse-event x y))
        (with-asynchronous-method-invocation-allowed
          (update-doc-pane
            '(" -- exits BaRT.")))
        (throw 'top-level " "))
```

```
;;;-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
(defun make-region-generic(x y name-string m-l-d m-r-d)
  (stringblt *choice-pane*
             (make-position (+ x 3) (+ y 16))
             *default-font* name-string)
  (make-active-region
    (make-region :x x :y y :extent *choice-extent*)
    :bitmap (viewport-bitmap *choice-pane*)
    :mouse-enter-region
    #'inverse-region
    :mouse-exit-region
    #'inverse-region
    :mouse-left-down
    m-l-d
    :mouse-right-down
    m-r-d
    :mouse-right-up
    #'hide-doc-pane))
```

bart-frame-3600.lisp


```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: BART-FRAME; Lowercase:Yes;Base: 10; -*-

(provide 'bart-frame)
#+Symbolics (in-package 'bart-frame :use '(scl lisp))
#-Symbolics (in-package 'bart-frame :use (package-use-list (find-package 'user)))
;(shadow 'whatever)                              ;any things to be shadowed
;(export '(clear-bart-window))
(require 'bart-util)                            ;modules to be loaded with this one
(require 'bart)                                 ;
;(use-package 'bart)                    ;we want to use the pcl package
;(import '(cnet:*all-nodes*
;          cnet:*all-links*
;          cnet:*all-nodesh*
;          cnet:*all-linksh*)
;;;=========================from main=======================================

                                                ;packages
(defvar *all-nodes-pos* nil)
(defvar *all-links-pos* nil)
;(defvar ttyo nil)                              ; to record

(defvar *program* nil)
(defvar *node-w*)
(defvar *node-h*)
(defvar *text-height*)
(defvar *text-attributes*)
(defvar *horizontal-spacing*)
(defvar *vertical-spacing*)

(setf *node-w* 80
      *node-h* 96
      *text-height* 16
      *text-attributes* nil
      *horizontal-spacing* (+ *node-w* 20)
      *vertical-spacing* (+ *node-h* 40))

(defun find-pos()
  "  finds the absolute co-ordinates of each node in the network
     (for the display window) and saves them in *all-nodes-pos* as a
     property list.
     Also finds the absolute co-ordinates of each link in the network
     and saves them in *all-links-pos* as a property list"
  ;;set the x-offset to center the nodes in each row
  (dotimes (jk bart::*m-y*)
    (setf (aref bart::*ma-x* jk 1)
          (floor (* (- bart::*m-x* (aref bart::*ma-x* jk 0))
                    *horizontal-spacing*) 2)))

  (dolist (x bart::*all-nodes*)                 ; node co-ordinates
            (let ((node-internal-name (gethash x bart::*myhash*)))
              (setf (get '*all-nodes-pos* x)
                    (list (+ (* (bart::relative-x node-internal-name) *horizontal-spacing*)
                             (aref bart::*ma-x* (bart::relative-y node-internal-name) 1))
                          (* (bart::relative-y node-internal-name) *vertical-spacing*))))))

  (dolist (x bart::*all-links*)                 ; link co-ordinates
            (let* ((node-name-internal (gethash x bart::*myhash*))
                   (y1 (get '*all-nodes-pos* (bart::tnode node-name-internal)))
                   (y2 (get '*all-nodes-pos* (bart::bnode node-name-internal)))
                   (x1 (car y1))
                   (x2 (car y2))
                   (node-h2 (floor *node-h* 2))
                   (node-w2 (floor *node-w* 2))
                   slope a1 b1 a2 b2)

              (setf x1 (+ node-w2 x1) x2 (+ node-w2 x2)
                    y1 (+ node-h2 (cadr y1)) y2 (+ node-h2 (cadr y2)))
              (cond ((= x1 x2)
                     (setf a1 x1 a2 x1 b1 (+ y1 node-h2) b2 (- y2 node-h2)))
                    ((> x1 x2)
                     (setf slope (/ (- y2 y1) (- x2 x1))
                           a1 (floor (max (- x1 node-w2)
                                          (+ x1 (/ node-h2 slope))))
                           b1 (ceiling (min (+ y1 node-h2)
                                            (- y1 (* node-w2 slope))))
                           a2 (ceiling (min (- x2 (/ node-h2 slope))
                                            (+ x2 node-w2)))
                           b2 (floor (max (- y2 node-h2)
```

```
                                    (+ y2 (* node-w2 slope))))))
                    (t (setf slope (/ (- y2 y1) (- x2 x1))
                             a1 (ceiling (min (+ x1 (/ node-h2 slope))
                                              (+ x1 node-w2)))
                             b1 (ceiling (min (+ y1 node-h2)
                                              (+ y1 (* node-w2 slope))))
                             a2 (floor (max (- x2 node-w2)
                                            (- x2 (/ node-h2 slope))))
                             b2 (floor (max (- y2 node-h2)
                                            (- y2 (* node-w2 slope)))))))
               (setf (get '*all-links-pos* x)  (list a1 b1 a2 b2)))))

(defun get-gray (gray)
  (nth (1- gray)
       (list tv:75%-gray tv:50%-gray tv:33%-gray
             tv:25%-gray tv:hes-gray tv:12%-gray
             tv:10%-gray tv:8%-gray tv:6%-gray)))

(defun draw-pic (node-name
                 &optional
                 (stream (dw:get-program-pane 'network-display-pane))
                 (view-xys (get '*all-nodes-pos* node-name)))
  "  draws the node and its beliefs as a histogram in the display
     at the proper place (ie. at the value of POS of that node."
  (let* ((node-name-internal (gethash node-name bart::*myhash*))
         (node-rank (bart::rank node-name-internal))
         (wh-gray (get-gray (bart::rel-ben node-name-internal)))
         (ng-h (- *node-h* 16))                    ;bar graph height
         (ng-w (floor (/ *node-w* (+ node-rank node-rank 1))))  ;bar graph width
         (view-x (car view-xys))
         (view-y (cadr view-xys)))
    (dw:with-redisplayable-output(:stream stream)
      (dw:with-output-as-presentation
        (:single-box t :stream stream :type '((bart::NODE)) :object node-name)
        ;; clear the space first and draw a rectangle with no fill
        (graphics:draw-rectangle view-x view-y (+ view-x *node-w*) (+ view-y *node-h*)
                                 :alu :erase :stream stream)
        (graphics:draw-rectangle view-x view-y (+ view-x *node-w*) (+ view-y *node-h*)
                                 :filled nil :stream stream)
        ;; graying the nodes
        (if wh-gray
            (graphics:draw-rectangle
              view-x (+ view-y 16) (+ view-x *node-w*) (+ view-y *node-h*)
              :alu :draw :pattern wh-gray :stream stream)
            (graphics:draw-rectangle
              view-x view-y (+ view-x *node-w*) (+ view-y *node-h*)
              :alu :draw :filled nil :stream stream))
        ;; draw node name
        (graphics:draw-rectangle view-x view-y (+ view-x *node-w*) (+ view-y 16)
                                 :stream stream :filled nil)
        ;; should be done at view-x and view-y ????*****????
        (with-character-style ('(nil :bold nil) stream)
          (send stream :set-cursorpos (+ 2 view-x) (+ 2 view-y))
          (dw::with-output-truncation(stream)
            (dw:redisplayable-format
              stream "~a"
              (subseq (bart::i-name node-name-internal)
                      0 (min 9 (length (bart::i-name node-name-internal))))))))

        ;;(with-character-style ('(nil :bold nil) stream)
          ;;(let ((temp-string
                  ;;(dw:with-output-to-presentation-recording-string (stream)
                    ;;(present (bart::i-name node-name-internal) 'string))))
            ;;(graphics:draw-string temp-string
                                  ;;view-x (+ view-y 14) :stream stream
                                  ;;:toward-x (+ view-x 76) :stretch-p t)))
        ;; draw belief histogram
        (do ((tmp1 (bart::belief node-name-internal) (cdr tmp1))
             (d-x1 (+ view-x
                      (floor (/ (- *node-w* (* ng-w (+ node-rank node-rank -1))) 2)))
                   (+ d-x1 ng-w ng-w)))
            ((null tmp1) t)
          (graphics:draw-rectangle d-x1 (+ view-y (- *node-h* (floor (* ng-h (car tmp1)))))
                                   (+ d-x1 ng-w) (+ view-y *node-h*) :stream stream))))
    ))

(defun display-net ()
  (clear-bart-window *program* 'network-display-pane)   ;clear network window
```

```
    (let ((stream (dw:get-program-pane 'network-display-pane)))
      (labels ((draw-link-arrow(link-name)
                 (let ((xys (get '*all-links-pos* link-name)))
                   (dw:with-output-as-presentation
                     (:stream stream :object link-name :type '((bart::LINK)))
                     (graphics:draw-arrow
                       (car xys) (cadr xys) (caddr xys) (cadddr xys) :stream stream)))))
        (mapcar #'draw-pic bart::*all-nodes*)
        (mapcar #'draw-link-arrow bart::*all-links*)))
      (display-a-node-or-link))

  (defun end-of-scroll-window (stream)
    (multiple-value-bind (nil nil nil a) (send stream :y-scroll-position)
      (send stream :y-scroll-to a :absolute)))

  (defun display-a-node-or-link
         (&optional (nd-lk-nm bart::*selected-node-or-link*)
          (flag  bart::*clear-each-time-p*))
    (let ((stream (dw:get-program-pane 'node-display-pane))
          (nd-lk-nmh (gethash nd-lk-nm bart::*myhash*)))
      (if flag (send stream :clear-history)
          (end-of-scroll-window stream))
      (with-character-style ('(nil :bold nil) stream)
        (format stream "~% NAME : ~a~%" (symbol-name nd-lk-nm)))
      (dw::with-output-truncation(stream)
        (cond ((typep nd-lk-nmh 'bart::node)
               (format stream "~a~%~%" (bart::doc nd-lk-nmh))
               (formatting-table (stream)
                 (formatting-column-headings (stream)
                   (formatting-cell (stream) "Values")
                   (formatting-cell (stream) "Belief")
                   (formatting-cell (stream) "Belief*")
                   (formatting-cell (stream) "Ext-evid")
                   (formatting-cell (stream) "All-evid"))
                 (do ((t-vals (bart::node-values nd-lk-nmh) (cdr t-vals))
                      (t-bels (bart::belief nd-lk-nmh) (cdr t-bels))
                      (t-bels* (bart::bel* nd-lk-nmh) (cdr t-bels*))
                      (t-evids (bart-util::termpro (bart::ext-evid nd-lk-nmh))
                               (cdr t-evids))
                      (all-evids (if (equal (car (bart::ext-evid nd-lk-nmh))
                                            (bart::unit-vec nd-lk-nmh))
                                     (bart-util::transpose (cdr (bart::ext-evid nd-lk-nmh)))
                                     (bart-util::transpose (bart::ext-evid nd-lk-nmh)))
                                 (cdr all-evids)))
                     ((null t-vals))
                   (formatting-row (stream)
                     (formatting-cell (stream)
                       (format stream "~a" (symbol-name (car t-vals))))
                     (formatting-cell (stream) (format stream "~4,3f" (car t-bels)))
                     (formatting-cell (stream) (format stream "~4,3f" (car t-bels*)))
                     (formatting-cell (stream) (format stream "~4,3f" (car t-evids)))
                     (formatting-cell (stream) (format stream "~a" (car all-evids)))))))
              ((typep nd-lk-nmh 'bart::link)
               (format stream "~%")
               (formatting-table (stream)
                 (formatting-column-headings (stream)
                   (formatting-cell (stream) "Pis")
                   (formatting-cell (stream) "Pi*")
                   (formatting-cell (stream) "Lambda")
                   (formatting-cell (stream) "Lambda*"))
                 (do ((t-pis (bart::link-pi nd-lk-nmh) (cdr t-pis))
                      (t-lams (bart::link-lambda nd-lk-nmh) (cdr t-lams))
                      (t-pis* (bart::link-pi* nd-lk-nmh) (cdr t-pis*))
                      (t-lams* (bart::link-lambda* nd-lk-nmh) (cdr t-lams*)))
                     ((null t-pis))
                   (formatting-row (stream)
                     (formatting-cell (stream) (format stream "~4,3f" (car t-pis)))
                     (formatting-cell (stream) (format stream "~4,3f" (car t-lams)))
                     (formatting-cell (stream) (format stream "~4,3f" (car t-pis*)))
                     (formatting-cell (stream) (format stream "~4,3f" (car t-lams*))))))
               (with-character-style ('(nil :bold nil) stream)
                 (format stream "~%~% Conditional Probability Matrix:~%"))
               (formatting-table (stream)
                 (formatting-column-headings (stream)
                   (formatting-cell (stream) "Values")
                   (dolist (parent-val (bart::node-values
                                         (gethash (bart::tnode nd-lk-nmh) bart::*myhash*)))
                     (formatting-cell (stream)
```

```
                    (format stream "~a" (symbol-name  parent-val)))))
                (do ((child-vals (bart::node-values
                                  (gethash (bart::bnode nd-lk-nmh) bart::*myhash*))
                                 (cdr child-vals))
                     (indpro-matrix (bart::indpro nd-lk-nmh) (cdr indpro-matrix)))
                    ((null child-vals))
                  (formatting-row (stream)
                    (formatting-cell (stream)
                      (format stream "~a" (symbol-name (car child-vals))))
                    (dolist (each-val (car indpro-matrix))
                      (formatting-cell (stream)
                        (format stream "~4,3f" each-val)))))))))
    (fresh-line stream)))

(defun get-new-evidence(prompt-string val-list default-vals)
  (multiple-value-list
    (dw:accept-values
      (do ((discriptions nil
                         (cons '(number :prompt ,(string (car temp-names))
                                        :default ,(car temp-vals))
                               discriptions))
           (temp-names val-list (cdr temp-names))
           (temp-vals default-vals (cdr temp-vals)))
          ((null temp-names) (nreverse discriptions)))
      :prompt (format nil "New External Evidence : ~a" prompt-string)
      :own-window t)))

(defun display-and-change(&optional (nd-lk-nm bart::*selected-node-or-link*))
  (let ((nd-lk-nmh (gethash nd-lk-nm bart::*myhash*))
        new-evid-supplied)
      (cond ((typep nd-lk-nmh 'bart::node)
             (setf new-evid-supplied
                   (get-new-evidence
                     (symbol-name nd-lk-nm)
                     (bart::node-values nd-lk-nmh)
                     (bart::unit-vec nd-lk-nmh)))
             (cond (new-evid-supplied
                    (push nd-lk-nm bart::*to-be-updated*)
                    (push nd-lk-nm bart::*to-be-updated**)
                    (setf bart::*equilibrium-p* nil)
                    (setf (bart::ext-evid nd-lk-nmh)
                          (cons new-evid-supplied
                                (bart::ext-evid nd-lk-nmh)))))))))


;;*************************************************************
#||
(defun display-and-change(&optional (nd-lk-nm bart::*selected-node-or-link*))
  (let ((stream (dw:get-program-pane 'node-display-pane))
        (nd-lk-nmh (gethash nd-lk-nm bart::*myhash*)))
    (if bart::*clear-each-time-p* (send stream :clear-history)
        (end-of-scroll-window stream))

    (dw::with-output-truncation(stream)          ;heading
      (cond ((typep nd-lk-nmh 'bart::node)
             (push nd-lk-nm bart::*to-be-updated*)
             (push nd-lk-nm bart::*to-be-updated**)
             (setf bart::*equilibrium-p* nil)
             ;; add a new unit vector at the end of ext-evid first to present
             ;; so the user can change that. Do this only if there is not already one.
             (if (equal (car (bart::ext-evid nd-lk-nmh))
                        (bart::unit-vec nd-lk-nmh))
                 nil
                 (setf (bart::ext-evid nd-lk-nmh)
                   (cons (copy-list (bart::unit-vec nd-lk-nmh))
                         (bart::ext-evid nd-lk-nmh))))
             (with-character-style ('(nil :bold nil) stream)
               (format stream "~% NAME : ~a~%"
                       (symbol-name nd-lk-nm)))
             (format stream "~a~% External Evidence:~%~%"
                     (bart::doc nd-lk-nmh))
             (formatting-table (stream :equalize-column-widths t)
               (do ((count1 0 (1+ count1))
                    (node-vals (bart::node-values nd-lk-nmh) (cdr node-vals))
                    (ext-evids (car (bart::ext-evid nd-lk-nmh))
                               (cdr ext-evids))
                    (all-evids (bart-util::transpose (cdr (bart::ext-evid nd-lk-nmh)))
```

```
                        (cdr all-evids)))
                  ((null node-vals))
                (formatting-row (stream)
                  (formatting-cell (stream :align :center)
                    (format stream "~a" (symbol-name (car node-vals))))
                  (formatting-cell (stream :align :center)
                    (present (car ext-evids) 'number :stream stream
                             :acceptably t
                             :location
                             (locf (nth count1 (car (bart::ext-evid nd-lk-nmh))))))
                  (formatting-cell (stream)
                    (format stream "~a" (car all-evids)))))))

          ((typep nd-lk-nmh 'bart::link)
           (setf bart::*condpro-changed-p* (cons nd-lk-nm bart::*condpro-changed-p*))
           (setf bart::*equilibrium-p* nil)
           (with-character-style ('(nil :bold nil) stream)
             (format stream "~% NAME : ~a~% Conditional Probability Matrix:~%~%"
                     (symbol-name nd-lk-nm)))
           (formatting-table (stream :equalize-column-widths t)
             (formatting-column-headings (stream)
               (formatting-cell (stream) "Values")
               (dolist (parent-val (bart::node-values
                                    (gethash (bart::tnode nd-lk-nmh) bart::*myhash*)))
                 (formatting-cell (stream :align :center)
                   (format stream "~a" (symbol-name  parent-val)))))
             (do ((count1 0 (1+ count1))
                  (count2 -1)
                  (child-vals (bart::node-values
                               (gethash (bart::bnode nd-lk-nmh) bart::*myhash*))
                              (cdr child-vals))
                  (indpro-matrix (bart::indpro nd-lk-nmh) (cdr indpro-matrix)))
                 ((null child-vals))
               (setf count2 -1)
               (formatting-row (stream)
                 (formatting-cell (stream)
                   (format stream "~a" (symbol-name (car child-vals))))
                 (dolist (each-val (car indpro-matrix))
                   (incf count2)
                   (formatting-cell (stream :align :center)
                     (present each-val 'number :stream stream
                              :acceptably t
                              :location
                              (locf (nth count2
                                         (nth count1
                                              (bart::indpro nd-lk-nmh))))))
                   ))))))))))
|#
;;; ****************************************************************************
;;;                          user interface
;;; ****************************************************************************
(defflavor bart-network-display-pane
        ()
        (dw::dynamic-window)
  :readable-instance-variables
  :writable-instance-variables
  :initable-instance-variables)

(defflavor bart-node-display-pane
        ()
        (dw::dynamic-window)
  :readable-instance-variables
  :writable-instance-variables
  :initable-instance-variables)

;;; Define a frame for the BaRT system
(defparameter *bart-command-menu-column-1*
              '("[a]dd"
                "[c]hange"
                "[e]val"
                " Explain"
                "[l]oad"
                "[p]ropagate"
                "[r]efresh"))
(defparameter *bart-command-menu-column-2*
              '(" Revert-net"
                "[s]elect&display"
                " Snapshot"
```

```
                "[t]argetnode"
                "[u]ser-modes"
                "[x]Exit"))

(defparameter *bart-interactor-character-style* '(:swiss :condensed-caps :normal))
(defparameter *bart-display-character-style* '(:swiss :condensed-caps :normal))
(defparameter *bart-heading-character-style* '(:swiss :bold-condensed-caps nil))

;;; This is a list of commands whose output is put into a display pane.
;;; Output from commands not in the list goes to the typeout window.
(defvar *bart-redisplay-hacking-commands* ())

(dw:define-program-framework bart
  :pretty-name "Bayesian Reasoning Tool (BaRT)"
  :select-key  #\\003
  :command-definer  t
  :command-table  (:inherit-from '("colon full command"
                                   "standard arguments"
                                   "standard scrolling")
                   :kbd-accelerator-p 't)
  :top-level (bart-top-level :prompt bart-prompt)
  :help bart-help
  :state-variables  ((title-pane)
                     (network-pane)
                     (node-pane)
                     (interactor-pane)
                     (global-parm-pane)
                     (menu-pane)
                     (network-file #p"local:>bart>data>reagan")
                     (network-loaded nil)
                    )
  :panes ((title :title
                 :height-in-lines 1)
          (network-display-pane :display
                                :flavor bart-network-display-pane
                                :margin-components
                                '((dw:margin-ragged-borders)
                                  (dw:margin-scroll-bar)
                                  (dw:margin-scroll-bar :margin :bottom)
                                  (dw:margin-borders :thickness 2)
                                  (dw:margin-whitespace :margin :left :thickness 20)
                                  (dw:margin-whitespace :margin :right :thickness 20)
                                  (dw:margin-whitespace :margin :bottom :thickness 20)
                                  (dw:margin-label :margin :top
                                                   :centered-p t
                                                   :style (:swiss :bold :small)
                                                   :string
                                                   "Belief Network Window")
                                  (dw:margin-whitespace :margin :top :thickness 20)
                                  )
                                :end-of-page-mode :truncate
                                :more-p nil
                                :redisplay-after-commands nil)
          (global-parm :display
                       :redisplay-function 'global-parm
                       :height-in-lines 2
                       :redisplay-after-commands t
                       :margin-components
                       '((dw:margin-borders :thickness 2)
                         (dw:margin-whitespace :margin :left :thickness 10)
                         (dw:margin-whitespace :margin :bottom :thickness 5)
                         (dw:margin-label
                           :margin :top
                           :centered-p t
                           :style (:swiss :bold :small)
                           :string "Global System Parameters")
                         (dw:margin-whitespace :margin :top :thickness 5)
                        ))
          (menu :command-menu
                :columns '(,*bart-command-menu-column-1*
                           ,*bart-command-menu-column-2*)
                :Menu-level :top-level
                :margin-components
                '((dw:margin-borders :thickness 2)
                  (dw:margin-whitespace :margin :left :thickness 30)
                  (dw:margin-whitespace :margin :bottom :thickness 10)
                  (dw:margin-label
                    :margin :top
```

```
                    :centered-p t
                    :style (:swiss :bold :small)
                    :string "Command Menu")
                   (dw:margin-whitespace :margin :top :thickness 10)
                  ))
          (node-display-pane :display
                             :flavor bart-network-display-pane
                             :margin-components
                             '((dw:margin-ragged-borders)
                               (dw:margin-scroll-bar :visibility :if-needed
                                                     :elevator-thickness 6)
                               (dw:margin-scroll-bar :margin :bottom
                                                     :visibility :if-needed
                                                     :elevator-thickness 6)
                               (dw:margin-borders :thickness 2)
                               (dw:margin-whitespace :margin :left :thickness 10)
                               (dw:margin-whitespace :margin :bottom :thickness 10)
                               (dw:margin-label :margin :top
                                                :centered-p t
                                                :style (:swiss :bold :small)
                                                :string "Node/Link Information Display")
                               (dw:margin-whitespace :margin :top :thickness 10)
                               )
                             :end-of-page-mode :truncate
                             :more-p nil
                             :redisplay-after-commands nil)
          (listener :listener
                    :default-character-style '(:fix :extra-condensed :normal)
                    :more-p nil
                    :margin-components
                    '((dw:margin-ragged-borders)
                      (dw:margin-scroll-bar :visibility :if-needed
                                            :elevator-thickness 6)
                      (dw:margin-scroll-bar :margin :bottom     ;
                                            :visibility :if-needed
                                            :elevator-thickness 6)
                      (dw:margin-white-borders :thickness 2)
                      (dw:margin-whitespace :margin :left :thickness 10)
                      (dw:margin-label
                        :margin :top
                        :centered-p t
                        :style (:swiss :bold :small)
                        :string "Interaction Window")
                      (dw:margin-whitespace :margin :top :thickness 10)
                      )
                    :height-in-lines 20))
  :configurations
  '((dw::main
     (:layout
       (dw::main :column title col-2)
       (col-2 :row network-display-pane column-1)
       (column-1 :column global-parm menu node-display-pane listener))
     (:sizes
       (dw::main (title :limit (1 2 :lines) .05) :then
                 (col-2 :even))
       (col-2 (column-1 :limit (40 80 :characters listener) 0.35) :then
              (network-display-pane :even))
       (column-1 (global-parm 2 :lines)
                 (menu :limit (5 8 :lines) :ask-window self :size-for-pane menu)
                 (node-display-pane 0.4) :then (listener :even))))))


(defgeneric bart-top-level (program &rest options)
  "This top-level function exists to get the help text printed out at the start and
   to allow us to use the state variables to store the programs window panes for
   later use with the graphics stuff."
  )

(defmethod (bart-top-level bart) (&rest options)
   (setq network-pane (dw:get-program-pane 'network-display-pane))
   (setq node-pane (dw:get-program-pane 'node-display-pane))
   (setq interactor-pane (dw:get-program-pane 'listener))
   (setq global-parm-pane (dw:get-program-pane 'global-parm))
   (setq menu-pane (dw:get-program-pane 'menu))
   (bart-help self interactor-pane nil) ;print the help message
   (apply #'dw:default-command-top-level self options)) ;and run the standard loop

;;; custom prompt is used beacuse the default prompt uses up so much of the width of
```

```
;;; the comand pane
(defun bart-prompt (stream ignore)
  (with-character-style ('(:fix :roman :normal) stream)
    (write-char #\arrow:right-fat-arrow stream)
    (write-char #\space stream)))

;;; This code is run by the program top-level and by the key
(defun bart-help (program stream string-so-far)
  (ignore program)
  (when (every (lambda (x) (char-equal x #\space)) string-so-far)
    (format stream "~&You are typing a command to the~@
                    Bayesian Reasoning Tool.~@
                    Use the single key equivalents in [ ]~@
                    in the menu above, or click on one with the mouse.~@
                    Note: you must issue a Load command before you do~@
                          anything else with the network.~&")
          (send-if-handles stream :increment-cursorpos 0 6)))

;;; The method for setting the global-parm
;;; Need to fix for window size and add the data file name for the network
(defmethod (global-parm bart) (stream)
  (format stream "   Network Name: ")
  (with-character-style
    ((if bart::*equilibrium-p* '(nil :bold nil) '(nil :italic nil)) stream)
    (if network-loaded
        (format stream "~A" network-file)))
  (format stream "~%   User Modes: ")
  (with-character-style
    ((if bart::*step-p* '(nil :bold nil) '(nil :italic nil)) stream)
    (format stream " Step "))
  (with-character-style
    ((if bart::*debug-mode* '(nil :bold nil) '(nil :italic nil)) stream)
    (format stream " Debug ")))
#||
(defmethod (global-parm bart) (stream)
  (si:x-centering-in-window (stream)
    (scl:present "Bayesian Reasoning Tool (BaRT)" 'string :stream stream)
    (with-character-style ('(nil :italic nil) stream)
      (if network-loaded (format stream "~%~A~%" network-file))))
    (formatting-table (stream)
      (formatting-row (stream)
        (formatting-cell (stream)
          (with-character-style
            ((if bart::*equilibrium-p* '(nil :bold nil) '(nil :italic nil)) stream)
            (format stream "   Equilibrium ")))
        (formatting-cell (stream)
          (with-character-style
            ((if bart::*step-p* '(nil :bold nil) '(nil :italic nil)) stream)
            (format stream " Step ")))
        (formatting-cell (stream)
          (with-character-style
            ((if bart::*debug-mode* '(nil :bold nil) '(nil :italic nil)) stream)
            (format stream " Debug "))))))
||#
;;; add a node or link..
(define-bart-command (com-add
                        :keyboard-accelerator #\a
                        :menu-accelerator "[a]dd")
                      ()
  (format t " Add -- Not yet implemented. "))

;;; change evidence in case of a node and conditional probability in case of a link
(define-bart-command (com-change
                        :keyboard-accelerator #\c
                        :menu-accelerator "[c]hange")
                      ()
  (display-and-change (accept 'bart::node)))



;;; Evaluate an expression
(define-bart-command (com-eval
                        :keyboard-accelerator #\e
                        :menu-accelerator "[e]val")
                      ()
  (let ((result (multiple-value-list
                  (eval (accept '((sys:expression))
                                :prompt "Eval ->"
```

```
                                :default nil
                                :provide-default t
                                :display-default nil)))))
    (fresh-line)
    (loop for x in result
          do (present x)
             (fresh-line))))

;;; Explain --
(define-bart-command (com-explain
                      :menu-accelerator " Explain")
                     ()
  (format t "Explain -- Not yet implemented."))

;;; Load -- Load in a data file and  reset everything first.
;;; Then draw the network, update and save the equilibrium state."
(define-bart-command (com-load
                      :keyboard-accelerator #\l
                      :menu-accelerator "[l]oad")
                     ((network-file 'cl:pathname
                                    :prompt "named"
                                    :default (bart-network-file dw:*program*)
                                    :confirm t))
  (setq *program* dw:*program*)
  (bart::do-reset)                              ; remove all old information
  (clear-bart-window *program* 'network-display-pane)   ;clear network window
  (clear-bart-window *program* 'node-display-pane)      ;clear node/link window
  (setf (bart-network-file dw:*program*) network-file)  ;remember the file for later
  (load network-file)                           ; load data file
  (cond (bart::*snapped-input-file-p*)
        (t
         (bart:init-net)                        ; initialize the network
         (bart:find-xys)                        ; find relative positions of each node
         (find-pos)                             ; find absolute co-ordinates of each node
         (setf bart::*selected-node-or-link*
               (setf bart:*targetnode* (car bart::*all-nodes*))
               bart::*first-pass-p* t
               bart::*to-be-updated* bart::*all-nodes*
               bart::*to-be-updated** bart::*all-nodes*
               bart::*equilibrium-p* nil)
         (bart:updateall-b)))                   ; to bring it into equilibrium
  ;; now display the net and shade the nodes depending on the the importance.
  (display-net)
  (setf (bart-network-loaded dw:*program*) t))

;;; Propagate -- update the network after changes and redisplay
(define-bart-command (com-propagate
                      :keyboard-accelerator #\p
                      :menu-accelerator "[p]ropagate")
                     ()
  ;; see if *condpro-changed-p* is set. if so reset the joint conditional probability
  ;; and set the flag to nil and then update.
  (cond (bart::*condpro-changed-p*
         (let (changed-links-bottom-nodes)
           (bart::re-init-net bart::*condpro-changed-p*)
           (setf changed-links-bottom-nodes
                 (mapcar #'(lambda(x) (bart::bnode (gethash x bart::*myhash*)))
                         bart::*condpro-changed-p*)
                 bart::*to-be-updated*
                 (append changed-links-bottom-nodes bart::*to-be-updated*)
                 bart::*to-be-updated**
                 (append changed-links-bottom-nodes bart::*to-be-updated**)
                 bart::*condpro-changed-p*
                 nil))))
  (bart::updateall-b)
  (display-net))

;;; Refresh -- refreshes the screen
(define-bart-command (com-refresh
                      :keyboard-accelerator #\r
                      :menu-accelerator "[r]efresh")
                     ()
  (clear-bart-window *program* 'network-display-pane)   ;clear network window
  (clear-bart-window *program* 'node-display-pane)      ;clear node/link window
  (display-net))

;;; Revert-net -- keeps the network in the initial equilibrium state.
(define-bart-command (com-revert-net
```

```
                        :menu-accelerator " Revert-net")
                      ()
  (cond (bart::*first-pass-p*
         (format t "~% Initial equilibrium has not been reached yet. "))
        (t
         (bart::revert-net)
         (display-net))))

;;; Select&display -- selects a node or link and displays it in node/link window.
(define-bart-command (com-select&display
                        :keyboard-accelerator #\s
                        :menu-accelerator "[s]elect&display")
                      ()
  (setf bart::*selected-node-or-link*
        (accept '(or bart::node bart::link))))
  (display-a-node-or-link))

;;; Snapshot -- saves the results/network in a file.
(define-bart-command (com-snapshot
                        :menu-accelerator " Snapshot")
                      ()
  (format t "Snapshot -- Not yet implemented."))        ; call the save routine here


;;; Targetnode -- sets the targetnode and updates the dependency relations and
;;; display.
(define-bart-command (com-targetnode
                        :keyboard-accelerator #\t
                        :menu-accelerator "[t]argetnode")
                      ()
  (setf bart:*targetnode* (accept 'bart::node))
  (bart:find-benefit-factors)
  (display-net))

;;; User-modes -- to set the user modes
(define-bart-command (com-user-modes
                        :keyboard-accelerator #\u
                        :menu-accelerator "[u]ser-modes")
                      ()
  (dw:accept-variable-values
    '((bart::*step-p*     "Step mode " boolean)
      (bart::*debug-mode* "debug mode" boolean)
      (bart::*clear-each-time-p* "clear node/link window each time" boolean))
    :own-window t
    :prompt "Select User Modes"))

;;; Exit --
(define-bart-command (com-xexit
                        :keyboard-accelerator #\x
                        :menu-accelerator "[x]Exit")
                      ()
  (send (dw::find-program-window 'bart
                                 :create-p nil)
        :bury))

;;; clear all of the network display pane
(defgeneric clear-bart-window
       (program pane)
 "Clear the window in question and its history.
  The window is given as defined in the panes option of the program definition.
  e.g., (clear-bart-window bart 'network-display-pane)"
  )
(defmethod (clear-bart-window bart)(pane)
           (send (dw:get-program-pane pane) :clear-history))

(defun net-work-state-in-lisp()
  ;;dump all globals
  ;;dump all nodes
  ;;dump all links
  )
```

FINAL REPORT

Deliverable No. A007

Prepared by:

JAYCOR

Prepared for:

Naval Research Laboratory
Washington, DC 20375-5000

In Response to:

Contract #N00014-86-C-2352

26 May 1988

1. INTRODUCTION

JAYCOR is pleased to submit this final report summarizing the tasks performed by JAYCOR at the Navy Center for Applied Research in Artificial Intelligence (NCARAI) and other offices of the Naval Research Laboratory under the first phase of Contract #N00014-86-C-2352. This report gives a brief overview of the work performed to meet the tasks, location on computers both at the NCARAI and on the main NRL campus where software written to meet the requirements of particular tasks is stored, specific documentation of any aspects of the software which may be unusual or possibly nonintuitive to use, and pertinent information on any problems encountered during the performance of the tasks.

The report is organized on a task by task basis, each task briefly explained and the work summarized. Any software produced, obtained, or otherwise installed to satisfy task requirements is available in source form on NRL computers or archive tapes. In addition to this summary final report, the reader is referred to the numerous monthly reports submitted over the entire period of the contract to more thoroughly document the work performed.

2. TASKS AND WORK PERFORMED

The work of this phase of the contract took on a number of different aspects: research using a Symbolics Lisp machine into Bayesian inferencing, the implementation of a software tool using the results of this research (an ongoing project), conversion and implementation of CK-LOG code and the design and implementation of a user interface for it, the design of a database and initial implementation of it, and the archiving of all project efforts.

The subsections below briefly summarize the approach taken to each task and provide pointers to further information and the location of implemented software, as well as observations on the direction that research into the varied topics has taken.

2.1. KNOWLEDGE ACQUISITION TOOL DESIGN

This task entailed an enhancement of work that was performed on a previous project. A graphics oriented user interface that provides both textual and pictorial representations for nodes on the network and the network as a whole was implemented. The tool also provides for the visual checking for cycles in the network as well as easing the specifying of relational dependencies.

A report containing source code was submitted earlier in the contract period. The reader is referred to this report to more thoroughly document some of the work performed to meet the requirements of this task.

2.2. BAYESIAN REASONING TOOL IMPLEMENTATION

The work performed to meet the requirements of this task is a continuing effort, evolving toward a general purpose reasoning tool which will be of great assistance to the NCARAI's ship classification project. This part of the contract work, done in

coordination with NRL researchers, has generated considerable interest in the research community and has been demonstrated repeatedly to them. Installed on a Symbolics Lisp machine under the PCL environment, the tool is continuing to evolve into a very powerful facility for analysis.

A report (and code) has been submitted. The reader is referred to this for more information on this task.

2.3. LOW LEVEL IMAGE PROCESSING SOFTWARE

Part way through the first phase of this contract a no-cost modification was made. One part of this modification was to implement certain basic low level image processing functions. This has been done. Thresholding, edge detection, and image output routines have been written in C and are present both on disk and archived on tape. To maintain as much generality as possible, the programs read raw images from the standard input and write raw images to the standard output. In this way, chains of functions can be invoked with the output of one piped to the input of another.

2.4. CK-LOG SYSTEM CONVERSION

All code for the CKLOG system that existed previously was converted from ELISP to CommonLisp. This conversion also resulted in it being ported to the LMI Lisp machines at the AI Center.

A report documenting this port and the interface below has been submitted. The reader is referred to this report for further information.

2.5. CK-LOG USER INTERFACE

A user interface to underlying data structures was written to meet the requirements of this task. This interface allows simple yet powerful graphical and textual methods to be used to both create and view information in the system.

A report containing figures and source code has been submitted. This relates a much more thorough understanding of the task's efforts. The reader is referred to this report to more thoroughly document some of the work performed to meet the requirements of this task.

2.6. DATABASE DESIGN

A database system was designed that used a PC-based communications link to a centralized bibliographic system. This system allows the downloading of bibliographic entries to the NCARAI, then from the PC to the VAX. Because the entries are genuine bibliographic entries as used by the Library of Congress, few changes were needed to port them to the VAX.

A report describing the dialed up database and the methods used to download entries has been submitted. The reader is referred to this report to more thoroughly document the task's efforts.

2.7. DATABASE IMPLEMENTATION

Because the database existed offline in a suitable form (the OCLC and DIALOG systems, for example), the initial implementation onsite at the NCARAI entailed installation of commercial software on the NCARAI's library PC. This has proven to be extremely valuable to the Center already, with multiple searches of the database taking place daily. The next phase of the contract will see the actual installation on the VAX of software for anyone to access the database as contained at NCARAI on their own terminals.

The report mentioned in the previous section more thoroughly documents the work of this task. the reader is referred to this report for more information.

3. CONCLUDING REMARKS

As with any research related efforts, the work performed to meet the requirements of the tasks of this contract continues at this writing even though the tasks themselves have been met. Systems developed then are still evolving into something "better" as ideas are conceived and discarded. JAYCOR believes this new work will build upon the old in a manner beneficial both to the Navy and the Naval Research Laboratory's mission. We look forward to continuing our relationship to the Navy's AI Center with Phase II of this contract and in the future .

DATABASE COMPLETION

Deliverable No. B003

Prepared by:

JAYCOR

Prepared for:

Naval Research Laboratory
Washington, DC 20375-5000

In Response to:

Contract #N00014-86-C-2352

26 May 1988

NCARAI Library Database

A database containing information about the books in the NCARAI library has been set up on the Vax 11/780. This database contains 1000 records, where each record has the title, author, publisher, publication date and call number of a book. This information was originally entered on an IBM PC by the librarian, then transfered electronically to the Vax. The data was run through program filters to remove special formatting characters produced by the IBM PC database programs.

A program BOOKSEARCH was written for the Vax which allows a user to search the database for a pattern, such as author or title. Records matching this pattern are run through another filter program to make the information readable by the user. The user manual page for this program is presented in this report, along with the source code for the filter programs. The code was written in the C language.

As new books are received by the librarian, they will be entered into the IBM PC database, and then copied to the Vax database.

NAME

booksearch – Search for a pattern in the NCARAI library database.

SYNOPSIS

booksearch [**pattern to search for**]

DESCRIPTION

booksearch is a command which scans the NCARAI library database for a pattern. This pattern will match any name, title, published date, or call number in the database. The search is case insensitive.

EXAMPLES

booksearch minsky

will list all the books written by Minsky currently in our library.

booksearch "artificial intel"

will list all books with "artificial intel" in the title.

FILES

/aic2/library/db/book01.out

```
#!/bin/sh
# booksearch - scan NCARAI library database for a string.

if [ "$#"  != "1" ]; then
        echo "Error, Usage: booksearch [string]"
        exit 1
fi

/usr/bin/fgrep -i "$1" /aic2/library/db/book01.out | /aic2/library/db/bin/fsplit
```

```
/* Filter for records from finder database. Splits records to make them */
/* more readable. */

#include <stdio.h>
main()
{
  char call_no[31], junk[100], author[31], title[151], pub[31], pub_date[5];
  char *spaces = "    ";
  int i;

  while(scanf("%30c",call_no) != EOF) {
        i = strlen(call_no) - 1;
        while(call_no[i] == ' ')call_no[i--] = '\0';
        printf("\n%s\n",call_no);

        scanf("%6c",junk);

        scanf("%30c",author);
        i = strlen(author) - 1;
        while(author[i] == ' ')author[i--] = '\0';
        printf("%s\n",author);

        scanf("%150c",title);
        i = strlen(title) - 1;
        while(title[i] == ' ')title[i--] = '\0';
        printf("%s\n",title);

        scanf("%30c",pub);
        i = strlen(pub) - 1;
        while(pub[i] == ' ')pub[i--] = '\0';

        scanf("%5c",pub_date);
        printf("%s\n\n",pub_date);
      }
}
```

```
/* Clean up FINDER segment one after PC to VAX transmission. */

#include <stdio.h>
/* If null found, skip next 5 chars, output newline. */
main()
{
  int i;
  char c;
  while ((c = getchar()) != EOF)
    if (c == 000) {
      for (i=0; i<5; i++) getchar();
      putchar('\n');
    }
    else
      putchar(c);
}
```

```
/* Clean up FINDER segment two after PC to VAX transmission. */

#include <stdio.h>
/* if null found, skip next nulls, then output newline */
main()
{
  char c;
  while ((c = getchar()) != EOF) {
    if (c == 000) {
      while ((c=getchar()) == 000);
      putchar('\n');
      putchar(c);    }
    else putchar(c);
  }
}
```

```
/* Clean up FINDER segment three after PC to VAX transmission. */

#include <stdio.h>
/* if null found, output newline */
main()
{
  char c;
  while ((c = getchar()) != EOF)
    if (c == 000) putchar('\n');
    else putchar(c);
}
```