IDEAL: Influence Diagram Evaluation and Analysis in Lisp
Documentation and Users Guide

Sampath Srinivas
srinivas@rpal.rockwell.com
Jack Breese
breese@rpal.rockwell.com
Rockwell International Science Center
Palo Alto Laboratory
444 High Street
Palo Alto, CA 94301
(415) 325–7174
(415) 325–1871

Bug reports requested

Technical Memorandum No. 23

Contents

1 Introduction

1.1 Overview

This document describes a system for building and evaluating influence diagrams. It is designed to provide a set of data structures and algorithms useful for solving probabilistic and decision-theoretic inference problems. The system provides a set of substrate capabilities for use in probabilistic expert systems and real-time applications.
An influence diagram is an acyclic directed graph whose nodes correspond to the state variables of interest in a decision-theoretic problem [3]. The arcs between the nodes represent influences between these variables. Associated with each node there is a set of possible states that it can take on. A node may be a chance node, which means the variable it represents is uncertain. A decision node represents a variable over which a decision maker has control, in that she can decide what state it takes. A value node encodes the preference of a decision maker in terms of other nodes in the diagram in terms of a scalar value or utility function.
Influence diagrams can be manipulated to solve decision problems in terms of expected value decision making or to perform probabilistic inference. Belief networks are influence diagrams consisting solely of chance nodes [6]. A number of algorithms can be used to perform probabilistic inference in belief networks. Readers are referred to [1, 3, 11, 6] for details on influence diagrams and belief networks and their evaluation.
The emphasis in this document is to provide sufficient background for development and solving influence diagrams and belief networks using IDEAL. The system provides:

- Data structures for representing influence diagrams and belief networks.
- Facilities for creating and editing influence diagrams and belief networks.
- Facilities for copying, saving (to file) and loading influence diagrams and belief networks.
- Utilities that are of use in coding influence diagram manipulation algorithms etc.
- Routines that perform some basic transformations of influence diagrams.
- Algorithms for performing inference in influence diagrams and inference and belief propagation in belief networks.
- Influence diagram evaluation algorithms.

The IDEAL system is entirely in Common Lisp and can be run on any Common Lisp platform. Some sub-sections of this document are titled "Implementation notes". These sections are of interest only to somebody who is interested in internal details of IDEAL's operation. Instructions on loading IDEAL are distributed with the code.
A CLIM based graphic user interface called IDEAL-EDIT is available and is separately documented [16].

2 Data structures

A node in an influence diagram represents a variable involved in the decision model represented by the diagram. Nodes can be of three types: chance, decision or value. In addition, nodes are also distinguished by the kind of relationship they have to their predecessors: deterministic or probabilistic. Chance nodes may have either deterministic or probabilistic relationships to their predecessors. Value functions of value nodes and decision node policies are deterministic relationships. Along another dimension, the relationship between a node and its predecessors can be either continuous or discrete. Only discrete relationships have been implemented in this system. The influence diagram is represented as a list of node structures. A diagram that contains only chance nodes is also known as a belief net. Such diagrams are used exclusively for probabilistic reasoning. The following subsections describe the basic data structures used in IDEAL. Each of the structure types have many fields in addition to those documented. Some of these fields are used by the belief-net algorithms (See Sec 9.2) and some are used for book keeping.

2.1 Nodes

The fields of a `node` structure are:

name A name for the node. Must be a Lisp symbol.

type Can be `:chance`, `:decision` or `:value`.

predecessors A list of predecessor nodes (the actual node structures).

successors A list of successor nodes.

distribution The distribution structure of the node is stored in this field. This distribution may be a probability distribution for a chance node, a decision policy for a decision node or a value distribution for a value node.

state This field stores the state of the node (if known).

2.2 Node distributions

The distribution of a node is represented by a structure type defined for this purpose. Presently only discrete distributions are implemented. The system does not handle continuous nodes. A discrete distribution is represented by a structure of type `discrete-dist`. A `discrete-dist` structure has the following fields:

relation-type Can be probabilistic (`:prob`) or deterministic (`:det`). The relation type field is automatically set to `:det` in the case of decision nodes and value nodes by the influence diagram creation functions (described later).

state-labels A discrete node can take one of a finite number of states. Each of these states is represented by a state label structure (See below). This field of the distribution structure stores a list of state label structures, one for each state that the node can take. Value nodes, as implemented, have no explicit states and so this field contains `nil` in the case of a value node.

number-of-states The number of states the node can take, which is the same as the number of state labels the node has.

array The array in which the actual distribution is stored. The array stores probabilities (numbers) in the case of a probabilistic chance node, state labels structures in the case of a deterministic chance node and values (numbers) in the case of a value node. In the case of a decision node the array stores the decision node policies and expected values after the influence diagram has been solved. Each location in the array contains a dot pair, the car of which is the action to be taken (a state label of the decision node) and the cdr of which is the expected value of this decision.

2.2.1 Noisy Or Distributions

IDEAL also implements nodes with a special type of discrete probabilistic chance distribution called a Noisy Or distribution. More details about Noisy Or nodes are in Section 10.

2.3 State label structures

Each state of the node is represented by a (state) `label` structure. A `label` structure has the following fields:

name A name for the label. Must be a symbol.

node This field stores the node to which the label structure belongs. It serves as a backpointer to the node owning the label.

2.4 Implementation details

The `node` structure has a field called `unused-slot` which is not used by IDEAL. This field is reserved for use by programmers who may wish to associate their own structures with influence diagram nodes.

3 Creating influence diagrams

A CLIM based graphic user interface to IDEAL called IDEAL-EDIT is available and is separately documented [16]. When a user interface is not available influence diagrams can be created using the following functions. Existing diagrams can edited using the functions described in Section 4.2.1. These methods for creating influence diagrams are quite crude since they have been developed with portability as a priority rather than ease of use.

`*diagram*` *[variable]*

The diagram bound to `*diagram*` is the default diagram for many functions that need a diagram as input. If this default is used by a function it is mentioned in its documentation in this manual. In addition, a special dispatch macro makes it easy to reference nodes in the diagram bound to `*diagram*` (See Sec 4.1). This variable exists only as a user convenience. It is not a global flag or a conventional global variable. It is not embedded in any code except as a default for an optional argument.

`create-complete-diagram &key` *number-of-nodes belief-net binary-nodes binary-node-labels noisy-or* *[function]*

Makes a complete diagram interactively and returns it. *number-of-nodes* if specified, specifies the number of nodes in the influence diagram to be created. If not specified, the user is prompted for the number. When *belief-net* is specified as `t`, the function assumes that every node in the diagram being created is a probabilistic chance node. If *binary-nodes* is specified as `t`, the function assumes that each node in the diagram has two states. *binary-node-labels*, when specified is assumed to be list of label names for the binary nodes. The list should consist of two symbols or numbers. All the nodes are assigned these two symbols as the names of the states. When *noisy-or* is specified as *t* all nodes are created as noisy-or nodes [1].

`create-belief-net &key` *number-of-nodes binary-nodes noisy-or* `(`*binary-node-labels* `'(:TRUE :FALSE))` *[function]*

This function is similar to `create-complete-diagram`. It creates a belief net (all nodes are assumed to be probabilistic chance nodes). When *binary-nodes* is `t` and *binary-node-labels* is not specified the names of the (two) states of each node are assumed to be `:TRUE` and `:FALSE`. When *noisy-or* is specified all nodes are created as noisy or nodes.

`create-node-set` *[function]*

Creates a set of nodes interactively and returns them. The name, type, label names etc are queried for.

The set of returned nodes is a valid belief net, albeit with no arcs, default uniform distributions for all chance nodes, all inhibitor probabilities set to zero and default settings for noisy or deterministic functions.

`create-arcs &optional (`*diagram* `*diagram*)` *[function]*

Takes a diagram as input then interactively queries user for the arcs in the diagram. Returns modified diagram with arcs.

This function deletes all existing arcs in the diagram before querying the user for new arcs. It is meant to be used on the output of the previous function.

`create-distributions &optional (`*diagram* `*diagram*)` *[function]*

Sets the distribution on each of the nodes in the diagram interactively and then returns the diagram.

You might want to do each stage of the influence diagram creation separately so that, if a mistake is made, only that stage has to be repeated. So instead of doing this:

```
 (setf diag
(create-complete-diagram))
```

You might do this:

[1] See Section 10 for details on noisy or nodes.

```
(setf diag (create-node-set))
 ...
(setf diag1 (create-arcs diag)))
 ...
(setf diag2 (create-distributions diag1))
```

4 Editing diagrams: Programmer Interface

This section describes functions that are used to edit influence diagrams.

4.1 User interface functions

`#n` *[Dispatch macro character]*

The letter `n` has been declared as a dispatch macro character to access nodes in diagrams. To access the node of name `aaa` in the diagram bound to the symbol `ddd` just type `'#n(aaa ddd)`. When the reader encounters this construct it automatically returns the appropriate node. If the second argument is not specified (eg. `'#n(aaa)`) the node is retrieved from the diagram bound to `*diagram*`. One some implementations it is not necessary to type the quote character and so the form `#n(aaa ddd)` can be used instead of `'#n(aaa ddd)`.

`display-dist` *node* *[function]*

Prints the distribution of *node* in a reasonably pretty format. Returns no values.

`diag-display-dist &optional (`*diagram* `*diagram*)` *[function]*

Same as above done for each node in *diagram*.

`diag-display-links &optional (`*diagram* `*diagram*)` *[function]*

Displays the predecessors of each node in the diagram.

4.2 Modifying diagrams within Lisp

4.2.1 High level functions

The following functions allow the user to make changes in a diagram. The functions usually return a modified diagram that is consistent (See Sec 7.3). If the returned diagram is inconsistent the system gives a warning to alert the user. These functions are meant to edit diagrams before they are solved. If the functions are used on a solved diagram, consistency of the decision policies on the decision nodes is not maintained. Except for the last two functions in this section, all the other functions either do not require interactive input or have a default mode which can be used instead of interactive input. Thus, they can be used embedded within code.

`add-arcs` *node pred-list* *[function]*

Arcs are added from each of the nodes in *node-list* to the node *node*. An arc to *node* is added from the first node in *pred-list*, then the second node and so on. Arcs that cause cycles are not added.

This function returns *node*. The distribution of *node* is updated to reflect the addition of these arcs. The new distribution of *node* is set such that *node* is functionally independent of its new predecessors, i.e. the probability associated with any particular state and conditioning case is the same irrespective of the state of a new predecessor in the conditioning case.
If *node* is a noisy or node[2], then all inhibitor probabilities for every new predecessor default to 0. The noisy or deterministic function is recomputed in the case of `:BINARY` and *:NARY* subtypes. In the case of the *:GENERIC* subtype the existing function is extended to be independent of the state of the new predecessors. The

`delete-arcs` *node pred-list* *[function]*
Arcs from each of the nodes in *pred-list* to *node* are deleted. If an arc does not exist from any of the nodes in *pred-list* to *node* the user is given a warning (though this does not affect the deletion of the arcs from the other predecessors). The node N (*node*) is assigned a new default distribution in the following manner:

- The 'centre-most' state s_{i_c} of each of the predecessors P_i of N in *pred-list* is chosen. This is done on the basis of the ordering induced by sequence in which the state labels were first created.
- For each new conditioning case and each state of N the probability is assigned as follows:
  $P_{new}(N = s_j|\text{cond-case}) = P_{old}(N = s_j|\text{cond-case}, \bigcup_i\{(P_i = s_{i_c})\})$
  where s_j represents any state of N and i indexes into the predecessor list of N.

Effectively, the new distribution of *node* is obtained by slicing the old distribution along the dimension of each predecessor at the coordinate of the 'center-most' state. This function returns *node*.
When *node* is a noisy or node, the deterministic function is assigned some random default value if the subtype is `:GENERIC`. Otherwise the function is recomputed.

`add-node` *diagram* `&key` *name type relation-type state-labels noisy-or noisy-or-subtype* *[function]*
Creates a new node for the diagram *diagram*. The new node is returned. There is extensive input checking. The modified diagram (i.e. including the new node) is consistent. The new node has no predecessors or successors. The distribution of the node is uniform if it is a probabilistic chance node, the value is uniformly zero if it is a value node and if it is a deterministic chance node, all cases map to the 'center-most' state of the node. To add predecessor and successors to the new node use the function `add-arcs`.
When *noisy-or* is specified as `t` then the node is created to be a noisy-or node of subtype *noisy-or-subtype*. *noisy-or-subtype* should be one of `:BINARY`, `:NARY` and `:GENERIC`.
The function returns two values: The first value is the updated diagram including the new node and the second value is the new node itself.

[2]See Section 10 for details on noisy or nodes.

`delete-node` *node* `&optional (`*diagram* `*diagram*) &key (`*default* `t)` *[function]*
This function deletes the node *node* from the diagram *diagram* and returns the modified diagram. Deletion of a node modifies the distributions of all the successors of the node. If the *:default* keyword argument is non-nil the system deletes the arc from the node to each successor using the function `delete-arcs`. The successor's new distribution is set by default by `delete-arcs` (see above). If not, it sets the distributions interactively. If the deleted node is a decision node which has both a preceding and succeeding decision, an arc is added from the decision preceding the deleted node to the decision succeeding the deleted node. This is done to preserve the chronological ordering and consistency of the diagram.

`add-state` *node* `&optional` *name* *[function]*
Adds a state to the possible states of the node *node* and then returns it. Adding a state to a node affects the distributions of the the successor nodes and the node *node*. `add-state` sets all these new distributions by default. For the node *node*, P(node=new-state|cond-case) is set to 0 for all conditioning cases. All other probabilities remain the same. For each successor node, the probability distribution of the successor node, given any conditioning case involving *node* in its new state, is made uniform. All other probabilities of the successor node remain the same. If the successor node is a value node its value, given any conditioning case involving *node* in its new state, is set to 0. All other values remain the same. These default assignments of values and probabilities guarantee that the diagram is consistent after the change. If any of the successors is a noisy-or node then it is given a inhibitor probability of 0 for the new state of *node*. The deterministic function of the successor is recomputed for `:BINARY` and `:NARY` noisy-or nodes, it is set to a default value for `:GENERIC` noisy-or nodes.
When *node* is a noisy-or node its deterministic function is recomputed or assigned a default value (in the case where the subtype is `:GENERIC`).
If you need something other than these default assignments use other editing functions after using `add-state`.

`delete-state` *node* `&optional` *state* *[function]*
Deletes a state from the possible states of the node *node* and then returns it. Deleting a state affects the distribution of the *node* and the successors of *node*. `delete-state` sets these new distributions by default. This is done as follows:

- Let the number of states of *node* be n and the state that is deleted by s_d. The new probabilities for *node* are:
  $P_{new}(s_i|\text{ cond-case}) = P_{old}(s_i|\text{ cond-case}) + P_{old}(s_d|\text{ cond-case})/(n-1)$ where $s_i \neq s_d$
  To put it simply, for each conditioning case the probability associated with s_d is distributed equally over the other states of *node*.

- For each successor node the same probabilities (or values) are retained except that all probabilities (values) associated with conditioning cases that involve the state s_d of *node* are dropped in the new distribution.

If any of the successors is a noisy-or node its deterministic function is recomputed or set to a default value. If *node* is a noisy or node, its deterministic function is recomputed or set to a default value. The default probability and value assignments guarantee that the diagram stays consistent.

`edit-distributions &optional (`*diagram* `*diagram*)` *[function]*
Interactively edits the distributions of each of the nodes in *diagram*. Policies of decision nodes cannot be edited. *diagram* is returned.

`edit-node-distribution` *node* *[function]*
Interactively edits the distribution of the node *node*. The policies of decision nodes cannot be edited. *node* is returned.

5 Copying and saving diagrams

This section describes functions for copying and saving influence diagrams. Many of the functions also take a *clique-diagram* argument. This argument is used in conjunction with the clustering algorithm (See Sec 9.5). It can be ignored under other circumstances.

5.1 Copying diagrams

`copy-diagram &optional (`*diagram* `*diagram*)` *clique-diagram* *[function]*
Returns a copy of *diagram* and if a clique diagram corresponding to the diagram is specified it returns a copy of the clique diagram as a second value. The new diagram and the old diagram do not share *any* structure, i.e. `copy-diagram` does a complete recursive copy. The same applies to the clique diagram. Making a recursive copy is complicated by the fact that the original diagram has circular references (in the predecessor/successor lists of each node and in the distributions of deterministic chance nodes). `copy-diagram` keeps track of these references and makes sure the copy has them too.

5.2 Saving and retrieving diagrams

Destructive changes are made to the diagram when it is manipulated and so it is a good idea to save diagrams to file before manipulating them. Ideally, a diagram should be saved to file as soon as it is created. It is also a good idea to keep a 'master copy' of the diagram in your lisp world and make manipulations only on copies of the diagram.

`save-diagram` *filename* `&optional (`*diagram* `*diagram*)` *clique-diagram*
`(`*default-path-name* `*default-diag-path-name*)` *[function]*
Writes the diagram *diagram* (and the *clique-diagram*, if specified) to the file of name *filename*. *filename* is merged with *default-path-name* to obtain the pathname of the file to which the diagram is written.

`load-diagram` *filename*
`&optional (`*default-path-name* `*default-diag-path-name*)` *[function]*
Reads and returns an influence diagram from the file whose pathname is obtained by merging *default-path-name* and *filename*. A *clique-diagram* is returned as a second value if a clique diagram is found in the file. When using `load-diagram`, if you get an error message saying that the diagram file is not in the current format, refer to Appendix A for help.

`*default-diag-path-name*` *[variable]*
Pathname with which filenames are merged by load-diagram and save-diagram.

5.3 Implementation notes

To save a diagram it is not possible to just print it because of the circular references within the overall structure of the diagram (e. g. The cross pointers in a predecessors list of a node). To get around this problem, a 'non-circular' copy of the diagram is made before printing. To do this, when copying a node's structure, references to top level nodes are replaced by a structure with the node's name. This structure is of type `node-ref`, standing for *node reference*. The diagram structure is thus effectively made non-circular. It is then written to file. On reading back, the 'non-circular' diagram is first read back and then all `node-ref`'s within the node structures are replaced by the actual nodes being pointed to.
In addition, any references to state-label structures (for example, in a distribution of a deterministic node) are replaced by the corresponding state-label structure in the `state-labels` field of the node. Thus state-labels are unique entities. When a reference is made to a state-label in a diagram, it is the actual structure which is in the state-labels field of the owning node and is *not* a copy.
When printing the diagram to file the node structures have to be printed in default lisp readable format. This is done by defining the print functions of the structures used in IDEAL such that they print in the default Lisp readable format when the special variable `*default-ideal-structure--printing*` is bound to a non-`nil` value. The macro `defidealprintfn` automatically defines the print functions such that they have this property.
To be able to print a structure in the default format a record of the names of the fields of the structure has to be available to the print function of the structure. Unfortunately, there does not seem to way to do this within Common Lisp. To get around this problem a call to the macro `store-ideal-struct-info` is made immediately after a structure is defined. This macro stores the required information on the property list of the structure type symbol and so it is available at run time to the print function of the structure. The macro `store-ideal-struct-info` also defines a recursive copy function for the structure which is used by `copy-diagram`.
The `node` structure has a slot called `unused-slot` that is reserved for IDEAL users. You can put any additional lisp objects that you want to associate with nodes in this slot. The functions `copy-diagram`, `load-diagram` and `save-diagram` can handle these objects if they satisfy some restrictions. These are listed below:

1. The object should be a cons, string, vector, number, symbol or structure.

2. If the object is a user defined structure then the user should also make a call to the macro `store-ideal-struct-info` immediately after the defstruct for the structure. The calling syntax for this macro is identical to defstruct. If the user defines a print function for this structure it should done with the macro `defidealprintfn`. IDEAL then defines a print function such that it can be turned off and the structure printed in default lisp readable format when saving diagrams. See the file `struct.lisp` for examples of the use of `defidealprintfn`.

3. The object should not have any circularities other than references to nodes and state labels. Only circularities of these two kinds can be handled when copying and saving diagrams.

6 Node distributions: Implementation details

6.1 Conditioning cases and Distribution arrays

Consider a probabilistic node A with predecessors B and C. We need to store a probability for each state of A given every combination of states of B and C, i.e., we need to store all numbers of the form $P(A = a_i/B = b_j, C = c_k)$ where i, j and k range over the states of A, B and C respectively. In the case of a deterministic node we need to store one value given every combination of states of the predecessors. A deterministic node can be a chance node, a value node or a decision node — For a deterministic chance node this value is one of the states of the node, for a value node this value is a number and for a decision node this value is a decision (after the decision has been solved for), i.e, one of the state-labels of the decision node.

In IDEAL, in the context of accessing a node A's distribution, we call a particular combination of predecessor states of A a *conditioning-case* and a particular state of A a *node-case*. We also refer to A as the *main node*, meaning we are talking about its distribution rather than B or C's. B and C are *conditioning nodes*. So, in the example above, viz., $P(A = a_i/B = b_j, C = c_k)$, the conditioning-case is $\{B = b_j,\ C = c_k\}$ and the node case is $\{A = a_i\}$.

IDEAL stores probability distributions and deterministic distributions as one dimensional arrays. Given a node-case and conditioning-case IDEAL generates a unique key into the distribution array. The probability that is required is stored at the location addressed by this key. In the case of a deterministic distribution the node-case is irrelevant and the conditioning-case is directly converted into the key. Though a conditioning-case has no ordering implicit in it, i.e., $\{B = b_j,\ C = c_k\ \}$ is the same conceptually as $\{C = c_k\ , B = b_j\ \}$, IDEAL imposes an order on conditioning cases so that it may extract keys efficiently. Every node in the influence diagram is assigned a unique ID number. These ID numbers are generated solely so that nodes can be sorted based on their ID number. *Conditioning cases* are implemented in IDEAL as lists of dotted pairs. Each dotted pair has a node as its car and a state of the node as its cdr. These dotted pairs are in arranged in increasing order of node id numbers starting from left to right.

In addition to assigning id numbers to nodes, IDEAL assigns integral id-numbers to states of the node starting from zero upwards. Having a number associated with each node and state makes it inexpensive to generate a numerical key from a conditioning case or a combination of a conditioning case and node state.

In the case of a probabilistic node, the size of a distribution array is the product of the number of states of each of the predecessors of the node and the number of states of the node itself. In the case of a deterministic node it is the product of the number of states of each of the predecessors. IDEAL has a simple key function which generates a unique key into the distribution array for each combination of conditioning case and node case (the node case is ignored in the case of deterministic nodes).

Whenever this document says that a function requires a conditioning case as input it is assumed to be an object which would satisfy `conditioning-case-p`, i.e., the restrictions mentioned above. If there are any irrelevant nodes in the conditioning case handed to a function they are ignored. For example, when querying A's probabilities, if A has predecessors B and C and the conditioning case provided to the probability access function also contains the irrelevant node D in addition to the states of B and C, the state of D is ignored by the function processing the conditioning case.

The following functions manipulate conditioning cases. When performing operations on condition-

ing cases one should always use these functions in preference to lower level LISP functions that seem to have the same functionality on the surface. Other than considerations of style, IDEAL optimizes significantly when creating and manipulating conditioning cases. Not using these functions could therefore result in some arcane structure sharing bugs.

`conditioning-case-p` *cond-case* *[function]*
Returns `t` if *cond-case* is a conditioning-case (see above for description). An error is signalled otherwise.

`make-conditioning-case` *raw-cond-case* *[function]*
raw-cond-case should be list of conses. Each cons has a node as its car and a state of the node as its cdr. This function sorts *raw-cond-case* (destructively) such that the the dot pairs are arranged in increasing order of node if number from left to right and returns the result.

`copy-conditioning-case` *case* *[function]*
Makes a copy of the conditioning case *case* which shares no structure with *case*. Returns the copy.

`combine-cond-cases &rest cond-cases` *[function]*
All the arguments are assumed to be conditioning cases. They are all merged to form one overall conditioning case. This overall conditioning case is returned. None of the input arguments is destroyed.

`make-cond-case` *node-names state-names* `&optional (`*diagram* `*diagram*`*`)`* *[function]*
This is a user interface function for creating conditioning cases. *node-names* is a list of node names. *state-names* is a list of state names, each state name being the name of a state of the corresponding node in *node-names*. *diagram* is the diagram that these nodes belong to. The function returns a conditioning case in which each node whose name is in *node-names* is associated with its corresponding state in *state-names*.

6.2 Access functions

This section uses the terminology defined in Section 6.1. A type *Access function* will be used to describe forms that can be used for both retrieving and setting a value, i.e. forms that are generalized variable references. For example:

`example-access-fn` *input-arg* *[Access function]*
This is an example that describes what the *Access function* type is all about. It means that you can use it both to return a value as in the line below:

(example-access-fn lisp-object) ⇒ *some-value*

Or, you can use it to set the location that stores that value:

(setf (example-access-fn lisp-object) some-value)

In addition to the access functions described in this section all the standard access functions that come along with the structure definitions of `node`, `discrete-dist` and `label` are available. These definitions are in the file `struct.lisp`. Please note that the user is responsible for maintaining consistency of the diagram when using these access functions to make changes to it. The system provides no guarantee of maintaining consistency.

`distribution-repn` *node* *[Function]*
This function returns the distribution array in the node structure.

`state-labels` *node* *[Access function]*
Accesses the list of label structures of the node.

`relation-type` *node* *[Access function]*
Returns the relation type of the node (`:prob` or `:det`).

`deterministic-state-of` *node cond-case* *[Access function]*
Used to retrieve and set the value corresponding to a conditioning case in a value node or, for a decision node, a decision corresponding to a conditioning case.

`prob-of` *node-case cond-case* *[Access function]*
Used to access and set probabilities in a node's distribution. *node-case* is just a conditioning case that contains one node alone, the 'main node' (See previous section). The probability accessed is Prob(*node-case* / *cond-case*).

`contents-of` *node-case cond-case* *[Access function]*
This is very similar to `prob-of`. The difference is that it can handle all types of nodes. In the case of value and decision nodes the state of node in *node-case* can be anything since the state is irrelevant. Using `nil` as the state in *node-case* when calling this function with a decision or value node aids clarity. This function exists because in some situations we carry out what is conceptually the same operation irrespective of the type of the node.

The rest of this section is of interest primarily to programmers. In some situations one has to use 'disembodied' distribution arrays, i.e. arrays which are not attached to any node. A typical situation might be during the course of some manipulation that changes the distribution of the node. The old distribution is needed during the update but the new distribution exists simultaneously and has to be stored somewhere. The standard procedure that has been used, and that you should use, is to :

- Save the old distribution (bound to some symbol with a `let`, perhaps).
- Set the `distribution-repn` of the node to the new, and maybe empty, distribution.
- Do the update. At the end of the update the old distribution becomes irrelevant and can be discarded. The new one, of course, is where it should be.

In the above procedure, the new distribution can be accessed at all times using the standard access functions (`state-of`, `prob-of` etc.). But to access the 'disembodied' old distribution array, some new routines are required. These routines are not access functions. They are ordinary functions that just return the value in the appropriate location.

`contents-of-dist-array-location` *array node-case cond-case reqd-nodes* *[function]*
Returns the contents of the location in the array *array* corresponding to the node case *node-case* and the conditioning case *cond-case*. *reqd-nodes* is a list of nodes of *cond-case* which are relevant. Usually *reqd-nodes* is a list of predecessors of the node that owns or owned the distribution array *array*. When the node in *node-case* is a deterministic node the state that is associated with is irrelevant and can be `nil`.

`node-in` *node-case* *[macro]*
Accesses the node in a lisp object of form `(( node . state))` (See `for-all-cond-cases`).

`state-in` *node-case* *[macro]*
Same as above except that the state is accessed instead of the node.

6.3 Mapping through distributions

This section is primarily of interest to programmers. Perhaps the most fundamental operation in influence diagram manipulation is the process of updating each location in a probability or deterministic distribution. The actual operation performed during the update may vary. The following macro makes this operation easy to perform. It is perhaps the most crucial bit of code in this system. This section makes use of the terminology defined in Section 6.1.

`for-all-cond-cases (`*case-variable node-list*`) &body` *body* *[macro]*

node-list is either a list of nodes or a single node. If node-list is a list of nodes: for every conditioning-case formed from the nodes in node-list, the symbol *case-variable* is bound to the conditioning case and the lisp code in *body* is executed.
When node-list is a single node: if the node is a probabilistic chance node the macro behaves exactly as if it was given a list consisting of this node alone. When the node is a deterministic node (a decision node, value node or a deterministic chance node) *case-variable* is bound to `'((,node-list . nil))` (a 'dummy' node case) and the body is executed once with this binding in effect. Giving the macro a node instead of a node-list tells the macro that the node is the 'main' node (See Sec 6.1). When the 'main' node is a deterministic node we need to map only through the conditioning-cases and not the node-cases. The macro ensures this by executing the mapping body only once when the node is deterministic.
In the case where node-list is nil, *case-variable* is bound to nil and *body* is executed once. This macro is used for side effects and does not return useful values.

An example of the use of `for-all-cond-cases` might be to print the entire probability distribution of the probabilistic chance node A:[3]

```
(for-all-cond-cases (cond-case (node-predecessors #n(a)))
   (for-all-cond-cases (node-case (list #n(a)))
       (format t "~%Prob{ ~A / ~A } = ~A" node-case cond-case
                                          (prob-of node-case cond-case))))
```

Another example could be the setting of the value distribution of a value node V by querying the user for the values:

```
(for-all-cond-cases (cond-case (node-predecessors #n(v)))
  (setf (state-of #n(v) cond-case)
        (query 'NUMBER "What is the value of case ~A ? =>" cond-case)))
```

The following rather contrived example shows how one might find the total of the values of all conditioning cases of a value node V (never mind why one might want to do that).

```
(let (( total 0))
  (for-all-cond-cases (cond-case (node-predecessors #n(v)))
        (incf total (state-of #n(v) cond-case)))
   (values total))
```

The general method of use of the macro is to use it in an 'outer loop' to generate the conditioning cases and in an 'inner loop' to generate the node-cases. For example, to set the distribution of any kind of node interactively we could write:

```
(for-all-cond-cases (cond-cases (node-predecessors <node>))
  (for-all-cond-cases (node-case <node> )
     (setf (contents-of node-case cond-case)
           (query-for-appropriate-type node
               "Whats the entry for [~A/~A}/ =>" node-case cond-case))))
```

It is interesting to note that if we had specified `(list <node>)` instead of `<node>` as the argument for the inner loop, then in the case of a deterministic chance node each useful query and set operation would be repeated n times where n is the number of states of the node when in reality the operation needs to be done just once.

6.4 Implementation note

The macro `for-all-cond-cases` had a slightly different syntax in earlier versions. This version still supports the old syntax. The macro creates a template for a conditioning case. As the macro maps through the conditioning cases, it creates a new conditioning case by *destructively* modifying the template. This is in interests of efficiency. Therefore, no code should destroy the conditioning case (i.e the object to which *case-variable* is bound) since this will interfere with the macro's operation. On the flip side, no code should depend on the conditioning case not being destroyed. All problems can be avoided by using only the conditioning case handling primitives described above when manipulating conditioning cases. These functions take the appropriate precautions.

[3]In the following examples *#n(aaa)* refers to the node (structure) of name *aaa*. See Sec 4.1 for details.

7 Utilities

7.1 Predecessors and successors

`ancestors` *node* *[function]*
Returns a list of those nodes from each of which there is a directed path to *node*.

`descendants` *node* *[function]*
Returns a list of those nodes to each of which there is a directed path from *node*.

7.2 Useful predicates

`chance-node-p` *node* *[function]*

`decision-node-p` *node* *[function]*

`value-node-p` *node* *[function]*
The functions listed above return non `nil` iff the node is of the appropriate type.

`discrete-dist-node-p` *node* *[function]*
Returns `t` if *node* is a node that has a distribution of type `discrete-dist`. `nil` otherwise.

`single-directed-path-p` *pred succ* *[function]*
Returns `t` if there is only one directed path from the node *pred* to the node *succ*. Returns `nil` otherwise.

`barren-node-p` *node* *[function]*
Returns `t` if *node* is a decision or chance node with no successors. `nil` otherwise.

`noisy-or-node-p` *node* *[Function]*
Returns `t` if *node* is a noisy-or node, `nil` otherwise.

7.3 Consistency Checking

The following functions are used to check whether a diagram is consistent. These functions return `t` if the diagram passes the test implemented by the function. If an inconsistency is found, the functions print a warning describing the inconsistency and return `nil`.
A note about the implementation of 'no-forgetting' arcs is in order here. The system does not require that the user add all no-forgetting arcs nor does it add all the arcs itself. Instead, it generates informational predecessors on the fly by a recursive algorithm when required. What the system *does* require, however, is that there be an arc from every decision node to the chronologically succeeding decision. To check whether the decisions are chronologically ordered the system checks to see whether it can construct a non-circular directed path that traverses all the decision nodes.

`consistent-p` *diagram* `&optional (`*tolerance* `1e-6)` *[function]*
This function returns `t` if:

- The top level of the diagram is consistent.
- The implementation details of each node are consistent.
- The topology of the diagram is consistent.
- The distributions of each node in the diagram are consistent.

When checking consistency of the probabilities, the function checks whether the probabilities sum up to 1 with a tolerance given by the *tolerance* argument. This means that it is acceptable for the sum of the probabilities to be anywhere between (1 - *tolerance*) and 1.

`acyclic-p` *diagram* *[function]*
Returns non `nil` iff the diagram has no directed cycles.

`strictly-positive-distributions-p` *diagram* *[function]*
Checks whether the distributions of all chance nodes in the diagram are strictly positive, meaning that no probabilities encoded in the diagram are 0.

`belief-net-p` *bel-net* *[function]*
Returns `t` if *bel-net* is a belief net. This requires that all nodes in *bel-net* be probabilistic chance nodes.

7.4 Interface functions

`find-node` *name* `&optional (`*diagram* `*diagram*) (`*error* `t)` *[function]*
Returns the node of name *name* in the diagram *diagram*. The `#n` macro makes use of this function. The function signals an error if the node is not found in the diagram if the *error* argument is not `nil`. If the *error* argument is `nil` and the node is not found the function returns `nil`.

`find-label` *label-name node-name* `&optional (`*diagram* `*diagram*)` *[function]*
Returns the label (structure) of name *label-name* found in the state-labels field of a node of name *node-name* in the diagram *diagram*.

7.5 Creating random belief networks

The following functions are related to creating a random belief network. These functions are sometimes useful for experimentation.

`create-random-diagram` *n* `&key` (*in-degree-limit* `3`)
(*max-number-of-states* `4`) *[function]*
Creates a random belief network with *n* nodes. The maximum number of predecessors that a node can have is *in-degree-limit*. The maximum number of states that a node can have is *max-number-of-states*. The resulting belief network has a strictly positive probability distribution. It is important to have a reasonably low value of *in-degree-limit* or the function can take forever to run. When `*ideal-debug*` is non-`nil` the function prints a trace of what it is doing.

`number-of-arcs` *diagram* *[function]*
Returns the total number of arcs in the input diagram.

`create-random-evidence` *diagram* `&optional`
max-number-of-evidence-nodes (*node-predicate* `#'node-p`) *[function]*
Of the nodes in the diagram that satisfy the predicate *node-predicate*, the function chooses at most *max-number-of-evidence-nodes* randomly and instantiates them with specific evidence.

`randomize-probabilities` *existing-belief-net* *[function]*
This function sets the probability distribution of each node in the diagram *existing-belief-net* randomly. The distributions are strictly positive.

7.6 Miscellaneous functions

`order` *node-list* `&key` (*check-complete-ordering* `nil`) (*last-node* `nil`)
(*mode* `:ERROR`) *[function]*
Orders *node-list* such that weak predecessors of a node always precede a node in the returned node list. When *mode* is `:WARN` it returns a second value which is `t` if the ordering was successful and `nil` if not. In addition, a warning is printed if the ordering was not successful. When *mode* is `:ERROR` an error is signalled describing why the ordering was not successful. When the second value returned by the function is `nil` the first value has no meaning.
If the *last-node* argument is specified: 1) If *last-node* is not a member of the input node-list it is ignored. 2) Else *last-node* is ordered last in the returned list, if it is possible. If this is not possible an error or warning is signalled depending on the mode. If a value node is present in *node-list* it is ordered last unless a *last-node* argument supersedes it.
If *check-complete-ordering* is non-`nil` and the nodes in *node-list* do not have a complete ordering with respect to each other an error or warning is signalled depending on the mode.

`generate-diagram` *node-or-node-list* *[function]*
The argument *node-or-node-list* can be a node or a list of nodes. This function returns the list of all nodes that are connected to the node or nodes in *node-or-node-list*.

`convert-and-smooth-diagram` *diagram* `&optional` (*factor* `1e-6`) *[function]*
Takes a diagram as input. Converts all deterministic chance nodes to probabilistic nodes. It also changes the distributions of all the nodes such that the diagram that is returned is consistent and is guaranteed to have only strictly positive distributions, i.e., no probability encoded in the diagram

is zero. The argument *factor* specifies the minimum value of the probabilities in these 'smoothed' distributions. If the default is used, it is guaranteed that all probabilities encoded in each node of the diagram are greater than or equal to `1e-6`.

7.7 Debugging

`*ideal-debug*` *[Variable]*
When `*ideal-debug*` is non-`nil` the various influence diagram and belief net manipulation functions print traces of their progress as they run. When the variable is set to `nil` these traces are not printed.

8 Influence diagram transformations

These are a set of high level routines that perform some basic transformations on influence diagrams. These routines can be used to code influence diagram algorithms. Some of them can also be used to edit diagrams. Each of these routines leave the diagram in an entirely consistent state: i.e. the predecessor, successor links of each node and the distribution of each node are in a consistent state. All these functions do extensive input checking. They usually signal error if inputs are not of the appropriate type. Each of these functions prints a trace of its progress when `*ideal-debug*` is set to `t` (See Sec 7.7).
The theory underlying these functions is found in [3, 9, 11].

`remove-barren-node` *node* `&optional (`*diagram* `*diagram*)` *[function]*
Removes the node *node* from the diagram *diagram* if it is barren. Error otherwise. The modified diagram with *node* removed is returned.

`remove-all-barren-nodes` *diagram* *[function]*
Removes barren nodes from the diagram till there are none left. Then returns the modified diagram.

`absorb-chance-node` *node* `&optional (`*diagram* `*diagram*)` *[function]*
Removes the chance node *node* by absorbing it into the lone succeeding value node or chance node and then returns the modified diagram. If *node* does not satisfy the preconditions for the removal of a chance node [11] an error is signaled.

`remove-decision-node` *node* `&optional (`*diagram* `*diagram*)` *[function]*
Removes the decision node that precedes the value node by maximizing the expected utility. The diagram, with *node* removed, is returned. If *node* does not satisfy the preconditions for its removal [11] an error is signalled.

`reverse-arc` *pred succ* `&optional (`*diagram* `*diagram*)` *[function]*
Reverses the arc from chance node *pred* to chance node *succ*. If *pred* and *succ* do not satisfy the preconditions for arc reversal [11] an error is signalled. The modified diagram is returned.

`reduce-deterministic-node` *det-node diagram* `&optional` *debug* *[function]*
Propagates the deterministic node *det-node* into each of its successors and then removes the resulting barren node from *diagram*. The resulting diagram is returned.

`reduce-probabilistic-node` *node diagram* `&key (`*last-node* `nil)` *[function]*
Progressively reverses arcs out of the probabilistic chance node *node* till *node* can be absorbed. Then *node* is absorbed and the updated diagram (i.e., without node) is returned. *last-node*, if specified, is the node that *node* is absorbed into.

`reduce-chance-node` *node diagram* `&key (`*last-node* `nil)` *[function]*
This function has been defined for the sake of completeness. It takes any chance node as an input and reduces it out of the diagram. The *last-node* argument is relevant only if the input is probabilistic (see above).

`reduce-all-deterministic-nodes &optional (`*diagram* `*diagram*)` *[function]*
Reduces all deterministic nodes out of the diagram and then returns diagram.

`convert-det-node-to-prob` *det-node* *[function]*
Converts the deterministic node *det-node* to a probabilistic node and returns it.

9 Algorithms

This section documents the various types of inference and evaluation algorithms implemented in IDEAL. The documentation does not go into the details of the algorithms. References to the appropriate literature are provided in each subsection for readers interested in the theory behind these algorithms.
The functions that implement the inference and evaluation algorithms do extensive input checking. An error is signaled if any inconsistencies are found. Each of these functions prints a trace of its progress when `*ideal-debug*` is set to `t` (See Sec 7.7).

9.1 Influence Diagram algorithms

This section describes influence diagram inference and evaluation algorithms. The evaluation algorithms return a list consisting of the decision nodes in the diagram and the value node. The optimal decision policies for the decision nodes are stored in the distribution of the decision node and the expected value of the decision problem is stored in the distribution of the value node. A policy is a mapping from the conditioning cases of the *relevant* informational predecessors of a decision node to the states of the node, i.e. to one of the possible alternative actions. The policy is similar to a value distribution and can be accessed with the same access functions. Each location in the policy array stores a dot pair: the car is the optimal decision and the cdr is the expected value corresponding to the decision. The relevant informational predecessors of a decision node are a subset of the actual informational predecessors of the node.
All the algorithms in this section operate by transforming the diagram either by removing nodes or reversing arcs. Inference algorithms take the goal node (the node whose updated distribution we

want) and the conditioning node set (the nodes upon which we want the goal node's distribution conditioned) as inputs and transform the diagram to yield the desired result.

`shac-eval &optional (`*diagram* `*diagram*)` *[function]*
The diagram *diagram* is solved for the optimal decision sequence. The function returns a list of decision nodes, chronologically earlier decisions being earlier in the list. Each decision node has an optimal policy in place. This function uses the algorithm outlined in [11].

`ides-eval &optional (`*diagram* `*diagram*)` *[function]*
Same as above except that the algorithm in [9] is used.

`shac-infer` *goal-node cond-node-set* `&optional (`*diagram* `*diagram*)` *[function]*
Transforms the diagram to yield P(goal-node/cond-node-set). A diagram consisting of the nodes in the list *cond-node-set* and the node *goal-node* is returned. This function uses the algorithm described in [11].

`ides-infer` *goal-node cond-node-set* `&optional (`*diagram* `*diagram*)` *[function]*
Same as above except that the algorithm in [9] is used.

9.2 Belief Net algorithms

Belief net algorithms operate on influence diagrams which consist only of chance nodes. Given some evidence **e** pertaining to some observed nodes in the belief net, these algorithms calculate the updated probability distribution for each node, A, in the net conditioned on this evidence — $P(A/\mathbf{e})$. This distribution is called the *belief* distribution. Belief net algorithms are of three basic kinds: Clustering algorithms, algorithms based on conditioning and Monte Carlo simulation type algorithms. In addition to the types of algorithms mentioned above, there are efficient algorithms that handle some special types of belief nets.
In these section dealing with belief net algorithms the term 'influence diagram' refers to a belief net unless mentioned otherwise. In addition, in the subsection dealing with singly connected belief nets the terms 'influence diagram','belief net' and 'polytree' connote singly connected belief nets (polytrees) unless mentioned otherwise.
When describing data structures, input formats etc. the term 'node' refers to the `node` structure representing the node and the term 'state' refers to the actual `state-label` structure in the `state-labels` field of the `node` structure.

9.3 Preliminaries

9.3.1 Beliefs

The data structure for storing beliefs in the belief net algorithms is as follows:

bel This is a field of the node structure that stores an array which in turn stores the beliefs. This array's size is equal to the number of states of the node.

The function that accesses the beliefs is as follows:.

`belief-of` *node-case* *[Access function]*
node-case is a conditioning case consisting of only one node. This function accesses the belief associated with the state contained in *node-case* of the node in *node-case*.
All belief net algorithms store the updated beliefs in the belief array. The beliefs can be viewed using the following utility functions:

`display-beliefs` *node* *[function]*
Displays the beliefs associated with the various states of the node *node* in a fairly readable format. Returns no values.

`diag-display-beliefs &optional (`*diagram* `*diagram*)` *[function]*
Displays the beliefs of each node in the diagram. Returns no values.

9.3.2 Evidence: types and data structures

Evidence in belief net algorithms can be of various types. We refer to the evidence pertaining to a particular node as an *evidence item*. Evidence items can be of different kinds. When an evidence item is declared, IDEAL stores a representation of the evidence item on the `state` field of the node that the item is relevant to. This representation is later processed by the inference algorithm. There are three kinds of evidence items:

- Specific Evidence: When the state of a particular node is known it is called specific evidence. A specific evidence item is represented by the `state` structure of the state that the node is in.

- Virtual evidence: Let X be a node with states $x_1, x_2, \ldots x_n$. Let $Z = z_{obs}$ be some observation that is affected by X such that we are able to estimate a likelihood ratio of $P(Z = z_{obs}/X = x_1) : P(Z = z_{obs}/X = x_2) : \ldots P(Z = z_{obs}/X = x_n)$. This ratio vector is known as virtual evidence. For example, given that one observes the footprint of an animal, one may be able to estimate a likelihood ratio (odds) of the chances of some (mutually exclusive) set of animals having caused it. A virtual evidence item is represented as an assoc list. The assoc list consists of dot pairs, each dot pair corresponds to one state of the node. The car of the dot pair is the `state` structure and the cdr of the dot pair is the odds of the observation being caused by the node being in that state.

- Retracted Evidence: Sometimes there is a need to retract an item of evidence that has been declared earlier (non-monotonic reasoning). The retraction of evidence pertaining to a node is treated as a new evidence item. A retraction evidence item is represented as the lisp object `nil`. Note that the exact evidence that was declared earlier with respect to the node is not required to retract it. All that is required is some representation of the fact that one wants to retract the previous evidence pertaining to the node. If there was no previous evidence item pertaining to the node then retracting evidence pertaining to the node has no effect, though it can be done with no ill effects.

The following functions can be used to manipulate evidence in a belief net:

`create-evidence` *belief-net* *[function]*
Queries the user for evidence and sets the evidence on the nodes affected by the evidence. Returns *belief-net*.

`reset-evidence` *node-list* *[function]*
The evidence on each node in node-list is 'reset', i.e if the node evidence has already been propagated, it is marked as "not propagated" so that it will be re-propagated on the next call of an inference algorithm.

`remove-evidence` *node-list* *[function]*
If evidence pertaining to any of the nodes in node-list has been declared, the evidence is deleted.

9.4 Singly connected belief nets

A *singly connected* graph is one in which there is at most one path from one vertex to another. In the case of directed graphs, the same definition applies, except that the directionality of the edges is ignored when applying the definition. A singly connected belief net is also known as a *polytree*. An efficient linear time algorithm [6] has been implemented for probabilistic inference in polytrees.

9.4.1 Data Structures

In addition to all the standard fields (See Sec 2) and the `bel` field (See Sec 9.3.1) each node structure has some extra fields that are used by the polytree algorithm. These fields are as follows:

pi-msg This field stores the pi messages that the node sends to its children and the overall pi msg of the node itself. The information is stored as an assoc list. The car of each member of this list is a node and the cdr is the message array corresponding to this node. The size of this one dimensional array is equal to the number of states of the node owning the pi-msg field.

lambda-msg This field stores the lambda messages that the node sends to its parents and the overall lambda message for the node itself. The actual data structure is similar to the pi message field except that the length of each array is equal to the number of states of the node associated with the array.

old-state This field records what the evidence related to the node when belief propagation is performed. When some new evidence pertaining to the node comes in, IDEAL checks to see whether this infomation is different from the information it already has. If it is, it propagates it. If not it does nothing.

9.4.2 Access functions

`lambda-of` *node-case* *[Access function]*

`pi-of` *node-case* *[Access function]*
The above two functions are very similar to `belief-of` except that they access the overall lambda and overall pi of a particular state of a node.

`lambda-msg-of` *node-case succ* *[Access function]*
node-case is a conditioning case consisting of only one node. *succ* is a node which should be a successor of the node in *node-case*. This function accesses the lambda message that is sent by *succ* to the node in *node-case* for the state in *node-case*.

`pi-msg-of` *node pred-case* *[Access function]*
pred-case is a conditioning case of only one node. *node* is a node. *node* should be a successor to the node in *pred-case*. This function accesses the pi message sent by the node in *pred-case* to *node* for the state in *pred-case*.

9.4.3 Initializing poly-trees

`set-up-for-poly-tree-infer &optional (`*poly-tree* `*diagram*)` *[function]*
Takes a poly-tree as an input and sets up and initializes the data structures for the beliefs and messages that are needed by the poly-tree algorithm. This function must be applied to a freshly created diagram before it can be used for inference.

`initialize-poly-tree` *diagram* *[function]*
This function initializes all the beliefs and messages in the diagram. When this function exits, the beliefs and messages are consistent with the state of there being no external evidence. After a diagram has been used for inference, it can be 'reset' to its initial state using this function. `set-up-for-poly-tree-infer` calls this function (internally) after setting up the raw data structures to initialize the diagram.

9.4.4 Inference in singly connected belief nets

The following function performs inference using polytrees.

`poly-tree-infer &optional (`*poly-tree* `*diagram*)` *[function]*
Diagram is a diagram that has been set up and initialized for polytree inference. It is assumed that the evidence has already been declared in the diagram (See Sec 9.3.2). The function returns the diagram with appropriately updated beliefs. This diagram can be used again if additional evidence arrives or if some previously declared evidence has to be retracted. To start afresh, the diagram has to be re-initialized. (Alternatively, all evidence can be retracted, though this is a more expensive way of achieving the same result.)

9.4.5 Utilities

`singly-connected-belief-net-p` *diagram* `&key (`*error-mode* `t)` *[function]*
Returns `t` if the diagram *diagram* is a singly connected belief net. If the diagram is not singly connected and *error-mode* is `t` an error is signaled. If `error-mode` is `nil` the function prints a warning and returns `nil`.

9.4.6 Implementation notes

The evidence is modeled as dummy nodes that send lambda messages to the affected node. Retraction of evidence is modeled by deleting any dummy node attached to the node of interest. These dummy nodes are created on the fly and are *still* linked to the nodes of the belief net when it is returned by `poly-tree-infer`. Therefore, evidence once declared, remains in effect (and is therefore noted by the next run of poly-tree-infer) unless it is retracted or the diagram is re-initialized. Evidence pertaining to a node can be superseded by new evidence pertaining to a node, though. This is handled by just changing the lambda message transmitted by the existing dummy node. The `state` field of a node is used to store the neighbor node that activated the update of the node. The `type` field of dummy nodes stores a keyword that allows dummy nodes to be identified.
The activation of the nodes proceeds as a depth first search of the polytree. This is not necessary though, and any activation order can be implemented without affecting the results.

9.5 Clustering

The presence of loops in belief nets cannot be handled by polytree type algorithms. Clustering algorithms can handle all belief nets [4, 6]. On the other hand, since the general problem of probabilistic inference is NP hard, such algorithms are practical only for sparsely connected belief nets.

9.5.1 Overview

The Clustering algorithm operates as follows:

1. The belief net is converted into a Markov network by interconnecting the parents of each node (with non-directional links) and then making the original directed edges in the belief net non-directional.

2. The graph is *triangulated* using *maximum cardinality search*. After triangulation, the Markov network can be decomposed into a set of maximal cliques that have the *running intersection property*. This set of cliques is assembled into a tree in accordance with this property.

3. The tree of cliques is now initialized with potentials or beliefs using the probability distributions encoded in the underlying belief network.

4. A simple update scheme is then run on the tree of cliques and finally, the belief distributions of the nodes in the underlying belief network are derived from the potentials or clique beliefs.

The first three steps can be considered as a kind of 'compilation' of the belief network and can be done off-line. The last step is the only one that needs to be carried out when fresh evidence arrives and updated beliefs are required.
There are two versions of the clustering algorithm implemented in IDEAL. They differ in the way in which the last two steps mentioned above are carried out. The first version looks at the tree of cliques as another belief net (a meta belief net, if you will). The probability distributions and the beliefs on the cliques in this tree are calculated from the underlying belief net. Now a simple version of the poly-tree algorithm that applies to trees is applied to this tree of cliques. Once the propagation is carried out, the beliefs of any node in the underlying belief net are obtained by

marginalizing the beliefs on any clique in which it is a member. This first method is an interpretation of the clustering method from Pearl [6].
The second version initializes the cliques with clique *potentials*. These potentials are derived from the belief network. A simple update scheme that manipulates these potentials is now applied to the tree of cliques. When the update scheme is done the belief distribution of any node in the underlying belief network is found by marginalizing and normalizing the potentials of any clique in which the node is a member. This version of the clustering algorithm is based on the interpretation of Jensen et.al. [5].
Obviously, the two versions are closely related. Nevertheless, it turns out that the second version is very much faster than the first version. In fact, it is the fastest running inference scheme in IDEAL and it is the one that should be used when speed is the most important criterion.

9.5.2 Clustering: Pearl Version

There are two steps in applying this inference method. First one 'compiles' the belief network by creating a *clique diagram*, which is a tree of cliques. Actual inference is carried out by an update function when evidence arrives. The update function takes the *clique diagram* and the belief network as inputs.

`create-clique-diagram` *belief-net* *[function]*
Creates a belief net as input and creates a clique diagram for the belief net. The data structures for the clique diagram are created and initialized. The clique diagram is returned. The function also creates and initializes the belief data structures for each node of the belief net.

`initialize-for-clustering` *belief-net clique-diagram* *[function]*
clique-diagram is a clique diagram that has previously been created for the belief net *belief-net*. This function reinitializes the clique diagram and the belief net. It can be used to clear all the evidence after using the clique diagram for inference. The function returns *clique-diagram*. It is assumed by the function that the user is providing a clique diagram that actually corresponds to the belief net *belief-net*. There is no input check for this.

`clustering-infer` *clique-diagram belief-net* *[function]*
clique-diagram is a clique diagram that has been created for the belief net *belief-net* using the function `create-clique-diagram` (See above). It is assumed that evidence pertaining to the belief net has already been declared (See above). This function updates the beliefs of the nodes of *belief-net* to reflect all the evidence declared.

9.5.3 Clustering: Jensen Version

This method is the fastest inference scheme available in IDEAL. This scheme should be used when inference speed is important. This scheme, as implemented, only handles specific evidence.
There are two steps in applying this inference method. First one 'compiles' the belief network by creating a *join tree*, which is a tree of cliques. Actual inference is carried out by an update function when evidence arrives. The update function takes the *join tree* and the belief network as inputs.

`create-jensen-join-tree` *belief-net* *[function]*
Creates a belief net as input and creates a join tree for the belief net. The data structures for the join tree are created and initialized. The join tree is returned. The function also creates and initializes the belief data structures for each node of the belief net.

`jensen-infer` *join-tree belief-net* *[function]*
join-tree is a join tree that has been created for the belief net *belief-net* using `create-jensen-join-tree` (See above). This function updates the beliefs of the nodes of *belief-net* to reflect all the evidence declared.

9.6 Conditioning

Inference in belief nets using conditioning works in the following way:

- A cycle cutset of belief net is generated. The cycle cutset is a set of nodes whose removal breaks all undirected cycles in the belief net.
- For every possible state combination of the nodes in the cycle cutset:
  - Each node in the cycle cutset is "clamped" to its state in the cycle cutset. This effectively converts the belief net to a poly-tree.
  - Evidence is propagated using the poly-tree algorithm. The beliefs of the nodes are now effectively conditioned on both the clamping and the evidence.
- The beliefs of the nodes are "averaged" over all the clampings (in accordance with Bayes' rule) to get the actual beliefs of the nodes conditioned on the evidence.

Details about this algorithm can be found in [6]. The cycle cutset is found using a greedy algorithm from [15]. Unlike the poly-tree and clustering algorithms, the conditioning algorithm, as implemented, does not handle incremental evidence propagation. This means that each time the algorithm is called, it initializes the appropriate data structures and propagates all the evidence. IDEAL also has a more efficient version of conditioning due to Peot and Shachter [7].

`conditioning-infer &optional (`*bel-net* `*diagram*)` *[function]*
This function propagates evidence using the conditioning algorithm. The function returns *bel-net*.

`lw-infer &optional (`*diagram* `*diagram*)` *[function]*
This function propagates evidence likelihood weighting, a version of the conditioning algorithm described in [7]. The function returns *diagram*.

9.7 Simulation

Two types of simulation algorithms are implemented in IDEAL[4]. The first simulation algorithm is based on Pearl's stochastic simulation algorithm [6]. During this type of simulation the evidence

[4]This section and most of the code described by it is authored by Mark Peot of the Rockwell Palo Alto Laboratory.

nodes are "clamped" to their known values. The algorithm repeatedly instantiates the state of randomly chosen parameter (unobserved) nodes and sets them to a new state based on the probability of the node given the instantiated Markov blanket. As long as there are none of the nodes in the belief net have zero (or one) probabilities, the probability of a node being in a certain state is equal to the long term frequency of that state being chosen during simulation.

`set-up-for-simulation-infer` *belief-net* *[function]*
This function sets up the data structures required for the simulation algorithm. The function returns the belief-net *belief-net*.

`simulation-infer` *belief-net* `&optional` (*iterations* `1000`) *[function]*
This function performs stochastic simulation on the the belief net *belief-net*. The *iterations* argument specifies the number of simulations that must be performed. The function returns *belief-net*.

IDEAL also offers a set of simulation algorithms based on forward sampling. The algorithms currently implemented are Logic Sampling [2] (with and without importance sampling) and Likelihood Weighting [12, 13] (also with and without importance sampling). In both algorithms, nodes are sampled in graph order (hence the name 'forward sampling').
In Logic Sampling, all of the nodes are sampled including evidence nodes. If the sampled evidence nodes are in the same state as the observed evidence, the frequency count of the sampled state of the parameter nodes is incremented by one. If the sampled evidence does not match the observed evidence, the trial is discarded and simulation starts again. After running the simulation a user-specified number of times, the frequency count vector for each node is normalized to yield a belief vector.
Likelihood Weighting accounts for the presence of evidence in a slightly different fashion. Evidence nodes are clamped to their observed values. The score for the parameter nodes in a trial is set to the product of all of the evidence likelihoods in the network. Likelihood weighting produces identical results to Logic Sampling when all of the evidence nodes are deterministic. Both algorithms can be used with importance sampling [12, 13, 2, 10]. Importance sampling allows the user to 'point' the algorithm toward the higher probability portions of the space of possible events resulting in faster convergence times in belief nets with observed evidence. Another way of improving convergence times is to use Markov blanket scoring [12, 13, 6] which reduces runtime variance by simultaneously scoring all of the states of each node.

`forward-simulation-infer` *node-listiterations* `&key` (*diagram nil*) (*algorithm* `:normal`) *[function]*
This function runs the simulation routines on a belief network. If *diagram* is supplied, it is used rather than building a new one. Algorithm can either be `:NORMAL` for likelihood weighting, `:MBS` for likelihood weighting with Markov blanket scoring, or `:LOGIC` for logic sampling. The function returns the belief-net *belief-net* and the data structure which manages the state of the simulation *diagram*.

`forward-simulation-initialize` *node-list* `&key` (*algorithm nil*) *[function]*
This function builds a simulation diagram for a node-list. The algorithm argument tells the function

whether it should build the markov blanket scoring arrays or not. The values *algorithm* can assume are listed in the definition of *forward-simulation-infer*.

`importance-prob-of` *node-case* `&optional` *cond-case* *[Access function]*
This is the accessor for the probabilities in the importance sampling arrays. The importance probability is used in importance sampling. The initial importance sampling probabilities are set to the conditional probability distributions of the nodes they occur in.

`sim-continue` *diagram iterations* *[function]*
One of the advantages of simulation is that it is an anytime algorithm. A simulation can be ran for a while, halted, and then be resumed if desired. Sim-continue is the mechanism for restarting a simulation that has halted.

`sim-reset` *diagram* *[function]*
Sim-reset resets the running belief of the nodes in the simulation diagram to zero and updates the diagram to account for new evidence. The running belief arrays in a node is where the simulation algorithm accumulates simulation samples. When sim-continue halts, the running belief is copied to the normal node belief arrays and is normalized. The running belief is not reset at the conclusion of a simulation run to allow the user to continue the simulation.

10 Noisy Or Nodes

The Noisy Or model describes a special class of discrete probability distributions. This class of distributions is frequently used to model probabilistic relationships (i.e., a node's distribution) in belief networks since it is a good description of a frequently encountered class of physical phenomena. In some circumstances, using the Noisy Or model can lead to more efficient inference algorithms since one is dealing with a restricted class of distributions. A good description of the the Noisy Or model in the case of binary nodes is found in [6].
IDEAL implements a generalized version of the Noisy Or model. The generalization extends the Noisy Or model from binary nodes with binary predecessors to generalized $n$ary nodes with $n$ary predecessors. A detailed description of the generalized model is in [14].
A brief synopsis of the generalized model is as follows: A noisy-or distribution for a Node N has an underlying deterministic function relation the predecessors of N to N – i.e, for each combination of states of the predecessors of N the underlying deterministic function determines a state of N. In addition to the function, N has a vector of *inhibitor* probabilities for each predecessor, indexed by the states of the predecessor. These probabilities describe the chance that a predecessor might fail in some particular state. When a predecessor fails in a particular state its input to the deterministic function determining N is always the failure state, regardless of other evidence regarding the state of the predecessor.
The failure probabilities of each predecessor are independent of each other, and the failure probabilities for all the states of a predecessor sum to less than or equal to 1 – in other words, there may be 0 or more probability that the predecessor doesn't fail at all.

This model, described by the possible failures of the predecessors (i.e, the *inhibitor* probabilities) and underlying deterministic function determines a probability distribution relating N to its predecessors.
A noisy-or node in IDEAL is described by inputting the underlying deterministic function and the inhibitor probabilities. IDEAL then compiles this description into a probability distribution relating the noisy-or node to its predecessors. After this compilation, the node is functionally identical to any other probabilistic chance node. In other words, the noisy-or feature is purely a modeling convenience in IDEAL for a particular class of probabilistic chance nodes – there are no specific algorithms tailored to using noisy-or nodes for specially efficient inference.
IDEAL provides three subtypes of noisy-or nodes. These subtypes are:

:BINARY This is the type of noisy-or node described in [6]. The node has to be binary and its underlying deterministic relationship to its predecessors is the Boolean Or function. The state numbered 0 of a node is taken as the *false* state and the state numbered 1 is taken as the *true* state. When predecessors are not binary themselves, all states of a predecessor other than the state with number 0 are equivalent to the *true* state [5].

The user needs to provide an *inhibitor* probability for the *false* state of each predecessor. The *inhibitor* probability for all other states of each predecessor is set to 0.

:NARY Both the node and its predecessors can have any number of states. The underlying deterministic function is an equal-weighted additive function relating the predecessors to the node (see [14] for details). The user needs to provide an *inhibitor* probability for each state of each predecessor. The `:BINARY` model is a special case of the `:NARY` model.

:GENERIC Both the node and its predecessors can have any number of states. The underlying deterministic function has to be specified by the user. The user also needs to provide an *inhibitor* probability for each state of each predecessor. The `:NARY` model is a special case of the `:GENERIC` model. The `:GENERIC` model implements the most general form of the Noisy-Or model described in [14].

10.0.1 Creating and Editing Noisy Or nodes

The standard belief net creation and editing functions also handle Noisy Or nodes (See Sections 3 and 4).

10.0.2 Programmer's Interface

Noisy-or nodes are implemented as a special class of probabilistic chance nodes, i.e, a noisy-or node is a node of type `:CHANCE` with relation type `:PROB` which also satisfies `noisy-or-node-p`. noisy-or nodes always contain a compiled probability table created from the noisy-or parameters. So any operation performed on a probabilistic chance node (for example, retrieving a table probability) will also work on a noisy-or node.
In all belief net algorithms, the algorithm does not recognize noisy-or nodes as different from chance nodes. In reduction algorithms when any transformation that affects the noisy-or node is made, it is converted to a chance node (for example, reversing an arc going out of a noisy-or node).

[5] IDEAL numbers the states of a node.

This design of the noisy-or implementation has tried to ensure that noisy-or nodes are truly transparent to the user. They are meant to be useful for modeling but transparent at the inference and evaluation level.

`noisy-or-node-p` *node* *[Function]*
Returns `t` if *node* is a noisy-or node, `nil` otherwise.

`noisy-or-subtype` *node* *[Access Function]*
Used to retrieve (and set) the subtype of the noisy-or node *node*. The subtype can be one of `:BINARY`, `:NARY` and `:GENERIC`.

`convert-noisy-or-node-to-chance-node` *node* *[Function]*
Converts the noisy-or node *node* to a probabilistic chance node with the distribution calculated from the noisy-or node parameters. Returns *node*.

`inhibitor-prob-of` *node pred-case* *[Access Function]*
Used to retrieve and set the inhibitor probability of node *node* associated with the state and predecessor contained in *pred-case*. *pred-case* should be a conditioning case containing exactly one node-state pair.

`noisy-or-det-fn-of` *node pred-case* *[Access Function]*
The deterministic function from the predecessors to the node *node* is stored as a table. This access function retrieves and sets the state (i.e the state label structure) associated with a conditioning case made from all the predecessors of *node*.

`compile-noisy-or-distribution` *node* *[Access function]*
Compiles a probability distribution in the form of the a table *node* from the noisy-or parameters. Returns no values. In more specific terms, after the inhibitor probabilities and deterministic function of *node* have been set using `inhibitor-prob-of` and `noisy-or-det-fn-of`, this function is called so that the probabilities described by these parameters is available using the standard `prob-of` interface to chance node distributions.

`noisy-or-false-case-p` *node-case* *[Function]*
Returns `t` if the state in the conditioning case *node-case* is the *false* state, i.e., if the state is numbered 0. *node-case* should contain only one node-state pair. This is used to recognize the distinguished *false* state when dealing with `:BINARY` noisy-or nodes.

`convert-noisy-or-node-to-subtype` *node new-subtype* *[Function]*
Used to convert a noisy-or node to a different subtype. *new-subtype* can be one of `:BINARY`, `:NARY` and `:GENERIC`. The function uses default values for the new inhibitor probabilities and deterministic function. These can be subsequently edited. Returns *node* as a value.

11 Diagrams

If you create diagrams for some interesting problems please save them for use by others and for debugging of the system as it is improved. Presently, the oil wild catter problem in Raiffa [8] is available in `oil.diag` and a singly connected belief net is in `beliefnet.diag`. These examples are distributed with the code.

References

[1] Breese, J. S., Horvitz E. J. and Henrion, M. (1988) Decision Theory in Expert Systems and Artificial Intelligence. Research Report **3**, Rockwell International Science Center, Palo Alto Laboratory.

[2] Henrion, M. (1988) Propagating Uncertainty in bayesian networks by probabilistic logic sampling. *Uncertainty in Artificial Intelligence* **2**, J. F. Lemmer and L. N. Kanal (Eds.), North-Holland, Amsterdam.

[3] Howard, R. A. and Matheson, J. E. (1981) Influence diagrams. *The Principles and Applications of Decision Analysis* **2**. Strategic Decisions Group, Menlo Park, CA.

[4] Lauritzen, S. L .and Spiegelhalter, D. J. (1988) Local computations with probabilities on graphical structures and their applications to expert systems *J. R. Statist. Soc. B*, **50**, *No. 2*, 157–224.

[5] Jensen, F. V., Lauritzen S. L. and Olesen K. G. (1989) Bayesian updating in recursive graphical models by local computations. Report R 89-15, Institute for Electronic Systems, Department of Mathematics and Computer Science, University of Aalborg, Denmark.

[6] Pearl, J. (1988) *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann Publishers, Inc., San Mateo, Calif.

[7] Peot, M. and Shachter, R. D. (1989) Fusion and Propagation with Multiple Observations in Belief Networks. *to appear.*

[8] Raiffa, H. (1968) Decision Analysis, pp 34–36, 47–50. Addison-Wesley, Reading, Mass.

[9] Rege, A. and Agogino, A. M. (1988) Topological framework for representing and solving probabilistic inference problems in expert systems. *IEEE transactions on Systems, Man and Cybernetics*,**18** (3).

[10] Rubinstein, R.Y. (1981) Simulation and the Monte Carlo Method. Wiley, New York.

[11] Schachter, R. D. (1986) Evaluating influence diagrams. *Operations Research* **34** (6), 871–882.

[12] Shachter, R. D. and Peot, M. A. (1989) Simulation approaches to general probabilistic inference on belief networks. Proceedings of the Fifth Workshop on Uncertainty in AI, Windsor, Ontario.

[13] Shachter, R. D. and Peot, M. A. (1989) Evidential Reasoning Using Likelihood Weighting. submitted to *Artificial Intelligence.*

[14] Srinivas, S. (1992) Generalizing the Noisy Or concept to non-binary variables. Technical Report No. 79, Rockwell International Science Center.

[15] Suermondt, H. J. and Cooper, G. F. (1988) Updating probabilities in multiply connected belief networks. Proceedings, Influence Diagram Workshop, University of California, Berkeley.

[16] Rockwell International Science Center (1992) IDEAL-EDIT : Documentation and User's guide. Technical Report No. 77, Rockwell International Science Center, Palo Alto, CA 94301.

Manual formatted on July 31, 1992.