This document details the architecture of the program that processes inference queries in various uncertainty metrics.

1 Non-recursive data vs recursive data

An important distinction that exists with data is whether it is non-recursive or recursive.

- **Non-recursive data**: Envisioning the data as a directed graph, non-recursive data takes the form of a directed tree. All edges flow from the single root towards the leafs. Non-recursive data structures are the core of the program's architecture. Note that non-recursive data instances can be nested in other instances of the same types, but an instance cannot contain a chain of references back to itself, nor can two chains of references converge.
- **Recursive data**: Envisioning the data as a directed graph, recursive data takes the form of an arbitrary directed graph, and can have loops both directed and undirected. Recursive data structures are the next layer of the program's architecture. Note that recursive data structures can reference themselves in any manner.

2 endless arrays

To aid with the implementation of various algorithms, a set of global arrays are maintained at all times:

The arrays are persistent, are extended on an as needs basis, but are never deallocated, ensuring that the allocated memory persists for the next call to an algorithm.

There are two types of array sets that are maintained: arrays of bytes (whose class has the suffix `i` = "inline"), and arrays of pointers (whose class has the suffix `p` = "pointer").

The arrays are arranged in the following manner:

- At the top level, the arrays are grouped into layers/pages. A counter that indexes the current layer that is "in use" is maintained. When a function is called, the counter is moved to the next layer so that the layer utilized by the parent function call remains unchanged. A the end of the function call, the counter is moved back to its previous value so that the parent function call can continue without its layer having been changed.
- Within each layer, there is an "array of arrays". There is an array of arrays as opposed to a single array due to the fact that some algorithms utilize more than 1 array.
- When bytes are used instead of pointers, data is written to and read from the array in blocks.

3 Non-recursive data

3.1 simple data

The class simple data is a **non-recursive** unified data type that can store any data that does not have a recursive structure. The class stores a type identifier describing the **type** of data that is being referenced, and an anonymous pointer to the data itself.

The following fields are present (the **bold** text denotes the data type):

type_identifier `the_type_id` stores an identifier for the **type** of data that is being referenced.

int `the_type_size` stores the size of the data block that is being referenced.

void* `the_element` is an anonymous pointer that references the data.

The various data structure **type**s that can be stored as `simple_data` are:

void : (`the_type_id` = `TI_VOID`) A null data structure.

An array of bytes : (`the_type_id` = `TI_MEM_BLOCK`) A generic array of bytes of length `the_type_size`.

bool : (`the_type_id` = `TI_BOOL`) A boolean value.

uchar : (`the_type_id` = `TI_UCHAR`) An unsigned byte.

uint16 : (`the_type_id` = `TI_UINT16`) An unsigned 16-bit integer.

int : (`the_type_id` = `TI_INT`) A signed 32-bit integer.

double : (`the_type_id` = `TI_DOUBLE`) A long floating point number.

doublex : (`the_type_id` = `TI_DOUBLEX`) A long floating point number that can attain special values such as $+\infty$, $-\infty$, and NaN.

fuzzy : (`the_type_id` = `TI_FUZZY`) A triangular fuzzy number.

string : (`the_type_id` = `TI_STRING`) A string of characters.

box : (`the_type_id` = `TI_BOX`) A container class that contains a single simple_data structure.

data_pair : (`the_type_id` = `TI_PAIR`) A pair of simple_data structures.

list : (`the_type_id` = `TI_LIST`) A list of simple_data structures.

3.2 boxes

The "box" is a simple container class with a single field that stores a single simple_data instance.

3.3 pairs

Often, simple data is needed to be bundled into pairs or more complex data structures to allow for collections of data to be passed from variable to variable. These structures are referred to as **data pair**s and are **non-recursive**.

3.4 lists

Similar to endless arrays, a common **non-recursive** data structure that will be encountered are arrays of simple_data that can dynamically extend themselves as more entries are required. These dynamically adjustable arrays are referred to as **list**s.

3.5 simple data ordering

Given any two simple data values d_1 and d_2, an ordering is established such that it is always the case that exactly one of the following is true: $d_1 < d_2$; $d_1 = d_2$; or $d_1 > d_2$. Simple data types are lumped together in the ordering, and the ordering of the simple data types is the same as the order in which the types are listed above.

4 Recursive data

4.1 nodes

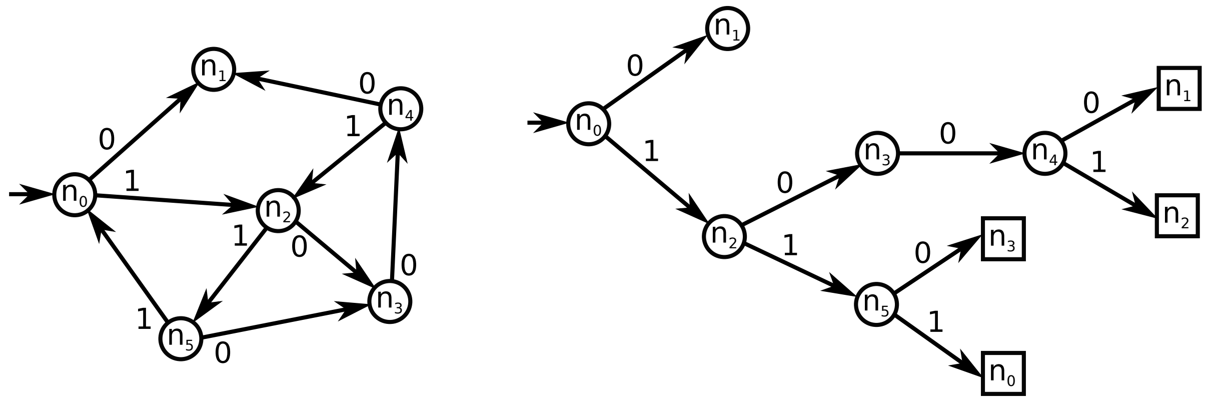
A node is a node in a directed graph that are used to implement **recursive** data structures. Complex data structures such as decision trees/diagrams are manifested via a directed graph that is comprised of nodes.

A node consists of:

- A simple data field.
- An ordered list of pointers to the child nodes. The list is indexed starting from 0.

4.2 sub-graph tracing

Many subroutines that process a directed graph require that the subgraph that is rooted at the input node be traced in a depth-first fashion.

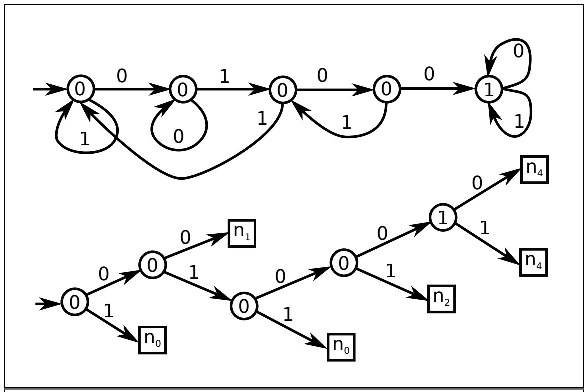


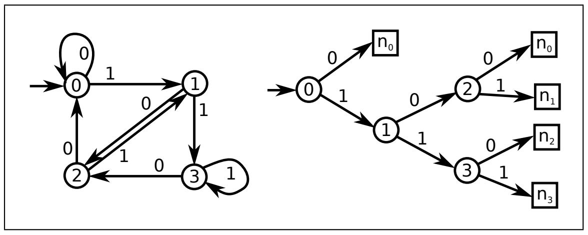
The tracing in this example proceeds as follows:

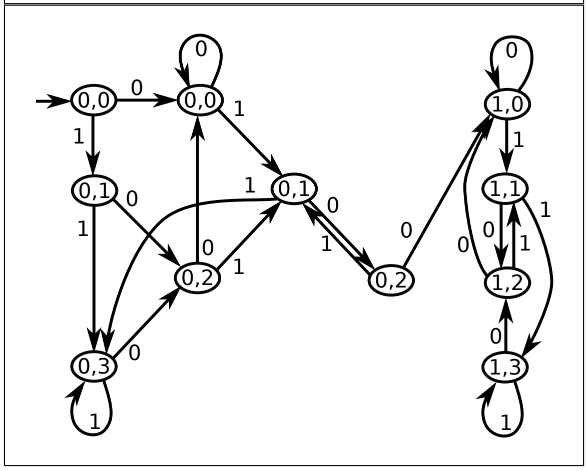
- Start at the root node n_0.
- Transition to child 0 of node n_0, which is n_1.
- Node n_1 has no children, so backtrack to node n_0.
- Transition to child 1 of n_0 which is n_2.
- Transition to child 0 of n_2 which is n_3.
- Transition to child 0 of n_3 which is n_4.
- Child 0 of n_4 (which is n_1) has already been visited, so remain at n_4.
- Child 1 of n_4 (which is n_2) has already been visited, so remain at n_4.
- The children of n_4 have been accounted for, so backtrack to n_3.
- The children of n_3 have been accounted for, so backtrack to n_2.
- Transition to child 1 of n_2 which is n_5.
- Child 0 of n_5 (which is n_3) has already been visited, so remain at n_5.
- Child 1 of n_5 (which is n_0) has already been visited, so remain at n_5.

- The children of n_5 have been accounted for, so backtrack to n_2.
- The children of n_2 have been accounted for, so backtrack to n_0.
- The children of n_0 have been accounted for, so the trace finishes.

4.3 product subgraphs







Top left: A directed graph and its associated depth-first tracer tree that denotes a finite state machine that returns 1 if and only if the sequence 0100 is detected.

Top right: A directed graph and its associated depth-first tracer tree that denotes a finite state machine that returns the decimal number equivalent to the final two bits of the input sequence.

Bottom left: The product directed graph of the two previous directed graphs.

5 File parsing and tokens

5.1 tokens

All input files are parsed into tokens. The rules for extracting the tokens are described in detail below:

To aid in the description, the set of all characters is given the following taxonomy:

Alphabetic

All letters `aAbB...zZ`

Underscore `_`

Numeric `0123456789`

Symbols

> **sign** `+-`
>
> **decimal point** `.`
>
> **double quote** `"`
>
> **escape character** \
>
> **other characters**

Whitespace

To aid in describing the tokens themselves, notation related to regular expressions will be used:

- $\emptyset$ denotes the empty string.
- $\boxed{\texttt{literal}}$ denotes the single string "literal".
- string$_1$string$_2$ denotes the concatenation of string$_1$ and string$_2$.
- (option$_1$ | option$_2$ | ... | option$_n$) denotes a choice between options option$_1$, option$_2$, ..., option$_n$.
- (fragment)* denotes "fragment" repeated 0 or more times. The parentheses may be omitted if not necessary.
- (fragment)$^+$ denotes "fragment" repeated 1 or more times. The parentheses may be omitted if not necessary.
- (options$_2$ − options$_1$) denotes options$_2$ with all options from "options$_1$" excluded.
- **A** denotes an alphabetic character (including the underscore).
- **N** denotes an arbitrary numeric digit.
- **S** denotes an arbitrary symbol.

The tokens themselves belong to one of the following categories:

Alpha-numeric An alpha-numeric token starts with an alphabetic character (letters + underscore) and consists of any sequence of alphabetic and numeric characters. An alpha-numeric token ends on the last alphabetic or numeric character before a non alphabetic and non numeric character. The general form is

$$\mathbf{A}(\mathbf{A} \mid \mathbf{N})^*$$

Integer An integer token is continuous string of numeric characters where the first character may be `+` or `-`. At least one numeric character is required. The plus sign `+` or minus sign `-` on its own is a symbol. The general form is

$$(\emptyset \mid \boxed{\texttt{+}} \mid \boxed{\texttt{-}})\mathbf{N}^+$$

Fixed-point A fixed point number. The general form that is read is

$$(\emptyset \mid \boxed{\texttt{+}} \mid \boxed{\texttt{-}})\mathbf{N}^*\boxed{\texttt{.}}\mathbf{N}^*$$

After processing, 0s pad the decimal point if necessary giving

$$(\emptyset \mid \boxed{\texttt{+}} \mid \boxed{\texttt{-}})(\mathbf{N}^+ \mid \boxed{\texttt{0}})\boxed{\texttt{.}}(\mathbf{N}^+ \mid \boxed{\texttt{0}})$$

Floating-point A floating point number. The general form that is read is

$$(\emptyset \mid \boxed{\texttt{+}} \mid \boxed{\texttt{-}})\mathbf{N}^*\boxed{\texttt{.}}\mathbf{N}^*\boxed{\texttt{e}}(\emptyset \mid \boxed{\texttt{+}} \mid \boxed{\texttt{-}})\mathbf{N}^*$$

After processing, 0s pad the decimal point and exponent if necessary giving

$$(\emptyset \mid \boxed{\texttt{+}} \mid \boxed{\texttt{-}})(\mathbf{N}^+ \mid \boxed{\texttt{0}})\boxed{\texttt{.}}(\mathbf{N}^+ \mid \boxed{\texttt{0}})\boxed{\texttt{e}}(\emptyset \mid \boxed{\texttt{+}} \mid \boxed{\texttt{-}})(\mathbf{N}^+ \mid \boxed{\texttt{0}})$$

String A string of characters. The general form that is read is

$$\boxed{\texttt{"}}(((\mathbf{A} \mid \mathbf{N} \mid \mathbf{S}) - (\boxed{\texttt{"}} \mid \boxed{\backslash} \mid \boxed{\{} \mid \boxed{\}})) \mid (\boxed{\backslash}(\mathbf{A} \mid \mathbf{N} \mid \mathbf{S})))^*\boxed{\texttt{"}}$$

After the encapsulating double quotes have been removed and the escape characters have been resolved, the general form is

$$(\mathbf{A} \mid \mathbf{N} \mid \mathbf{S})^*$$

Symbol An isolated symbol:

$$\mathbf{S} - (\boxed{\texttt{.}} \mid \boxed{\texttt{"}} \mid \boxed{\{} \mid \boxed{\}})$$

In addition, a comment is inert text that is skipped by the token reader. A comment is enclosed by curly braces {}.

5.2 simple data syntax

For data input and output, it is necessary to represent simple data as a list of tokens that can be read or written by both a human and machine. This section will describe the text syntax that is used for various types using the same syntax for regular expressions that was described above. When whitespace is present, the amount of type of whitespace is irrelevant. The underlined text denotes fields that filled with relevant data.

void : $\boxed{\texttt{0}}$

An array of bytes : No text representation.

bool : $\boxed{\texttt{bool }}(\boxed{\texttt{0}} \mid \boxed{\texttt{1}})$

uchar : $\boxed{\texttt{c }}\,\underline{\text{the character}}$

uint16 : $\boxed{\texttt{uint16 }}\,\underline{\text{the integer}}$

int : $\boxed{\texttt{i }}\,\underline{\text{the integer}}$

double : $\boxed{\texttt{d }}\,\underline{\text{the double}}$

doublex : $\boxed{\texttt{dx }}\,(\underline{\text{the double}} \mid \boxed{\texttt{pos\_inf}} \mid \boxed{\texttt{neg\_inf}} \mid \boxed{\texttt{NaN}})$

fuzzy : $\boxed{\texttt{fuzzy (}}\,\underline{\text{lower doublex}}\,\boxed{\texttt{ , }}\,\underline{\text{center doublex}}\,\boxed{\texttt{ , }}\,\underline{\text{upper doublex}}\,\boxed{\texttt{)}}$

string : $\boxed{\texttt{s "}}\,\underline{\text{the string}}\,\boxed{\texttt{"}}$
Use \ to escape \, ", {, or } (\\ ↦ \; \" ↦ "; \{ ↦ {; and \} ↦ })

box : $\boxed{\texttt{box }}\,\underline{\text{the data}}$

data_pair : $\boxed{\texttt{pair < }}\,\underline{\text{data 1}}\,\boxed{\texttt{ , }}\,\underline{\text{data 2}}\,\boxed{\texttt{ >}}$

list : $\boxed{\texttt{list }}\,\underline{\text{length}}\,\boxed{\texttt{ < }}\,(\emptyset \mid (\underline{\text{data}}(\boxed{\texttt{ , }}\,\underline{\text{data}})^*))\,\boxed{\texttt{ >}}$

5.3 recursive data syntax

A directed graph starts with a definition of its root node.

A root node has the following syntax:

$$\underline{\text{root node}} = (\underline{\text{new node}} \mid \underline{\text{molded table}})$$

An interior/child node has the following syntax:

$$\underline{\text{child node}} = (\underline{\text{new node}} \mid \underline{\text{back reference}} \mid \underline{\text{molded table}})$$

The meaning of the fields new node, back reference, and molded table are explained below:

5.3.1 new node syntax

A new node in the subgraph, denoted by field new node, in the directed graph is represented in text via the following format:

$$\underline{\text{new node}} = \boxed{\texttt{new}}\ \underline{\text{the data}}(\boxed{\texttt{*}} \mid (\boxed{\texttt{<}}\ \underline{\text{child node}}(\boxed{\texttt{,}}\ \underline{\text{child node}})^{*}\boxed{\texttt{>}}))$$

Field the data refers to the data stored directly in the current node.

The symbol `*` indicates the absence of children. A pair of triangular braces `< >` denotes a list of children separated by commas (`,`).

5.3.2 back reference syntax

A "back reference", denoted by field back reference, references a previously defined node in the subgraph's definition.

absolute reference Denoted by $\boxed{\texttt{ref}}$ node index, an absolute reference is an integer that indexes the referenced node's position (starting from 0) when the nodes are ordered according to when they were defined.

relative reference Denoted by $\boxed{\texttt{addr}}$ relative address $\boxed{\texttt{@}}$, a relative reference is a list of symbols that describes the position of the referenced node relative to the node whose child is currently being described. The relative address is a list of the following symbols. Starting with a reference to the node whose child is currently being described,

- `!` moves the reference pointer to the root node.
- `^` moves the reference pointer up to the prime parent of the node currently being referenced. The "prime parent" of a node is the parent that precedes the node during a depth-first trace.
- an integer token moves the reference pointer to the child indexed by the integer.

5.3.3 molded table syntax

In many scenarios, such as the representation of conditional probability tables, the directed graph depicts a multi-dimensional array in the form of a decision diagram. This decision diagram will often have a full tree structure except for some layers where the decision variables have no impact. In inactive decision layers, all children of each decision node reference the same child. Denoting a full tree using the above notation is often cumbersome, and a simpler approach would be to list all of the leaf values, alongside data describing the shape of the decision tree/diagram.

A molded table, denoted by field $\underline{\text{molded table}}$, has the syntax:

$$\underline{\text{molded table}}$$
$$= \boxed{\texttt{table}}\ \underline{\text{num of layers}} \sqcup (\ \underline{\text{variable description}} \sqcup)^{*} \sqcup \underline{\text{default value}} \sqcup (\underline{\text{array entry}} \sqcup)^{+}$$

The field $\underline{\text{variable description}}$ has the syntax:

$$\underline{\text{variable description}}$$
$$= (\boxed{\texttt{normal}} \mid \boxed{\texttt{inactive}} \mid \boxed{\texttt{specific}})\ \underline{\text{variable name}} \sqcup \underline{\text{domain size}} \sqcup (\emptyset \mid \underline{\text{specific child index}})$$

Field $\underline{\text{num of layers}}$ denotes the nonnegative number of decision layers in the decision tree, not counting the leaf node layer. The $\underline{\text{num of layers}}$ field is followed by a list of details about each layer, starting from the root layer. Each description is denoted by the field $\underline{\text{variable description}}$, whose syntax is given above.

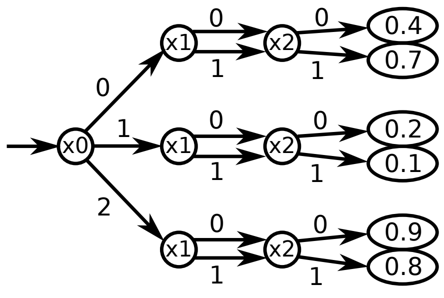
Each decision layer corresponds to a variable whose name is any simple data value. The variable's name, denoted by the field $\underline{\text{variable name}}$ is assigned to the data field of each node in the variable's decision layer. In addition, all nodes in a variable's decision layer have the same number of children equal to the size of the variable's domain, indicated by the field $\underline{\text{domain size}}$. Each variable is one of three types described below:

- $\boxed{\texttt{normal}}$ indicates that the children of each node in the current layer all reference different children, and that the variable has an impact on the decision. The field $\underline{\text{specific child index}}$ is not used in this case.
- $\boxed{\texttt{inactive}}$ indicates that the children of each node in the current layer all reference the same child. This effectively renders the variable "inactive" in the decision making process. The field $\underline{\text{specific child index}}$ is not used in this case.
- $\boxed{\texttt{specific}}$ indicates that all child references except for the child indexed by $\underline{\text{specific child index}}$ reference a node that leads to a leaf node labeled with the simple data $\underline{\text{default value}}$.

After the list of $\underline{\text{variable description}}$ fields, a simple data default value $\underline{\text{default value}}$ is provided, which is then followed by a list of simple data values $\underline{\text{array entry}}$ that populate the data field of each leaf node.

Below is shown an example molded table: the syntax on the left, and the manifested directed graph/decision diagram is shown on the right. There are 3 decision variables named `x0`, `x1`, and `x2`, with respective domain sizes 3, 2, and 2. Variables `x0` and `x2` are "normal", while variable `x1` is "inactive".

```
table 3
normal   s "x0" 3
inactive s "x1" 2
normal   s "x2" 2
0
d 0.4  d 0.7  d 0.2  d 0.1  d 0.9  d 0.8
```

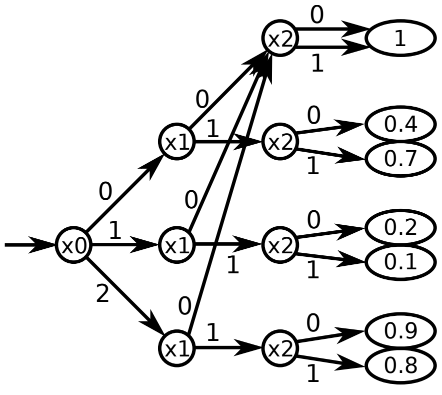


Without the molded table syntax, the decision diagram would be denoted by the following syntax:

```
new s "x0" <
    new s "x1" < new s "x2" <
        new d 0.4 * ,
        new d 0.7 * > ,
    addr 0 @ > ,
    new s "x1" < new s "x2" <
        new d 0.2 * ,
        new d 0.1 * > ,
    addr 0 @ > ,
    new s "x1" < new s "x2" <
        new d 0.9 * ,
        new d 0.8 * > ,
    addr 0 @ > >
```

Below is shown another example molded table: the syntax on the left, and the manifested directed graph/decision diagram is shown on the right. There are 3 decision variables named `x0`, `x1`, and `x2`, with respective domain sizes 3, 2, and 2. Variables `x0` and `x2` are "normal", while variable `x1` is "specific". For the nodes labeled with `x1`, all children except for child 1 reference an identity decision diagram that always returns the specified default value: `bool 1`

```
table 3
normal   s "x0" 3
specific s "x1" 2 1
normal   s "x2" 2
bool 1
d 0.4  d 0.7  d 0.2  d 0.1  d 0.9  d 0.8
```

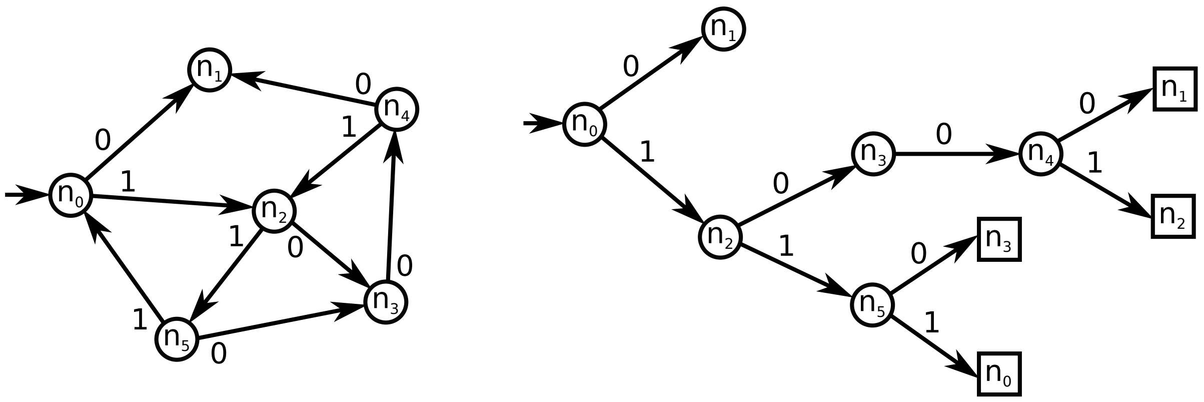


Without the molded table syntax, the decision diagram would be denoted by the following syntax:

```
new s "x0" <
    new s "x1" <
        new s "x2" <
            new bool 1 * ,
            addr 0 @ > ,
        new s "x2" <
            new d 0.4 * ,
            new d 0.7 * > > ,
    new s "x1" <
        addr ! 0 0 @ ,
        new s "x2" <
            new d 0.2 * ,
            new d 0.1 * > > ,
    new s "x1" <
        addr ! 0 0 @ ,
        new s "x2" <
            new d 0.9 * ,
            new d 0.8 * > > >
```

5.3.4 example directed graph

The following directed graph can be denoted by the following strings:



The directed graph is shown on the left, while the depth-first tracer tree is shown on the right.

Using absolute back references gives the string:

```
new s "n_0" < new s "n_1" * , new s "n_2" < new s "n_3" < new s "n_4" < ref 1 , ref 2 > >
, new s "n_5" < ref 3 , ref 0 > > >
```

Using relative back references gives the string:

```
new s "n_0" < new s "n_1" * , new s "n_2" < new s "n_3" < new s "n_4" < addr ! 0 @ , addr
! 1 @ > > , new s "n_5" < addr ! 1 0 @ , addr ! @> > >
```

or the string:

```
new s "n_0" < new s "n_1" * , new s "n_2" < new s "n_3" < new s "n_4" < addr ^ ^ ^ 0 @ , addr
^ ^ @ > > , new s "n_5" < addr ^ 0 @ , addr ^ ^ @ > > >
```

6 Arithmetic

Arithmetic can be performed on elements of all data types, even on elements with different data types. The main unary operators are negation and inverting, and the main binary operators are addition and multiplication.

7 Input Syntax

Given an experiment number *i*, two input files are required: "`Test Files/`*i*`_data.txt`" and "`Test Files/`*i*`_instructions.txt`".

The file "`Test Files/`*i*`_data.txt`" contains the input data that is to be processed. The input data takes the form of a list of directed graphs.The format and syntax of "`Test Files/`*i*`_data.txt`" is the following:

number of directed graphs

name 1 graph 1

name 2 graph 2

...

```
name n graph n
```

Each name is a single token, and each graph is self contained.

The file "`Test Files/`*i*`_instructions.txt`" contains the instructional data that describes how the input data is to be processed. The format and syntax of "`Test Files/`*i*`_instructions.txt`" is the following:

```
command 1 argument list 1

command 2 argument list 2

...

command m argument_list_m
```

Each command is followed by a list of arguments that are relevant to that command. The possible commands are listed below. Arguments are denoted by the underlined text. Each string argument must be read as a single token, which is done by enclosing the string in double quotes.

- `assign_new_subgraph` dest name new graph : Creates a new subgraph defined by new graph, and assigns the root node to dest name.
- `assign_node` dest name source name depth path indices : Assigns a node to dest name that is determined by the following: Starting from the node assigned to source name, a total of depth directed edges are traversed, indexed by path indices.
- `copy_node` dest name source name : Copies the node assigned to source name and assigns the copy to dest name. The copy node has the same children as the copied node.
- `copy_subgraph` dest name source name : Copies the subgraph rooted at the node assigned to source name and assigns the root of the copy to dest name.
- `condense` dest name source name : Creates a condensed copy of the subgraph rooted at the node assigned to source name and assigns the root of the condensed copy to dest name.
- `expand` dest name source name level : Copies the subgraph rooted at the node assigned to source name and expands the copy subgraph at level level (0 is the root level) and assigns the root of the expanded copy to dest name.
- `subgraph_product` dest name max degree flag source name 1 source name 2 : Creates a "product subgraph" from the subgraphs rooted at source name 1 and source name 2, and assigns the root node to dest name. max degree flag is `0` if children that are not common to both nodes are excluded, and is `1` if otherwise.
- `binary_operator` dest name operator max degree flag source name 1 source name 2 : Applies the binary operator operator to the subgraphs rooted at source name 1 and source name 2 and assigns the result to dest name. `+` denotes addition; `*` denotes multiplication; `-` denotes subtraction; `/` denotes division; `max` denotes maximization; and `min` denotes minimization. max degree flag is `0` if children that are not common to both nodes are excluded, and is `1` if otherwise.

- `add_subgraphs` dest name max_degree_flag n arg_1 arg_2 ... arg_n : Assigns the sum of the n subgraphs rooted at arg_1 arg_2 ... arg_n to dest name. max_degree_flag is `0` if children that are not common to both nodes are excluded, and is `1` if otherwise.

- `multiply_subgraphs` dest name max_degree_flag n arg_1 arg_2 ... arg_n : Assigns the product of the n subgraphs rooted at arg_1 arg_2 ... arg_n to dest name. max_degree_flag is `0` if children that are not common to both nodes are excluded, and is `1` if otherwise.

- `unary_operator` dest name operator source name : Applies the unary operator operator to the subgraph rooted at source name and assigns the result to dest name. `-` denotes negation; and `/` denotes inversion.

- `negate_subgraph` dest name source name : Assigns the negative of the subgraph rooted at source name to dest name.

- `invert_subgraph` dest name source name : Assigns the inverse (reciprocal) of the subgraph rooted at source name to dest name.

- `marginalize` dest name keep degree flag source name level : Copies the subgraph rooted at source name to the dest name. The copy is then modified as follows: Trace through the subgraph rooted at dest name. When a node in the depth-first tracer tree is encountered at a depth of level, all child subgraphs are added together into a single subgraph that the child pointers now point to. If keep degree flag is `1`, then the degree of the node remains constant with all child pointers now pointing to the sum node, otherwise at most 1 child pointer that points to the sum node exists (if the degree is 0, then no children exist). Level 0 refers to the root level.

- `condition` dest name keep degree flag source name level child index : Copies the subgraph rooted at source name to the dest name. The copy is then modified as follows: Trace through the subgraph rooted at dest name. When a node in the depth-first tracer tree is encountered at a depth of level, all child subgraphs, save for the child subgraph indexed by child index, are deleted. If keep degree flag is `1`, then the degree of the node remains constant with all child pointers now pointing to the remaining child subgraph, otherwise 1 child pointer points to the remaining subgraph. Level 0 refers to the root level.

- `DS_binary_operator` dest name operator max degree flag source name 1 source name 2 : Applies the binary operator operator to the subgraphs rooted at source name 1 and source name 2 and assigns the result to dest name. The root nodes are handled differently from the command `binary_operator`. The resultant root node has a child corresponding to each pairwise application of the binary operator to a child of source name 1 and a child of source name 2. `+` denotes addition; `*` denotes multiplication; `-` denotes subtraction; `/` denotes division; `max` denotes maximization; and `min` denotes minimization. max degree flag is `0` if children that are not common to both nodes are excluded, and is `1` if otherwise.

- `DS_collapse` dest name source name : Copies the DS structure rooted at source name and collapses it by unifying together focal element pairs with common focal elements and adding the weights; excluding empty focal elements; as well as normalizing the weights so that all weights sum to 1.

- `print_string` the string : Prints the string the string to the file "`Test Files/output_`i`.txt`" and to the console. the string must be encapsulated by double quotes to be read as a single token.

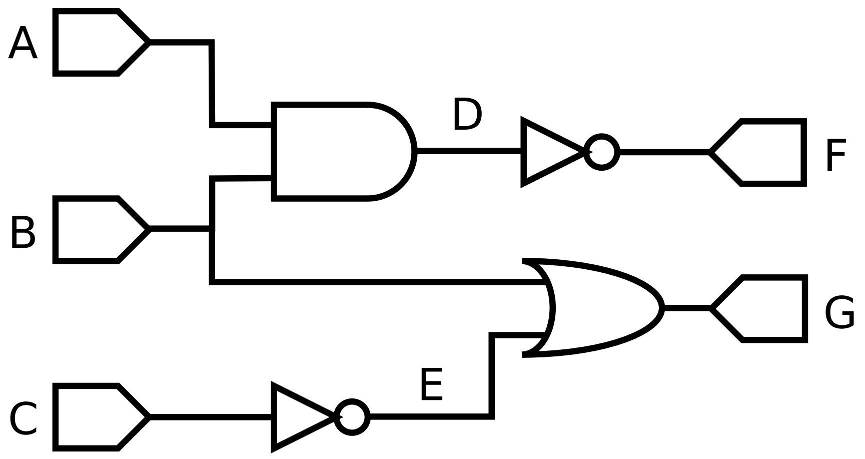
- `print_data` the source : Prints the simple data stored in the node indexed by the source to the file "`Test Files/output_`i`.txt`" and to the console.

- `print_subgraph` the source : Prints the subgraph rooted at the node assigned to the source to the file "`Test Files/output_`i`.txt`" and to the console.

- `comment` text `comment_end` : This command does nothing, and exists only to allow comments in the instructions file.

8 Example

This section will demonstrate the utility of the system by performing an analysis of a noisy multivalued logic circuit. The multivalued logic that is being used is ternary (base 3) logic. The noisy ternary circuit that will be under consideration is shown below:



The prior probability distributions that describe the input variables are given below:

A	0	1	2
$\Pr(A)$	0.33	0.33	0.34

B	0	1	2
$\Pr(B)$	0.7	0.2	0.1

C	0	1	2
$\Pr(C)$	0.1	0.8	0.1

$D = A$ AND B: A multivalued AND ideally returns the smaller of the two operands, but in this scenario, the output might not be pulled down to the smaller value.

$E =$ NOT C: A multivalued NOT ideally returns the complement of the operand, but in this scenario, the output might not be pulled to the correct output.

D	0	1	2
$\Pr(D\|A=0,B=0)$	1.00	0.00	0.00
$\Pr(D\|A=0,B=1)$	0.90	0.10	0.00
$\Pr(D\|A=0,B=2)$	0.80	0.10	0.10
$\Pr(D\|A=1,B=0)$	0.90	0.10	0.00
$\Pr(D\|A=1,B=1)$	0.00	1.00	0.00
$\Pr(D\|A=1,B=2)$	0.00	0.90	0.10
$\Pr(D\|A=2,B=0)$	0.80	0.10	0.10
$\Pr(D\|A=2,B=1)$	0.00	0.90	0.10
$\Pr(D\|A=2,B=2)$	0.00	0.00	1.00

E	0	1	2
$\Pr(E\|C=0)$	0.10	0.10	0.80
$\Pr(E\|C=1)$	0.10	0.80	0.10
$\Pr(E\|C=2)$	0.80	0.10	0.10

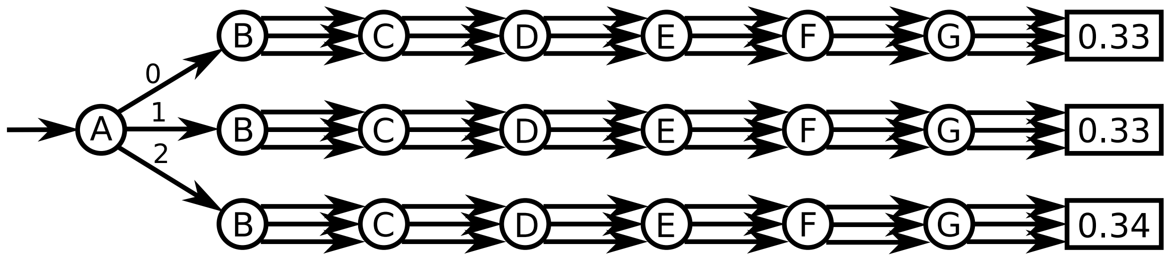
$F =$ NOT D: A multivalued NOT ideally returns the complement of the operand, but in this scenario, the output might not be pulled to the correct output.

$G = B$ OR E: A multivalued OR ideally returns the larger of the two operands, but in this scenario, the output might not be pulled up to the larger value.

F	0	1	2
$\Pr(F\|D=0)$	0.10	0.10	0.80
$\Pr(F\|D=1)$	0.10	0.80	0.10
$\Pr(F\|D=2)$	0.80	0.10	0.10

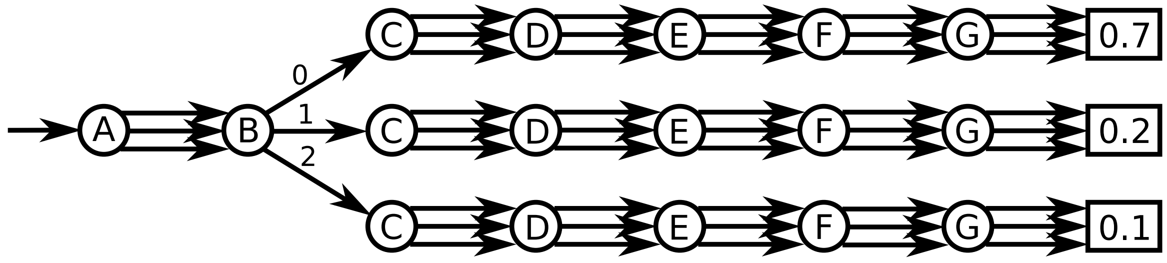
G	0	1	2
$\Pr(G\|B=0,E=0)$	1.00	0.00	0.00
$\Pr(G\|B=0,E=1)$	0.10	0.90	0.00
$\Pr(G\|B=0,E=2)$	0.10	0.10	0.80
$\Pr(G\|B=1,E=0)$	0.10	0.90	0.00
$\Pr(G\|B=1,E=1)$	0.00	1.00	0.00
$\Pr(G\|B=1,E=2)$	0.00	0.10	0.90
$\Pr(G\|B=2,E=0)$	0.10	0.10	0.80
$\Pr(G\|B=2,E=1)$	0.00	0.10	0.90
$\Pr(G\|B=2,E=2)$	0.00	0.00	1.00

The decision diagram and syntax for $\Pr(A)$:



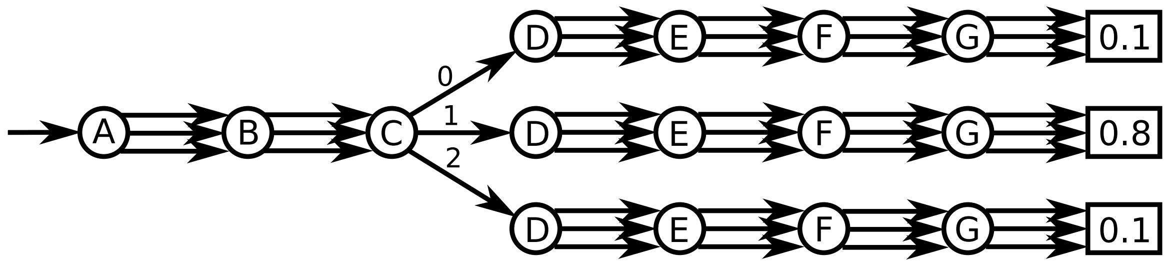
```
table 7
normal   s "A" 3  inactive s "B" 3  inactive s "C" 3  inactive s "D" 3
inactive s "E" 3  inactive s "F" 3  inactive s "G" 3
0
d 0.33  d 0.33  d 0.34
```

The decision diagram and syntax for $\Pr(B)$:



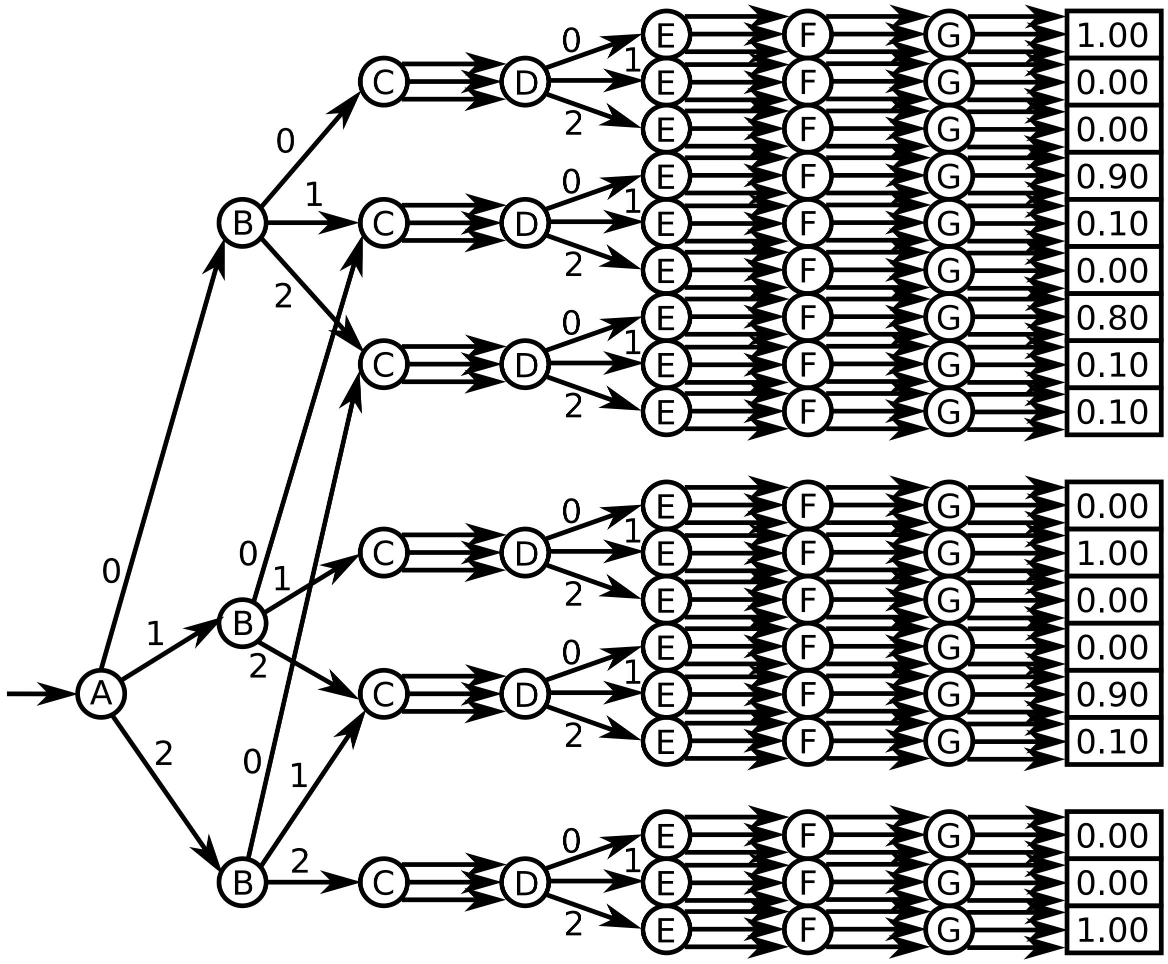
```
table 7
inactive s "A" 3  normal   s "B" 3  inactive s "C" 3  inactive s "D" 3
inactive s "E" 3  inactive s "F" 3  inactive s "G" 3
0
d 0.7  d 0.2  d 0.1
```

The decision diagram and syntax for $\Pr(C)$:



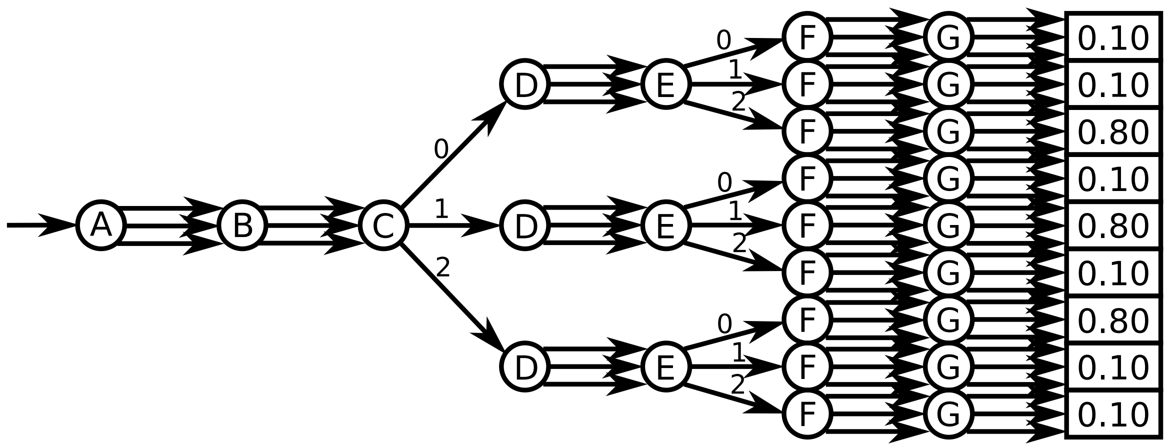
```
table 7
inactive s "A" 3  inactive s "B" 3  normal   s "C" 3  inactive s "D" 3
inactive s "E" 3  inactive s "F" 3  inactive s "G" 3
0
d 0.1  d 0.8  d 0.1
```

The decision diagram and syntax for $\Pr(D|A,B)$:



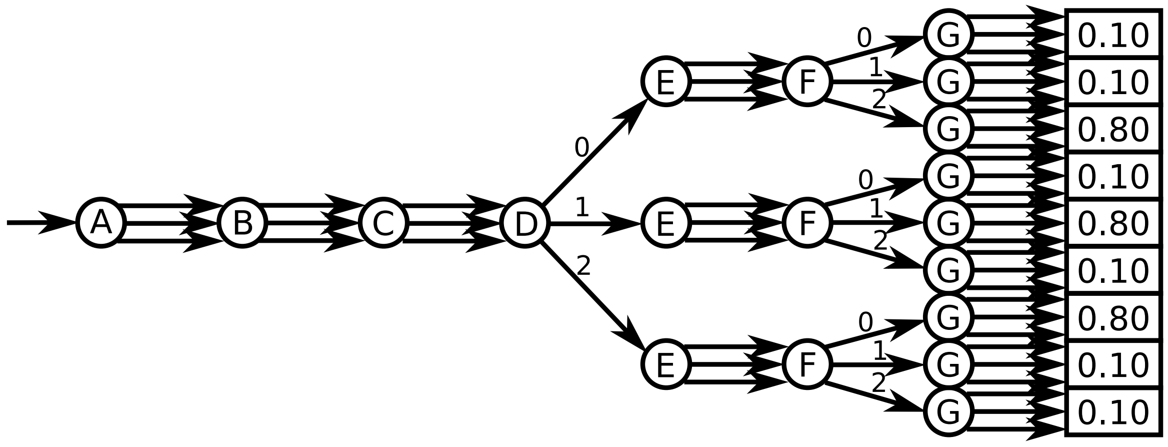
```
new s "A" <
    {A=0} new s "B" <
        {B=0} table 5
        inactive s "C" 3  normal   s "D" 3  inactive s "E" 3  inactive s "F" 3
        inactive s "G" 3
        0
        d 1.00  d 0.00  d 0.00 ,
        {B=1} table 5
        inactive s "C" 3  normal   s "D" 3  inactive s "E" 3  inactive s "F" 3
        inactive s "G" 3
        0
        d 0.90  d 0.10  d 0.00 ,
        {B=2} table 5
        inactive s "C" 3  normal   s "D" 3  inactive s "E" 3  inactive s "F" 3
        inactive s "G" 3
        0
        d 0.80  d 0.10  d 0.10 > ,
    {A=1} new s "B" <
        {B=0} addr ! 0 1 @ ,
        {B=1} table 5
        inactive s "C" 3  normal   s "D" 3  inactive s "E" 3  inactive s "F" 3
        inactive s "G" 3
        0
        d 0.00  d 1.00  d 0.00 ,
        {B=2} table 5
        inactive s "C" 3  normal   s "D" 3  inactive s "E" 3  inactive s "F" 3
        inactive s "G" 3
        0
        d 0.00  d 0.90  d 0.10 > ,
    {A=2} new s "B" <
        {B=0} addr ! 0 2 @ ,
        {B=1} addr ! 1 2 @ ,
        {B=2} table 5
        inactive s "C" 3  normal   s "D" 3  inactive s "E" 3  inactive s "F" 3
        inactive s "G" 3
        0
        d 0.00  d 0.00  d 1.00 > >
```

The decision diagram and syntax for $\Pr(E|C)$:



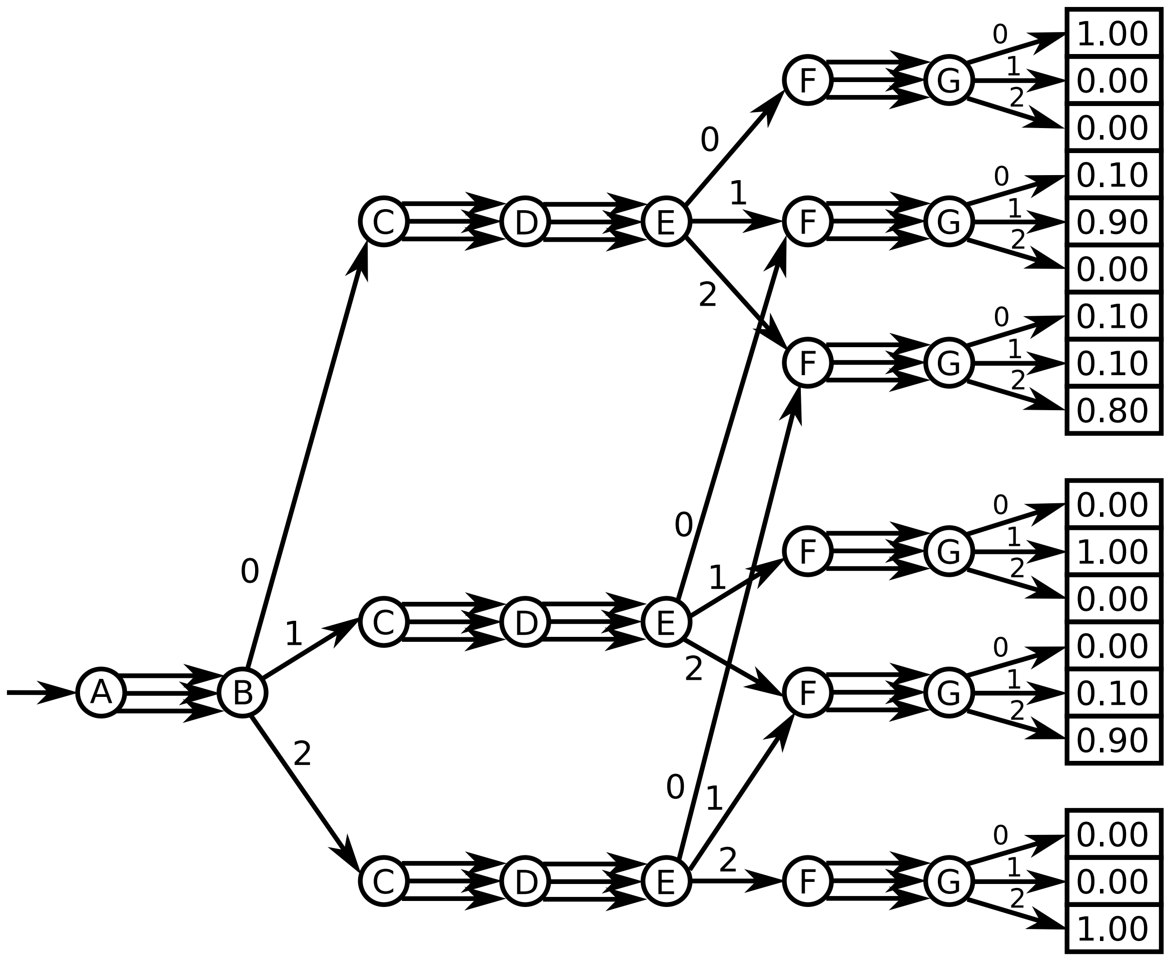
```
table 7
inactive s "A" 3  inactive s "B" 3  normal   s "C" 3  inactive s "D" 3
normal   s "E" 3  inactive s "F" 3  inactive s "G" 3
0
d 0.10  d 0.10  d 0.80   d 0.10  d 0.80  d 0.10   d 0.10  d 0.10  d 0.80
```

The decision diagram and syntax for $\Pr(F|D)$:



```
table 7
inactive s "A" 3  inactive s "B" 3  inactive s "C" 3  normal   s "D" 3
inactive s "E" 3  normal   s "F" 3  inactive s "G" 3
0
d 0.10  d 0.10  d 0.80   d 0.10  d 0.80  d 0.10   d 0.10  d 0.10  d 0.80
```

The decision diagram and syntax for $\Pr(G|B,E)$:



```
new s "A" < new s "B" <
    {B=0} new s "C" < new s "D" < new s "E" <
        {E=0} table 2
        inactive s "F" 3  normal   s "G" 3
        0
        d 1.00  d 0.00  d 0.00 ,
        {E=1} table 2
        inactive s "F" 3  normal   s "G" 3
        0
        d 0.10  d 0.90  d 0.00 ,
        {E=2} table 2
        inactive s "F" 3  normal   s "G" 3
        0
        d 0.10  d 0.10  d 0.80 > ,
    addr 0 @ , addr 0 @ > , addr 0 @ , addr 0 @ > ,
    {B=1} new s "C" < new s "D" < new s "E" <
        {E=0} addr ! 0 0 0 0 1 @ ,
        {E=1} table 2
        inactive s "F" 3  normal   s "G" 3
        0
        d 0.00  d 1.00  d 0.00 ,
        {E=2} table 2
        inactive s "F" 3  normal   s "G" 3
        0
        d 0.00  d 0.10  d 0.90 > ,
    addr 0 @ , addr 0 @ > , addr 0 @ , addr 0 @ > ,
    {B=2} new s "C" < new s "D" < new s "E" <
        {E=0} addr ! 0 0 0 0 2 @ ,
        {E=1} addr ! 0 1 0 0 2 @ ,
        {E=2} table 2
        inactive s "F" 3
        normal   s "G" 3
        0
        d 0.00  d 0.00  d 1.00 > ,
    addr 0 @ , addr 0 @ > , addr 0 @ , addr 0 @ > > ,
addr 0 @ , addr 0 @ >
```

8.1 Probabilistic Inference Example

This section will detail using the above data alongside the above instructions to extract data about the above noisy ternary logic circuit.

Consider a scenario where it is known that $A = 1$, $B = 0$, and $C = 0$. Of interest is the joint marginal probability distribution for the outputs F and G.

The contents of the instructions file `Test Files/1_instructions.txt` is:

```
{Applying the evidence}

copy_subgraph "Pr(A|evidence)" "Pr(A)"
condition "Pr(A|evidence)" 0 "Pr(A|evidence)" 0 1
condition "Pr(A|evidence)" 0 "Pr(A|evidence)" 1 0
condition "Pr(A|evidence)" 0 "Pr(A|evidence)" 2 0

copy_subgraph "Pr(B|evidence)" "Pr(B)"
condition "Pr(B|evidence)" 0 "Pr(B|evidence)" 0 1
condition "Pr(B|evidence)" 0 "Pr(B|evidence)" 1 0
condition "Pr(B|evidence)" 0 "Pr(B|evidence)" 2 0

copy_subgraph "Pr(C|evidence)" "Pr(C)"
condition "Pr(C|evidence)" 0 "Pr(C|evidence)" 0 1
condition "Pr(C|evidence)" 0 "Pr(C|evidence)" 1 0
condition "Pr(C|evidence)" 0 "Pr(C|evidence)" 2 0

copy_subgraph "Pr(D|evidence)" "Pr(D|A,B)"
condition "Pr(D|evidence)" 0 "Pr(D|evidence)" 0 1
condition "Pr(D|evidence)" 0 "Pr(D|evidence)" 1 0
condition "Pr(D|evidence)" 0 "Pr(D|evidence)" 2 0

copy_subgraph "Pr(E|evidence)" "Pr(E|C)"
condition "Pr(E|evidence)" 0 "Pr(E|evidence)" 0 1
condition "Pr(E|evidence)" 0 "Pr(E|evidence)" 1 0
condition "Pr(E|evidence)" 0 "Pr(E|evidence)" 2 0

copy_subgraph "Pr(F|evidence)" "Pr(F|D)"
condition "Pr(F|evidence)" 0 "Pr(F|evidence)" 0 1
condition "Pr(F|evidence)" 0 "Pr(F|evidence)" 1 0
condition "Pr(F|evidence)" 0 "Pr(F|evidence)" 2 0

copy_subgraph "Pr(G|evidence)" "Pr(G|B,E)"
condition "Pr(G|evidence)" 0 "Pr(G|evidence)" 0 1
condition "Pr(G|evidence)" 0 "Pr(G|evidence)" 1 0
condition "Pr(G|evidence)" 0 "Pr(G|evidence)" 2 0

{multiplying together the conditioned factors}

binary_operator "Pr(total)" * 1 "Pr(A|evidence)" "Pr(B|evidence)"
binary_operator "Pr(total)" * 1 "Pr(total)" "Pr(C|evidence)"
binary_operator "Pr(total)" * 1 "Pr(total)" "Pr(D|evidence)"
binary_operator "Pr(total)" * 1 "Pr(total)" "Pr(E|evidence)"
binary_operator "Pr(total)" * 1 "Pr(total)" "Pr(F|evidence)"
binary_operator "Pr(total)" * 1 "Pr(total)" "Pr(G|evidence)"

{marginalizing out the unnecessary variables}

marginalize "Pr(total)" 0 "Pr(total)" 3
marginalize "Pr(total)" 0 "Pr(total)" 4

{calculating and applying the normalization constant}

copy_subgraph "Z" "Pr(total)"
marginalize "Z" 1 "Z" 5
marginalize "Z" 1 "Z" 6

binary_operator "Pr(total)" / 1 "Pr(total)" "Z"

{print the posterior probabilities}

assign_node "probe" "Pr(total)" 7 0 0 0 0 0 0 0
print_data "probe"
assign_node "probe" "Pr(total)" 7 0 0 0 0 0 0 1
print_data "probe"
assign_node "probe" "Pr(total)" 7 0 0 0 0 0 0 2
print_data "probe"
print_string ""
assign_node "probe" "Pr(total)" 7 0 0 0 0 0 1 0
print_data "probe"
assign_node "probe" "Pr(total)" 7 0 0 0 0 0 1 1
print_data "probe"
assign_node "probe" "Pr(total)" 7 0 0 0 0 0 1 2
print_data "probe"
print_string ""
```

```
assign_node "probe" "Pr(total)" 7 0 0 0 0 0 2 0
print_data "probe"
assign_node "probe" "Pr(total)" 7 0 0 0 0 0 2 1
print_data "probe"
assign_node "probe" "Pr(total)" 7 0 0 0 0 0 2 2
print_data "probe"
```

The contents of `Test Files/1_output.txt` is:

```
d 0.019000

d 0.017000

d 0.064000



d 0.032300

d 0.028900

d 0.108800



d 0.138700

d 0.124100

d 0.467200
```

This yields the following posterior probability distribution over F and G:

G	0	1	2
$\Pr(F=0,G)$	0.0190	0.0170	0.0640
$\Pr(F=1,G)$	0.0323	0.0289	0.1088
$\Pr(F=2,G)$	0.1387	0.1241	0.4672