# Probabilistic Logic Programming
# Review of sintax and semantics. Inference

Elena Bellodi

Department of Engineering
University of Ferrara, Italy

University of Udine, PhD Course, December 4th-5th 2018

# Outline

# Logic Programs with Annotated Disjunctions (LPADs)

$$sneezing(X) : 0.7 \vee null : 0.3 \leftarrow flu(X).$$
$$sneezing(X) : 0.8 \vee null : 0.2 \leftarrow hay\_fever(X).$$
$$flu(bob).$$
$$hay\_fever(bob).$$

- **Distributions over the head of rules**
- *null* does not appear in the body of any rule, and is usually omitted

# ProbLog

$$sneezing(X) \leftarrow flu(X), flu\_sneezing(X).$$
$$sneezing(X) \leftarrow hay\_fever(X), hay\_fever\_sneezing(X).$$
$$flu(bob).$$
$$hay\_fever(bob).$$
$$0.7 :: flu\_sneezing(X).$$
$$0.8 :: hay\_fever\_sneezing(X).$$

- **Distributions over (probabilistic) facts**
- Worlds obtained by selecting or not every grounding of each probabilistic fact

This program has $2 \cdot 2 = 4$ possible worlds.

# Distribution Semantics

- We consider the case of programs without functions symbols
- An LPAD defines a **probability distribution over normal logic programs called worlds**
- A world is obtained from an LPAD by:
  1. grounding the program
  2. selecting a single head atom for each ground clause
  3. including in the world the clause with the selected head atom and the body
- The probability of a world is the product of the probabilities associated to the heads selected
- The probability of a ground atom (the query) is given by the sum of the probabilities of the worlds where the query is true
- If the LPAD contains function symbols, the definition is more complex

# Worlds for the LPAD Program

$sneezing(bob) \leftarrow flu(bob).$
$sneezing(bob) \leftarrow hay\_fever(bob).$
$flu(bob).$
$hay\_fever(bob).$
$\mathbf{P(w_1) = 0.7 \times 0.8}$

$null \leftarrow flu(bob).$
$sneezing(bob) \leftarrow hay\_fever(bob).$
$flu(bob).$
$hay\_fever(bob).$
$\mathbf{P(w_2) = 0.3 \times 0.8}$

$sneezing(bob) \leftarrow flu(bob).$
$null \leftarrow hay\_fever(bob).$
$flu(bob).$
$hay\_fever(bob).$
$\mathbf{P(w_3) = 0.7 \times 0.2}$

$null \leftarrow flu(bob).$
$null \leftarrow hay\_fever(bob).$
$flu(bob).$
$hay\_fever(bob).$
$\mathbf{P(w_4) = 0.3 \times 0.2}$

- $sneezing(bob)$ is true in 3 worlds out of 4
- $P(sneezing(bob)) = 0.7 \times 0.8 + 0.3 \times 0.8 + 0.7 \times 0.2 = 0.94$

# Worlds for the ProbLog Program

$sneezing(X) \leftarrow flu(X), flu\_sneezing(X).$
$sneezing(X) \leftarrow hay\_fever(X), hay\_fever\_sneezing(X).$
$flu(bob).$
$hay\_fever(bob).$

$flu\_sneezing(bob).$
$hay\_fever\_sneezing(bob).$
$P(w_1) = 0.7 \times 0.8$

$hay\_fever\_sneezing(bob).$
$P(w_2) = 0.3 \times 0.8$

$flu\_sneezing(bob).$
$P(w_3) = 0.7 \times 0.2$

$P(w_4) = 0.3 \times 0.2$

- $sneezing(bob)$ is true in 3 worlds out of 4
- $P(sneezing(bob)) = 0.7 \times 0.8 + 0.3 \times 0.8 + 0.7 \times 0.2 = 0.94$

# Reasoning Tasks

- **Inference**: we want to compute the probability of a query given the model and, possibly, some evidence
- **Weight learning**: we know the structural part of the model (the logic formulas) but not the numeric part (the weights) and we want to infer the weights from data
- **Structure learning**: we want to infer both the structure and the weights of the model from data

# Inference under the Distribution Semantics

- Let **At** be the set of all ground (probabilistic and derived) atoms in a given LPAD program
- Assume that we are given a set $\mathbf{E} \subset \mathbf{At}$ of observed atoms (*evidence* atoms)
  - a vector $\mathbf{e}$ with their observed truth values means that the evidence is $\mathbf{E} = \mathbf{e}$
- We are also given a set $\mathbf{Q} \subset \mathbf{At}$ of atoms of interest (*query* atoms)
- $q$ indicates a single ground atom of interest
- Different inference tasks are possible

# Inference under the Distribution Semantics

- Unconditional inference: computing the unconditional probability $P(\mathbf{e})$, the probability of evidence
- Conditional or marginal inference: computing the conditional probability distribution of every query atom given the evidence, i.e., computing $P(q|\mathbf{E} = \mathbf{e})$, for each $q \in \mathbf{Q}$
- Computing the probability distribution or density of the non-ground arguments of a conjunction of literals $\mathbf{Q} = \mathbf{q}$

# Inference under the Distribution Semantics

- MPE task, or *most probable explanation*: finds the most likely joint truth value of all query (unobserved) atoms given the evidence, i.e., solves the optimization problem $argmax_{\mathbf{q}} P(\mathbf{Q} = \mathbf{q} | \mathbf{E} = \mathbf{e})$
- MAP task, or *maximum a posteriori*: finds the most likely joint truth value of a set of query (unobserved) atoms given the evidence, i.e., solves the optimization problem $argmax_{\mathbf{q}} P(\mathbf{Q} = \mathbf{q} | \mathbf{E} = \mathbf{e})$

# Inference under the Distribution Semantics

## Example

```
heads(Coin):1/2; tails(Coin):1/2:- toss(Coin),\+biased(Coin).
heads(Coin):0.6; tails(Coin):0.4:- toss(Coin),biased(Coin).
fair(Coin):0.9; biased(Coin):0.1.
toss(coin).
```

- Unconditional inference: $P(heads(coin))$, $P(tails(coin))$
- Conditional inference: $P(heads(coin)|biased(coin))$

# Inference under the Distribution Semantics

## Example

- **MPE**: Most likely world where *heads*(*coin*) is true?
- $2 \times 2 \times 2 = 8$ possible worlds
- H=heads(coin), T=tails(coin), F=fair(coin), B=biased(coin)
- $HHF : 0.5 * 0.6 * 0.9 = \mathbf{0.27}$
  $HHB : 0.5 * 0.6 * 0.1 = 0.03$
  $HTF : 0.5 * 0.4 * 0.9 = 0.18$
  $HTB : 0.5 * 0.4 * 0.1 = 0.02$
  $THF : 0.5 * 0.6 * 0.9 = \mathbf{0.27}$
  $THB : 0.5 * 0.6 * 0.1 = 0.03$
  $TTF : 0.5 * 0.4 * 0.9 = 0.18$
  $TTB : 0.5 * 0.4 * 0.1 = 0.02$
- *heads*(*coin*), *heads*(*coin*), *fair*(*coin*)
- *tails*(*coin*), *heads*(*coin*), *fair*(*coin*)

# Inference under the Distribution Semantics

The inference problem is #**P-hard** and for large models is intractable. Possible **solutions**:

1. Exact inference
   - **Knowledge Compilation**: compile the probabilistic logic program to an *intermediate representation* and then compute the probability by Weighted Model Counting
   - **Bayesian Network (BN) based**: convert the probabilistic logic program into a BN and apply BN inference algorithms (Meert et al. (2010))
   - **Lifted** inference (Poole (2003)): exploits symmetries in the model to speed up inference
2. Approximate inference

# Knowledge Compilation

A probabilistic logic program can be compiled into the following intermediate representations:

- **Binary Decision Diagrams** (BDD): De Raedt et al. (2007), `cplint` (Riguzzi (2007),Riguzzi (2009)), `PITA` (Riguzzi and Swift (2010a))
- deterministic-Decomposable Negation Normal Form circuits **(d-DNNF)**: `ProbLog2`, Fierens et al. (2015)
- **Sentential Decision Diagrams** (SDD), Darwiche (2011)

# Knowledge Compilation to BDD

1. **Assign Boolean random variables** (r.v.) to the probabilistic rules
2. Given a query $Q$, compute its **explanations**, i.e. assignments to the random variables that are sufficient for entailing the query
3. Let $K$ be the set of all possible explanations
4. Build a Boolean formula $f_K$ representing $K$
5. **Build a BDD** encoding $f_K$
6. **Compute the probability of evidence** from the BDD (unconditional inference)

# Knowledge Compilation to BDD

## (2) Computing explanations for $Q$

- **Atomic choice** $(C, \theta, k)$: selection of the $k$-th atom for a grounding $C\theta$ of a switch/clause $C$ and a substitution $\theta$: $(C_1, \{X/bob\}, 1)$ selects $sneezing(bob)$ in the LPAD
- Composite choice $\kappa$: consistent set of atomic choices $(C_i, \theta_j, k)$
- A composite choice $\kappa$ identifies a set of worlds $\omega_\kappa = \{w_\sigma | \kappa \subseteq \sigma\}$ whose selections $\sigma$ are a superset of $\kappa$
- A composite choice $\kappa$ is an explanation for a query $Q$ if $Q$ is true in every world of $\omega_\kappa$
- In practice, a *covering* set of explanations must be found
- A set of composite choices $K$ is covering with respect to $Q$ if every world $w$ in which $Q$ is true is such that $w \in \omega_K$ where $\omega_K = \bigcup_{\kappa \in K} \omega_\kappa$

# Knowledge Compilation to BDD

## (2) Computing explanations for $Q$

- Two composite choices $\kappa_1$ and $\kappa_1$ are exclusive if their union is inconsistent
- If $\kappa_1 = \{(C_1, X/bob, 1)\}, \kappa_2 = \{(C_1, X/bob, 2), (C_2, X/bob, 1)\}$ then $\kappa_1 \cup \kappa_2$ is inconsistent
- A set $K$ of composite choices is mutually exclusive if for all $\kappa_1 \in K, \kappa_2 \in K, \kappa_1 \neq \kappa_2 \Rightarrow \kappa_1$ and $\kappa_2$ are exclusive
- if $K$ is mutually exclusive, $P(K) = \sum_{\kappa \in K} P(\kappa)$

# Knowledge Compilation to BDD

## (3,4) Building a formula for the explanations

- $K$ = set of covering explanations found for $Q$
- Boolean formula representing $K$, over a set $\mathbf{X}$ of multi-valued r. v.

$$f_K(\mathbf{X}) = \bigvee_{\kappa \in K} \bigwedge_{(C_i,\theta_j,k)\in\kappa} (X_{ij} = k)$$

- $X_{ij}$: random variable whose domain is $1, ..., n_i$ (no of head atoms)
- $P(X_{ij} = k) = p(C_i, k)$
- $f_K(\mathbf{X}) = 1$ if the values of the variables correspond to an explanation for $Q$
- Equations for a single explanation are conjoined and the conjunctions for the different explanations are disjoined

# Knowledge Compilation to BDD

## (2,3,4) Example

- For the (LPAD) program:

$$sneezing(X) : 0.7 \leftarrow flu(X).$$
$$sneezing(X) : 0.8 \leftarrow hay\_fever(X).$$
$$flu(bob).$$
$$hay\_fever(bob).$$

  a set of covering explanations for $Q = sneezing(bob)$ is $K = \{\kappa_1, \kappa_2\}$

- $\kappa_1 = \{(C_1, \{X/bob\}, 1)\} \quad \kappa_2 = \{(C_2, \{X/bob\}, 1)\}$

# Knowledge Compilation to BDD

## (2,3,4) Example

- $\theta_1 = X/bob$
- $X_{C_1\theta_1} \rightarrow X_{11}$ From $\kappa_1, X_{11} = 1$
- $X_{C_2\theta_1} \rightarrow X_{21}$ From $\kappa_2, X_{21} = 1$
- $f_K(\mathbf{X}) = (X_{11} = 1) \vee (X_{21} = 1)$
- $P(Q) = P(f_K(\mathbf{X})) = P(X_{11} \vee X_{21}) = P(X_{11}) + P(X_{21}) - P(X_{11})P(X_{21})$

# Knowledge Compilation to BDD

## And Now What?

- Worlds are mutually exclusive, and in theory we could compute $P(Q)$ as $\sum_{w \in W_T : w \models Q} P(w)$,
  - BUT in practice, it is unfeasible to find all the worlds $w$ where the query is true
- It's easier to find explanations for the query
  - BUT explanations might not be mutually exclusive
  - **they must be made mutually exclusive in order to sum up probabilities**
  - Binary Decision Diagrams (BDD): they split paths on the basis of the values of binary variables, so **the branches are mutually disjoint**

# Knowledge Compilation to BDD

## (3,4) Building a formula for the explanations

- A BDD performs a *Shannon expansion* of $f_K(\mathbf{X})$
- $f_K(\mathbf{X}) = X_{11} \times f_K^{X_{11}}(\mathbf{X}) + \overline{X_{11}} \times f_K^{\overline{X_{11}}}(\mathbf{X})$
- The two disjuncts above are mutually exclusive
- $f_K^X(\mathbf{X})(f_K^{\overline{X}}(\mathbf{X}))$ is the formula obtained by $f_K(\mathbf{X})$ by setting $X$ to 1 (0)
- $P(f_K(\mathbf{X})) = P(X_{11})P(f_K^{X_{11}}(\mathbf{X})) + (1 - P(X_{11}))P(f_K^{\overline{X_{11}}}(\mathbf{X}))$
- $P(f_K(\mathbf{X})) = 0.7 \cdot P(f_K^{X_{11}}(\mathbf{X})) + 0.3 \cdot P(f_K^{\overline{X_{11}}}(\mathbf{X}))$

# Knowledge Compilation to BDD

## (3,4) Building a formula for the explanations

- In order to use BDD, a multi-valued random variable $X_{ij}$ with $n_i$ values must be converted into $n_i - 1$ Boolean variables $X_{ij1}, ..., X_{ijn_i-1}$
- $X_{ij} = k$ for $k = 1, ..., n_i - 1$ is represented by the conjunction $\overline{X_{ij1}} \wedge \ldots \wedge \overline{X_{ijk-1}} \wedge X_{ijk}$
- $X_{ij} = n_i$ by $\overline{X_{ij1}} \wedge \ldots \wedge \overline{X_{ijn_i-1}}$

# Knowledge Compilation to BDD

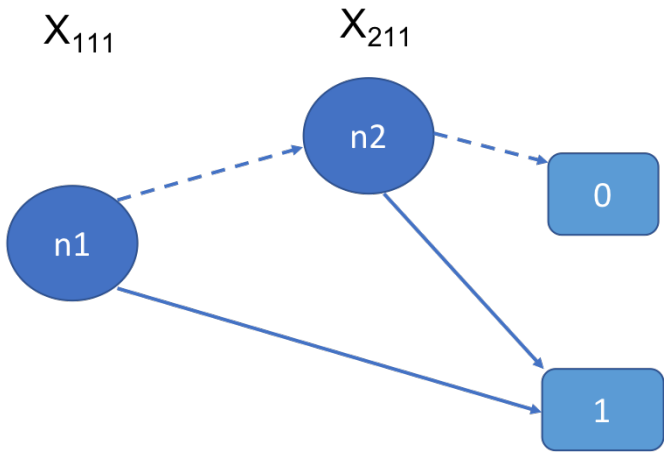
## (3,4) Example

- For both $C_1$ and $C_2$ : $n_i = 2$
- (multi-valued) $X_{11} = 1 \rightarrow$ (Boolean) $X_{111}$
- (multi-valued) $X_{11} = 2 \rightarrow$ (Boolean) $\overline{X_{111}}$
- (multi-valued) $X_{21} = 1 \rightarrow$ (Boolean) $X_{211}$
- (multi-valued) $X_{21} = 2 \rightarrow$ (Boolean) $\overline{X_{211}}$
- $f'_K(\mathbf{X}) = X_{111} \vee X_{211}$

# Knowledge Compilation to BDD

## (5) Building a BDD

- A BDD for a function of Boolean variables is a rooted graph that has one level for each Boolean variable
- A node $n$ has two children: one corresponding to the 1 value of the variable associated with $n$ (solid arc) and one corresponding the 0 value of the variable (dashed arc)
- The leaves store either 0 or 1: a path to a 1-leaf corresponds to an explanation for $Q$
- BDD for $f'_K(\mathbf{X}) = X_{111} \vee X_{211}$:



# Knowledge Compilation to BDD

## (5) Building a BDD

- BDDs can be built by combining simpler BDDs using Boolean operators
- While building BDDs, simplification operations can be applied that delete or merge nodes
- Merging is performed when the diagram contains two identical sub-diagrams
- Deletion is performed when both arcs from a node point to the same node
- A reduced BDD often has a much smaller number of nodes with respect to the original BDD

# Knowledge Compilation to BDD

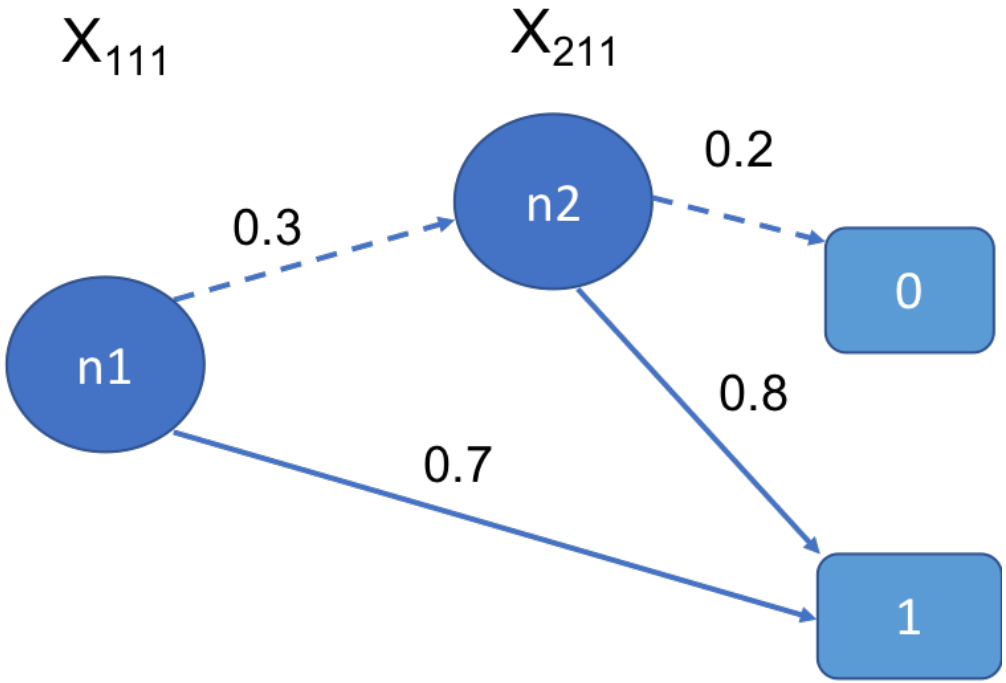
## (6) Computing the probability from a BDD

**Algorithm 4** Function PROB: Computation of the probability of a BDD.

```
1:  function PROB(node)
2:      if node is a terminal then
3:          return 1
4:      else
5:          if Table(node) ≠ null then
6:              return Table(node)
7:          else
8:              p0 ←PROB(child_0(node))
9:              p1 ←PROB(child_1(node))
10:             let π be the probability of being true of var(node)
11:             Res ← p1 · π + p0 · (1 − π)
12:             add node → Res to Table
13:             return Res
14:         end if
15:     end if
16: end function
```

# Knowledge Compilation to BDD

## (6) Example




$$P(n_2) = 0 \cdot 0.2 + 1 \cdot 0.8 = 0.8$$
$$P(n_1) = P(root) = 1 \cdot 0.7 + 0.3 \cdot 0.8 = 0.94 = P(Q) = P(sneezing(bob))$$

# MDDs vs BDDs

- A Multivalued Decision Diagrams (MDD) represents a function taking Boolean values on a set of **multivalued variables X**, each node is associated to the variable of its level and has one child for each possible value of the variable
- A BDD represents a function taking Boolean values on a set of **binary variables X**, each node is associated to the variable of its level and has 2 children
- Most packages for the manipulation of decision diagrams are restricted to work on Binary Decision Diagrams (see CUDD @`http://vlsicad.eecs.umich.edu/BK/Slots/cache/vlsi.colorado.edu/~fabio/`), as they offer several operators for handling BDDs

# Tabling-based Inference

- Idea: **maintain in a *table* both subgoals encountered in a query evaluation and answers to these subgoals**
- If a subgoal is encountered more than once, **the evaluation reuses information from the table** rather than re-performing resolution against program clauses
- Tabling can be used to evaluate programs with negation
- Tabling integrates closely with Prolog: a predicate p/n is evaluated using SLDNF by default; the predicate can be made to use tabling by the directive `:- table p/n` that is added by the user or compiler

# Tabling-based Inference

## PITA: *Probabilistic Inference with Tabling and Answer subsumption*

- Introduced by Riguzzi and Swift (2010b)
- PITA computes the probability of queries from LPADs with tabling
- PITA builds all explanations for every subgoal encountered during a derivation of the query
- Explanations for subgoals are stored with tabling
- Explanations are compactly represented using BDDs that also allow an efficient computation of the probability
- Subgoals (ground atoms) have an extra argument storing a BDD that represents the explanations for their answers
- When an answer $q(\mathbf{x}, bdd)$ is found, $bdd$ represents the explanations for $q(\mathbf{x})$

# Tabling-based Inference

## Answer Subsumption

- A feature of tabling in XSB and SWI Prolog
- **Combine different answers for the same goal**
- E.g `:-table path(X,Y,lattice(or/3))` means that, if two explanations `path(a,b,bdd0)` and `path(a,b,bdd1)` are found, the single answer `path(a,b,bdd)` will be stored in the table where `or(bdd0,bdd1,bdd)`

# Tabling-based Inference

## PITA: Program Transformation

- The first step of the PITA algorithm is to apply a program transformation to an LPAD to create a normal logic program that contains calls for manipulating BDDs
- Prolog interface to the CUDD C library with the predicates:
  - *init*, *end*: allocation and deallocation of a BDD manager
  - $zero(-BDD)$, $one(-BDD)$, $and(+BDD1, +BDD2, -BDDO)$, $or(+BDD1, +BDD2, -BDDO)$, $not(+BDDI, -BDDO)$: BDD operations
  - $add\_var(+N\_Val, +Probs, -Var)$: adds a new random variable *Var* associated to a new instantiation of a rule with $N\_Val$ head atoms and parameters list *Probs*
  - $equality(+Var, +Value, -BDD)$: BDD represents Var=Value
  - $ret\_prob(+BDD, -P)$: returns the probability of the formula encoded by BDD

# Tabling-based Inference

## PITA: Program Transformation

- $get\_var\_n(R, C, Probs, Var)$ wraps $add\_var/3$ and avoids adding a new variable when one already exists for an instantiation
- $R$ is an identifier for the LPAD rule, $C$ is a list of constants, one for each variables of the clause, and $Var$ is a integer that identifies the random variable associated with clause $R$ under a particular grounding; $Probs$ is a list of floats that stores the parameters in the head of rule $R$

$$
\begin{aligned}
get\_var\_n(R, C, Probs, Var) :- \quad & (var(R, C, Var) \rightarrow true; \\
& length(Probs, L), \\
& add\_var(L, Probs, Var), \\
& assert(var(R, C, Var))).
\end{aligned}
$$

# Tabling-based Inference

## PITA: Program Transformation

The disjunctive clause $C_r = H_1 : \alpha_1 \vee \ldots \vee H_n : \alpha_n \leftarrow L_1, \ldots, L_m.$
is transformed into the set of clauses $PITA(C_r)$

$$PITA(C_r, 1) = PITA(H_1) \leftarrow \begin{array}{l} one(BB_0), \\ PITA(L_1), and(BB_0, B_1, BB_1), \\ \ldots, \\ PITA(L_m), and(BB_{m-1}, B_m, BB_m), \\ get\_var\_n(r, C, [\alpha_1, \ldots, \alpha_n], Var), \\ equality(Var, 1, BB), and(BB_m, BB, BDD). \end{array}$$

$$\ldots$$

$$PITA(C_r, n) = PITA(H_n) \leftarrow \begin{array}{l} one(BB_0), \\ PITA(L_1), and(BB_0, B_1, BB_1), \\ \ldots, \\ PITA(L_m), and(BB_{m-1}, B_m, BB_m), \\ get\_var\_n(r, C, [\alpha_1, \ldots, \alpha_n], Var), \\ equality(Var, n, BB), and(BB_m, BB, BDD). \end{array}$$

# Tabling-based Inference

## PITA Example

```
:- table path(X,Y,lattice(or/3)),edge(X,Y,lattice(or/3)).
```

LPAD

```
path(X,X).
path(X,Y):- path(X,Z),edge(Z,Y).
edge(a,b):0.3.
....
```

PITA Transformation

```
path(X,X,One):- one(One).
path(X,Y,BDD):- one(One),path(X,Z,BDD0),and(One,BDD0,BDD1),
                edge(Z,Y,BDD2),and(BDD1,BDD2,BDD).
edge(a,b,BDD):- one(One),get_var_n(3,[],[0.3,0.7],Var),
                equality(Var,1,BDD0),and(BDD0,One,BDD).
...
```

# Tabling-based Inference

## PITA Example

Query: `path(a,b)`

```
:- init,
   path(?HGNC_620?,?HGNC_983?,BDD),
   ret_prob(BDD,P),
   end.
```

# Tabling-based Inference

## PITA Results

- PITA was compared with CVE (Meert et al. (2010)) and ProbLog1 (Kimmig et al. (2008a)).
  - CVE transforms an LPAD into an equivalent Bayesian network and then performs inference on the network using the variable elimination algorithm
  - ProbLog employs BDDs for efficient inference
- The algorithm was able to successfully solve more complex queries than the other algorithms in most cases and it was also almost always faster

# Inference Systems based on BDDs: `cplint`

- Suite of programs for reasoning with LPADs
- Inference and learning
- Versions for Yap Prolog and SWI-Prolog
- Distributed as a pack of SWI-Prolog. To install it, use
  `?- pack_install(cplint).`
- Available in the web application *cplint on SWISH*: http://cplint.eu/
- Exact Inference: module `PITA`

# Inference Systems based on BDDs: cplint

- *Input*: Prolog file where you must
  - load the inference module (pita)
  - initialize it with a directive (`:- pita.`)
  - enclose the LPAD clauses in `:-begin_lpad.` and `:-end_lpad.`
- *Example*: coin.pl

```
:- use_module(library(pita)).
:- pita.
:- begin_lpad.
 heads(Coin):1/2; tails(Coin):1/2:- toss(Coin),\+biased(Coin).
 heads(Coin):0.6; tails(Coin):0.4:- toss(Coin),biased(Coin).
 fair(Coin):0.9; biased(Coin):0.1.
 toss(coin).
:- end_lpad.
```

# Inference Systems based on BDDs: cplint

- You can have also (non-probabilistic) clauses outside `:-begin/end_lpad.`, considered as database clauses.
- Subgoals in the body of probabilistic clauses can query them by enclosing the query in `db/1`.
- *Example*

```
:- use_module(library(pita)).
:- pita.
:- begin_lpad.
    sampled_male(X):0.5:- db(male(X)).
:- end_lpad.
male(john).
male(david).
```

# Hands-on: Coin example

- http://cplint.eu/example/inference/coin.pl
- Unconditional inference:
  - What is the probability that *coin* lands heads? `prob(heads(coin),Prob).`
  - What is the probability that *coin* lands tails? `prob(tails(coin),Prob).`
- Conditional inference:
  - What is the probability that *coin* lands heads, given that I know it is biased? `prob(heads(coin),biased(coin),Prob).`

# Hands-on: Sneezing example

- http://cplint.eu/p/sneezing_simple.pl
- Unconditional inference:
  - What's the probability of Bob sneezing?
    `prob(sneezing(bob),Prob).`
- We should expect `Prob` = ?

# Knowledge Compilation to d-DNNF

- A tractable logical form known as Deterministic, Decomposable Negation Normal Form, which **permits some generally intractable logical queries to be computed in time polynomial** in the form size (Darwiche (2004))
- Superset of OBDDs (Ordered BDD): BDDs with a defined variable ordering
- Allows to perform unconditional, conditional and MPE inference in **ProbLog2**

# Knowledge Compilation to d-DNNF

- A negation normal form (NNF) is a rooted directed acyclic graph in which each leaf node is labeled with a literal, *true* or *false*, and each internal node is labeled with a conjunction or disjunction

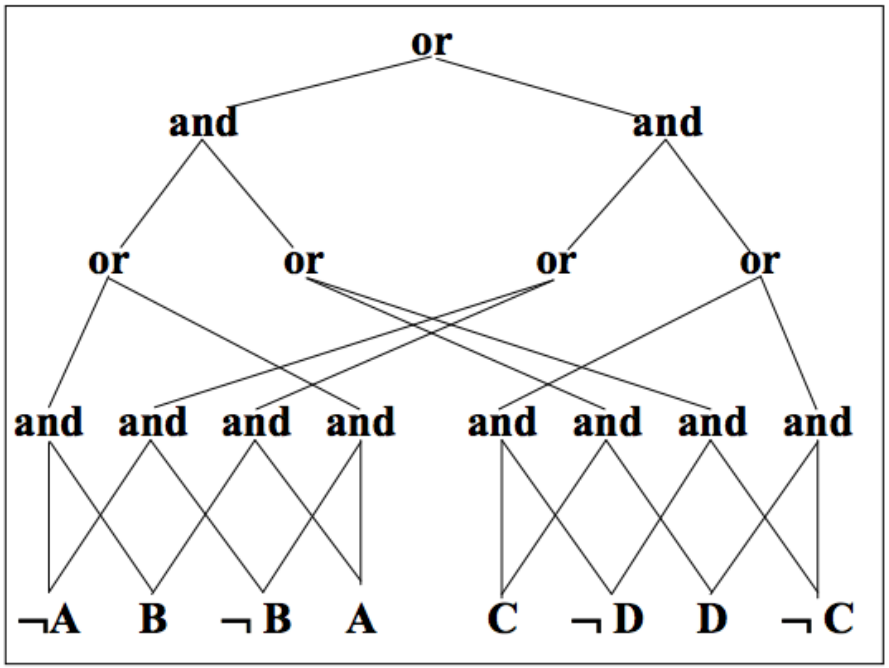



**Figure 1.** A negation normal form.

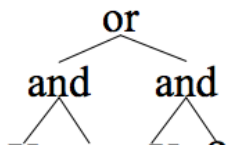
# Knowledge Compilation to d-DNNF

- For any node $n$, $Vars(n)$ denotes all propositional variables that appear in the subgraph rooted at $n$, and $\Delta(n)$ denotes the formula represented by $n$ and its descendants.

## Properties

- **Decomposability**: $Vars(n_i) \cap Vars(n_j) = \emptyset$ for any two children $n_i$ and $n_j$ of an and-node $n$. The NNF in the figure is decomposable.
- **Determinism**: $\Delta(n_i) \wedge \Delta(n_j)$ is logically inconsistent for any two children $n_i$ and $n_j$ of an or-node $n$. The NNF in the figure is deterministic.

# Knowledge Compilation to d-DNNF

- **Decision**: holds when the root node of the NNF graph is a *decision node*. A decision node is a node labeled with true, false, or is an or-node having the form 

where $X$ is a variable, $\alpha$ and $\beta$ are decision nodes. The NNF in the figure does not satisfy the decision property since its root is not a decision node.
- **Ordering**: only for NNFs that satisfy the decision property. Ordering holds when decision variables appear in the same order along any path from the root to any leaf.

# Knowledge Compilation to d-DNNF

## Procedure

1. Ground the probabilistic logic program and convert it into an equivalent Boolean formula $\phi_r$
2. Build $\phi_e$, the Boolean formula for the evidence
3. Rewrite $\phi = \phi_r \wedge \phi_e$ in CNF
4. Construct a *weighted* Boolean formula for $\phi$
5. CNF formula $\rightarrow$ d-DNNF formula (#P hard step)
6. d-DNNF formula $\rightarrow$ aritmetic circuit (AC)
7. Compute the probability of evidence from the AC

# Knowledge Compilation to d-DNNF

## (1) Ground the PL program and convert it into a Boolean formula $\phi_r$

- **ProbLog program**

$$
\begin{aligned}
&0.1 :: burglary.\\
&0.2 :: earthquake.\\
&0.7 :: hears\_alarm(X) \leftarrow person(X).\\
&alarm \leftarrow burglary.\\
&alarm \leftarrow earthquake.\\
&calls(X) \leftarrow alarm, hears\_alarm(X).\\
&person(mary).\\
&person(john).
\end{aligned}
$$

# Knowledge Compilation to d-DNNF

## (1) Ground the PL program and convert it into a Boolean formula $\phi_r$

- Suppose $e = calls(john)$
- **Relevant Ground Program (RGP)** (taking into account $e$ (and eventually $q$), in order to consider only the part of the program that is relevant to the (query given the) evidence):

$$0.1 :: burglary.$$
$$0.2 :: earthquake.$$
$$0.7 :: hears\_alarm(john).$$
$$alarm \leftarrow burglary.$$
$$alarm \leftarrow earthquake.$$
$$calls(john) \leftarrow alarm, hears\_alarm(john).$$

# Knowledge Compilation to d-DNNF

## (1-2) Ground the PL program and convert it into a Boolean formula $\phi_r$. Build $\phi_e$

- Apply specific transformation rules (Lloyd (1987); Janhunen (2004); Mantadelis and Janssens (2010)) to generate a Boolean formula $\phi_r$
- $\phi_e = calls(john)$
- In the ProbLog example, $\phi$ is the conjunction of the following three sub-formulas (the first two for $\phi_r$, the last one for $\phi_e$)

$$alarm \leftrightarrow burglary \vee earthquake$$
$$calls(john) \leftrightarrow alarm \wedge hears\_alarm(john)$$
$$calls(john)$$

# Knowledge Compilation to d-DNNF

## (3) Rewrite $\phi$ in CNF

- CNF:

$$l_{11} \vee \ldots \vee l_{1m_1} \wedge \ldots \wedge l_{n1} \vee \ldots \vee l_{nm_n}$$

  where each $l_{ij}$ is a literal
- $alarm \leftrightarrow burglary \vee earthquake$ becomes $(alarm \vee \neg burglary) \wedge (alarm \vee \neg earthquake) \wedge (\neg alarm \vee burglary \vee earthquake)$
- $calls(john) \leftrightarrow alarm \wedge hears\_alarm(john)$ becomes $(\neg calls(john) \vee alarm) \wedge (\neg calls(john) \vee hears\_alarm(john)) \wedge (\neg alarm \vee \neg hears\_alarm(john) \vee calls(john))$
- $\phi$ in CNF: $(alarm \vee \neg burglary) \wedge (alarm \vee \neg earthquake) \wedge (\neg alarm \vee burglary \vee earthquake) \wedge (\neg calls(john) \vee alarm) \wedge (\neg calls(john) \vee hears\_alarm(john)) \wedge (\neg alarm \vee \neg hears\_alarm(john) \vee calls(john)) \wedge calls(john)$

# Knowledge Compilation to d-DNNF

## (4) Construct a weighted Boolean formula

- Define the weight function for all literals in $\phi$
- The weight of a probabilistic literal is derived from the probabilistic facts in the program
- If the RGP contains a probabilistic fact $p :: f$, then we assign weight $p$ to $f$ and weight $1 - p$ to $\neg f$
- For literals not occurring in a probabilistic fact (derived literals), the weight is 1

# Knowledge Compilation to d-DNNF

## (4) Construct a weighted Boolean formula

$burglary \mapsto 0.1$ $\quad$ $\neg burglary \mapsto 0.9$

$earthquake \mapsto 0.2$ $\quad$ $\neg earthquake \mapsto 0.8$

$hears\_alarm(john) \mapsto 0.7$ $\quad$ $\neg hears\_alarm(john) \mapsto 0.3$

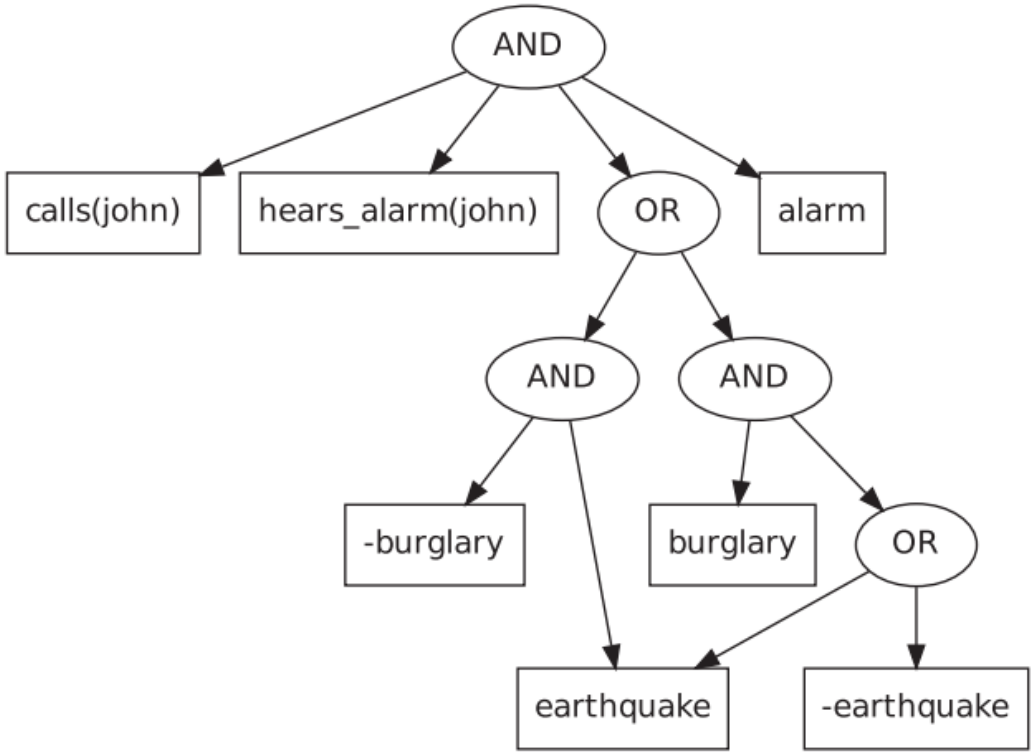
$alarm \mapsto 1$ $\quad$ $\neg alarm \mapsto 1$

$calls(john) \mapsto 1$ $\quad$ $\neg calls(john) \mapsto 1$

# Knowledge Compilation to d-DNNF

## (5) Convert the CNF into a d-DNNF

- Conversion is made by compilers: c2d (Darwiche (2004)), DSHARP (Muise et al. (2012)), irrespective of the weighting function



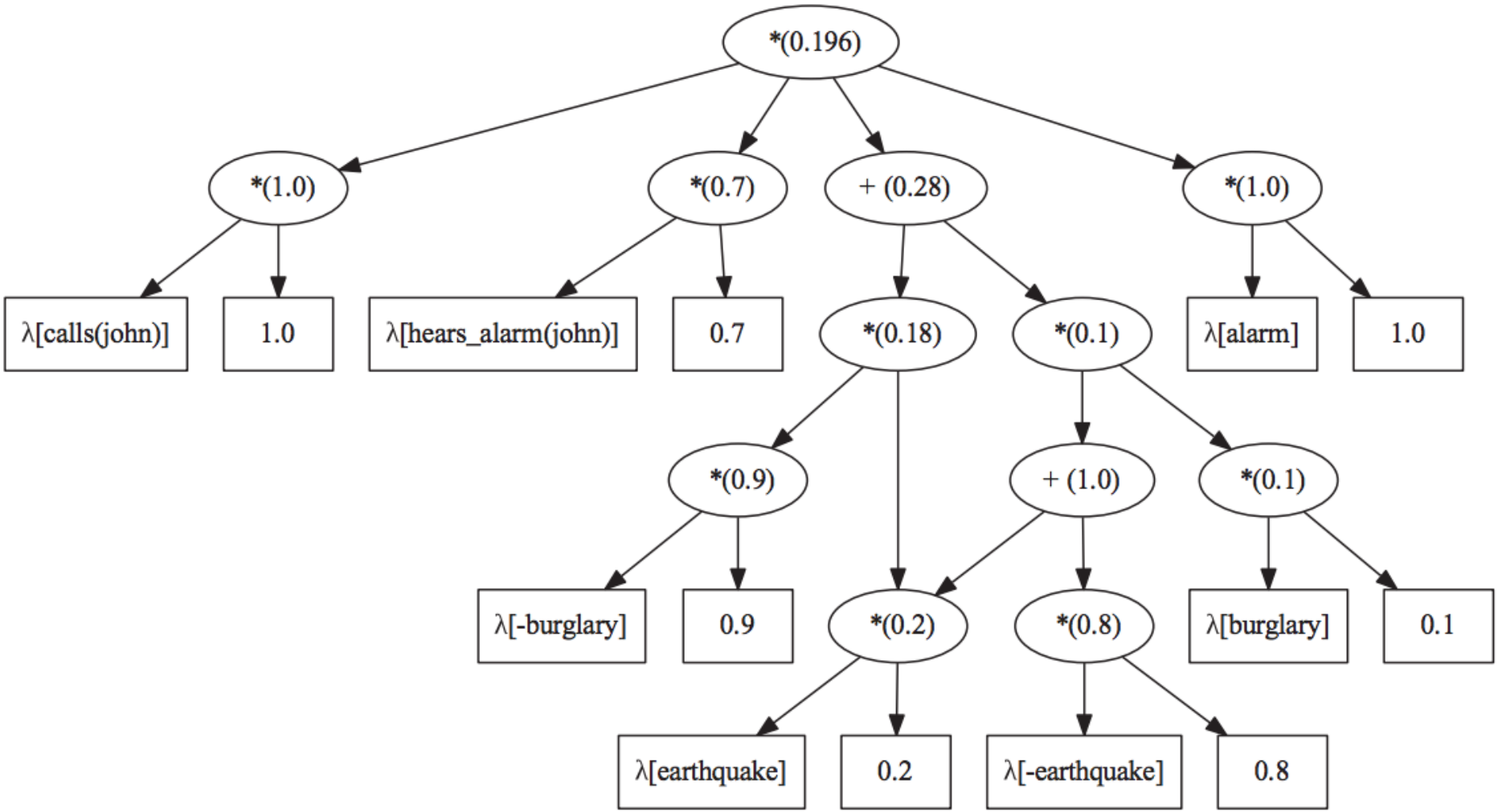
# Knowledge Compilation to d-DNNF

## (6) Convert the d-DNNF formula into an aritmetic circuit

1. Replace all conjunctions in the internal nodes of the d-DNNF by multiplications, and all disjunctions by summations
2. Replace every leaf node involving a literal $l$ by a subtree consisting of a multiplication node having two children, namely, a leaf node with an indicator variable for the literal $l$ and a leaf node with the weight of $l$ according to the weighted formula
3. Numbers in parentheses represent results of the intermediate computations

# Knowledge Compilation to d-DNNF

## (6) Convert the d-DNNF formula into an aritmetic circuit



# Knowledge Compilation to d-DNNF

## (7) Compute the probability from the AC

- Evaluate the circuit bottom-up after having assigned the value 1 to all the indicator variables
- $P(root) = P(\mathbf{e}) = P(calls(john)) = 0.196$
- $P(\mathbf{e})$ is called weighted model count (WMC)
- **Indicator variables** allow us to add **further evidence**, provided that it extends the initial evidence for which the circuit has been built
- To compute $P(\mathbf{e}, l_1 \dots l_n)$ for any conjunction of literals $l_1, \dots, l_n$ it is enough to set the indicator variables as $\lambda(l_i) = 1$, $\lambda(\neg l_i) = 0$ and $\lambda(l) = 1$ for the other literals $l$, and evaluate the circuit
  - Ex.: $P(calls(john) = true \wedge earthquake = true)$

# Knowledge Compilation to d-DNNF

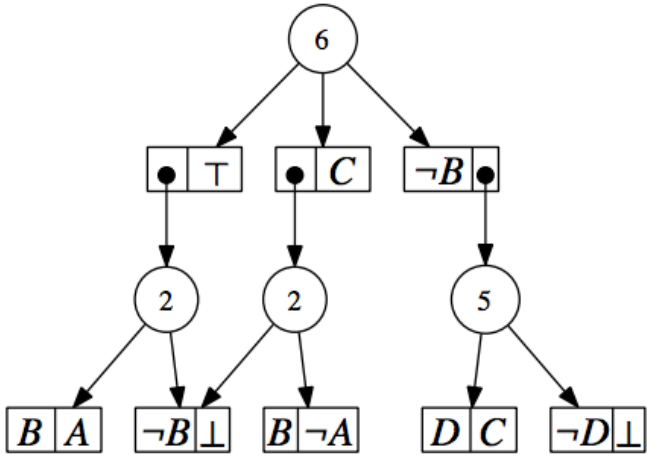
## BDD vs d-DNNF

- A BDD is a special kind of d-DNNF, namely, one that satisfies the additional properties of ordering and decision
- In the approach seen earlier, we can replace a d-DNNF compiler with a BDD compiler
- Computing the probability of evidence can then be done by either operating directly on the BDD, or by converting the BDD to an arithmetic circuit and evaluating the circuit
- d-DNNFs outperform BDDs (Darwiche (2004))

# Knowledge Compilation by SDD

- An SDD (Vlasselaer et al. (2014); Darwiche (2011)) contains two types of nodes
- *Decision nodes* (circles): **disjunctions** over mutually exclusive sentences
- *Elements* (paired boxes [p|s]): **conjunctions** between the two children
  - p: "prime"; s: "sub"
  - Elements are decision nodes' children and each box in an element can contain a pointer to a decision node or a terminal node, either a literal or the constants 0 or 1
- A decision node with children $[p_1|s_1], \ldots, [p_n|s_n]$ represents the function $(p_1 \wedge s_1) \vee \ldots \vee (p_n \wedge s_n)$
- Primes must form a partition: $p_i \neq 0$ (primes are consistent), $p_i \wedge p_j = 0$ for $i \neq j$ (every pair of distinct primes are mutually exclusive), and $p_1 \vee \ldots \vee p_n = 1$ (the disjunction of all primes is valid)

# Knowledge Compilation by SDD




(b) Graphical depiction of an SDD

$$f = (A \wedge B) \vee (B \wedge C) \vee (C \wedge D).$$

The top level decomposition has three elements, with primes representing $A \wedge B, \neg A \wedge B, \neg B$ and corresponding subs representing *true*, $C$, and $C \wedge D$.

# Knowledge Compilation by SDD

## Procedure

1. Ground the probabilistic logic program and convert it into an equivalent propositional formula
2. **Compile directly the formula into an SDD** OR convert it into a CNF Boolean formula to be compiled into an SDD
3. Compute the probability of evidence from the SDD

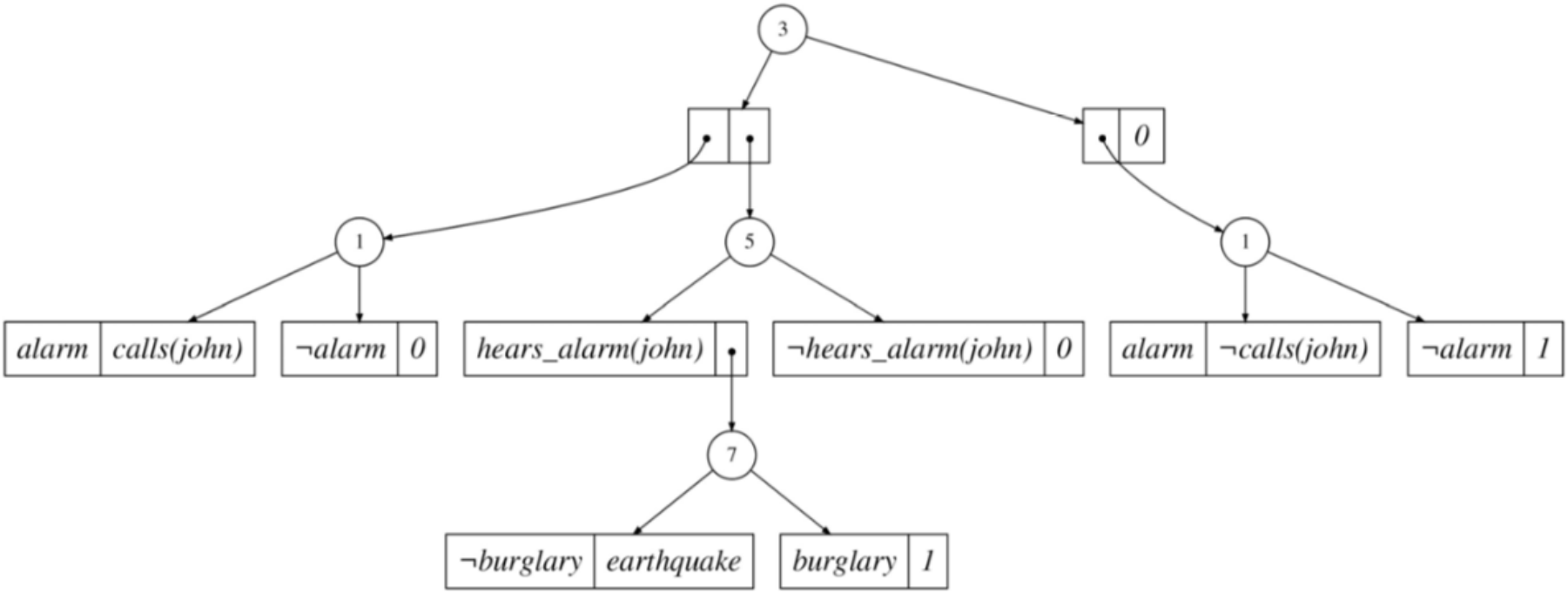
# Knowledge Compilation by SDD

## (1) Ground the probabilistic logic program and convert it into an equivalent propositional formula

Already done

# Knowledge Compilation by SDD

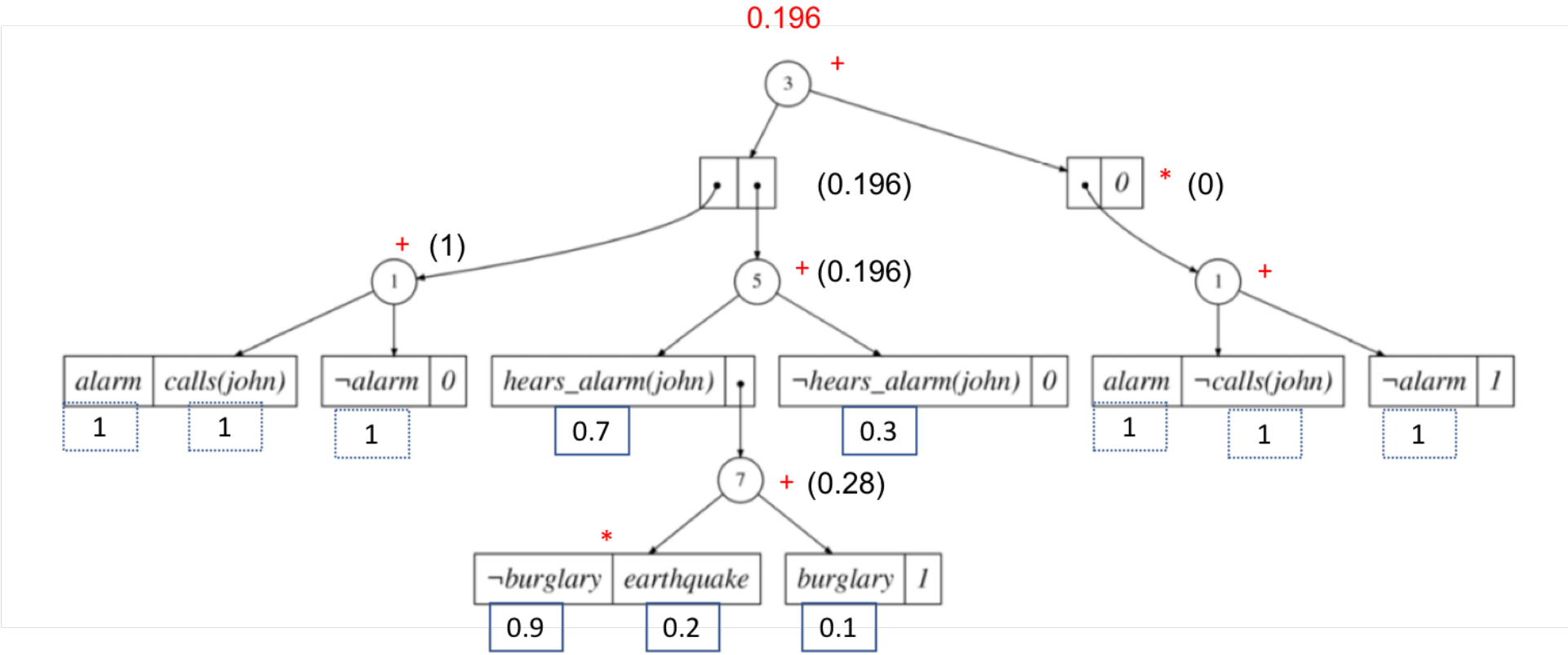
## (2) Compile directly the formula into an SDD




- The compilation is made by a compiler (`http://reasoning.cs.ucla.edu/`)

# Knowledge Compilation by SDD

## (3) Compute the probability of evidence from the SDD




- $P(root) = P(\mathbf{e}) = P(calls(john))$

# Knowledge Compilation by SDD

## Comparison of languages

- Succinctness: size of the smallest compiled circuit for every Boolean formula
  - $d - DNNF < SDD \leq OBDD$
  - There exists a Boolean formula whose smallest SDD representation is exponentially larger than its smallest d-DNNF representation, but the smallest OBDD for any formula is at least as big as its smallest SDD
- SDDs are special cases of d-DNNFs: if one replaces circle-nodes with or-nodes, and paired-boxes with and-nodes, one obtains a d-DNNF, with additional properties (structured decomposability and strong determinism)
- SDDs outperform d-DNNFs

# Knowledge Compilation by SDD

## Comparison of languages

- **Strongly deterministic decomposition**: An $(\mathbf{X}, \mathbf{Y})$-*decomposition* of a function $f(\mathbf{X}, \mathbf{Y})$ over non-overlapping variables $\mathbf{X}$ and $\mathbf{Y}$ is a set $\{(p_1, s_1), \ldots, (p_n, s_n)\}$ such that

$$f = p_1(\mathbf{X}) \wedge s_1(\mathbf{Y}) \vee \ldots \vee p_n(\mathbf{X}) \wedge s_n(\mathbf{Y})$$

  If $p_i \wedge p_j = 0$ for $i \neq j$, the decomposition is said to be *strongly deterministic*

- BDDs are a special case of SDDs where decompositions are all Shannon: formula $f$ is decomposed into $\{(X, f^X), (\neg X, f^{\neg X})\}$.
- SDDs generalize BDDs by considering non-binary decisions based on the value of primes

# Approximate Inference

- Approximate inference aims at computing the results of inference (probability of evidence $P(\mathbf{e})$) in an approximate way so that the process is cheaper than the exact computation of the results
- Two approaches: those that modify an exact inference algorithm and those based on sampling

- Iterative deepening
- *k-best*
- Monte Carlo

# Approximate Inference

## Iterative deepening

- De Raedt et al. (2007); Kimmig et al. (2008b)
- Input: an error bound $\epsilon$, a depth bound $d$, and a query $q$
- Construct an SLD tree for $q$ up to depth $d$
- Build two sets of explanations
  - $K_l$: set of composite choices corresponding to the successful proofs present in the tree
  - $K_u$: set of composite choices corresponding to the successful and still open proofs present in the tree
- $P(K_l)$: lower bound / $P(K_u)$: upper bound of the exact probability

# Approximate Inference

## Iterative deepening

- If the difference $P(K_u) - P(K_l)$ is smaller than $\epsilon$, this means that a solution with a satisfying accuracy has been found and the interval $[P(K_l), P(K_u)]$ is returned
- Otherwise, the depth bound is increased and a new SLD tree is built up to the new depth bound
- Stops when the difference $\leq \epsilon$

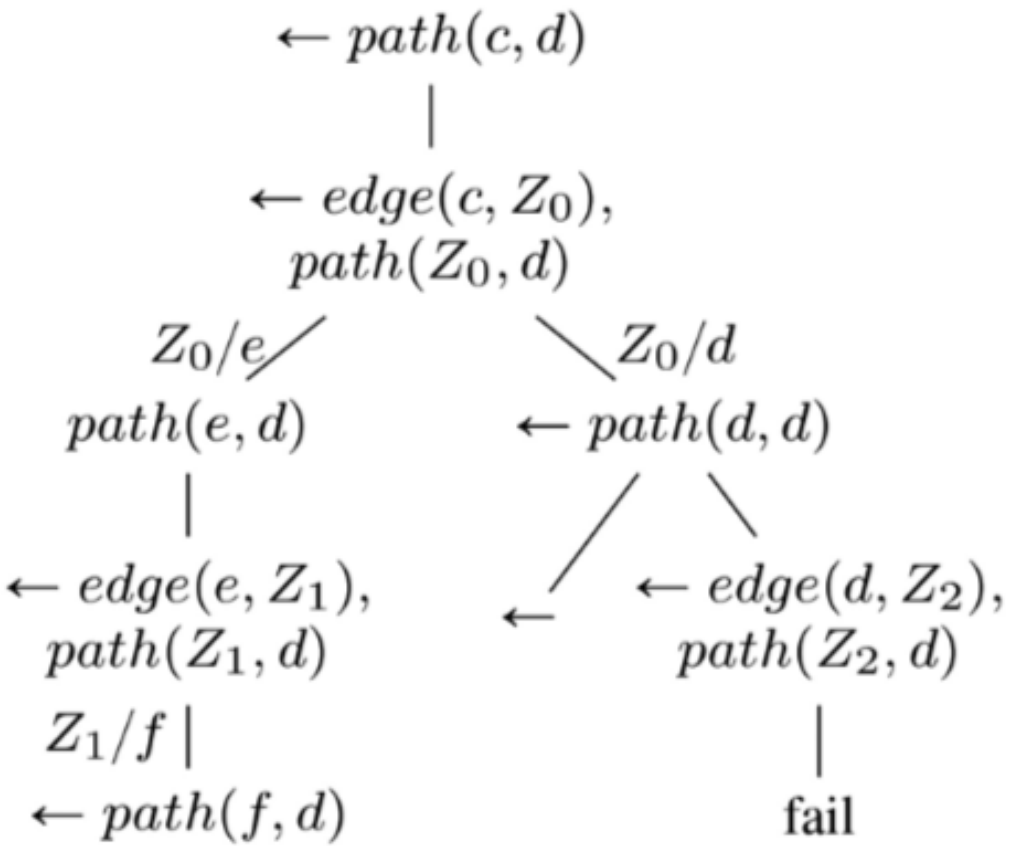
# Approximate Inference

## Iterative deepening

$$path(X, X).$$
$$path(X, Y) \leftarrow edge(X, Z), path(Z, Y).$$
$$0.8 :: edge(a, c).$$
$$0.7 :: edge(a, b).$$
$$0.8 :: edge(c, e).$$
$$0.6 :: edge(b, c).$$
$$0.9 :: edge(c, d).$$
$$0.625 :: edge(e, f).$$
$$0.8 :: edge(f, d).$$

# Approximate Inference

## Iterative deepening




SLD tree up to depth 4 for the query $path(c, d)$ from the program

# Approximate Inference

*k-best*

- Uses a fixed number $k$ of proofs to obtain a lower bound of the probability of the query: the larger the k, the better the bound
- Given $k$, the best $k$ proofs are found, corresponding to the set of best $k$ explanations $K_k$
- $P(K_k)$ is a (lower) estimate of the probability of the query
- **Best** is intended in terms of probability: an explanation is better than another if its probability is higher
- **Branch and bound** approach: prune a derivation if its probability *falls below* that of the k-th best explanation

# Approximate Inference

## Monte Carlo

- Iterative procedure, until convergence
  1. Sample a world, by sampling each ground probabilistic fact/clause in turn
  2. Check whether the query is true in the world
  3. Compute the probability $\hat{p}$ of the query as the fraction of samples where the query is true
- Convergence is reached when the size of the confidence interval of $\hat{p}$ drops below a user-defined threshold $\delta$
- The binomial proportion confidence interval of $\hat{p}$ is used

$$\hat{p} \pm z_{1-\alpha/2} \sqrt{\frac{\hat{p}(1-\hat{p})}{n}}$$

  where $n$ is the number of samples and $z_{1-\alpha/2}$ is the $1-\alpha/2$ percentile of a standard normal distribution with $\alpha = 0.05$ usually
- If the width of the interval $< \delta$, it stops and returns $\hat{p}$

# Approximate Inference

## Monte Carlo

- The approach is not efficient on large programs, as proofs are often short while the generation of a world requires sampling many probabilistic facts
- Idea: generate samples lazily, by sampling probabilistic facts/clauses only when required by a proof
- In fact, it is not necessary to sample facts not needed by a proof, as any value for them would do

# Approximate Inference

## Monte Carlo Implementations

- **ProbLog1**: (Kimmig et al. (2011))
- **MCINTYRE** (Monte Carlo INference wiTh Yap REcord) (Riguzzi (2013)): applies the Monte Carlo approach of ProbLog1 to LPADs using the YAP internal database for storing all samples and using tabling for speeding up inference
- Also available in SWI-Prolog (included in the cplint suite)

# Approximate Inference

## MCINTYRE: *Monte Carlo INference wiTh Yap REcord*

- The LPAD clause $C_r = H_1 : \alpha_1 \vee \ldots \vee H_n : \alpha_n \leftarrow L_1, \ldots, L_m.$ is transformed into the set of clauses $MC(C_r)$:

$$MC(C_r, 1) = H_1 \leftarrow L_1, \ldots, L_m, sample\_head(ParList, r, VC, NH), NH = 1.$$
$$\ldots$$
$$MC(C_r, n) = H_n \leftarrow L_1, \ldots, L_m, sample\_head(ParList, r, VC, NH), NH = n.$$

  where $VC$ is a list containing each variable appearing in $C_r$ and $ParList$ is $[\Pi_{r1}, \ldots, \Pi_{rn_r}]$. If the parameters do not sum up to 1 the last clause (the one for *null*) is omitted

- It creates a clause for each head and samples a head index $NH$ with `sample_head/4`.
- If this index coincides with the index of the head disjunct, the derivation succeeds, otherwise it fails

# Approximate Inference

## MCINTYRE: *Monte Carlo INference wiTh Yap REcord*

*Example*: The following LPAD models the development of an epidemic or a pandemic:

$$
\begin{aligned}
C_1 &= epidemic : 0.6; pandemic : 0.3 : -flu(X), cold.\\
C_2 &= cold : 0.7.\\
C_3 &= flu(david).\\
C_4 &= flu(robert).
\end{aligned}
$$

Clause $C_1$ is transformed in:

$$
\begin{aligned}
MC(C_1, 1) = \; & epidemic : -flu(X), cold,\\
& sample\_head([0.6, 0.3, 0.1], 1, [X], NH), NH = 1.\\
MC(C_1, 2) = \; & pandemic : -flu(X), cold,\\
& sample\_head([0.6, 0.3, 0.1], 1, [X], NH), NH = 2.
\end{aligned}
$$

# Approximate Inference

## MCINTYRE: *Monte Carlo INference wiTh Yap REcord*

- If $Q = epidemic$, resolution matches the goal with the head of clause $MC(C_1, 1)$.
- Suppose $flu(X)$ succeeds with $X/david$ and $cold$ succeeds as well.
- Then

$$sample\_head([0.6, 0.3, 0.1], 1, [david], NH)$$

  is called.
- Since clause 1 with $X$ replaced by $david$ was not yet sampled, a number between 1 and 3 is sampled according to the distribution $[0.6, 0.3, 0.1]$ and stored in $NH$.
- If $NH = 1$, the derivation succeeds and the goal is true in the sample, if $NH = 2$ or $NH = 3$ then the derivation fails and backtracking is performed.

# Approximate Inference

## MCINTYRE: *Monte Carlo INference wiTh Yap REcord*

- This involves finding the solution $X/robert$ for $flu(X)$. $cold$ was sampled as true before so it succeeds again.
- Then

$$sample\_head([0.6, 0.3, 0.1], 1, [robert], NH)$$

is called to take another sample.

# References I

Darwiche, A. (2004). New advances in compiling cnf to decomposable negation normal form. In *Proceedings of the 16th European Conference on Artificial Intelligence*, ECAI'04, pages 318–322, Amsterdam, The Netherlands, The Netherlands. IOS Press.

Darwiche, A. (2011). SDD: A new canonical representation of propositional knowledge bases. In *IJCAI*, pages 819–826. IJCAI/AAAI.

De Raedt, L., Kimmig, A., and Toivonen, H. (2007). Problog: A probabilistic prolog and its application in link discovery. In *International Joint Conference on Artificial Intelligence*, pages 2462–2467.

Fierens, D., Van den Broeck, G., Renkens, J., Shterionov, D., Gutmann, B., Thon, I., Janssens, G., and De Raedt, L. (2015). Inference and learning in probabilistic logic programs using weighted Boolean formulas. *Theory and Practice of Logic Programming*, 15:358–401.

# References II

Janhunen, T. (2004). Representing normal programs with clauses. In de Mántaras, R. L. and Saitta, L., editors, *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*, pages 358–362. IOS Press.

Kimmig, A., Demoen, B., De Raedt, L., Costa, V. S., and Rocha, R. (2011). On the implementation of the probabilistic logic programming language ProbLog. 11(2-3):235–262.

Kimmig, A., Santos Costa, V., Rocha, R., Demoen, B., and De Raedt, L. (2008a). On the efficient execution of problog programs. In Garcia de la Banda, M. and Pontelli, E., editors, *Logic Programming*, pages 175–189, Berlin, Heidelberg. Springer Berlin Heidelberg.

Kimmig, A., Santos Costa, V., Rocha, R., Demoen, B., and De Raedt, L. (2008b). On the efficient execution of ProbLog programs. volume 5366 of *LNCS*, pages 175–189.

# References III

Lloyd, J. W. (1987). *Foundations of Logic Programming, 2nd Edition*.

Mantadelis, T. and Janssens, G. (2010). Dedicated tabling for a probabilistic setting. volume 7 of *LIPIcs*, pages 124–133.

Meert, W., Struyf, J., and Blockeel, H. (2010). Cp-logic theory inference with contextual variable elimination and comparison to bdd based inference methods. In De Raedt, L., editor, *Inductive Logic Programming*, pages 96–109, Berlin, Heidelberg. Springer Berlin Heidelberg.

Muise, C. J., McIlraith, S. A., Beck, J. C., and Hsu, E. I. (2012). Dsharp: Fast d-DNNF compilation with sharpSAT. volume 7310, pages 356–361.

# References IV

Poole, D. (2003). First-order probabilistic inference. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, IJCAI'03, pages 985–991, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Riguzzi, F. (2007). A top down interpreter for LPAD and CP-logic. In *Congress of the Italian Association for Artificial Intelligence*, number 4733 in LNAI, pages 109–120. Springer.

Riguzzi, F. (2009). Extended semantics and inference for the independent choice logic. *Logic Journal of the IGPL*, 17(6):589–629.

Riguzzi, F. (2013). MCINTYRE: A Monte Carlo system for probabilistic logic programming. 124(4):521–541.

# References V

Riguzzi, F. and Swift, T. (2010a). Tabling and Answer Subsumption for Reasoning on Logic Programs with Annotated Disjunctions. In Hermenegildo, M. and Schaub, T., editors, *Technical Communications of the 26th Int'l. Conference on Logic Programming (ICLP'10)*, volume 7 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 162–171, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Riguzzi, F. and Swift, T. (2010b). Tabling and answer subsumption for reasoning on logic programs with annotated disjunctions. In *ICLP (Technical Communications)*, volume 7 of *LIPIcs*, pages 162–171. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

Vlasselaer, J., Renkens, J., Van den Broeck, G., and De Raedt, L. (2014). Compiling probabilistic logic programs into sentential decision diagrams. pages 1–10.