Notes on a Java translator from BNs to join-tree Prism programs
(version 1.3)

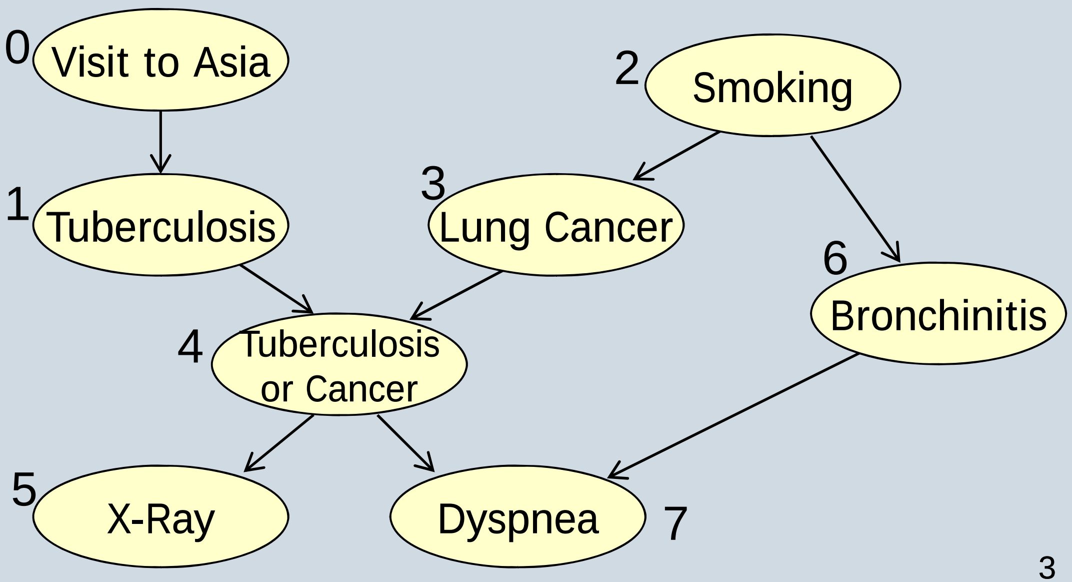
Yoshitaka Kameya, Munehiro Okada
Tokyo Institute of Technology

Contents

- Background:
  Translation from BNs to join-tree Prism programs
- How to run the translator
- How to run the translated Prism programs
- Additional notes

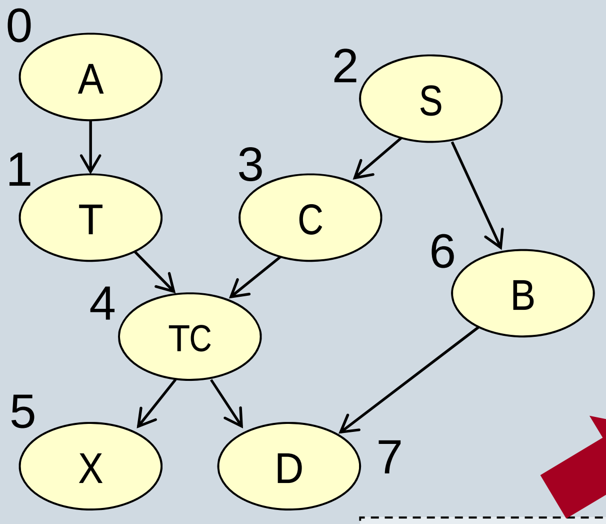
Background (1 of 3)

- Translation from BNs to join-tree Prism programs
  1. Construct a join tree (a bucket tree) following [Kask et al. 2005]
  2. Translate the join tree to a Prism program
- This translator is a simplified (less optimized) version of the one used in [Sato & Kameya 2008]
- The theoretical background is given in [Sato 2007] and [Sato & Kameya 2008]
- Running example:
  Asia network



Background (2 of 3)

- Step 1: Building a JT (a bucket tree)
  - Determine the order *O* of variables to be eliminated
    - Minimum deficiency ordering (MDO) [D'Ambrosio 1999] is used by default
  - Construct clusters and connect them under *O*

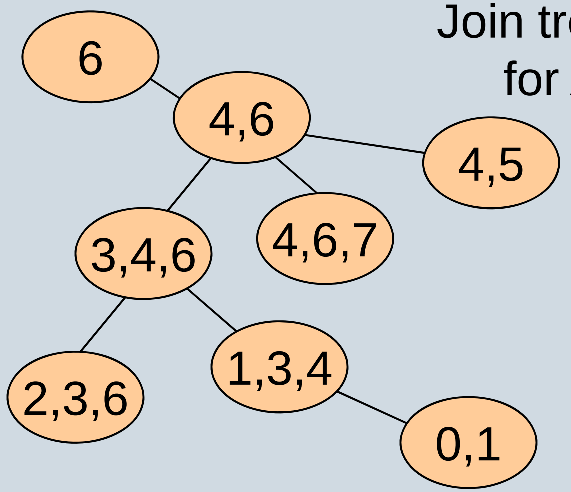


Bucket [0]: { p(0), p(1|0) }
Bucket [1]: { p(4|1,3) }
Bucket [5]: { p(5|4) }
Bucket [7]: { p(7|4,6) }
Bucket [2]: { p(2), p(3|2), p(6|2) }
Bucket [3]: { }
Bucket [4]: { }
Bucket [6]: { }
(initial)

CPT entry
for variable 6 ("B")

Bucket [0]: { p(0), p(1|0) } ➔ **cl_0(1)**
Bucket [1]: { p(4|1,3), **cl_0(1)** } ➔ cl_1(3,4)
Bucket [5]: { p(5|4) } ➔ cl_5(4)
Bucket [7]: { p(7|4,6) } ➔ cl_7(4,6)
Bucket [2]: { p(2), p(3|2), p(6|2) } ➔ cl_2(3,6)
Bucket [3]: { cl_1(3,4), cl_2(3,6) } ➔ cl_3(4,6)
Bucket [4]: { cl_5(4), cl_7(4,6), cl_3(4,6) } ➔ cl_4(6)
Bucket [6]: { cl_4(6) } ➔ cl_6()

Prob. marginalized with
variable 0 ("A")

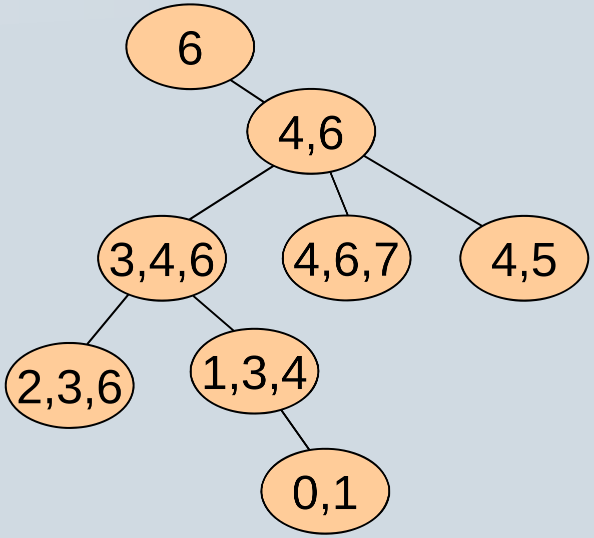


Join tree
for Asia

Background (3 of 3)

- Step 2: Translate the constructed JT to Prism

Bucket [0]: { p(0), p(1|0) } ➔ cl_0(1)
Bucket [1]: **{ p(4|1,3), cl_0(1) } ➔ cl_1(3,4)**
Bucket [5]: { p(5|4) } ➔ cl_5(4)
Bucket [7]: { p(7|4,6) } ➔ cl_7(4,6)
Bucket [2]: { p(2), p(3|2), p(6|2) } ➔ cl_2(3,6)
Bucket [3]: { cl_1(3,4), cl_2(3,6) } ➔ cl_3(4,6)
Bucket [4]: { cl_5(4), cl_7(4,6), cl_3(4,6) } ➔ cl_4(6)
Bucket [6]: { cl_4(6) } ➔ cl_6()




```
node_x1(X3,X4) :- pot_x1(X1,X3,X4).
pot_x1(X1,X3,X4) :- node_x0(X1), cpt(x4,[x1=X1,x3=X3],X4).

cpt(X,Pa,V) :-
  ( evidence(X,V) -> msw(bn(X,Pa),V)
  ; msw(bn(X,Pa),V)
  ).
```

Prism clauses

How to install the translator

- Requirements:
  - JDK 1.5 or later is installed
    - does not work on JDK 1.4 or earlier
    - Add $(TOP_DIR)/lib to CLASSPATH appropriately

- Procedure:
  - Let $(TOP_DIR) be the top directory for BN2Prism
  - Compile the translator:

```
> cd $(TOP_DIR)/src
> make
```

Edit Makefile according to your environment

How to run the translator (1 of 4)

- BN specification files (input):
  - the filename is assumed to be "BASENAME.xml"
    (e.g. BASENAME = asia, alarm, etc.)
  - Two format types are available
    - XMLBIF
    - XBIF (used in UAI-06 Evaluation Workshop; just a temporary name in this document)
  - To specify the evidences in XMLBIF, a separate evidence file is required
    - The format needs to follow the XMLBIF-EVIDENCE format
    - In XBIF, on the other hand, the information on evidences is buried in the BN specification file

How to run the translator (2 of 4)

- Output:
  - Translated join-tree Prism program
    - BASENAME.psm
  - Translated evidence file (if evidences are given)
    - BASENAME_evid.psm;
  - [optional] A file that contains randomly generated evidences
    - BASENAME_revid[0-3].psm
    - BASENAME_revid[0-3].pnl

0 - 0% of all variables (i.e. no evidences)
1 - 25% of all variables
2 - 50% of all variables
3 - 75% of all variables

How to run the translator (3 of 4)

- Command:

```
> java BN2Prism [-f FORMAT][-o][-t][-r][-e][-z][-n][-v][-h] BASENAME
```

- Command line options:
  - -f FORMAT : the translator reads the input whose format is FORMAT. FORMAT is either xmlbif (for XMLBIF) or xbif (for XBIF)
  - -o : the translator reads an external file (named BASENAME.num) that specifies the elimination order.
  - -t : the translator reads transposed CPTs (see below).
  - -r : the translator assigns random probabilities to the parameters.
  - -e : the translator generates auxiliary files that specify the random evidences
  - -z : the translator will make zero-compression of CPTs
  - -n: the translator normalizes the entries in CPTs
  - -v : the translator outputs detailed messages as comments.
  - -h : a brief help is displayed.

How to run the translator (4 of 4)

- Example:
  - Assume we have asia.xml (in XMLBIF) and asia_evid.xml (in XMLBIF-EVIDENCE)
  - Basic:

```
> java BN2Prism -f xmlbif asia
```

or simply

```
> java BN2Prism asia
```

  - When the entries in the CPTs in asia.xml are *transposed* :

```
> java BN2Prism -f xmlbif -t asia
```

  - When asia.xml follows the XBIF (not XMLBIF) format:

```
> java BN2Prism -f xbif asia
```

How to run the translated Prism program (1 of 3)

- Three types of model description are generated:
  - For belief propagation on JT (world/0-1)
    - *As described before*
  - For naïve probability computation (world_n/0-1)

```
world_n(Es):- assert_evidence(Es),!,world_n.
world_n:- world_n(_,_,_,_,_,_,_,_).

world_n(X0,X1,X2,X3,X4,X5,X6,X7) :-
   cpt(x0,[],X0), cpt(x1,[x0=X0],X1), cpt(x2,[],X2), cpt(x3,[x2=X2],X3), cpt(x4,[x1=X1,x3=X3],X4),
   cpt(x5,[x4=X4],X5), cpt(x6,[x2=X2],X6), cpt(x7,[x4=X4,x6=X6],X7).
```

  - For sampling (world_s/1)

```
world_s([x0=X0,x1=X1,x2=X2,x3=X3,x4=X4,x5=X5,x6=X6,x7=X7]):-
    msw(bn(x0,[]),X0), msw(bn(x1,[x0=X0]),X1), msw(bn(x2,[]),X2),
    msw(bn(x3,[x2=X2]),X3), msw(bn(x4,[x1=X1,x3=X3]),X4), msw(bn(x5,[x4=X4]),X5),
    msw(bn(x6,[x2=X2]),X6), msw(bn(x7,[x4=X4,x6=X6]),X7).
```

How to run the translated Prism program (2 of 3)

- Load the program ($(TOP_DIR)/includes/bn_common.psm needs to be copied to the current working directory in advance):

```
?- prism(asia).
```

- Check the distribution on each variable:

```
?- check_j_dist.
?- check_n_dist.
?- check_dist.
```

← BP on JT
← Naïve style
← Both styles

[Note]
Naïve style takes exponential time!

- Give an evidence list to check_dist/1, check_j_dist/1, check_n_dist/1 for computing conditional distributions:

```
?- check_j_dist([x3=v3_0]).
?- check_n_dist([x5=v5_1]).
?- check_dist([x3=v3_0,x5=v5_1]).
```

[Note]
x3 refers to the 4th variable (*Cancer*)
v3_0 refers to the 1st value (*Present*) of the 4th variable (*Cancer*)

Probability computation above is made by "hindsight" computation provided since Prism 1.9

How to run the translated Prism program (3 of 3)

- Batch execution:

```
> upprism BASENAME [Evid1 Evid2 ....]
```

  – Prism system does the followings for each *Evid1*, *Evid2*, ...
    - Read an evidence list Es from *EvidN*
    - Run check_j_dist(Es)

  – Example:

```
> upprism asia asia_evid.psm asia_revid0.psm asia_revid1.psm
```

Note: Handling evidences

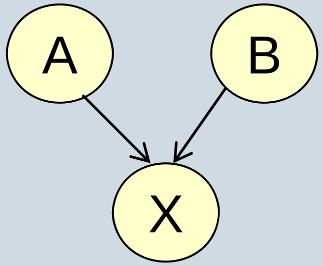
- In the PRISM program generated by BN2Prism, for each evidence X=V, evidence(X,V) will be added into the Prolog database, before probabilistic inferences performed
- Instead of calling msw/2 directly, we call a wrapper predicate cpt/3 that checks the existence of a evidence:

```
cpt(X,Pa,V):-
  ( evidence(X,V) -> msw(bn(X,Pa),V)
                                   % use only the value of the evidence if it exists
  ; msw(bn(X,Pa),V)      % otherwise, choose a value from possible ones
  ).
```

Note: Compressing CPT entries with zero probability

- If an entry in a CPT for a variable X has zero probability, we wish to remove such an entry to reduce the search space
- In other words, we change the set of outcomes of X according to the context (an instanciation of parent nodes A and B)
  - X takes on {0, 1} when B = 1
  - X takes on {1} when A = 0 and B = 0
  - X takes on {0} when A = 1 and B = 0
- In PRISM programs, this can be realized with more fine-grained `values' declarations

  → One computational advantage of logical (propositional) approaches



A	B	X	P(X|A,B)
0	0	0	0.0
0	0	1	1.0
0	1	0	0.7
0	1	1	0.3
1	0	0	1.0
1	0	1	0.0
1	1	0	0.4
1	1	1	0.6

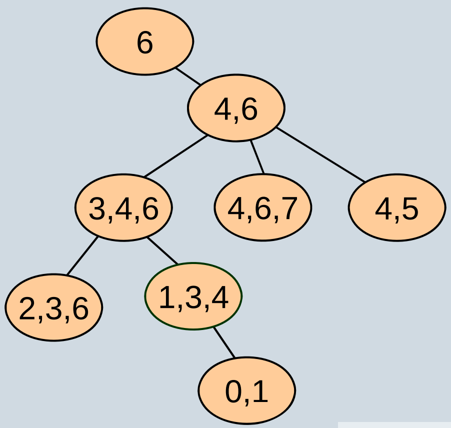
Note: Input/output dependencies in a bucket (1 of 3)

- Literals in the clause body are ordered according to the input/output dependencies among the probabilistic predicates

Add Input/output modes:

Bucket [0]: { p(-0), p(-1|+0) } ➔ cl_0(-1)
Bucket [1]: { p(-4|+1,+3), cl_0(-1) } ➔ cl_1(+3,-4)
Bucket [5]: { p(-5|+4) } ➔ cl_5(+4)
Bucket [7]: { p(-7|+4,+6) } ➔ cl_7(+4,+6)
Bucket [2]: { p(-2), p(-3|+2), p(-6|+2) } ➔ cl_2(-3,-6)
Bucket [3]: { cl_1(+3,-4), cl_2(-3,-6) } ➔ cl_3(-4,-6)
Bucket [4]: { cl_5(+4), cl_7(+4,+6), cl_3(-4,-6) } ➔ cl_4(-6)
Bucket [6]: { cl_4(-6) } ➔ cl_6()

Order the elements in a bucket (left-to-right) according to the input/output dependencies

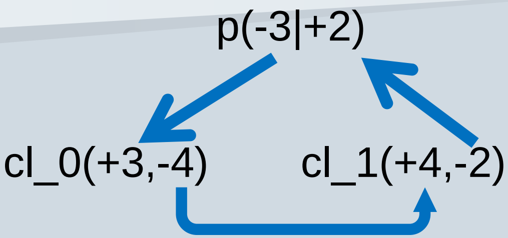


Bucket [0]: { p(-0), p(-1|+0) } ➔ cl_0(-1)
Bucket [1]: { cl_0(-1), p(-4|+1,+3) } ➔ cl_1(+3,-4)
Bucket [5]: { p(-5|+4) } ➔ cl_5(+4)
Bucket [7]: { p(-7|+4,+6) } ➔ cl_7(+4,+6)
Bucket [2]: { p(-2), p(-3|+2), p(-6|+2) } ➔ cl_2(-3,-6)
Bucket [3]: { cl_2(-3,-6), cl_1(+3,-4) } ➔ cl_3(-4,-6)
Bucket [4]: { cl_3(-4,-6), cl_5(+4), cl_7(+4,+6) } ➔ cl_4(-6)
Bucket [6]: { cl_4(-6) } ➔ cl_6()

```
node_x1(X3,X4) :- pot_x1(X1,X3,X4).
pot_x1(X1,X3,X4) :- node_x0(X1), cpt(x4,[x1=X1,x3=X3],X4).
```

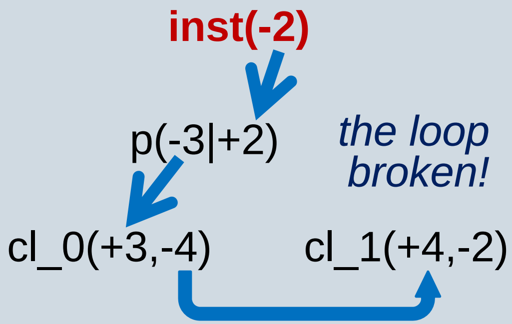
Note: Input/output dependencies in a bucket (2 of 3)

- Sometimes we have a loop of dependency in a bucket
- We insert some “instanciate” literals to break such a loop



...

Bucket [*n*]: { p(-3|+2), cl_0(+3,-4), cl_1(+4, -2) }
... ➔ cl_3(-2,-4)



...

Bucket [*n*]: { **inst(-2)**, p(-3|+2), cl_0(+3,-4), cl_1(+4, -2) }
... ➔ cl_3(-2,-4)

```
node_x3(X2,X4) :- pot_x3(X2,X3,X4).
pot_x3(X2,X3,X4) :-
        instanciate(x2,X2), cpt(x3,[x2=X2],X3), node_x0(X3,X4), node_x1(X4,X2).

instanciate(X,V) :-  % forcedly instanciate variable X as V
        range(X,Values), (evidence(X,V) -> true ; member(V,Values)).
```

Note: Input/output dependencies in a bucket (3 of 3)

- Unfortunately, the “instanciate” predicates cannot exploit the context-specific information (and hence cannot compress the CPT entries with zero probability)
- As a side effect, we can give the mode declarations to the probabilistic predicates:

```
Bucket [0]: { p(-0), p(-1|+0) } ➔ cl_0(-1)
Bucket [1]: { cl_0(-1), p(-4|+1,+3) } ➔ cl_1(+3,-4)
Bucket [5]: { p(-5|+4) } ➔ cl_5(+4)
Bucket [7]: { p(-7|+4,+6) } ➔ cl_7(+4,+6)
Bucket [2]: { p(-2), p(-3|+2), p(-6|+2) } ➔ cl_2(-3,-6)
Bucket [3]: { cl_2(-3,-6), cl_1(+3,-4) } ➔ cl_3(-4,-6)
Bucket [4]: { cl_3(-4,-6), cl_5(+4), cl_7(+4,+6) } ➔ cl_4(-6)
Bucket [6]: { cl_4(-6) } ➔ cl_6()
```

➔

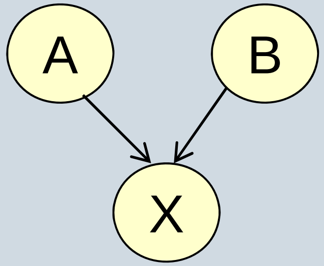
```
:- mode node_x0(-).
:- mode node_x1(+,-).
:- mode node_x5(+).
...

:- mode pot_x0(-,-).
:- mode pot_x1(-,+,-).
:- mode pot_x5(+,-).
...
```

Note: File format of network specification

- In both XMLBIF and XBIF, CPTs are specified in <TABLE>...</TABLE>
  - There can be different orders of entries (conditional probabilities or parameters) in a CPT.
  - Basically the translator assumes the order illustrated in http://www.cs.cmu.edu/afs/cs/user/fgcozman/www/Research/InterchangeFormat/ :

<TABLE>0.9 0.1 0.7 0.3 0.8 0.2 0.4 0.6</TABLE>



A	B	X	P(X|A,B)
0	0	0	0.9
0	0	1	0.1
0	1	0	0.7
0	1	1	0.3
1	0	0	0.8
1	0	1	0.2
1	1	0	0.4
1	1	1	0.6

  - On the other hand, some network files adopt a *transposed* order:

<TABLE>0.9 0.7 0.8 0.4 0.1 0.3 0.2 0.6</TABLE>

  - BN2Prism accepts the transposed CPTs when the -t option is specified.
  - Furthermore, with the –n option, the CPT entries will be normalized inside the translator:

<TABLE>1.0 3.0 1.0 1.0 2.0 1.0 2.0 0.5</TABLE>

→ <TABLE>0.25 0.75 0.5 0.5 0.666666 0.333333 0.8 0.2</TABLE>

References

- [D’Ambrosio 1999]
  B. D’Ambrosio (1999). Inference in Bayesian networks. AI Magazine 20 (2), pp.21-36.
- [Kask et al. 2005]
  K. Kask, R. Dechter, J. Larrosa and A. Dechter (2005). Unifying tree decompositions for reasoning in graphical models. Artificial Intelligence 166 (1-2), pp.165-193.
- [Sato 2007]
  T. Sato (2007). Inside-outside probability computation for belief propagation. *Proceedings of the 20th International Joint Conference on Artificial Intelligence* (IJCAI-2007), pp.2605–2610, 2007.
- [Sato and Kameya 2008]
  T. Sato and Y. Kameya (2008). New advances in logic-based probabilistic modeling by PRISM. In De Raedt et al. (editors), *Probabilistic Inductive Logic Programming*, LNCS 4911, Springer, pp.118–155, 2008.