BNS-JT toolkit manual
Developed by Ji-Eun Byun, PhD under supervision Prof. Junho Song

This toolkit is developed to implement Bayesian network (BN) for system reliability analysis (SRA) using Junction Tree (JT) algorithm so as to handle complex and large-scale system events, e.g. high-dimensional distributions, continuous distributions, and approximate inferences. This toolkit is developed based on the main reference Byun and Song (In review, a) and the supplementary references Byun et al. (2019), Byun and Song (In review, b), Koller and Friedman (2009), and Barber (2012). This toolkit also includes the codes for matrix-based Bayesian network (MBN), which can efficiently handle discrete distributions (Byun et al. 2019). We highly recommend the user to use MBN for handling discrete distributions.

This document focuses on the implementation of the toolkit, while theoretical details can be found in the references. In the followings, the notations follow those used in the code scripts and may differ from those used in the references. Sections 1 to 7 illustrate the major classes and functions, while their applications are demonstrated by the example in Section 8. This document would be better understood when being accompanied with the code script of the example.

Note that the folder "/BNS-JT" must be added to the path in Matlab.

1 Matrix-based Bayesian network (MBN) @Cpm

This class features the matrix-based Bayesian network (MBN) (Byun et al. 2019; Byun and Song, In review, b), which utilizes the conditional probability matrix (CPM) as the data structure. A CPM represents a probability distribution using two matrices: event matrix **C** and probability matrix **p**.

The MBN can represent either the probability mass functions (PMFs) (for discrete r.v.'s) or a set of samples (for both continuous and discrete r.v.'s). On the other hand, the MBN cannot represent the probability density functions (PDFs) (for continuous r.v.'s). If there is some continuous r.v.'s in the given problem, they should be separately handled in Clique functions (ref: Section 4.1.1).

1.1 Properties

1.1.1 variables

An array of indices of the variables in the given scope[1]. It may include the conditional variables, which must be put in the last, i.e. the child nodes come first, then the parent nodes follow.

1.1.2 numChild

A numeric scalar that denotes the number of child nodes in variables

1.1.3 C

The event matrix that stores the states of the given instances. The variables represented by each column should follow the same order with that of variables.

1.1.4 p

The probability vector that stores the probabilities of given instances. The instances stored in each row should follow the same order with the rows in C.

[1] A scope of a probability distribution denotes the set of r.v.'s on which the distribution is defined, e.g. the scope of $P(X,Y)$ is $\{X,Y\}$.

In case the instances have been sampled, this vector can be left empty. Or, for importance sampling (IS), it can store the true probability (or unnormalized true probability) of the samples, i.e. the numerator of the sampling weight (ref: Section 3.5).

1.1.5 q

This is the vector of the sampling probabilities, i.e. the denominator of the sampling weight (ref: Section 3.5). If the given instances are not samples, this vector is left empty.

1.1.6 sampleIndex

This is an array of samples' indices, by which we can keep track of the samples across CPMs. Such tracking is required for different samples should not be computed together (i.e. instances from different samples are always not compatible with each other).

2 Encoding composite states of MBN @Variable

One of the main advantages of the MBN is that the states can have an extended definition to represent multiple states at once, namely *composite state* (Byun and Song, In review, b). In this toolkit, this is encoded using the property B. Each column of B represents each *basic state*, i.e. the singular state, while each row represents a state (both the basic and composite states). In the *i*-th row, the *j*-th column is 1 if the state *i* is representing the basic state *j*; and otherwise, 0. This setting mandates that (1) *the states must be positive integers* (i.e. they start from 1), and (2) *the basic states must have smaller indices than the composite states*. By construction, the upper part of B is an identity matrix. When B is empty, it is assumed that there is no composite state, i.e. the matrix is assumed to be an identity matrix.

The property v stores the additional descriptions of the (mostly basic) states. It can take any form (e.g. a numeric array or a string cell) or be left empty.

3 Functions for conditional probability matrix (selected)

3.1 condition.m

Conditions the given CPMs on the given instances

Input variables	Illustrations
M	A single Cpm or an array of Cpms
condVars	An array of indices of the variables to be conditioned
condStates	An array of the states to be conditioned
vars	An array of Variable
sampleInd	In case the given instances have been sampled, an array of the indices of the samples; otherwise, left empty; Recall that instances from different samples are always not compatible

Output variables	Illustrations
Mcond	An array of conditioned cliques
vars	An array of Variable (might be updated during conditioning)

3.2 createCompositeStates.m

Automatically creates the composite states by investigating the given CPM. vars is updated if any composite state is created.

3.3 mcsProduct.m

Multiply a set of CPMs by Monte Carlo Simulation (MCS). This applies only to discrete r.v.'s. The PMFs should be given in a way that after the multiplication, the distribution must not have any parental nodes, i.e. the conditional r.v.'s (otherwise, the sampling is impossible).

3.4 plus.m

Given a pair of CPMs, simply sums up the probability values of the instances. The two CPMs must have the same scope.

3.5 computeSample.m

Computes the mean and the standard deviation of the samples in the given Cpm. It can evaluate the samples from MCS, unnormalized IS, and normalized IS. In case of IS, the sample weights are computed as cpm.p ./ cpm.q.

4 Junction tree algorithm runJunctionTree.m

This code runs the junction tree algorithm.

4.1 Input: @JunctionTree

Input variables	Illustrations
jtree	Junction tree structure

4.1.1 CliqueFunctionCell

This is a cell array of function handles. Each cell element represents the computation in each clique, i.e. (1) the multiplication to compute the assigned distributions and received messages and (2) the marginalization to compute the message being sent out. The function must have three outputs (following the presented order): the distribution to be stored in the clique after the computation, the message being sent, and the variables information (i.e. Variable). The variables information is always included in the input and output of the functions for it might be updated during the operations of product and conditioning in the clique functions.

While cliques can be of any form, they should be stored in a way that the user can understand and retrieve the data after running the algorithm. On the other hand, messages can also be of any form, but they should be designed considering that they will be used by the receiver clique, i.e. the message should be compatible with the receiver's clique function.

If one does not want to store the distributions of the clique after the computation (for efficient memory management), the first output can be declared as an empty array at the end of the clique function. For the last clique, which does not send any message, the message can be declared as an empty array.

On the other hand, the first three inputs must be defined as follows (in the presented order): the array of the CPMs assigned to the clique, the messages coming to the clique, and the variables information. In addition to the first three arguments, one can add as many inputs as desired; however, since these inputs are not directly introduced to the function runJunctionTree.m, they should be defined in the main script.

While designing the clique function, one should be aware of the formats of the distributions and the messages since they are automatically input during the algorithm. The distributions are given as an array of Cpm that are assigned in reference to cpmInd (Section 4.1.2) and cpm (Section 4.1.5). On the other hand, the messages are given as a cell array of the messages from the sender cliques. While it is common to deliver the information as a distribution (i.e. Cpm), the algorithm assembles the messages using a cell so that the format of the messages can be more flexible.

As illustrated in Section 4.1.4, the number of samples is fixed across all clique functions (in the context of the *collapse particles* technique). Therefore, if the computation involves sampling, the variable nSample should be declared as a global variable and used without specifying the value.

4.1.2 cpmInd

This is a cell array of the indices of the CPMs assigned to the cliques (the indices used in cpm - Section 4.1.5 - and variables - Section 4.1.6).

4.1.3 messageSchedule

This is a two-column array that stores the indices of the sender cliques (first column) and receiver cliques (second column). For the last clique, which does not send any message, the index is set as 0.

4.1.4 nSample

This is a scalar that indicates the number of samples. This variable is declared globally in the function so that it can remain constant over all clique functions. This is intended to apply the sampling in the context of the *collapse particle* (also known as *Rao-Blackwellized particles*), for which the details can be found in Chapter 12.4 of Koller and Friedman (2009) and Byun and Song (In review, a).

This property does not have to be defined when none of the clique functions operates the sampling.

4.1.5 cpm

An array of Cpm (Section 1).

4.1.6 variables

An array of Variable (Section 2).

4.2 Output

Output variables	Illustrations
cliques	A cell array of computed cliques
messages	A cell array of computed messages
vars	An array of Variable

5 Junction tree algorithm by conditioning runJunctionTree_conditioning.m

This function employs the *conditioning* technique to inference the junction tree. The conditioning is effective when being applied to the r.v.'s having a large number of child nodes. For details of *conditioning,* the user is referred to Chapter 9.5 of Koller and Friedman (2009) and Byun and Song (In review, a).

5.1 Input

Input variables	Illustrations
jtree	Junction tree structure; Since the CPMs are conditioned by the algorithm, the Clique functions should be designed while presuming that the first input argument is given as the conditioned CPMs; Note that the distributions of the conditioned variables do not need to be assigned to the cliques of JT
conditionedCpms	An array of CPMs of r.v.'s to which the conditioning is applied; [Caution] The CPMs in the array must not include any parental variables, i.e. the variables being conditioned; if there are, a subset of them can be multiplied to remove any parental variables

Others remain the same with the original junction tree algorithm.

6 Parameter sensitivity runJunctionTree_sensitivity.m

This code evaluates the parameter sensitivity using JT.

6.1 Input

Input variables	Illustrations
jtree	Junction tree structure
sensitiveCliqueIndArray	An array of indices of the cliques that involve the query parameter
sensitiveCliqueFunCell	A cell array of functions as the derivatives of the distributions (which replace the original distributions); Must have the same length with that of sensitiveCliqueIndArray

6.2 Computation

The junction tree algorithm is repeatedly performed while replacing each of the clique functions in reference to sensitiveCliqueIndArray and sensitiveCliqueFunCell. Then, the results of these iterations are summed up to draw the final result.

6.3 Output

Output variables	Illustrations
cliques_sensitivity	The parameter sensitivity of the distributions of the cliques
messages_sensitivity	The parameter sensitivity of the distributions of the edges
vars	An array of Variable

7 Parameter sensitivity and conditioning runJunctionTree_sensitivity_conditioning.m

This function evaluates the parameter sensitivity by employing the conditioning. The inputs are the same with those in Sections 5 and 6.

8 Example: A series system demoSeriesSystem.m

8.1 Quantification

Consider a series system, whose state is represented by r.v. S, with two component events X_1 and X_2. The probability distributions of the components are parameterized by the fault intensity $v(\lambda_n^h)$ as

$$P(x_n^1) = \exp\left(-v(\lambda_n^h)\right), n = 1,2 \tag{1}$$

where $v(\lambda_n^h)$ depends on h which is the state of hazard event H. This system can be modelled by the BN in Figure 1.

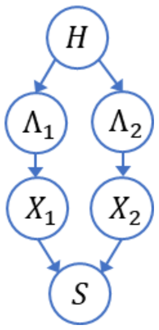

Figure 1. BN for example system

The r.v. H can take one of the two states, i.e. the intensity being severe (h^1) and mild (h^2), whose probability distribution can be represented by the CPM $\mathcal{M}_H = \langle \mathbf{C}_H; \mathbf{p}_H \rangle$ such that

$$\mathbf{C}_H = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \mathbf{p}_H = \begin{bmatrix} 0.1 \\ 0.9 \end{bmatrix} \tag{2}$$

Then, the parameters are represented by the r.v.'s Λ_1 and Λ_2, whose CPMs $\mathcal{M}_{\Lambda_1}$ and $\mathcal{M}_{\Lambda_2}$ are quantified as

$$\mathbf{C}_{\Lambda_n} = \begin{bmatrix} 1 & 1 \\ 2 & 2 \end{bmatrix}, \mathbf{p}_{\Lambda_n} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \ n = 1,2 \tag{3}$$

where the first and the second columns of $\mathbf{C}_{\Lambda_n}$ represent the states of Λ_n and H, respectively; and the states λ_1^1, λ_1^2, λ_2^1, and λ_2^2 refer to the fault intensity $v(\cdot)$ being 0.05, 0.001, 0.03, and 0.005, respectively.

On the other hand, the component events take the binary-state, i.e. either survival (x_n^1) or failure (x_n^2), $n = 1,2$, having the CPM $\mathcal{M}_{X_n} = \langle \mathbf{C}_{X_n}; \mathbf{p}_{X_n} \rangle$ derived from Eq. (1) as

$$\mathbf{C}_{X_n} = \begin{bmatrix} 1 & 1 \\ 2 & 1 \\ 1 & 2 \\ 2 & 2 \end{bmatrix}, \mathbf{p}_{X_n} = \begin{bmatrix} \exp\left(-v(\lambda_n^1)\right) \\ 1 - \exp\left(-v(\lambda_n^1)\right) \\ \exp\left(-v(\lambda_n^2)\right) \\ 1 - \exp\left(-v(\lambda_n^2)\right) \end{bmatrix} \tag{4}$$

where the first and second columns of $\mathbf{C}_{X_n}$ respectively denote the states of X_n and Λ_n.

Since the system event S is a series system, the distribution $P(S|X_1, X_2)$ can be represented by the CPM $\mathcal{M}_S = \langle \mathbf{C}_S; \mathbf{p}_S \rangle$ such that

$$\mathbf{C}_S = \begin{bmatrix} 2 & 2 & 3 \\ 2 & 1 & 2 \\ 1 & 1 & 1 \end{bmatrix}, \mathbf{p}_S = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \tag{5}$$

where the columns of $\mathbf{C}_S$ stand for the states of S, X_1, and X_2, sequentially; and the states s^1 and s^2 respectively denote the survival and the failure. In the equation, the composite state x_2^3 is introduced to represent both x_2^1 and x_2^2 (Byun et al. 2019 and Byun and Song, In review, b).

8.2 Junction tree (JT) computation runJunctionTree.m

The junction tree (JT) can be designed as in Figure 2, which presents the graph, message-passing schedule, and distribution assignment.

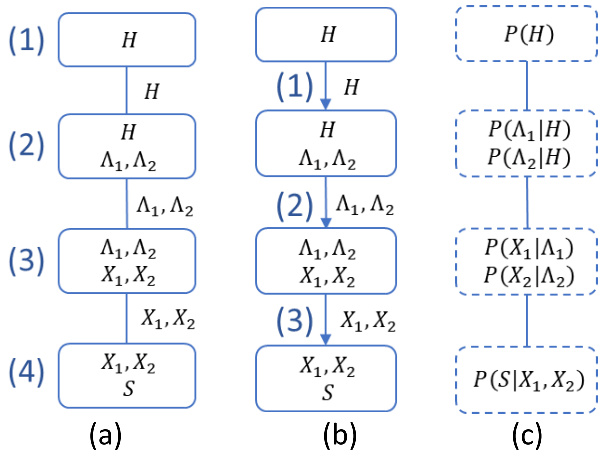
(1)
H
H
(2)
H Λ_1, Λ_2
Λ_1, Λ_2
(3)
Λ_1, Λ_2 X_1, X_2
X_1, X_2
(4)
X_1, X_2 S
(a)
H
(1) H
H Λ_1, Λ_2
(2) Λ_1, Λ_2
Λ_1, Λ_2 X_1, X_2
(3) X_1, X_2
X_1, X_2 S
(b)
$P(H)$
$P(\Lambda_1|H)$ $P(\Lambda_2|H)$
$P(X_1|\Lambda_1)$ $P(X_2|\Lambda_2)$
$P(S|X_1, X_2)$
(c)

Figure 2. JT of example: (a) JT graph, (b) message-passing schedule, and (c) distributions assignment

The JT implies the inference procedure as follows. Clique 1 does not perform any computation and sends off the assigned distribution $P(H)$ as the message. Then, Clique 2 receives the message and updates the distribution as

$$P(\Lambda_1, \Lambda_2, H) = P(\Lambda_1|H) \cdot P(\Lambda_2|H) \cdot P(H) \tag{6}$$

while sending off the message

$$P(\Lambda_1, \Lambda_2) = \sum_H P(\Lambda_1, \Lambda_2, H) \tag{7}$$

This message is received by Clique 3, whereby it computes the internal distribution as

$$P(X_1, X_2, \Lambda_1, \Lambda_2) = P(X_1|\Lambda_1) \cdot P(X_2|\Lambda_2) \cdot P(\Lambda_1, \Lambda_2) \tag{8}$$

and the message as

$$P(X_1, X_2) = \sum_{\Lambda_1, \Lambda_2} P(X_1, X_2, \Lambda_1, \Lambda_2) \tag{9}$$

Finally, Clique 4 updates its distribution as

$$P(S, X_1, X_2) = P(S|X_1, X_2) \cdot P(X_1, X_2) \tag{10}$$

After completing the inference, the system failure probability $P(s^2)$ can be retrieved from Clique 4 whose scope includes S, i.e.

$$P(s^2) = \sum_{X_1,X_2} P(s^0, X_1, X_2) \tag{11}$$

which is calculated as $P(s^2) = 0.0131$.

8.3 JT computation runJunctionTree_conditioning.m

The conditioning can be applied to r.v. H so that the cliques can be broken up for each component as illustrated in Figure 3. Although the effect is not dramatic in this example, it becomes more significant as the number of components increases.

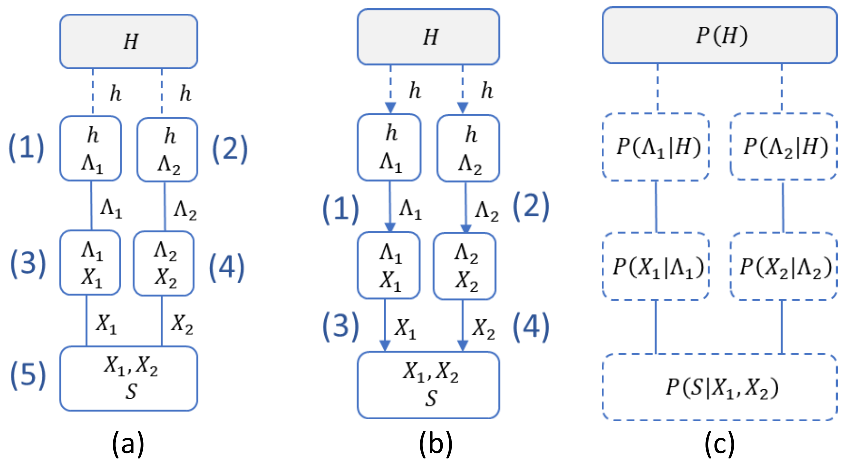

Figure 3. JT of example with conditioning on H: (a) JT graph, (b) message-passing schedule, and (c) distributions assignment

Given the conditioning state h, Cliques 1 and 2 condition the distributions $P(\Lambda_n|H)$ as $P(\Lambda_n|h)$, $n = 1,2$, and send them to Cliques 3 and 4, respectively. These cliques receive the messages to compute the internal distribution, i.e.

$$P(X_n, \Lambda_n|h) = P(X_n|\Lambda_n) \cdot P(\Lambda_n|h),\ n = 1,2 \tag{12}$$

and send the message, i.e.

$$P(X_n|h) = \sum_{\Lambda_n} P(X_n, \Lambda_n|h) \tag{13}$$

Finally, the two messages are accepted by Clique 5 to compute the distribution as

$$P(S, X_1, X_2|h) = P(S|X_1, X_2) \cdot P(X_1|h) \cdot P(X_2|h) \tag{14}$$

After repeating the inference for h^1 and h^2 and summing up the results, the system failure probability is computed as $P(s^2) = 0.0131$, which is identical to the result in Section 8.2.

8.4 Parameter sensitivity runJunctionTree_sensitivity.m

We investigate the sensitivity of the system failure probability $P(s^2)$ with respect to the parameters λ_n^h, $n = 1,2$ and $h = 1,2$. Since the probabilities of $P(X_n|\Lambda_n)$ depend on those parameters, the distributions are replaced by $\partial P(X_n|\Lambda_n)/\partial\lambda_n^h$, which can be represented by the CPMs $\mathcal{M}_{\partial X_{n,h}} = \langle \mathbf{C}_{\partial X_{n,h}}; \mathbf{p}_{\partial X_{n,h}} \rangle$, $n = 1,2$ and $h = 1,2$, such that

$$\mathbf{C}_{\partial X_{n,1}} = \begin{bmatrix} 1 & 1 \\ 2 & 1 \\ 1 & 2 \\ 2 & 2 \end{bmatrix}, \mathbf{p}_{\partial X_{n,1}} = \begin{bmatrix} -v(\lambda_n^1) \cdot \exp\left(-v(\lambda_n^1)\right) \\ v(\lambda_n^1) \cdot \exp\left(-v(\lambda_n^1)\right) \\ 0 \\ 0 \end{bmatrix} \tag{15}$$

and

$$\mathbf{C}_{\partial X_{n,2}} = \begin{bmatrix} 1 & 1 \\ 2 & 1 \\ 1 & 2 \\ 2 & 2 \end{bmatrix}, \mathbf{p}_{\partial X_{n,2}} = \begin{bmatrix} 0 \\ 0 \\ -v(\lambda_n^2) \cdot \exp\left(-v(\lambda_n^2)\right) \\ v(\lambda_n^2) \cdot \exp\left(-v(\lambda_n^2)\right) \end{bmatrix} \tag{16}$$

The parameter sensitivity is evaluated as 4.62×10^{-3}, 8.95×10^{-4}, 2.77×10^{-3}, and 4.47×10^{-3} respectively for $v(\lambda_1^1)$, $v(\lambda_1^2)$, $v(\lambda_2^1)$, and $v(\lambda_2^2)$.

8.5 Parameter sensitivity by conditioning runJunctionTree_sensitivity_conditioning.m

The parameter sensitivity can also be evaluated while employing the conditioning. Using this function, the same result can be obtained with that of Section 8.4.

9 References

Barber, D. (2012). Bayesian reasoning and machine learning. Cambridge University Press: Cambridge, UK.

Byun, J.-E., and Song, J. (a) On the use of Bayesian network and junction tree for reliability analysis of complex and large-scale systems. (In review)

Byun, J.-E., and Song, J. (b) Generalized matrix-based Bayesian network for multi-state systems. (In review)

Byun, J.-E., Zwirglmaier, K., Straub, D., and Song, J. (2019). Matrix-based Bayesian Network for efficient memory storage and flexible inference. Reliability Engineering and System Safety, 185, 533–545.

Koller, D., and Friedman, N. (2009). Probabilistic Graphical Models: Principles and Techniques. Foundations. MIT Press: Cambridge, US.