Easy Factor Graph: the flexible and efficient tool for managing undirected graphical models

Casalino Andrea
andrecasa91@gmail.com

Chapter 1

What is EFG

Easy Factor Graph (EFG), is a simple and efficient C++ library for managing undirected graphical models.
EFG allows you to build step by step a graphical model made of unary or binary potentials, i.e. factors involving one or two variables. It contains several tools for exporting and importing graphs from textual file. EFG allows you to perform all the probabilistic queries described in Chapter 2, from marginal probabilities computation to learning the tunable parameters of a graph. All the work is internally done by EFG: you just have to focus on what you need to compute.
A nice Graphic User Interface application, described in **??**, can be exploited to handle small and medium size structure.
The rest of this guide is structured as follows. Chapter 2 will introduce the main theoretical concepts about factor graphs, with the aim of explaining the capabilities of EFG. Chapter 3 will explain the format of the xml files adopted to represent factor graphs, exploited when importing or exporting the models to or from textual files. Chapter 4 will present the examples adopted for showing how EFG works. All the remaining Chapters, will describe the structure of the classes constituting EFG [1].

[1]A similar guide, but in a html format, is also available at http://www.andreacasalino.altervista.org/__EFG_doxy_guide/index.html.

Chapter 2

Theoretical background on factor graphs

This Section will provide a background about the basic concepts in probabilistic graphical models. Moreover, a precise notation will be introduced and used for the rest of this guide.

2.1 Preliminaries

This library is intended for managing network of categorical variables. Formally, the generic categorical variable V has a discrete domain Dom:

$$Dom(V) = \{v_0, \cdots, v_n\} \tag{2.1}$$

Essentially, $Dom(V)$ contains all the possible realizations of V. The above notation will be adopted for the rest of the guide: capital letters will refer to variable names, while non capital refer to their realizations. Group of categorical variables can be considered categorical variables too, having a domain that is the Cartesian product of the domains of the variables constituting the group. Suppose X is obtained as the union of variables $V_{1,2,3,4}$, i.e. $X = \bigcup_{i=1}^{4} V_i$, then:

$$Dom(X) = Dom(V_1) \times Dom(V_2) \times Dom(V_3) \times Dom(V_4) \tag{2.2}$$

The generic realization x of X is a set of realizations of the variables $V_{1,2,3,4}$, i.e. $x = \{v_1, v_2, v_3, v_4\}$. Suppose $V_{1,2,3}$ have the domains reported in the tables 2.1. The union $X = \bigcup_{i=1}^{3} V_i$ is a categoric variable whose domain is made by the combinations reported in table 2.2.

The entire population of variables contained in a model is a set denoted as $\mathcal{V} = \{V_1, \cdots, V_m\}$. As will be exposed in the following, the probability of $\bigcup_{V_i \in \mathcal{V}} V_i$ [1] is computed as the product of a certain number of components called factors.

Knowing the joint probability of $V_{1,\cdots,m}$, the probability distribution of a subset $S \subset \{V_1, \cdots, V_m\}$ can be in general (not only for graphical models) obtained through marginalization. Assume C is the complement of S: $C \cup S = \bigcup_{i=1}^{m} V_i$ and $C \cap S = \emptyset$, then:

$$\mathbb{P}(S = s) = \sum_{\forall \hat{c} \in Dom(C)} \mathbb{P}(S = s, C = \hat{c}) \tag{2.3}$$

[1] Which is the joint probability distribution of all the variables in a model

$Dom(V_1)$
v_{10}
v_{11}

$Dom(V_2)$
v_{20}
v_{21}
v_{22}

$Dom(V_3)$
v_{30}
v_{31}

Table 2.1 Example of domains for the group of variables $V_{1,2,3}$.

$Dom(X) = Dom(V_1 \cup V_2 \cup V_3)$
$x_0 = \{v_{10}, v_{20}, v_{30}\}$
$x_1 = \{v_{10}, v_{20}, v_{31}\}$
$x_2 = \{v_{11}, v_{20}, v_{30}\}$
$x_3 = \{v_{11}, v_{20}, v_{31}\}$
$x_4 = \{v_{10}, v_{21}, v_{30}\}$
$x_5 = \{v_{10}, v_{21}, v_{31}\}$
$x_6 = \{v_{11}, v_{21}, v_{30}\}$
$x_7 = \{v_{11}, v_{21}, v_{31}\}$
$x_8 = \{v_{10}, v_{22}, v_{30}\}$
$x_9 = \{v_{10}, v_{22}, v_{31}\}$
$x_{10} = \{v_{11}, v_{22}, v_{30}\}$
$x_{11} = \{v_{11}, v_{22}, v_{31}\}$

Table 2.2 Example of domains for the group of variables $V_{1,2,3}$.

In the above computation, variables in C were marginalized. Indeed they were in a certain sense eliminated, since the probability of the sub set S was of interest, no matter the realizations of all the variables in C.

A factor, sometimes also called a potential, is a positive real function describing the correlation existing among a subset of variables $D^i \subset \mathcal{V}$. Suppose factor Φ_i involves $\{X, Y, Z\}$, i.e. $D^i = \{X, Y, Z\}$. Then, $\Phi_i(X, Y, Z)$ is a function defined over $Dom(D^i)$. More formally:

$$\Phi_i(D^i) = \Phi_i(X, Y, Z) : \textsc{Domain}(X) \times \textsc{Domain}(Y) \times \textsc{Domain}(Z) \longrightarrow \mathbb{R}^+ \tag{2.4}$$

The aim of Φ_i is to assume 'high' values for those combinations $d^i = \{x, y, z\}$ that are probable and low values (at least a null) for those being improbable. The entire population of factors $\{\Phi_1, \cdots \Phi_p\}$ is considered for computing $\mathbb{P}(V_{1,\cdots,m})$, i.e. the joint probability distribution of all the variables in the model. The energy function E of a graph is defined as the product of the factors:

$$E(V_{1,\cdots,m}) = \Phi_1(D^1) \cdot \cdots \cdot \Phi_p(D^p) = \prod_{i=1}^{p} \Phi_i(D^i) \tag{2.5}$$

E is addressed for computing the joint probability distribution of the variables in $\mathcal{V}$:

$$\mathbb{P}(V_{1,\cdots,m}) = \frac{E(V_{1,\cdots,m})}{\mathcal{Z}} \tag{2.6}$$

where $\mathcal{Z}$ is a normalization coefficient defined as follows:

$$\mathcal{Z} = \sum_{\forall \tilde{V}_{1,\cdots,m} \in Dom(\bigcup_{i=1,\cdots,m} V_i))} E(\tilde{V}_{1,\cdots,m}) \tag{2.7}$$

Although the general theory behind graphical models supports the existance of generic multivaried factors, this library will address only two possible types:

- Binary potentials: they involve a pair of variables.
- Unary potentials: they involve a single variable.

We can store the values in the image of a Binary potential in a two dimensional table. For instance, suppose Φ_b involves variables A and B, whose domains contains 3 and 5 possible values respectively:

$$\begin{aligned} \textsc{Dom}(A) &= \{a_1, a_2, a_3\} \\ \textsc{Dom}(B) &= \{b_1, b_2, b_3, b_4, b_5\} \end{aligned} \tag{2.8}$$

The values assumed by $\Phi_b(A, B)$ are described by table 2.3. Essentially, $\Phi_b(A, B)$ tells us that the combinations $\{a_0, b_1\}$, $\{a_2, b_2\}$ are highly probable; while $\{a_0, b_0\}$, $\{a_1, b_1\}$ and $\{a_2, b_4\}$ are moderately probable. Let be

	b_0	b_1	b_2	b_3	b_4
a_0	1	4	0	0	0
a_1	0	1	0	0	0
a_2	0	0	5	0	1

Table 2.3 The values in the image of $\Phi_b(A,B)$.

a_0	a_1	a_2
0	2	0.5

Table 2.4 The values in the image of $\Phi_u(A)$.

$\Phi_u(A)$ a Unary potential involving variable A. The values characterizing Φ_u can be stored in a simple vector, see table 2.4. If $\Phi_b(A,B)$ would be the only potential in the model, the joint probability density of A and B will assume the following values [2]:

$$\mathbb{P}(a_0,b_1) = \frac{\Phi_b(a_0,b_1)}{\mathcal{Z}} = \frac{4}{\mathcal{Z}} = 0.3333 \tag{2.9}$$

$$\mathbb{P}(a_2,b_2) = \frac{\Phi_b(a_2,b_2)}{\mathcal{Z}} = \frac{5}{\mathcal{Z}} = 0.4167 \tag{2.10}$$

$$\mathbb{P}(a_0,b_0) = \frac{\Phi_b(a_0,b_0)}{\mathcal{Z}} = \mathbb{P}(a_1,b_1) = \mathbb{P}(a_2,b_4) = \frac{1}{\mathcal{Z}} = 0.0833 \tag{2.11}$$

since $\mathcal{Z}$ is equal to:

$$\mathcal{Z} = \sum_{\forall i=\{0,1,2\},\forall j=\{0,1,2,3,4\}} \Phi_b(A=a_i, B=b_j) = 12 \tag{2.12}$$

Both Unary and Binary potentials, can be of two possible classes:

- Simple shape. The potential is simply described by a set of values characterizing the image of the factor. $\Phi_b(A,B)$ and $\Phi_u(A)$ of the previous example are both Simple shapes. Class Potential_Shape handles this kind of factors.
- Exponential shape. They are indicated with Ψ_i and their image set is defined as follows:

$$\Psi_i(X) = exp(w \cdot \Phi_i(X)) \tag{2.13}$$

  where Φ_i is a Simple shape. Class Potential_Exp_Shape handles this kind of factors. The weight w, can be tunable or not. In the first case, w is a free parameter whose value is decided after training the model (see Section 2.6), otherwise is a constant . Exponential shapes with fixed weight will be denoted with $\overline{\Psi}_i$.

Figure 2.1 resumes all the possible categories of factors that can be present in the models handled by this library.

Figure 2.2 reports an example of undirected graph. Set $\mathcal{V}$ is made of 4 variables: A, B, C, D. There are 5 Binary potentials and 2 Unary ones. The graphical notation adopted for Fig. 2.2 will be adopted for the rest of this guide. Weights α, β, γ and δ are assumed for respectively $\Psi_{AC}, \Psi_{AB}, \Psi_{CD}, \Psi_B$. For the sake of clarity, the joint probability of the variables in Fig. 2.2 is computable as follows:

$$\begin{aligned}
\mathbb{P}(A,B,C,D) &= \frac{E(A,B,C,D)}{\mathcal{Z}(\alpha,\beta,\gamma,\delta)} = \frac{E(A,B,CD)}{\sum_{\tilde{A},\tilde{B},\tilde{C},\tilde{D}} E(\tilde{A},\tilde{B},\tilde{C},\tilde{D})} \\
E(A,B,C,D) &= \Phi_A(A) \cdot exp(\alpha\Phi_{AC}(A,C)) \cdot exp(\beta\Phi_{AB}(A,B)) \cdots \\
\cdots & \Phi_{BC}(B,C) \cdot exp(\gamma\Phi_{CD}(C,D)) \cdot \Phi_{BD}(B,D) \cdot exp(\delta\Phi_B(B))
\end{aligned} \tag{2.14}$$

Graphical models are mainly used for performing belief propagation. Subset $\mathcal{O} = \{O_1, \cdots, O_f\} \subset \mathcal{V}$ is adopted for denoting the set of evidences: those variables in the net whose value become known. $\mathcal{O}$ can be dynamical or

[2] combinations having a null probability were omitted

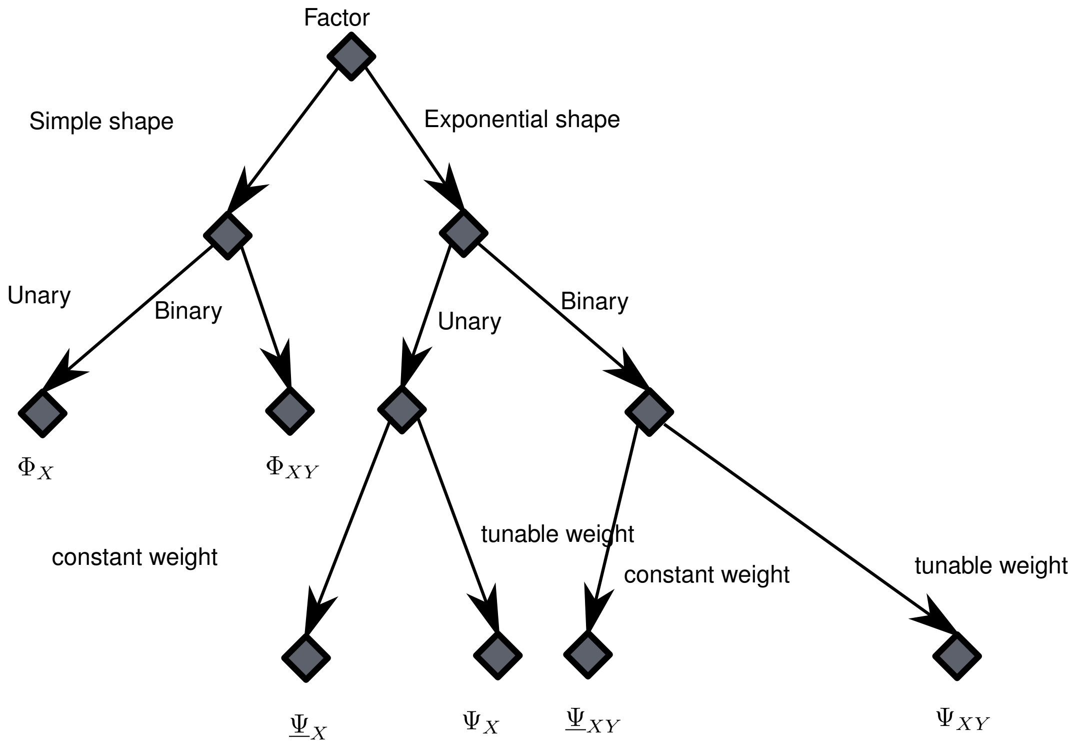


Figure 2.1 All the possible categories of factors, with the corresponding notation.

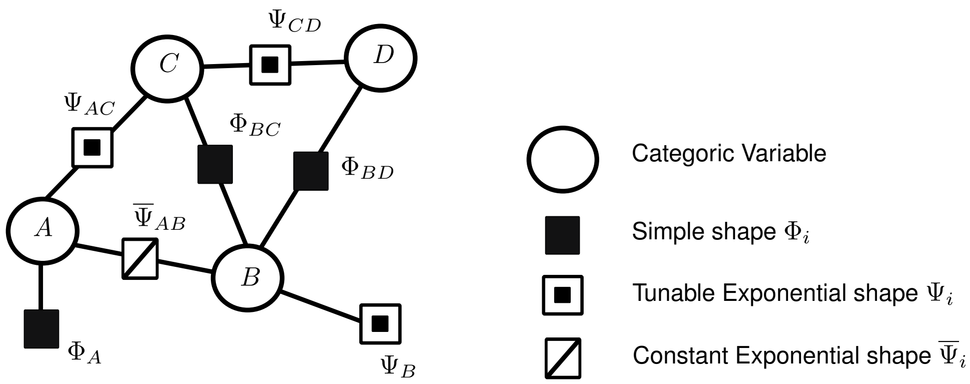


Figure 2.2 Example of graph: the legend of the right applies.

not. The hidden variables are contained in the complementary set $\mathcal{H} = \{H_1, \cdots, H_t\}$. Clearly $\mathcal{O} \cup \mathcal{H} = \mathcal{V}$ and $\mathcal{O} \cap \mathcal{H} = \emptyset$. H will be used for referring to the union of all the variables in the hidden set:

$$H = \bigcup_{i=1}^{t} H_i \tag{2.15}$$

while O is used for indicating the evidences:

$$O = \bigcup_{i=1}^{f} O_i \tag{2.16}$$

Knowing the joint probability distribution of variables in $\mathcal{V}$ (equation (2.6)) the conditional distribution of H w.r.t. O can be determined as follows:

$$\begin{aligned} \mathbb{P}(H = h|O = o) &= \frac{\mathbb{P}(H = h, O = o)}{\sum_{\forall \hat{h} \in Dom(H)} \mathbb{P}(H = \hat{h}, O = o)} \\ &= \frac{E(h, o)}{\sum_{\forall \hat{h} \in Dom(H)} E(\hat{h}, o)} = \frac{E(h, o)}{\mathcal{Z}(o)} \end{aligned} \tag{2.17}$$

The above computations are not actually done, since the number of combinations in the domain of $\mathcal{H}$ is huge also when considering a low-medium size graph. On the opposite, the marginal probability $\mathbb{P}(H_i = h_i|O = 0)$ of a single variable in $H_i \in \mathcal{H}$ is computationally tractable. Formally $\mathbb{P}(H_i = h_i|O = 0)$ is defined as follows:

$$\mathbb{P}(H_i = h_i|O = o) = \sum_{\forall \tilde{h} \in \{\mathcal{H} \setminus H_i\}} \mathbb{P}(H_i = h_i, \tilde{h}|O = o) \tag{2.18}$$

The above marginal distribution is essentially the conditional distribution of H_i w.r.t. O, no matter the other variables in $\mathcal{H}$.

A generic Random Field is a graphical model for which set $\mathcal{O}$ (and consequently $\mathcal{H}$) is dynamical: the set of observations as well the values assumed by the evidences may change during time. Random field are handled by class Random_Field. Conditional Random Field are Random Field for which set $\mathcal{O}$ must be decided once and cannot change after. Only the values of the evidences during time may change. Class Conditional_Random_Field is in charge of handling Conditional Random Field. Both Random Fields and Conditional Random Fields can be learnt knowing a training set, see Section 2.6. On the opposite, class Graph handles constant graphs: they are conceptually similar to Random Fields but learning is not possible. Indeed, all the Exponential Shape involved must be constant.

The rest of this Chapter is structured as follows. Section 2.2.2 will introduce the message passing algorithm, which is the pillar for performing belief propagation. Section 2.3 will expose the concept of maximum a posteriori estimation, useful when querying a graph, while Section 2.4 will address Gibbs sampling for producing a training set of a known model. Section 2.5 will present the concept of subgraph which is a useful way for computing the marginal probabilities of a sub group of variables in $\mathcal{H}$. Finally, 2.6 will discuss how the learning of a graphical model is done, with the aim of computing the weights of the Exponential shapes that are tunable.

2.2 Message Passing

Message passing is a powerful but conceptually simple algorithm adopted for propagating the belief across a net. Such a propagation is the starting point for performing many important operations, like computing the marginal distributions of single variables or obtaining sub graphs. Before detailing the steps involved in the message passing algorithm, let's start from an example of belief propagation. Without loss of generality we assume all the factors as Simple shapes.

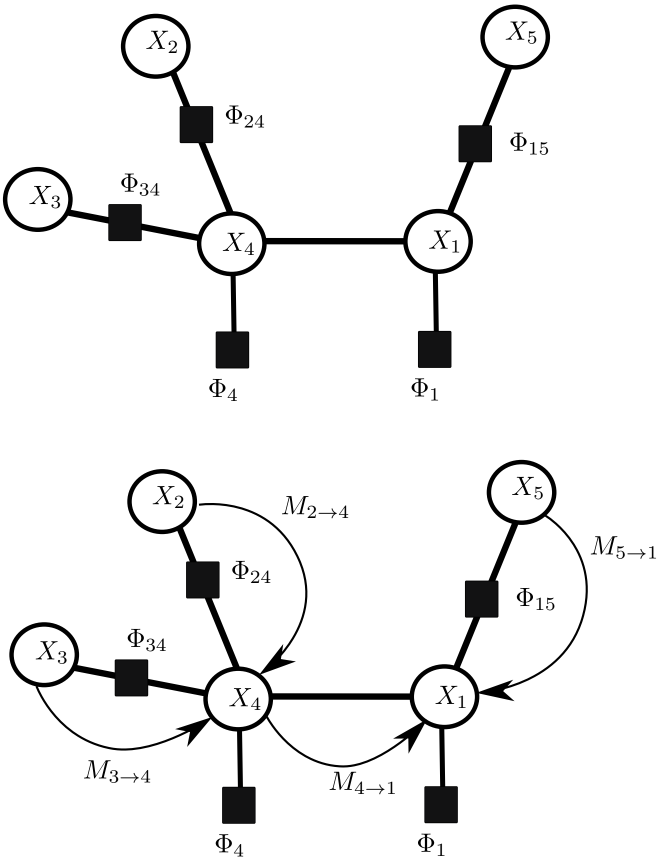


Figure 2.3 Example of graph adopted for explaining the message passing algorithm. Below are reported the messages to compute for obtaining the marginal probability of variable x_1

2.2.1 Belief propagation

Consider the graph reported in Figure 2.3. Supposing for the sake of simplicity that no evidences are available (i.e. $\mathcal{O} = \emptyset$). We are interested in computing $\mathbb{P}(X_1)$, i.e. the marginal probability of X_1. Recalling the definition introduced in the previous Section, the marginal probability is obtained by the following computation:

$$\mathbb{P}(x_1) = \sum_{\forall \tilde{x}_{2,3,4,5} \in \cup_{i=2}^{5} X_i} \mathbb{P}(x_1, \tilde{x}_{2,3,4,5}) \tag{2.19}$$

Simplifying the notation and getting rid of the normalization coefficient $\mathcal{Z}$ we can state the following:

$$\mathbb{P}(x_1) \propto \sum_{\tilde{x}_{2,3,4,5}} E(x_1, \tilde{x}_{2,3,4,5}) \tag{2.20}$$

Adopting the algebraic properties of the sums-products we can distribute the computations as follows:

$$\mathbb{P}(x_1) \propto \Phi_1(x_1) \sum_{\tilde{x}_5} \Phi_{15}(x_1, \tilde{x}_5) \sum_{\tilde{x}_4} \Phi_{14}(x_1, \tilde{x}_4) \Phi_4(\tilde{x}_4) \sum_{\tilde{x}_2} \Phi_{24}(\tilde{x}_{2,4}) \sum_{\tilde{x}_3} \Phi_{34}(\tilde{x}_{3,4}) \tag{2.21}$$

The first variable to marginalize can be $\tilde{x}_2$ or $\tilde{x}_3$, since they are involved in the last terms of the sums products. The 'messages' $M_{2\to 4}$, $M_{3\to 4}$ are defined as follows:

$$\begin{aligned} M_{2\to 4}(\tilde{x}_4) &= \sum_{\tilde{x}_2} \Phi_{24}(\tilde{x}_{2,4}) \\ M_{3\to 4}(\tilde{x}_4) &= \sum_{\tilde{x}_3} \Phi_{34}(\tilde{x}_{3,4}) \end{aligned} \tag{2.22}$$

Inserting $M_{2\to 4}$ and $M_{3\to 4}$ into equation (2.21) leads to:

$$\mathbb{P}(x_1) \propto \Phi_1(x_1) \sum_{\tilde{x}_5} \Phi_{15}(x_1, \tilde{x}_5) \sum_{\tilde{x}_4} \Phi_{14}(x_1, \tilde{x}_4) \Phi_4(\tilde{x}_4) M_{2\to 4}(\tilde{x}_4) M_{3\to 4}(\tilde{x}_4) \tag{2.23}$$

At this point the messages $M_{4\rightarrow 1}$ and $M_{5\rightarrow 1}$ can be computed in the following way:

$$M_{4\rightarrow 1(x_1)} = \sum_{\tilde{x}_4} \Phi_{14}(x_1, \tilde{x}_4)\Phi_4(\tilde{x}_4)M_{2\rightarrow 4}(\tilde{x}_4)M_{3\rightarrow 4}(\tilde{x}_4)$$

$$M_{5\rightarrow 1}(x_1) = \sum_{\tilde{x}_5} \Phi_{15}(x_1, \tilde{x}_5) \tag{2.24}$$

After inserting $M_{4\rightarrow 1}$ and $M_{5\rightarrow 1}$ into equation (2.23) we obtain:

$$\begin{aligned} \mathbb{P}(x_1) &\propto \Phi_1(x_1)M_{4\rightarrow 1}(x_1)M_{5\rightarrow 1}(x_1) \\ \mathbb{P}(x_1) &= \frac{\Phi_1(x_1)M_{4\rightarrow 1}(x_1)M_{5\rightarrow 1}(x_1)}{\sum_{\tilde{x}_1} \Phi_1(\tilde{x}_1)M_{4\rightarrow 1}(\tilde{x}_1)M_{5\rightarrow 1}(\tilde{x}_1)} \end{aligned} \tag{2.25}$$

which ends the computations. Messages are, in a certain sense, able to simplify the graph sending some information from an area of the graph to another one. Indeed, variables can be replace by messages, which can be treated as additional factors. Figure 2.3 resumes the computations exposed. Notice that the computation of $M_{4\rightarrow 1}$ must be done after computing the messages $M_{2\rightarrow 4}$ and $M_{3\rightarrow 4}$, while $M_{5\rightarrow 1}$ can be computed independently from all the others.

2.2.2 Message Passing

The aforementioned considerations can be extended to a general structured graph. Look at Figure 2.4: the computation of Message $M_{B\rightarrow A}$ can be performed only after having computed all the messages $M_{V_{1,\cdots,m}\rightarrow B}$, i.e. the messages incoming from all the neighbours of B a part from A. Clearly $M_{B\rightarrow A}$ is computed as follows:

$$\begin{aligned} M_{B\rightarrow A}(a) &= \sum_{\tilde{b}} \Phi_{AB}(a, \tilde{b})M_{V1\rightarrow B}(\tilde{b}) \cdot \cdots \cdot M_{Vm\rightarrow B}(\tilde{b}) \\ &= \sum_{\tilde{b}} \Phi_{AB}(a, \tilde{b}) \prod_{i=1}^{m} M_{V_i\rightarrow B}(\tilde{b}) \end{aligned} \tag{2.26}$$

Essentially, it's like having simplified the graph: we can append to A the message $M_{B\rightarrow A}(a)$ as it's a Simple shape, deleting factor Ψ_{AB} and all the other portions of the graph, see Figure 2.4. In turn, $M_{B\rightarrow A}(a)$ will be adopted for computing the message outgoing from A.
The above elimination is not actually done: all messages incoming to all nodes of a graph are computed by a derivation of the interface class I_belief_propagation_strategy and are stored to be used for subsequent queries. This is partially not true when considering the evidences. Indeed, when the values of the evidences are retrieved, variables in $\mathcal{O}$ are temporary deleted and replaced with messages, see Figure 2.5. Suppose variable C is connected to a variable A through a binary potential $\Phi_{AC}(A, C)$ and to variable B through $\Phi_{B,C}$. Suppose also that variable C is an evidence assuming a value equal to $\hat{c}$, then the messages sent to A and B can be computed independently as follows:

$$\begin{aligned} M_{C\rightarrow A}(a) &= \Phi_{AC}(a, \hat{c}) \\ M_{C\rightarrow B}(b) &= \Phi_{BC}(b, \hat{c}) \end{aligned} \tag{2.27}$$

Therefore all the variables that become evidences can be considered as leaves of the graph, sending messages to all the neighbouring nodes, possibly splitting an initial compact graph into many subgraphs, refer to Figure 2.5. Such computations are automatically handled by the library.

All the above considerations are valid when considering politree, i.e. graph without loops. Indeed, for these kind of graphs the message passing algorithm is able in a finite number of iterations to compute all the messages, see Figure 2.6. The same is not true when having loopy graphs (see Figure 2.7), since deadlocking situations arise: no further messages can be computed since for every nodes some incoming ones are missing. In such cases a variant of the message passing called loopy belief propagation can be adopted. Loopy belief propagation initializes all the messages to basic shapes having the values of the image all equal to 1 and then recomputes all the messages of all the variables till convergence.
You don't have to handle the latter aspect: when a belief propagation is performed, the library automatically chooses to deploy class Messagge_Passing or Loopy_belief_propagation , according to the structure of the graph for which the propagation is asked.

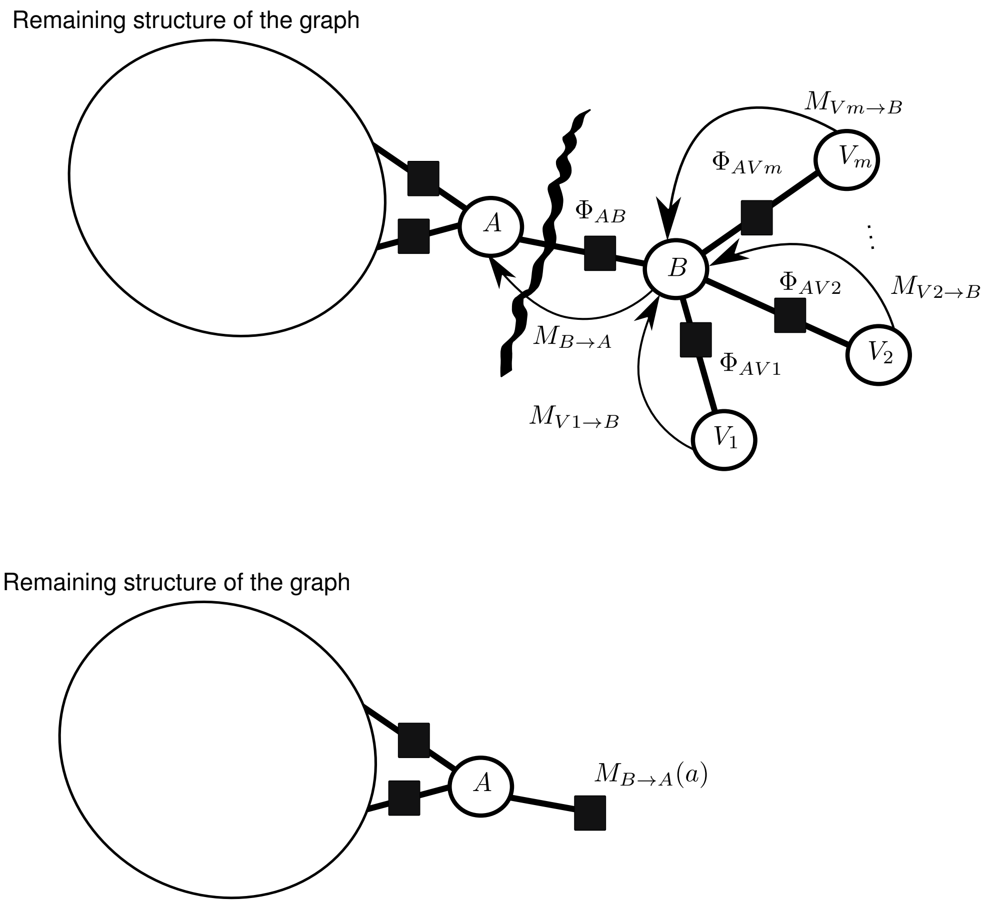


Figure 2.4 On the top the general mechanism involved in the message computation; on the bottom the simplification of the graph considering the computed message.

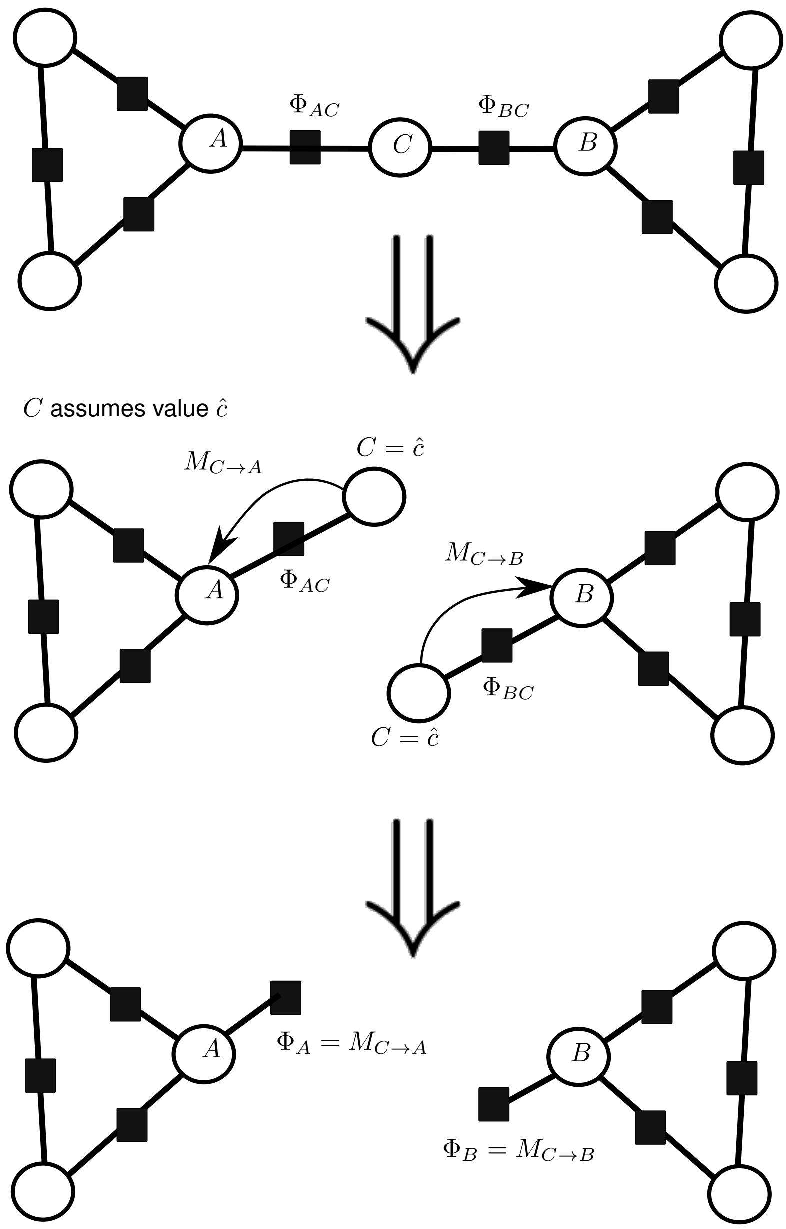


Figure 2.5 When variable C become an evidence, is temporary deleted from the graph, replaced by messages.

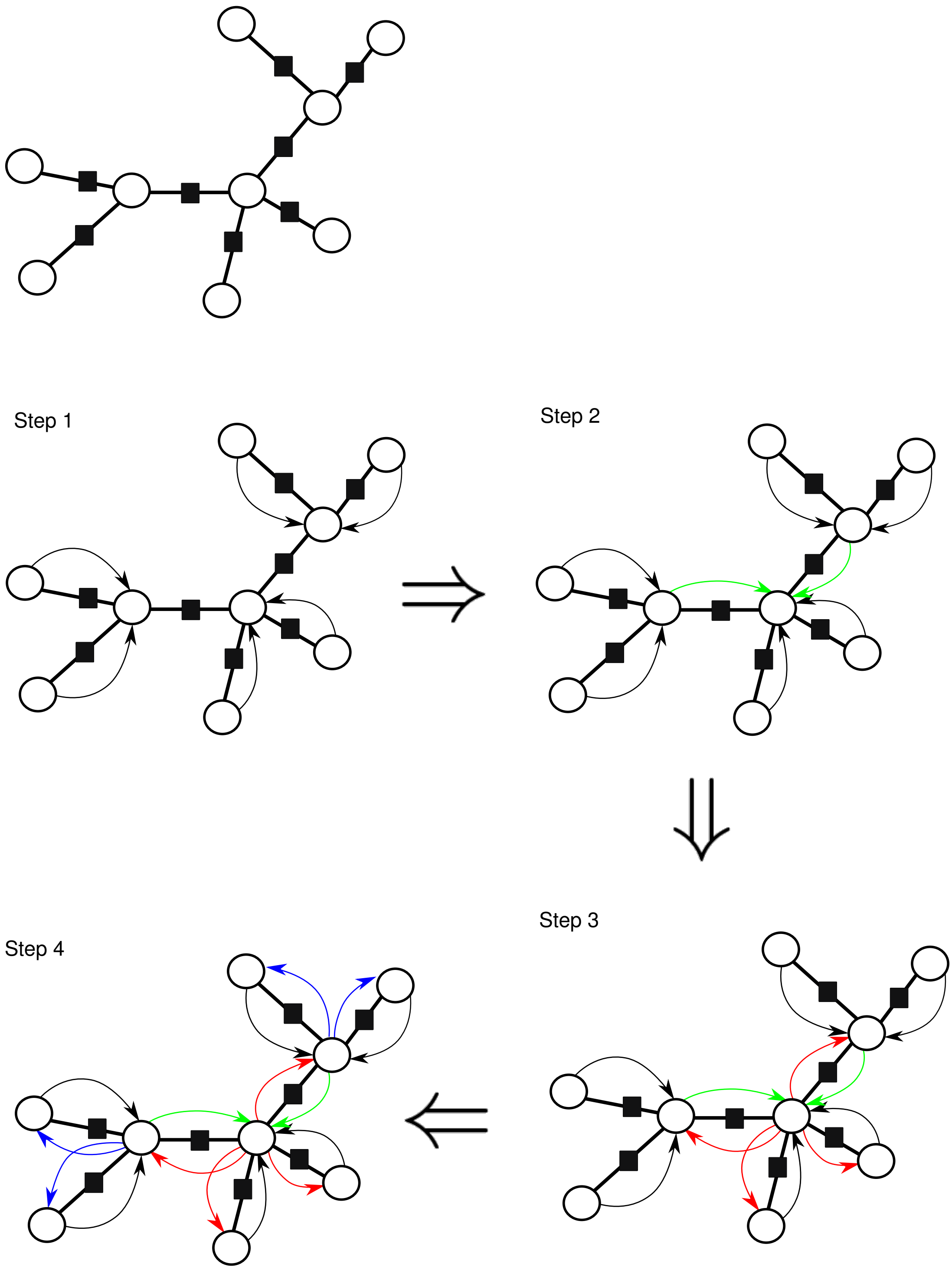


Figure 2.6 Steps involved for computing the messages of the politree represented at the top. The leaves are the first nodes for which the outgoing messages can be computed.

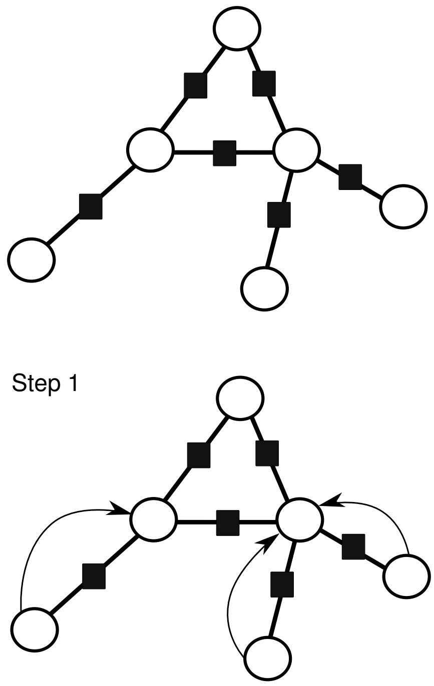


Figure 2.7 Steps involved for computing the messages on a loopy graph: after computing the messages outgoing from the leaves, a deadlock is reached since no further messages are computable.

2.3 Maximum a posteriori estimation

Suppose we are not interested in determining the marginal probability of a specific variable, but rather we want the combination in the hidden set $\mathcal{H}$ that maximises the probability $\mathbb{P}(H_{1,\cdots,n}|O)$. Clearly, we could try to compute the entire distribution $\mathbb{P}(H_{1,\cdots,n}|O)$ and then take the value of H maximising that distribution. However, this is not computationally possible since even for low medium size graphs the size of $Dom(\cup_{\forall H_i \in \mathcal{H}} H_i)$ can be huge.
Maximum a posteriori estimations solve this problem: the value maximising $\mathbb{P}(H_{1,\cdots,n}|O)$ is computed, without explicitly building the entire distribution $\mathbb{P}(H_{1,\cdots,n}|O)$. This is achieved by performing belief propagation with a slightly different version of the message passing algorithm presented in Section 2.2.2. Referring to Figure 2.4, the message to A is computed as follows when performing a maximum a posteriori estimation:

$$M_{B \to A}(a) = max_{\tilde{b}} \{ \Phi_{AB}(a, \tilde{b}) \prod_{i=1}^{m} M_{V_i \to B}(\tilde{b}) \} \} \tag{2.28}$$

Essentially, the summation in equation (2.26) is replaced with the max operator. After all messages are computed, the estimation $h_{MAP} = \{h_{1MAP}, h_{2MAP}, \cdots \}$ is obtained by considering for every variable in $\mathcal{H}$ the value maximising:

$$h_{iMAP} = argmax \{ \Phi_{Hi}(h_{iMAP}) \prod_{k=1}^{L} M_k(h_{iMAP}) \} \tag{2.29}$$

where $M_{1,\cdots,L}$ refer to all the messages incoming to H_i. To be precise, this procedure is not guaranteed to return the value actually maximising $\mathbb{P}(H_{1,\cdots,n}|O)$, but at least a strong local maximum is obtained.
At this point it is worthy to clarify that the combination $h_{MAP} = \{h_{1MAP}, h_{2MAP}, \cdots \}$ could not be obtained by simply assuming for every H_i the realization maximising the marginal distribution:

$$h_{MAP} \neq \{ argmax(\mathbb{P}(h_1)), \cdots, argmax(\mathbb{P}(h_n)) \} \tag{2.30}$$

This is due to the fact that $\mathbb{P}(H_{1,\cdots,n}|O)$ is a joint probability distribution, while the marginals $\mathbb{P}(H_i)$ are not. For better understanding this aspect consider the graph reported in Figure 2.8, with the potentials Φ_{XA}, Φ_{AB} and Φ_{YB} having the images defined in table 2.5. Suppose discovering that $X = 0$ and $Y = 1$. Then, performing the standard message passing algorithm explained in the previous Section we obtain the messages reported in Figure

	b_0	b_1
a_0	2	0
a_1	0	2

	x_0	x_1
a_0	1	0.1
a_1	0.1	1

	y_0	y_1
b_0	1	0.1
b_1	0.1	1

Table 2.5 Factors involved in the graph of Figure 2.8.

A	B	$E(A, B, X = 0, Y = 1)$
0	0	0.2
0	1	0
1	0	0
1	1	0.2

Table 2.6 Factors involved in the graph of Figure 2.8.

2.8. Clearly individual marginals for A and B would be equal to:

$$\mathbb{P}(A) = \begin{pmatrix} \mathbb{P}(A=0) \\ \mathbb{P}(A=1) \end{pmatrix} = \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix}$$
$$\mathbb{P}(B) = \begin{pmatrix} \mathbb{P}(B=0) \\ \mathbb{P}(B=1) \end{pmatrix} = \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix} \tag{2.31}$$

Therefore, all the combinations $\{A = 0, B = 0\}$, $\{A = 0, B = 1\}$, $\{A = 1, B = 0\}$, $\{A = 1, B = 1\}$ maximise $\mathbb{P}(A, B|O)$. However, it easy to prove that $E(A, B, X, Y)$ assumes the values reported in table 2.6. Therefore, the combinations actually maximising the joint distribution $\mathbb{P}(A, B|O)$ are $\{A = 0, B = 0\}$ and $\{A = 1, B = 1\}$, leading to a different result.
Maximum a posteriori estimation can be performing invoking MAP_on_Hidden_set **??** on a particular derivation of class Node_factory.

2.4 Gibbs sampling

Gibbs sampling is a Monte Carlo method for obtaining samples from a joint distribution of variables $X_{1,\cdots,m}$, without explicitly compute that distribution. Indeed, Gibbs sampling is an iterative method which requires every time to determine the conditional distribution of a single variable X_i w.r.t to all the others in the group.
More formally the algorithm starts with an initial combination of values $\{x^1_{1,\cdots,m}\}$ for the variable $\cup_{i=\{1,\cdots,m\}} X_i$. At every iteration, all the values of that combination are recomputed. At the j^{th} iteration the value of x^{j+1}_k for the subsequent iteration is obtaining by sampling from the following marginal distribution:

$$x^{j+1}_k \sim \mathbb{P}(x_k | x^j_{\{1,\cdots,m\}\setminus k}) \tag{2.32}$$

After an initial transient, the samples cumulated during the iterations can be considered as drawn from the joint distribution involving group $X_{1,\cdots,m}$.
This algorithm can be easily applied to graphical model. Indeed the methodologies exposed in Section 2.2.2 can be applied for determining the conditional distribution of a single variable $H_i \in \mathcal{H}$ w.r.t all the others (as well the evidences in $\mathcal{O}$), assuming all variables in $\mathcal{H} \setminus H_i$ as additional observations and computing the marginal probability of H_i. Gibbs_Sampling_on_Hidden_set **??** is in charge of performing Gibbs sampling on a generic graph.

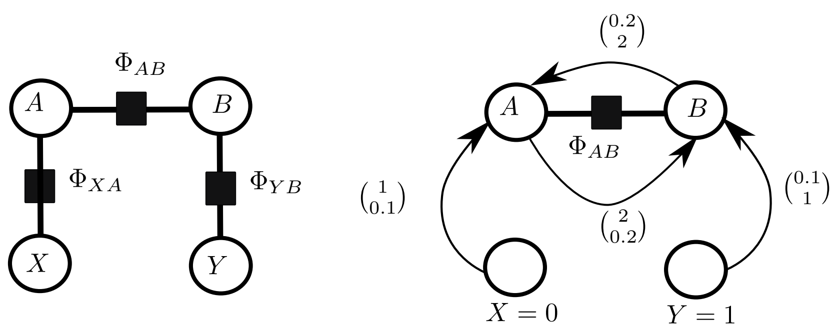


Figure 2.8 Example of graph adopted. When the evidences are retrieved, the messages computed by making use of the message passing algorithm are reported below.

2.5 Sub graphs

As explained in Section 2.2.2, the marginal probability of a variable $H_i \in \mathcal{H}$ can be efficiently computed by considering the messages produced by the message passing algorithm. The same messages can be also used for performing graph reduction, with the aim to model the joint probability distribution of a subset of variables $\{H_1, H_2, H_3\} \subset \mathcal{H}$, i.e. $\mathbb{P}(H_{1,2,3}|O)$. The latter quantity is the marginal probability of the subset of variables of interest.

The aim of message passing is essentially to simplify the graph, condensing all the belief information into the messages. Such property is exploited for computing sub graphs. Without loss of generality assume from now on $\mathcal{O} = \emptyset$. Consider the graph in Figure 2.9 and suppose we are interested in modelling $\mathbb{P}(A, B, C)$, no matter the values of the other variables. After computing all the messages exploiting message passing, the sub graph reported in Figure 2.9 is the one modelling $\mathbb{P}(A, B, C)$. Actually, that sub graph is a graphical model itself, for which all the properties exposed so far hold. For example the energy function E is computable as follows:

$$E(A = a, B = b, C = c) = \Phi_{AB}(a, b)\Phi_{BC}(b, c)\Phi_{AC}(a, c)M_{X\rightarrow A}(a)M_{Y\rightarrow B}(b) \tag{2.33}$$

while the joint probability of A, B and C can be computed in this way:

$$\mathbb{P}(A = a, B = b, C = c) = \frac{E(a, b, c)}{\sum_{\forall \tilde{a}, \tilde{b}, \mathbf{c}} E(\tilde{a}, \tilde{b}, \tilde{c})} \tag{2.34}$$

Notice that in this case the graph is significantly smaller than the originating one, implying that the above computations can be performed in an acceptable time.

Also Gibbs sampling can be applied to a reduced graph, producing samples drawn from the marginal probability $\mathbb{P}(A, B, C)$.

The reduction described so far is always possible when considering a subset of variables forming a connected subportion of the original graph, i.e. after reduction there must be a unique sub structure. For instance, variables X and Y of the graph in Figure 2.10 do not respect the latter specification, meaning that it is not possible to build a sub graph involving X and Y.

The class in charge of handling graph reduction is Node_factory::_SubGraph.

2.6 Learning

The aim of learning is to determine the optimal values for the w (equation (2.13)) of all the tunable potentials (see Section 2.1) Ψ. To this aim two cases must be distinguished:

- Learning must be performed for a Graph or a Random_Field: see Section 2.6.1
- Learning must be performed for a Conditional_Random_Field: see Section 2.6.2

No matter the case, the population of tunable weights will be indicated with W:

$$W = \{w_1, \cdots, w_D\} \tag{2.35}$$

w_i will refer to the i^{th} free parameter of the model. Learning is internally handled by EFG, exploiting class I_Trainer.

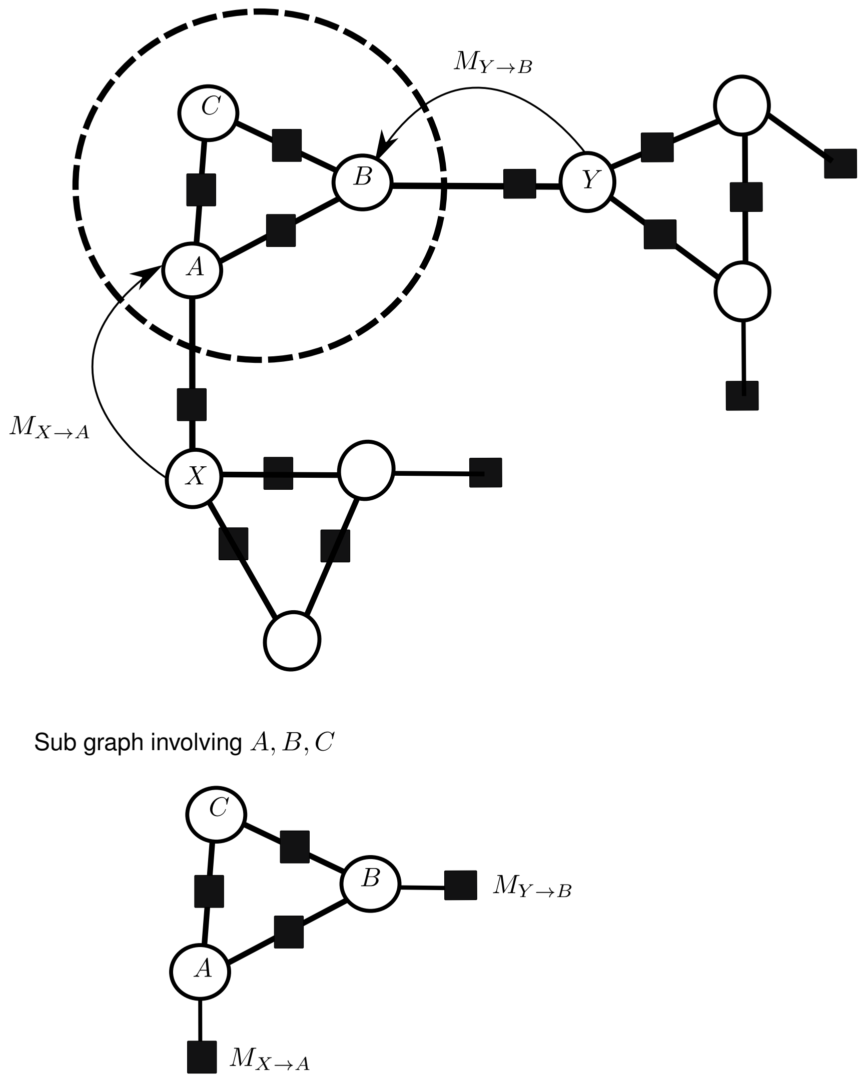


Figure 2.9 Example of graph reduction.

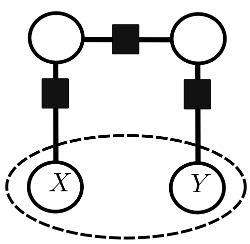


Figure 2.10 Example of a subset of variables for which the graph reduction is not possible.

2.6.1 Learning of unconditioned model

For the purpose of learning, we assume $\mathcal{O} = \emptyset$. Learning considers a training set $T = \{t_1, \cdots, t_N\}$ made of realizations of the joint distribution correlating all the variables in $\mathcal{V}$, no matter the fact that they are involved in tunable or non tunable potentials. As exposed in Section 2.1, if W is known, the probability of a combination t_j can be evaluated as follows:

$$\mathbb{P}(t_j) = \frac{E(t_j, W)}{\mathcal{Z}(W)} \tag{2.36}$$

At this point we can observe that the energy function is the product of two main factors: one depending from t_j and W and the other depending only upon t_j representing the contribution of all the non tunable potentials (Simple shapes and fixed Exponential shapes, see Section 2.1):

$$\begin{aligned} E(t_j, W) &= exp\big(w_1 \Phi_1(t_j)\big) \cdot \cdots \cdot exp\big(w_D \Phi_D(t_j)\big) \cdot E_0(t_j) \\ &= exp\big(\sum_{i=1}^{D} w_i \Phi_i(t_j)\big) \cdot E_0(t_j) \end{aligned} \tag{2.37}$$

The likelihood function L can be defined as follows:

$$L = \prod_{t_j \in T} \mathbb{P}(t_j) \tag{2.38}$$

passing to the logarithmic likelihood and dividing by the training set size N we obtain:

$$\begin{aligned} J = \frac{log(L)}{N} &= \sum_{t_j \in T} \frac{log(\mathbb{P}(t_j))}{N} \\ &= \sum_{t_j \in T} \frac{log(E(t_j, W)) - log(\mathcal{Z}(W)}{N} \\ &= \frac{1}{N} \sum_{t_j \in T} log(E(t_j, W)) - log(\mathcal{Z}(W)) \\ &= \frac{1}{N} \sum_{t_j \in T} \big(\sum_{i=1}^{D} w_i \Phi_i(t_j)\big) - log(\mathcal{Z}(W)) + \cdots \\ &+ \frac{1}{N} \sum_{t_j \in T} log(E_0(t_j)) \end{aligned} \tag{2.39}$$

The aim of learning is to find the value of W maximising J. This is done iteratively, exploiting a gradient descend approach. The computations to perform for evaluating the gradient $\frac{\partial J}{\partial W}$ will be exposed in the following part of this Section. Notice that in equation (2.39), term $\sum_{t_j \in T} log(E_0(t_j))$ is constant and consequently will be not considered for computing the gradient of J. Equation (2.39) can be rewritten as follows:

$$\begin{aligned} J &= \alpha(T, W) - \beta(W) \\ \alpha &= \frac{1}{N} \sum_{t_j \in T} \big(\sum_{i=1}^{D} w_i \Phi_i(t_j)\big) && (2.40) \\ \beta &= log(\mathcal{Z}(W)) && (2.41) \end{aligned}$$

α is influenced by T, while the same is not valid for β.

2.6.1.1 Gradient of α

By the analysis of the equation (2.40) it is clear that:

$$\frac{\partial \alpha}{\partial w_i} = \frac{1}{N} \sum_{t_j \in T} \Phi_i(t_j) \tag{2.42}$$

2.6.1.2 Gradient of β

The computation of $\frac{\partial \beta}{\partial w_i}$ requires to manipulate a little bit equation (2.41). Firstly the derivative of the logarithm must be computed:

$$\frac{\partial \beta}{\partial w_i} = \frac{1}{\mathcal{Z}} \frac{\partial \mathcal{Z}}{\partial w_i} \tag{2.43}$$

The normalizing coefficient $\mathcal{Z}$ is made of the following terms (see also equation (2.6)):

$$\mathcal{Z}(W) = \sum_{\tilde{V} \in \bigcup_{i=1}^{p} V_i} \left(exp(\sum_{i=1}^{D} w_i \Phi_i(\tilde{V})) \cdot E_0(\tilde{V}) \right) \tag{2.44}$$

Introducing equation (2.44) into (2.43) leads to:

$$\begin{aligned}
\frac{\partial \beta}{\partial w_i} &= \frac{1}{\mathcal{Z}} \frac{\partial}{\partial w_i} \left(\sum_{\tilde{V}} exp(\sum_{i=1}^{D} w_i \Phi_i(\tilde{V})) E_0(\tilde{V}) \right) \\
&= \frac{1}{\mathcal{Z}} \sum_{\tilde{V}} \frac{\partial}{\partial w_i} \left(exp(\sum_{i=1}^{D} w_i \Phi_i(\tilde{V})) \right) E_0(\tilde{V}) \\
&= \frac{1}{\mathcal{Z}} \sum_{\tilde{V}} exp(\sum_{i=1}^{D} w_i \Phi_i(\tilde{V})) E_0(\tilde{V}) \Phi_i(\tilde{V}) \\
&= \sum_{\tilde{V}} \frac{exp(\sum_{i=1}^{D} w_i \Phi_i(\tilde{V})) E_0(\tilde{V})}{\mathcal{Z}} \Phi_i(\tilde{V}) \\
&= \sum_{\tilde{V}} \frac{E(\tilde{V})}{\mathcal{Z}} \Phi_i(\tilde{V}) \\
&= \sum_{\tilde{V}} \mathbb{P}(\tilde{V}) \Phi_i(\tilde{V}) \qquad (2.45)
\end{aligned}$$

Last term in the above equations can be further elaborated. Assume that the variables involved in potential Φ_j are $V_{1,2}$, then:

$$\begin{aligned}
\frac{\partial \beta}{\partial w_i} &= \sum_{\tilde{V}} \mathbb{P}(\tilde{V}) \Phi_i(\tilde{V}) \\
&= \sum_{\tilde{V}_{1,2}} \Phi_i(\tilde{V}_{1,2}) \sum_{\tilde{V}_{3,4,\dots}} \mathbb{P}(\tilde{V}_{1,2,3,4,\dots}) \\
&= \sum_{\tilde{V}_{1,2}} \Phi_i(\tilde{V}_{1,2}) \mathbb{P}(\tilde{V}_{1,2}) \qquad (2.46)
\end{aligned}$$

where $\mathbb{P}(\tilde{V}_{1,2})$ is the marginal probability (see the initial part of Section 2.1) of the variables involved in the potential Φ_i, which can be easily computable by considering the sub graph containing only V_1 and V_2 as variables (see Section 2.5). Notice that in case Φ_i is a unary potential the same holds, considering the marginal distribution of the single variable involved by Φ_i:

$$\frac{\partial \beta}{\partial w_i} = \sum_{\forall \tilde{V}_1} \Phi_i(\tilde{V}_1) \mathbb{P}(\tilde{V}_1) \tag{2.47}$$

which can be easily obtained through the message passing algorithm (Section 2.2.2).

After all the manipulations performed, the gradient $\frac{\partial J}{\partial w_i}$ has the following compact expression:

$$\frac{\partial J}{\partial w_i} = \frac{1}{N} \sum_{j=1}^{N} \Phi_i(D_j^i) - \sum_{\tilde{D}^i} \mathbb{P}(\tilde{D}^i) \Phi_i(\tilde{D}^i) \tag{2.48}$$

2.6.2 Learning of conditioned model

For such models leaning is more demanding as will be exposed. Recalling the definition provided in the final part of Section 2.1, Conditional Random Fields are graphs for which the set of observations $\mathcal{O}$ is fixed. The training set T is made of realizations of both $\mathcal{H}$ and $\mathcal{O}$:

$$\begin{aligned} T &= \{t_1, \cdots, t_N\} \\ &= \{\{h_1, o_1\}, \cdots, \{h_N, o_N\}\} \end{aligned} \tag{2.49}$$

We recall , equation (2.17), that the conditional probability of the hidden variables w.r.t. the observed ones is defined as follows:

$$\begin{aligned} \mathbb{P}(h_j, o_j) &= \frac{E(h_j, o_j, W)}{\mathcal{Z}(o_j, W)} \\ E(h_j, o_j, W) &= exp\big(\sum_{i=1}^{D} w_i \Phi_i(h_j, o_j)\big) E_0(h_j, o_j) \\ \mathcal{Z}(o_j, W) &= \sum_{\tilde{h}} E(\tilde{h}, o_j, W) \end{aligned} \tag{2.50}$$

The aim of learning is to maximise a likelihood unction L defined in this case as follows:

$$L = \prod_{h_j \in T} \mathbb{P}(h_j | o_j) \tag{2.51}$$

Passing to the logarithms and dividing by the training set size we obtain the following objective function J:

$$\begin{aligned} J &= \frac{log(L)}{N} \\ &= \frac{1}{N} \sum_{h_j, o_j \in T} log(E(h_j, o_j, W)) - \frac{1}{N} \sum_{h_j, o_j \in T} log(Z(o_j, W)) \\ &= \frac{1}{N} \sum_{h_j, o_j \in T} \big(\sum_{i=1}^{D} w_i \Phi_i(h_j, o_j)\big) - \frac{1}{N} \sum_{h_j, o_j \in T} log(Z(o_j, W)) \\ &+ \frac{1}{N} \sum_{h_j, o_j \in T} log(E_0(h_j, o_j)) \end{aligned} \tag{2.52}$$

Neglecting E_0 which not depends upon W, equation (2.52) can be rewritten as follows:

$$\begin{aligned} J &= \alpha(T, W) - \beta(T, W) \\ \alpha(T, W) &= \frac{1}{N} \sum_{h_j, o_j} \big(\sum_{i=1}^{D} w_i \Phi_i(h_j, o_j)\big) \\ \beta(T, W) &= \frac{1}{N} \sum_{o_j} log(\mathcal{Z}(o_j, W)) \end{aligned} \tag{2.53}$$

At this point, an important remark must be done: differently from the β defined in equation (2.41), $\beta(T, W)$ of conditioned model is a function of the training set. The latter observation has an important consequence: when performing learning of unconditioned model, belief propagation (i.e. the computation of the messages through message passing with the aim of computing the marginal probabilities of the groups of variables involved in the factor of the model) must be performed once for every iteration of the gradient descend; on the opposite when considering conditioned model, belief propagation must be performed at every iteration for every element of the training set, see equation (2.57). This makes the learning of conditioned models much more computationally demanding. This price is paid in order to not model the correlation among the observations [3], which can be interesting for many applications. The computation of $\frac{\partial \alpha}{\partial w_i}$ is analogous to the one of non conditioned model, equation (2.42).

[3] that can be highly correlated

2.6.2.1 Gradient of β

Following the same approach in Section 2.6.1.2, the gradient of β can be computed as follows:

$$\begin{aligned}
\frac{\partial \beta}{\partial w_i} &= \frac{1}{N}\sum_{j=1}^{N}\frac{\partial log(\mathcal{Z}(o_j, W))}{\partial w_i} \\
&= \frac{1}{N}\sum_{j=1}^{N}\frac{1}{\mathcal{Z}(o_j)}\frac{\partial \mathcal{Z}(o_j, W)}{\partial w_i} \\
&= \frac{1}{N}\sum_{j=1}^{N}\frac{\partial}{\partial w_i}\Big(\sum_{\tilde{h}} exp(\sum_{i=1}^{D} w_i \Phi_i(\tilde{h}, o_j))E_0(\tilde{h}, o_j)\Big) \\
&= \frac{1}{N}\sum_{j=1}^{N}\sum_{\tilde{h}}\Big(exp(\sum_{i=1}^{D} w_i \Phi_j(\tilde{h}, o_j))E_0(\tilde{h}, o_j)\Phi_i(\tilde{h}, o_j)\Big) \\
&= \frac{1}{N}\sum_{j=1}^{N}\sum_{\tilde{h}}\frac{E(\tilde{h}, o_j, W)}{\mathcal{Z}(o_1)}\Phi_i(\tilde{h}, o_j) \\
&= \frac{1}{N}\sum_{j=1}^{N}\sum_{\tilde{h}}\mathbb{P}(\tilde{h}|o_j)\Phi_i(\tilde{h}, o_j)
\end{aligned} \tag{2.54}$$

Suppose the variables involved in the factor Φ_j are $\tilde{h}_{1,2}$, then:

$$\begin{aligned}
\frac{\partial \beta}{\partial w_i} &= \frac{1}{N}\sum_{j=1}^{N}\sum_{\tilde{h}}\mathbb{P}(\tilde{h}|o_j)\Phi_i(\tilde{h}, o_j) \\
&= \frac{1}{N}\sum_{j=1}^{N}\sum_{\tilde{h}_{1,2}}\Phi_i(\tilde{h}_{1,2})\sum_{\tilde{h}_{3,4,\dots}}\mathbb{P}(\tilde{h}_{1,2,3,4,\dots}|o_j) \\
&= \frac{1}{N}\sum_{j=1}^{N}\sum_{\tilde{h}_{1,2}}\Phi_i(\tilde{h}_{1,2})\mathbb{P}(\tilde{h}_{1,2}|o_j)
\end{aligned} \tag{2.55}$$

where $\mathbb{P}(\tilde{h}_{1,2}|o_j)$ is the conditioned marginal probability of group $\tilde{h}_{1,2}$ w.r.t. the observations o_j.

Grouping all the simplifications we obtain:

$$\frac{\partial J}{\partial w_i} = \frac{1}{N}\sum_{j=1}^{N}\Phi_i(h_j, o_j) - \frac{1}{N}\sum_{j=1}^{N}\Big(\sum_{\tilde{h}_{1,2}}\mathbb{P}(\tilde{h}_{1,2}|o_j)\Phi_i(\tilde{h}_{1,2})\Big) \tag{2.56}$$

Generalizing:

$$\frac{\partial J}{\partial w_i} = \frac{1}{N}\sum_{j=1}^{N}\Phi_i(D_j^i, o_j) - \frac{1}{N}\sum_{j=1}^{N}\Big(\sum_{\tilde{D}^i}\mathbb{P}(\tilde{D}^i|o_j)\Phi_i(\tilde{D}^i, o_j)\Big) \tag{2.57}$$

2.6.3 Learning of modular structure

Suppose to have a modular structure made of repeating units as for example the graph in Figure 2.11. Every single unit has the same population of potentials and we would like to enforce this fact when performing learning. In particular we'll have some sets of Exponential shape sharing the same weight w_1 (see Figure 2.11). Motivated by this example, we included in the library the possibility to specify that a potential must share its weight with another one. Then, learning is done consistently with the aforementioned specification.

2.6.3.1 Gradient of α

Considering the model in Figure 2.11, the α part of J (equation (2.40)) can be computed as follows:

$$\alpha = \frac{1}{N}\sum_{t_j}\big(w_1\Phi_1(a_{1j},b_{1j}) + w_1\Phi_2(a_{2j},b_{2j}) + w_1\Phi_3(a_{3j},b_{3j}) + \cdots + \\ \cdots + \sum_{i=2}^{D} w_i\Phi_i(t_j)\big) \tag{2.58}$$

which leads to:

$$\frac{\partial\alpha}{\partial w_1} = \frac{1}{N}\sum_{t_j}\big(\Phi_1(a_{1j},b_{1j}) + \Phi_2(a_{2j},b_{2j}) + \Phi_3(a_{3j},b_{3j})\big) \tag{2.59}$$

2.6.3.2 Gradient of β

Regarding the β part of J we can write what follows:

$$\begin{aligned}
\frac{\partial\beta}{\partial w_1} &= \frac{1}{Z}\frac{\partial Z}{\partial w_1} \\
&= \frac{1}{Z}\frac{\partial}{\partial w_1}\Big(\sum_{\tilde{V}}\Big(exp\big(w_1(\Psi_1(a_{1j},b_{1j}) + \cdots \\
\cdots &+ \Psi_2(a_{2j},b_{2j}) + \Psi_3(a_{3j},b_{3j})) + \sum_{i=2}^{D} w_i\Phi_i(\tilde{V})\big)E_0(\tilde{V})\Big)\Big) \\
&= \sum_{\tilde{V}}\mathbb{P}(\tilde{V})(\Phi_1(\tilde{a}_1,\tilde{b}_1) + \Phi_2(\tilde{a}_2,\tilde{b}_2) + \Phi_3(\tilde{a}_3,\tilde{b}_3)) \\
&= \sum_{\tilde{V}}\mathbb{P}(\tilde{V})\Phi_1(\tilde{a}_1,\tilde{b}_1) + \sum_{\tilde{V}}\mathbb{P}(\tilde{V})\Phi_2(\tilde{a}_2,\tilde{b}_2) + \sum_{\tilde{V}}\mathbb{P}(\tilde{V})\Phi_3(\tilde{a}_3,\tilde{b}_3) \\
&= \sum_{\tilde{A}_1,\tilde{B}_1}\mathbb{P}(\tilde{A}_1,\tilde{B}_1)\Phi_1(\tilde{A}_1,\tilde{B}_1) + \sum_{\tilde{A}_2,\tilde{B}_2}\mathbb{P}(\tilde{A}_2,\tilde{B}_2)\Phi_2(\tilde{A}_2,\tilde{B}_2) + \cdots \\
\cdots &+ \sum_{\tilde{A}_3,\tilde{B}_3}\mathbb{P}(\tilde{A}_3,\tilde{B}_3)\Phi_3(\tilde{A}_3,\tilde{B}_3)
\end{aligned} \tag{2.60}$$

Notice that the gradient $\frac{\partial J}{\partial w_1}$ is the summation of three terms: the ones that would have been obtained considering separately the three potentials in which w_1 is involved (equation (2.48)):

$$\begin{aligned}
\frac{\partial J}{\partial w_1} &= \frac{1}{N}\sum_{j=1}^{N}\Phi_1(a_i^1,b_i^1) - \sum_{\tilde{a}^1,\tilde{b}^1}\mathbb{P}(\tilde{a}^1,\tilde{b}^1)\Phi_1(\tilde{a}^1,\tilde{b}^1) + \cdots \\
&+ \frac{1}{N}\sum_{j=1}^{N}\Phi_2(a_i^2,b_i^2) - \sum_{\tilde{a}^2,\tilde{b}^2}\mathbb{P}(\tilde{a}^2,\tilde{b}^2)\Phi_2(\tilde{a}^2,\tilde{b}^2) + \cdots \\
&+ \frac{1}{N}\sum_{j=1}^{N}\Phi_3(a_i^3,b_i^3) - \sum_{\tilde{a}^3,\tilde{b}^3}\mathbb{P}(\tilde{a}^3,\tilde{b}^3)\Phi_3(\tilde{a}^3,\tilde{b}^3) +
\end{aligned} \tag{2.61}$$

The above result has a general validity, also considering conditioned graphs.

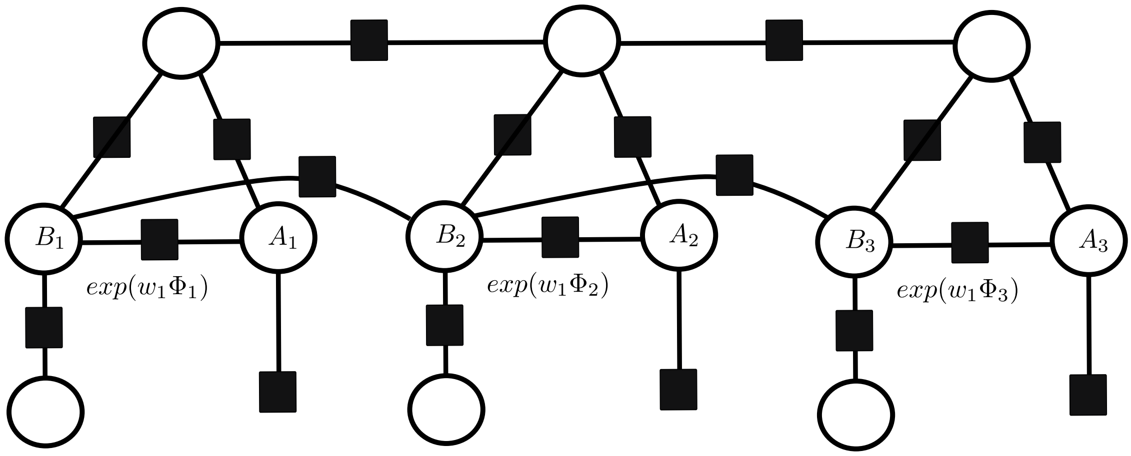


Figure 2.11 Example of modular structure: weight w_1 is simultaneously involved into potentials Φ_1, Φ_2 and Φ_3.

Chapter 3

Files representing factor graphs

The aim of this Section is to expose how to build graphical models from XML files describing their structures. In particular, the syntax of such an XML will be clarified. XMl files can be passed as input for the constructor of Graph **??**, Random_Field **??** and Conditional_Random_Field **??**. Figure 3.1 visually explains the structure of a valid XML. Essentially two kind of tags must be incorporated:

- Variable: describes the information related to a variable present in the graph. There must a tag of this kind for every variable constituting the model. Fields description:
  - name: is a string indicating the name of this variable.
  - Size: is the size of the variable, i.e. the size of Dom, see Section 2.1.
  - flag[optional] : is a flag that can assume two possible values, 'O' or 'H' according to the fact that this variable is in set $\mathcal{O}$ (Section 2.1) or not respectively. When non specifying this flag 'H' is assumed.
- Potential: describes the information related to a unary or a binary potential present in the graph (see Section 2.1). Fields description:
  - var: the name of the first variable involved.
  - var[optional]: the name of the second variable involved. Is omitted when considering unary potentials, while is mandatory when a binary potentials is described by this tag.
  - weight[optional]: when specifying an Exponential shape (Section 2.1) it must be present for indicating the value of the weight w (equation (2.13)). When omitting, the potential is assumed as a Simple shape one.
  - tunability[optional]: it is a flag for specifying whether the weight of this Exponential shape is tunable or not (see Section 2.1). Is ignored in case weight is omitted. It can assumes two possible values, 'Y' or 'N' according to the fact that the weight involved is tunable or not respectively. When weight is specified and tunability is omitted, a value equal to 'Y' is assumed.
- Share[optional]: you must specify this sub tag when the containing Exponential shape shares its weight with another potential in the model. Sub fields var are exploited for specifying the variables involved by the potential whose weight is to share. If weight is omitted in the containing Potential tag, this sub tag is ignored, even though the value assigned to weight is ignored since it is shared with another potential. The potential sharing its weight must be clearly an Exponential shape, otherwise the sharing directive is ignored.

  The following components are exclusive: only one of them can be specified in a Potential tag and at the same time at least one must be present.

  - Correlation: it can assume two possible values, 'T' or 'F'. When 'T' is passed, this potential is assumed to be a simple shape correlating shape (see **??**), otherwise when passing 'F' a simple anti correlating shape is assumed (see **??**). It is invalid in case this Potential is a unary one. In case weight was specified, an Exponential shape is built, wrapping a simple correlating or anti-correlating shape.

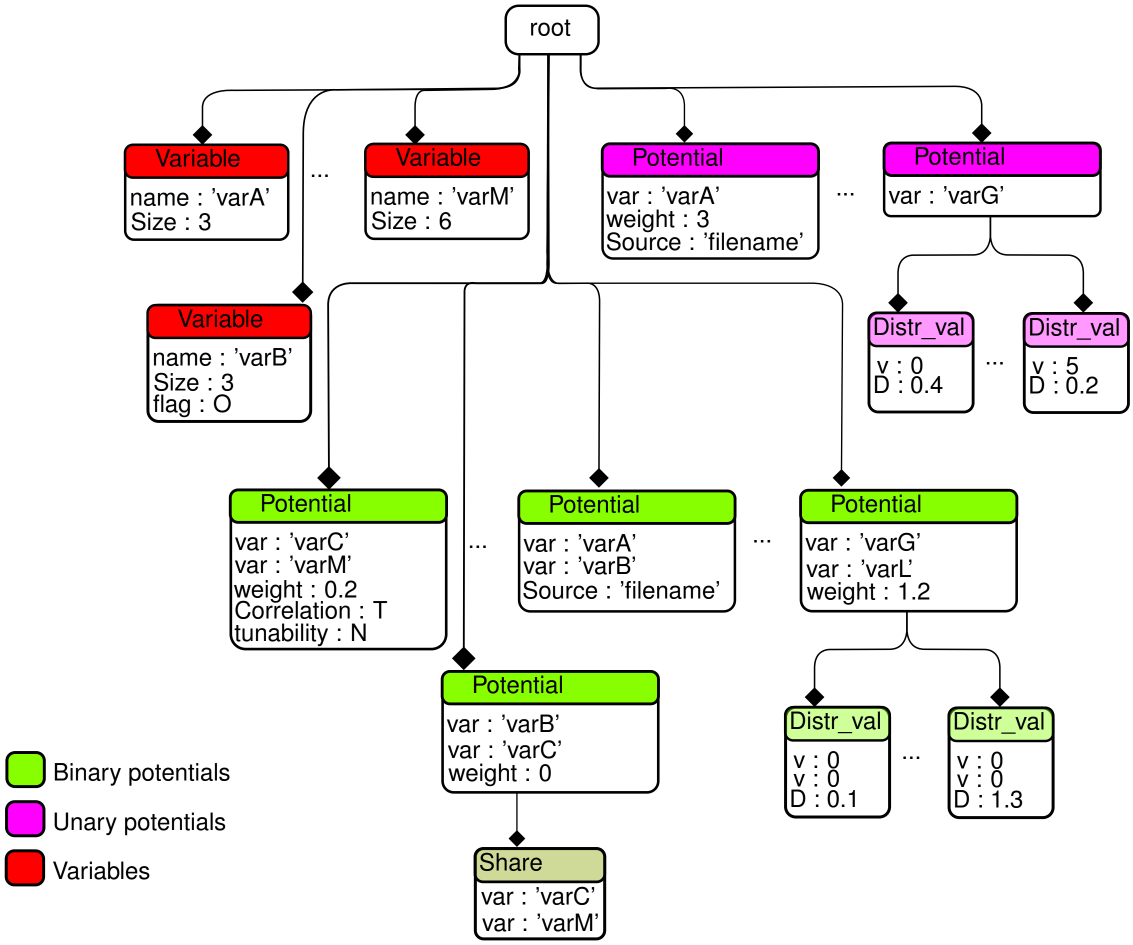


Figure 3.1 The structure of the XML describing a graphical model.

– Source: it is the location of a textual file describing the values of the distribution characterizing this potential. Rows of this file contain the values charactering the image of the potential. Combinations not specified are assumed to have an image value equal to 0. Clearly the number of values charactering the distribution must be consistent with the number of specified var fields. In case weight was specified, an Exponential shape is built, wrapping the Simple shape whose values are specified in the aforementioned file. For instance, the potential Φ_b of Section 2.1 would have been described by a file containing the following rows:

$$\begin{matrix} 0 & 0 & 1 \\ 0 & 1 & 4 \\ 1 & 1 & 1 \\ 2 & 2 & 5 \\ 2 & 4 & 1 \end{matrix} \tag{3.1}$$

– Set of sub tags Distr_val: is a set of nested tags describing the distribution of the this potential. Similarly to Source, every element use fields v for describing the combination, while D is used for specifying the value assumed by the distribution. For example the potential Φ_b of Section 2.1 would have been described by the syntax reported in Figure 3.2. In case weight was specified, an Exponential shape is built, wrapping the Simple shape whose distribution is specified by the aforementioned sub tags.

```
<Potential ···>
        <Distr_val "v"=0 "v"=0 "D"=1>
        <Distr_val "v"=0 "v"=1 "D"=4>
        <Distr_val "v"=1 "v"=1 "D"=1>
        <Distr_val "v"=2 "v"=2 "D"=5>
        <Distr_val "v"=2 "v"=4 "D"=1>
</Potential>
```

Figure 3.2 Syntax to adopt for describing the potential Φ_b of Section 2.1, using a population of Distr_val sub tags.

Chapter 4

Samples

4.1 Sample 01: Potential handling

The aim of this series of examples is mainly to show how to handle the creation of variables and factors.

4.1.1 part 01

Part 01 creates a shape factor Φ_{AB}, involving the pair of variables A and B. Both that variables have a domain size equal to 4, i.e. $Dom(A) = \{a_0 = 0, a_1 = 1, a_2 = 2, a_3 = 3\}$ and $Dom(B) = \{b_0 = 0, b_1 = 1, b_2 = 2, b_3 = 3\}$. The generic value in the image Φ_{AB} is equal to:

$$\Phi_{AB}(A = a, B = b) = a + 2 \cdot b \tag{4.1}$$

Table 4.1 reports the entire image of Φ_{AB}.

4.1.2 part 02

Part 02 considers a ternary correlating factor $\Phi_{C\ V123}$, involving variables V_1, V_2 and V_3, each having a domain size equal to 3. Ternary factors cannot be part of a graph, but it is anyway possible to build them using class Potential_Shape. The values in the image of $\Phi_{C\ V123}$ are all 0, except for those combination for which V_1, V_2 and V_3 assume the same value ($0, 1$ or 2) and in such cases, the image is equal to 1.
The same example builds at a second stage a ternary anti-correlating factor $\Phi_{A\ V123}$. The values in the image of $\Phi_{A\ V123}$ are all 1, except for those combination for which V_1, V_2 and V_3 assume the same value ($0, 1$ or 2) and in such cases the image is equal to 1.
When considering a graph having only $\Psi_{C,\,V123}(\Phi_{C\ V123} \cdot w)$ as a factor, the ripartition function Z is equal to:

$$Z = (4^3 - 4) + 4 \cdot exp(w) \tag{4.2}$$

	$b_0 = 0$	$b_1 = 1$	$b_2 = 2$	$b_3 = 3$
$a_0 = 0$	0	2	4	6
$a_1 = 1$	1	3	5	7
$a_2 = 2$	2	4	6	8
$a_3 = 3$	3	5	7	9

Table 4.1 The values in the image of Φ_{AB}.

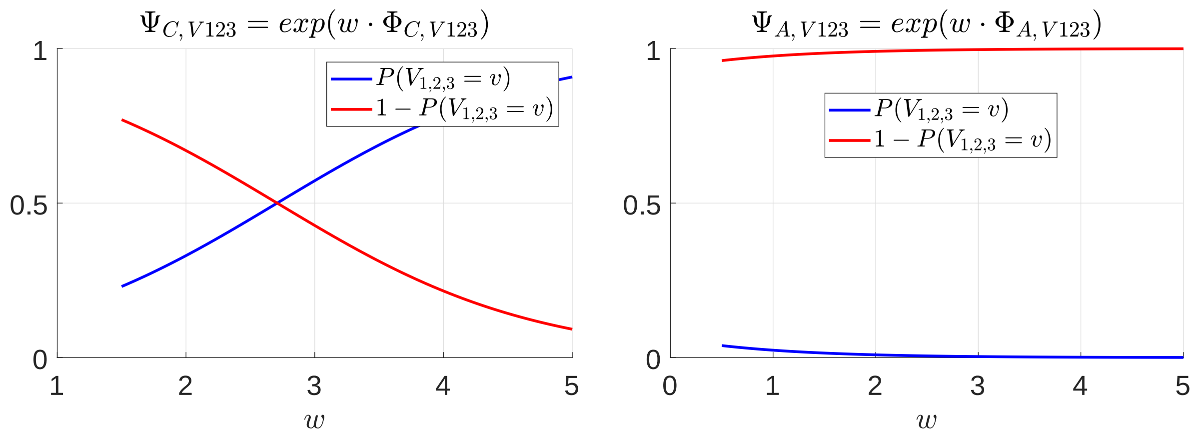


Figure 4.1 The probability $\mathbb{P}(V_1 = v, V_2 = v, V_3 = v|w)$ and its complement, when considering a ternary correlating factor, on the left, and an anti-correlating one, on the right.

The probability to have as a realization a combination with the same values is equal to:

$$\begin{aligned}
\mathbb{P}(V_1 = v, V_2 = v, V_3 = v) &= \sum_{i=0}^{3} \mathbb{P}(V_1 = i, V_2 = i, V_3 = i) && (4.3)\\
&= \sum_{i=0}^{3} \frac{\Psi_{C\ V123}(i,i,i)}{Z} && (4.4)\\
&= 4 \cdot \frac{\Psi_{C\ V123}(0,0,0)}{Z} && (4.5)\\
&= \frac{4 \cdot exp(w)}{(4^3 - 4) + 4 \cdot exp(w)} && (4.6)
\end{aligned}$$

The value assumed by $\mathbb{P}(V_1 = v, V_2 = v, V_3 = v|w)$ is reported in Figure 4.1, together with the complementary probability $1 - \mathbb{P}(V_1 = v, V_2 = v, V_3 = v|w)$.
When considering a graph having only $\Psi_{A,V123} = wexp(\Phi_{A\ V123})$ as factor, the ripartition function Z is equal to:

$$Z = (4^3 - 4) \cdot exp(w) + 4 \quad (4.7)$$

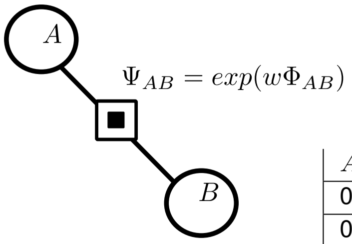
The probability to have as a realization a combination with the same values is equal to:

$$\begin{aligned}
\mathbb{P}(V_1 = v, V_2 = v, V_3 = v) &= \sum_{i=0}^{3} \mathbb{P}(V_1 = i, V_2 = i, V_3 = i) && (4.8)\\
&= \sum_{i=0}^{3} \frac{\Psi_{A\ V123}(i,i,i)}{Z} && (4.9)\\
&= 4 \cdot \frac{\Psi_{A\ V123}(0,0,0)}{Z} && (4.10)\\
&= \frac{4}{(4^3 - 4) \cdot exp(w) + 4} && (4.11)
\end{aligned}$$

The value assumed by $\mathbb{P}(V_1 = v, V_2 = v, V_3 = v|w)$ is reported in Figure 4.1, together with its complement $1 - \mathbb{P}(V_1 = v, V_2 = v, V_3 = v|w)$. Indeed, when the variables are correlated, i.e. they share $\Psi_{C\ V123}$, the probability $\mathbb{P}(V_1 = v, V_2 = v, V_3 = v|w)$ is big. Moreover, the more w is high (i.e. the more the variables are correlated), the more the latter probability is big. On the opposite, when the variables are anti-correlated, the opposite situation arises.

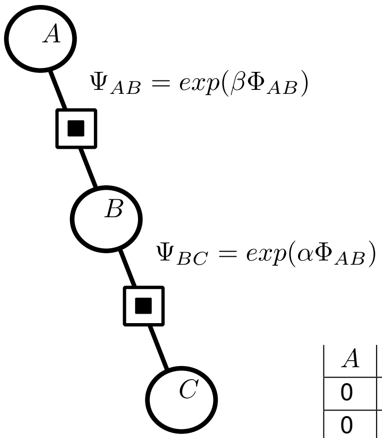
4.2 Sample 02: Belief propagation, part A

The aim of this series of examples is to show how to perform probabilistic queries on factor graphs.



A	B	$\Psi_{AB}(A,B) = E(A,B)$
0	0	exp(w)
0	1	1
1	0	1
1	1	exp(w)

Figure 4.2 On the left the graph considered in this example, while on the right the image of factor Ψ_{AB}. Since that potential is the only one present in the graph, the values in the image of Ψ_{AB} are also the ones assume by the energy function E.



A	B	$\Psi_{AB}(A,B)$
0	0	exp(β)
0	1	1
1	0	1
1	1	exp(β)

B	C	$\Psi_{BC}(B,C)$
0	0	exp(α)
0	1	1
1	0	1
1	1	exp(α)

Figure 4.3 On the left the graph considered in this example, while on the right the images of factor Ψ_{AB} and Ψ_{BC} having, respectively, a weight equal to β and α.

4.2.1 part 01

This example creates a graph having a single binary exponential shape Ψ_{AB}, see Figure 4.2, with a A and B having a Dom size equal to 2. $\Psi_{AB} = exp(w\Phi_{AB})$, where Φ_{AB} is a simple correlating factor. The image of Ψ_{AB} is reported in the right part of Figure 4.2.
Variable B is considered as an evidence, whose value is equal, for the first part of the example, to 0, while a value of 1 is assumed in the second part. The probability of A conditioned to B, is equal to (see equation (2.17)):

$$\mathbb{P}(A = a|B = 0) = \frac{E(A = a, B = 0)}{E(A = 0, B = 0) + E(A = 1, B = 0)} \Rightarrow \begin{bmatrix} \mathbb{P}(A = 0|B = 0) = \frac{exp(w)}{1+exp(w)} \\ \mathbb{P}(A = 1|B = 0) = \frac{1}{1+exp(w)} \end{bmatrix} \tag{4.12}$$

$$\mathbb{P}(A = a|B = 1) = \frac{E(A = a, B = 1)}{E(A = 0, B = 1) + E(A = 1, B = 1)} \Rightarrow \begin{bmatrix} \mathbb{P}(A = 0|B = 1) = \frac{1}{1+exp(w)} \\ \mathbb{P}(A = 1|B = 1) = \frac{exp(w)}{1+exp(w)} \end{bmatrix} \tag{4.13}$$

4.2.2 part 02

A slightly more complex graph, made of two exponential correlating factors Ψ_{BC} and Ψ_{AB}, is built in this sample. The considered graph is reported in Figure 4.3. The two involved factors have two different weights, α and β: the resulting image sets are reported in the right part of Figure 4.3.
In the first part, $C = 1$ is assumed as evidence and the marginal probabilities of A and B conditioned to C are computed. They are compared with the theoretical results, obtained by applying the message passing algorithm

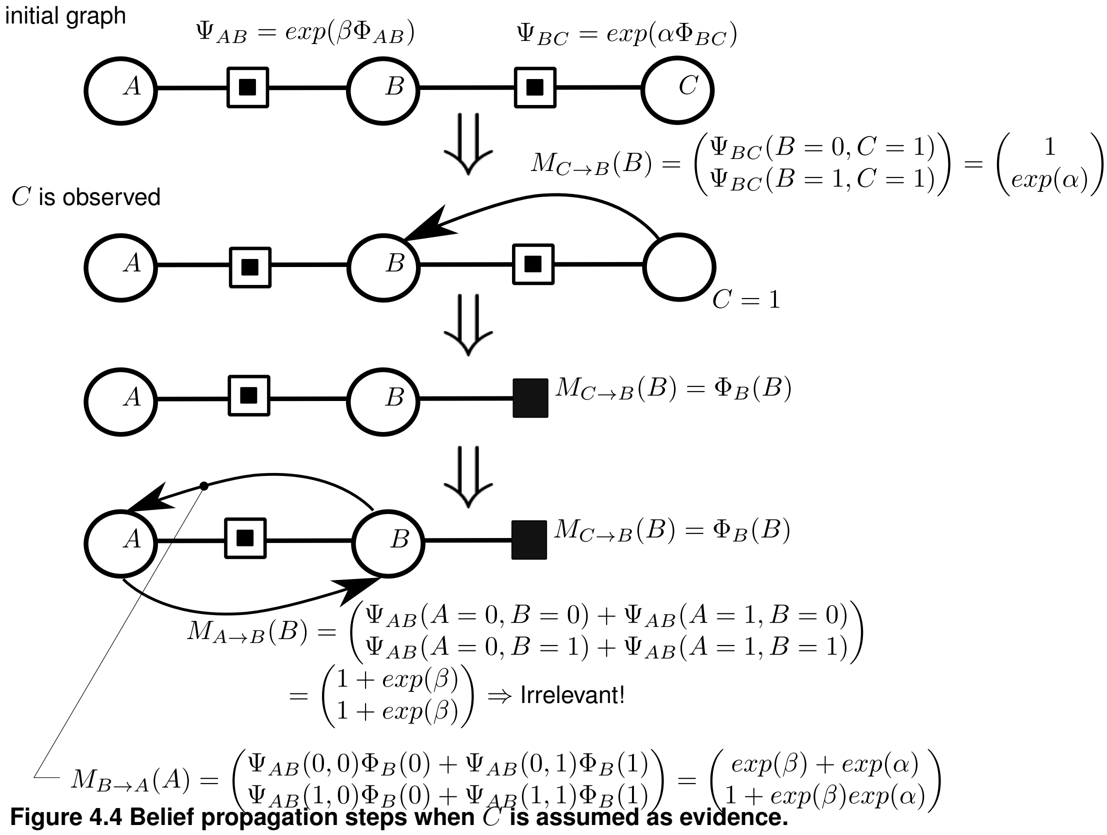


Figure 4.4 Belief propagation steps when C is assumed as evidence.

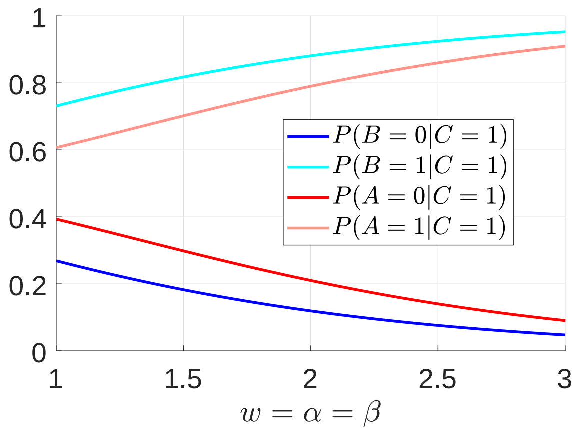


Figure 4.5 The marginals of variables B and A, when having a $C = 1$ as evidence of the graph reported in Figure 4.4.

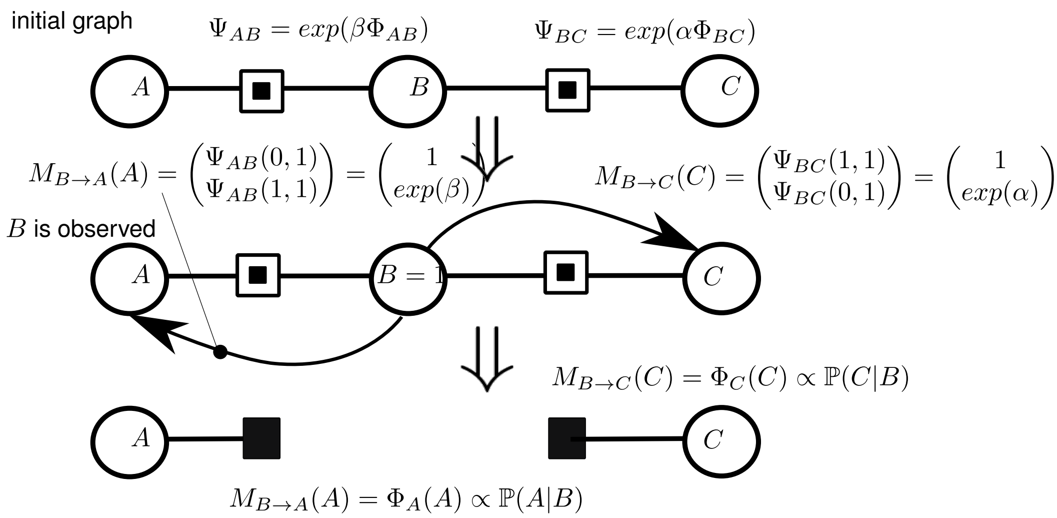


Figure 4.6 Belief propagation steps when B is assumed as evidence.

(Section 2.2.2), whose steps are here detailed [1]. The message passing steps are summarized in Figure 4.4. After having computed all the messages, it is clear that the marginal probabilities are equal to:

$$\mathbb{P}(A|C=1) = \frac{1}{Z}M_{B\to A}(A) = \frac{1}{Z}\begin{bmatrix} exp(\alpha) + exp(\beta) \\ 1 + exp(\alpha)\cdot exp(\beta)\end{bmatrix} \tag{4.14}$$

$$\mathbb{P}(B|C=1) = \frac{1}{Z}\Phi_B(B)\cdot M_{A\to B}(B) = \frac{1}{Z}\Phi_B(B) = \frac{1}{Z}\begin{bmatrix} 1 \\ exp(\alpha)\end{bmatrix} \tag{4.15}$$

Figure 4.5 shows the values assumed by the marginals when varying the coefficients α and β. As can be seen, the more A, B and C are correlated (i.e. the more α and β are big) the more $\mathbb{P}(B = 1|C = 1)$ and $\mathbb{P}(A = 1|C = 1)$ are big. Notice also that when assuming $\alpha = \beta$, $\mathbb{P}(B = 1|C = 1)$ is always bigger than $\mathbb{P}(A = 1|C = 1)$. This is intuitively explained by the fact that C is directly connected to B, while A is indirectly connected to C, through B.
In the second part, $B = 1$ is assumed as evidence and the marginal probabilities of A and C conditioned to B are computed. The theoretical results can be computed again considering the message passing, whoe steps are reported in Figure 4.6. The marginal probabilities are in this case equal to:

$$\mathbb{P}(A|B=1) = \frac{1}{Z}\Phi_A(A) = \frac{1}{Z}\begin{bmatrix} 1 \\ exp(\beta)\end{bmatrix} \tag{4.16}$$

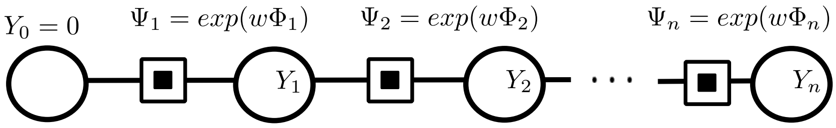
$$\mathbb{P}(C|B=1) = \frac{1}{Z}\Phi_C(C) = \frac{1}{Z}\begin{bmatrix} 1 \\ exp(\alpha)\end{bmatrix} \tag{4.17}$$

4.2.3 part 03

In this sample, a linear chain of variables $Y_{0,1,2,\cdots,n}$ is considered. All variables in the chain have the same Dom size and all the factors $\Psi_{1,\cdots,n}$, Figure 4.7, are simple exponential correlating factors. The image of the generic factor Ψ_i is reported in the right part of Figure 4.7.
The evidence $Y_0 = 0$ is assumed and the marginals of the last variable in the chain Y_n, i.e. the one furthest to Y_0, are computed. Figure 4.8 reports the probability $\mathbb{P}(Y_n = 0|Y_0 = 0)$, when varying the chain size, as well the domain size of the variables. As can be seen, the more the chain is longer, the lower is the aforementioned probability, as Y_n is more and more indirectly correlated to Y_0. Similar considerations hold for the domain size.

4.3 Sample 03: Belief propagation, part B

The aim of this series of examples is to show how to perform probabilistic queries on articulated complex graphs.



$\Psi_i(Y_{i-1}, Y_i)$	$Y_i = 0$	$Y_i = 1$	$\cdots$	$Y_i = m$
$Y_{i-1} = 0$	$exp(w)$	1	$\cdots$	1
$Y_{i-1} = 1$	1	$exp(w)$	$\cdots$	1
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$Y_{i-1} = m$	1	1	$\cdots$	$exp(w)$

Figure 4.7 On the top the chain considered in this example, while on the bottom the image of the generic factor $\Psi_i(Y_{i-1}, Y_i)$.

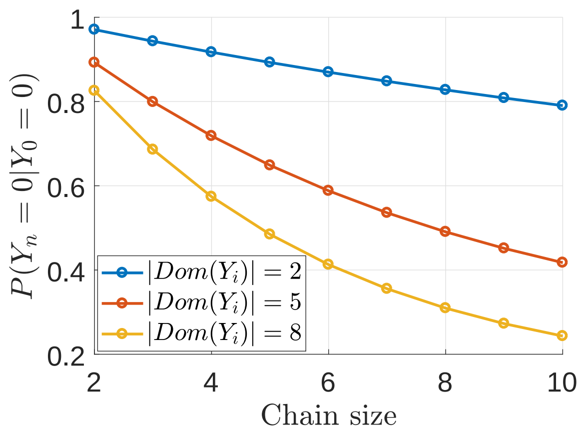


Figure 4.8 Marginal probability of Y_n when varying the chain size of the structure presented in Figure 4.7. w is assumed equal to 3.5.

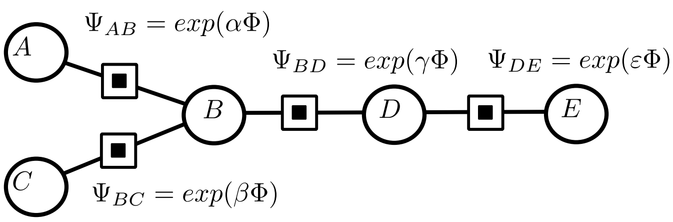


Figure 4.9 The factor graph considered by part 01.

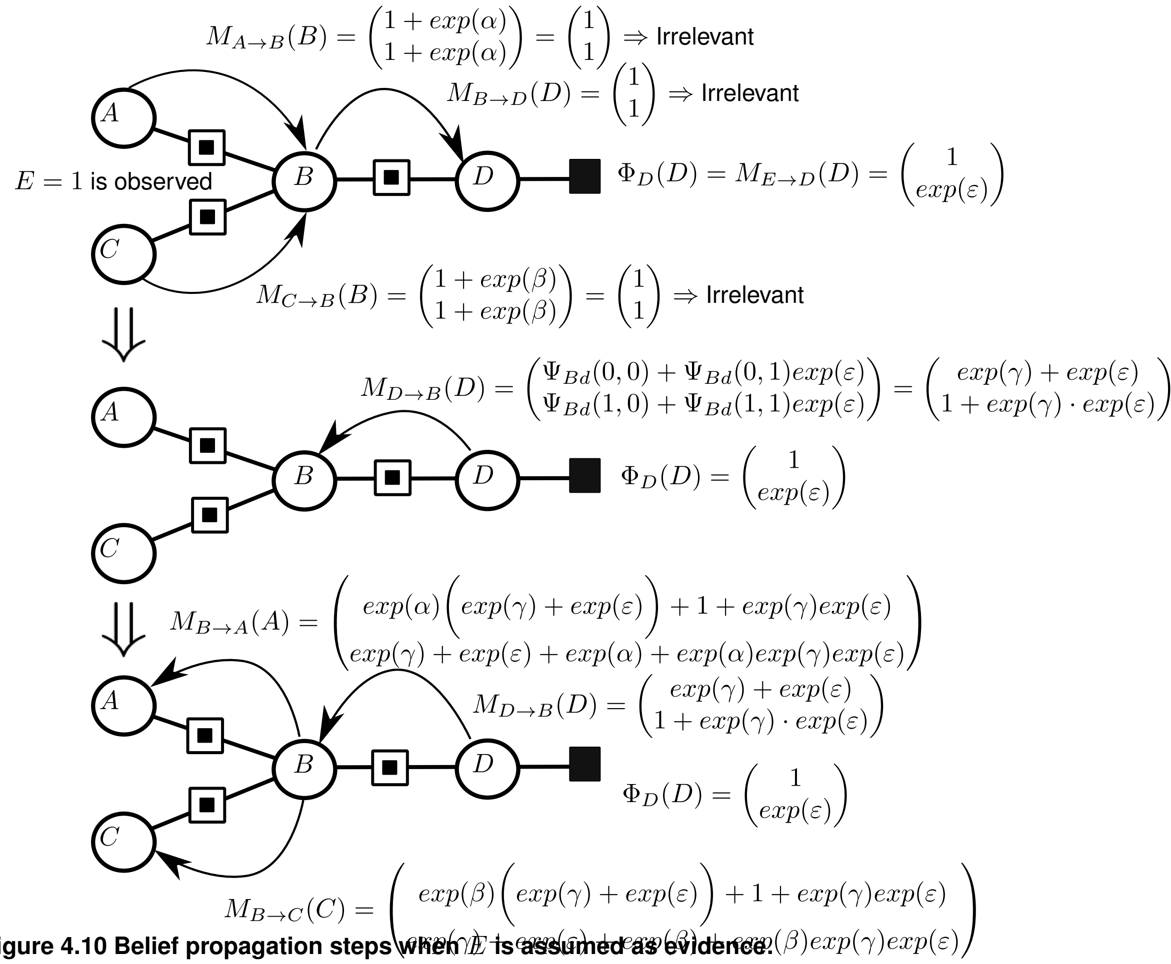


Figure 4.10 Belief propagation steps when E is assumed as evidence.

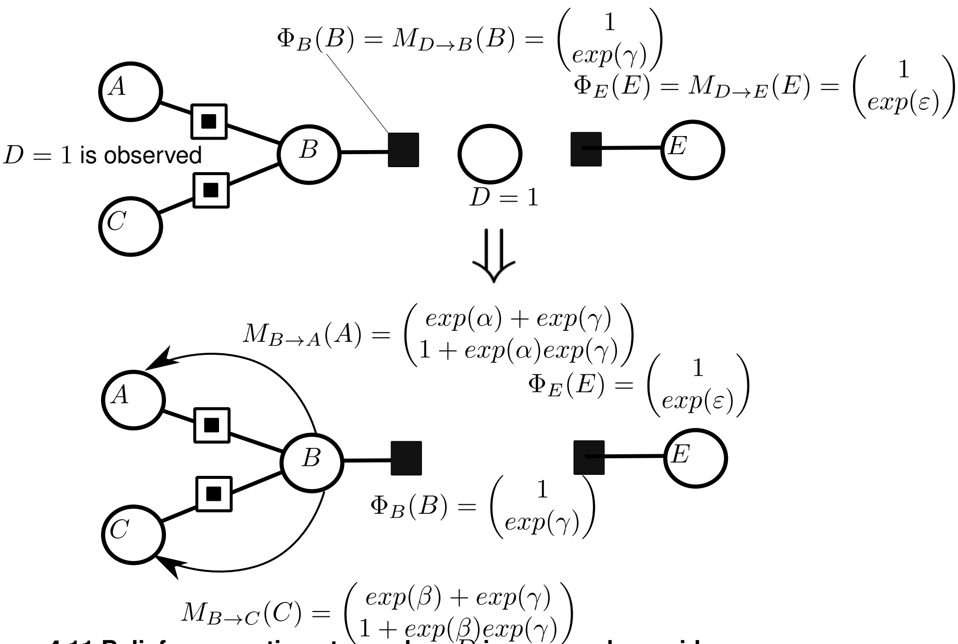


Figure 4.11 Belief propagation steps when D is assumed as evidence.

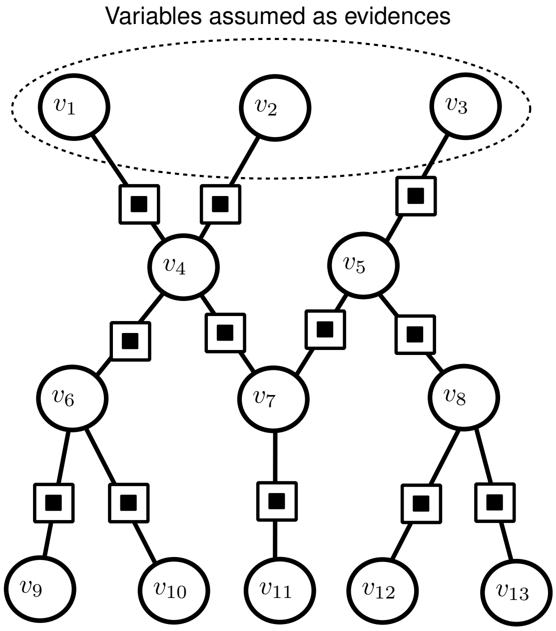


Figure 4.12 The factor graph considered by part 02.

4.3.1 part 01

Part 01 considers a graph made of 5 variables with a Dom size equal to 2 and some simple exponential correlating factors, having different weights. The graph is reported in Figure 4.9, together with the weights of factor $\alpha, \beta, \gamma, \varepsilon$.
At first stage, the evidence $E = 1$ is assumed and the marginal probabilities of the other variables are computed with the message passing, whose steps are summarized in Figure 4.10. After the convergence of the message passing, the marginals of the variables are computed as similarly done for the previous examples. In a second phase, the evidence $E = 0$ is assumed. The computation of the marginals is omitted since it is specular to the previous case.
Finally, $D = 1$ is assumed and a new belief propagation is done, whose computations are reported in Figure 4.11.

4.3.2 part 02

Part 02 considers the graph reported in Figure 4.12. All the variables in Figure 4.12 have a domain size equal to 2, and all the factors are simply correlating exponential shape, having a unitary weight. Variables v_1, v_2 and v_3 are treated as evidences and the belief is propagated across the other ones, leading to the computation of the individual marginal probabilities. Since, the addressed structure is a politree (refer to Figure 2.6), the message passing algorithm converges within a finite number of steps.
In principle, the same approach followed in the previous examples can be followed to compute some theoretical results, with the aim of performing the comparisons. Anyway, for this kind of graph such an approach would be too complex. For this reason, results are compared with a Gibbs sampling approach: a series of samples $\mathcal{T} = \{T_1, \cdots, T_N\}$ are taken from the joint conditioned distribution $\mathbb{P}(T = v_{4,5,6,7,8,9,10,11,12,13} | v_{1,2,3})$. Then, to evaluate the marginal probability $\mathbb{P}(v_i | v_{1,2,3})$ of a generic hidden variable v_i, the following empirical frequency is computed:

$$\mathbb{P}(v_i = v | v_{1,2,3}) = \frac{\sum_{T_j \in \mathcal{T}} L_{Ti}(T_j, v)}{N} \tag{4.18}$$

where $L_{Ti}(T_j, v)$ is an indicator function equal to 1 only for those samples for which v_i assumed a value equal to v.

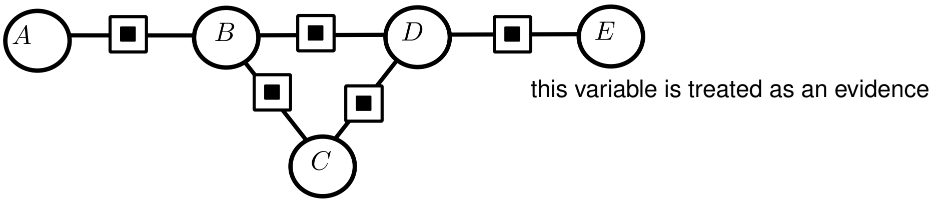


Figure 4.13 The factor graph considered by part 03.

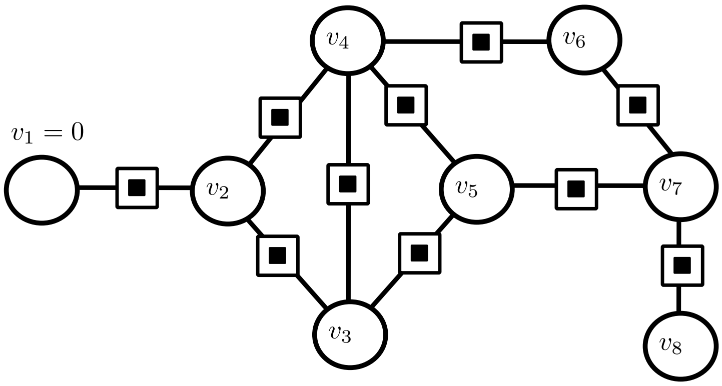


Figure 4.14 The factor graph considered by part 04.

4.3.3 part 03

Part 03 considers the graph reported in Figure 4.13. As for the example in the previous part, all variables are binary, and the potentials are simply exponential correlating with a unitary weight. However, this structure is loopy. E is treated as an evidence and the belief propagation is performed considering the loopy version of the message passing discussed in Section 2.2.2.

4.3.4 part 04

The last example in this series, considers a complex loopy graph, represented in Figure 4.14. As for other examples, all the variables are binary and the factors are exponential simply correlating with unitary weights. v_1 is assumed as evidence and the belief is propagated with the loopy version of message passing. Results are compared to the empirical frequencies obtained with a Gibbs sampler, as similarly done for the example of part 02.

4.4 Sample 04: Hidden Markov model like structure

The structure reported in Figure 4.15 is considered in this example. The reported chain is similar to those considered in Hidden Markov models, for which the chain of hidden variables $Y_{1,2,...}$ are connected to the chain of evidences $X_{1,2,...}$. The potential Φ_{YY0}, represents an a-priori knowledge about variable Y_1. All the variables are binary and the potentials are simply correlating exponential potentials. In particular, the ones connecting the hidden variables have a weight equal to w_{YY}, while the ones connecting the evidences to the hidden set share a weight equal to w_{XY}. The evidences are set as indicated in Figure 4.15, i.e. 0 and 1 are alternated in the chain represented by $X_{1,2,...}$. The MAP estimation [2] of the hidden set (Section 2.3) is computed into two different situations:

- case a): $w_{XY} >> w_{YY}$

[1]The same steps are internally execute by Node_factory.

[2]The Node_factory class is in charge of invoking the proper version of the message passing algorithm that leads to the MAP estimation computation.

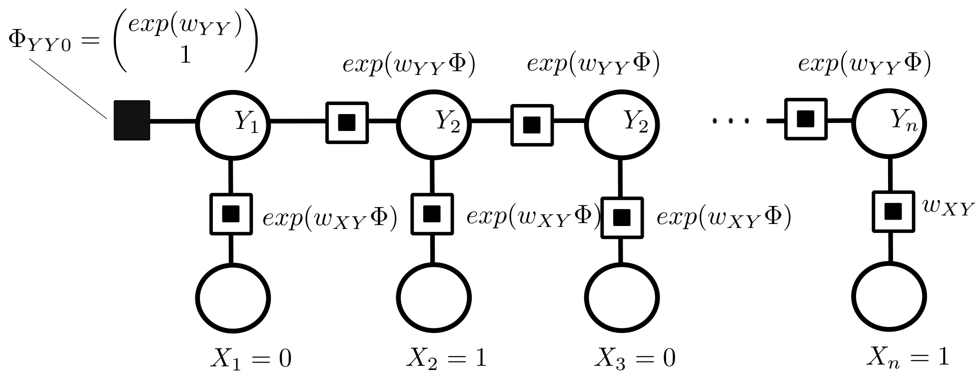


Figure 4.15 The chain structure considered by Sample 04.

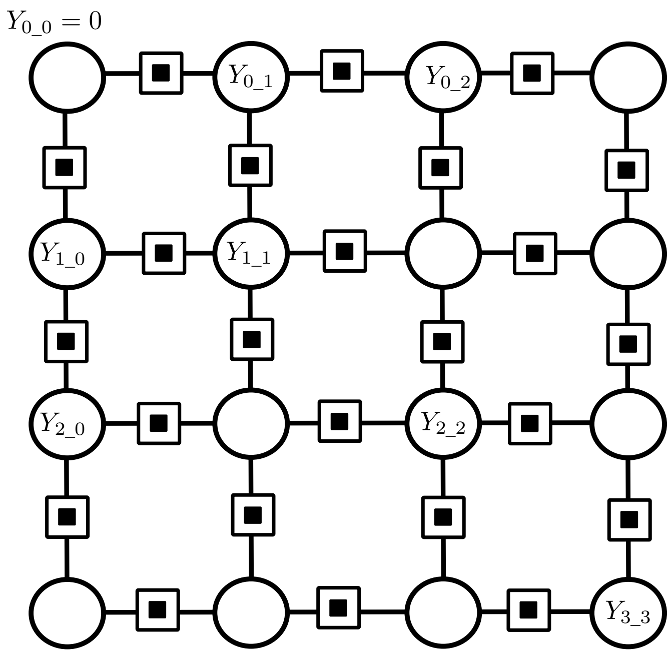


Figure 4.16 The matricial structure considered by Sample 05.

- case b): $w_{XY} << w_{YY}$

Here the point is that when considering case a), the information about the evidences and the correlations between $Y_{1,2,\cdots}$ and $X_{1,2,\cdots}$ is predominant. On the opposite, when dealing with case b), the correlations among the hidden variables as well as the prior knowledge about Y_0 is predominant. For this reason, for case a) the MAP estimation of the hidden variables is equal to $h^a_{MAP} = \{0, 1, 0, 1, \cdots\}$, while for case b) the MAP estimation is equal to $h^b_{MAP} = \{0, 0, 0, 0, \cdots\}$.

4.5 Sample 05: Matricial structure

The matrix-like structure reported in Figure 4.16 is considered in this example. The variables in the matrix have all the same domain size and the are correlated by the potentials populating the matrix, which are all simple exponential correlating factors sharing the same weight. The example builds the matrix and then assumes $Y_{0\_0} = 0$ as an evidence. Then, the marginals of the variables along the diagonal of the matrix, i.e. Y_{i_i}, are evaluated. As can be seen, the marginal probability $\mathbb{P}(Y_{i\_i} = 0|Y_{0\_0})$ decreases descending the diagonal. Figure 4.17 reports the results obtained when varying the weight of the correlating factors, on a matrix made of $10x10$ variables having a Dom size equal to 3.

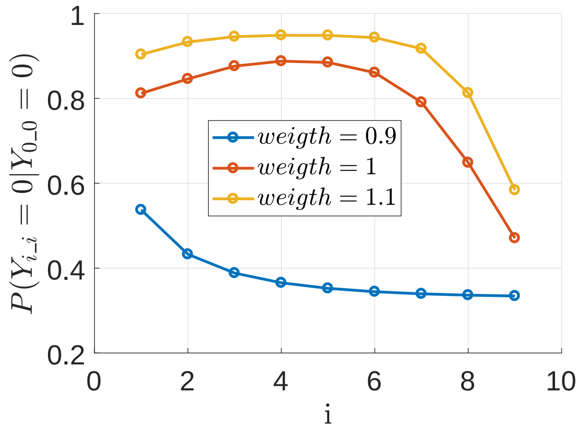


Figure 4.17 The marginals of variables Y_{i_i}, conditioned to $Y_{0\_0} = 0$ as evidence of the graph reported in Figure 4.16, when varying the weight of the correlating exponential factors involved in the structure.

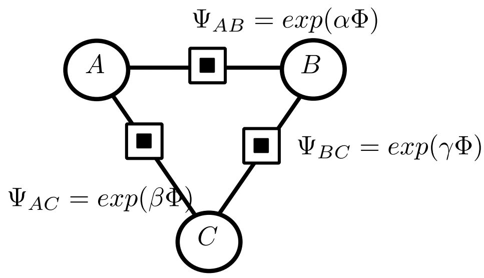


Figure 4.18 The graph considered by part 01.

4.6 Sample 06: Learning, part A

The aim of this series of examples is to show how to perform the learning of factor graphs. In all the examples contained in this Section, learning is done with the following methodology. A Gibbs sampler is used to take samples from the joint distribution correlating all the variables in a model A. Model A is actually the model to learn. The training set obtained from model A, is used to train a model B. Model B has the same variables and factors of model A, but with different values for the free parameters $w_{1,2,\ldots}$ (Section 2.6). In this way, after having performed the learning, the value of the free parameters in model B will assume similar values to the ones in model A, showing the effectiveness of the functionalities contained in EFG. Clearly, this is not the approach followed in real applications, were the real coefficient of the model are unknown and only a training set of examples are available.

4.6.1 part 01

Part 01 considers the loopy graph reported in Figure 4.18. A, B and C are all binary variables, while Ψ_{AB}, Ψ_{AC} and Ψ_{BC} are simple correlating exponential distributions having as weights, respectively, α, β and γ.
In the initial part of this example, a Gibbs sampler draw samples from the joint distribution of A, B and C, with the aim of validating the Gibbs sampler. Indeed, some empirical frequencies of some specific combinations are compared with the theoretical probabilities. The theoretical results are computed considering the energy function $E(A, B, C)$ of the graph, reported in table 4.2 (for instance $\mathbb{P}(A = 0, B = 0, C = 1) = \frac{E(0,0,1)}{Z}$).
At a second stage, the samples obtained by the Gibbs sampler are exploited for performing learning on model B (see the initial part of this Section).

4.6.2 part 02

Part 02 considers a structure made of both tunable and non-tunable factors. The considered structure is reported in Figure 4.19. Weights β and γ must be tuned through learning, while α and γ are constant and known (refer also to the formalism described in Figure 2.2).

A	B	C	Ψ_{AB}	Ψ_{BC}	Ψ_{AC}	$E(A, B, C) = \Psi_{AB} \cdot \Psi_{BC} \cdot \Psi_{AC}$
0	0	0	$exp(\alpha)$	$exp(\gamma)$	$exp(\beta)$	$exp(\alpha)exp(\beta)exp(\gamma)$
1	0	0	1	$exp(\gamma)$	1	$exp(\gamma)$
0	1	0	1	1	$exp(\beta)$	$exp(\beta)$
1	1	0	$exp(\alpha)$	1	1	$exp(\alpha)$
0	0	1	$exp(\alpha)$	1	1	$exp(\alpha)$
1	0	1	1	1	$exp(\beta)$	$exp(\beta)$
0	1	1	1	$exp(\gamma)$	1	$exp(\gamma)$
1	1	1	$exp(\alpha)$	$exp(\gamma)$	$exp(\beta)$	$exp(\alpha)exp(\beta)exp(\gamma)$

Table 4.2 Factors involved in the graph of Figure 2.8. Summing all the possible values of E, the ripartition function results equal to $Z = \sum E(A, B, C) = 2\Big(exp(\alpha) + exp(\beta) + exp(\gamma) + exp(\alpha)exp(\beta)exp(\gamma) \Big)$.

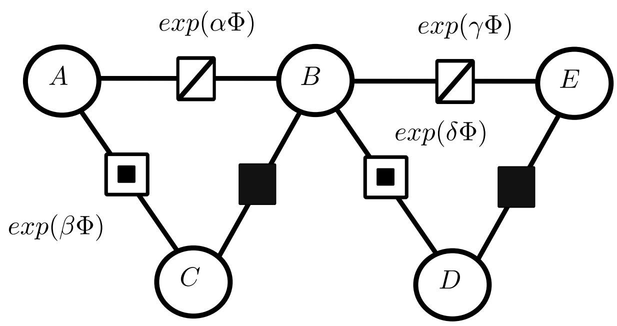


Figure 4.19 The graph considered by part 02.

4.6.3 part 03

Part 03 considers the loopy structure reported in Section 4.3.4. However, here instead of having constant exponential shapes, all the factors are made of tunable exponentials. The value assumed by the weight of model A (see the introduction of this Section) are showed in Figure 4.20.

4.6.4 part 04

Part 04 considers the structure reported in Figure 4.21, for which all the potentials connecting pairs Y_{i-1}, Y_i share the same weight α, while the factors connecting pairs X_i, Y_i share the weight β. The approach described in Section 2.6.3 is internally followed by EFG to learn such a structure.

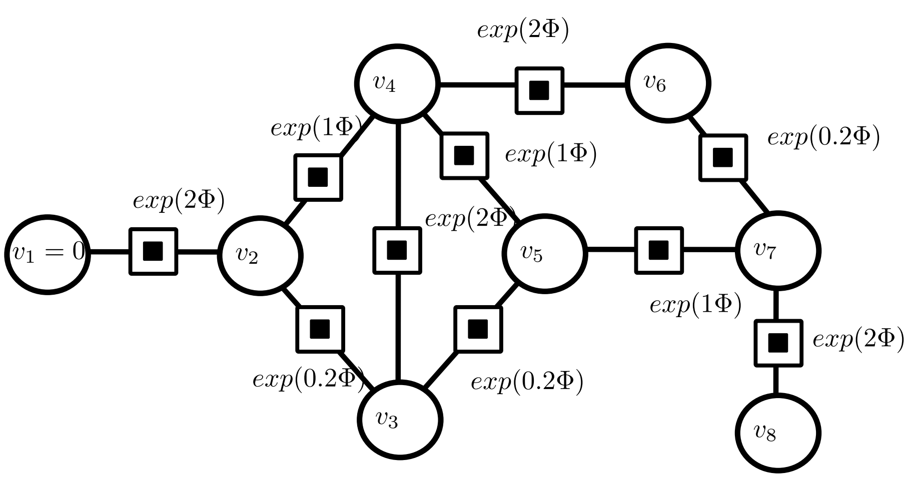


Figure 4.20 The graph considered by part 03.

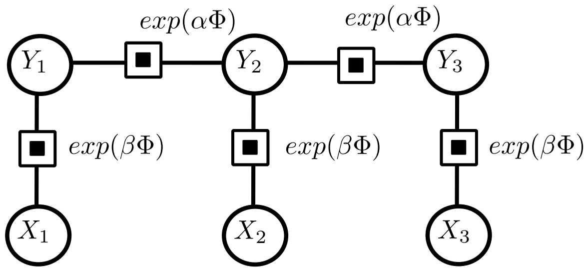


Figure 4.21 The graph considered by part 04.

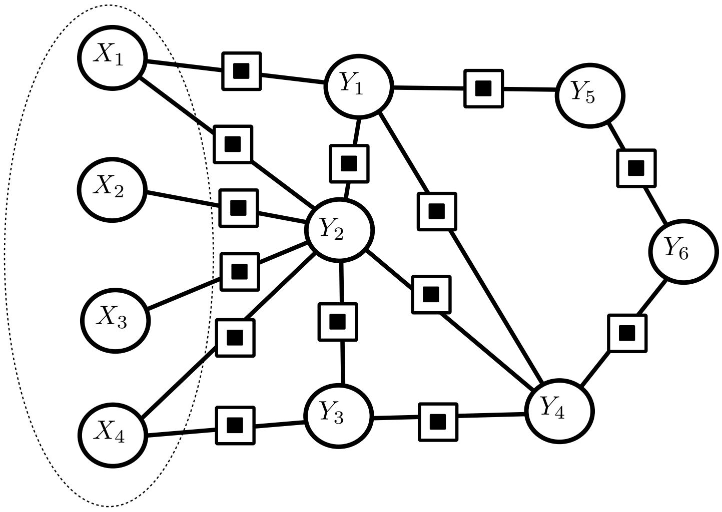


Figure 4.22 The conditional random field considered in Sample 07.

4.7 Sample 07: Learning, part B

The aim of this example is to show how the learning process can be done when dealing with Conditional random fields. In particular, the structure reported in Figure 4.22 is considered (values of the free parameters are not indicated, since the reader may refer to the sources provided).
The approach adopted is similar to the one followed in the previous series of example, considering a couple of model A and B (see the initial part of the previous Section). However, in this case we cannot simply draw samples from the joint distribution correlating the variables in the model, since such a distribution does not exists. Indeed, the conditional random field of Figure 4.22, models the conditional distribution of variables $Y_{1,2,\cdots}$ w.r.t the evidences $X_{1,2,\cdots}$. For this reason, all the possible combination of evidences are determined, considering all $x \in \{Dom(X_1) \cup Dom(X_2) \cup \cdots\}$. For each x, samples from the conditioned distribution $\mathbb{P}(Y_{1,2,\cdots}|x)$ are taken with a Gibbs sampler. The entire population of samples determined is actually the training set adopted fro training the conditional random field in Figure 4.22.

4.8 Sample 08: Sub-graphing

4.8.1 part 01

The chain structure described in Figure 4.23 is addressed in this example. The sub-graphs containing variables A, B, C and A, B are built, in order to compute the joint marginal probability distributions of that two groups of variables.
The values are compared to the real ones, obtained by considering the joint distribution [3] of all the variables in the

[3] Which is significantly time consuming to compute. For this reason, the SubGraph class is able to compute the marginals without explicitly compute the entire joint distribution of the variables in the model. Here we want just to compare the theoretical result with the one obtained by the SubGraph class.

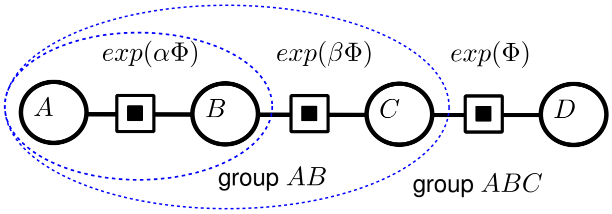


Figure 4.23 The chain considered in the example. All the underlying simple shapes are simple correlating.

A	B	$\mathbb{P}(A,B)$
0	0	$\frac{exp(\alpha)}{1+exp(\alpha)}$
0	1	$\frac{1}{1+exp(\alpha)}$
1	0	$\frac{1}{1+exp(\alpha)}$
1	1	$\frac{exp(\alpha)}{1+exp(\alpha)}$

A	B	C	$\mathbb{P}(A,B,C)$
0	0	0	$\frac{1}{Z_{ABC}} \cdot exp(\alpha)exp(\beta)$
0	1	0	$\frac{1}{Z_{ABC}}$
1	0	0	$\frac{1}{Z_{ABC}} \cdot exp(\beta)$
1	1	0	$\frac{1}{Z_{ABC}} \cdot exp(\alpha)$
0	0	1	$\frac{1}{Z_{ABC}} \cdot exp(\alpha)$
0	1	1	$\frac{1}{Z_{ABC}} \cdot exp(\beta)$
1	0	1	$\frac{1}{Z_{ABC}}$
1	1	1	$\frac{1}{Z_{ABC}} \cdot exp(\alpha)exp(\beta)$

Figure 4.24 Marginal probabilities of the sub-groups $\{A, B, C\}$ and $\{A, B\}$. The normalization coefficient Z_{ABC} is equal to $Z_{ABC} = 2\bigg(1 + exp(\alpha) + exp(\beta) + exp(\alpha)exp(\beta)\bigg)$.

chain, which can be obtained by computing the energy function E, equation (2.5) (computations are here omitted for brevity). Knowing the joint distribution of a group of variables, the marginal distribution of a sub-group can be obtained by marginalization, equation (2.3):

$$\begin{aligned} \mathbb{P}(A=a, B=b, C=c) &= \sum_{\tilde{d} \in Dom(D)} \mathbb{P}(A=a, B=b, C=c, D=\tilde{d}) \\ \mathbb{P}(A=a, B=b) &= \sum_{\tilde{c} \in Dom(C), \tilde{d} \in Dom(D)} \mathbb{P}(A=a, B=b, C=\tilde{c}, D=\tilde{d}) \end{aligned} \tag{4.19}$$

Applying the above rules to the chain of Figure 4.23 leads to obtain the theoretical marginal distributions indicated in Figure 4.24.

4.8.2 part 02

The aim of this example is to show how sub-graphs (see Section 2.5) can be computed using SubGraph. The example starts building the structure described in Figure 4.25 (refer to the sources for the details regarding the variables and factors involved in the structure) and assumes the following evidences: $X_1 = 0$ and $X_2 = 0$. Then, the two sub-graphs considering the sub-group of variables $A_{1,2,3,4}$ and $B_{1,2,3}$ are computed, refer to Figure 4.25. The marginal probabilities of some combinations for $A_{1,2,3,4}$ conditioned to the evidences $X_{1,2}$ are computed and compared with the empirical frequencies computed considering a samples set produced by a Gibb sampler on the entire graph: samples for $t = A_{1,2,3,4}, B_{1,23}$ are drawn and the empirical frequencies of specific combinations of $A_{1,2,3,4}$ are computed as similarly done in 4.3.2. The same thing is done for the sub-graph $B_{1,2,3}$.
At a second stage, the evidences $X_{1,2}$ are changed and the sub-graphs, as well as the marginal probabilities, are consequently recomputed.

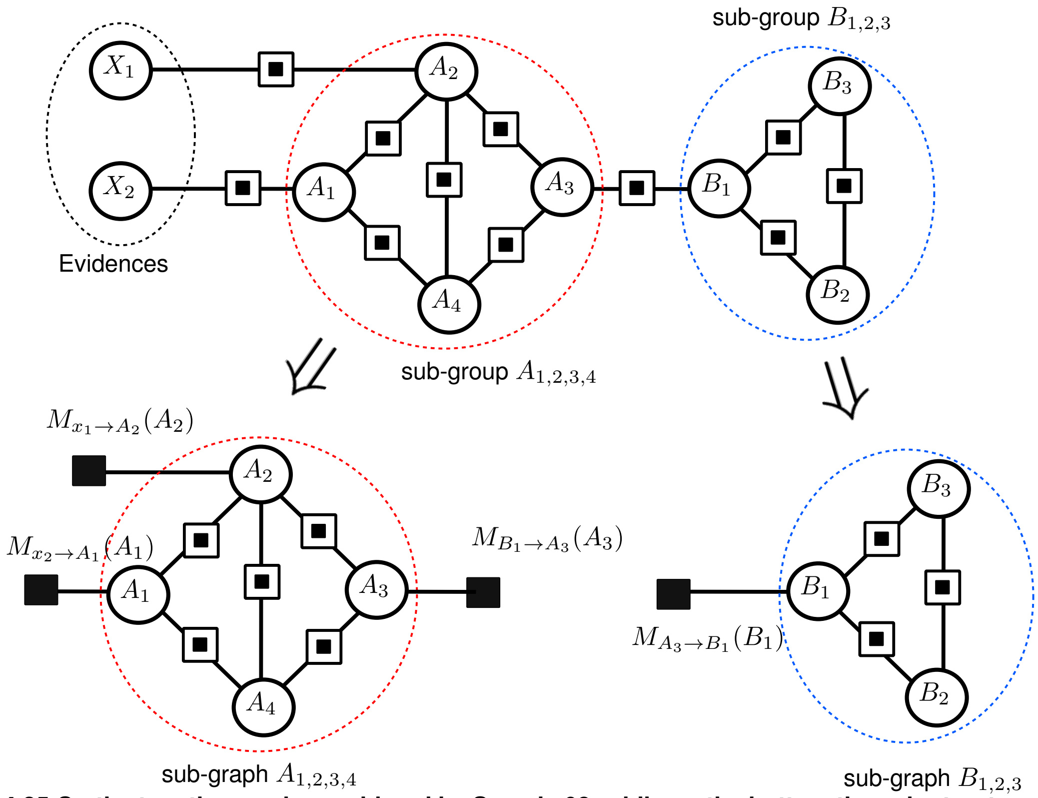


Figure 4.25 On the top, the graph considered by Sample 08, while on the bottom the sub-structures of the two groups $A_{1,2,3,4}$ ad $B_{1,2,3}$.

Chapter 5

Namespace Index

5.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

Chapter 6

Hierarchical Index

6.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 7

Class Index

7.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 8

Namespace Documentation

8.1 EFG Namespace Reference

Namespaces

- categoric
- distribution
- io
- iterator
- model
- nodes
- train

Classes

- class Error

  A runtime error that can be raised when using any object in EFG::

8.1.1 Detailed Description

Author: Andrea Casalino Created: 01.01.2021

report any bug to `andrecasa91@gmail.com`.

8.2 EFG::categoric Namespace Reference

Classes

- class Combination

  An immutable combination of discrete values.
- class Group

  An ensemble of categoric variables. Each variable in the ensemble should have its own unique name.
- class Range

  This object allows you to iterate all the elements in the joint domain of a group of variables, without precomputing all the elements in such domain. For example when having a domain made by variables = { A (size = 2), B (size = 3), C (size = 2) }, the elements in the joint domain that will be iterated are: <0,0,0> <0,0,1> <0,1,0> <0,1,1> <0,2,0> <0,2,1> <1,0,0> <1,0,1> <1,1,0> <1,1,1> <1,2,0> <1,2,1> After construction, the Range object starts to point to the first element in the joint domain <0,0,...>. Then, when incrementing the object, the following element is pointed. When calling get() the current pointed element can be accessed.
- class Variable

  An object representing an immutable categoric variable.

Typedefs

- typedef std::shared_ptr< Variable > **VariablePtr**

Functions

- bool **operator<** (const VariablePtr &a, const VariablePtr &b)
- bool **operator==** (const VariablePtr &a, const VariablePtr &b)
- std::set< VariablePtr > getComplementary (const std::set< VariablePtr > &set, const std::set< VariablePtr > &subset)

    get the complementary group of the subset w.r.t a certain set. For instance the complementary of <A,C> w.r.t. <A,B,C,D,E> is <B,D,E>
- VariablePtr **makeVariable** (const std::size_t &size, const std::string &name)

8.2.1 Detailed Description

Author: Andrea Casalino Created: 01.01.2021

report any bug to andrecasa91@gmail.com.

8.3 EFG::distribution Namespace Reference

Namespaces

- factor

Classes

- class Distribution

    Base object for any kind of categoric distribution. Any kind of categoric distribution has:
- class DistributionFinder

    An object used to search for big combinations inside a Distribution.
- class DistributionInstantiable
- class DistributionIterator

    An object able to iterate the domain/images of a distribution.
- class DistributionSetter
- class Evaluator

Typedefs

- typedef std::shared_ptr< Distribution > **DistributionPtr**
- typedef std::shared_ptr< Evaluator > **EvaluatorPtr**

8.3.1 Detailed Description

Author: Andrea Casalino Created: 01.01.2021

report any bug to andrecasa91@gmail.com.

8.4 EFG::distribution::factor Namespace Reference

Namespaces

- cnst
- modif

Classes

- class EvaluatorBasic

    image = exp(w ∗ rowImage)
- class EvaluatorExponential

    An exponential function with weight w is used to obtain the image, i.e. image = exp(w ∗ rowImage)

8.4.1 Detailed Description

Author: Andrea Casalino Created: 01.01.2021

report any bug to andrecasa91@gmail.com.

8.5 EFG::distribution::factor::cnst Namespace Reference

Classes

- class Factor

    A factor using the EvaluatorBasic object to convert the raw images into images.
- class FactorExponential

    A factor using the EvaluatorExponential object to convert the raw images into images.
- class IndicatorFactor

    An indicator distirbution having only one combination explicitly stated, whose image is equal to 1.

8.5.1 Detailed Description

Author: Andrea Casalino Created: 01.01.2021

report any bug to andrecasa91@gmail.com.

8.6 EFG::distribution::factor::modif Namespace Reference

Classes

- class Factor
- class FactorExponential

8.6.1 Detailed Description

Author: Andrea Casalino Created: 01.01.2021

report any bug to andrecasa91@gmail.com.

8.7 EFG::io Namespace Reference

Namespaces

- json
- xml

Classes

- class Exporter
- class Importer

8.7.1 Detailed Description

Author: Andrea Casalino Created: 01.01.2021

report any bug to andrecasa91@gmail.com.

8.8 EFG::io::json Namespace Reference

Classes

- class Exporter

8.8.1 Detailed Description

Author: Andrea Casalino Created: 01.01.2021

report any bug to andrecasa91@gmail.com.

8.9 EFG::io::xml Namespace Reference

Classes

- class Exporter
- class Importer

8.9.1 Detailed Description

Author: Andrea Casalino Created: 01.01.2021

report any bug to andrecasa91@gmail.com.

8.10 EFG::iterator Namespace Reference

Classes

- class Bidirectional

    A Bidirectional iterable object, both incrementable and decrementable.
- class Forward

    A Forward iterable object.
- class StlBidirectional

    A bidirectional iterator built on top of an std iterator type.

Functions

- template<typename Iter , typename Action >
  void forEach (Iter &iter, Action action)

    takes an iterator to increment till the end, calling at every iteration the passed action.
- template<typename Iter , typename ActionCondition >
  void forEachConditioned (Iter &iter, ActionCondition action)

    similar to forEach(...), but in this case the action should be a predicate, taking as input the iterator and returning true when the loop should be terminated before reaching the end of the iterator.

8.10.1 Detailed Description

Author: Andrea Casalino Created: 01.01.2021

report any bug to andrecasa91@gmail.com.

8.10.2 Function Documentation

8.10.2.1 forEach()

```
template<typename Iter , typename Action >
void EFG::iterator::forEach (
            Iter & iter,
            Action action )
```

takes an iterator to increment till the end, calling at every iteration the passed action.

Parameters

the	iterator to increment
an	action taking as input the iterator for every iteration

8.10.2.2 forEachConditioned()

```
template<typename Iter , typename ActionCondition >
void EFG::iterator::forEachConditioned (
            Iter & iter,
            ActionCondition action )
```

similar to forEach(...), but in this case the action should be a predicate, taking as input the iterator and returning true when the loop should be terminated before reaching the end of the iterator.

Parameters

the	iterator to increment
an	action taking as input the iterator for every iteration and returning true when the loop should break

8.11 EFG::model Namespace Reference

Classes

- class ConditionalRandomField
- class Graph

  A simple graph object, that can't store tunable factors.
- class RandomField

8.11.1 Detailed Description

Author: Andrea Casalino Created: 01.01.2021

report any bug to andrecasa91@gmail.com.

8.12 EFG::nodes Namespace Reference

Classes

- class Base
- class BeliefAware
- class BeliefPropagator
- struct Connection

- class EvidenceAware
- class EvidencesChanger
- class EvidencesSetter
- class GibbsSampler
- struct HiddenClusters
- class InsertCapable
- class InsertTunableCapable
- struct Node
- class NodesAware
- struct PropagationResult
- class QueryHandler
- class StructureAware
- class StructureTunableAware

Enumerations

- enum **PropagationKind** { **Sum**, **MAP** }

Functions

- void **copyCluster** (std::set$<$ Node $*$ $>$ &recipient, const std::set$<$ Node $*$ $>$ &toAdd)

8.12.1 Detailed Description

Author: Andrea Casalino Created: 01.01.2021

report any bug to `andrecasa91@gmail.com`.

8.13 EFG::train Namespace Reference

Namespaces

- handler

Classes

- class BasicExtractor
- class GradientDescend
- class StochasticExtractor
- class Trainable

    An object storing tunable factors, whose weights can be tuned with training.
- class Trainer
- class TrainHandler
- class TrainSet
- class TrainSetExtractor

Typedefs

- typedef std::unique_ptr< TrainHandler > **TrainHandlerPtr**
- typedef std::shared_ptr< categoric::Combination > **CombinationPtr**
- typedef std::shared_ptr< TrainSet > **TrainSetPtr**

Functions

- void **printTrainSet** (const TrainSet &trainSet, const std::string &fileName)

8.13.1 Detailed Description

Author: Andrea Casalino Created: 01.01.2021

report any bug to andrecasa91@gmail.com.

8.14 EFG::train::handler Namespace Reference

Classes

- class BaseHandler
- class BinaryHandler
- class CompositeHandler
- class UnaryHandler

8.14.1 Detailed Description

Author: Andrea Casalino Created: 01.01.2021

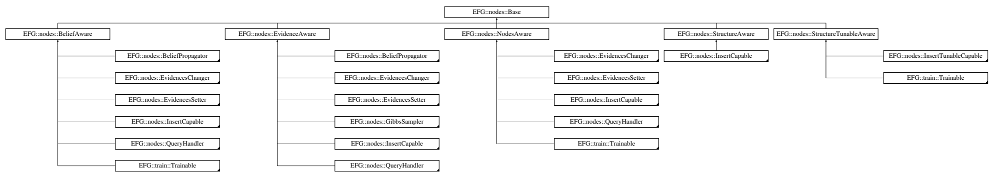
report any bug to andrecasa91@gmail.com.

Chapter 9

Class Documentation

9.1 EFG::nodes::Base Class Reference

Inheritance diagram for EFG::nodes::Base:



Public Member Functions

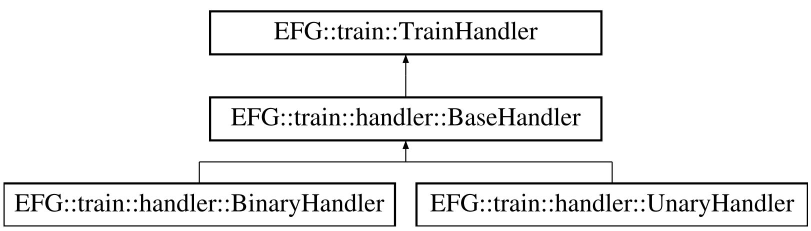
- **Base** (const Base &)=delete
- Base & **operator=** (const Base &)=delete

The documentation for this class was generated from the following file:

- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/nodes/bases/Base.h

9.2 EFG::train::handler::BaseHandler Class Reference

Inheritance diagram for EFG::train::handler::BaseHandler:



Public Member Functions

- void **setTrainSet** (TrainSetPtr newSet, const std::set< categoric::VariablePtr > &modelVariables) final
- float **getGradientAlpha** () final
- void **setWeight** (const float &w) final

Protected Member Functions

- **BaseHandler** (std::shared_ptr< distribution::factor::modif::FactorExponential > factor)
- float **dotProduct** (const std::vector< float > &prob) const

Protected Attributes

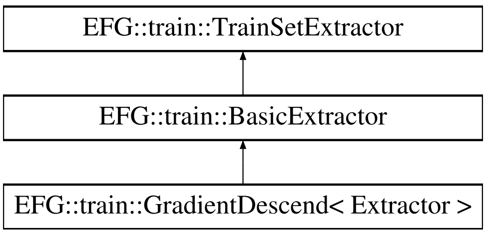
- std::shared_ptr< distribution::factor::modif::FactorExponential > **factor**
- float **gradientAlpha** = 0.f

The documentation for this class was generated from the following file:

- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/train/handlers/BaseHandler.h

9.3 EFG::train::BasicExtractor Class Reference

Inheritance diagram for EFG::train::BasicExtractor:



Protected Member Functions

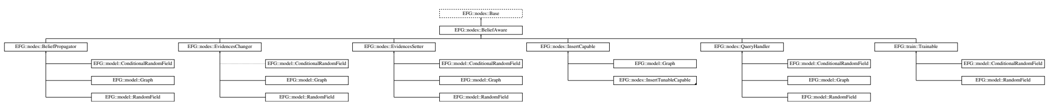
- TrainSetPtr **getTrainSet** () override

The documentation for this class was generated from the following file:

- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/train/trainers/TrainSetExtractor.h

9.4 EFG::nodes::BeliefAware Class Reference

Inheritance diagram for EFG::nodes::BeliefAware:



Public Member Functions

- void **setMaxIterationsLoopyPropagation** (std::size_t iterations)
- std::size_t **getMaxIterationsLoopyPropagation** () const
- PropagationResult **getLastPropagationResult** () const

Protected Member Functions

- virtual void **propagateBelief** (const PropagationKind &kind)=0

Protected Attributes

- std::size_t maxIterationsLoopyPropagtion = 100

  maximum number of iterations considered when doing loopy propagation
- std::unique_ptr< PropagationResult > lastPropagation

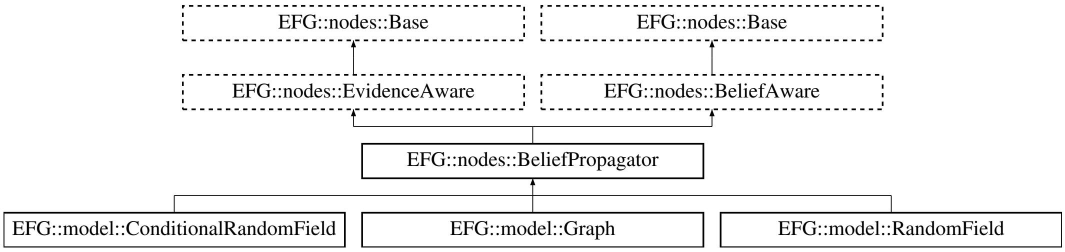
  results about the last belief propagation done. It is a nullptr until the first propagation is triggered

The documentation for this class was generated from the following file:

- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/nodes/bases/BeliefAware.h

9.5 EFG::nodes::BeliefPropagator Class Reference

Inheritance diagram for EFG::nodes::BeliefPropagator:



Protected Member Functions

- void **propagateBelief** (const PropagationKind &kind) override

Additional Inherited Members

The documentation for this class was generated from the following file:

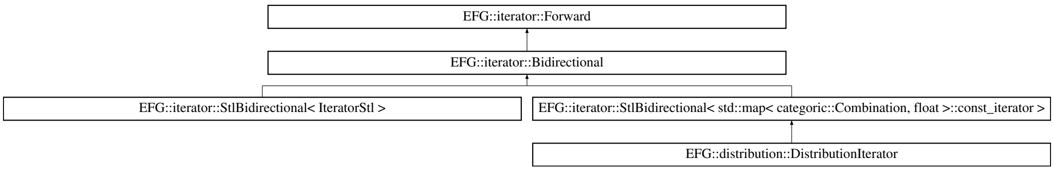
- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/nodes/BeliefPropagator.h

9.6 EFG::iterator::Bidirectional Class Reference

A Bidirectional iterable object, both incrementable and decrementable.

```
#include <Bidirectional.h>
```

Inheritance diagram for EFG::iterator::Bidirectional:



Public Member Functions

- virtual void **operator--** ()=0

9.6.1 Detailed Description

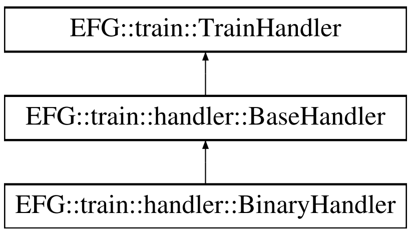
A Bidirectional iterable object, both incrementable and decrementable.

The documentation for this class was generated from the following file:

- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/iterator/Bidirectional.h

9.7 EFG::train::handler::BinaryHandler Class Reference

Inheritance diagram for EFG::train::handler::BinaryHandler:



Public Member Functions

- **BinaryHandler** (nodes::Node &nodeA, nodes::Node &nodeB, std::shared_ptr< distribution::factor::modif::FactorExponential > factor)
- float **getGradientBeta** () final

Protected Attributes

- nodes::Node ∗ **nodeA**
- nodes::Node ∗ **nodeB**

Additional Inherited Members

The documentation for this class was generated from the following file:

- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/train/handlers/BinaryHandler.h

9.8 EFG::categoric::Combination Class Reference

An immutable combination of discrete values.

```
#include <Combination.h>
```

Public Member Functions

- Combination (std::size_t bufferSize)

  A buffer of zeros with the passed size is created.
- Combination (const std::size_t ∗buffer, std::size_t bufferSize)

  The passed buffer is copied to create this one.
- **Combination** (const Combination &o)
- Combination & **operator=** (const Combination &o)
- bool operator< (const Combination &o) const

  compare two equally sized combination. Examples of ordering: <0,0,0> < <0,1,0> <0,1> < <1,0>
- std::size_t **size** () const
- const std::size_t ∗ **data** () const
- std::size_t ∗ **data** ()

9.8.1 Detailed Description

An immutable combination of discrete values.

9.8.2 Constructor & Destructor Documentation

9.8.2.1 Combination()

```
EFG::categoric::Combination::Combination (
            const std::size_t * buffer,
            std::size_t bufferSize )
```

The passed buffer is copied to create this one.

Parameters

the	buffer to clone
the	buffer size

9.8.3 Member Function Documentation

9.8.3.1 operator<()

```
bool EFG::categoric::Combination::operator< (
            const Combination & o ) const
```

compare two equally sized combination. Examples of ordering: <0,0,0> < <0,1,0> <0,1> < <1,0>

Exceptions

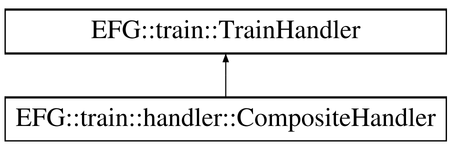
if	the 2 combinations don't have the same number of values

The documentation for this class was generated from the following file:

- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/categoric/Combination.h

9.9 EFG::train::handler::CompositeHandler Class Reference

Inheritance diagram for EFG::train::handler::CompositeHandler:



Public Member Functions

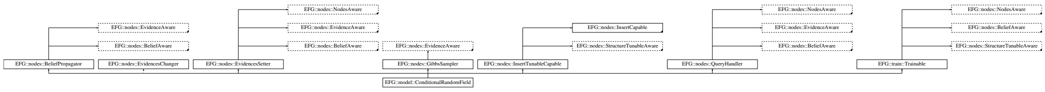
- **CompositeHandler** (TrainHandlerPtr elementA, TrainHandlerPtr elementB)
- void **setTrainSet** (TrainSetPtr newSet, const std::set< categoric::VariablePtr > &modelVariables) final
- float **getGradientAlpha** () final
- float **getGradientBeta** () final
- void **setWeight** (const float &w) final
- void **addElement** (TrainHandlerPtr element)

The documentation for this class was generated from the following file:

- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/train/handlers/CompositeHandler.h

9.10 EFG::model::ConditionalRandomField Class Reference

Inheritance diagram for EFG::model::ConditionalRandomField:



Public Member Functions

- template< typename Model >
  ConditionalRandomField (const Model &o)
- **ConditionalRandomField** (const ConditionalRandomField &o)
- ConditionalRandomField (const std::string &filePath, const std::string &fileName)

  import the model from an xml file
- void insertTunable (std::shared_ptr< distribution::factor::modif::FactorExponential > toInsert) override

  insert the passed tunable factor.
- void insertTunable (std::shared_ptr< distribution::factor::modif::FactorExponential > toInsert, const std::set< categoric::VariablePtr > &potentialSharingWeight) override

  insert the passed tunable factor, sharing the weight with an already inserted one.

Additional Inherited Members

9.10.1 Constructor & Destructor Documentation

9.10.1.1 ConditionalRandomField() [1/2]

```
template<typename Model >
EFG::model::ConditionalRandomField::ConditionalRandomField (
            const Model & o )  [inline], [explicit]
```

Exceptions

in	case no evidences are present in the passed model

9.10.1.2 ConditionalRandomField() [2/2]

```
EFG::model::ConditionalRandomField::ConditionalRandomField (
            const std::string & filePath,
            const std::string & fileName )
```

import the model from an xml file

Parameters

the	folder storing the xml to read
the	name of the xml to read

Exceptions

in	case no evidences are set in the file

9.10.2 Member Function Documentation

9.10.2.1 insertTunable()

```
void EFG::model::ConditionalRandomField::insertTunable (
            std::shared_ptr< distribution::factor::modif::FactorExponential > toInsert,
            const std::set< categoric::VariablePtr > & potentialSharingWeight )  [override],
[virtual]
```

insert the passed tunable factor, sharing the weight with an already inserted one.

Parameters

the	factor to insert
the	set of variables identifying the potential whose weight is to share

Reimplemented from EFG::nodes::InsertTunableCapable.

The documentation for this class was generated from the following file:

- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/model/ConditionalRandomField.h

9.11 EFG::nodes::Connection Struct Reference

Public Member Functions

- **Connection** (distribution::DistributionPtr factor, std::unique_ptr< distribution::Distribution > message=nullptr)
- **Connection** (Connection &&o)

Public Attributes

- distribution::DistributionPtr **factor**
- std::unique_ptr< distribution::Distribution > **message2This**

The documentation for this struct was generated from the following file:

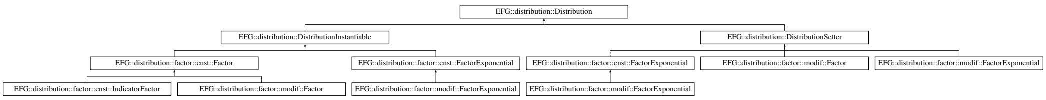
- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/nodes/Node.h

9.12 EFG::distribution::Distribution Class Reference

Base object for any kind of categoric distribution. Any kind of categoric distribution has:

```
#include <Distribution.h>
```

Inheritance diagram for EFG::distribution::Distribution:



Public Member Functions

- const categoric::Group & **getGroup** () const
- DistributionIterator getIterator () const
- float find (const categoric::Combination &comb) const

  searches for the image associated to an element in the domain
- float findRaw (const categoric::Combination &comb) const

  searches for the raw image associated to an element in the domain
- DistributionFinder getFinder (const std::set< categoric::VariablePtr > &containingGroup) const
- std::vector< float > getProbabilities () const

Protected Member Functions

- void **checkCombination** (const categoric::Combination &comb, const float &value) const

Protected Attributes

- std::unique_ptr< categoric::Group > **group**
- std::shared_ptr< std::map< categoric::Combination, float > > values

  the ordered pairs of <domain combination, raw image value>
- EvaluatorPtr evaluator

  the function used to convert raw images into images

Friends

- class **DistributionIterator**
- class **DistributionFinder**

9.12.1 Detailed Description

Base object for any kind of categoric distribution. Any kind of categoric distribution has:

- A domain, represented by the combinations in the joint domain of the Group associated to this distribution
- Raw images set, which are positive values associated to each element in the domain
- Images set, which are the image values associated to each element in the domain. They can be obtained by applying a certain function f(x) to the raw images In order to save memory, the combinations having an image equal to 0 are not explicitly saved even if they are accounted for the opreations involving this distribution.

9.12.2 Member Function Documentation

9.12.2.1 find()

```
float EFG::distribution::Distribution::find (
            const categoric::Combination & comb ) const
```

searches for the image associated to an element in the domain

Returns

the value of the image.

9.12.2.2 findRaw()

```
float EFG::distribution::Distribution::findRaw (
            const categoric::Combination & comb ) const
```

searches for the raw image associated to an element in the domain

Returns

the value of the raw image.

9.12.2.3 getFinder()

```
DistributionFinder EFG::distribution::Distribution::getFinder (
            const std::set< categoric::VariablePtr > & containingGroup ) const
```

Returns

a DistributionFinder referring to this object

9.12.2.4 getIterator()

```
DistributionIterator EFG::distribution::Distribution::getIterator ( ) const
```

Returns

a DistributionIterator referring to this object

9.12.2.5 getProbabilities()

```
std::vector<float> EFG::distribution::Distribution::getProbabilities ( ) const
```

Returns

the probabilities associated to each combination in the domain, when assuming only the existance of this distribution. Such probabilities are the normalized images. The order of returned values, refer to the combination order obtained by iterating with the categoric::Range object.

The documentation for this class was generated from the following file:

- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/distribution/Distribution.h

9.13 EFG::distribution::DistributionFinder Class Reference

An object used to search for big combinations inside a Distribution.

```
#include <DistributionFinder.h>
```

Public Member Functions

- DistributionFinder (const Distribution &distribution, const std::set< categoric::VariablePtr > &containing↩ Group)
- **DistributionFinder** (const DistributionFinder &)=default
- DistributionFinder & **operator=** (const DistributionFinder &)=default
- std::pair< const categoric::Combination ∗, float > find (const categoric::Combination &comb) const

  searches for matches. For example assume having built this object with a containingGroup equal to <A,B,C,D> and the variables describing the domain of the reference distribution equal to <B,D>. When passing comb ad <0,1,2,0>, it searches for the <combination,image> pretaining to this combination combination <B,D> = <1,0>.
- std::pair< const categoric::Combination ∗, float > findRaw (const categoric::Combination &comb) const

  similar to DistributionFinder::find(...), but returning the raw image value.

9.13.1 Detailed Description

An object used to search for big combinations inside a Distribution.

9.13.2 Constructor & Destructor Documentation

9.13.2.1 DistributionFinder()

```
EFG::distribution::DistributionFinder::DistributionFinder (
            const Distribution & distribution,
            const std::set< categoric::VariablePtr > & containingGroup )
```

Parameters

the	reference distribution
the	variables referring to the combinations to search. This kind of set should contain the subset of variables describing the domain of distribution

9.13.3 Member Function Documentation

9.13.3.1 find()

```
std::pair<const categoric::Combination*, float> EFG::distribution::DistributionFinder::find (
            const categoric::Combination & comb ) const
```

searches for matches. For example assume having built this object with a containingGroup equal to <A,B,C,D> and the variables describing the domain of the reference distribution equal to <B,D>. When passing comb ad <0,1,2,0>, it searches for the <combination,image> pretaining to this combination combination <B,D> = <1,0>.

Parameters

the	combination to search, referring to the set of variables passed when building this object.

Returns

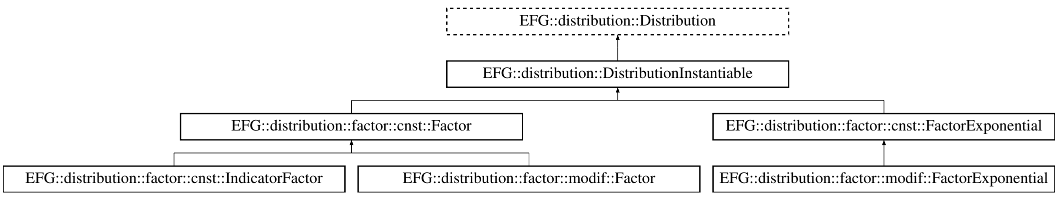
the pair <combination,image> of the the matching combination. <nullptr,0> is returned in case such a combination was not explicitly put in the distribution.

The documentation for this class was generated from the following file:

- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/distribution/DistributionFinder.h

9.14 EFG::distribution::DistributionInstantiable Class Reference

Inheritance diagram for EFG::distribution::DistributionInstantiable:



Protected Member Functions

- **DistributionInstantiable** (const std::set< categoric::VariablePtr > &group, EvaluatorPtr evaluator)
- **DistributionInstantiable** (const DistributionInstantiable &o)
- DistributionInstantiable & **operator=** (const DistributionInstantiable &o)
- **DistributionInstantiable** (DistributionInstantiable &&o)
- DistributionInstantiable & **operator=** (DistributionInstantiable &&o)

Additional Inherited Members

The documentation for this class was generated from the following file:

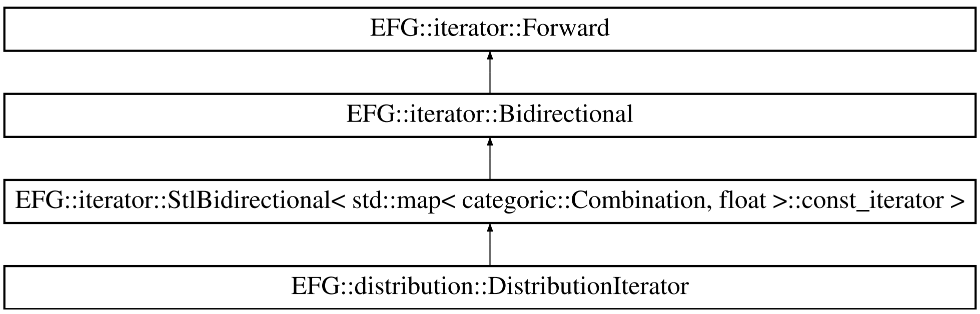
- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/distribution/DistributionInstantiable.h

9.15 EFG::distribution::DistributionIterator Class Reference

An object able to iterate the domain/images of a distribution.

```
#include <DistributionIterator.h>
```

Inheritance diagram for EFG::distribution::DistributionIterator:



Public Member Functions

- DistributionIterator (const Distribution &distribution)
- **DistributionIterator** (const DistributionIterator &)=default
- DistributionIterator & **operator=** (const DistributionIterator &)=default
- const categoric::Combination & getCombination () const
- float getImage () const
- float getImageRaw () const
- std::size_t getNumberOfValues () const

Additional Inherited Members

9.15.1 Detailed Description

An object able to iterate the domain/images of a distribution.

9.15.2 Constructor & Destructor Documentation

9.15.2.1 DistributionIterator()

```
EFG::distribution::DistributionIterator::DistributionIterator (
            const Distribution & distribution )  [explicit]
```

Parameters

the	distribution to iterate

9.15.3 Member Function Documentation

9.15.3.1 getCombination()

```
const categoric::Combination& EFG::distribution::DistributionIterator::getCombination ( )
const  [inline]
```

Returns

the combination currently pointed by the iterator

9.15.3.2 getImage()

```
float EFG::distribution::DistributionIterator::getImage ( ) const  [inline]
```

Returns

the image of the combination currently pointed by the iterator

9.15.3.3 getImageRaw()

```
float EFG::distribution::DistributionIterator::getImageRaw ( ) const  [inline]
```

Returns

the raw image of the combination currently pointed by the iterator

9.15.3.4 getNumberOfValues()

```
std::size_t EFG::distribution::DistributionIterator::getNumberOfValues ( ) const  [inline]
```

Returns

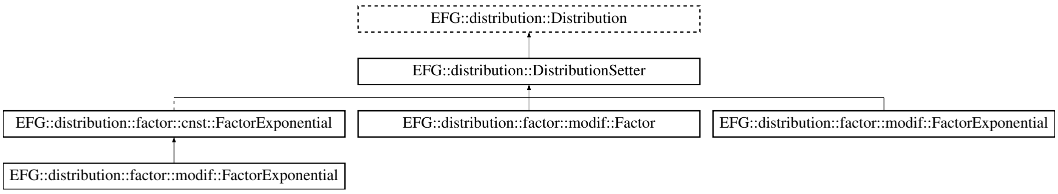
the number of combinations having a non 0 image value.

The documentation for this class was generated from the following file:

- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/distribution/DistributionIterator.h

9.16 EFG::distribution::DistributionSetter Class Reference

Inheritance diagram for EFG::distribution::DistributionSetter:



Public Member Functions

- void replaceGroup (const categoric::Group &newGroup)

  replace the variables describing the domain of this distribution
- void setImageRaw (const categoric::Combination &comb, const float &value)

  sets the image of the passed combination. In case the combination is currently not part of the distribution, it is added with the passe raw image value.
- void fillDomain ()

  creates all the non explicitly set combinations and assumed for them a 0 raw image value.
- void setAllImagesRaw (const float &value)

  sets the raw images of all the combinations equal to the passed value

Additional Inherited Members

9.16.1 Member Function Documentation

9.16.1.1 setImageRaw()

```
void EFG::distribution::DistributionSetter::setImageRaw (
            const categoric::Combination & comb,
            const float & value )
```

sets the image of the passed combination. In case the combination is currently not part of the distribution, it is added with the passe raw image value.

Parameters

the	combination whose raw image must be set
the	raw image value to assume

The documentation for this class was generated from the following file:

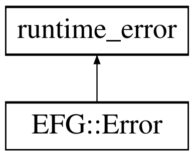
- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/distribution/DistributionSetter.h

9.17 EFG::Error Class Reference

A runtime error that can be raised when using any object in EFG::

```
#include <Error.h>
```

Inheritance diagram for EFG::Error:



Public Member Functions

- **Error** (const std::string &what)

9.17.1 Detailed Description

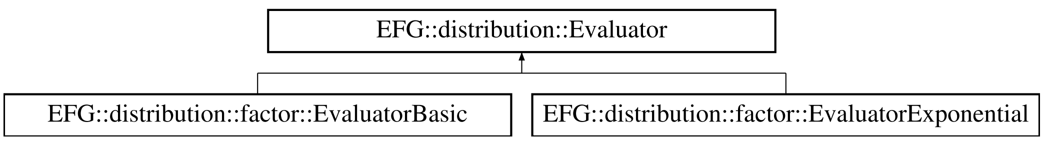
A runtime error that can be raised when using any object in EFG::

The documentation for this class was generated from the following file:

- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/Error.h

9.18 EFG::distribution::Evaluator Class Reference

Inheritance diagram for EFG::distribution::Evaluator:



Public Member Functions

- virtual float evaluate (const float &toConvert) const =0

  applies a specific function to obtain the image from a the raw image value
- virtual std::shared_ptr< Evaluator > **copy** () const =0

9.18.1 Member Function Documentation

9.18.1.1 evaluate()

```
virtual float EFG::distribution::Evaluator::evaluate (
            const float & toConvert ) const  [pure virtual]
```

applies a specific function to obtain the image from a the raw image value

Parameters

the	raw value to convert

Returns

the converted image

Implemented in EFG::distribution::factor::EvaluatorBasic, and EFG::distribution::factor::EvaluatorExponential.

The documentation for this class was generated from the following file:

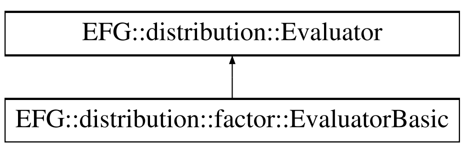
- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/distribution/Evaluator.h

9.19 EFG::distribution::factor::EvaluatorBasic Class Reference

image = exp(w $*$ rowImage)

```
#include <EvaluatorBasic.h>
```

Inheritance diagram for EFG::distribution::factor::EvaluatorBasic:



Public Member Functions

- float evaluate (const float &toConvert) const override

    applies a specific function to obtain the image from a the raw image value
- std::shared_ptr< Evaluator > **copy** () const override

9.19.1 Detailed Description

image = exp(w ∗ rowImage)

9.19.2 Member Function Documentation

9.19.2.1 evaluate()

```
float EFG::distribution::factor::EvaluatorBasic::evaluate (
            const float & toConvert ) const  [inline], [override], [virtual]
```

applies a specific function to obtain the image from a the raw image value

Parameters

the	raw value to convert

Returns

the converted image

Implements EFG::distribution::Evaluator.

The documentation for this class was generated from the following file:

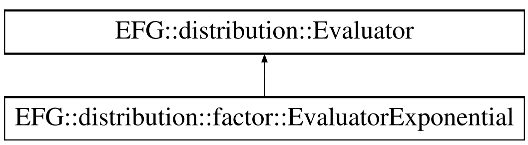
- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/distribution/factor/EvaluatorBasic.h

9.20 EFG::distribution::factor::EvaluatorExponential Class Reference

An exponential function with weight w is used to obtain the image, i.e. image = exp(w ∗ rowImage)

```
#include <EvaluatorExponential.h>
```

Inheritance diagram for EFG::distribution::factor::EvaluatorExponential:



Public Member Functions

- **EvaluatorExponential** (const float &weight)
- float **getWeight** () const
- void **setWeight** (float w)
- float evaluate (const float &toConvert) const

    applies a specific function to obtain the image from a the raw image value
- std::shared_ptr< Evaluator > **copy** () const override

9.20.1 Detailed Description

An exponential function with weight w is used to obtain the image, i.e. image = exp(w $*$ rowImage)

9.20.2 Member Function Documentation

9.20.2.1 evaluate()

```
float EFG::distribution::factor::EvaluatorExponential::evaluate (
            const float & toConvert ) const  [inline], [virtual]
```

applies a specific function to obtain the image from a the raw image value

Parameters

the	raw value to convert

Returns

the converted image

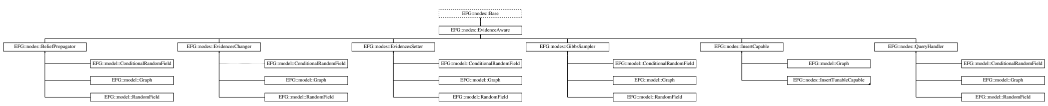
Implements EFG::distribution::Evaluator.

The documentation for this class was generated from the following file:

- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/distribution/factor/EvaluatorExponential.h

9.21 EFG::nodes::EvidenceAware Class Reference

Inheritance diagram for EFG::nodes::EvidenceAware:



Public Member Functions

- std::set< categoric::VariablePtr > **getHiddenVariables** () const
- std::set< categoric::VariablePtr > **getObservedVariables** () const
- const std::map< categoric::VariablePtr, const std::size_t > & **getEvidences** () const

Protected Attributes

- HiddenClusters hidden

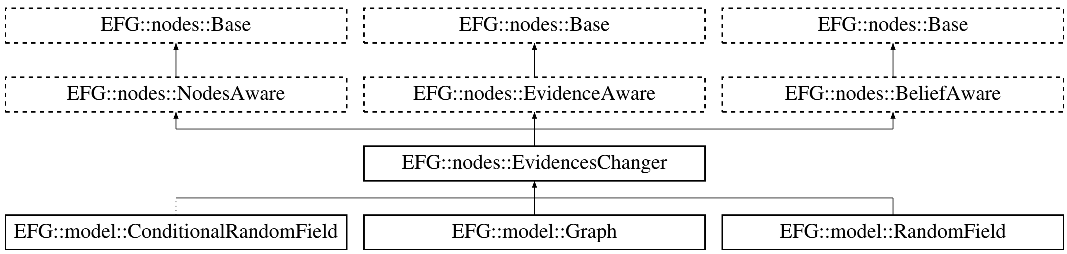
  Clusters of hidden node. Each cluster is a group of connected hidden nodes. Nodes in different clusters are not currently connected (due to the model structure or the kind of evidences currently set)
- std::map< categoric::VariablePtr, const std::size_t > **evidences**

The documentation for this class was generated from the following file:

- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/nodes/bases/EvidenceAware.h

9.22 EFG::nodes::EvidencesChanger Class Reference

Inheritance diagram for EFG::nodes::EvidencesChanger:



Public Member Functions

- void addEvidence (const std::string &name, std::size_t value)

  add a new evidence to the model
- void resetEvidences (const std::map< std::string, std::size_t > &evidences)

  reset the evidences, deleting the previous ones.

Additional Inherited Members

9.22.1 Member Function Documentation

9.22.1.1 addEvidence()

```
void EFG::nodes::EvidencesChanger::addEvidence (
            const std::string & name,
            std::size_t value )
```

add a new evidence to the model

Parameters

the	name of the variable observed
the	value of the evidence

9.22.1.2 resetEvidences()

```
void EFG::nodes::EvidencesChanger::resetEvidences (
            const std::map< std::string, std::size_t > & evidences )
```

reset the evidences, deleting the previous ones.

Parameters

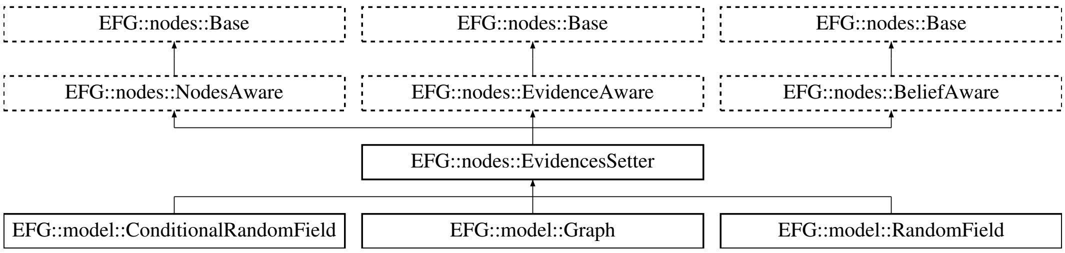
the	new evidences to assume: <variable name, evidence value>

The documentation for this class was generated from the following file:

- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/nodes/EvidenceChanger.h

9.23 EFG::nodes::EvidencesSetter Class Reference

Inheritance diagram for EFG::nodes::EvidencesSetter:



Public Member Functions

- void setEvidences (const std::vector< std::size_t > &observations)
  reset evidence values. The group of observed variables is left unchanged, but the evidence values are updated.

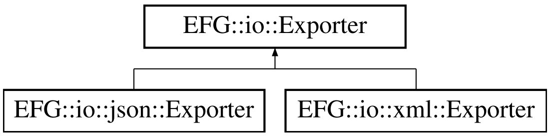
Additional Inherited Members

The documentation for this class was generated from the following file:

- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/nodes/EvidenceSetter.h

9.24 EFG::io::Exporter Class Reference

Inheritance diagram for EFG::io::Exporter:



Protected Member Functions

- virtual void **exportComponents** (const std::string &filePath, const std::string &modelName, const std::tuple$<$ const nodes::EvidenceAware $*$, const nodes::StructureAware $*$, const nodes::StructureTunableAware $*$ $>$ &components)=0

Static Protected Member Functions

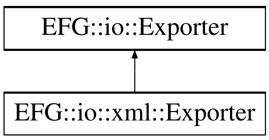
- template$<$typename Model $>$
  static std::tuple$<$ const nodes::EvidenceAware $*$, const nodes::StructureAware $*$, const nodes::StructureTunableAware $*$ $>$ **getComponents** (const Model &model)

The documentation for this class was generated from the following file:

- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/io/Exporter.h

9.25 EFG::io::xml::Exporter Class Reference

Inheritance diagram for EFG::io::xml::Exporter:



Static Public Member Functions

- template$<$typename Model $>$
  static void exportToXml (const Model &model, const std::string &filePath, const std::string &modelName="")
  exports the model (variables and factors) into an xml file

Additional Inherited Members

9.25.1 Member Function Documentation

9.25.1.1 exportToXml()

```
template<typename Model >
static void EFG::io::xml::Exporter::exportToXml (
            const Model & model,
            const std::string & filePath,
            const std::string & modelName = "" )  [inline], [static]
```

exports the model (variables and factors) into an xml file

Parameters

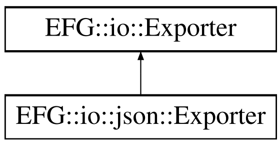
the	model to export
the	folder that will store the xml
the	xml file name

The documentation for this class was generated from the following file:

- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/io/xml/Exporter.h

9.26 EFG::io::json::Exporter Class Reference

Inheritance diagram for EFG::io::json::Exporter:



Static Public Member Functions

- template<typename Model >
  static void exportToJson (const Model &model, const std::string &filePath, const std::string &modelName="")

  exports the model (variables and factors) into a json file

Additional Inherited Members

9.26.1 Member Function Documentation

9.26.1.1 exportToJson()

```
template<typename Model >
static void EFG::io::json::Exporter::exportToJson (
            const Model & model,
            const std::string & filePath,
            const std::string & modelName = "" )  [inline], [static]
```

exports the model (variables and factors) into a json file

Parameters

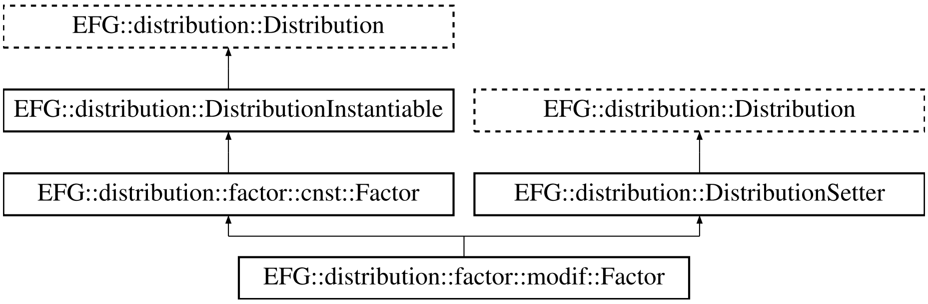
the	model to export
the	folder that will store the json
the	json file name

The documentation for this class was generated from the following file:

- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/io/json/Exporter.h

9.27 EFG::distribution::factor::modif::Factor Class Reference

Inheritance diagram for EFG::distribution::factor::modif::Factor:



Public Member Functions

- template<typename ... Args>
  Factor (Args &&... args)
- **Factor** (const std::set< categoric::VariablePtr > &group)
- **Factor** (const Factor &o)
- **Factor** (Factor &&o)
- Factor & **operator=** (const Factor &o)
- Factor & **operator=** (Factor &&o)
- void clear ()

    sets all raw images equal to 0 and deallocate all combinations

Additional Inherited Members

The documentation for this class was generated from the following file:

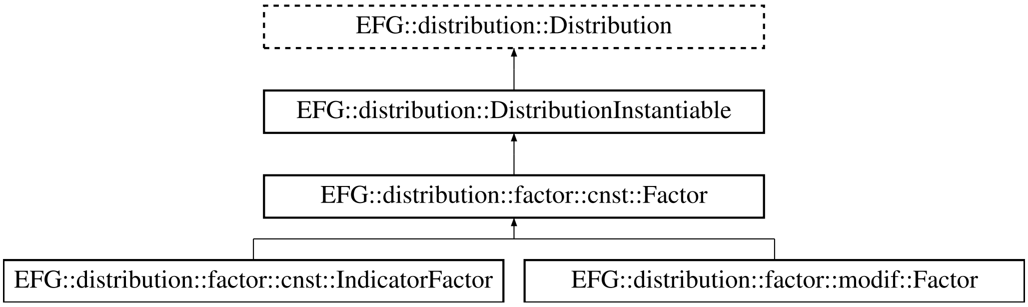
- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/distribution/factor/modifiable/Factor.h

9.28 EFG::distribution::factor::cnst::Factor Class Reference

A factor using the EvaluatorBasic object to convert the raw images into images.

```
#include <Factor.h>
```

Inheritance diagram for EFG::distribution::factor::cnst::Factor:



Public Member Functions

- Factor (const std::set< categoric::VariablePtr > &group, bool corrOrAnti)

  Builds a simple correlating or anticorrelating factor.
- **Factor** (const Factor &o)
- **Factor** (Factor &&o)
- Factor (const Distribution &o)

  Copies all the images (not raw) of the passed distribution to build a generic factor.
- template<typename ... Distributions>
  Factor (const Distribution ∗first, const Distribution ∗second, Distributions ... distr)

  Merges all the passed distribution into a single Factor. The domain of the Factor is obtained merging the domains of the distributions, while the image are obtained multiplying the images of the passed distributions.
- Factor (const std::set< const Distribution ∗ > &distr)

  Merges all the passed distribution into a single Factor. The domain of the Factor is obtained merging the domains of the distributions, while the image are obtained multiplying the images of the passed distributions.
- Factor (const Distribution &toMarginalize, const categoric::Combination &comb, const std::set< categoric↩ ::VariablePtr > &evidences)

  Builds the factor by taking only the combinations of the passed distribution matching with the passed combination. Suppose toMarginalize has a domain of variables equal to <A,B,C,D> and the passed comb is <0,1> and evidences is <B,C>. The built factor will have a domain of variables equal to <A,D>, with the combinations, raw images of toMarginalize (taking only the part referring to A,D) that have B=0 and C=1.
- Factor (const std::set< categoric::VariablePtr > &group, const std::string &fileName)

  Build the Factor by importing the information from the file.

Protected Member Functions

- **Factor** (const std::set< categoric::VariablePtr > &group)

Static Protected Member Functions

- template<typename ... Distributions>
  static std::set< const Distribution ∗ > **pack** (const Distribution ∗first, const Distribution ∗second, Distributions ... distr)
- template<typename ... Distributions>
  static void **pack** (std::set< const Distribution ∗ > &packed, const Distribution ∗first, Distributions ... distr)
- static void **pack** (std::set< const Distribution ∗ > &packed, const Distribution ∗first)

Additional Inherited Members

9.28.1 Detailed Description

A factor using the EvaluatorBasic object to convert the raw images into images.

9.28.2 Constructor & Destructor Documentation

9.28.2.1 Factor() [1/2]

```
EFG::distribution::factor::cnst::Factor::Factor (
            const std::set< categoric::VariablePtr > & group,
            bool corrOrAnti )
```

Builds a simple correlating or anticorrelating factor.

Parameters

the	variables representing the domain of this distribution
when	passing: • true, a simple correlating potential is built. Such a distribution has the images equal to 1 only for those combinations for which the variables have all the same values (<1,1,1>, <2,2,2>, <0,0>, etc...) and 0 for all the others • false, a simple anticorrelating potential is built. Such a distribution has the images equal to 0 for those combinations for which the variables have all the same values (<1,1,1>, <2,2,2>, <0,0>, etc...) and 1 for all the others

9.28.2.2 Factor() [2/2]

```
EFG::distribution::factor::cnst::Factor::Factor (
            const std::set< categoric::VariablePtr > & group,
            const std::string & fileName )
```

Build the Factor by importing the information from the file.

Parameters

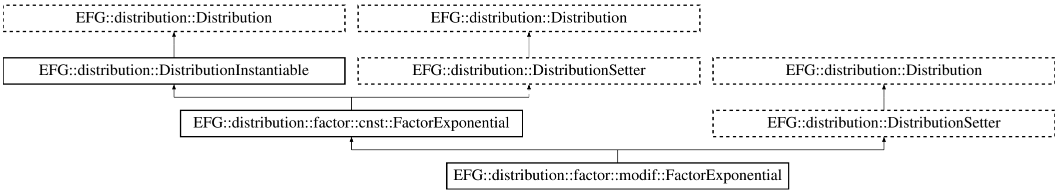
the	group of variables describing the domain
the	location of a file storing the combinations and the raw images in a matrix of numbers: each row has the combination values and at the end the raw image

The documentation for this class was generated from the following file:

- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/distribution/factor/const/Factor.h

9.29 EFG::distribution::factor::modif::FactorExponential Class Reference

Inheritance diagram for EFG::distribution::factor::modif::FactorExponential:



Public Member Functions

- **FactorExponential** (const cnst::Factor &factor, float weight)
- **FactorExponential** (const cnst::FactorExponential &o)
- **FactorExponential** (const FactorExponential &o)
- FactorExponential & **operator=** (const FactorExponential &o)
- void setWeight (float w)

  sets the weight used by teh exponential function converting the raw images

Additional Inherited Members

The documentation for this class was generated from the following file:

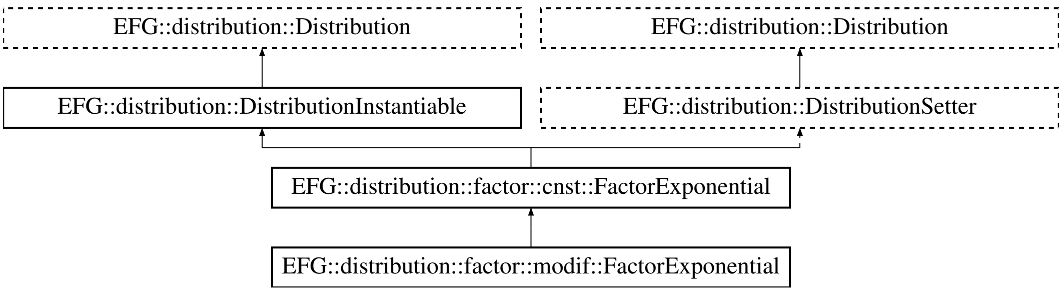
- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/distribution/factor/modifiable/FactorExponential.↩ h

9.30 EFG::distribution::factor::cnst::FactorExponential Class Reference

A factor using the EvaluatorExponential object to convert the raw images into images.

```
#include <FactorExponential.h>
```

Inheritance diagram for EFG::distribution::factor::cnst::FactorExponential:



Public Member Functions

- FactorExponential (const Factor &factor, float weight)
- **FactorExponential** (const FactorExponential &o)
- float getWeight () const

Additional Inherited Members

9.30.1 Detailed Description

A factor using the EvaluatorExponential object to convert the raw images into images.

9.30.2 Constructor & Destructor Documentation

9.30.2.1 FactorExponential()

```
EFG::distribution::factor::cnst::FactorExponential::FactorExponential (
            const Factor & factor,
            float weight )
```

Parameters

the	factor whose raw images are copied
the	weight to pass to the EvaluatorExponential

9.30.3 Member Function Documentation

9.30.3.1 getWeight()

```
float EFG::distribution::factor::cnst::FactorExponential::getWeight ( ) const
```

Returns

the weight of the EvaluatorExponential

The documentation for this class was generated from the following file:

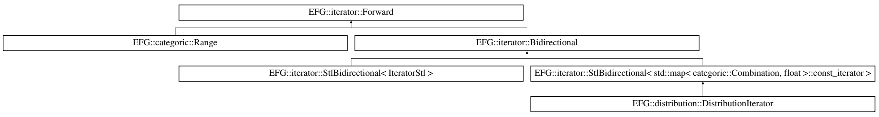
- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/distribution/factor/const/FactorExponential.h

9.31 EFG::iterator::Forward Class Reference

A Forward iterable object.

```
#include <Forward.h>
```

Inheritance diagram for EFG::iterator::Forward:



Public Member Functions

- virtual void **operator++** ()=0
- virtual bool operator== (std::nullptr_t) const =0
- bool **operator!=** (std::nullptr_t) const

9.31.1 Detailed Description

A Forward iterable object.

9.31.2 Member Function Documentation

9.31.2.1 operator==()

```
virtual bool EFG::iterator::Forward::operator== (
            std::nullptr_t ) const [pure virtual]
```

Returns

true when the iterator is at the end, i.e. can't be incremented further.

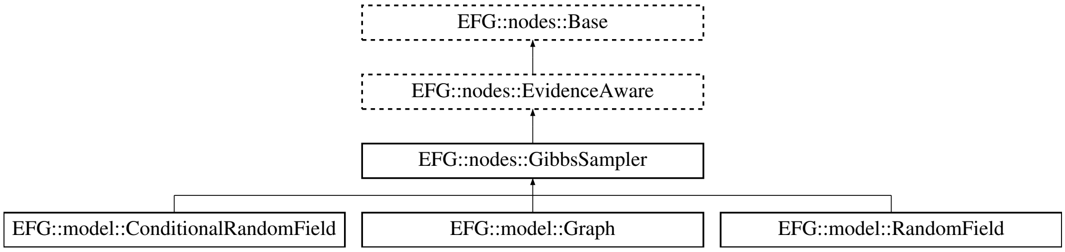
Implemented in EFG::categoric::Range, EFG::iterator::StlBidirectional< IteratorStl >, and EFG::iterator::StlBidirectional< std::map<

The documentation for this class was generated from the following file:

- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/iterator/Forward.h

9.32 EFG::nodes::GibbsSampler Class Reference

Inheritance diagram for EFG::nodes::GibbsSampler:



Public Member Functions

- std::vector< categoric::Combination > getHiddenSetSamples (std::size_t numberOfSamples, std::size_↩ t deltaIteration=100) const

    Use Gibbs sampling to draw samples for the hidden variables, conditioned to the current evidences.

Additional Inherited Members

9.32.1 Member Function Documentation

9.32.1.1 getHiddenSetSamples()

```
std::vector<categoric::Combination> EFG::nodes::GibbsSampler::getHiddenSetSamples (
            std::size_t numberOfSamples,
            std::size_t deltaIteration = 100 ) const
```

Use Gibbs sampling to draw samples for the hidden variables, conditioned to the current evidences.

Parameters

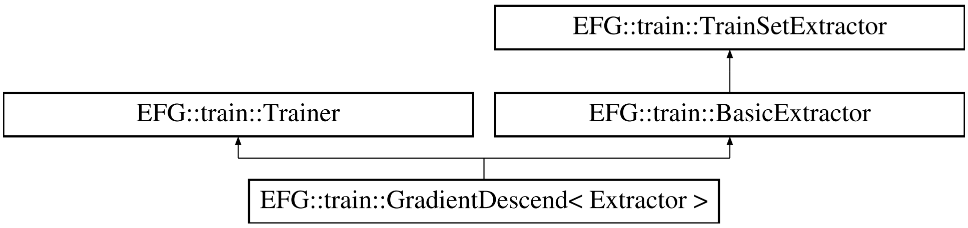
number	of samples to draw
number	of iterations used to evolve the model between the drawing of one sample and another

The documentation for this class was generated from the following file:

- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/nodes/GibbsSampler.h

9.33 EFG::train::GradientDescend< Extractor > Class Template Reference

Inheritance diagram for EFG::train::GradientDescend< Extractor >:



Public Member Functions

- void train (Trainable &model, TrainSetPtr trainSet) override

  trains the passed model, using the passed training set
- float **getAdvancement** () const
- void **setAdvancement** (float adv)

Additional Inherited Members

9.33.1 Member Function Documentation

9.33.1.1 train()

```
template<typename Extractor = BasicExtractor>
void EFG::train::GradientDescend< Extractor >::train (
            Trainable & model,
            TrainSetPtr trainSet )  [inline], [override], [virtual]
```

trains the passed model, using the passed training set

Parameters

the	model to train
the	training set to use

Implements EFG::train::Trainer.

The documentation for this class was generated from the following file:

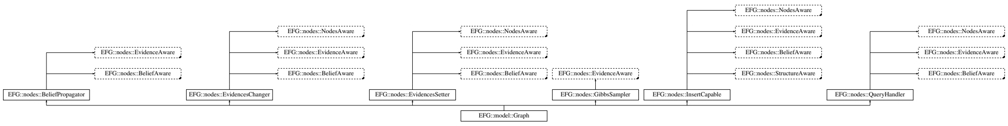
- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/train/trainers/GradientDescend.h

9.34 EFG::model::Graph Class Reference

A simple graph object, that can't store tunable factors.

```
#include <Graph.h>
```

Inheritance diagram for EFG::model::Graph:



Public Member Functions

- template< typename Model >
  Graph (const Model &o)
- **Graph** (const Graph &o)

Additional Inherited Members

9.34.1 Detailed Description

A simple graph object, that can't store tunable factors.

The documentation for this class was generated from the following file:

- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/model/Graph.h

9.35 EFG::categoric::Group Class Reference

An ensemble of categoric variables. Each variable in the ensemble should have its own unique name.

```
#include <Group.h>
```

Public Member Functions

- Group (const std::set< VariablePtr > &group)
- Group (VariablePtr var)
- Group (VariablePtr varA, VariablePtr varB)
- template<typename ... Vars>
  Group (VariablePtr varA, VariablePtr varB, Vars ... vars)
- **Group** (const Group &)=default
- Group & operator= (const Group &o)
- bool **operator==** (const Group &o) const
- void add (VariablePtr var)
- void replace (const std::set< VariablePtr > &newGroup)

  replaces the group of variables.
- template<typename ... Vars>
  void replace (VariablePtr varA, VariablePtr varB, Vars ... vars)

  replaces the group of variables.
- std::size_t size () const
- const std::set< VariablePtr > & **getVariables** () const

Protected Member Functions

- template<typename ... Vars>
  void **add** (VariablePtr var, Vars ... vars)

Protected Attributes

- std::set< VariablePtr > **group**

9.35.1 Detailed Description

An ensemble of categoric variables. Each variable in the ensemble should have its own unique name.

9.35.2 Constructor & Destructor Documentation

9.35.2.1 Group() [1/4]

```
EFG::categoric::Group::Group (
            const std::set< VariablePtr > & group )  [explicit]
```

Parameters

the	initial variables of the group

9.35.2.2 Group() [2/4]

```
EFG::categoric::Group::Group (
            VariablePtr var )  [explicit]
```

Parameters

the	initial variable to put in the group

9.35.2.3 Group() [3/4]

```
EFG::categoric::Group::Group (
            VariablePtr varA,
            VariablePtr varB )
```

Parameters

the	first initial variable to put in the group
the	second initial variable to put in the group

Exceptions

when	the 2 variables have the same names

9.35.2.4 Group() [4/4]

```
template<typename ... Vars>
EFG::categoric::Group::Group (
            VariablePtr varA,
            VariablePtr varB,
            Vars ... vars ) [inline]
```

Parameters

the	first initial variable to put in the group
the	second initial variable to put in the group
all	the other initial variables

9.35.3 Member Function Documentation

9.35.3.1 add()

```
void EFG::categoric::Group::add (
            VariablePtr var )
```

Parameters

the	variable to add in the group

Exceptions

in	case a variable with the same name is already part of the group

9.35.3.2 operator=()

```
Group& EFG::categoric::Group::operator= (
            const Group & o )
```

Exceptions

In	case of size mismatch with the previous variables set: the sizes of the 2 groups should be the same and the elements in the same positions must have the same domain size.

9.35.3.3 replace() [1/2]

```
void EFG::categoric::Group::replace (
            const std::set< VariablePtr > & newGroup )  [inline]
```

replaces the group of variables.

Exceptions

In	case of size mismatch with the previous variables set: the sizes of the 2 groups should be the same and the elements in the same positions must have the same domain size

9.35.3.4 replace() [2/2]

```
template<typename ...  Vars>
void EFG::categoric::Group::replace (
            VariablePtr varA,
            VariablePtr varB,
            Vars ...  vars )  [inline]
```

replaces the group of variables.

Exceptions

In	case of size mismatch with the previous variables set: the sizes of the 2 groups should be the same and the elements in the same positions must have the same domain size

9.35.3.5 size()

```
std::size_t EFG::categoric::Group::size ( ) const
```

Returns

the size of the joint domain of the group. For example the group <A,B,C> with sizes <2,4,3> will have a joint domain of size 2x4x3 = 24

The documentation for this class was generated from the following file:

- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/categoric/Group.h

9.36 EFG::nodes::HiddenClusters Struct Reference

Public Member Functions

- **HiddenClusters** (const std::set< Node ∗ > &toSplit)
- void **add** (const std::list< std::set< Node ∗ >> &toAdd)
- std::list< std::set< Node ∗ > >::iterator **find** (Node &node)

Public Attributes

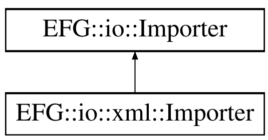
- std::list< std::set< Node ∗ > > **clusters**

The documentation for this struct was generated from the following file:

- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/nodes/bases/EvidenceAware.h

9.37 EFG::io::xml::Importer Class Reference

Inheritance diagram for EFG::io::xml::Importer:



Static Public Member Functions

- template<typename Model >
  static std::map< std::string, std::size_t > importFromXml (Model &model, const std::string &filePath, const std::string &fileName)

  imports the structure (variables and factors) described in an xml file and add it to the passed model

Additional Inherited Members

9.37.1 Member Function Documentation

9.37.1.1 importFromXml()

```
template<typename Model >
static std::map<std::string, std::size_t> EFG::io::xml::Importer::importFromXml (
            Model & model,
            const std::string & filePath,
            const std::string & fileName )  [inline], [static]
```

imports the structure (variables and factors) described in an xml file and add it to the passed model

Parameters

the	model receiving the parsed data
the	folder storing the xml
the	xml file name

Returns

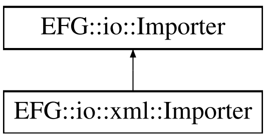
the set of evidences red from the file. Attention! this quantities are returned to allow the user to set such evidence, since this is not automatically done when importing

The documentation for this class was generated from the following file:

- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/io/xml/Importer.h

9.38 EFG::io::Importer Class Reference

Inheritance diagram for EFG::io::Importer:



Protected Member Functions

- virtual std::map< std::string, std::size_t > **importComponents** (const std::string &filePath, const std::string &fileName, const std::pair< nodes::InsertCapable ∗, nodes::InsertTunableCapable ∗ > &components)=0

Static Protected Member Functions

- template<typename Model >
  static std::pair< nodes::InsertCapable ∗, nodes::InsertTunableCapable ∗ > **getComponents** (Model &model)

The documentation for this class was generated from the following file:

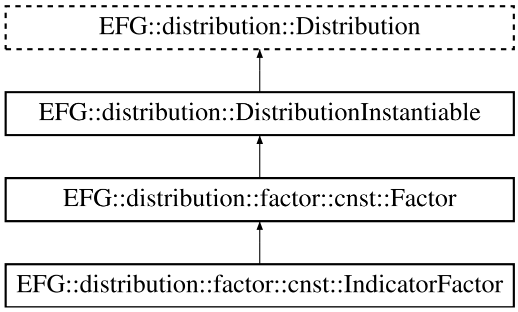
- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/io/Importer.h

9.39 EFG::distribution::factor::cnst::IndicatorFactor Class Reference

An indicator distirbution having only one combination explicitly stated, whose image is equal to 1.

```
#include <Indicator.h>
```

Inheritance diagram for EFG::distribution::factor::cnst::IndicatorFactor:



Public Member Functions

- IndicatorFactor (categoric::VariablePtr var, std::size_t evidence)

Additional Inherited Members

9.39.1 Detailed Description

An indicator distirbution having only one combination explicitly stated, whose image is equal to 1.

9.39.2 Constructor & Destructor Documentation

9.39.2.1 IndicatorFactor()

```
EFG::distribution::factor::cnst::IndicatorFactor::IndicatorFactor (
            categoric::VariablePtr var,
            std::size_t evidence )
```

Parameters

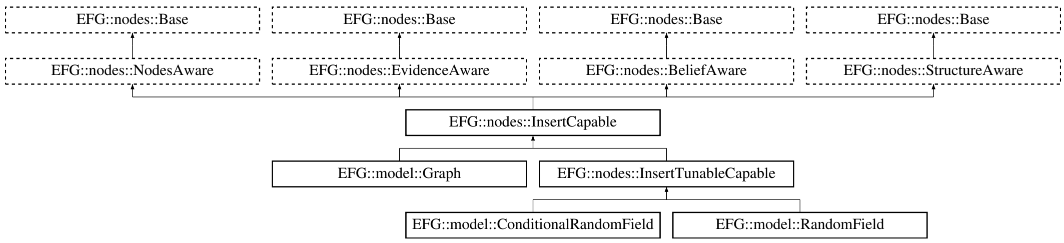
the	variable this indicator function must refer to
the	only combination to consider for the indicator distribution

The documentation for this class was generated from the following file:

- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/distribution/factor/const/Indicator.h

9.40 EFG::nodes::InsertCapable Class Reference

Inheritance diagram for EFG::nodes::InsertCapable:



Public Member Functions

- void insert (std::shared_ptr< distribution::factor::cnst::Factor > factor)

  insert the passed factor.
- void insertCopy (const distribution::Distribution &factor)

  insert a copy of the passed factor.
- void insert (std::shared_ptr< distribution::factor::cnst::FactorExponential > factor)

  insert the passed epxonential factor.
- void insertCopy (const distribution::factor::cnst::FactorExponential &factor)

  insert a copy of the passed exponential factor.
- template< typename Model >
  void absorbModel (const Model &model, const bool &useCopyInsertion=false)

  insert all the factors contained in the passed model

Protected Member Functions

- std::set< categoric::VariablePtr > **convertUsingLocals** (const std::set< categoric::VariablePtr > &to←
  Convert)
- void **absorb** (const StructureAware &toAbsorb, const bool &useCopyInsertion)
- virtual void **absorb** (const StructureTunableAware &toAbsorb, const bool &useCopyInsertion)

Additional Inherited Members

9.40.1 Member Function Documentation

9.40.1.1 absorbModel()

```
template<typename Model >
void EFG::nodes::InsertCapable::absorbModel (
             const Model & model,
             const bool & useCopyInsertion = false )   [inline]
```

insert all the factors contained in the passed model

Parameters

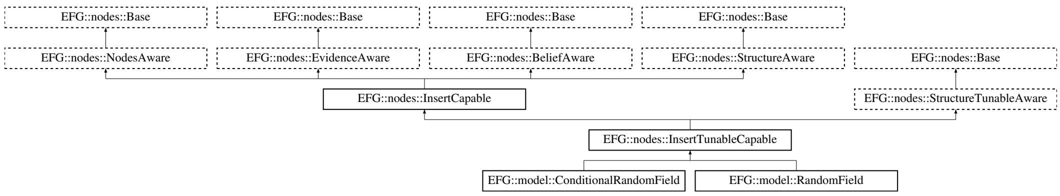
the	model to absorb
when	passing true the factors from the passed model are copied, otherwise the shared pointers are copied

The documentation for this class was generated from the following file:

- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/nodes/InsertCapable.h

9.41 EFG::nodes::InsertTunableCapable Class Reference

Inheritance diagram for EFG::nodes::InsertTunableCapable:



Public Member Functions

- void insertTunableCopy (const distribution::factor::modif::FactorExponential &factor)

  insert a copy of the passed tunable factor. A new tunable cluster is created containing only the passed factor.
- void insertTunableCopy (const distribution::factor::modif::FactorExponential &factor, const std::set< categoric::VariablePtr > &potentialSharingWeight)

  insert a copy of the passed tunable factor, sharing the weight with an already inserted one.

Protected Member Functions

- virtual void **insertTunable** (std::shared_ptr< distribution::factor::modif::FactorExponential > toInsert)
- virtual void **insertTunable** (std::shared_ptr< distribution::factor::modif::FactorExponential > toInsert, const std::set< categoric::VariablePtr > &potentialSharingWeight)
- void **absorb** (const StructureTunableAware &toAbsorb, const bool &useCopyInsertion) override

Additional Inherited Members

9.41.1 Member Function Documentation

9.41.1.1 insertTunableCopy()

```
void EFG::nodes::InsertTunableCapable::insertTunableCopy (
            const distribution::factor::modif::FactorExponential & factor,
            const std::set< categoric::VariablePtr > & potentialSharingWeight )
```

insert a copy of the passed tunable factor, sharing the weight with an already inserted one.

Parameters

the	factor to insert
the	set of variables identifying the potential whose weight is to share

The documentation for this class was generated from the following file:

- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/nodes/InsertTunableCapable.h

9.42 EFG::nodes::Node Struct Reference

Public Member Functions

- **Node** (categoric::VariablePtr var)

Public Attributes

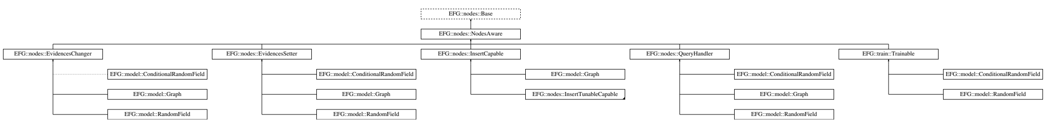
- categoric::VariablePtr **variable**
- std::list< distribution::DistributionPtr > **unaryFactors**
- std::map< Node *, Connection > **activeConnections**
- std::map< Node *, Connection > **disabledConnections**

The documentation for this struct was generated from the following file:

- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/nodes/Node.h

9.43 EFG::nodes::NodesAware Class Reference

Inheritance diagram for EFG::nodes::NodesAware:



Public Member Functions

- std::set< categoric::VariablePtr > getVariables () const
- categoric::VariablePtr **findVariable** (const std::string &name) const

Protected Attributes

- std::map< categoric::VariablePtr, Node > nodes

  The set of variables part of the model, with the connectivity information.

9.43.1 Member Function Documentation

9.43.1.1 getVariables()

`std::set<categoric::VariablePtr> EFG::nodes::NodesAware::getVariables ( ) const`

Returns

all the variables (hidden or observed) in the model

The documentation for this class was generated from the following file:

- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/nodes/bases/NodesAware.h

9.44 EFG::nodes::PropagationResult Struct Reference

Public Attributes

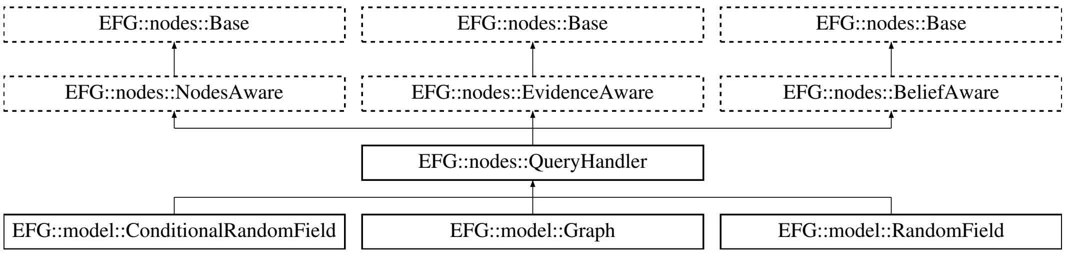
- PropagationKind **kindDone**
- bool **wasTerminated**

The documentation for this struct was generated from the following file:

- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/nodes/bases/BeliefAware.h

9.45 EFG::nodes::QueryHandler Class Reference

Inheritance diagram for EFG::nodes::QueryHandler:



Public Member Functions

- std::vector< float > getMarginalDistribution (const std::string &var)
- distribution::factor::cnst::Factor getJointMarginalDistribution (const std::set< std::string > &subgroup)
- std::size_t getMAP (const std::string &var)
- std::vector< size_t > getHiddenSetMAP ()

Additional Inherited Members

9.45.1 Member Function Documentation

9.45.1.1 getHiddenSetMAP()

```
std::vector<size_t> EFG::nodes::QueryHandler::getHiddenSetMAP ( )
```

Returns

the Maximum a Posteriori estimation of the hidden variables, conditioned to the last set evidences. values are ordered in the same way the variables in the hiddden set can be ordered (alfabetic order)

9.45.1.2 getJointMarginalDistribution()

```
distribution::factor::cnst::Factor EFG::nodes::QueryHandler::getJointMarginalDistribution (
            const std::set< std::string > & subgroup )
```

Returns

a factor representing the joint distribution of the subgraph described by the passed variables.

9.45.1.3 getMAP()

```
std::size_t EFG::nodes::QueryHandler::getMAP (
            const std::string & var )
```

Returns

the Maximum a Posteriori estimation of a specific variable in the model, conditioned to the last set evidences.

9.45.1.4 getMarginalDistribution()

```
std::vector<float> EFG::nodes::QueryHandler::getMarginalDistribution (
            const std::string & var )
```

Returns

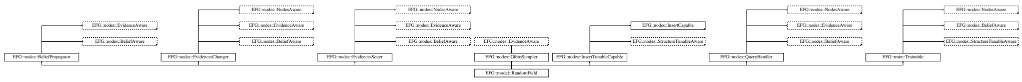
the marginal probabilty of the passed variable, i.e. P(var|model, observations), conditioned to the last set evidences.

The documentation for this class was generated from the following file:

- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/nodes/QueryHandler.h

9.46 EFG::model::RandomField Class Reference

Inheritance diagram for EFG::model::RandomField:



Public Member Functions

- template< typename Model >
  RandomField (const Model &o)
- **RandomField** (const RandomField &o)
- void insertTunable (std::shared_ptr< distribution::factor::modif::FactorExponential > toInsert) override

  insert the passed tunable factor.
- void insertTunable (std::shared_ptr< distribution::factor::modif::FactorExponential > toInsert, const std::set< categoric::VariablePtr > &potentialSharingWeight) override

  insert the passed tunable factor, sharing the weight with an already inserted one.
- std::vector< float > getGradient (train::TrainSetPtr trainSet) override

Additional Inherited Members

9.46.1 Member Function Documentation

9.46.1.1 getGradient()

```
std::vector<float> EFG::model::RandomField::getGradient (
            train::TrainSetPtr trainSet )  [override], [virtual]
```

Returns

the gradient of the weights of the tunable clusters w.r.t. the passed training set

Implements EFG::train::Trainable.

9.46.1.2 insertTunable()

```
void EFG::model::RandomField::insertTunable (
            std::shared_ptr< distribution::factor::modif::FactorExponential > toInsert,
            const std::set< categoric::VariablePtr > & potentialSharingWeight )  [override],
[virtual]
```

insert the passed tunable factor, sharing the weight with an already inserted one.

Parameters

the	factor to insert
the	set of variables identifying the potential whose weight is to share

Reimplemented from EFG::nodes::InsertTunableCapable.

The documentation for this class was generated from the following file:

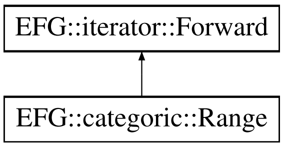
- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/model/RandomField.h

9.47 EFG::categoric::Range Class Reference

This object allows you to iterate all the elements in the joint domain of a group of variables, without precomputing all the elements in such domain. For example when having a domain made by variables = { A (size = 2), B (size = 3), C (size = 2) }, the elements in the joint domain that will be iterated are: <0,0,0> <0,0,1> <0,1,0> <0,1,1> <0,2,0> <0,2,1> <1,0,0> <1,0,1> <1,1,0> <1,1,1> <1,2,0> <1,2,1> After construction, the Range object starts to point to the first element in the joint domain <0,0,...>. Then, when incrementing the object, the following element is pointed. When calling get() the current pointed element can be accessed.

```
#include <Range.h>
```

Inheritance diagram for EFG::categoric::Range:



Public Member Functions

- Range (const std::set< VariablePtr > &group)
- **Range** (const Range &)=default
- Range & **operator=** (const Range &)=default
- const Combination & get () const
- void operator++ () final

  Make the object to point to the next element in the joint domain.
- bool operator== (std::nullptr_t) const final
- void reset ()

  Make the object to point to the first element of the joint domain <0,0,...>, i.e. reset the status as it is after construction.

9.47.1 Detailed Description

This object allows you to iterate all the elements in the joint domain of a group of variables, without precomputing all the elements in such domain. For example when having a domain made by variables = { A (size = 2), B (size = 3), C (size = 2) }, the elements in the joint domain that will be iterated are: <0,0,0> <0,0,1> <0,1,0> <0,1,1> <0,2,0> <0,2,1> <1,0,0> <1,0,1> <1,1,0> <1,1,1> <1,2,0> <1,2,1> After construction, the Range object starts to point to the first element in the joint domain <0,0,...>. Then, when incrementing the object, the following element is pointed. When calling get() the current pointed element can be accessed.

9.47.2 Constructor & Destructor Documentation

9.47.2.1 Range()

```
EFG::categoric::Range::Range (
            const std::set< VariablePtr > & group )  [explicit]
```

Parameters

the	group of variables whose joint domain must be iterated

9.47.3 Member Function Documentation

9.47.3.1 get()

```
const Combination& EFG::categoric::Range::get ( ) const  [inline]
```

Returns

the current pointed combination

9.47.3.2 operator++()

```
void EFG::categoric::Range::operator++ ( )  [final], [virtual]
```

Make the object to point to the next element in the joint domain.

Exceptions

if	the current pointed element is the last one.

Implements EFG::iterator::Forward.

9.47.3.3 operator==()

```
bool EFG::categoric::Range::operator== (
            std::nullptr_t  ) const  [inline], [final], [virtual]
```

Returns

true when the iterator is at the end, i.e. can't be incremented further.

Implements EFG::iterator::Forward.

The documentation for this class was generated from the following file:

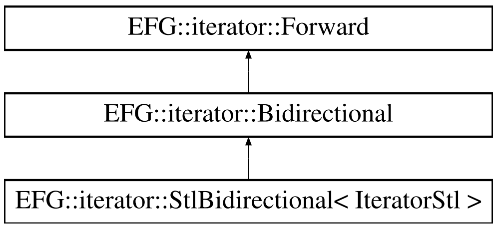
- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/categoric/Range.h

9.48 EFG::iterator::StlBidirectional< IteratorStl > Class Template Reference

A bidirectional iterator built on top of an std iterator type.

```
#include <StlBidirectional.h>
```

Inheritance diagram for EFG::iterator::StlBidirectional< IteratorStl >:



Public Member Functions

- **StlBidirectional** (const IteratorStl &begin, const IteratorStl &end)
- **StlBidirectional** (const StlBidirectional &)=default
- StlBidirectional & **operator=** (const StlBidirectional &)=default
- void **operator++** () final
- void **operator--** () final
- bool operator== (std::nullptr_t) const final

Protected Attributes

- IteratorStl **cursor**
- const IteratorStl **end**

9.48.1 Detailed Description

template<typename IteratorStl>
class EFG::iterator::StlBidirectional< IteratorStl >

A bidirectional iterator built on top of an std iterator type.

9.48.2 Member Function Documentation

9.48.2.1 operator==()

```
template<typename IteratorStl >
bool EFG::iterator::StlBidirectional< IteratorStl >::operator== (
            std::nullptr_t ) const [inline], [final], [virtual]
```

Returns

true when the iterator is at the end, i.e. can't be incremented further.

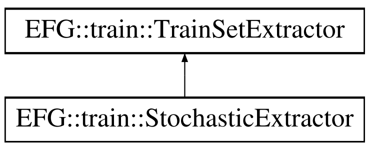
Implements EFG::iterator::Forward.

The documentation for this class was generated from the following file:

- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/iterator/StlBidirectional.h

9.49 EFG::train::StochasticExtractor Class Reference

Inheritance diagram for EFG::train::StochasticExtractor:



Public Member Functions

- void **setPercentage** (const float &percentage)

Protected Member Functions

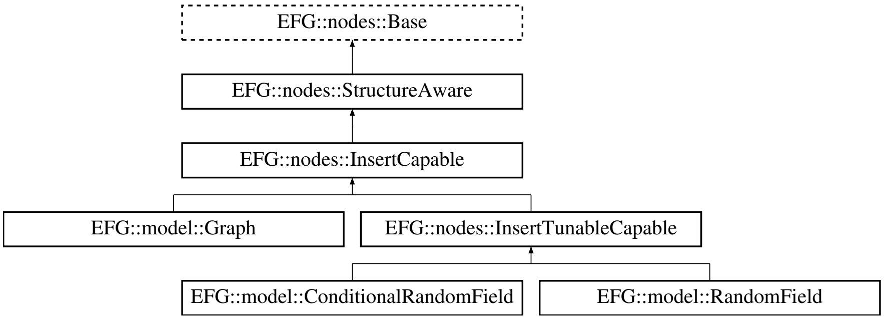
- TrainSetPtr **getTrainSet** () override

The documentation for this class was generated from the following file:

- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/train/trainers/TrainSetExtractor.h

9.50 EFG::nodes::StructureAware Class Reference

Inheritance diagram for EFG::nodes::StructureAware:



Public Member Functions

- const std::set< std::shared_ptr< distribution::factor::cnst::Factor > > & **getFactors** () const
- const std::set< std::shared_ptr< distribution::factor::cnst::FactorExponential > > & **getFactorsExp** () const

Protected Attributes

- std::set< std::shared_ptr< distribution::factor::cnst::Factor > > factors
- std::set< std::shared_ptr< distribution::factor::cnst::FactorExponential > > factorsExp

9.50.1 Member Data Documentation

9.50.1.1 factors

```
std::set<std::shared_ptr<distribution::factor::cnst::Factor> > EFG::nodes::StructureAware↩
::factors  [protected]
```

Returns

the constant factors stored in the model

9.50.1.2 factorsExp

```
std::set<std::shared_ptr<distribution::factor::cnst::FactorExponential> > EFG::nodes::↩
StructureAware::factorsExp  [protected]
```

Returns

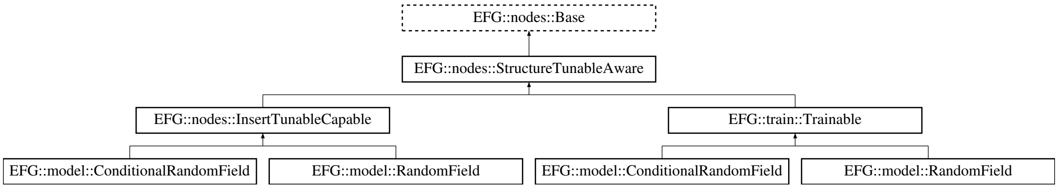
the constant exponential factors stored in the model

The documentation for this class was generated from the following file:

- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/nodes/bases/StructureAware.h

9.51 EFG::nodes::StructureTunableAware Class Reference

Inheritance diagram for EFG::nodes::StructureTunableAware:



Public Member Functions

- std::vector< std::vector< std::shared_ptr< distribution::factor::modif::FactorExponential > > > getFactorsTunable () const
- std::vector< float > getWeights () const

Protected Attributes

- std::size_t **numberOfClusters** = 0
- std::map< std::shared_ptr< distribution::factor::modif::FactorExponential >, std::size_t > factorsTunable

9.51.1 Member Function Documentation

9.51.1.1 getFactorsTunable()

```
std::vector<std::vector<std::shared_ptr<distribution::factor::modif::FactorExponential> > >
EFG::nodes::StructureTunableAware::getFactorsTunable ( ) const
```

Returns

the clusters of tunable exponential factors. Each element in the returned vector, is a cluster of exponential factors sharing the same weight value.

9.51.1.2 getWeights()

```
std::vector<float> EFG::nodes::StructureTunableAware::getWeights ( ) const
```

Returns

the weights of the tuanble clusters. For each cluster only 1 value is returned, since it is shared among the elements in the same cluster.

9.51.2 Member Data Documentation

9.51.2.1 factorsTunable

```
std::map<std::shared_ptr<distribution::factor::modif::FactorExponential>, std::size_t> EFG↩
::nodes::StructureTunableAware::factorsTunable [protected]
```

Returns

collection of tunable exponential factors. Each element is <exp factor, cluster id>

The documentation for this class was generated from the following file:

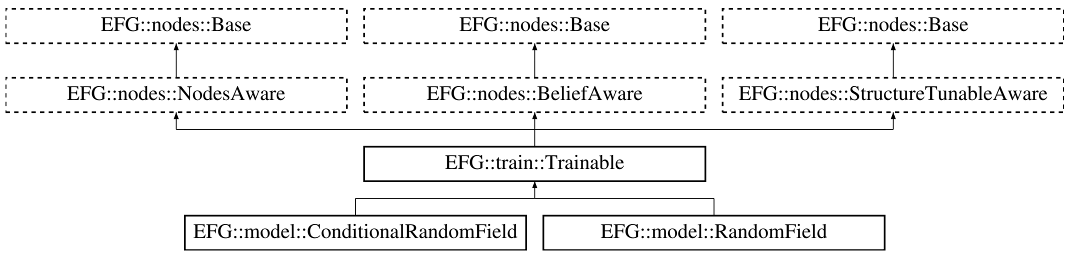
- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/nodes/bases/StructureTunableAware.h

9.52 EFG::train::Trainable Class Reference

An object storing tunable factors, whose weights can be tuned with training.

```
#include <Trainable.h>
```

Inheritance diagram for EFG::train::Trainable:



Public Member Functions

- void setWeights (const std::vector< float > &w)
- void setOnes ()
- virtual std::vector< float > getGradient (TrainSetPtr trainSet)=0

Protected Member Functions

- virtual TrainHandlerPtr **makeHandler** (std::shared_ptr< distribution::factor::modif::FactorExponential > factor)
- void **insertHandler** (std::shared_ptr< distribution::factor::modif::FactorExponential > factor)
- void **setTrainSet** (TrainSetPtr newSet)
- TrainSetPtr **getLastTrainSet** () const

Protected Attributes

- std::list< TrainHandlerPtr > **handlers**

9.52.1 Detailed Description

An object storing tunable factors, whose weights can be tuned with training.

9.52.2 Member Function Documentation

9.52.2.1 getGradient()

```
virtual std::vector<float> EFG::train::Trainable::getGradient (
            TrainSetPtr trainSet )  [pure virtual]
```

Returns

the gradient of the weights of the tunable clusters w.r.t. the passed training set

Implemented in EFG::model::RandomField.

9.52.2.2 setOnes()

```
void EFG::train::Trainable::setOnes ( )
```

Parameters

sets	equal to 1 the weight of all the tunable clusters

9.52.2.3 setWeights()

```
void EFG::train::Trainable::setWeights (
            const std::vector< float > & w )
```

Parameters

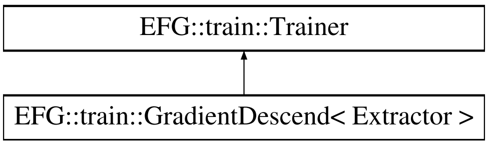
the	new set of weights to assume for the tunable clusters

The documentation for this class was generated from the following file:

- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/train/Trainable.h

9.53 EFG::train::Trainer Class Reference

Inheritance diagram for EFG::train::Trainer:



Public Member Functions

- virtual void train (Trainable &model, TrainSetPtr trainSet)=0

  trains the passed model, using the passed training set
- std::size_t **getMaxIterations** () const
- void **setMaxIterations** (std::size_t iter)

Protected Attributes

- std::size_t **maxIterations** = 100

9.53.1 Member Function Documentation

9.53.1.1 train()

```
virtual void EFG::train::Trainer::train (
            Trainable & model,
            TrainSetPtr trainSet )  [pure virtual]
```

trains the passed model, using the passed training set

Parameters

the	model to train
the	training set to use

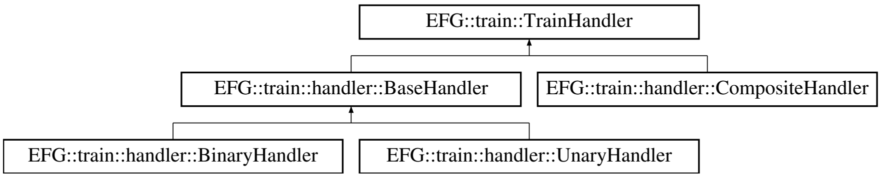
Implemented in EFG::train::GradientDescend$<$ Extractor $>$.

The documentation for this class was generated from the following file:

- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/train/Trainer.h

9.54 EFG::train::TrainHandler Class Reference

Inheritance diagram for EFG::train::TrainHandler:



Public Member Functions

- virtual void **setTrainSet** (TrainSetPtr trainSet, const std::set< categoric::VariablePtr > &modelVariables)=0
- virtual float **getGradientAlpha** ()=0
- virtual float **getGradientBeta** ()=0
- virtual void **setWeight** (const float &w)=0

The documentation for this class was generated from the following file:

- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/train/Trainable.h

9.55 EFG::train::TrainSet Class Reference

Public Member Functions

- TrainSet (const std::vector< categoric::Combination > &combinations)
- TrainSet (const std::string &fileName)
- TrainSet getRandomSubSet (const float &percentage) const
- const std::vector< CombinationPtr > & **getSet** () const

9.55.1 Constructor & Destructor Documentation

9.55.1.1 TrainSet() `[1/2]`

```
EFG::train::TrainSet::TrainSet (
            const std::vector< categoric::Combination > & combinations )  [explicit]
```

Parameters

the	set of combinations that will be part of the train set.

Exceptions

if	the combinations don't have all the same size

9.55.1.2 TrainSet() [2/2]

```
EFG::train::TrainSet::TrainSet (
            const std::string & fileName )  [explicit]
```

Parameters

import	the combinations from a textual file where each row represent a combination

9.55.2 Member Function Documentation

9.55.2.1 getRandomSubSet()

```
TrainSet EFG::train::TrainSet::getRandomSubSet (
            const float & percentage ) const
```

Returns

a TrainSet containg some of the combinations stored into this object. The combination to take are randomly decided.

Parameters

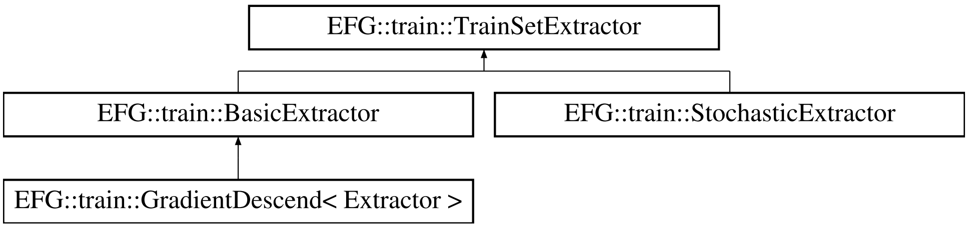
the	percentage of combinations to extract from this object.

The documentation for this class was generated from the following file:

- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/train/TrainSet.h

9.56 EFG::train::TrainSetExtractor Class Reference

Inheritance diagram for EFG::train::TrainSetExtractor:



Protected Member Functions

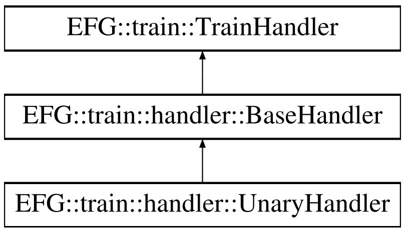
- virtual TrainSetPtr **getTrainSet** ()=0
- void **setCompleteTrainSet** (TrainSetPtr trainSet)
- TrainSetPtr **getCompleteTrainSet** ()

The documentation for this class was generated from the following file:

- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/train/trainers/TrainSetExtractor.h

9.57 EFG::train::handler::UnaryHandler Class Reference

Inheritance diagram for EFG::train::handler::UnaryHandler:



Public Member Functions

- **UnaryHandler** (nodes::Node &node, std::shared_ptr< distribution::factor::modif::FactorExponential > factor)
- float **getGradientBeta** () final

Protected Attributes

- nodes::Node * **node**

Additional Inherited Members

The documentation for this class was generated from the following file:

- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/train/handlers/UnaryHandler.h

9.58 EFG::categoric::Variable Class Reference

An object representing an immutable categoric variable.

```
#include <Variable.h>
```

Public Member Functions

- Variable (const std::size_t &size, const std::string &name)
- **Variable** (const Variable &)=default
- Variable & **operator=** (const Variable &)=delete
- bool **operator==** (const Variable &o) const
- std::size_t **size** () const
- const std::string & **name** () const

Protected Attributes

- const size_t **Size**
- const std::string **Name**

9.58.1 Detailed Description

An object representing an immutable categoric variable.

9.58.2 Constructor & Destructor Documentation

9.58.2.1 Variable()

```
EFG::categoric::Variable::Variable (
            const std::size_t & size,
            const std::string & name )
```

Parameters

domain	size of this variable
name	used to label this varaible.

Exceptions

passing	an empty string

The documentation for this class was generated from the following file:

- C:/Users/andre/Desktop/Easy-Factor-Graph/Lib/EFG/Header/categoric/Variable.h

Index