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 5, 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

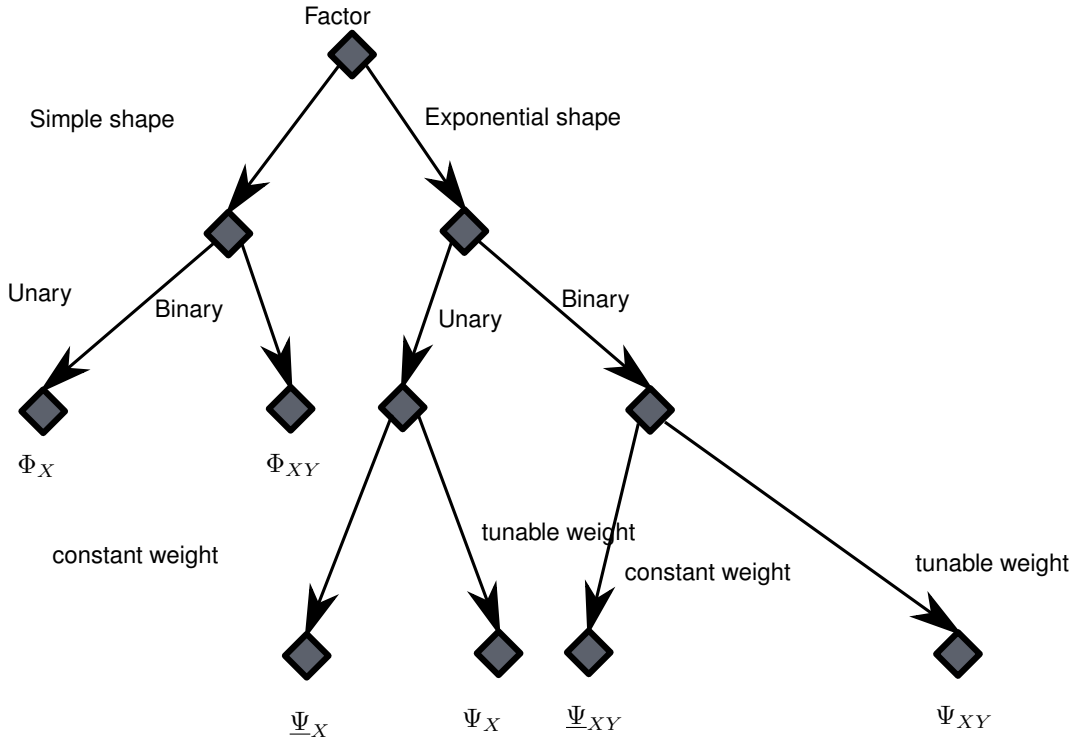


Figure 2.1 All the possible categories of factors, with the corresponding notation.

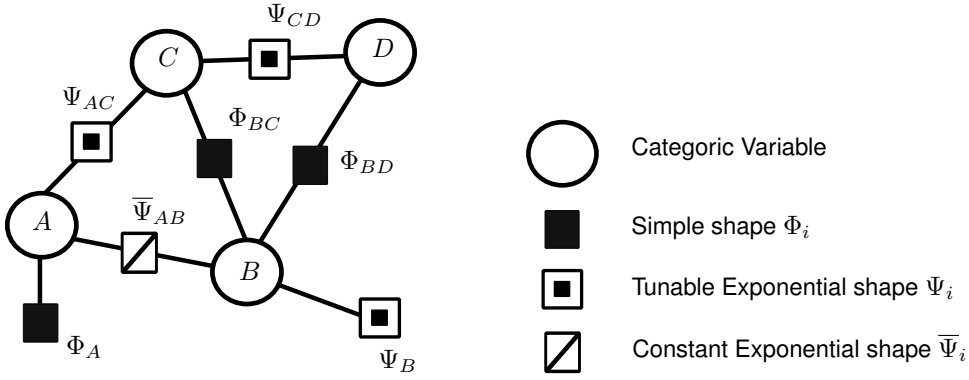


Figure 2.2 Example of graph: the legend of the right applies.

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.

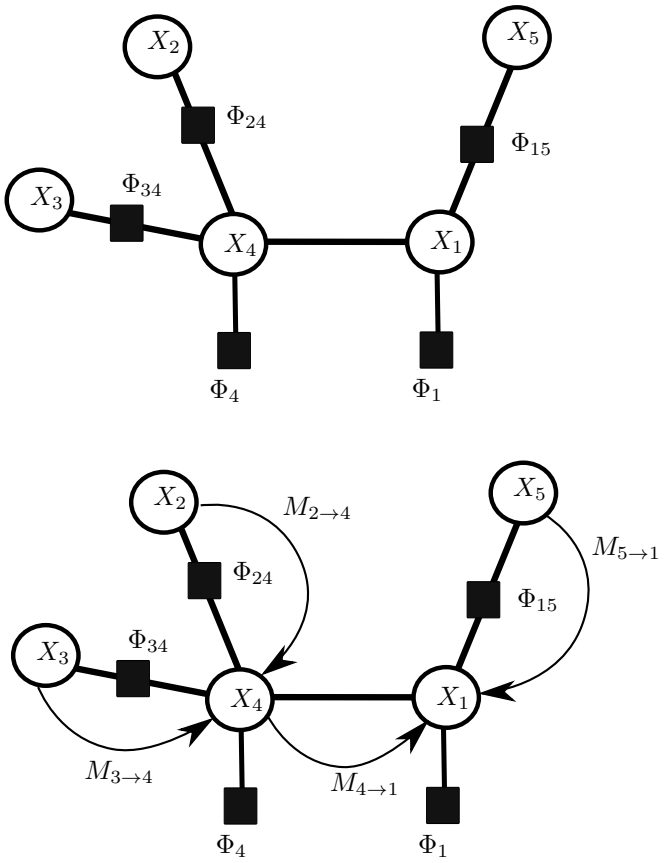


Figure 2.3 Example of graph adopted for explaining the message passing algorithm. Below are reported the messages to compute for obtaining the marginal probability of variable x_1

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' $M_{2\to4}$, $M_{3\to4}$ are defined as follows:

$$\begin{aligned} M_{2\to4}(\tilde{x}_4) &= \sum_{\tilde{x}_2} \Phi_{24}(\tilde{x}_{2,4}) \\ M_{3\to4}(\tilde{x}_4) &= \sum_{\tilde{x}_3} \Phi_{34}(\tilde{x}_{3,4}) \end{aligned} \tag{2.22}$$

Inserting $M_{2\to4}$ and $M_{3\to4}$ into equation (2.21) leads to:

$$\mathbb{P}(x_1) \propto \Phi_1(x_1) \sum_{\tilde{x}_5} \Phi_{15}(x_1, \tilde{x}_5) \sum_{\tilde{x}_4} \Phi_{14}(x_1, \tilde{x}_4) \Phi_4(\tilde{x}_4) M_{2\to4}(\tilde{x}_4) M_{3\to4}(\tilde{x}_4) \tag{2.23}$$

At this point the messages $M_{4\to1}$ and $M_{5\to1}$ can be computed in the following way:

$$M_{4\to1(x_1)} = \sum_{\tilde{x}_4} \Phi_{14}(x_1, \tilde{x}_4)\Phi_4(\tilde{x}_4)M_{2\to4}(\tilde{x}_4)M_{3\to4}(\tilde{x}_4)$$
$$M_{5\to1}(x_1) = \sum_{\tilde{x}_5} \Phi_{15}(x_1, \tilde{x}_5) \tag{2.24}$$

After inserting $M_{4\to1}$ and $M_{5\to1}$ into equation (2.23) we obtain:

$$\begin{aligned} \mathbb{P}(x_1) &\propto \Phi_1(x_1)M_{4\to1}(x_1)M_{5\to1}(x_1) \\ \mathbb{P}(x_1) &= \frac{\Phi_1(x_1)M_{4\to1}(x_1)M_{5\to1}(x_1)}{\sum_{\tilde{x}_1}\Phi_1(\tilde{x}_1)M_{4\to1}(\tilde{x}_1)M_{5\to1}(\tilde{x}_1)} \end{aligned} \tag{2.25}$$

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 $M_{4\to1}$ must be done after computing the messages $M_{2\to4}$ and $M_{3\to4}$, while $M_{5\to1}$ 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 $M_{B\to A}$ can be performed only after having computed all the messages $M_{V_{1,\cdots,m}\to B}$, i.e. the messages incoming from all the neighbours of B a part from A. Clearly $M_{B\to A}$ is computed as follows:

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

Essentially, it's like having simplified the graph: we can append to A the message $M_{B\to A}(a)$ as it's a Simple shape, deleting factor Ψ_{AB} and all the other portions of the graph, see Figure 2.4. In turn, $M_{B\to A}(a)$ 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:

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

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.

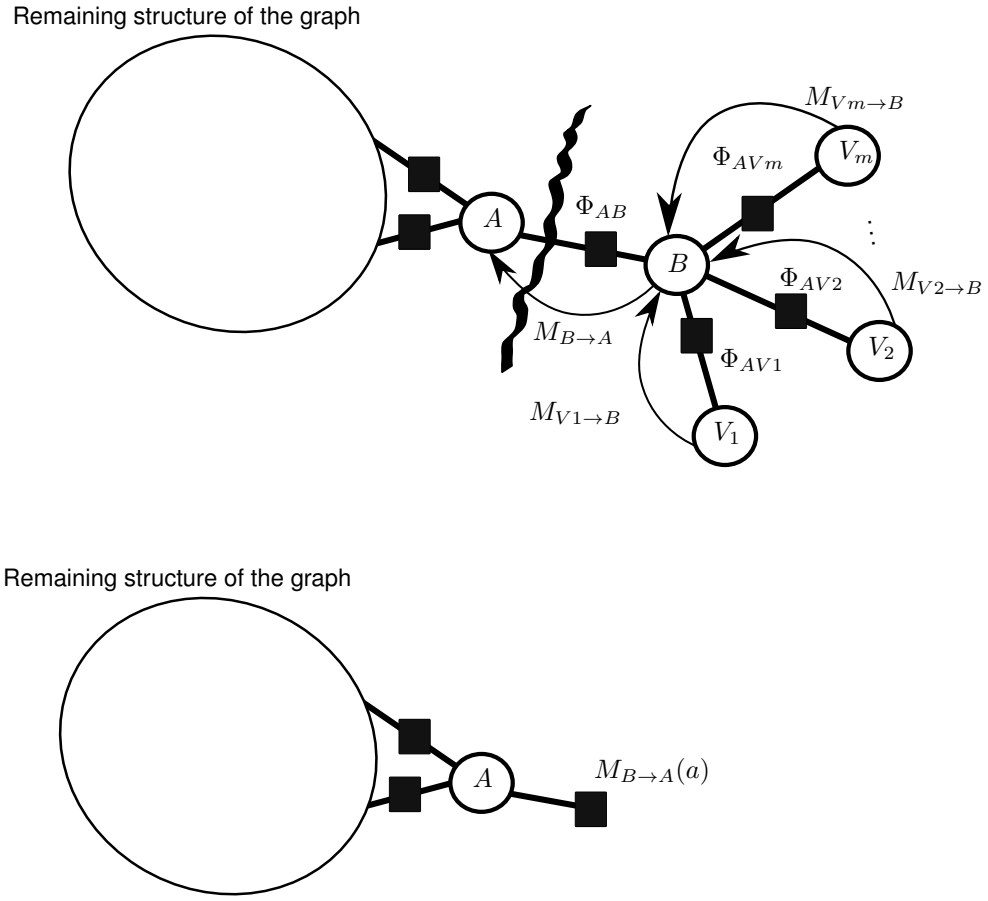


Figure 2.4 On the top the general mechanism involved in the message computation; on the bottom the simplification of the graph considering the computed message.

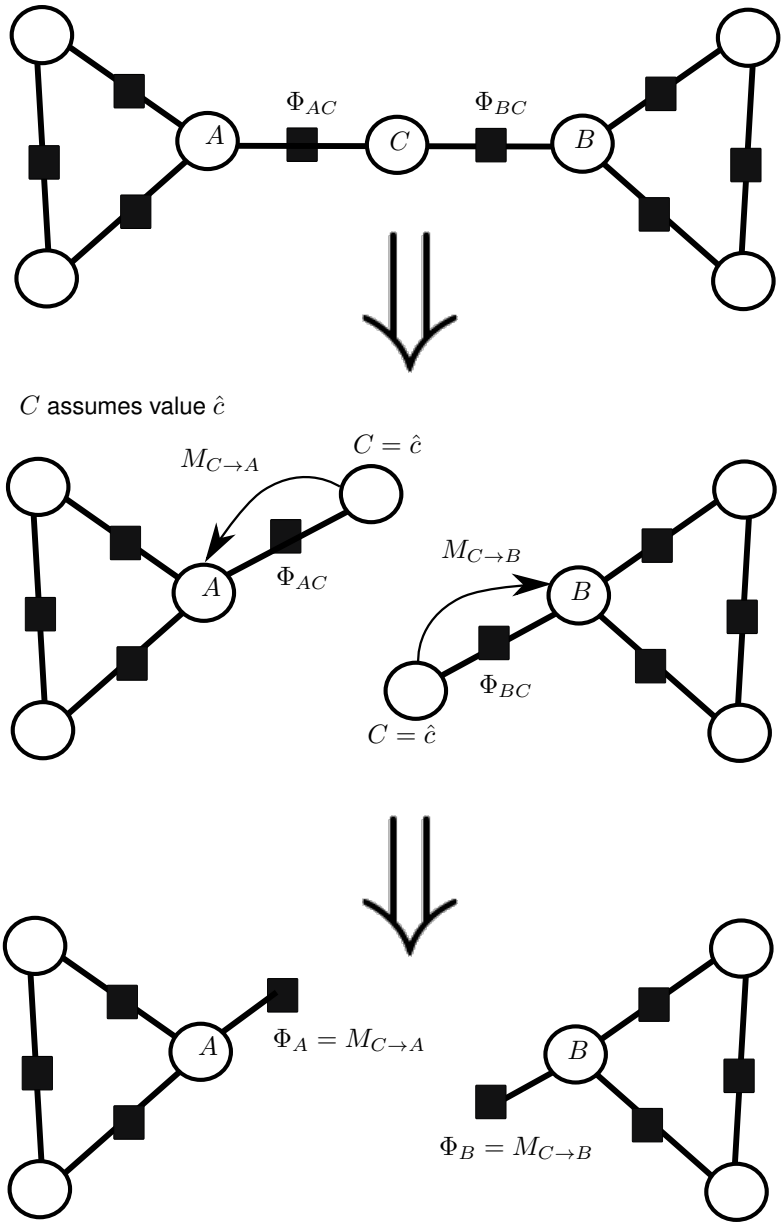


Figure 2.5 When variable C become an evidence, is temporary deleted from the graph, replaced by messages.

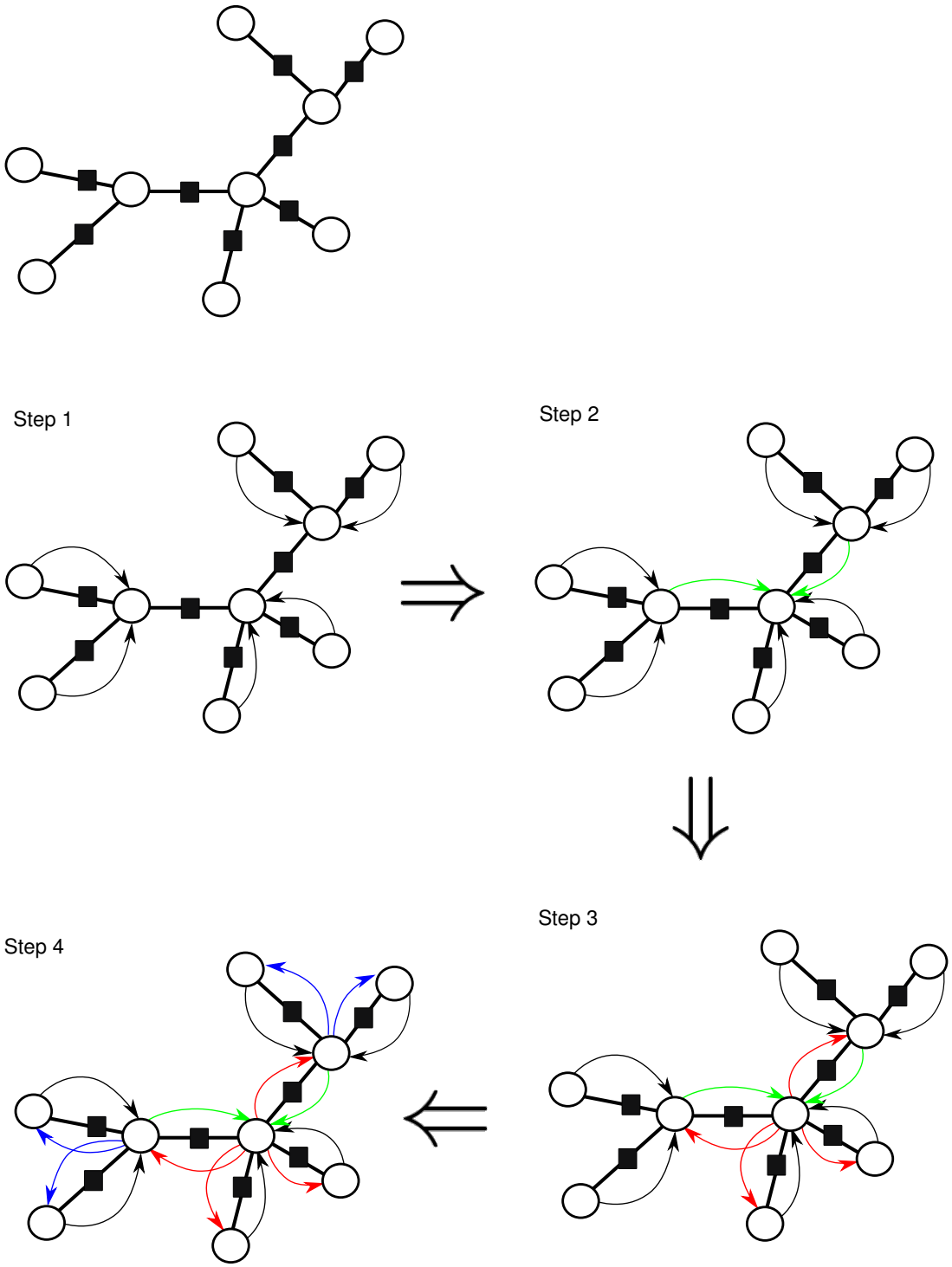


Figure 2.6 Steps involved for computing the messages of the politree represented at the top. The leaves are the first nodes for which the outgoing messages can be computed.

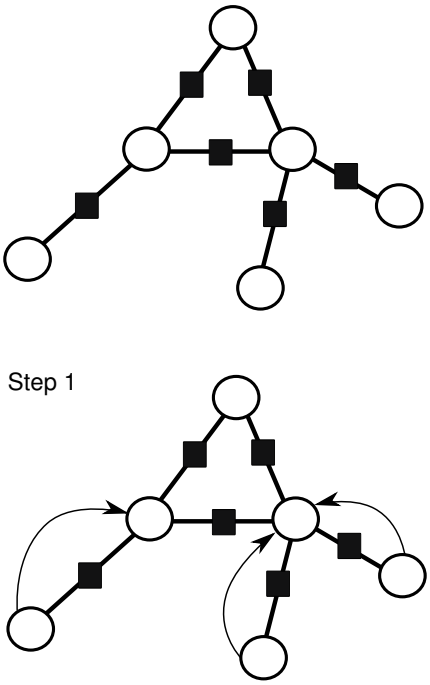


Figure 2.7 Steps involved for computing the messages on a loopy graph: after computing the messages outgoing from the leaves, a deadlock is reached since no further messages are computable.

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 8.33.2.6 on a particular derivation of class Node_factory. Maximum a posteriori estimation for sub graphs (see Section 2.5) is also supported by method MAP 8.1.2.4.

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_k^{j+1} for the subsequent iteration is obtaining by sampling from the following marginal distribution:

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

After an initial transient, the samples cumulated during the iterations can be considered as drawn from the joint distribution involving group $X_{1,\cdots,m}$.

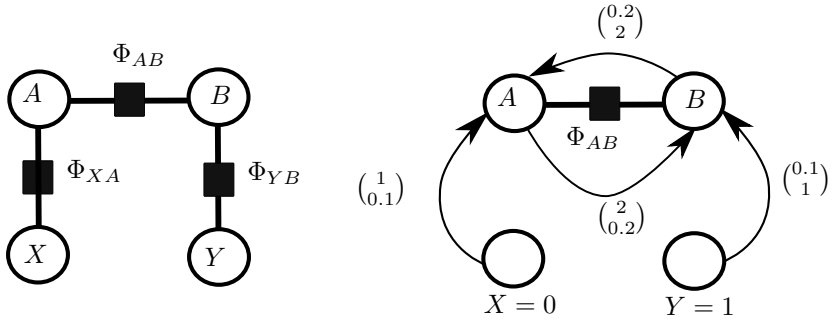


Figure 2.8 Example of graph adopted. When the evidences are retrieved, the messages computed by making use of the message passing algorithm are reported below.

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 8.33.2.5 is in charge of performing Gibbs sampling on a generic graph, while method Gibbs_Sampling 8.1.2.3 performs the same for sub graphs (see 2.5).

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:

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

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

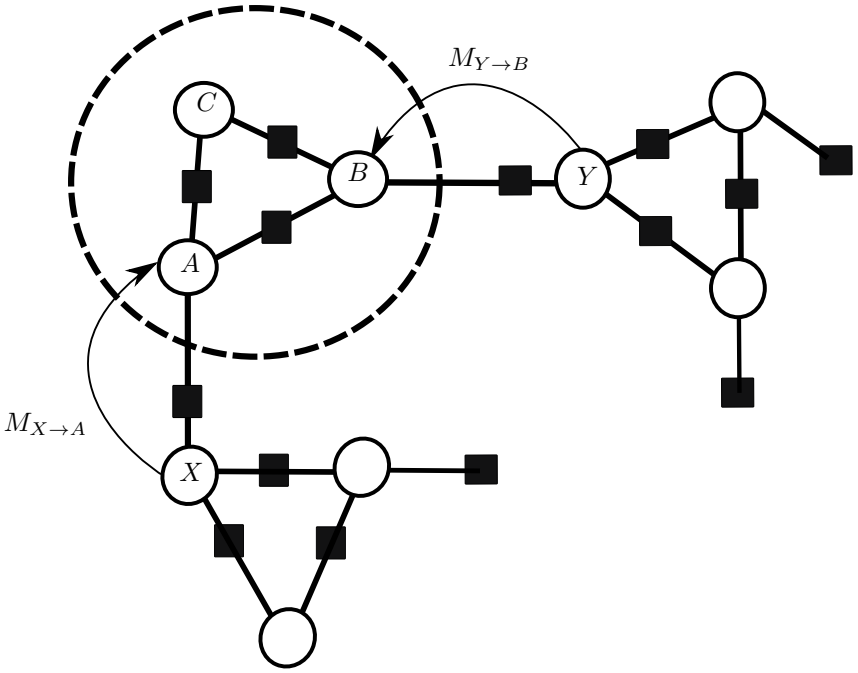
The aim of learning is to determine the optimal values for the w (equation (2.13)) of all the tunable potentials (see Section 2.1) Ψ. To this aim two cases must be distinguished:

- 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.



Sub graph involving A, B, C

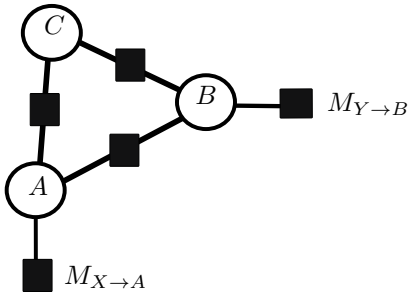


Figure 2.9 Example of graph reduction.

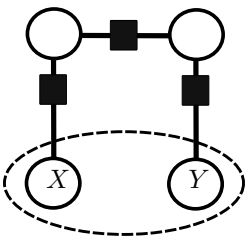


Figure 2.10 Example of a subset of variables for which the graph reduction is not possible.

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:

$$J = \alpha(T, W) - \beta(W)$$

$$\alpha = \frac{1}{N} \sum_{t_j \in T} \big(\sum_{i=1}^{D} w_i\Phi_i(t_j)\big) \tag{2.40}$$

$$\beta = log(\mathcal{Z}(W)) \tag{2.41}$$

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:

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

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)):

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

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

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

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

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

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:

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

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

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

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:

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

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.

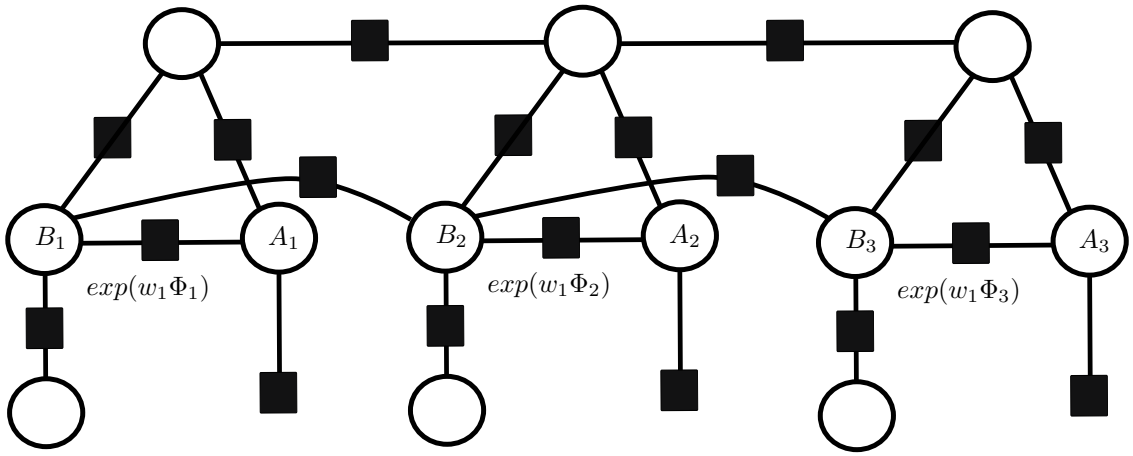


Figure 2.11 Example of modular structure: weight w_1 is simultaneously involved into potentials Φ_1, Φ_2 and Φ_3.

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 8.16.2.2, Random_Field 8.38.2.2 and Conditional_Random_Field 8.9.2.1. 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 8.37.2.3), otherwise when passing 'F' a simple anti correlating shape is assumed (see 8.37.2.3). 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.

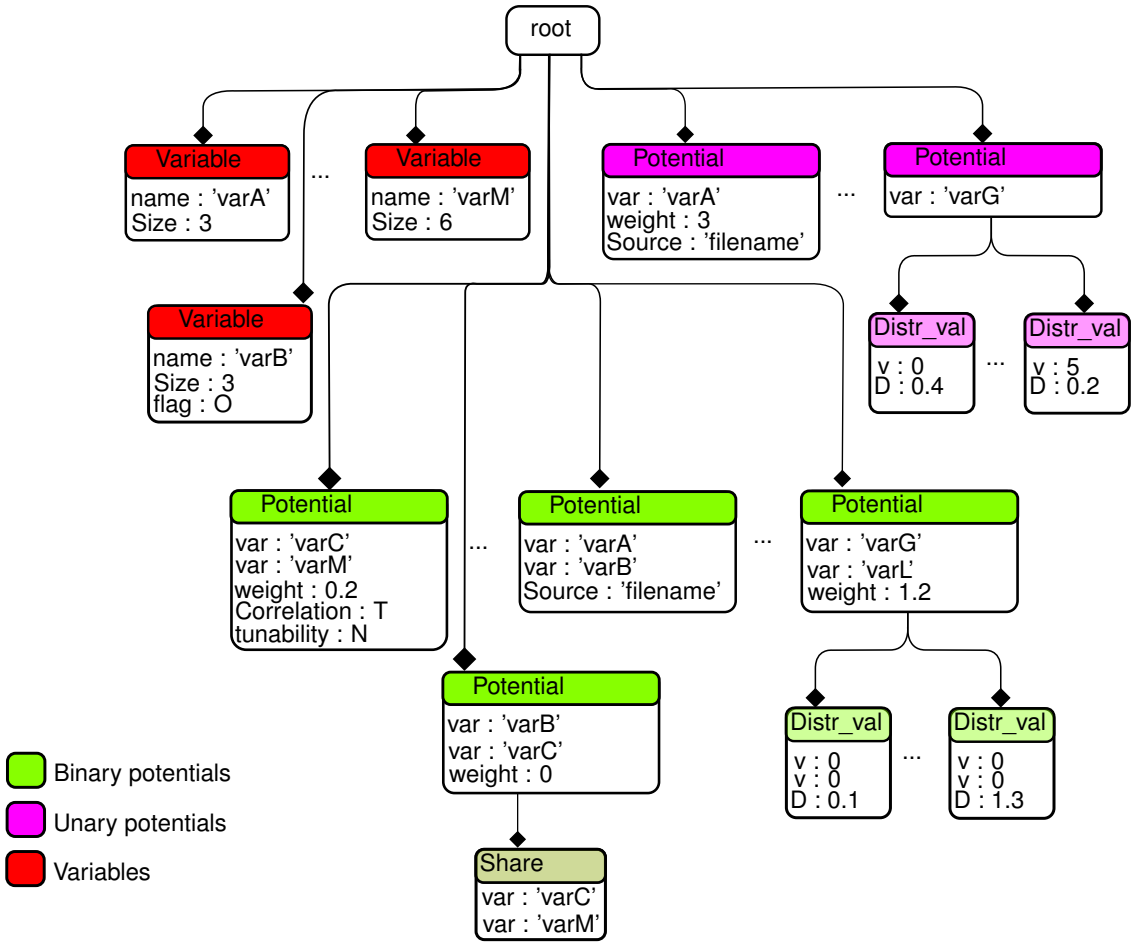


Figure 3.1 The structure of the XML describing a graphical model.

– 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.

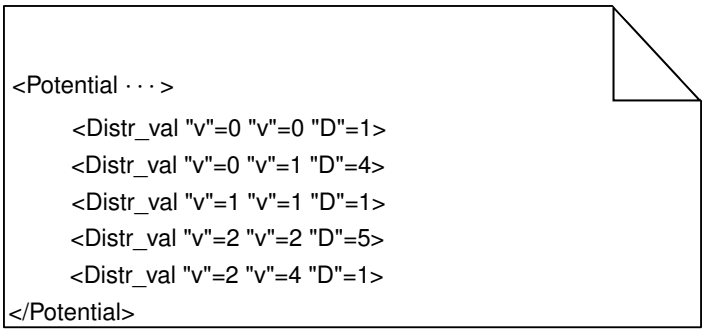


Figure 3.2 Syntax to adopt for describing the potential Φ_b of Section 2.1, using a population of Distr_val sub tags.

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}.

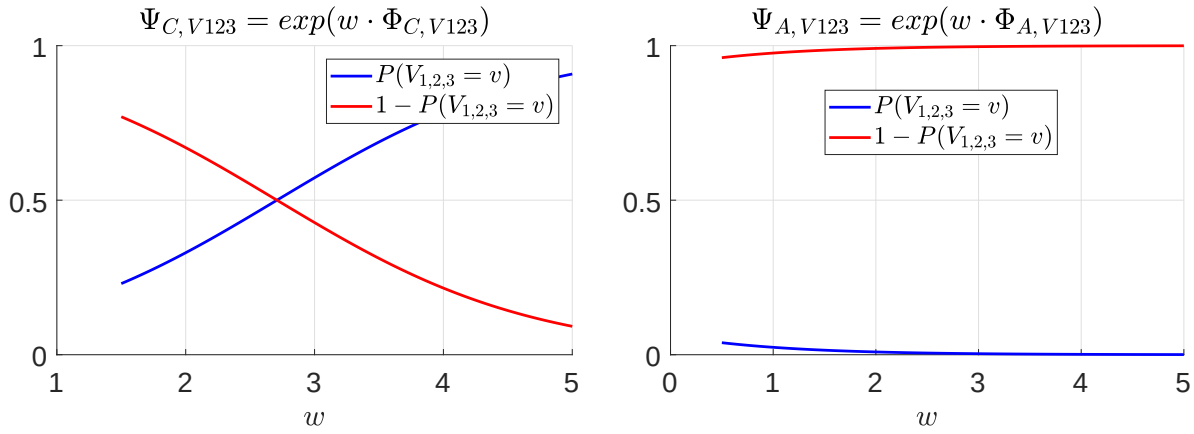


Figure 4.1 The probability $\mathbb{P}(V_1 = v, V_2 = v, V_3 = v|w)$ and its complement, when considering a ternary correlating factor, on the left, and an anti-correlating one, on the right.

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

$$\mathbb{P}(V_1 = v, V_2 = v, V_3 = v) = \sum_{i=0}^{3} \mathbb{P}(V_1 = i, V_2 = i, V_3 = i) \tag{4.3}$$

$$= \sum_{i=0}^{3} \frac{\Psi_{C\ V123}(i,i,i)}{Z} \tag{4.4}$$

$$= 4 \cdot \frac{\Psi_{C\ V123}(0,0,0)}{Z} \tag{4.5}$$

$$= \frac{4 \cdot exp(w)}{(4^3 - 4) + 4 \cdot exp(w)} \tag{4.6}$$

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:

$$Z = (4^3 - 4) \cdot exp(w) + 4 \tag{4.7}$$

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

$$\mathbb{P}(V_1 = v, V_2 = v, V_3 = v) = \sum_{i=0}^{3} \mathbb{P}(V_1 = i, V_2 = i, V_3 = i) \tag{4.8}$$

$$= \sum_{i=0}^{3} \frac{\Psi_{A\ V123}(i,i,i)}{Z} \tag{4.9}$$

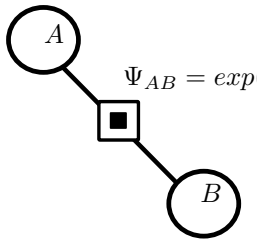
$$= 4 \cdot \frac{\Psi_{A\ V123}(0,0,0)}{Z} \tag{4.10}$$

$$= \frac{4}{(4^3 - 4) \cdot exp(w) + 4} \tag{4.11}$$

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.

4.1.3 part 03

The potential $\Phi_b(A, B)$ described by table 2.3 is created and inserted as unique factor of a graph. The computation of the probabilities reported in equations 2.9, 2.10 and 2.11 is done. Such probabilities are computed



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.

considering the joint distribution of A and B: a subgraph (Section 2.5) involving that variables is built, since Node_factory::Get_marginal_distribution only the individual marginals of A and B can be computed.

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.

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)):

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

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

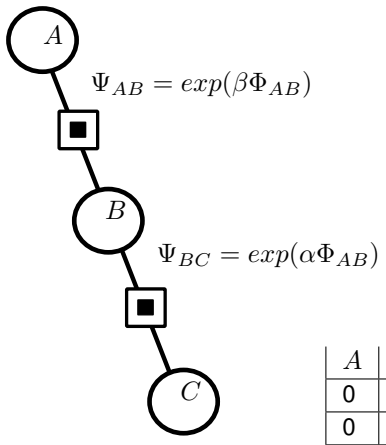
4.2.2 part 02

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 (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:

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

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

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



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.

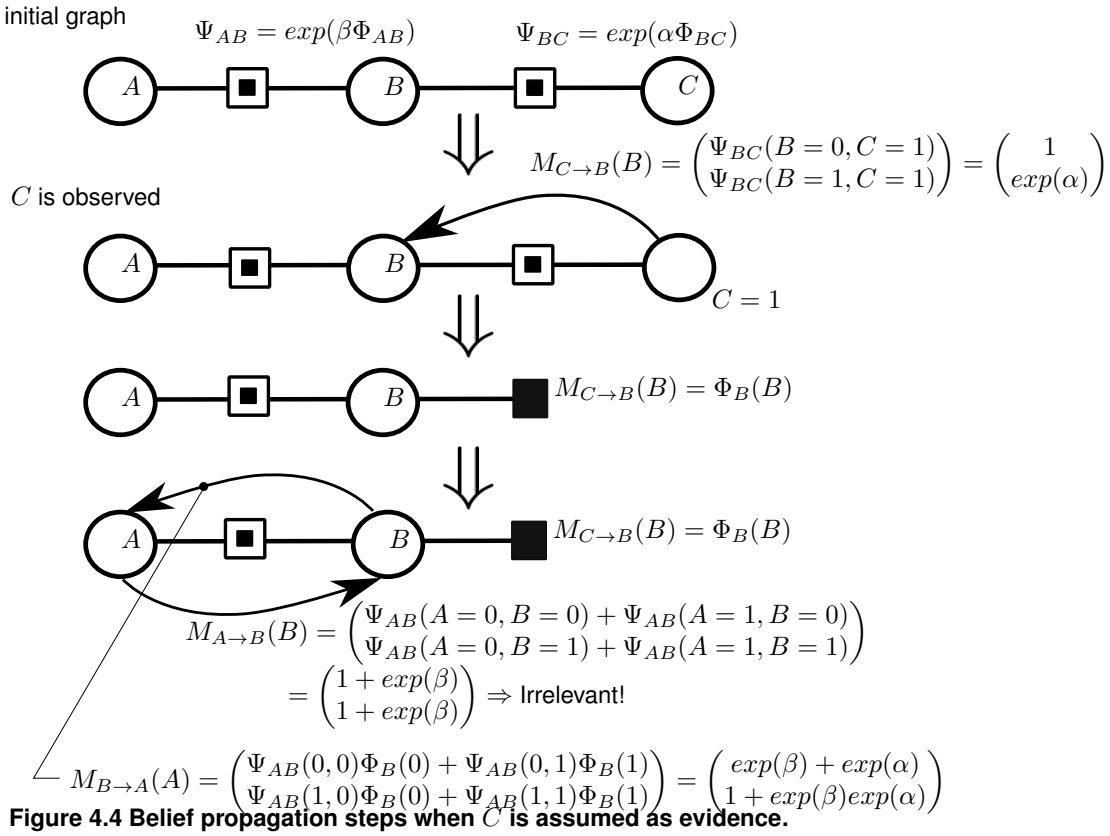


Figure 4.4 Belief propagation steps when C is assumed as evidence.

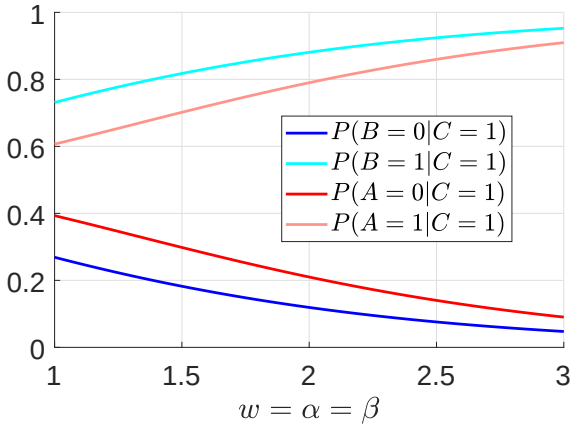


Figure 4.5 The marginals of variables B and A, when having a $C = 1$ as evidence of the graph reported in Figure 4.4.

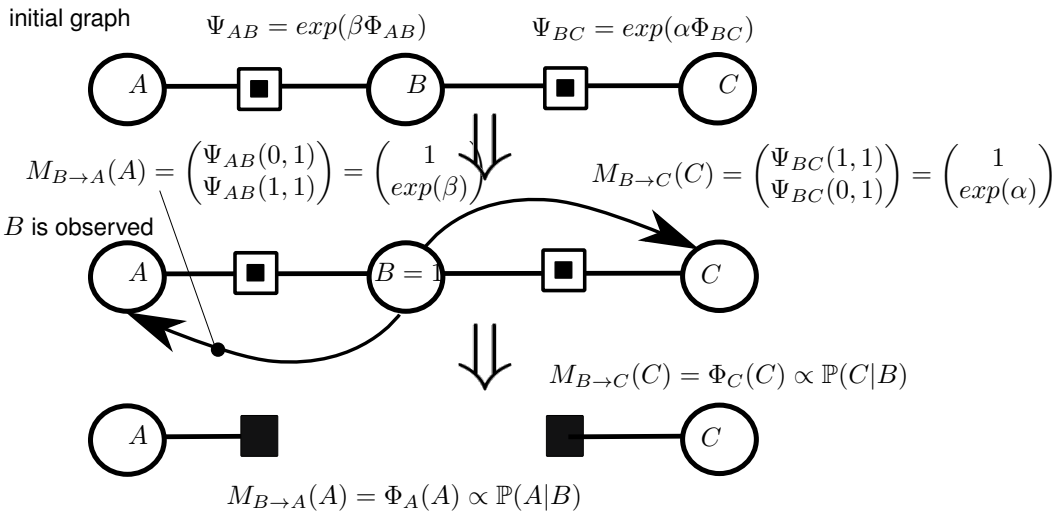
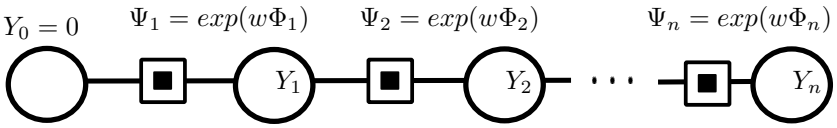


Figure 4.6 Belief propagation steps when B is assumed as evidence.



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

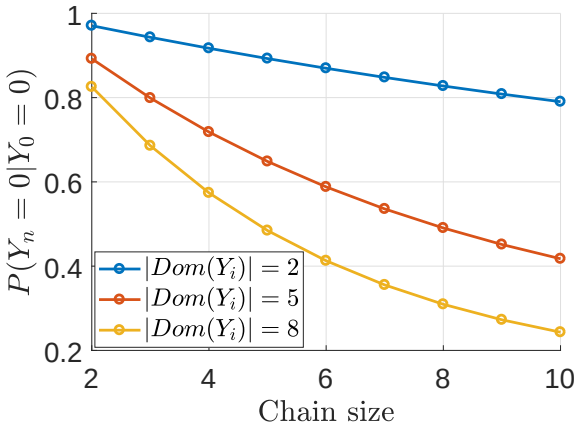


Figure 4.8 Marginal probability of Y_n when varying the chain size of the structure presented in Figure 4.7. w is assumed equal to 3.5.

Figure 4.5 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.

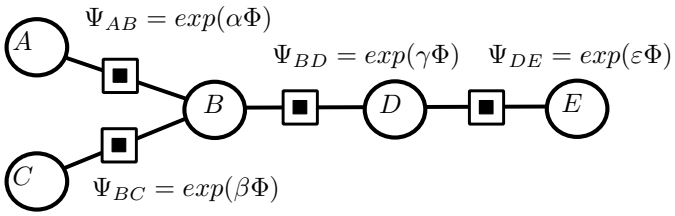


Figure 4.9 The factor graph considered by part 01.

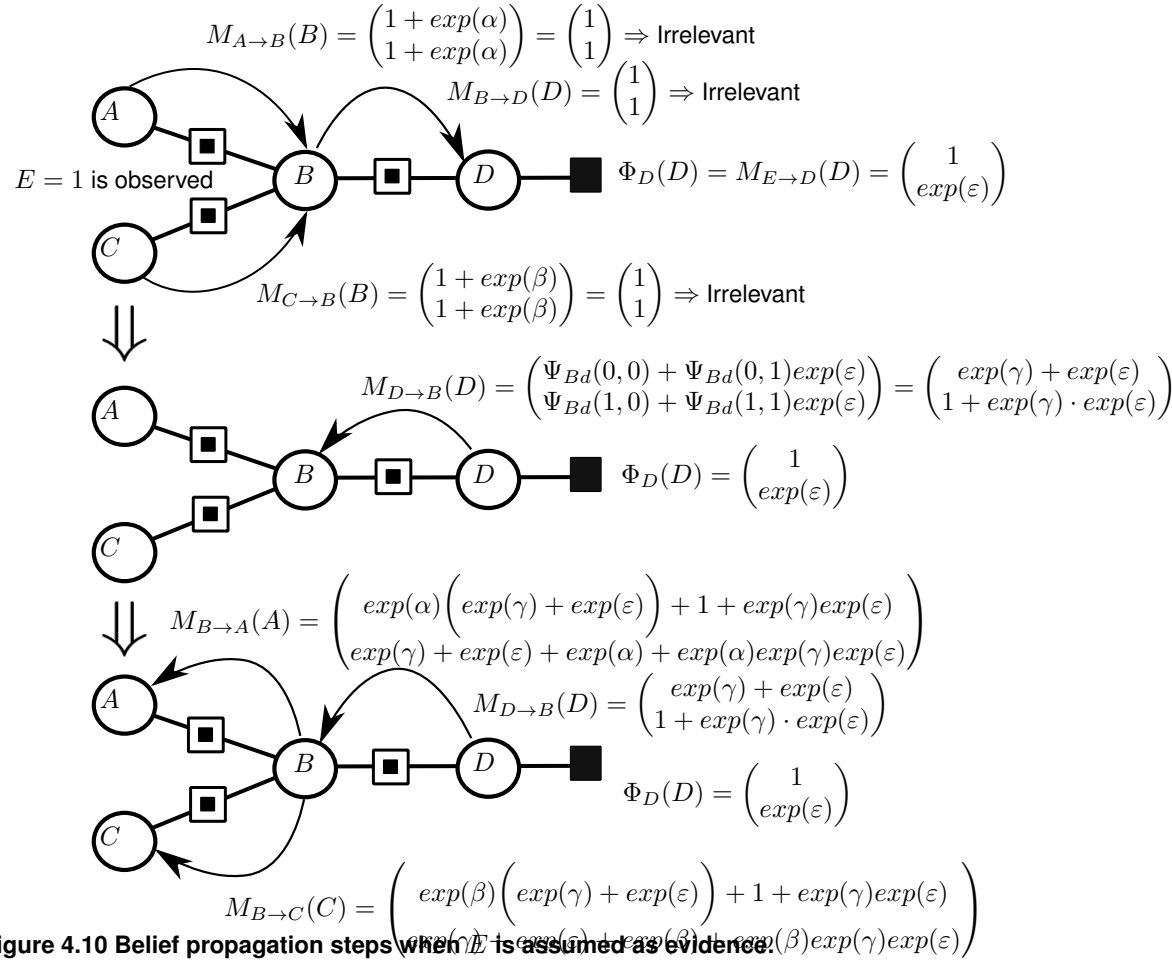


Figure 4.10 Belief propagation steps when E is assumed as evidence.

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.

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.

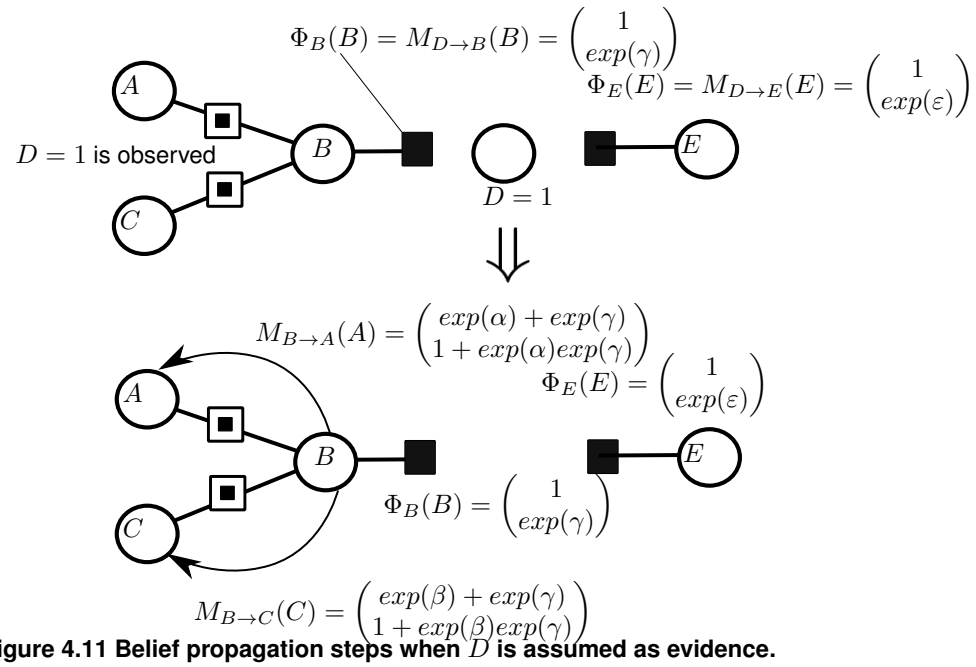


Figure 4.11 Belief propagation steps when D is assumed as evidence.

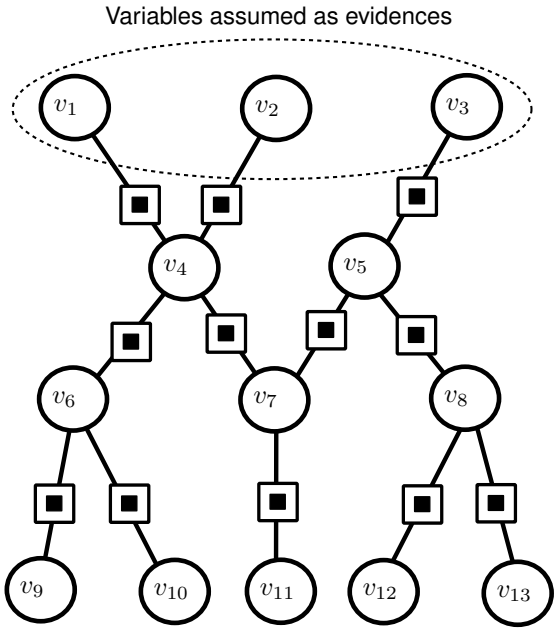


Figure 4.12 The factor graph considered by part 02.

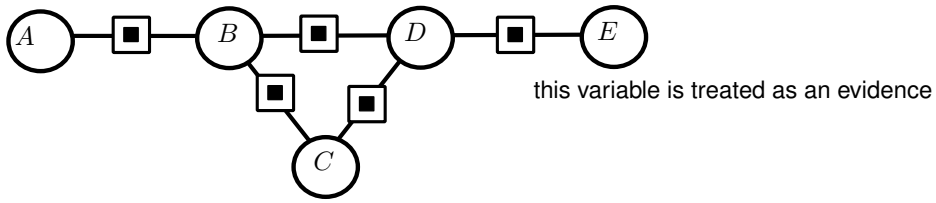


Figure 4.13 The factor graph considered by part 03.

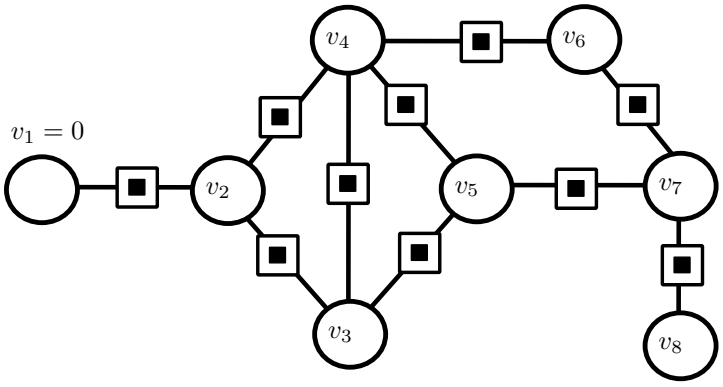


Figure 4.14 The factor graph considered by part 04.

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.

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.

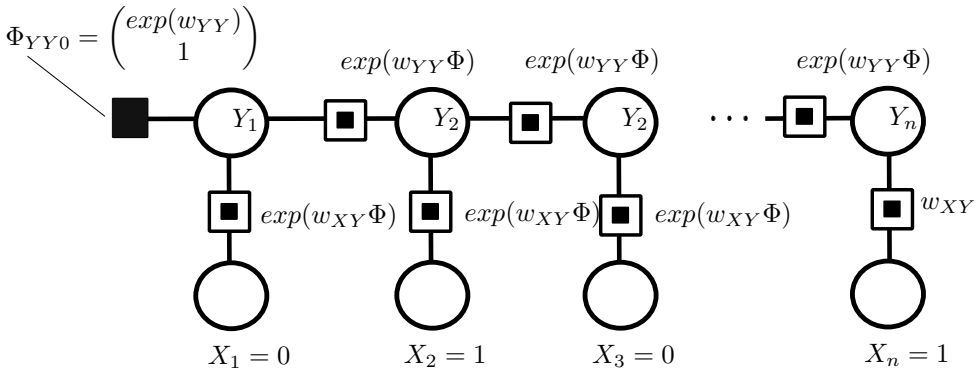


Figure 4.15 The chain structure considered by Sample 04.

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

Here the point is that when considering case a), the information about the evidences and the correlations between $Y_{1,2,...}$ and $X_{1,2,...}$ 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.

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 $w_{1,2,...}$ (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.

[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.

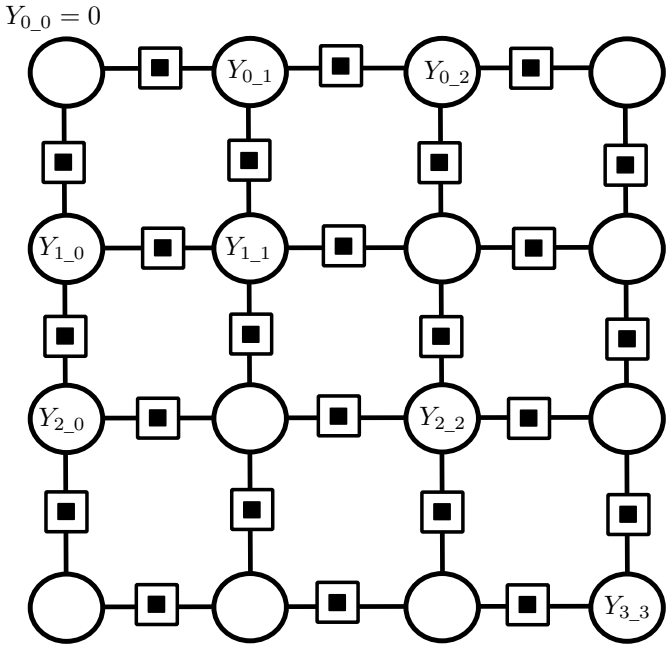


Figure 4.16 The matricial structure considered by Sample 05.

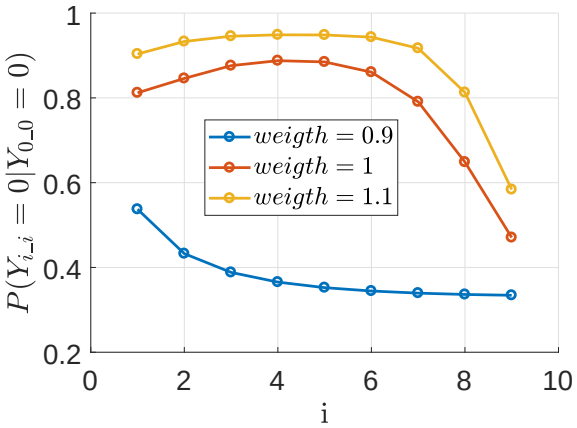


Figure 4.17 The marginals of variables Y_{i_i}, conditioned to $Y_{0\_0} = 0$ as evidence of the graph reported in Figure 4.16, when varying the weight of the correlating exponential factors involved in the structure.

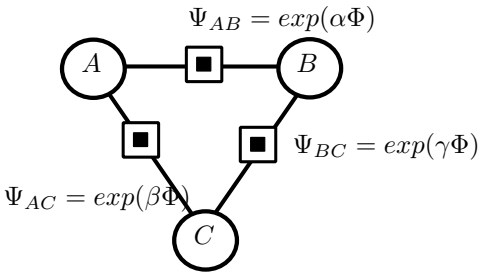


Figure 4.18 The graph considered by part 01.

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 $Z = \sum E(A,B,C) = 2\Big(exp(\alpha) + exp(\beta) + exp(\gamma) + exp(\alpha)exp(\beta)exp(\gamma)\Big)$.

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).

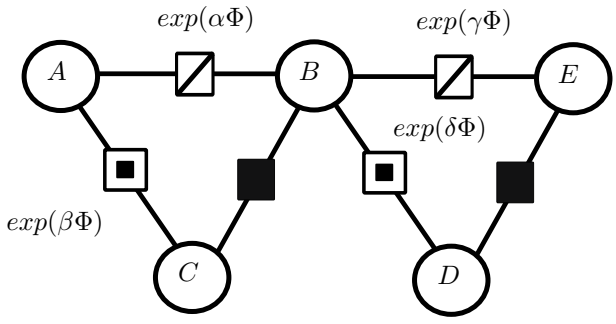


Figure 4.19 The graph considered by part 02.

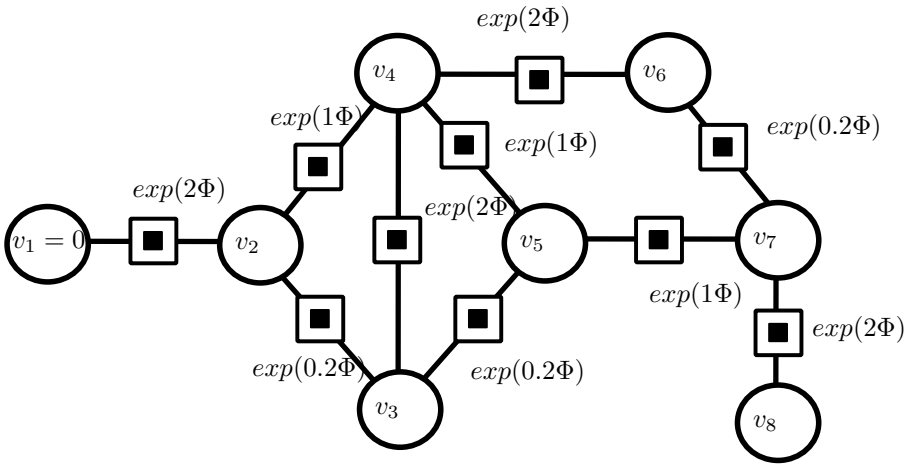


Figure 4.20 The graph considered by part 03.

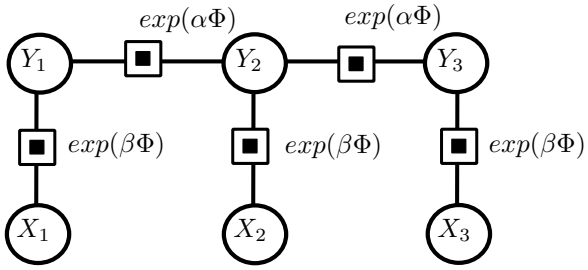


Figure 4.21 The graph considered by part 04.

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.

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 $Y_{1,2,...}$ w.r.t the evidences $X_{1,2,...}$. 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 $\mathbb{P}(Y_{1,2,...}|x)$ 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.

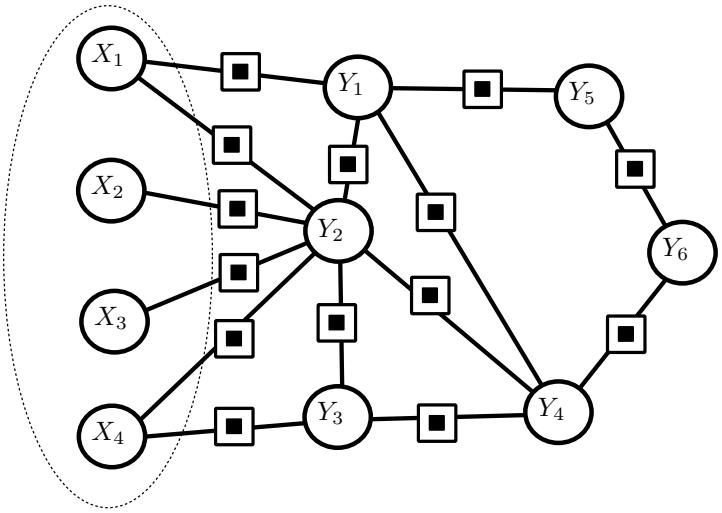


Figure 4.22 The conditional random field considered in Sample 07.

4.8 Sample 08: Sub-graphing

The aim of this example is to show how sub-graphs (see Section 2.5) can be computed using SubGraph::SubGraph. The example starts building the structure described in Figure 4.23 (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 4.23. 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.

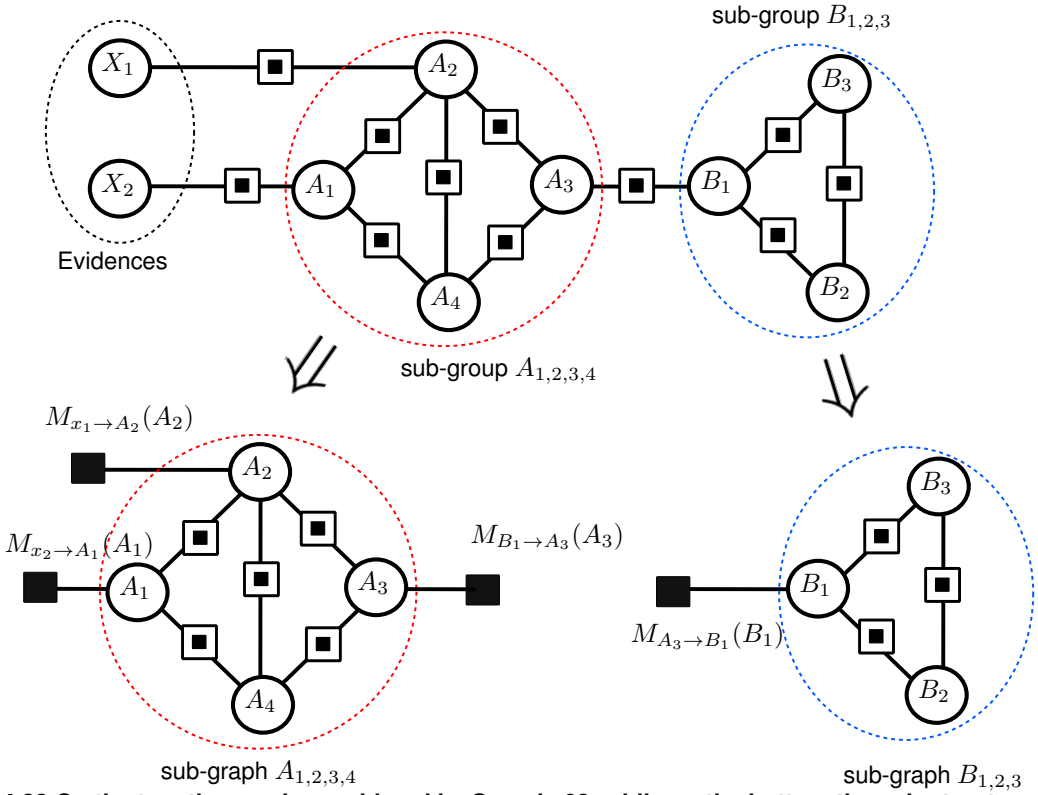


Figure 4.23 On the top, the graph considered by Sample 08, while on the bottom the sub-structures of the two groups $A_{1,2,3,4}$ ad $B_{1,2,3}$.

Chapter 5

Interactive application for factor graphs: EFG_GUI

EFG_GUI is a graphical user interface designed to manipulate medium-size factor graphs. It is essentially composed by two parts: EFG_GUI.exe and EFG_GUI.html. EFG_GUI.exe implements an Http server, listening for the user requests and satisfying them using the EFG library. When you double click EFG_GUI.exe, EFG_GUI.html is automatically launched, using your default browser. EFG_GUI.html is an html interface through which the user communicates its need to the executable file.
When you close from the browser an EFG_GUI.html window, the executable is automatically terminated: never close the application directly.
Always keep EFG_GUI.exe, EFG_GUI.html , src/, img_GUI/, image/ in the same folder.

When you run EFG_GUI.exe, it will appear a window similar to Fig. 5.1. The white region on the bottom it is a dashboard that will contain the structure you will work on, while the one on the top a list of always available commands that you can click. In the following of this Chapter, every command of the interface will be commented.
EFG_GUI exploits the html5 technology, but it does not require a connection: the front-end .html is executed locally by your browser.

5.1 Modifying commands

This set of commands alter the structure of graph you are working on. By clicking the top left button, left part of Figure 5.2, the set of modifying commands will appear, right part of Figure 5.2.

5.1.0.1 Import command

By clicking command a of the toolbar in the right part of Figure 5.2, you can import a new structure from an xml file (such files must be compliant with the format described in Chapter 3): a pop-up will appear that lets you browser the folders on your pc for finding the xml you are interested in. The current structure is deleted and the new one is imported.

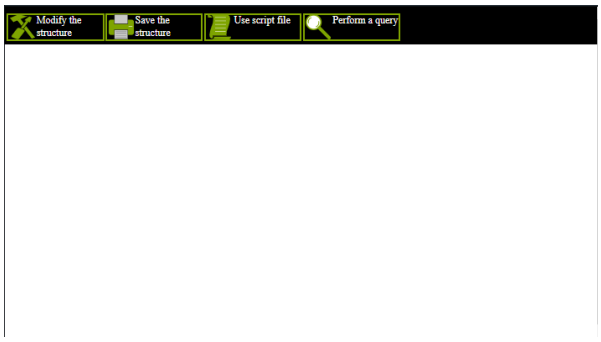


Figure 5.1 The home page of the interface.

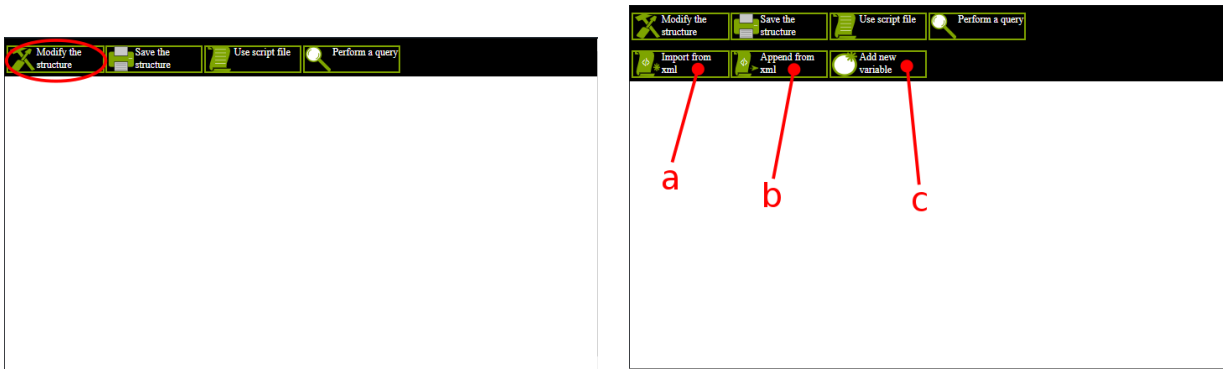


Figure 5.2 The location of the button enabling the modifying commands is indicated on the left. On the right the tool bar with the modifying commands.

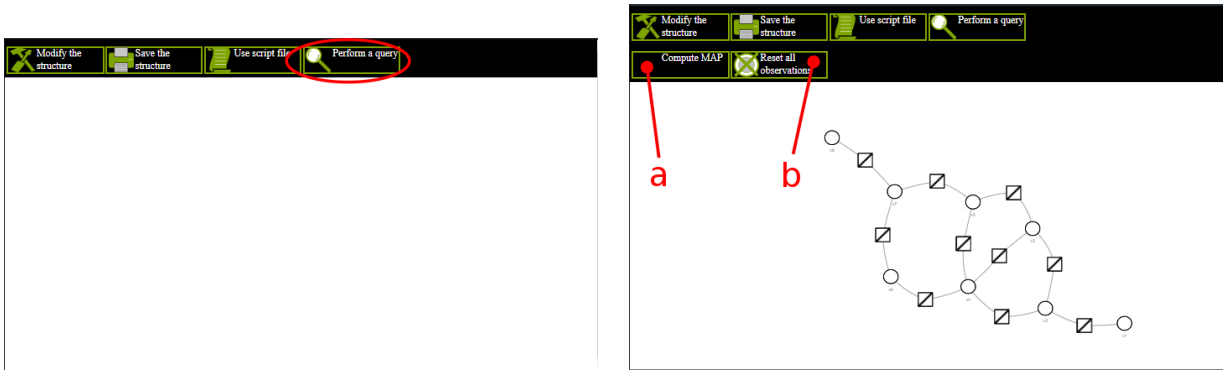


Figure 5.3 The location of the button enabling the global query commands is indicated on the left. On the right the tool bar with the global query commands.

5.1.0.2 Append command

By clicking command b of the toolbar in the right part of Figure 5.2, you can append the structure described by an xml file (such files must be compliant with the format described in Chapter 3) to the current one: a pop-up will appear that lets you browser the folders on your pc for finding the xml you are interested in. In the back end, $Node\_factory :: \_\_Absorb$ is invoked. In case the actual structure is empty, the effect of this command is the same of the Import command.

5.1.0.3 Create variable command

By clicking command c of the toolbar in the right part of Figure 5.2, you can add a new variable: a first pop-up will appear asking you for the name to give to the new variable and a second one will ask you the Dom size. The name of the new variable must be different from any other variable present at the time of calling the Create variable command (when you give an already used name, the variable is simply not created). When the variable is successfully created, it is not attached to the actual graph: it starts to exists in the dashboard as a separate node. Then, you can create some potentials that connects the new variable to some others already in the graph, see the buttons described in 5.5.1.

5.2 Global query commands

This set of commands perform some queries on the entire graph present in the dashboard. By clicking the top right button, left part of Figure 5.3, this set of commands will appear, right part of Figure 5.3.

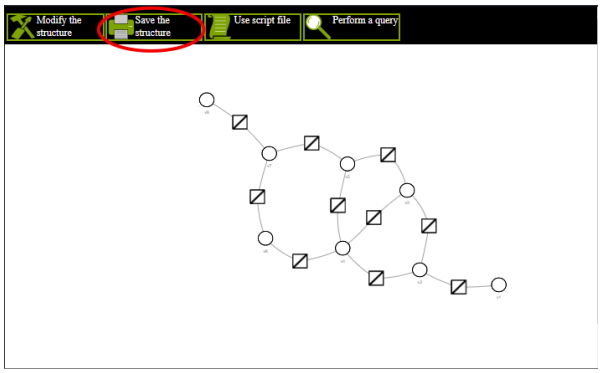


Figure 5.4 The saving command.

5.2.0.1 Compute MAP command

By clicking command a of the toolbar in the right part of Figure 5.3, you can ask for a MAP estimation (Section 2.3) of the set of variables that currently are hidden in the current graph. After back-end ends the MAP computation, the visualization of the graph in the dashboard is updated, adding to the labels of the hidden variables the value resulting from the MAP estimation.

5.2.0.2 Reset observations command

By clicking command b of the toolbar in the right part of Figure 5.3, you can delete all the evidences of the graph, i.e. set the of evidences $\mathcal{O}$ (Section 2.1) to an empty set. Consequently all the variables in the model will be treated as hidden. Clearly, any probabilistic query results previously computed (MAP estimation, Section 5.2.0.1, or marginal computations, button d described in Section 5.5.2) is deleted.

5.3 Save command

Clicking the command indicated in Figure 5.4 you can save the actual graph into an xml file (the format described in Chapter 3 is assumed). A pop-up will appear asking you the name of the file were you want to save the graph in the dashboard. You can give as prefix to the name of the file both relative and absolute paths.

5.4 Script command

Clicking the command indicated in Figure 5.5 you can execute the list of commands indicated in a script file. A pop-up will appear letting you select the scripting file. Such a script file must be a simple plain text file. Each row of that file is interpreted as a command to perform. Any accepted instruction must begin with a letter and can have a certain number of options. the options must be separated by the special character $\$$. For instance '$P\$v\$name_1\$v\$name_2\$c\$T$' is a command having P as initial symbol and two options of type v and an additional one of type c. The list of accepted commands is reported in table 5.1.

5.5 Variable specific commands

This series of commands appear in the menu on the left of the dashboard, left part of Figure 5.6, when you click on a variable in the dashboard. They handle variable specific commands. According to the kind of clicked variable, different options will be available. Anyway, for any variable clicked, the inspector represented will show at least the name of the clicked variable as well as its Dom size, see again left part of Figure 5.6. Then, additional commands may appear according to the kind of clicked variable, see the right picture of Figure 5.6.

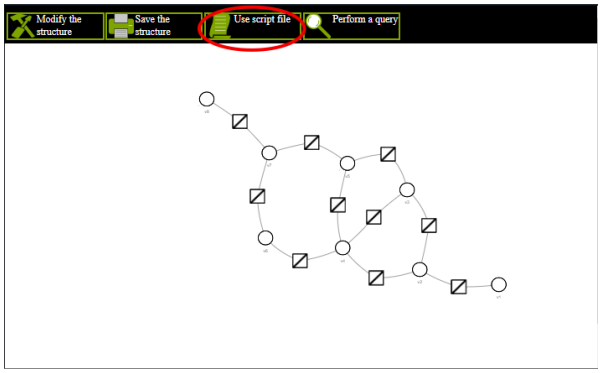


Figure 5.5 the script execution command.

Syntax	Description
X	Delete all the variables in the dashboard.
$X\$f\$S$	Import the graph described by the file at the f location.
$X\$p\$S\$f\$S$	Import the graph described by the file at the p/f location: p is simply the folder were to read the file.
$A\$f\$S$	Append to the current structure, the graph described by the file at the f location.
$A\$p\$S\$f\$S$	Append to the current structure, the graph described by the file at the p/f location: p is simply the folder were to read the file.
$R\$f\$S$	Print the current graph into an xml file at the f location.
$V\$v\$S\$s\$S\$v\$S\$s\$S$	Create new isolated variables: one for every specified pair $v - s$. In each pair v represents the name to give to the new variable, while s is its Dom size.
$V\$v\$S\$s\$S\$v\$S\$s\$S$	Create new isolated variables: one for every specified pair $v - s$. In each pair v represents the name to give to the new variable, while s is its Dom size. There is no limit to the number of variables you can create with a single command.
$O\$v\$S\$n\$N\$v\$S\$n\$N$	Add the specified variable to the set of evidences. Each pair $v - n$ represents the additional evidence to add: v specifies the name of the variable that is observed and n the value assumed. There is no limit to the number of variables you can specify.
O	Delete all the evidences, i.e. $\mathcal{O} = \emptyset$.
$I\$v\$S\$v\$S$	Computes the marginals of the specified set of variables: the v options are the name of the variables whose marginals must be computed. There is no limit to the number of variables you can specify.
M	Computes the MAP estimation of the entire hidden set $\mathcal{H}$.
$B\$f\$S$	Executes the series of command specified in the script file at the location specified by f. Indeed, you can call a script file from another.
$P\$v\$S\$s\$S$	Add a unary simple shape potential involving variable v and whose image is described by a file (with the format describe by equation (3.1)) at s.
$P\$v\$S\$s\$S\$w\$N$	Add a unary exponential potential involving variable v and whose image is described by a file (with the format describe by equation (3.1)) at s, with a weight equal to w.
$P\$v\$S\$v\$S\$s\$S$	Add a binary simple shape potential involving the variables specified by options v and whose image is described by a file (with the format describe by equation (3.1)) at s.
$P\$v\$S\$v\$S\$s\$S\$w\$N$	Add a binary exponential shape potential involving the variables specified by options v and whose image is described by a file (with the format describe by equation (3.1)) at s, with a weight equal to w.
$P\$v\$S\$v\$S\$c\$S$	Add a binary simple shape potential involving the variables specified by options v that can be: a simple correlating shape in case $c = T$ or an anti-correlating one in case $c = F$.
$P\$v\$S\$v\$S\$c\$S\$w\$N$	Add a binary exponential shape potential involving the variables specified by options v that can be: an exponential correlating shape in case $c = T$ or an anti-correlating one in case $c = F$, with a weight equal to w.

Table 5.1 List of accepted commands. S indicates a string value, while N a numerical one.

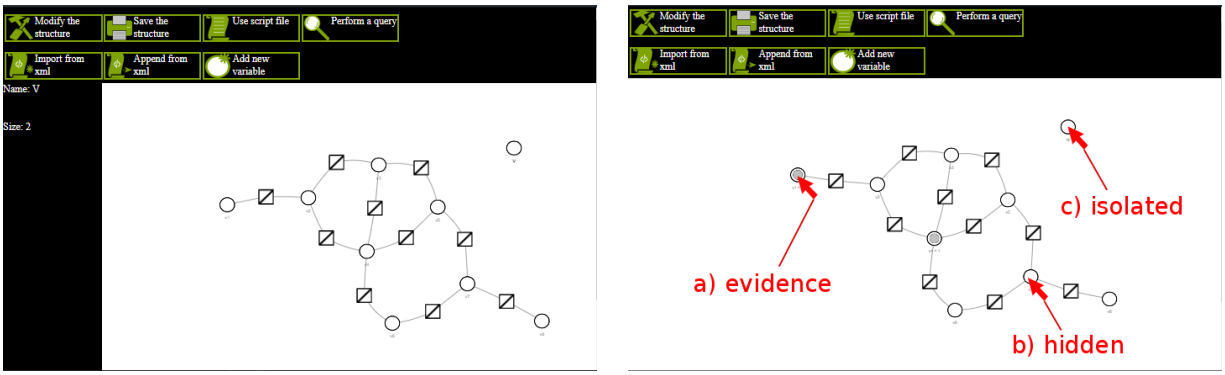


Figure 5.6 The inspector on the left part of the left picture appears when you click on a variable in the dashboard. The right part of the picture reports the three kind of variables that can be clicked: a) a variable in the evidence set, b) an hidden variable that is part of the graph and c) an isolated variable.

Figure 5.7 The inspector options for an isolated variable.

5.5.1 Isolated variable

When you click on an isolated variable (case c in the right picture of Figure 5.6) the command indicated in the left part of Figure 5.7 appears, but only in case you previously clicked an hidden variable in the graph (case b in the right picture of Figure 5.6). Indeed, in such a case the indicated button allows you to add a factor going from the previously clicked variable and the isolated one. After click that button, the options indicated in the right part of Figure 5.7 will show up. More precisely, button a and b will appear only in the case the two clicked variables have the same Dom size, while the c button will always appear. Button a allows you to add a simple correlating potential (the one created by using Potential_Shape::Potential_Shape(const std::list<Categoric_var*>& var_involved, const bool& correlated_or_not = true)), while button b an anti correlating one (the one created by using Potential_Shape::Potential_Shape(const std::list<Categoric_var*>& var_involved, const bool& correlated_or_not = false)). After clicking one of that two button, a pop-up will ask you to insert the value of the weight: if you submit a value equal to 0 a simple shape (refer to Figure 2.1) correlating/anti-correlating factor will be created, while if you pass a positive number an exponential (refer to Figure 2.1) correlating/anti-correlating factor, with the passed value for the weight, will be inserted to the graph. If you click on button c, firstly a pop-up asking for the value of the weight will appear and after that you will be asked to navigate the folders searching for a file that describes the values in the image of the potential to insert (the same format indicated in equation (3.1) is assumed). Also in this case, when passing a null weight, a simple shape factor is inserted, while an exponential factor is created in the opposite case.

5.5.2 Hidden variable in the model

When you click on an hidden variable (case b in the right picture of Figure 5.6) the commands indicated in the left part of Figure 5.8 may appear. Buttons a, b and d will always appear, while button c will appear only in the case that you have previously clicked another hidden variable. Button c allows you to add a binary factor involving the clicked variable and works exactly as the one indicated in Section 5.5.1.

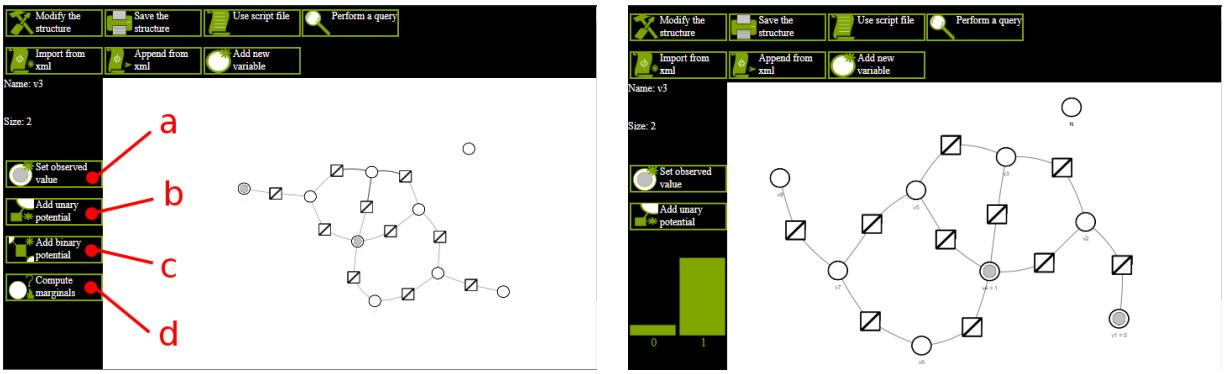


Figure 5.8 The inspector options for an hidden variable.

Button a allows you to make the clicked a variable a new observation: a pop-up will ask you to insert the observed value to assume. In case such a value is inconsistent with the Dom size of the clicked variable, the instruction is ignored.
Button b allows you to add a unary potential involving the clicked variable. A pop-up will ask you to insert the value of the weight and then you will be asked to indicate the file that contains the values in the image of the potential (refer to equation (3.1)).
Button d allows you to ask for the marginal probability of the clicked variable (conditioned to all the evidences). After clicking that button, the back-end process will compute the marginals and will communicate them to the interface. After that, if you click again that variable, an histogram showing the marginals will appear. By clicking one of the bar in that histogram a pop-up will show you the exact value of probability that bar represents, see the right part of Figure 5.8. Results about the marginals are saved for future inspection of that variable. Clearly, when the structure is modified or the evidences are changed, the information about the marginal probabilities are deleted.

5.5.3 Evidence variable

When you click on an observed variable (case a in the right picture of Figure 5.6) no further options will be available in the inspector.

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

Class Documentation

8.1 EFG::Node::Node_factory::_SubGraph Class Reference

Public Member Functions

- _SubGraph (Node_factory *Original_graph, const std::list< Categoric_var * > &sub_set_to_consider)

  Builds a reduction of the actual net, considering the actual observation values.
- Categoric_var * Find_Variable (const std::string &var_name)

  Returns a pointer to the variable in this graph with that name.
- void Get_All_variables_in_model (std::list< Categoric_var * > *result)

  Returns the set of all variable contained in the net.
- void Get_marginal_prob_combinations (std::list< float > *result, const std::list< std::list< size_t >> &combinations, const std::list< Categoric_var * > &var_order_in_combination)

  Returns the marginal probabilty of a some particular combinations of values assumed by the variables in this subgraph.
- void Get_marginal_prob_combinations (std::list< float > *result, const std::list< size_t * > &combinations, const std::list< Categoric_var * > &var_order_in_combination)

  Similar to Get_marginal_prob_combinations(std::list<float> result, const std::list < std::list<size_t>>& combinations, const std::list<(*passing the combinations as pointer arrays.*
- void MAP (std::list< size_t > *result)

  Returns the Maximum a Posteriori estimation of the hidden set in the sugraph. .
- void Gibbs_Sampling (std::list< std::list< size_t >> *result, const unsigned int &N_samples, const unsigned int &initial_sample_to_skip)

  Returns a set of samples for the variables involved in this subgraph. .
- void **Get_message_from_outside** (std::list< float > *distribution, Categoric_var *var)

8.1.1 Constructor & Destructor Documentation

8.1.1.1 _SubGraph()

```
EFG::Node::Node_factory::_SubGraph::_SubGraph (
            Node_factory * Original_graph,
            const std::list< Categoric_var * > & sub_set_to_consider )
```

Builds a reduction of the actual net, considering the actual observation values.

The subgraph is not automatically updated w.r.t. modifications of the originating net: in such cases just create a novel subgraph with the same sub_set of variables involved

8.1.2 Member Function Documentation

8.1.2.1 Find_Variable()

```
Categoric_var * EFG::Node::Node_factory::_SubGraph::Find_Variable (
            const std::string & var_name )
```

Returns a pointer to the variable in this graph with that name.

Returns NULL when the variable is not present in the graph.

Parameters

in	*var_name*	name to search

8.1.2.2 Get_marginal_prob_combinations()

```
void EFG::Node::Node_factory::_SubGraph::Get_marginal_prob_combinations (
            std::list< float > * result,
            const std::list< std::list< size_t >> & combinations,
            const std::list< Categoric_var * > & var_order_in_combination )
```

Returns the marginal probabilty of a some particular combinations of values assumed by the variables in this subgraph.

The marginal probabilities computed are conditioned to the observations set when extracting this subgraph.

Parameters

out	*result*	the computed marginal probabilities
in	*combinations*	combinations of values for which the marginals are computed: must have same size of var_order_in_combination.
in	*var_order_in_combination*	order of variables considered when assembling the combinations.

8.1.2.3 Gibbs_Sampling()

```
void EFG::Node::Node_factory::_SubGraph::Gibbs_Sampling (
            std::list< std::list< size_t >> * result,
            const unsigned int & N_samples,
            const unsigned int & initial_sample_to_skip )
```

Returns a set of samples for the variables involved in this subgraph. .

Sampling is done considering the marginal probability distribution of this cluster of variables, conditioned to the observations set at the time this subgraph was created. Samples are obtained through Gibbs sampling. Calculations are done considering the last last observations set (see Node_factory::Set_Observation_Set_var)

Parameters

in	*N_samples*	number of desired samples
in	*initial_sample_to_skip*	number of samples to skip for performing Gibbs sampling
out	*result*	returned samples: every element of the list is a combination of values for the hidden set, with the same order returned when calling _SubGraph::Get_All_variables

8.1.2.4 MAP()

```
void EFG::Node::Node_factory::_SubGraph::MAP (
            std::list< size_t > * result )
```

Returns the Maximum a Posteriori estimation of the hidden set in the sugraph. .

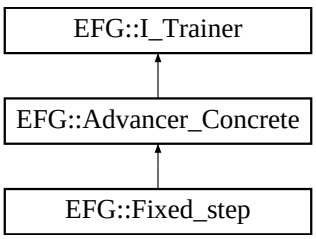
Values are ordered as returned by _SubGraph::Get_All_variables. This MAP is conditioned to the observations set at the time this subgraph was created.

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

- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Header/Node.h
- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Source/Subgraph.cpp

8.2 EFG::Advancer_Concrete Class Reference

Inheritance diagram for EFG::Advancer_Concrete:



Public Member Functions

- virtual void **Reset** ()
- void **Train** (Graph_Learnable *model_to_train, Training_set *Train_set, const unsigned int &Max_Iterations, std::list< float > *descend_story)
- virtual float **_advance** (Graph_Learnable *model_to_advance, const std::list< size_t * > &comb_in_train↩ _set, const std::list< Categoric_var * > &comb_var)=0

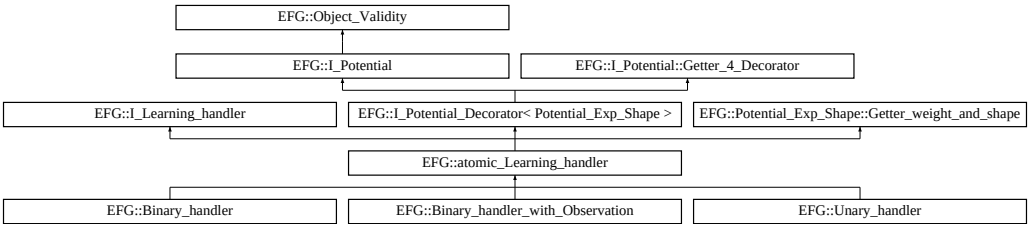
Additional Inherited Members

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

- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Source/Trainer.cpp

8.3 EFG::atomic_Learning_handler Class Reference

Inheritance diagram for EFG::atomic_Learning_handler:



Public Member Functions

- virtual void **Get_weight** (float *w)
- virtual void **Set_weight** (const float &w_new)
- virtual void **Get_grad_alfa_part** (float *alfa, const std::list< size_t * > &comb_in_train_set, const std::list< Categoric_var * > &comb_var)
- bool **is_here_Pot_to_share** (const std::list< Categoric_var * > &vars_of_pot_whose_weight_is_to_share)
- Potential_Exp_Shape * **Get_wrapped** ()

Protected Member Functions

- **atomic_Learning_handler** (Potential_Exp_Shape *pot_to_handle)
- **atomic_Learning_handler** (atomic_Learning_handler *other)

Protected Attributes

- float * **pWeight**
- std::list< I_Distribution_value * > **Extended_shape_domain**

Additional Inherited Members

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

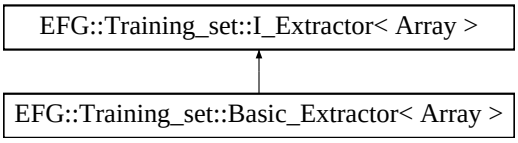
- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Header/Graphical_model.h
- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Source/Graphical_model.cpp

8.4 EFG::Training_set::Basic_Extractor< Array > Class Template Reference

Basic extractor, see Training_set(const std::list<std::string>& variable_names, std::list<Array> samples, I_↩ Extractor<Array>* extractor)

`#include <Training_set.h>`

Inheritance diagram for EFG::Training_set::Basic_Extractor< Array >:



Additional Inherited Members

8.4.1 Detailed Description

template<typename Array>
class EFG::Training_set::Basic_Extractor< Array >

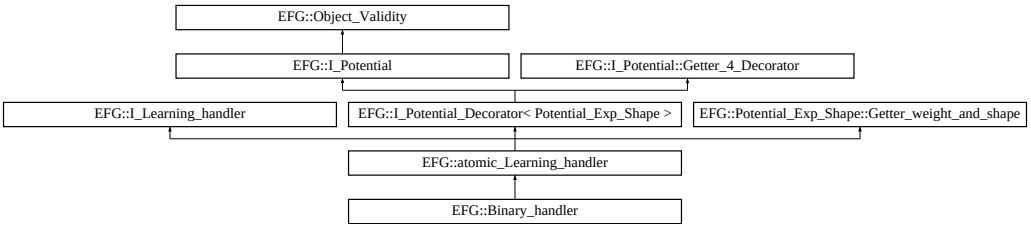
Basic extractor, see Training_set(const std::list<std::string>& variable_names, std::list<Array> samples, I_↩ Extractor<Array>* extractor)

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

- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Header/Training_set.h

8.5 EFG::Binary_handler Class Reference

Inheritance diagram for EFG::Binary_handler:



Public Member Functions

- **Binary_handler** (Node *N1, Node *N2, Potential_Exp_Shape *pot_to_handle)

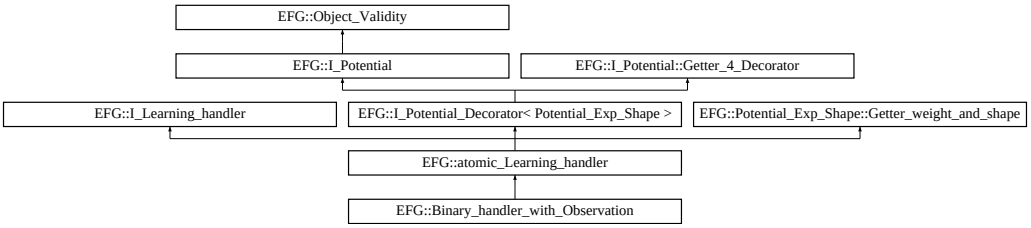
Additional Inherited Members

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

- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Source/Graphical_model.cpp

8.6 EFG::Binary_handler_with_Observation Class Reference

Inheritance diagram for EFG::Binary_handler_with_Observation:



Public Member Functions

- **Binary_handler_with_Observation** (Node ∗Hidden_var, size_t ∗observed_val, atomic_Learning_handler ∗∗handle_to_substitute)

Additional Inherited Members

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

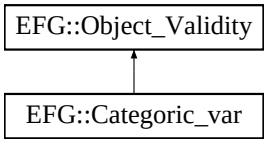
- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Source/Graphical_model.cpp

8.7 EFG::Categoric_var Class Reference

Describes a categoric variable.

`#include <Potential.h>`

Inheritance diagram for EFG::Categoric_var:



Public Member Functions

- Categoric_var (const size_t &size, const std::string &name)

  domain is assumed to be {0,1,2,3,...,size}
- const size_t & **size** () const
- const std::string & **Get_name** ()

Protected Attributes

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

8.7.1 Detailed Description

Describes a categoric variable.

, having a finite set as domain, assumed by default as {0,1,2,3,...,size}

8.7.2 Constructor & Destructor Documentation

8.7.2.1 Categoric_var()

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

domain is assumed to be {0,1,2,3,...,size}

Parameters

in	*size*	domain size of this variable
in	*name*	name to attach to this variable. It cannot be an empty string ""

8.7.3 Member Data Documentation

8.7.3.1 Name

```
std::string EFG::Categoric_var::Name  [protected]
```

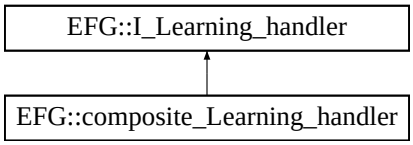
domain size

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

- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Header/Potential.h
- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Source/Potential.cpp

8.8 EFG::composite_Learning_handler Class Reference

Inheritance diagram for EFG::composite_Learning_handler:



Public Member Functions

- **composite_Learning_handler** (atomic_Learning_handler ∗initial_A, atomic_Learning_handler ∗initial_B)
- virtual void **Get_weight** (float ∗w)
- virtual void **Set_weight** (const float &w_new)
- virtual void **Get_grad_alfa_part** (float ∗alfa, const std::list< size_t ∗ > &comb_in_train_set, const std::list< Categoric_var ∗ > &comb_var)
- virtual void **Get_grad_beta_part** (float ∗beta)
- void **Append** (atomic_Learning_handler ∗to_add)
- bool **is_here_Pot_to_share** (const std::list< Categoric_var ∗ > &vars_of_pot_whose_weight_is_to_share)
- std::list< atomic_Learning_handler ∗ > ∗ **Get_Components** ()

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

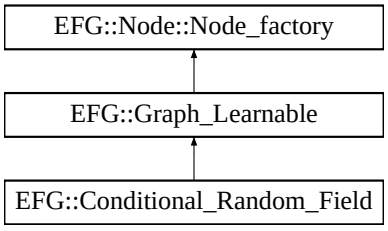
- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Header/Graphical_model.h
- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Source/Graphical_model.cpp

8.9 EFG::Conditional_Random_Field Class Reference

This class describes Conditional Random fields.

`#include <Graphical_model.h>`

Inheritance diagram for EFG::Conditional_Random_Field:



Public Member Functions

- Conditional_Random_Field (const std::string &config_xml_file, const std::string &prefix_config_xml_file="")

  The model is built considering the information contained in an xml configuration file. .
- Conditional_Random_Field (const std::list< Potential_Exp_Shape * > &potentials, const std::list< Categoric_var * > &observed_var, const bool &use_cloning_Insert=true, const std::list< bool > &tunable↩ _mask={}, const std::list< Potential_Shape * > &shapes={})

  This constructor initializes the graph with the specified potentials passed as input, setting the variables passed as the one observed.
- void Set_Evidences (const std::list< size_t > &new_observed_vals)

  see Node::Node_factory::Set_Evidences(const std::list<size_t>& new_observed_vals)

Additional Inherited Members

8.9.1 Detailed Description

This class describes Conditional Random fields.

Set_Observation_Set_var is depracated: the observed set of variables cannot be changed after construction.

8.9.2 Constructor & Destructor Documentation

8.9.2.1 Conditional_Random_Field() [1/2]

```
EFG::Conditional_Random_Field::Conditional_Random_Field (
            const std::string & config_xml_file,
            const std::string & prefix_config_xml_file = "" )
```

The model is built considering the information contained in an xml configuration file. .

See Section 3 of the documentation for the syntax to adopt.

Parameters

in	*configuration*	file
in	*prefix*	to use. The file prefix_config_xml_file/config_xml_file is searched.

8.9.2.2 Conditional_Random_Field() [2/2]

```
EFG::Conditional_Random_Field::Conditional_Random_Field (
            const std::list< Potential_Exp_Shape * > & potentials,
            const std::list< Categoric_var * > & observed_var,
            const bool & use_cloning_Insert = true,
```

```
const std::list< bool > & tunable_mask = {},
const std::list< Potential_Shape * > & shapes = {} )
```

This constructor initializes the graph with the specified potentials passed as input, setting the variables passed as the one observed.

Parameters

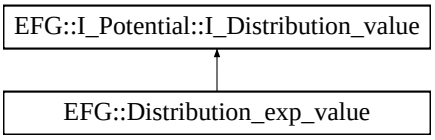
in	*potentials*	the initial set of exponential potentials to insert (can be empty)
in	*observed_var*	the set of variables to assume as observations
in	*use_cloning_Insert*	when is true, every time an Insert of a novel potential is called (this includes the passed potentials), a copy of that potential is actually inserted. Otherwise, the passed potential is inserted as is: this can be dangerous, cause that potential cna be externally modified, but the construction of a novel graph is faster.
in	*tunable_mask*	when passed as non default value, it is must have the same size of potentials. Every value in this list is true if the corresponfing potential in the potentials list is tunable, i.e. has a weight whose value can vary with learning
in	*shapes*	A list of additional non learnable potentials to insert in the model

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

- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Header/Graphical_model.h
- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Source/Graphical_model.cpp

8.10 EFG::Distribution_exp_value Struct Reference

Inheritance diagram for EFG::Distribution_exp_value:



Public Member Functions

- **Distribution_exp_value** (Distribution_value ∗to_wrap, float ∗weight)
- void **Set_val** (const float &v)
- void **Get_val** (float ∗result)
- size_t ∗ **Get_indeces** ()

Protected Attributes

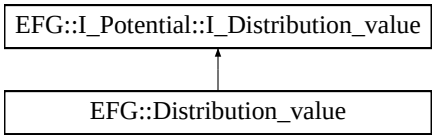
- float ∗ **w**
- Distribution_value ∗ **wrapped**

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

- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Source/Potential.cpp

8.11 EFG::Distribution_value Struct Reference

Inheritance diagram for EFG::Distribution_value:



Public Member Functions

- **Distribution_value** (size_t *ind, const float &v=0.f)
- void **Set_val** (const float &v)
- void **Get_val** (float *result)
- size_t * **Get_indeces** ()

Protected Attributes

- size_t * **indices**
- float **val**

Friends

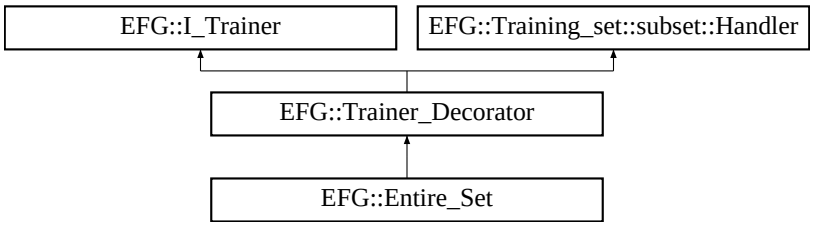
- struct **Distribution_exp_value**

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

- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Source/Potential.cpp

8.12 EFG::Entire_Set Class Reference

Inheritance diagram for EFG::Entire_Set:



Public Member Functions

- **Entire_Set** (Advancer_Concrete *to_wrap)
- void **Train** (Graph_Learnable *model_to_train, Training_set *Train_set, const unsigned int &Max_Iterations, std::list< float > *descend_story)

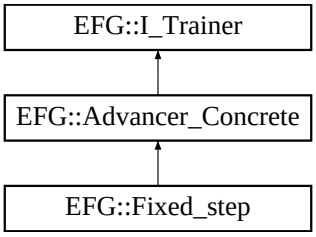
Additional Inherited Members

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

- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Source/Trainer.cpp

8.13 EFG::Fixed_step Class Reference

Inheritance diagram for EFG::Fixed_step:



Public Member Functions

- **Fixed_step** (const float &step)

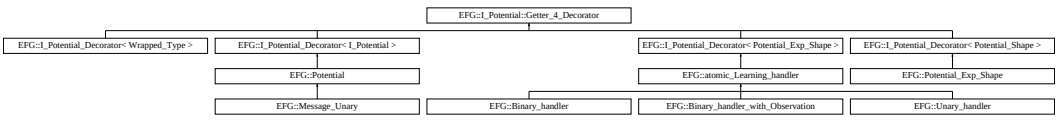
Additional Inherited Members

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

- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Source/Trainer.cpp

8.14 EFG::I_Potential::Getter_4_Decorator Struct Reference

Inheritance diagram for EFG::I_Potential::Getter_4_Decorator:



Static Protected Member Functions

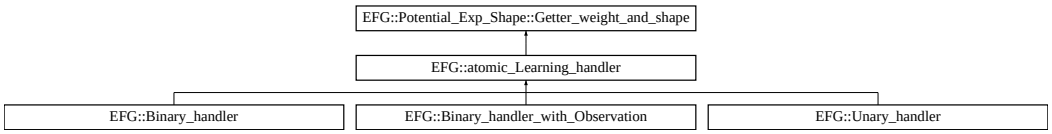
- static const std::list< Categoric_var * > * **Get_involved_var** (I_Potential *pot)
- static std::list< I_Distribution_value * > * **Get_distr** (I_Potential *pot)

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

- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Header/Potential.h

8.15 EFG::Potential_Exp_Shape::Getter_weight_and_shape Struct Reference

Inheritance diagram for EFG::Potential_Exp_Shape::Getter_weight_and_shape:



Static Protected Member Functions

- static float * **Get_weight** (Potential_Exp_Shape *pot)
- static Potential_Shape * **Get_shape** (Potential_Exp_Shape *pot)

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

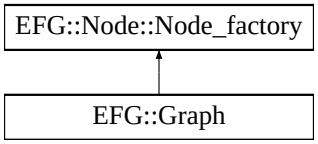
- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Header/Potential.h

8.16 EFG::Graph Class Reference

Interface for managing generic graphs.

`#include <Graphical_model.h>`

Inheritance diagram for EFG::Graph:



Public Member Functions

- Graph (const bool &use_cloning_Insert=true)

  empty constructor
- Graph (const std::string &config_xml_file, const std::string &prefix_config_xml_file="")

  The model is built considering the information contained in an xml configuration file. .
- Graph (const std::list< Potential_Shape * > &potentials, const std::list< Potential_Exp_Shape * > &potentials_exp, const bool &use_cloning_Insert=true)

  This constructor initializes the graph with the specified potentials passed as input.
- void Insert (Potential_Shape *pot)

  The model is built considering the information contained in an xml configuration file.
- void Insert (Potential_Exp_Shape *pot)

  The model is built considering the information contained in an xml configuration file.
- void Set_Evidences (const std::list< Categoric_var * > &new_observed_vars, const std::list< size_t > &new_observed_vals)

  see Node::Node_factory::Set_Evidences(const std::list<Categoric_var>& new_observed_vars, const std::list<size_t>& new_observe*
- void Set_Evidences (const std::list< size_t > &new_observed_vals)

  see Node::Node_factory::Set_Evidences(const std::list<size_t>& new_observed_vals)
- void Absorb (Node_factory *to_absorb)

  Absorbs all the variables and the potentials contained in the model passed as input.

Additional Inherited Members

8.16.1 Detailed Description

Interface for managing generic graphs.

Both Exponential and normal shapes can be included into the model. Learning is not possible: all belief propagation operations are performed assuming the mdoel as is. Every Potential_Shape or Potential_Exp_Shape is copied and that copy is inserted into the model.

8.16.2 Constructor & Destructor Documentation

8.16.2.1 Graph() [1/3]

```
EFG::Graph::Graph (
            const bool & use_cloning_Insert = true )  [inline]
```

empty constructor

Parameters

in	*use_cloning_Insert*	when is true, every time an Insert of a novel potential is called, a copy of that potential is actually inserted. Otherwise, the passed potential is inserted as is: this can be dangerous, cause that potential cna be externally modified, but the construction of a novel graph is faster.

8.16.2.2 Graph() [2/3]

```
EFG::Graph::Graph (
            const std::string & config_xml_file,
            const std::string & prefix_config_xml_file = "" )
```

The model is built considering the information contained in an xml configuration file. .

See Section 3 of the documentation for the syntax to adopt.

Parameters

in	*configuration*	file
in	*prefix*	to use. The file prefix_config_xml_file/config_xml_file is searched.

8.16.2.3 Graph() [3/3]

```
EFG::Graph::Graph (
            const std::list< Potential_Shape * > & potentials,
            const std::list< Potential_Exp_Shape * > & potentials_exp,
            const bool & use_cloning_Insert = true )
```

This constructor initializes the graph with the specified potentials passed as input.

Parameters

in	*potentials*	the initial set of potentials to insert (can be empty)
in	*potentials_exp*	the initial set of exponential potentials to insert (can be empty)
in	*use_cloning_Insert*	when is true, every time an Insert of a novel potential is called (this includes the passed potentials), a copy of that potential is actually inserted. Otherwise, the passed potential is inserted as is: this can be dangerous, cause that potential cna be externally modified, but the construction of a novel graph is faster.

8.16.3 Member Function Documentation

8.16.3.1 Absorb()

```
void EFG::Graph::Absorb (
            Node_factory * to_absorb )  [inline]
```

Absorbs all the variables and the potentials contained in the model passed as input.

Consistency checks are performed: it is possible that some inconsistent components in the model passed will be not absorbed.

8.16.3.2 Insert() [1/2]

```
void EFG::Graph::Insert (
            Potential_Shape * pot )  [inline]
```

The model is built considering the information contained in an xml configuration file.

Parameters

in	*the*	potential to insert. It can be a unary or a binary potential. In case it is binary, at least one of the variable involved must be already inserted to the model before (with a previous Insert having as input a potential which involves that variable).

8.16.3.3 Insert() [2/2]

```
void EFG::Graph::Insert (
            Potential_Exp_Shape * pot )  [inline]
```

The model is built considering the information contained in an xml configuration file.

Parameters

in	*the*	potential to insert. It can be a unary or a binary potential. In case it is binary, at least one of the variable involved must be already inserted to the model before (with a previous Insert having as input a potential which involves that variable).

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

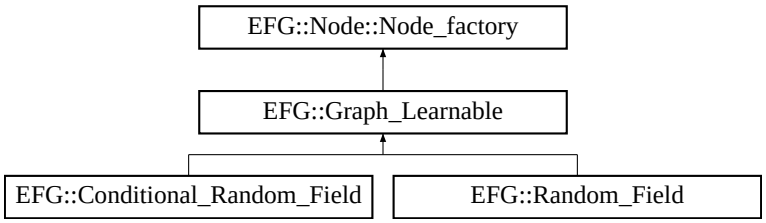
- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Header/Graphical_model.h
- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Source/Graphical_model.cpp

8.17 EFG::Graph_Learnable Class Reference

Interface for managing learnable graphs, i.e. graphs for which it is possible perform learning.

`#include <Graphical_model.h>`

Inheritance diagram for EFG::Graph_Learnable:



Classes

- struct Weights_Manager

Public Member Functions

- size_t Get_model_size ()

  Returns the model size, i.e. the number of tunable parameters of the model, i.e. the number of weigths that can vary with learning.
- void **Get_Likelihood_estimation** (float *result, const std::list< size_t * > &comb_train_set, const std::list< Categoric_var * > &comb_var_order)

Protected Member Functions

- virtual Potential_Exp_Shape * **__Insert** (Potential_Exp_Shape *pot, const bool &weight_tunability)
- **Graph_Learnable** (const bool &use_cloning_Insert)
- **Graph_Learnable** (const std::list< Potential_Exp_Shape * > &potentials_exp, const bool &use_cloning_↩ Insert, const std::list< bool > &tunable_mask, const std::list< Potential_Shape * > &shapes)
- void **Get_complete_atomic_handler_list** (std::list< atomic_Learning_handler ** > *atomic_list)
- void **Remove** (atomic_Learning_handler *to_remove)
- void **Share_weight** (I_Learning_handler *pot_involved, const std::list< Categoric_var * > &vars_of_pot_↩ whose_weight_is_to_share)
- void **Import_XML_sharing_weight_info** (XML_reader &reader)
- virtual void **__Absorb** (Node_factory *to_absorb)

Protected Attributes

- std::list< I_Learning_handler * > **Model_handlers**

Additional Inherited Members

8.17.1 Detailed Description

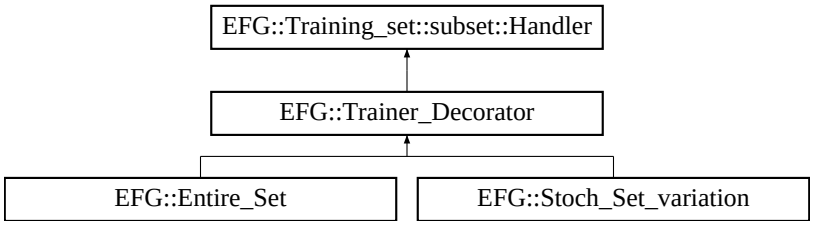
Interface for managing learnable graphs, i.e. graphs for which it is possible perform learning.

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

- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Header/Graphical_model.h
- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Source/Graphical_model.cpp

8.18 EFG::Training_set::subset::Handler Struct Reference

Inheritance diagram for EFG::Training_set::subset::Handler:



Static Protected Member Functions

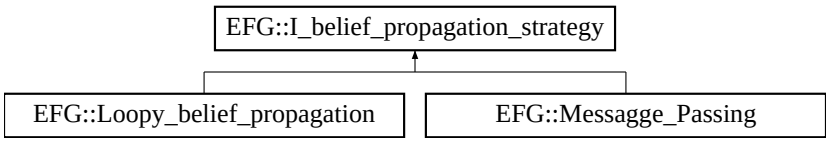
- static std::list< size_t * > * **Get_list** (subset *sub_set)
- static std::list< std::string > * **Get_names** (subset *sub_set)
- static std::list< std::string > * **Get_names** (Training_set *set)

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

- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Header/Training_set.h

8.19 EFG::I_belief_propagation_strategy Class Reference

Inheritance diagram for EFG::I_belief_propagation_strategy:



Static Public Member Functions

- static bool **Propagate** (std::list< Node * > &cluster, const bool &sum_or_MAP=true, const unsigned int &Iterations=1000)

Protected Member Functions

- void **Instantiate_message** (Node::Neighbour_connection *outgoing_mex_to_compute, const bool &sum↩_or_MAP)
- void **Update_message** (float *variation_to_previous, Node::Neighbour_connection *outgoing_mex_to_↩compute, const bool &sum_or_MAP)
- void **Gather_incoming_messages** (std::list< Potential * > *result, Node::Neighbour_connection *outgoing_mex_to_compute)
- std::list< Node::Neighbour_connection * > * **Get_Neighbourhood** (Node::Neighbour_connection *conn)
- Message_Unary ** **Get_Mex_to_This** (Node::Neighbour_connection *conn)
- Message_Unary ** **Get_Mex_to_Neigh** (Node::Neighbour_connection *conn)

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

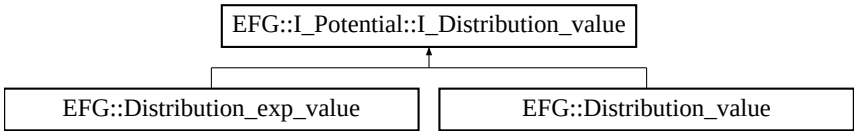
- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Header/Node.h
- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Source/Belief_propagation.cpp

8.20 EFG::I_Potential::I_Distribution_value Struct Reference

Abstract interface for describing a value in the domain of a potential.

`#include <Potential.h>`

Inheritance diagram for EFG::I_Potential::I_Distribution_value:



Public Member Functions

- virtual void **Set_val** (const float &v)=0
- virtual void **Get_val** (float *result)=0
- virtual size_t * **Get_indeces** ()=0

8.20.1 Detailed Description

Abstract interface for describing a value in the domain of a potential.

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

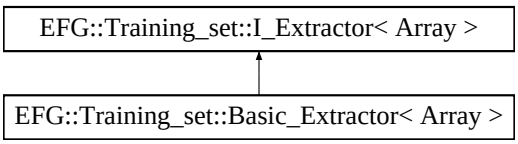
- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Header/Potential.h

8.21 EFG::Training_set::I_Extractor< Array > Class Template Reference

This class is adopted for parsing a set of samples to import as a novel training set. You have to derive yout custom extractor, implementing the two vritual method.

`#include <Training_set.h>`

Inheritance diagram for EFG::Training_set::I_Extractor< Array >:



Public Member Functions

- virtual const size_t & **get_val_in_pos** (const Array &container, const size_t &pos)=0
- virtual size_t **get_size** (const Array &container)=0

8.21.1 Detailed Description

template<typename Array>
class EFG::Training_set::I_Extractor< Array >

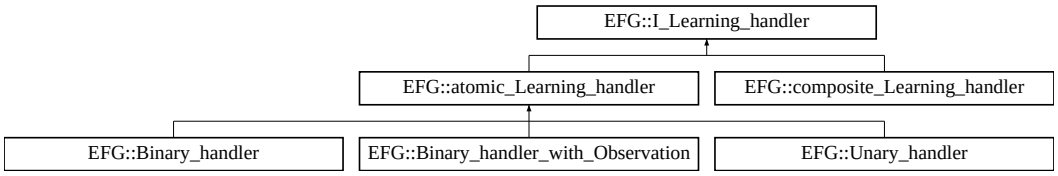
This class is adopted for parsing a set of samples to import as a novel training set. You have to derive yout custom extractor, implementing the two vritual method.

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

- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Header/Training_set.h

8.22 EFG::I_Learning_handler Class Reference

Inheritance diagram for EFG::I_Learning_handler:



Public Member Functions

- virtual void **Get_weight** (float *w)=0
- virtual void **Set_weight** (const float &w_new)=0
- virtual void **Get_grad_alfa_part** (float *alfa, const std::list< size_t * > &comb_in_train_set, const std::list< Categoric_var * > &comb_var)=0
- virtual void **Get_grad_beta_part** (float *beta)=0

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

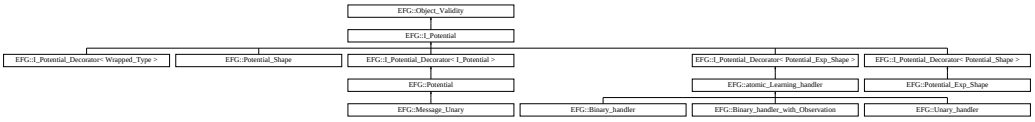
- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Header/Graphical_model.h

8.23 EFG::I_Potential Class Reference

Abstract interface for potentials handled by graphs.

`#include <Potential.h>`

Inheritance diagram for EFG::I_Potential:



Classes

- struct Getter_4_Decorator
- struct I_Distribution_value

  Abstract interface for describing a value in the domain of a potential.

Public Member Functions

- **I_Potential** (const I_Potential &to_copy)
- void Print_distribution (std::ostream &f, const bool &print_entire_domain=false)

  when print_entire_domain is true, the entire domain is printed, even though the potential has a sparse distribution
- const std::list< Categoric_var * > * Get_involved_var_safe () const

  return list of references to the variables representing the domain of this Potential
- void Find_Comb_in_distribution (std::list< float > *result, const std::list< size_t * > &comb_to_search, const std::list< Categoric_var * > &comb_to_search_var_order)
- float max_in_distribution ()

  Returns the maximum value in the distribution describing this potential.

Static Public Member Functions

- static void Get_entire_domain (std::list< std::list< size_t >> *domain, const std::list< Categoric_var * > &Vars_in_domain)

  get entire domain of a group of variables: list of possible combinations
- static void Get_entire_domain (std::list< size_t * > *domain, const std::list< Categoric_var * > &Vars_in_domain)

  Same as Get_entire_domain(std::list<std::list<size_t>> domain, const std::list<Categoric_var*>& Vars_in_domain), but adopting array internally allocated with malloc instead of list: remembre to delete combinations.*

Protected Member Functions

- virtual const std::list< Categoric_var * > * **Get_involved_var** () const =0
- virtual std::list< I_Distribution_value * > * **Get_distr** ()=0

Static Protected Member Functions

- static void **Find_Comb_in_distribution** (std::list< I_Distribution_value * > *result, const std::list< size_t * > &comb_to_search, const std::list< Categoric_var * > &comb_to_search_var_order, I_Potential *pot)
- static void **Find_Comb_in_distribution** (std::list< I_Distribution_value * > *result, size_t *partial_comb_to_search, const std::list< Categoric_var * > &partial_comb_to_search_var_order, I_Potential *pot)

Additional Inherited Members

8.23.1 Detailed Description

Abstract interface for potentials handled by graphs.

8.23.2 Member Function Documentation

8.23.2.1 Find_Comb_in_distribution()

```
void EFG::I_Potential::Find_Comb_in_distribution (
            std::list< float > * result,
            const std::list< size_t * > & comb_to_search,
            const std::list< Categoric_var * > & comb_to_search_var_order )
```

Parameters

out	*result*	the list of values matching the combinations to find sent as input
in	*comb_to_search*	domain list of combinations (i.e. values of the domain) whose values are to find
in	*comb_to_search_var_order*	order of variables used for assembling the combinations to find

8.23.2.2 Get_entire_domain() [1/2]

```
void EFG::I_Potential::Get_entire_domain (
            std::list< std::list< size_t >> * domain,
            const std::list< Categoric_var * > & Vars_in_domain )  [static]
```

get entire domain of a group of variables: list of possible combinations

Parameters

out	*domain*	the entire set of possible combinations
in	*Vars_in_domain*	variables involved whose domain has to be compute

8.23.2.3 Get_entire_domain() [2/2]

```
static void EFG::I_Potential::Get_entire_domain (
            std::list< size_t * > * domain,
            const std::list< Categoric_var * > & Vars_in_domain )  [static]
```

Same as Get_entire_domain(std::list<std::list<size_t>>* domain, const std::list<Categoric_var*>& Vars_in_domain), but adopting array internally allocated with malloc instead of list: remembre to delete combinations.

Parameters

out	*domain*	the entire set of possible combinations
in	*Vars_in_domain*	variables involved whose domain has to be compute

8.23.2.4 Print_distribution()

```
void EFG::I_Potential::Print_distribution (
            std::ostream & f,
            const bool & print_entire_domain = false )
```

when print_entire_domain is true, the entire domain is printed, even though the potential has a sparse distribution

Parameters

in	*f*	out stream to target
in	*print_entire_domain*	

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

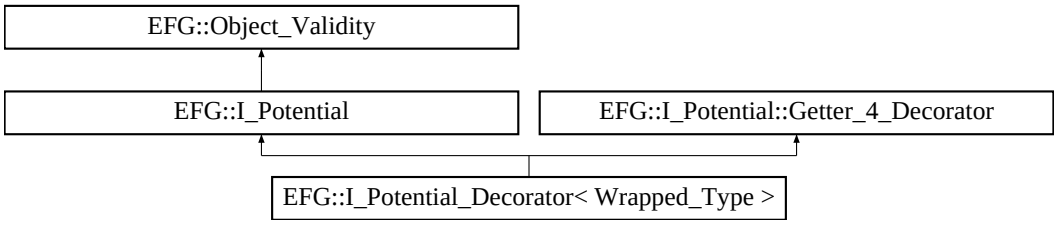
- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Header/Potential.h
- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Source/Potential.cpp

8.24 EFG::I_Potential_Decorator< Wrapped_Type > Class Template Reference

Abstract decorator of a Potential, wrapping an Abstract potential.

`#include <Potential.h>`

Inheritance diagram for EFG::I_Potential_Decorator< Wrapped_Type >:



Protected Member Functions

- **I_Potential_Decorator** (Wrapped_Type *to_wrap)
- virtual const std::list< Categoric_var * > * **Get_involved_var** () const
- virtual std::list< I_Distribution_value * > * **Get_distr** ()

Protected Attributes

- bool **Destroy_wrapped**
- Wrapped_Type * pwrapped

Additional Inherited Members

8.24.1 Detailed Description

template<typename Wrapped_Type>
class EFG::I_Potential_Decorator< Wrapped_Type >

Abstract decorator of a Potential, wrapping an Abstract potential.

8.24.2 Member Data Documentation

8.24.2.1 pwrapped

```
template<typename Wrapped_Type>
Wrapped_Type* EFG::I_Potential_Decorator< Wrapped_Type >::pwrapped  [protected]
```

when false, the wrapped abstract potential is wrapped also in another decorator, whihc is in charge of deleting the wrapped potential

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

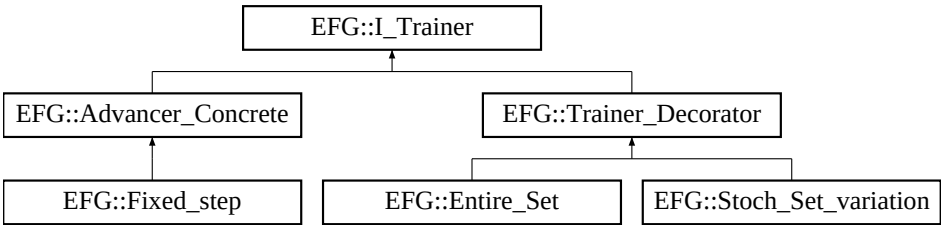
- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Header/Potential.h

8.25 EFG::I_Trainer Class Reference

This class is used by a Graph_Learnable, to perform training with an instance of a Training_set.

```
#include <Trainer.h>
```

Inheritance diagram for EFG::I_Trainer:



Public Member Functions

- virtual void **Train** (Graph_Learnable *model_to_train, Training_set *Train_set, const unsigned int &Max_↩ Iterations=100, std::list< float > *descend_story=NULL)=0

Static Public Member Functions

- static I_Trainer * Get_fixed_step (const float &step_size=0.1f, const float &stoch_grad_percentage=1.f)

  Creates a fixed step gradient descend solver.

Protected Member Functions

- virtual void **Clean_Up** ()
- void **Get_w_grad** (Graph_Learnable *model, std::list< float > *grad_w, const std::list< size_t * > &comb↩ _in_train_set, const std::list< Categoric_var * > &comb_var)
- void **Set_w** (const std::list< float > &w, Graph_Learnable *model)

Static Protected Member Functions

- static void **Clean_Up** (I_Trainer *to_Clean)

8.25.1 Detailed Description

This class is used by a Graph_Learnable, to perform training with an instance of a Training_set.

Instantiate a particular class of trainer to use by calling Get_fixed_step or Get_BFGS. That methods allocate in the heap a trainer to use later, for multiple tranining sessions. Remember to delete the instantiated trainer.

8.25.2 Member Function Documentation

8.25.2.1 Get_fixed_step()

```
I_Trainer * EFG::I_Trainer::Get_fixed_step (
            const float & step_size = 0.1f,
            const float & stoch_grad_percentage = 1.f )  [static]
```

Creates a fixed step gradient descend solver.

Parameters

in	*step_size*	learinig degree
in	*stoch_grad_percentage*	percentage of the training set to use every time for evaluating the gradient

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

- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Header/Trainer.h
- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Source/Trainer.cpp

8.26 EFG::info_neighbourhood::info_neigh Struct Reference

Public Attributes

- Potential * **shared_potential**
- Categoric_var * **Var**
- size_t **Var_pos**

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

- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Source/Node.cpp

8.27 EFG::info_neighbourhood Struct Reference

Classes

- struct info_neigh

Public Attributes

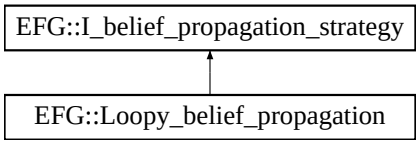
- size_t **Involved_var_pos**
- list< info_neigh > **Info**
- list< Potential * > **Unary_potentials**

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

- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Source/Node.cpp

8.28 EFG::Loopy_belief_propagation Class Reference

Inheritance diagram for EFG::Loopy_belief_propagation:



Public Member Functions

- **Loopy_belief_propagation** (const int &max_iter)
- bool **_propagate** (std::list< Node * > &cluster, const bool &sum_or_MAP)

Protected Attributes

- unsigned int **Iter**

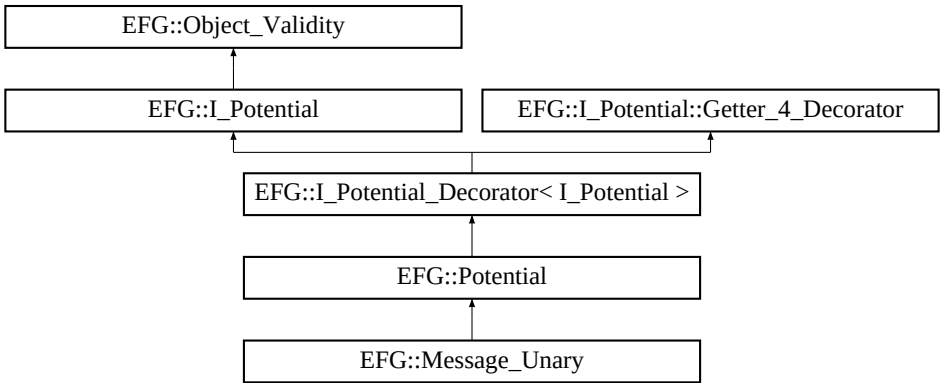
Additional Inherited Members

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

- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Header/Belief_propagation.h
- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Source/Belief_propagation.cpp

8.29 EFG::Message_Unary Class Reference

Inheritance diagram for EFG::Message_Unary:



Public Member Functions

- **Message_Unary** (Categoric_var *var_involved)
- **Message_Unary** (Potential *binary_to_merge, const std::list< Potential * > &potential_to_merge, const bool &Sum_or_MAP=true)
- **Message_Unary** (Potential *binary_to_merge, Categoric_var *var_to_marginalize, const bool &Sum_or_↩ MAP=true)
- void **Update** (float *diff_to_previous, Potential *binary_to_merge, const std::list< Potential * > &potential↩ _to_merge, const bool &Sum_or_MAP=true)
- void **Update** (float *diff_to_previous, Potential *binary_to_merge, Categoric_var *var_to_marginalize, const bool &Sum_or_MAP=true)

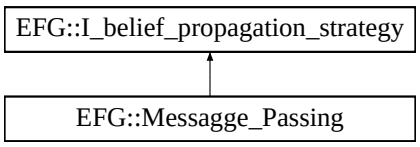
Additional Inherited Members

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

- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Header/Potential.h
- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Source/Potential.cpp

8.30 EFG::Messagge_Passing Class Reference

Inheritance diagram for EFG::Messagge_Passing:



Public Member Functions

- bool **_propagate** (std::list< Node * > &cluster, const bool &sum_or_MAP)

Additional Inherited Members

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

- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Header/Belief_propagation.h
- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Source/Belief_propagation.cpp

8.31 EFG::Node::Neighbour_connection Struct Reference

Friends

- class **Node**
- class **I_belief_propagation_strategy**

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

- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Header/Node.h
- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Source/Node.cpp

8.32 EFG::Node Class Reference

Classes

- struct Neighbour_connection
- class Node_factory

  Interface for describing a net: set of nodes representing random variables.

Public Member Functions

- Categoric_var * **Get_var** ()
- void **Gather_all_Unaries** (std::list< Potential * > *result)
- void **Append_temporary_permanent_Unaries** (std::list< Potential * > *result)
- void **Append_permanent_Unaries** (std::list< Potential * > *result)
- const std::list< Neighbour_connection * > * **Get_Active_connections** ()
- void **Compute_neighbour_set** (std::list< Node * > *Neigh_set)
- void **Compute_neighbour_set** (std::list< Node * > *Neigh_set, std::list< Potential * > *binary_involved)
- void **Compute_neighbourhood_messages** (std::list< Potential * > *messages, Node *node_involved_↩ in_connection)

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

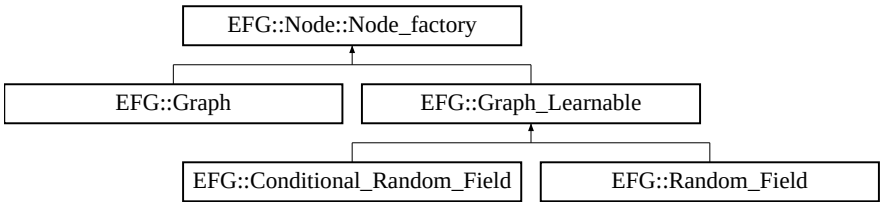
- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Header/Node.h
- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Source/Node.cpp

8.33 EFG::Node::Node_factory Class Reference

Interface for describing a net: set of nodes representing random variables.

`#include <Node.h>`

Inheritance diagram for EFG::Node::Node_factory:



Classes

- class _SubGraph

Public Member Functions

- Categoric_var * Find_Variable (const std::string &var_name)

  Returns a pointer to the variable in this graph with that name.
- void Get_Actual_Hidden_Set (std::list< Categoric_var * > *result)

  Returns the current set of hidden variables.
- void Get_Actual_Observation_Set_Var (std::list< Categoric_var * > *result)

  Returns the current set of observed variables.
- void Get_Actual_Observation_Set_Val (std::list< size_t > *result)

  Returns the current observations.
- void Get_All_variables_in_model (std::list< Categoric_var * > *result)

  Returns the set of all variable contained in the net.
- void Get_marginal_distribution (std::list< float > *result, Categoric_var *var)

  Returns the marginal probabilty of the variable passed P(var|model, observations),.
- void MAP_on_Hidden_set (std::list< size_t > *result)

  Returns the Maximum a Posteriori estimation of the hidden set. .
- void Gibbs_Sampling_on_Hidden_set (std::list< std::list< size_t >> *result, const unsigned int &N_↩ samples, const unsigned int &initial_sample_to_skip)

  Returns a set of samples of the conditional distribution P(hidden variables | model, observed variables). .
- const unsigned int & Get_Iteration_4_belief_propagation ()

  Returns the current value adopted when performing a loopy belief propagation.
- void Set_Iteration_4_belief_propagation (const unsigned int &iter_to_use)

  Returns the value to adopt when performing a loopy belief propagation.
- void Eval_Log_Energy_function (float *result, size_t *combination, const std::list< Categoric_var * > &var↩ _order_in_combination)

  Returns the logartihmic value of the energy function.
- void Eval_Log_Energy_function (float *result, const std::list< size_t > &combination, const std::list< Categoric_var * > &var_order_in_combination)

  Same as Eval_Log_Energy_function(float result, size_t* combination, const std::list<Categoric_var*>& var_order_in_combination), passing a list instead of an array size_t*, a list<size_t> for describing the combination for which you want to evaluate the energy.*
- void Eval_Log_Energy_function (std::list< float > *result, const std::list< size_t * > &combinations, const std::list< Categoric_var * > &var_order_in_combination)

  Same as Eval_Log_Energy_function(float result, size_t* combination, const std::list<Categoric_var*>& var_order_in_combination), passing a list of combinations: don't iterate yourself many times using Eval_Log_Energy_function(float* result, size_t* combination, const but call this function.*
- void Eval_Log_Energy_function_normalized (float *result, size_t *combination, const std::list< Categoric_var * > &var_order_in_combination)

  Similar as Eval_Log_Energy_function(float result, size_t* combination, const std::list<Categoric_var*>& var_order_in_combination), but computing the Energy function normalized: E_norm = E(Y_1,2,....,n) / max possible { E }. E_norm is in [0,1]. The logarthmic value of E_norm is actually returned.*
- void Eval_Log_Energy_function_normalized (float *result, const std::list< size_t > &combination, const std::list< Categoric_var * > &var_order_in_combination)

  Similar as Eval_Log_Energy_function(float result, const std::list<size_t>& combination, const std::list<Categoric_var*>& var_order_i but computing the Energy function normalized.*
- void Eval_Log_Energy_function_normalized (std::list< float > *result, const std::list< size_t * > &combinations, const std::list< Categoric_var * > &var_order_in_combination)

  Similar as Eval_Log_Energy_function(std::list<float> result, const std::list<size_t*>& combinations, const std::list<Categoric_var*>& but computing the Energy function normalized.*
- void Get_structure (std::list< const Potential_Shape * > *shapes, std::list< std::list< const Potential_Exp_Shape * >> *learnable_exp, std::list< const Potential_Exp_Shape * > *constant_exp)

  Returns the list of potentials constituting the net.
- size_t Get_structure_size ()

Returns the number of potentials constituting the graph, no matter of their type (simple shape, exponential shape fixed or exponential shape tunable)

- void Reprint (const std::string &target_file)

  Print an xml file describing the actual structure of the net.

Protected Member Functions

- **Node_factory** (const bool &use_cloning_Insert)
- virtual void **__Absorb** (Node_factory *to_absorb)
- void **Import_from_XML** (XML_reader *xml_data, const std::string &prefix_config_xml_file)
- Node * **__Find_Node** (Categoric_var *var)
- size_t * **__Get_observed_val** (Categoric_var *var)
- void **__Get_simple_shapes** (std::list< Potential_Shape * > *shapes)
- virtual void **__Get_exponential_shapes** (std::list< std::list< Potential_Exp_Shape * >> *learnable_exp, std::list< Potential_Exp_Shape * > *constant_exp)
- virtual void **__Insert** (Potential_Shape *pot)
- virtual Potential_Exp_Shape * **__Insert** (Potential_Exp_Shape *pot, const bool &weight_tunability)
- void **Insert** (const std::list< Potential_Exp_Shape * > &exponential_potentials, const std::list< bool > &tunability)
- void **Insert** (const std::list< Potential_Shape * > &simple_potentials)
- void Set_Evidences (const std::list< Categoric_var * > &new_observed_vars, const std::list< size_t > &new_observed_vals)

  Set the evidences: identify the variables in the hidden set and the values assumed.
- void Set_Evidences (const std::list< size_t > &new_observed_vals)

  Similar to Node_factory::Set_Evidences(const std::list<Categoric_var>& new_observed_vars, const std::list<size_t>& new_observed*
- void **Belief_Propagation** (const bool &sum_or_MAP)

Static Protected Member Functions

- static void **__Get_simple_shapes** (std::list< Potential_Shape * > *shapes, Node_factory *model)
- static void **__Get_exponential_shapes** (std::list< std::list< Potential_Exp_Shape * >> *learnable_exp, std::list< Potential_Exp_Shape * > *constant_exp, Node_factory *model)

8.33.1 Detailed Description

Interface for describing a net: set of nodes representing random variables.

8.33.2 Member Function Documentation

8.33.2.1 Eval_Log_Energy_function()

```
void EFG::Node::Node_factory::Eval_Log_Energy_function (
            float * result,
            size_t * combination,
            const std::list< Categoric_var * > & var_order_in_combination )
```

Returns the logartihmic value of the energy function.

Energy function E=Pot_1(Y_1,2,....,n) * Pot_2(Y_1,2,....,n) .... * Pot_m(Y_1,2,....,n). The combinations passed as input must contains values for all the variables present in this graph.

Parameters

out	*result*	
in	*combination*	set of values in the combination for which the energy function has to be eveluated
in	*var_order_in_combination*	order of variables considered when assembling combination. They must be references to the variables actually wrapped by this graph.

8.33.2.2 Find_Variable()

```
Categoric_var * EFG::Node::Node_factory::Find_Variable (
            const std::string & var_name )
```

Returns a pointer to the variable in this graph with that name.

Returns NULL when the variable is not present in the graph.

Parameters

in	*var_name*	name to search

8.33.2.3 Get_marginal_distribution()

```
void EFG::Node::Node_factory::Get_marginal_distribution (
            std::list< float > * result,
            Categoric_var * var )
```

Returns the marginal probabilty of the variable passed P(var|model, observations),.

on the basis of the last observations set (see Node_factory::Set_Observation_Set_var)

8.33.2.4 Get_structure()

```
void EFG::Node::Node_factory::Get_structure (
            std::list< const Potential_Shape * > * shapes,
            std::list< std::list< const Potential_Exp_Shape * >> * learnable_exp,
            std::list< const Potential_Exp_Shape * > * constant_exp )
```

Returns the list of potentials constituting the net.

The potentials returned cannot be used for initializing a model. For performing such a task you can build an empty model and then use Absorb.

Parameters

out	*shapes*	list of Simple shapes contained in the model
~~out~~	~~*learnable_exp*~~	~~list of Exponential tunable potentials contained in the model: every sub group share the same weight~~ **Casalino Andrea**
out	*constant_exp*	list of Exponential constant potentials contained in the model

8.33.2.5 Gibbs_Sampling_on_Hidden_set()

```
void EFG::Node::Node_factory::Gibbs_Sampling_on_Hidden_set (
            std::list< std::list< size_t >> * result,
            const unsigned int & N_samples,
            const unsigned int & initial_sample_to_skip )
```

Returns a set of samples of the conditional distribution P(hidden variables | model, observed variables). .

Samples are obtained through Gibbs sampling. Calculations are done considering the last last observations set (see Node_factory::Set_Observation_Set_var)

Parameters

in	*N_samples*	number of desired samples
in	*initial_sample_to_skip*	number of samples to skip for performing Gibbs sampling
out	*result*	returned samples: every element of the list is a combination of values for the hidden set, with the same order returned when calling Node_factory::Get_Actual_Hidden_Set

8.33.2.6 MAP_on_Hidden_set()

```
void EFG::Node::Node_factory::MAP_on_Hidden_set (
            std::list< size_t > * result )
```

Returns the Maximum a Posteriori estimation of the hidden set. .

Values are ordered as returned by Node_factory::Get_Actual_Hidden_Set. Calculations are done considering the last last observations set (see Node_factory::Set_Observation_Set_var)

8.33.2.7 Reprint()

```
void EFG::Node::Node_factory::Reprint (
            const std::string & target_file )
```

Print an xml file describing the actual structure of the net.

Parameters

in	*target_file*	the name of the file were to print the net

8.33.2.8 Set_Evidences() [1/2]

```
void EFG::Node::Node_factory::Set_Evidences (
            const std::list< Categoric_var * > & new_observed_vars,
            const std::list< size_t > & new_observed_vals )  [protected]
```

Set the evidences: identify the variables in the hidden set and the values assumed.

When passing both input as empty list, all the evidences are deleted.

Parameters

in	*new_observed_vars*	the set of variables to consider as evidence
in	*new_observed_vals*	the values assumed as evidences

8.33.2.9 Set_Evidences() [2/2]

```
void EFG::Node::Node_factory::Set_Evidences (
            const std::list< size_t > & new_observed_vals )  [protected]
```

Similar to Node_factory::Set_Evidences(const std::list<Categoric_var*>& new_observed_vars, const std::list<size_t>& new_obser

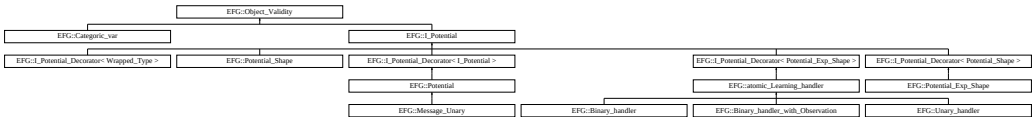
Here only the evidence values are set, assuming the last oberved set imposed

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

- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Header/Node.h
- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Source/Node.cpp

8.34 EFG::Object_Validity Class Reference

Inheritance diagram for EFG::Object_Validity:



Public Member Functions

- const bool & **get_validity** ()

Protected Attributes

- bool **validity_flag**

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

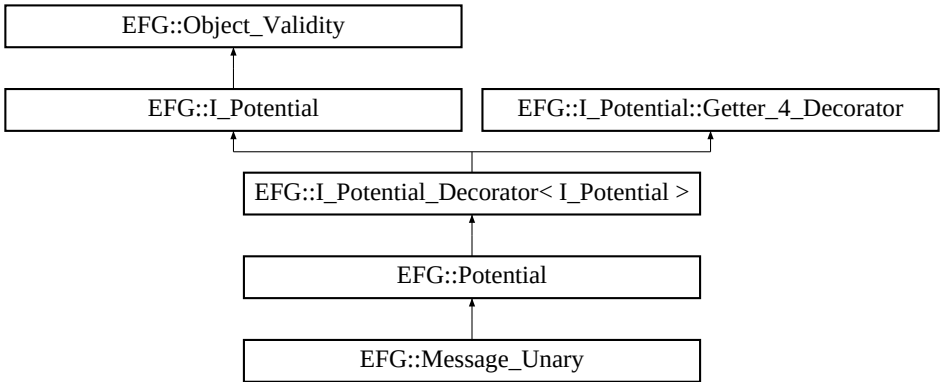
- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Header/Potential.h

8.35 EFG::Potential Class Reference

This class is mainly adopted for computing operations on potentials.

`#include <Potential.h>`

Inheritance diagram for EFG::Potential:



Public Member Functions

- Potential (Potential_Shape *pot)
- Potential (Potential_Exp_Shape *pot)
- Potential (const std::list< Potential * > &potential_to_merge, const bool &use_sparse_format=true)

  The potential to create is obtained by merging a set of potentials referring to the same variables (i.e. values in the image are obtained as a product of the ones in the potential_to_merge set)
- Potential (const std::list< size_t > &val_observed, const std::list< Categoric_var * > &var_observed, Potential *pot_to_reduce)

  The potential to create is obtained by marginalizing the observed variable passed as input.
- void Get_marginals (std::list< float > *prob_distr)

  Obtain the marginal probabilities of the variables in the domain of this potential, when considering this potential only.
- void clone_distribution (Potential_Shape *shape)

  Transfer the values in the distirbution of a potential into a simple shape.

Additional Inherited Members

8.35.1 Detailed Description

This class is mainly adopted for computing operations on potentials.

8.35.2 Constructor & Destructor Documentation

8.35.2.1 Potential() [1/4]

```
EFG::Potential::Potential (
            Potential_Shape * pot )  [inline]
```

Parameters

in	*pot*	potential shape to wrap

8.35.2.2 Potential() [2/4]

```
EFG::Potential::Potential (
            Potential_Exp_Shape * pot )  [inline]
```

Parameters

in	*pot*	exponential potential shape to wrap

8.35.2.3 Potential() [3/4]

```
EFG::Potential::Potential (
            const std::list< Potential * > & potential_to_merge,
            const bool & use_sparse_format = true )
```

The potential to create is obtained by merging a set of potentials referring to the same variables (i.e. values in the image are obtained as a product of the ones in the potential_to_merge set)

Parameters

in	*potential_to_merge*	list of potential to merge, i.e. compute their product
in	*use_sparse_format*	when false, the entire domain is allocated even if some values are equal to 0

8.35.2.4 Potential() [4/4]

```
EFG::Potential::Potential (
            const std::list< size_t > & val_observed,
            const std::list< Categoric_var * > & var_observed,
            Potential * pot_to_reduce )
```

The potential to create is obtained by marginalizing the observed variable passed as input.

Parameters

in	*pot_to_reduce*	the potential from which the variables observed are marginalized
in	*var_observed*	variables observed in pot_to_reduce
in	*val_observed*	values observed (same oreder of var_observed)

8.35.3 Member Function Documentation

8.35.3.1 clone_distribution()

```
void EFG::Potential::clone_distribution (
            Potential_Shape * shape )
```

Transfer the values in the distirbution of a potential into a simple shape.

The variables involved in the shape passed must be the same of this potential when considering this potential only

Parameters

in	*shape*	the potential shape that will received the cloned values

8.35.3.2 Get_marginals()

```
void EFG::Potential::Get_marginals (
            std::list< float > * prob_distr )
```

Obtain the marginal probabilities of the variables in the domain of this potential, when considering this potential only.

Parameters

in	*prob_distr*	marginals

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

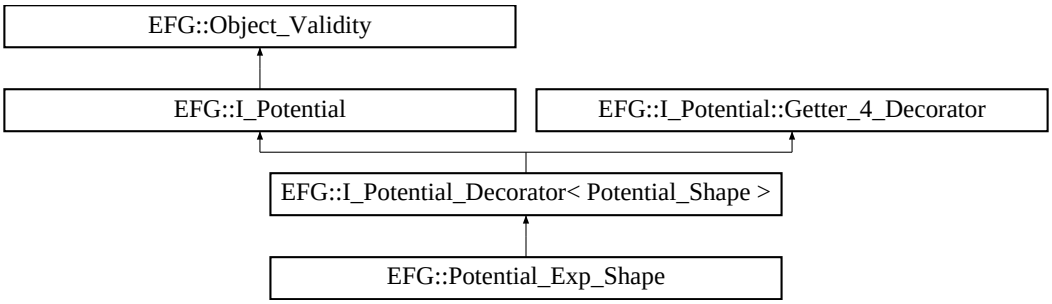
- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Header/Potential.h
- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Source/Potential.cpp

8.36 EFG::Potential_Exp_Shape Class Reference

Represents an exponential potential, wrapping a normal shape one: every value of the domain are assumed as exp(mWeight * val_in_shape_wrapped)

```
#include <Potential.h>
```

Inheritance diagram for EFG::Potential_Exp_Shape:



Classes

- struct Getter_weight_and_shape

Public Member Functions

- Potential_Exp_Shape (Potential_Shape *shape, const float &w=1.f)

  When building a new exponential shape potential, all the values of the domain are computed according to the new shape passed as input.
- Potential_Exp_Shape (const std::list< Categoric_var * > &var_involved, const std::string &file_to_read, const float &w=1.f)

  When building a new exponential shape potential, all the values of the domain are computed according to the potential shape to wrap, which is instantiated in the constructor by considering the textual file provided, see also Potential_Shape(const std::list<Categoric_var>& var_involved, const std::string& file_to_read)*
- Potential_Exp_Shape (const Potential_Exp_Shape *to_copy, const std::list< Categoric_var * > &var_involved)
- const float & get_weight ()

  Returns the weight assigned to this potential.
- void Substitute_variables (const std::list< Categoric_var * > &new_var)

  Use this method for replacing the set of variables this potential must refer. Variables in new_var must be equal in number to the original set of variables and must have the same sizes.
- const Potential_Shape * Get_wrapped_Shape () const

  returns the wrapped Potential_Shape

Protected Member Functions

- virtual std::list< I_Distribution_value * > * **Get_distr** ()
- void **Wrap** (Potential_Shape *shape)

Protected Attributes

- float **mWeight**
- std::list< I_Distribution_value * > Distribution

Additional Inherited Members

8.36.1 Detailed Description

Represents an exponential potential, wrapping a normal shape one: every value of the domain are assumed as exp(mWeight * val_in_shape_wrapped)

8.36.2 Constructor & Destructor Documentation

8.36.2.1 Potential_Exp_Shape() [1/3]

```
EFG::Potential_Exp_Shape::Potential_Exp_Shape (
            Potential_Shape * shape,
            const float & w = 1.f )
```

When building a new exponential shape potential, all the values of the domain are computed according to the new shape passed as input.

Parameters

in	*shape*	shape distribution to wrap
in	*w*	weight of the exponential

8.36.2.2 Potential_Exp_Shape() [2/3]

```
EFG::Potential_Exp_Shape::Potential_Exp_Shape (
            const std::list< Categoric_var * > & var_involved,
            const std::string & file_to_read,
            const float & w = 1.f )
```

When building a new exponential shape potential, all the values of the domain are computed according to the potential shape to wrap, which is instantiated in the constructor by considering the textual file provided, see also Potential_Shape(const std::list<Categoric_var*>& var_involved, const std::string& file_to_read)

Parameters

in	*var_involved*	variables involved in the domain of this variables
in	*file_to_read*	textual file to read containing the values for the image
in	*w*	weight of the exponential

8.36.2.3 Potential_Exp_Shape() [3/3]

```
EFG::Potential_Exp_Shape::Potential_Exp_Shape (
            const Potential_Exp_Shape * to_copy,
            const std::list< Categoric_var * > & var_involved )
```

Use this constructor for cloning a shape, but considering a different set of variables. Variables in var_involved must be equal in number to those in the potential to clone and must have the same sizes of the variables involved in the potential to clone.

Parameters

in	*to_copy*	exp_shape to clone
in	*var_involved*	new set of variables to consider when cloning

8.36.3 Member Function Documentation

8.36.3.1 Substitute_variables()

```
void EFG::Potential_Exp_Shape::Substitute_variables (
            const std::list< Categoric_var * > & new_var )  [inline]
```

Use this method for replacing the set of variables this potential must refer. Variables in new_var must be equal in number to the original set of variables and must have the same sizes.

Parameters

in	*new_var*	variables to consider for the substitution

8.36.4 Member Data Documentation

8.36.4.1 Distribution

```
std::list<I_Distribution_value*> EFG::Potential_Exp_Shape::Distribution  [protected]
```

Weight assumed for modulating the exponential (see description of the class)

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

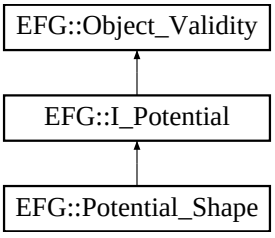
- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Header/Potential.h
- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Source/Potential.cpp

8.37 EFG::Potential_Shape Class Reference

It's the only possible concrete potential. It contains the domain and the image of the potential.

`#include <Potential.h>`

Inheritance diagram for EFG::Potential_Shape:



Public Member Functions

- Potential_Shape (const std::list< Categoric_var * > &var_involved)

  When building a new shape potential, all values of the image are assumed as all zeros.
- Potential_Shape (const std::list< Categoric_var * > &var_involved, const std::string &file_to_read)
- Potential_Shape (const std::list< Categoric_var * > &var_involved, const bool &correlated_or_not)

  Returns simple correlating or anti_correlating shapes. .
- Potential_Shape (const Potential_Shape *to_copy, const std::list< Categoric_var * > &var_involved)
- void Add_value (const std::list< size_t > &new_indeces, const float &new_val)

  Add a new value in the image set.
- void Set_ones ()

  All values in the image of the domain are set to 1.
- void Set_random (const float zeroing_threashold=1.f)

  All values in the image of the domain are randomly set.
- void Normalize_distribution ()

  All values in the image of the domain are multipled by a scaling factor, in order to to have maximal value equal to 1. Exploited for computing messages.
- void Substitute_variables (const std::list< Categoric_var * > &new_var)

  Use this method for replacing the set of variables this potential must refer. Variables in new_var must be equal in number to the original set of variables and must have the same sizes.
- void Copy_Distribution (std::list< std::list< size_t >> *combinations, std::list< float > *values) const

  Returns the distribution describing this potential (i.e. the value assumed for every possible combination of the involved variables). If the potential is not valid, nothing is returned.

Protected Member Functions

- bool **__Check_add_value** (const std::list< size_t > &indices)
- virtual const std::list< Categoric_var * > * **Get_involved_var** () const
- virtual std::list< I_Distribution_value * > * **Get_distr** ()

Additional Inherited Members

8.37.1 Detailed Description

It's the only possible concrete potential. It contains the domain and the image of the potential.

8.37.2 Constructor & Destructor Documentation

8.37.2.1 Potential_Shape() [1/4]

```
EFG::Potential_Shape::Potential_Shape (
            const std::list< Categoric_var * > & var_involved )
```

When building a new shape potential, all values of the image are assumed as all zeros.

Parameters

in	*var_involved*	variables involved in the domain of this variables

8.37.2.2 Potential_Shape() [2/4]

```
EFG::Potential_Shape::Potential_Shape (
            const std::list< Categoric_var * > & var_involved,
            const std::string & file_to_read )
```

Parameters

in	*var_involved*	variables involved in the domain of this variables
in	*file_to_read*	textual file to read containing the values for the image

8.37.2.3 Potential_Shape() [3/4]

```
EFG::Potential_Shape::Potential_Shape (
            const std::list< Categoric_var * > & var_involved,
            const bool & correlated_or_not )
```

Returns simple correlating or anti_correlating shapes. .

A simple correlating shape is a distribution having a value of 1 for every combinations {0,0,...,0}; {1,1,...,1} etc. and 0 for all other combinations. A simple anti_correlating shape is a distribution having a value of 0 for every combinations {0,0,...,0}; {1,1,...,1} etc. and 1 for all other combinations.

Parameters

in	*var_involved*	variables involved in the domain of this variables: they must have all the same size
in	*correlated_or_not*	when true produce a simple correlating shape, when false produce a anti_correlating function

8.37.2.4 Potential_Shape() [4/4]

```
EFG::Potential_Shape::Potential_Shape (
            const Potential_Shape * to_copy,
            const std::list< Categoric_var * > & var_involved )
```

Use this constructor for cloning a shape, but considering a different set of variables. Variables in var_involved must be equal in number to those in the potential to clone and must have the same sizes of the variables involved in the potential to clone.

Parameters

in	*to_copy*	shape to clone
in	*var_involved*	new set of variables to consider when cloning

8.37.3 Member Function Documentation

8.37.3.1 Add_value()

```
void EFG::Potential_Shape::Add_value (
            const std::list< size_t > & new_indeces,
            const float & new_val )
```

Add a new value in the image set.

Parameters

in	*new_indices*	combination related to the new value to add for the image
in	*new_val*	new val to insert

8.37.3.2 Copy_Distribution()

```
void EFG::Potential_Shape::Copy_Distribution (
            std::list< std::list< size_t >> * combinations,
            std::list< float > * values ) const
```

Returns the distribution describing this potential (i.e. the value assumed for every possible combination of the involved variables). If the potential is not valid, nothing is returned.

Parameters

out	*combinations*	combinations in the domain of this potential
out	*values*	the values assumed in the above combinations by the distribution

8.37.3.3 Substitute_variables()

```
void EFG::Potential_Shape::Substitute_variables (
            const std::list< Categoric_var * > & new_var )
```

Use this method for replacing the set of variables this potential must refer. Variables in new_var must be equal in number to the original set of variables and must have the same sizes.

Parameters

in	*new_var*	variables to consider for the substitution

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

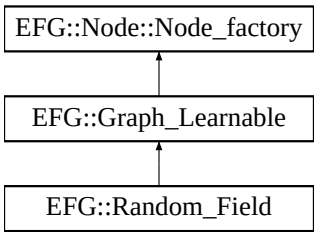
- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Header/Potential.h
- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Source/Potential.cpp

8.38 EFG::Random_Field Class Reference

This class describes a generic Random Field, not having a particular set of variables observed.

`#include <Graphical_model.h>`

Inheritance diagram for EFG::Random_Field:



Public Member Functions

- Random_Field (const bool &use_cloning_Insert=true)

  empty constructor
- Random_Field (const std::string &config_xml_file, const std::string &prefix_config_xml_file="")

  The model is built considering the information contained in an xml configuration file. .
- Random_Field (const std::list< Potential_Exp_Shape * > &potentials_exp, const bool &use_cloning_↩ Insert=true, const std::list< bool > &tunable_mask={}, const std::list< Potential_Shape * > &shapes={})

  This constructor initializes the graph with the specified potentials passed as input.
- void Insert (Potential_Shape *pot)

  Similar to Graph::Insert(Potential_Shape pot)*
- void Insert (Potential_Exp_Shape *pot, const bool &is_weight_tunable=true)

  Similar to Graph::Insert(Potential_Exp_Shape pot).*
- void Insert (Potential_Exp_Shape *pot, const std::list< Categoric_var * > &vars_of_pot_whose_weight_is↩ _to_share)

  Insert a tunable exponential shape, whose weight is shared with another already inserted tunable shape.
- void Set_Evidences (const std::list< Categoric_var * > &new_observed_vars, const std::list< size_t > &new_observed_vals)

  see Node::Node_factory::Set_Evidences(const std::list<Categoric_var>& new_observed_vars, const std::list<size_t>& new_observe*
- void Set_Evidences (const std::list< size_t > &new_observed_vals)

  see Node::Node_factory::Set_Evidences(const std::list<size_t>& new_observed_vals)
- void Absorb (Node_factory *to_absorb)

  Absorbs all the variables and the potentials contained in the model passed as input.

Additional Inherited Members

8.38.1 Detailed Description

This class describes a generic Random Field, not having a particular set of variables observed.

8.38.2 Constructor & Destructor Documentation

8.38.2.1 Random_Field() [1/3]

```
EFG::Random_Field::Random_Field (
            const bool & use_cloning_Insert = true )  [inline]
```

empty constructor

Parameters

in	*use_cloning_Insert*	when is true, every time an Insert of a novel potential is called, a copy of that potential is actually inserted. Otherwise, the passed potential is inserted as is: this can be dangerous, cause that potential cna be externally modified, but the construction of a novel graph is faster.

8.38.2.2 Random_Field() [2/3]

```
EFG::Random_Field::Random_Field (
            const std::string & config_xml_file,
            const std::string & prefix_config_xml_file = "" )
```

The model is built considering the information contained in an xml configuration file. .

See Section 3 of the documentation for the syntax to adopt.

Parameters

in	*configuration*	file
in	*prefix*	to use. The file prefix_config_xml_file/config_xml_file is searched.

8.38.2.3 Random_Field() [3/3]

```
EFG::Random_Field::Random_Field (
            const std::list< Potential_Exp_Shape * > & potentials_exp,
```

```
            const bool & use_cloning_Insert = true,
            const std::list< bool > & tunable_mask = {},
            const std::list< Potential_Shape * > & shapes = {} )
```

This constructor initializes the graph with the specified potentials passed as input.

Parameters

in	*potentials_exp*	the initial set of exponential potentials to insert (can be empty)
in	*use_cloning_Insert*	when is true, every time an Insert of a novel potential is called (this includes the passed potentials), a copy of that potential is actually inserted. Otherwise, the passed potential is inserted as is: this can be dangerous, cause that potential cna be externally modified, but the construction of a novel graph is faster.
in	*tunable_mask*	when passed as non default value, it is must have the same size of potentials. Every value in this list is true if the corresponfing potential in the potentials list is tunable, i.e. has a weight whose value can vary with learning
in	*shapes*	A list of additional non learnable potentials to insert in the model

8.38.3 Member Function Documentation

8.38.3.1 Absorb()

```
void EFG::Random_Field::Absorb (
            Node_factory * to_absorb )  [inline]
```

Absorbs all the variables and the potentials contained in the model passed as input.

Consistency checks are performed: it is possible that some inconsistent components in the model passed will be not absorbed.

8.38.3.2 Insert() [1/2]

```
void EFG::Random_Field::Insert (
            Potential_Exp_Shape * pot,
            const bool & is_weight_tunable = true )  [inline]
```

Similar to Graph::Insert(Potential_Exp_Shape* pot).

Parameters

in	*is_weight_tunable*	When true, you are specifying that this potential has a weight learnable, otherwise the value of the weight is assumed constant.

8.38.3.3 Insert() [2/2]

```
void EFG::Random_Field::Insert (
            Potential_Exp_Shape * pot,
            const std::list< Categoric_var * > & vars_of_pot_whose_weight_is_to_share )
```

Insert a tunable exponential shape, whose weight is shared with another already inserted tunable shape.

This allows having many exponential tunable potetials which share the value of the weight: this is automatically account for when performing learning.

Parameters

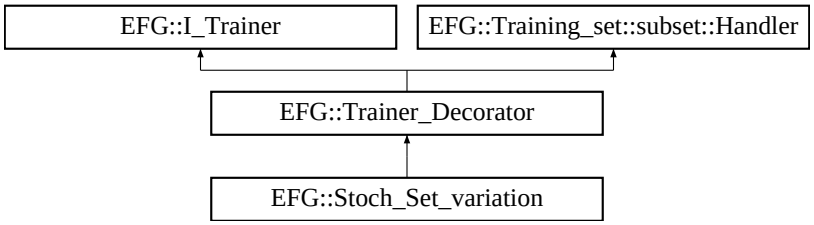
in	*vars_of_pot_whose_weight_is_to_share*	the list of varaibles involved in a potential already inserted whose weight is to share with the potential passed. They must be references to the variables actually wrapped into the model.

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

- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Header/Graphical_model.h
- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Source/Graphical_model.cpp

8.39 EFG::Stoch_Set_variation Class Reference

Inheritance diagram for EFG::Stoch_Set_variation:



Public Member Functions

- **Stoch_Set_variation** (Advancer_Concrete *to_wrap, const float &percentage_to_use)
- void **Train** (Graph_Learnable *model_to_train, Training_set *Train_set, const unsigned int &Max_Iterations, std::list< float > *descend_story)

Additional Inherited Members

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

- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Source/Trainer.cpp

8.40 EFG::Training_set::subset Struct Reference

This class is describes a portion of a training set, obtained by sampling values in the original set. Mainly used by stochastic gradient computation strategies.

`#include <Training_set.h>`

Classes

- struct Handler

Public Member Functions

- subset (Training_set *set, const float &size_percentage=1.f)
- const bool & **Get_validity** ()

8.40.1 Detailed Description

This class is describes a portion of a training set, obtained by sampling values in the original set. Mainly used by stochastic gradient computation strategies.

8.40.2 Constructor & Destructor Documentation

8.40.2.1 subset()

```
EFG::Training_set::subset::subset (
            Training_set * set,
            const float & size_percentage = 1.f )
```

Parameters

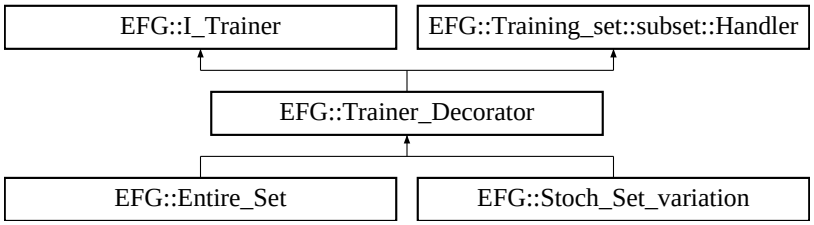
in	*set*	the training set from which this subset must be extracted
in	*size_percentage*	percentage to use for the extraction

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

- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Header/Training_set.h
- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Source/Training_set.cpp

8.41 EFG::Trainer_Decorator Class Reference

Inheritance diagram for EFG::Trainer_Decorator:



Public Member Functions

- **Trainer_Decorator** (Advancer_Concrete ∗to_wrap)
- void **Clean_Up** ()

Protected Member Functions

- void **__check_training_possible** (Graph_Learnable ∗model_to_train, Training_set ∗Train_set)

Protected Attributes

- Advancer_Concrete ∗ **Wrapped**

Additional Inherited Members

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

- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Source/Trainer.cpp

8.42 EFG::Training_set Class Reference

This class is used for describing a training set for a graph.

`#include <Training_set.h>`

Classes

- class Basic_Extractor

  Basic extractor, see Training_set(const std::list<std::string>& variable_names, std::list<Array> samples, I_↩ Extractor<Array>∗ extractor)
- class I_Extractor

  This class is adopted for parsing a set of samples to import as a novel training set. You have to derive yout custom extractor, implementing the two vritual method.
- struct subset

  This class is describes a portion of a training set, obtained by sampling values in the original set. Mainly used by stochastic gradient computation strategies.

Public Member Functions

- Training_set (const std::string &file_to_import)
- template<typename Array >
  Training_set (const std::list< std::string > &variable_names, std::list< Array > &samples, I_Extractor< Array > *extractor)

  Similar to Training_set(const std::string& file_to_import),.
- template<typename Array >
  Training_set (const std::list< Categoric_var * > &variable_in_the_net, std::list< Array > &samples, I_Extractor< Array > *extractor)

  Same as Training_set(const std::list<std::string>& variable_names, std::list<Array> samples, I_Extractor<Array> extractor) passing the variables involved instead of the names.*
- void Print (const std::string &file_name)

  This training set is reprinted in the location specified.
- const bool & Get_validity ()

  Returns true in case this set can be used for performing the training of a model, otherwise is false.

8.42.1 Detailed Description

This class is used for describing a training set for a graph.

A set is described in a textual file, where the first row must contain the list of names of the variables (all the variables) constituting a graph. All other rows are a single sample of the set, reporting the values assumed by the variables, with the order described by the first row

8.42.2 Constructor & Destructor Documentation

8.42.2.1 Training_set() [1/2]

```
EFG::Training_set::Training_set (
            const std::string & file_to_import )
```

Parameters

in	*file_to_import*	file containing the set to import

8.42.2.2 Training_set() [2/2]

```
template<typename Array >
EFG::Training_set::Training_set (
            const std::list< std::string > & variable_names,
            std::list< Array > & samples,
            I_Extractor< Array > * extractor )  [inline]
```

Similar to Training_set(const std::string& file_to_import),.

with the difference that the training set is not red from a textual file but it is imported from a list of container (generic can be list, vector or other) describing the samples of the set. You have to derived your own extractor for managing your particular container. Basic_Extractor is a baseline extractor that can be used for all those type having the method size() and the operator[].

Parameters

in	*variable_names*	the ordered list of variables to assume for the samples
in	*samples*	the list of generic Array representing the samples of the training set
in	*extractor*	the particular extractor to use, see I_Extractor

8.42.3 Member Function Documentation

8.42.3.1 Print()

```
void EFG::Training_set::Print (
            const std::string & file_name )
```

This training set is reprinted in the location specified.

Parameters

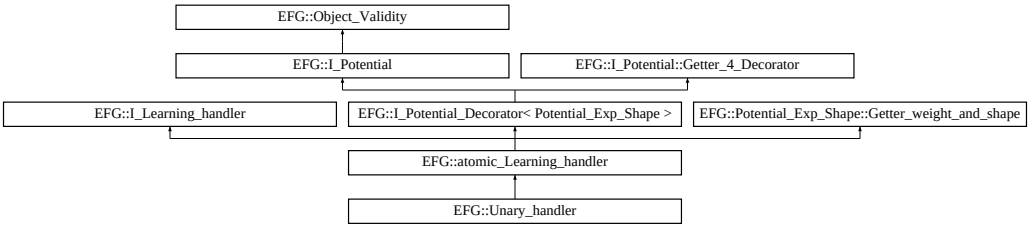
in	*file_name*	is the path of the file where the set must be printed

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

- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Header/Training_set.h
- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Source/Training_set.cpp

8.43 EFG::Unary_handler Class Reference

Inheritance diagram for EFG::Unary_handler:



Public Member Functions

- **Unary_handler** (Node *N, Potential_Exp_Shape *pot_to_handle)

Additional Inherited Members

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

- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Source/Graphical_model.cpp

8.44 EFG::Graph_Learnable::Weights_Manager Struct Reference

Static Public Member Functions

- static void Get_tunable_w (std::list< float > *w, Graph_Learnable *model)

  Returns the values of the tunable weights, those that can vary when learning the model.

Friends

- class **I_Trainer**

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

- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Header/Graphical_model.h
- C:/Librerie_C/My_Code/Easy_Factor_Graphs/EFG/Source/Graphical_model.cpp

Index