Segugio; the flexible and efficient tool for reasoning on graphical models

Casalino Andrea and Nicola Massarenti
andrecasa91@gmail.com , nicola.massarenti@gmail.com

Chapter 1

Foundamental concepts about graphical models

METTERE subito in evidenza caratteristiche strane che questa libreria ha rispetto alle altre.

This Section will provide the reader a brief background about the basic concepts in probabilistic graphical models. Moreover, a precise notation will be introduced and used for the rest of this guide. If you are familiar with the basic concepts, just read the following Section.

1.1 Preliminaries

Undirect Graphical models are networks made of variables and factors.
In particular, this library is intended for managing categorical variables. Formally, the generic categorical variable V has a discrete domain Dom:

$$Dom(V) = \{v_0, \cdots, v_n\} \tag{1.1}$$

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:

$$Dom(X) = Dom(V_1) \times Dom(V_2) \times Dom(V_3) \times Dom(V_4) \tag{1.2}$$

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 1.1. The union $X = \bigcup_{i=1}^{3} V_i$ is a categoric variable whose domain is made by the combinations reported in table 1.2.

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.

[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 1.1 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 1.2 Example of domains for the group of variables $V_{1,2,3}$.

Knowing the joint probability of $V_{1,\cdots,m}$, the probability distribution of a subset $S\{S_1, \cdots\} \subset \{V_1, \cdots, X_m\}$ 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:

$$\mathbb{P}(S = s) = \sum_{\forall \hat{c} \in Dom(C)} \mathbb{P}(S = s, C = \hat{c}) \tag{1.3}$$

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:

$$\Phi_i(D^i) = \Phi_i(X, Y, Z) : \textsc{Domain}(X) \times \textsc{Domain}(Y) \times \textsc{Domain}(Z) \longrightarrow \mathbb{R}^+ \tag{1.4}$$

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:

$$E(V_{1,\cdots,m}) = \Phi_1(D^1) \cdot \cdots \cdot \Phi_p(D^p) = \prod_{i=1}^{p} \Phi_i(D^i) \tag{1.5}$$

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

$$\mathbb{P}(V_{1,\cdots,m}) = \frac{E(V_{1,\cdots,m})}{\mathcal{Z}} \tag{1.6}$$

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

$$\mathcal{Z} = \sum_{\forall \tilde{V}_{1,\cdots,m} \in Dom(\bigcup_{i=1,\cdots,m} V_i))} E(\tilde{V}_{1,\cdots,m}) \tag{1.7}$$

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.

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

a_0	a_1	a_2
0	2	0.5

Table 1.4 The values in the image of $\Phi_u(A)$.

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

$$\begin{aligned} \textsc{Dom}(A) &= \{a_1, a_2, a_3\} \\ \textsc{Dom}(B) &= \{b_1, b_2, b_3, b_4, b_5\} \end{aligned} \tag{1.8}$$

The values assumed by $\Phi_b(A,B)$ are described by table 1.3. 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 $\Phi_u(A)$ a Unary potential involving variable A. The values characterizing Φ_u can be stored in a simple vector, see table 1.4. 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]:

$$\begin{aligned} \mathbb{P}(a_0, b_1) &= \frac{\Phi_b(a_0, b_1)}{\mathcal{Z}} = \frac{4}{\mathcal{Z}} = 0.3333 \\ \mathbb{P}(a_2, b_2) &= \frac{\Phi_b(a_2, b_2)}{\mathcal{Z}} = \frac{5}{\mathcal{Z}} = 0.4167 \\ \mathbb{P}(a_0, b_0) &= \frac{\Phi_b(a_0, b_0)}{\mathcal{Z}} = \mathbb{P}(a_1, b_1) = \mathbb{P}(a_2, b_4) = \frac{1}{\mathcal{Z}} = 0.0833 \end{aligned} \tag{1.9}$$

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

$$\mathcal{Z} = \sum_{\forall i=\{0,1,2\}, \forall j=\{0,1,2,3,4\}} \Phi_b(A = a_i, B = b_j) = 12 \tag{1.10}$$

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 5.40 handles this kind of factors.
- Exponential shape. This kind of factors are indicated with Ψ_i and their image set is defined as follows:

  $$\Psi_i(X) = exp(w \cdot \Phi_i(X)) \tag{1.11}$$

  where Φ_i is a Simple shape. 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 1.6), otherwise is a constant . Exponential shapes with fixed weight will be denoted with $\overline{\Psi}_i$. Class Potential_Exp_Shape 5.39 handles this kind of factors.

Figure 1.1 resumes all the possible categories of factors that can be present in the models handled by this library.

Figure 1.2 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. 1.2 will be adopted for the rest of this

[2] combinations having a null probability were omitted

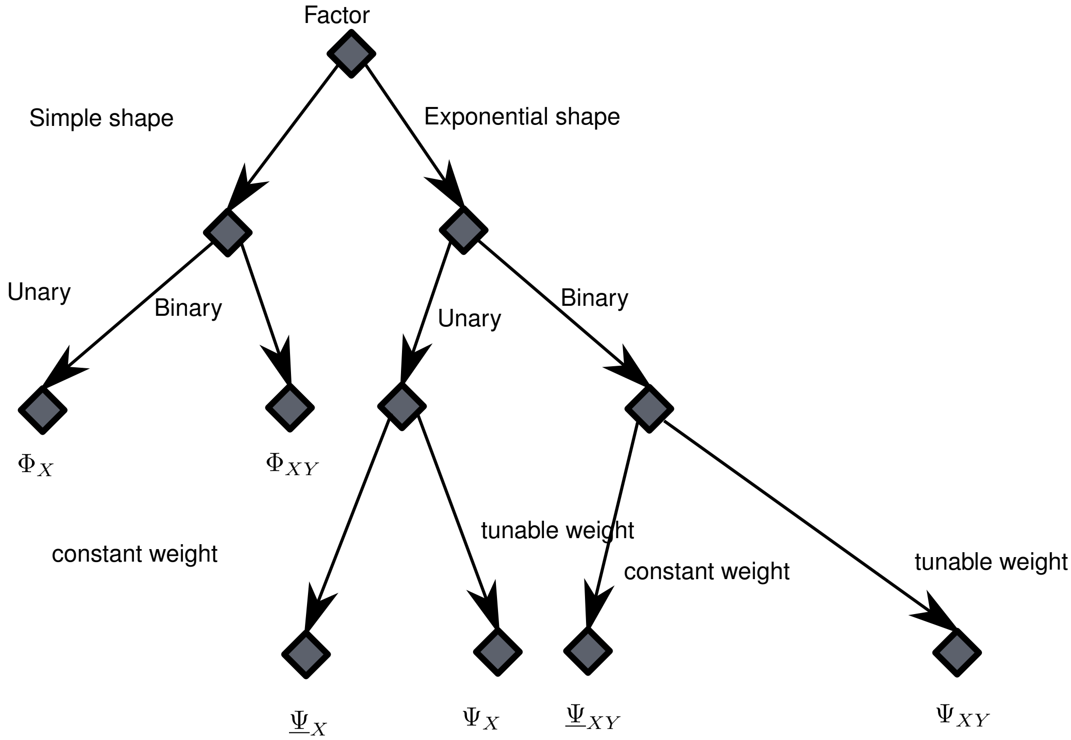


Figure 1.1 All the possible categories of factors, with the corresponding notation.

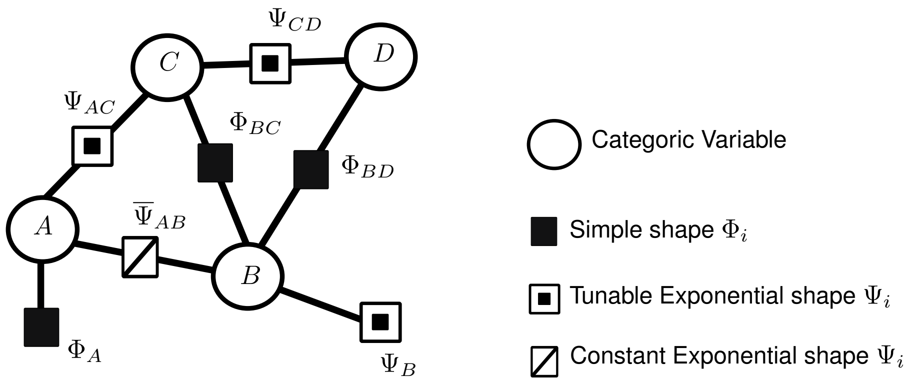


Figure 1.2 Example of graph: the legend of the right applies.

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. 1.2 is computable as follows:

$$
\begin{aligned}
\mathbb{P}(A,B,C,D) &= \frac{E(A,B,C,D)}{\mathcal{Z}(\alpha,\beta,\gamma,\delta)} = \frac{E(A,B,CD)}{\sum_{\tilde{A},\tilde{B},\tilde{C},\tilde{D}} E(\tilde{A},\tilde{B},\tilde{C},\tilde{D})} \\
E(A,B,C,D) &= \Phi_A(A) \cdot exp(\alpha \Phi_{AC}(A,C)) \cdot exp(\beta \Phi_{AB}(A,B)) \cdots \\
\cdots & \Phi_{BC}(B,C) \cdot exp(\gamma \Phi_{CD}(C,D)) \cdot \Phi_{BD}(B,D) \cdot exp(\delta \Phi_B(B))
\end{aligned}
\tag{1.12}
$$

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 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 is the variable obtained merging all the $H_1, \cdots, H_t$, while O is similar defined considering $O_1, \cdots, O_f$. Knowing the joint probability distribution of variables in $\mathcal{V}$ (equation (1.6)) the conditional distribution of H w.r.t. O can be determined as follows:

$$\begin{aligned}\mathbb{P}(H = h|O = o) &= \frac{\mathbb{P}(H = h, O = o)}{\sum_{\forall \hat{h} \in Dom(H)} \mathbb{P}(H = \hat{h}, O = o)} \\ &= \frac{E(h, o)}{\sum_{\forall \hat{h} \in Dom(H)} E(\hat{h}, o)} = \frac{E(h, o)}{\mathcal{Z}(o)}\end{aligned} \quad (1.13)$$

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:

$$\mathbb{P}(H_i = h_i|O = o) = \sum_{\forall \tilde{h} \in \{\mathcal{H} \setminus H_i\}} \mathbb{P}(H_i = h_i, \tilde{h}|O = o) \quad (1.14)$$

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 5.41. 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 5.13 is in charge of handling Conditional Random Field. Both Random Fields and Conditional Random Fields can be learnt knowing a training set, see Section 1.6. On the opposite, class Graph 5.20 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 1.2 will introduce the message passing algorithm, which is the pillar for performing belief propagation. Section 1.3 will expose the concept of maximum a posteriori estimation, useful when querying a graph, while Section 1.4 will address Gibbs sampling for producing a training set of a known model. Section 1.5 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, 1.6 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.

1.2 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.

1.2.1 Belief propagation

Consider the graph reported in [3] Figure 1.3. 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:

$$\mathbb{P}(x_1) = \sum_{\forall \tilde{x}_{2,3,4,5} \in \cup_{i=2}^{5} X_i} \mathbb{P}(x_1, \tilde{x}_{2,3,4,5}) \quad (1.15)$$

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

$$\mathbb{P}(x_1) \propto \sum_{\tilde{x}_{2,3,4,5}} E(x_1, \tilde{x}_{2,3,4,5}) \quad (1.16)$$

[3] Without loss of generality we assumed all the factors as simple shapes

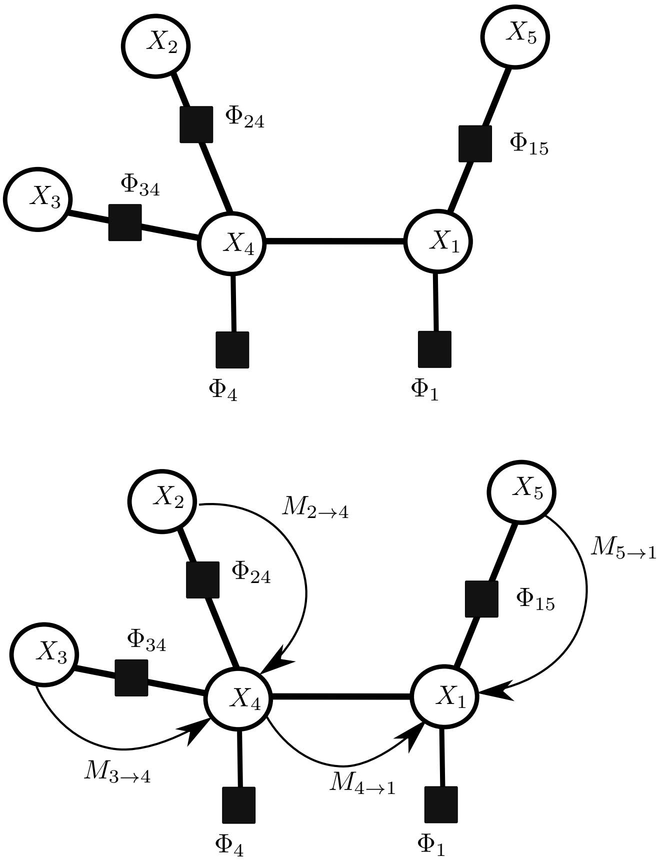


Figure 1.3 Example of graph adopted for explaining the message passing algorithm. Below are reported the messages to compute for obtaining the marginal probability of variable x_1

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

$$\mathbb{P}(x_1) \propto \Phi_1(x_1) \sum_{\tilde{x}_5} \Phi_{15}(x_1, \tilde{x}_5) \sum_{\tilde{x}_4} \Phi_{14}(x_1, \tilde{x}_4)\Phi_4(\tilde{x}_4) \sum_{\tilde{x}_2} \Phi_{24}(\tilde{x}_{2,4}) \sum_{\tilde{x}_3} \Phi_{34}(\tilde{x}_{3,4}) \tag{1.17}$$

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:

$$\begin{aligned} M_{2\to4}(\tilde{x}_4) &= \sum_{\tilde{x}_2} \Phi_{24}(\tilde{x}_{2,4}) \\ M_{3\to4}(\tilde{x}_4) &= \sum_{\tilde{x}_3} \Phi_{34}(\tilde{x}_{3,4}) \end{aligned} \tag{1.18}$$

Inserting $M_{2\to4}$ and $M_{3\to4}$ into equation (1.17) leads to:

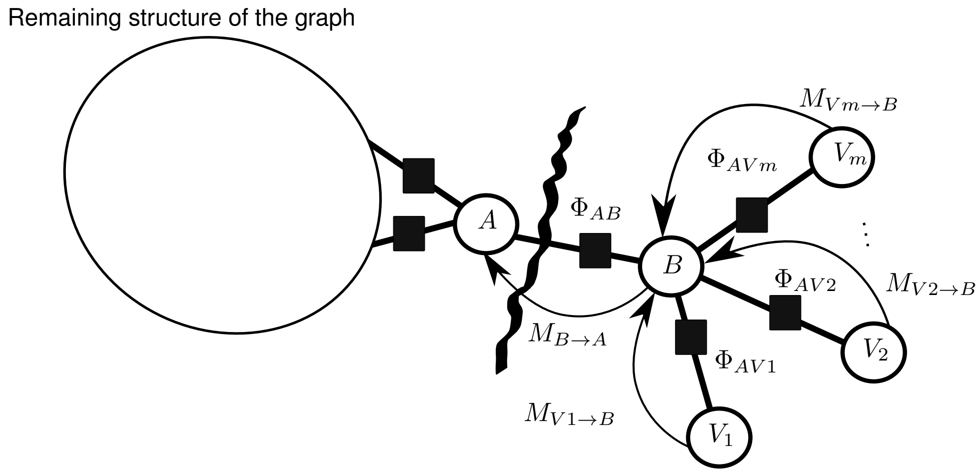
$$\mathbb{P}(x_1) \propto \Phi_1(x_1) \sum_{\tilde{x}_5} \Phi_{15}(x_1, \tilde{x}_5) \sum_{\tilde{x}_4} \Phi_{14}(x_1, \tilde{x}_4)\Phi_4(\tilde{x}_4) M_{2\to4}(\tilde{x}_4) M_{3\to4}(\tilde{x}_4) \tag{1.19}$$

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

$$\begin{aligned} M_{4\to1(x_1)} &= \sum_{\tilde{x}_4} \Phi_{14}(x_1, \tilde{x}_4)\Phi_4(\tilde{x}_4) M_{2\to4}(\tilde{x}_4) M_{3\to4}(\tilde{x}_4) \\ M_{5\to1}(x_1) &= \sum_{\tilde{x}_5} \Phi_{15}(x_1, \tilde{x}_5) \end{aligned} \tag{1.20}$$

After inserting $M_{4\to1}$ and $M_{5\to1}$ into equation (1.19) we obtain:

$$\begin{aligned} \mathbb{P}(x_1) &\propto \Phi_1(x_1) M_{4\to1}(x_1) M_{5\to1}(x_1) \\ \mathbb{P}(x_1) &= \frac{\Phi_1(x_1) M_{4\to1}(x_1) M_{5\to1}(x_1)}{\sum_{\tilde{x}_1} \Phi_1(\tilde{x}_1) M_{4\to1}(\tilde{x}_1) M_{5\to1}(\tilde{x}_1)} \end{aligned} \tag{1.21}$$



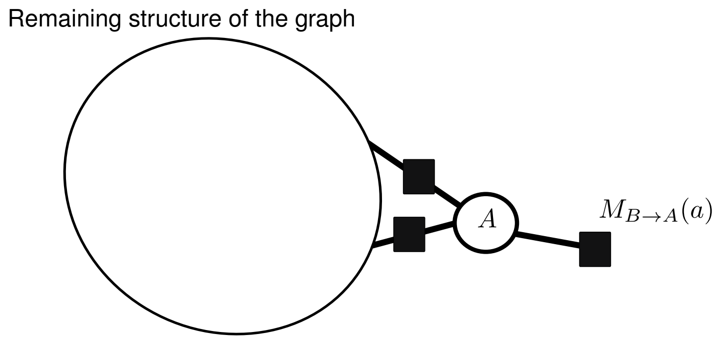


Figure 1.4 On the top the general mechanism involved in the message computation; on the bottom the simplification of the graph considering the computed message.

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 1.3 resumes the computations exposed. Notice that the computation of $M_{4\to1}$ must be done after computing the messages $M_{2\to4}$ and $M_{2\to4}$, while $M_{5\to1}$ can be computed independently from all the others.

1.2.2 Message Passing

The aforementioned considerations can be extended to a general structured graph. Look at Figure 1.4: 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:

$$M_{B\to A}(a) = \sum_{\tilde{b}} \Phi_{AB}(a,\tilde{b}) M_{V1\to B}(\tilde{b}) \cdot \cdots \cdot M_{Vm\to B}(\tilde{b}) \qquad (1.22)$$

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 1.4. 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 5.23 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 1.5. 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

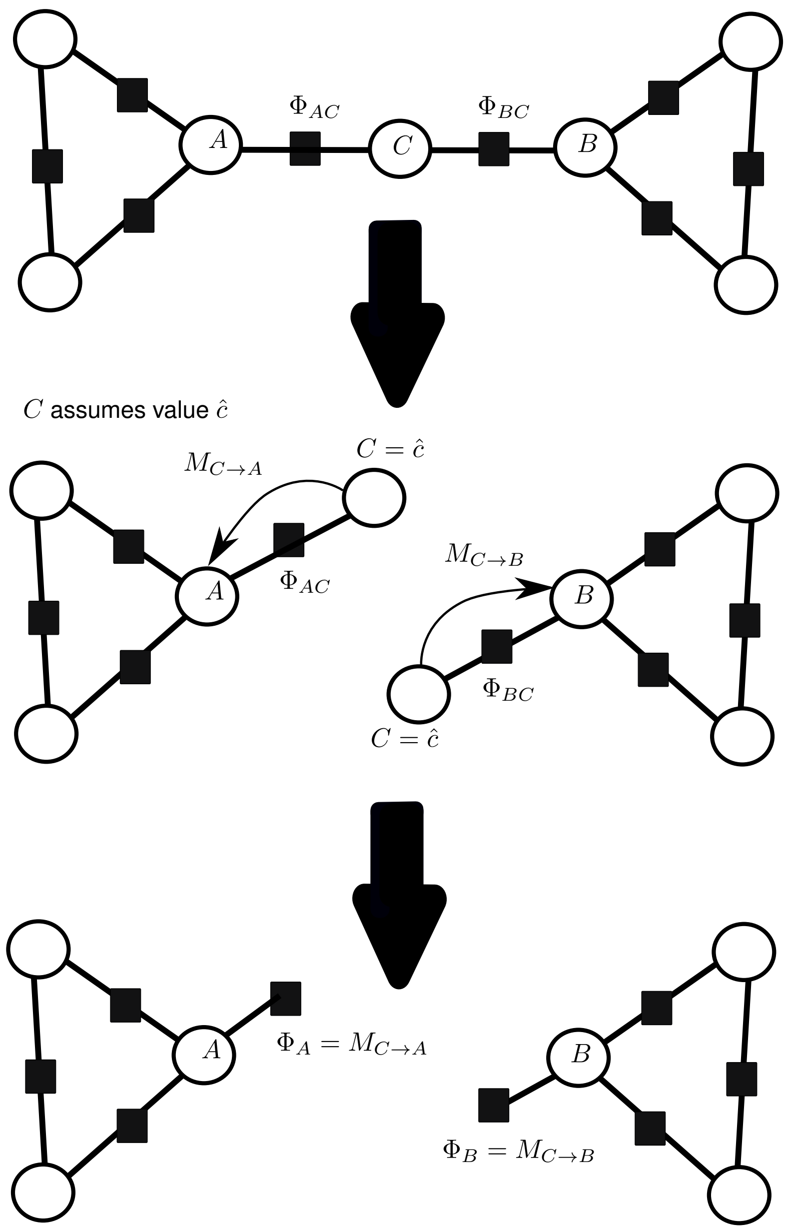


Figure 1.5 When variable C become an evidence, is temporary deleted from the graph, replaced by messages.

an evidence assuming a value equal to $\hat{c}$, then the messages sent to A and B can be computed independently as follows:

$$\begin{aligned} M_{C \to A}(a) &= \Phi_{AC}(a, \hat{c}) \\ M_{C \to B}(b) &= \Phi_{BC}(b, \hat{c}) \end{aligned} \tag{1.23}$$

Therefore all the variables that become evidences can be considered as a leaves of the graph sending messages to all the neighbouring nodes, possibly splitting an initial compact graph into many subgraphs, refer to Figure 1.5. 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 1.6. The same is not true when having loopy graphs (see Figure 1.7), 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.

	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 1.5 Factors involved in the graph of Figure 1.8.

You don't have to handle the latter aspect: when a belief propagation is performed, the library automatically chooses to deploy 5.34 or 5.32, according to the structure of the graph for which the propagation is asked.

1.3 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 1.2. Referring to Figure 1.4, the message to A is computed as follows when performing a maximum a posteriori estimation:

$$M_{B\to A}(a) = \max_{\tilde{b}}\{\Phi_{AB}(a,\tilde{b})M_{V1\to B}(\tilde{b}) \cdot \cdots \cdot M_{Vm\to B}(\tilde{b})\} \tag{1.24}$$

Essentially, the summation in equation (1.22) 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:

$$h_{iMAP} = argmax\{\Phi_{Hi}(h_{iMAP})M_1(h_{iMAP}) \cdot \cdots \cdot M_L(h_{iMAP})\} \tag{1.25}$$

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:

$$h_{MAP} \neq \{argmax(\mathbb{P}(h_1)), \cdots, argmax(\mathbb{P}(h_n))\} \tag{1.26}$$

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 1.8, with the potentials Φ_{XA}, Φ_{AB} and Φ_{YB} having the images defined in table 1.5. 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 1.8. Clearly individual marginals for A and B would be equal to:

$$\begin{aligned}\mathbb{P}(A) &= \begin{pmatrix}\mathbb{P}(A=0)\\ \mathbb{P}(A=1)\end{pmatrix} = \begin{pmatrix}0.5\\ 0.5\end{pmatrix}\\ \mathbb{P}(B) &= \begin{pmatrix}\mathbb{P}(B=0)\\ \mathbb{P}(B=1)\end{pmatrix} = \begin{pmatrix}0.5\\ 0.5\end{pmatrix}\end{aligned} \tag{1.27}$$

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 1.6. 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 5.37.2.6 on a particular derivation of class 5.37. Maximum a posteriori estimation for sub graphs (see Section 1.5) is also supported by method MAP 5.3.2.3.

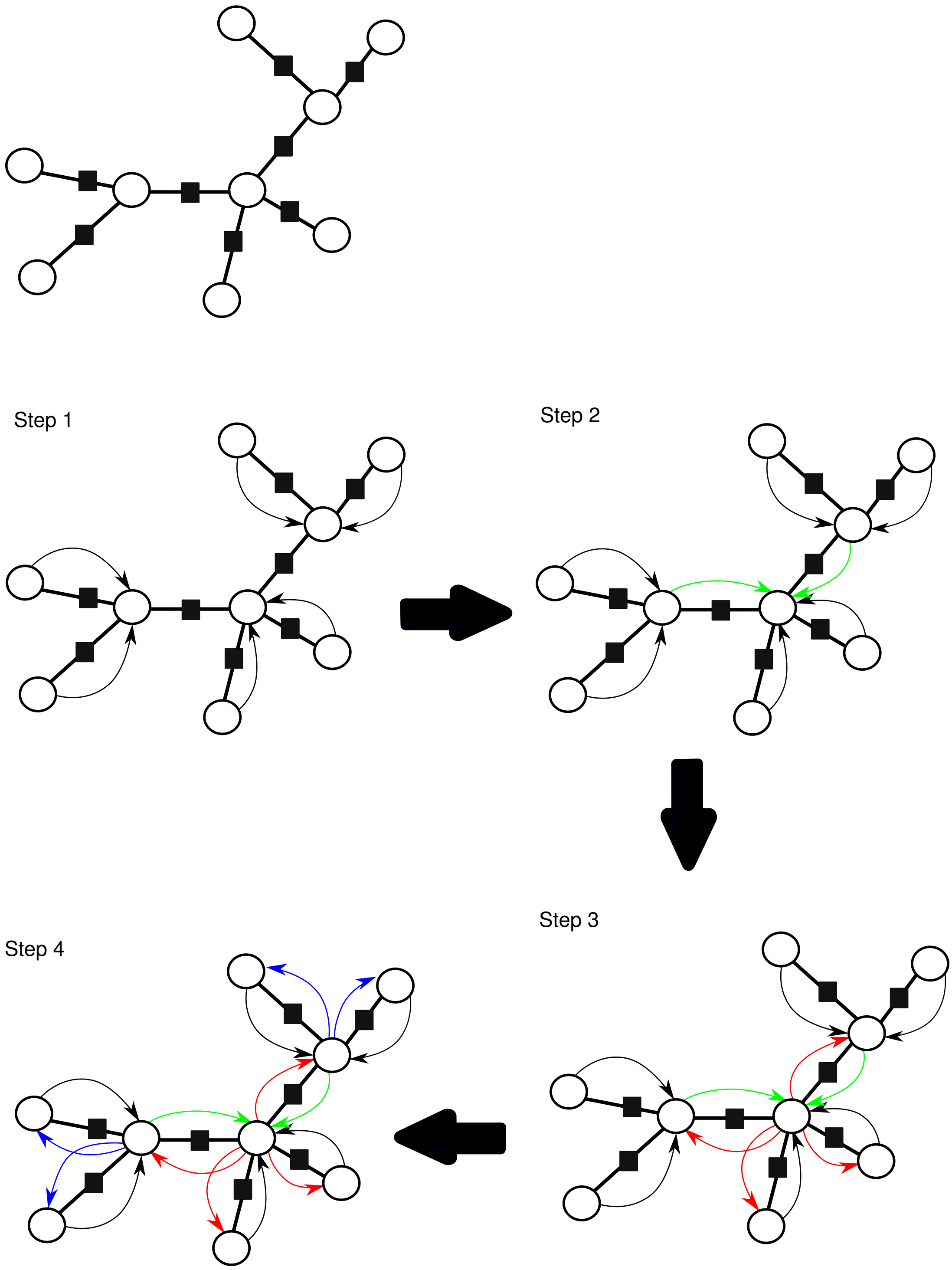


Figure 1.6 Steps involved for computing the messages of the politree represented at the top. The leaves are the first nodes for which the outgoing messages can be computed.

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

Table 1.6 Factors involved in the graph of Figure 1.8.

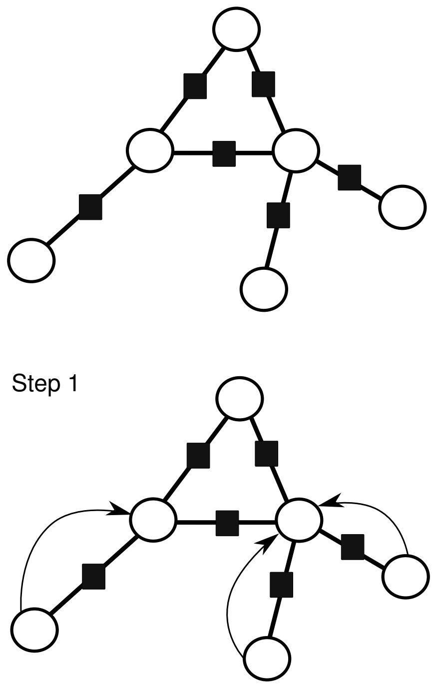


Figure 1.7 Steps involved for computing the messages on a loopy graph: after computing the messages outgoing from the leaves, a deadlock is reached since no further messages are computable.

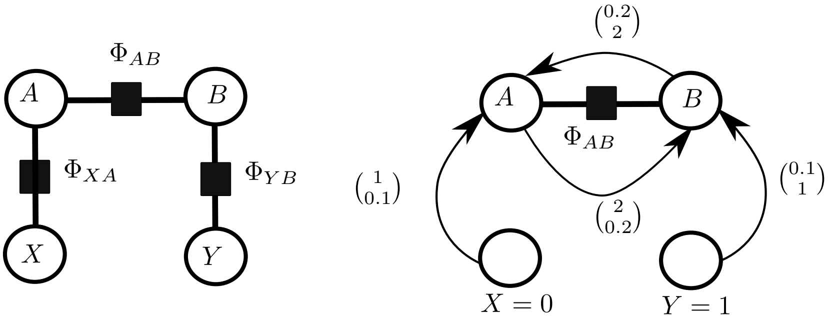


Figure 1.8 Example of graph adopted. When the evidences are retrieved, the messages computed by making use of the message passing algorithm are reported below.

1.4 Gibbs sampling

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

$$x^{j+1}_k \sim \mathbb{P}(x_k | x^j_{\{1,\cdots,m\} \setminus k}) \tag{1.28}$$

After an initial transient, the samples cumulated during the iterations can be considered as drawn from the joint distribution involving group $X_{1,\cdots,m}$.
This algorithm can be easily applied to graphical model. Indeed the methodologies exposed in Section 1.2 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 5.37.2.5 is in charge of performing Gibbs sampling on a generic graph, while method Gibbs_Sampling 5.3.2.2 performs the same for sub graphs (see 1.5).

1.5 Sub graphs

As explained in Section 1.2, 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 1.9 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 1.9 is the one modelling $\mathbb{P}(A, B, C)$. Actually, that sub graph is a graphical model itself, for which all the properties exposed so far hold. For example the energy function E is computable as follows:

$$E(A = a, B = b, C = c) = \Phi_{AB}(a, b)\Phi_{BC}(b, c)\Phi_{AC}(a, c)M_{X\rightarrow A}(a)M_{Y\rightarrow B}(b) \tag{1.29}$$

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

$$\mathbb{P}(A = a, B = b, C = c) = \frac{E(a, b, c)}{\sum_{\forall \tilde{a},\tilde{b},\mathbf{c}} E(\tilde{a}, \tilde{b}, \tilde{c})} \tag{1.30}$$

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 1.10 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 5.3.

1.6 Learning

The aim of learning is to determine the optimal values for the w (equation (1.11)) of all the tunable potentials (see Section 1.1) Ψ. To this aim two cases must be distinguished:

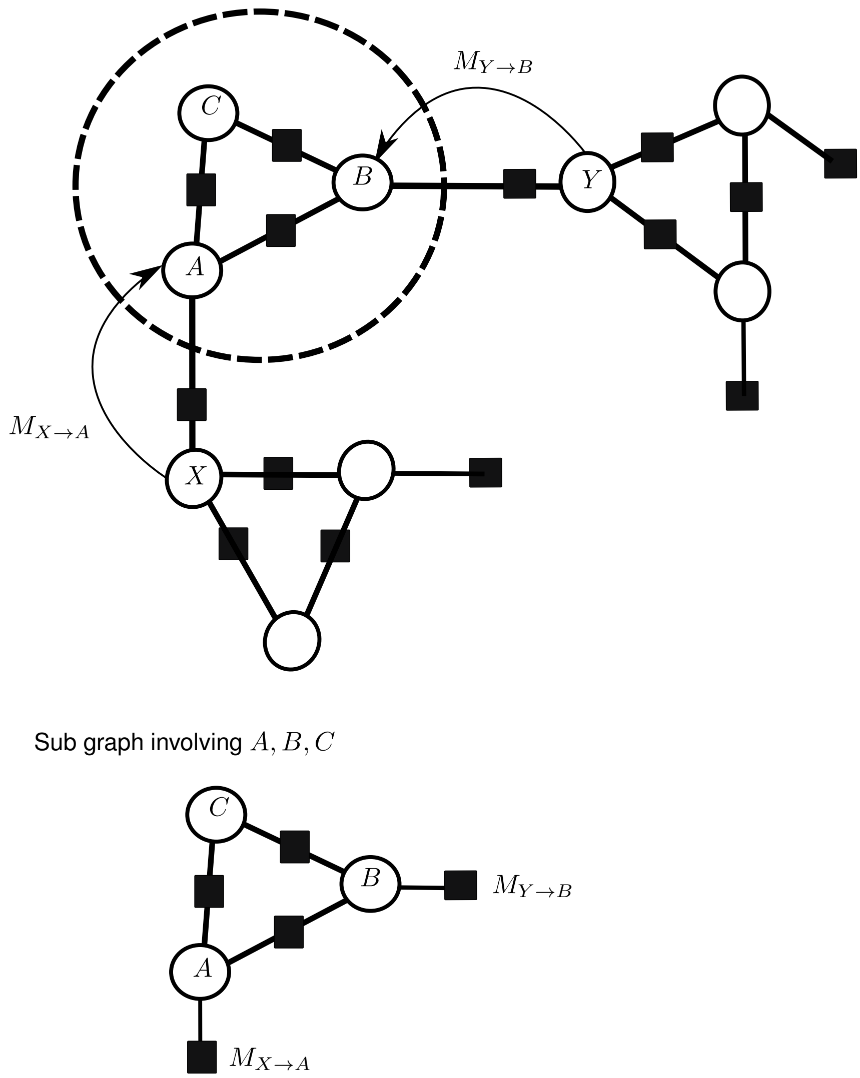


Figure 1.9 Example of graph reduction.

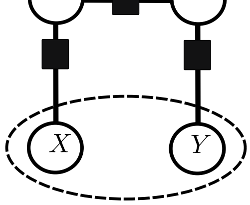


Figure 1.10 Example of a subset of variables for which the graph reduction is not possible.

- Learning must be performed for a Graph 5.20 or a Random_Field 5.41: see Section 1.6.1
- Learning must be performed for a Conditional_Random_Field 5.13: see Section 1.6.2

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

$$W = \{w_1, \cdots, w_D\} \tag{1.31}$$

w_i will refer to the i^{th} free parameter of the model. Learning can be performed using class METTERE.

1.6.1 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 variable in $\mathcal{V}$, no matter the fact that they are involved in tunable or non tunable potentials. As exposed in Section 1.1, if W is known, the probability of a combination t_j can be evaluated as follows:

$$\mathbb{P}(t_j) = \frac{E(t_j, W)}{\mathcal{Z}(W)} \tag{1.32}$$

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 1.1):

$$\begin{aligned} E(t_j, W) &= exp\big(w_1\Phi_1(t_j)\big) \cdot \cdots \cdot exp\big(w_D\Phi_D(t_j)\big) \cdot E_0(t_j) \\ &= exp\big(\sum_{i=1}^{D} w_i\Phi_i(t_j)\big) \cdot E_0(t_j) \end{aligned} \tag{1.33}$$

The likelihood function L can be defined as follows:

$$L = \prod_{t_j \in T} \mathbb{P}(t_j) \tag{1.34}$$

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

$$\begin{aligned} J = \frac{log(L)}{N} &= \sum_{t_j \in T} \frac{log(\mathbb{P}(t_j))}{N} \\ &= \sum_{t_j \in T} \frac{log(E(t_j, W)) - log(\mathcal{Z}(W)}{N} \\ &= \frac{1}{N}\sum_{t_j \in T} log(E(t_j, W)) - log(\mathcal{Z}(W)) \\ &= \frac{1}{N}\sum_{t_j \in T} \big(\sum_{i=1}^{D} w_i\Phi_i(t_j)\big) - log(\mathcal{Z}(W)) - \frac{1}{N}\sum_{t_j \in T} log(E_0(t_j)) \end{aligned} \tag{1.35}$$

The aim of learning is to find the value of W maximising J. This is done iteratively, exploiting a gradient descend approach (see METTERE possibili strategie). 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 (1.35), 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 (1.35) can be rewritten as follows:

$$\begin{aligned} J &= \alpha(T, W) - \beta(W) \\ \alpha &= \frac{1}{N}\sum_{t_j \in T} \big(\sum_{i=1}^{D} w_i\Phi_i(t_j)\big) \qquad (1.36) \\ \beta &= log(\mathcal{Z}(W)) \qquad (1.37) \end{aligned}$$

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

1.6.1.1 Gradient of α

By the analysis of the equation (1.36) it is clear that:

$$\frac{\partial \alpha}{\partial w_i} = \frac{1}{N} \sum_{t_j \in T} w_i \Phi_i(t_j) \tag{1.38}$$

1.6.1.2 Gradient of β

The computation of $\frac{\partial \beta}{\partial w_i}$ requires to manipulate a little bit equation (1.37). Firstly the derivative of the logarithm must be computed:

$$\frac{\partial \beta}{\partial w_i} = \frac{1}{\mathcal{Z}} \frac{\partial \mathcal{Z}}{\partial w_i} \tag{1.39}$$

The normalizing coefficient $\mathcal{Z}$ is made of the following terms (see also equation (1.6)):

$$\mathcal{Z}(W) = \sum_{\tilde{V} \in \bigcup_{i=1}^{p} V_i} \left(exp(\sum_{i=1}^{D} w_i \Phi_i(\tilde{V})) \cdot E_0(\tilde{V}) \right) \tag{1.40}$$

Introducing equation (1.40) into (1.39) leads to:

$$\begin{aligned}
\frac{\partial \beta}{\partial w_i} &= \frac{1}{\mathcal{Z}} \frac{\partial}{\partial w_i} \left(\sum_{\tilde{V}} exp(\sum_{i=1}^{D} w_i \Phi_i(\tilde{V})) E_0(\tilde{V}) \right) \\
&= \frac{1}{\mathcal{Z}} \sum_{\tilde{V}} \frac{\partial}{\partial w_i} \left(exp(\sum_{i=1}^{D} w_i \Phi_i(\tilde{V})) \right) E_0(\tilde{V}) \\
&= \frac{1}{\mathcal{Z}} \sum_{\tilde{V}} exp(\sum_{i=1}^{D} w_i \Phi_i(\tilde{V})) E_0(\tilde{V}) \Phi_i(\tilde{V}) \\
&= \sum_{\tilde{V}} \frac{exp(\sum_{i=1}^{D} w_i \Phi_i(\tilde{V})) E_0(\tilde{V})}{\mathcal{Z}} \Phi_i(\tilde{V}) \\
&= \sum_{\tilde{V}} \frac{E(\tilde{V})}{\mathcal{Z}} \Phi_i(\tilde{V}) \\
&= \sum_{\tilde{V}} \mathbb{P}(\tilde{V}) \Phi_i(\tilde{V})
\end{aligned} \tag{1.41}$$

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

$$\begin{aligned}
\frac{\partial \beta}{\partial w_i} &= \sum_{\tilde{V}} \mathbb{P}(\tilde{V}) \Phi_i(\tilde{V}) \\
&= \sum_{\tilde{V}_{1,2}} \Phi_i(\tilde{V}_{1,2}) \sum_{\tilde{V}_{3,4,\dots}} \mathbb{P}(\tilde{V}_{1,2,3,4,\dots}) \\
&= \sum_{\tilde{V}_{1,2}} \Phi_i(\tilde{V}_{1,2}) \mathbb{P}(\tilde{V}_{1,2})
\end{aligned} \tag{1.42}$$

where $\mathbb{P}(\tilde{V}_{1,2})$ is the marginal probability (see the initial part of Section 1.1) 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 1.5). 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:

$$\frac{\partial \beta}{\partial w_i} = \sum_{\forall \tilde{V}_1} \Phi_i(\tilde{V}_1) \mathbb{P}(\tilde{V}_1) \tag{1.43}$$

which can be easily obtained through the message passing algorithm (Section 1.2).

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

$$\frac{\partial J}{\partial w_i} = \frac{1}{N}\sum_{j=1}^{N}\Phi_i(D_j^i) - \sum_{\tilde{D}^i}\mathbb{P}(\tilde{D}^i)\Phi_i(\tilde{D}^i) \tag{1.44}$$

1.6.2 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 1.1, 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 $\mathcal{O}$:

$$\begin{aligned} T &= \{t_1, \cdots, t_N\} \\ &= \{\{h_1, o_1\}, \cdots, \{h_N, o_N\}\} \end{aligned} \tag{1.45}$$

We recall , equation (1.13), that the conditional probability of the hidden variables w.r.t. the observed ones is defined as follows:

$$\begin{aligned} \mathbb{P}(h_j, o_j) &= \frac{E(h_j, o_j, W)}{\mathcal{Z}(o_j, W)} \\ E(h_j, o_j, W) &= exp(\sum_{i=1}^{D} w_i\Phi_i(h_j, o_j))E_0(h_j, o_j) \\ \mathcal{Z}(o_j, W) &= \sum_{\tilde{h}} E(\tilde{h}, o_j, W) \end{aligned} \tag{1.46}$$

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

$$L = \prod_{h_j \in T}\mathbb{P}(h_j|o_j) \tag{1.47}$$

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

$$\begin{aligned} J &= \frac{log(L)}{N} \\ &= \frac{1}{N}\sum_{h_j, o_j \in T} log(E(h_j, o_j, W)) - \frac{1}{N}\sum_{h_j, o_j \in T} log(Z(o_j, W)) \\ &= \frac{1}{N}\sum_{h_j, o_j \in T}(\sum_{i=1}^{D} w_i\Phi_i(h_j, o_j)) - \frac{1}{N}\sum_{h_j, o_j \in T} log(Z(o_j, W)) + \frac{1}{N}\sum_{h_j, o_j \in T} log(E_0(h_j, o_j)) \end{aligned} \tag{1.48}$$

Neglecting the term not depending upon W and T, Equations 1.48 can be rewritten as follows:

$$\begin{aligned} J &= \alpha(T, W) - \beta(T, W) \\ \alpha(T, W) &= \frac{1}{N}\sum_{h_j, o_j}(\sum_{i=1}^{D} w_i\Phi_i(h_j, o_j)) \\ \beta(T, W) &= \frac{1}{N}\sum_{o_j} log(\mathcal{Z}(o_j, W)) \end{aligned} \tag{1.49}$$

At this point, an important remark must be done: differently from the β defined in equation (1.37), $\beta(T, W)$ 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 (1.44). 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 [4], 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 (1.38).

[4] that can be highly correlated

1.6.2.1 Gradient of β

Following the same approach in Section 1.6.1.2, the gradient of β can be computed as follows:

$$
\begin{aligned}
\frac{\partial \beta}{\partial w_i} &= \frac{1}{N}\sum_{j=1}^{N} \frac{\partial log(\mathcal{Z}(o_j, W))}{\partial w_i} \\
&= \frac{1}{N}\sum_{j=1}^{N} \frac{1}{\mathcal{Z}(o_j)} \frac{\partial \mathcal{Z}(o_j, W)}{\partial w_i} \\
&= \frac{1}{N}\sum_{j=1}^{N} \frac{\partial}{\partial w_i}\Big(\sum_{\tilde{h}} exp(\sum_{i=1}^{D} w_i \Phi_i(\tilde{h}, o_j)) E_0(\tilde{h}, o_j) \Big) \\
&= \frac{1}{N}\sum_{j=1}^{N} \sum_{\tilde{h}} \Big(exp(\sum_{i=1}^{D} w_i \Phi_j(\tilde{h}, o_j)) E_0(\tilde{h}, o_j) \Phi_i(\tilde{h}, o_j) \Big) \\
&= \frac{1}{N}\sum_{j=1}^{N} \sum_{\tilde{h}} \frac{E(\tilde{h}, o_j, W)}{\mathcal{Z}(o_1)} \Phi_i(\tilde{h}, o_j) \\
&= \frac{1}{N}\sum_{j=1}^{N} \sum_{\tilde{h}} \mathbb{P}(\tilde{h}|o_j) \Phi_i(\tilde{h}, o_j)
\end{aligned} \tag{1.50}
$$

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

$$
\begin{aligned}
\frac{\partial \beta}{\partial w_i} &= \frac{1}{N}\sum_{j=1}^{N} \sum_{\tilde{h}} \mathbb{P}(\tilde{h}|o_j) \Phi_i(\tilde{h}, o_j) \\
&= \frac{1}{N}\sum_{j=1}^{N} \sum_{\tilde{h}_{1,2}} \Phi_i(\tilde{h}_{1,2}, o_j) \sum_{\tilde{h}_{3,4,\cdots}} \mathbb{P}(\tilde{h}_{1,2,3,4,\cdots}|o_j) \\
&= \frac{1}{N}\sum_{j=1}^{N} \sum_{\tilde{h}_{1,2}} \Phi_i(\tilde{h}_{1,2}, o_j) \mathbb{P}(\tilde{h}_{1,2}|o_j)
\end{aligned} \tag{1.51}
$$

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:

$$
\frac{\partial J}{\partial w_i} = \frac{1}{N}\sum_{j=1}^{N} \Phi_i(h_j, o_j) - \frac{1}{N}\sum_{j=1}^{N} \Big(\sum_{\tilde{h}_{1,2}} \mathbb{P}(\tilde{h}_{1,2}|o_j) \Phi_i(\tilde{h}_{1,2}) \Big) \tag{1.52}
$$

where in the above equation, without loss of generality, we have assumed variables $\tilde{h}_{1,2}$ as the ones involved by potential Φ_j.

1.6.3 Learning of modular structure

Suppose to have a modular structure made of repeating units as for example the graph in Figure 1.11. 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 1.11). 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.

1.6.3.1 Gradient of α

Considering the model in Figure 1.11, the α part of J (equation (1.36)) can be computed as follows:

$$\alpha = \frac{1}{N}\sum_{t_j}\big(w_1\Phi_1(a_{1j},b_{1j}) + w_1\Phi_2(a_{2j},b_{2j}) + w_1\Phi_3(a_{3j},b_{3j}) + \cdots + \\ \cdots + \sum_{i=2}^{D} w_i\Phi_i(t_j)\big) \tag{1.53}$$

which leads to:

$$\frac{\partial\alpha}{\partial w_1} = \frac{1}{N}\sum_{t_j}\big(\Phi_1(a_{1j},b_{1j}) + \Phi_2(a_{2j},b_{2j}) + \Phi_3(a_{3j},b_{3j})\big) \tag{1.54}$$

1.6.3.2 Gradient of β

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

$$\begin{aligned}
\frac{\partial\beta}{\partial w_1} &= \frac{1}{Z}\frac{\partial Z}{\partial w_1} \\
&= \frac{1}{Z}\frac{\partial}{\partial w_1}\Big(\sum_{\tilde{V}}\Big(exp\big(w_1(\Psi_1(a_{1j},b_{1j}) + \cdots \\
\cdots &+ \Psi_2(a_{2j},b_{2j}) + \Psi_3(a_{3j},b_{3j})) + \sum_{i=2}^{D} w_i\Phi_i(\tilde{V})\big)E_0(\tilde{V}))\Big)\Big) \\
&= \sum_{\tilde{V}}\mathbb{P}(\tilde{V})(\Phi_1(\tilde{a}_1,\tilde{b}_1) + \Phi_2(\tilde{a}_2,\tilde{b}_2) + \Phi_3(\tilde{a}_3,\tilde{b}_3)) \\
&= \sum_{\tilde{V}}\mathbb{P}(\tilde{V})\Phi_1(\tilde{a}_1,\tilde{b}_1) + \sum_{\tilde{V}}\mathbb{P}(\tilde{V})\Phi_2(\tilde{a}_2,\tilde{b}_2) + \sum_{\tilde{V}}\mathbb{P}(\tilde{V})\Phi_3(\tilde{a}_3,\tilde{b}_3) \\
&= \sum_{\tilde{A}_1,\tilde{B}_1}\mathbb{P}(\tilde{A}_1,\tilde{B}_1)\Phi_1(\tilde{A}_1,\tilde{B}_1) + \sum_{\tilde{A}_2,\tilde{B}_2}\mathbb{P}(\tilde{A}_2,\tilde{B}_2)\Phi_2(\tilde{A}_2,\tilde{B}_2) + \cdots \\
\cdots &+ \sum_{\tilde{A}_3,\tilde{B}_3}\mathbb{P}(\tilde{A}_3,\tilde{B}_3)\Phi_3(\tilde{A}_3,\tilde{B}_3)
\end{aligned} \tag{1.55}$$

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 (1.44)):

$$\begin{aligned}
\frac{\partial J}{\partial w_1} &= \frac{1}{N}\sum_{j=1}^{N}\Phi_1(a_i^1,b_i^1) - \sum_{\tilde{a}^1,\tilde{b}^1}\mathbb{P}(\tilde{a}^1,\tilde{b}^1)\Phi_1(\tilde{a}^1,\tilde{b}^1) + \cdots \\
&+ \frac{1}{N}\sum_{j=1}^{N}\Phi_2(a_i^2,b_i^2) - \sum_{\tilde{a}^2,\tilde{b}^2}\mathbb{P}(\tilde{a}^2,\tilde{b}^2)\Phi_2(\tilde{a}^2,\tilde{b}^2) + \cdots \\
&+ \frac{1}{N}\sum_{j=1}^{N}\Phi_3(a_i^3,b_i^3) - \sum_{\tilde{a}^3,\tilde{b}^3}\mathbb{P}(\tilde{a}^3,\tilde{b}^3)\Phi_3(\tilde{a}^3,\tilde{b}^3) +
\end{aligned} \tag{1.56}$$

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

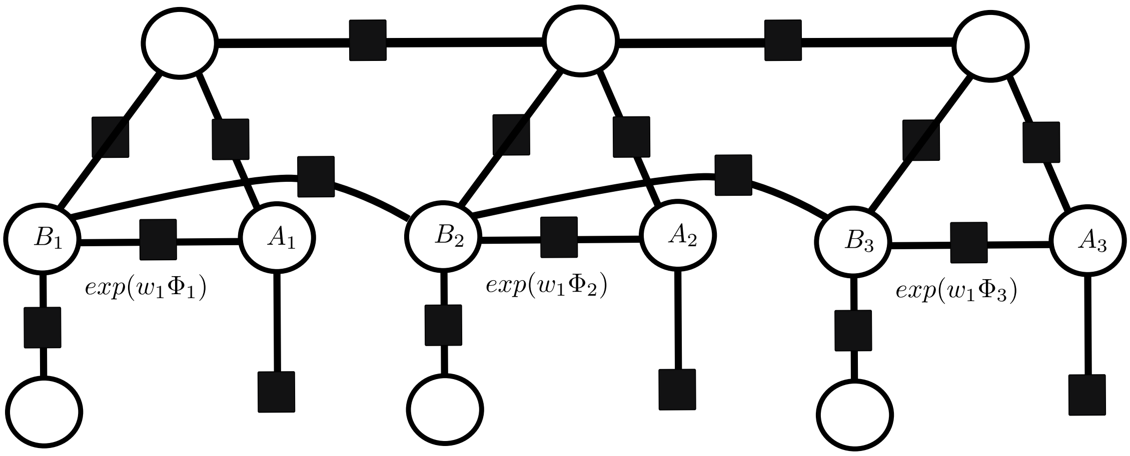


Figure 1.11 Example of modular structure: weight w_1 is simultaneously involved into potentials Φ_1, Φ_2 and Φ_3.

Chapter 2

XML notation

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 5.20.2.2, Random_Field 5.41.2.2 and Conditional_Random_Field 5.13.2.1. Figure 2.1 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 1.1.
  - 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 1.1) 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 1.1). 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 1.1) it must be present for indicating the value of the weight w (equation (1.11)). 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 1.1). 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 5.40.2.3), otherwise when passing 'F' a simple anti correlating shape is assumed (see 5.40.2.3). 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.

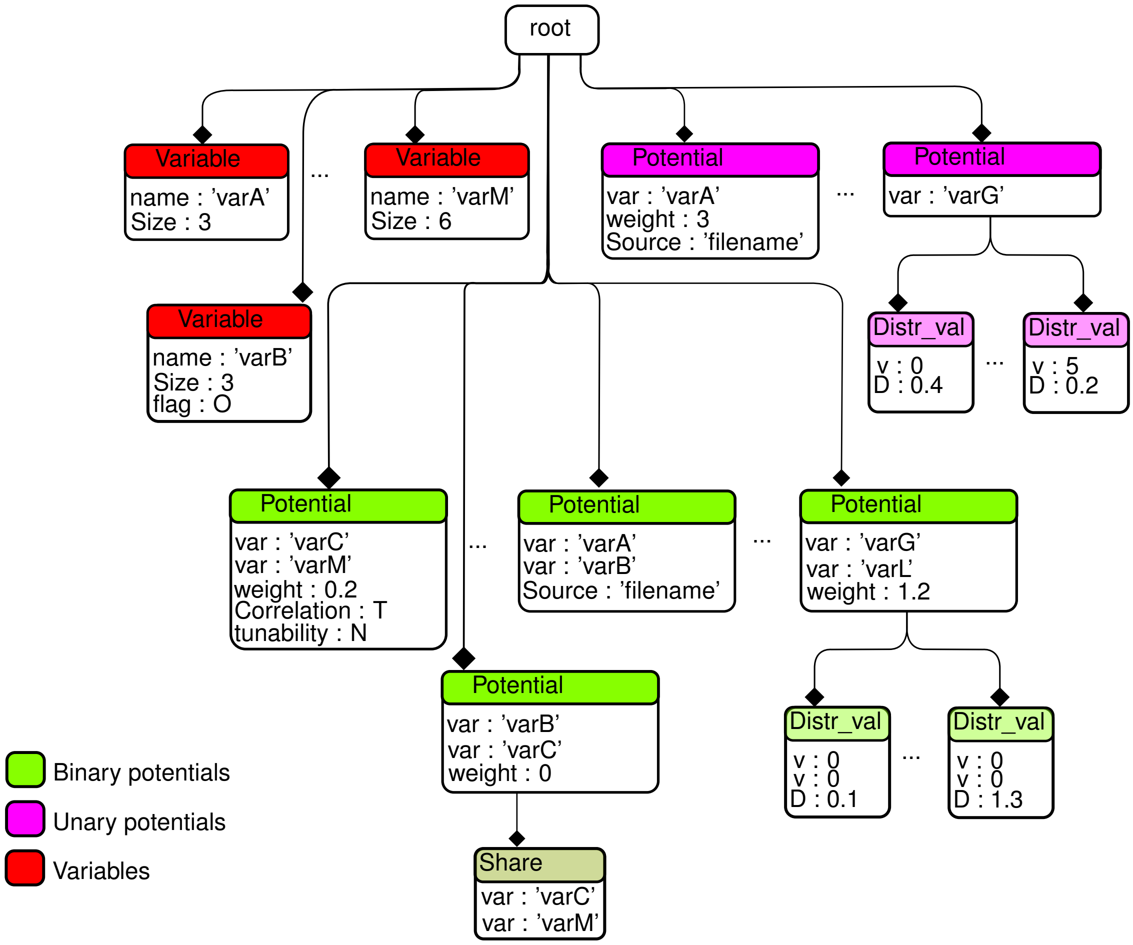


Figure 2.1 The structure of the XML describing a graphical model.

– 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 1.1 would have been described by a file containing the following rows:

$$
\begin{array}{ccc}
0 & 0 & 1 \\
0 & 1 & 4 \\
1 & 1 & 1 \\
2 & 2 & 5 \\
2 & 4 & 1
\end{array}
$$

– 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 1.1 would have been described by the syntax reported in Figure 2.2. In case weight was specified, an Exponential shape is built, wrapping the Simple shape whose distribution is specified by the aforementioned sub tags.

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

Figure 2.2 Syntax to adopt for describing the potential Φ_b of Section 1.1, using a population of Distr_val sub tags.

Chapter 3

Hierarchical Index

3.1 Class Hierarchy

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

Chapter 4

Class Index

4.1 Class List

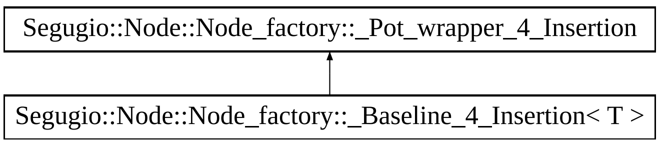
Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 5

Class Documentation

5.1 Segugio::Node::Node_factory::_Baseline_4_Insertion< T > Struct Template Reference

Inheritance diagram for Segugio::Node::Node_factory::_Baseline_4_Insertion< T >:



Public Member Functions

- **_Baseline_4_Insertion** (T *wrp)
- virtual const std::list< Categoric_var * > * **Get_involved_var_safe** ()
- virtual Potential * **Get_Potential_to_Insert** (const std::list< Categoric_var * > &var_involved, const bool &get_cloned)

Protected Attributes

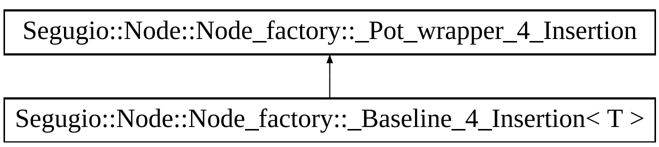
- T * **wrapped**

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

- C:/Users/andre/Desktop/CRF/CRF/Header/Node.h

5.2 Segugio::Node::Node_factory::_Pot_wrapper_4_Insertion Struct Reference

Inheritance diagram for Segugio::Node::Node_factory::_Pot_wrapper_4_Insertion:



Public Member Functions

- virtual const std::list< Categoric_var ∗ > ∗ **Get_involved_var_safe** ()=0
- virtual Potential ∗ **Get_Potential_to_Insert** (const std::list< Categoric_var ∗ > &var_involved, const bool &get_cloned)=0

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

- C:/Users/andre/Desktop/CRF/CRF/Header/Node.h

5.3 Segugio::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.
- 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_All_variables (std::list< Categoric_var ∗ > ∗result)

  Returns the cluster of varaibles involved in this sub graph.

5.3.1 Constructor & Destructor Documentation

5.3.1.1 _SubGraph()

```
Segugio::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

5.3.2 Member Function Documentation

5.3.2.1 Get_marginal_prob_combinations()

```
void Segugio::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.

5.3.2.2 Gibbs_Sampling()

```
void Segugio::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 )  [inline]
```

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

5.3.2.3 MAP()

```
void Segugio::Node::Node_factory::_SubGraph::MAP (
            std::list< size_t > * result ) [inline]
```

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

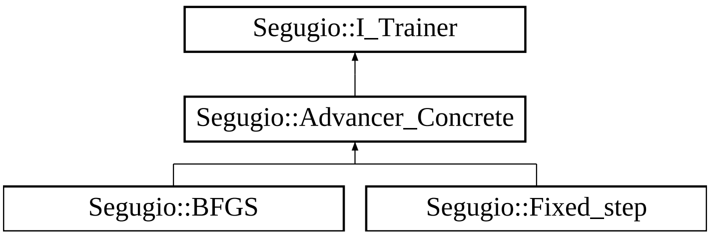
Values are ordered as returned by _SubGraph::Get_All_variables. This MAP is conditioned to the observations set at the time this subgraph was created.

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

- C:/Users/andre/Desktop/CRF/CRF/Header/Node.h
- C:/Users/andre/Desktop/CRF/CRF/Source/Subgraph.cpp

5.4 Segugio::Advancer_Concrete Class Reference

Inheritance diagram for Segugio::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

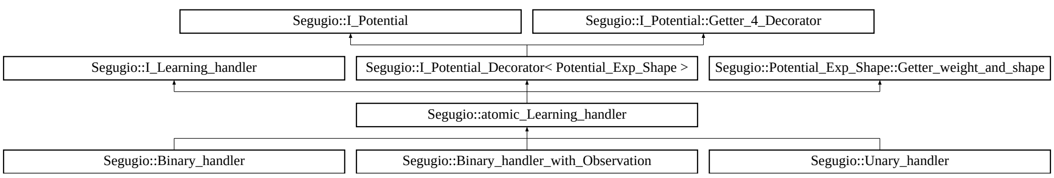
Additional Inherited Members

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

- C:/Users/andre/Desktop/CRF/CRF/Source/Trainer.cpp

5.5 Segugio::atomic_Learning_handler Class Reference

Inheritance diagram for Segugio::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)

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:

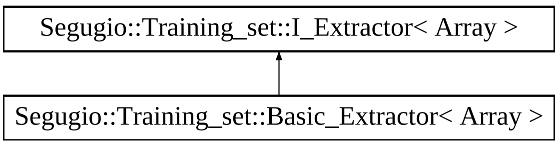
- C:/Users/andre/Desktop/CRF/CRF/Header/Graphical_model.h
- C:/Users/andre/Desktop/CRF/CRF/Source/Graphical_model.cpp

5.6 Segugio::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 Segugio::Training_set::Basic_Extractor< Array >:



Additional Inherited Members

5.6.1 Detailed Description

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

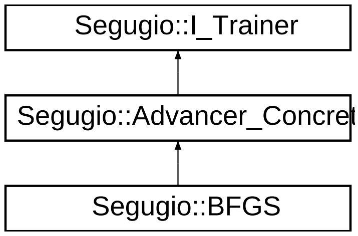
Basic extractor, see Training_set(const std::list<std::string>& variable_names, std::list<Array> samples, I_↩ Extractor<Array>∗ extractor)

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

- C:/Users/andre/Desktop/CRF/CRF/Header/Training_set.h

5.7 Segugio::BFGS Class Reference

Inheritance diagram for Segugio::BFGS:



Public Member Functions

- void **Reset** ()

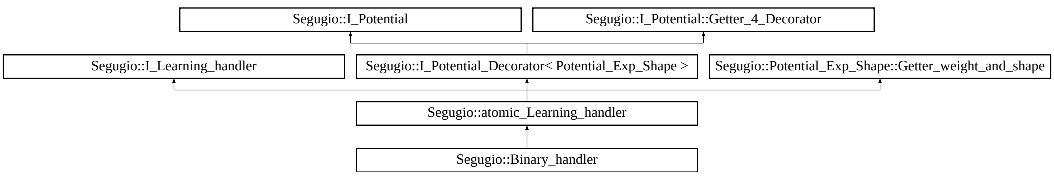
Additional Inherited Members

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

- C:/Users/andre/Desktop/CRF/CRF/Source/Trainer.cpp

5.8 Segugio::Binary_handler Class Reference

Inheritance diagram for Segugio::Binary_handler:



Public Member Functions

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

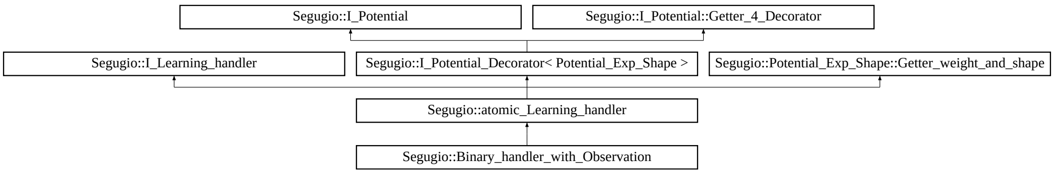
Additional Inherited Members

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

- C:/Users/andre/Desktop/CRF/CRF/Source/Graphical_model.cpp

5.9 Segugio::Binary_handler_with_Observation Class Reference

Inheritance diagram for Segugio::Binary_handler_with_Observation:



Public Member Functions

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

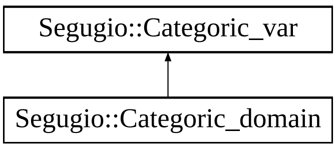
Additional Inherited Members

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

- C:/Users/andre/Desktop/CRF/CRF/Source/Graphical_model.cpp

5.10 Segugio::Categoric_domain Class Reference

Inheritance diagram for Segugio::Categoric_domain:



Public Member Functions

- const float & **operator[ ]** (const size_t &pos)

Additional Inherited Members

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

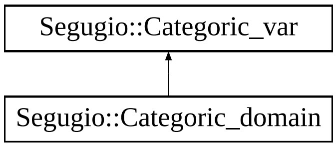
- C:/Users/andre/Desktop/CRF/CRF/Header/Potential.h
- C:/Users/andre/Desktop/CRF/CRF/Source/Potential.cpp

5.11 Segugio::Categoric_var Class Reference

Describes a categoric variable.

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

Inheritance diagram for Segugio::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}
- **Categoric_var** (const Categoric_var &to_copy)
- const size_t & **size** () const
- const std::string & **Get_name** ()

Protected Attributes

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

5.11.1 Detailed Description

Describes a categoric variable.

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

5.11.2 Constructor & Destructor Documentation

5.11.2.1 Categoric_var()

```
Segugio::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 ""

5.11.3 Member Data Documentation

5.11.3.1 Name

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

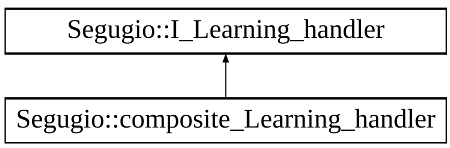
domain size

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

- C:/Users/andre/Desktop/CRF/CRF/Header/Potential.h
- C:/Users/andre/Desktop/CRF/CRF/Source/Potential.cpp

5.12 Segugio::composite_Learning_handler Class Reference

Inheritance diagram for Segugio::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:

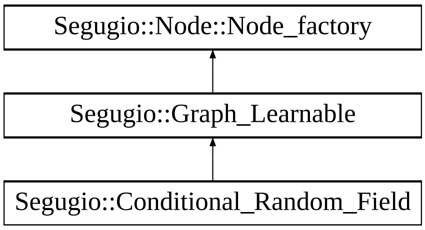
- C:/Users/andre/Desktop/CRF/CRF/Header/Graphical_model.h
- C:/Users/andre/Desktop/CRF/CRF/Source/Graphical_model.cpp

5.13 Segugio::Conditional_Random_Field Class Reference

This class describes Conditional Random fields.

```
#include <Graphical_model.h>
```

Inheritance diagram for Segugio::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_Observation_Set_val (const std::list< size_t > &new_observed_vals)
  see Node::Node_factory::Set_Observation_Set_val(const std::list<size_t>& new_observed_vals)

Additional Inherited Members

5.13.1 Detailed Description

This class describes Conditional Random fields.

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

5.13.2 Constructor & Destructor Documentation

5.13.2.1 Conditional_Random_Field() [1/2]

```
Segugio::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 2 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.

5.13.2.2 Conditional_Random_Field() [2/2]

```
Segugio::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

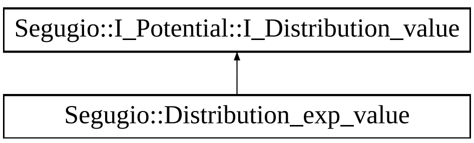
in	*potentials*	the initial set of exponential potentials to insert (can be empty)
in	*observed_var*	the set of variables to assume as observations
in	*use_cloning_Insert*	when is true, every time an Insert of a novel potential is called (this includes the passed potentials), a copy of that potential 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:/Users/andre/Desktop/CRF/CRF/Header/Graphical_model.h
- C:/Users/andre/Desktop/CRF/CRF/Source/Graphical_model.cpp

5.14 Segugio::Distribution_exp_value Struct Reference

Inheritance diagram for Segugio::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

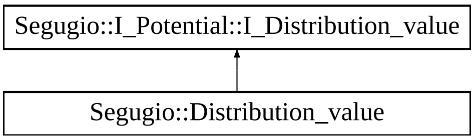
- float ∗ **w**
- Distribution_value ∗ **wrapped**

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

- C:/Users/andre/Desktop/CRF/CRF/Source/Potential.cpp

5.15 Segugio::Distribution_value Struct Reference

Inheritance diagram for Segugio::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

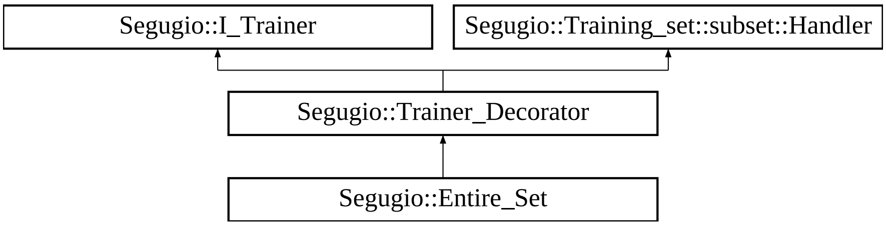
- struct **Distribution_exp_value**

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

- C:/Users/andre/Desktop/CRF/CRF/Source/Potential.cpp

5.16 Segugio::Entire_Set Class Reference

Inheritance diagram for Segugio::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)

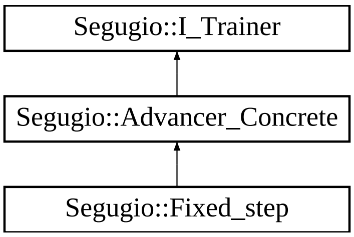
Additional Inherited Members

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

- C:/Users/andre/Desktop/CRF/CRF/Source/Trainer.cpp

5.17 Segugio::Fixed_step Class Reference

Inheritance diagram for Segugio::Fixed_step:



Public Member Functions

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

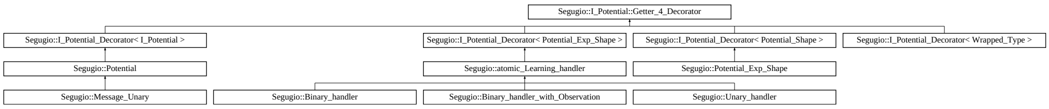
Additional Inherited Members

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

- C:/Users/andre/Desktop/CRF/CRF/Source/Trainer.cpp

5.18 Segugio::I_Potential::Getter_4_Decorator Struct Reference

Inheritance diagram for Segugio::I_Potential::Getter_4_Decorator:



Static Protected Member Functions

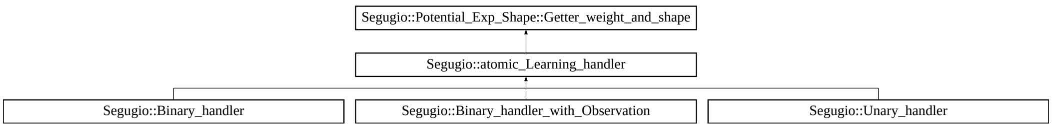
- static const std::list< Categoric_var * > * **Get_involved_var** (I_Potential *pot)
- static std::list< I_Distribution_value * > * **Get_distr** (I_Potential *pot)

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

- C:/Users/andre/Desktop/CRF/CRF/Header/Potential.h

5.19 Segugio::Potential_Exp_Shape::Getter_weight_and_shape Struct Reference

Inheritance diagram for Segugio::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:

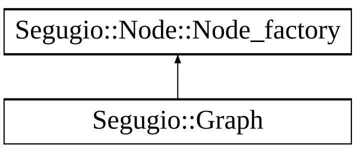
- C:/Users/andre/Desktop/CRF/CRF/Header/Potential.h

5.20 Segugio::Graph Class Reference

Interface for managing generic graphs.

```
#include <Graphical_model.h>
```

Inheritance diagram for Segugio::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_Observation_Set_var (const std::list< Categoric_var ∗ > &new_observed_vars)
  see Node::Node_factory::Set_Observation_Set_var(const std::list<Categoric_var>& new_observed_vars)*
- void Set_Observation_Set_val (const std::list< size_t > &new_observed_vals)
  see Node::Node_factory::Set_Observation_Set_val(const std::list<size_t>& new_observed_vals)

Additional Inherited Members

5.20.1 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.

5.20.2 Constructor & Destructor Documentation

5.20.2.1 Graph() [1/3]

```
Segugio::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.

5.20.2.2 Graph() [2/3]

```
Segugio::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 2 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.

5.20.2.3 Graph() [3/3]

```
Segugio::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.

5.20.3 Member Function Documentation

5.20.3.1 Insert() [1/2]

```
void Segugio::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).

5.20.3.2 Insert() [2/2]

```
void Segugio::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:

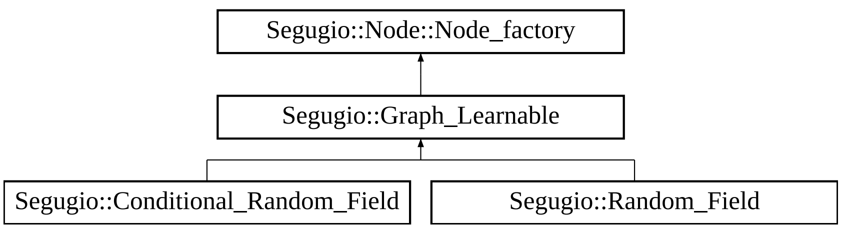
- C:/Users/andre/Desktop/CRF/CRF/Header/Graphical_model.h
- C:/Users/andre/Desktop/CRF/CRF/Source/Graphical_model.cpp

5.21 Segugio::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 Segugio::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 _Pot_wrapper_4_Insertion ∗ **Get_Inserter** (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)

Protected Attributes

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

5.21.1 Detailed Description

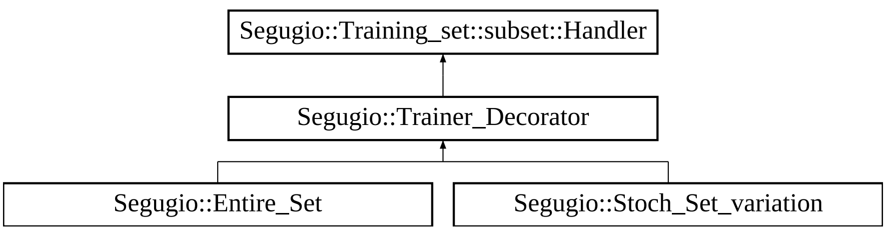
Interface for managing learnable graphs, i.e. graphs for which it is possible perform learning.

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

- C:/Users/andre/Desktop/CRF/CRF/Header/Graphical_model.h
- C:/Users/andre/Desktop/CRF/CRF/Source/Graphical_model.cpp

5.22 Segugio::Training_set::subset::Handler Struct Reference

Inheritance diagram for Segugio::Training_set::subset::Handler:



Static Protected Member Functions

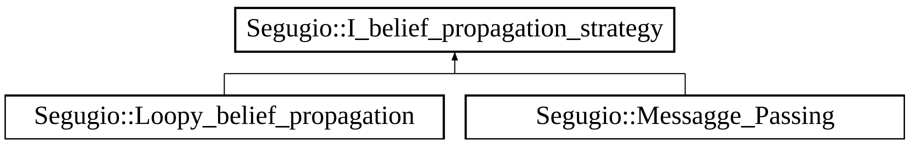
- static std::list< size_t * > * **Get_list** (subset *sub_set)
- static std::list< std::string > * **Get_names** (subset *sub_set)
- static std::list< std::string > * **Get_names** (Training_set *set)

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

- C:/Users/andre/Desktop/CRF/CRF/Header/Training_set.h

5.23 Segugio::I_belief_propagation_strategy Class Reference

Inheritance diagram for Segugio::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:

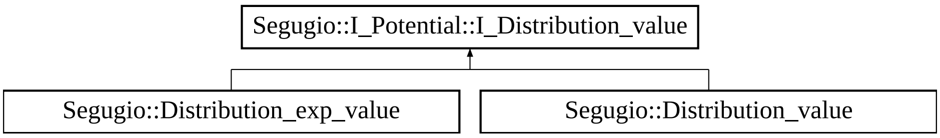
- C:/Users/andre/Desktop/CRF/CRF/Header/Node.h
- C:/Users/andre/Desktop/CRF/CRF/Source/Belief_propagation.cpp

5.24 Segugio::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 Segugio::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

5.24.1 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:

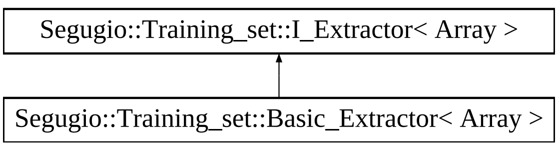
- C:/Users/andre/Desktop/CRF/CRF/Header/Potential.h

5.25 Segugio::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 Segugio::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

5.25.1 Detailed Description

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

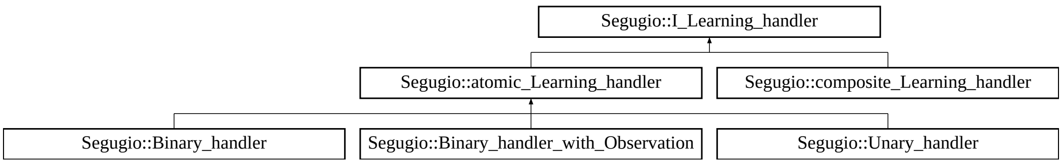
This class is adopted for parsing a set of samples to import as a novel training set. You have to derive yout custom extractor, implementing the two vritual method.

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

- C:/Users/andre/Desktop/CRF/CRF/Header/Training_set.h

5.26 Segugio::I_Learning_handler Class Reference

Inheritance diagram for Segugio::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:

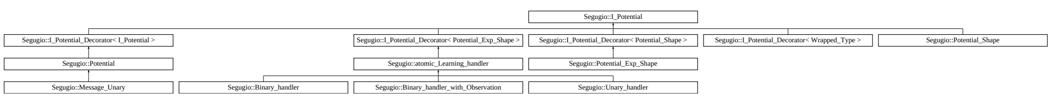
- C:/Users/andre/Desktop/CRF/CRF/Header/Graphical_model.h

5.27 Segugio::I_Potential Class Reference

Abstract interface for potentials handled by graphs.

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

Inheritance diagram for Segugio::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)

5.27.1 Detailed Description

Abstract interface for potentials handled by graphs.

5.27.2 Member Function Documentation

5.27.2.1 Find_Comb_in_distribution()

```
void Segugio::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

5.27.2.2 Get_entire_domain() [1/2]

```
void Segugio::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

5.27.2.3 Get_entire_domain() [2/2]

```
static void Segugio::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

5.27.2.4 Print_distribution()

```
void Segugio::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:

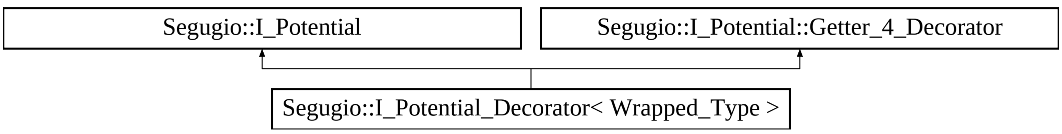
- C:/Users/andre/Desktop/CRF/CRF/Header/Potential.h
- C:/Users/andre/Desktop/CRF/CRF/Source/Potential.cpp

5.28 Segugio::I_Potential_Decorator< Wrapped_Type > Class Template Reference

Abstract decorator of a Potential, wrapping an Abstract potential.

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

Inheritance diagram for Segugio::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

5.28.1 Detailed Description

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

Abstract decorator of a Potential, wrapping an Abstract potential.

5.28.2 Member Data Documentation

5.28.2.1 pwrapped

```
template<typename Wrapped_Type>
Wrapped_Type* Segugio::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:

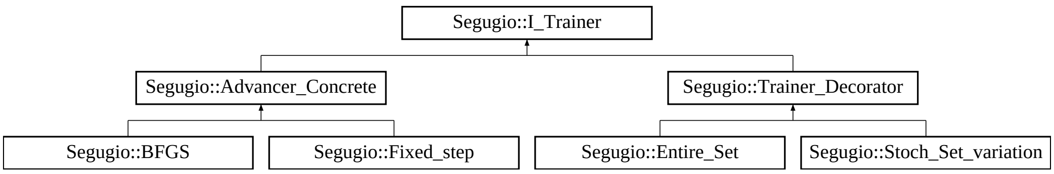
- C:/Users/andre/Desktop/CRF/CRF/Header/Potential.h

5.29 Segugio::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 Segugio::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.
- static I_Trainer ∗ Get_BFGS (const float &stoch_grad_percentage=1.f)

  Creates a BFGS gradient descend solver (`https://en.wikipedia.org/wiki/Broyden%E2%80%93↩ Fletcher%E2%80%93Goldfarb%E2%80%93Shanno_algorithm`*)*

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)

5.29.1 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.

5.29.2 Member Function Documentation

5.29.2.1 Get_BFGS()

```
I_Trainer * Segugio::I_Trainer::Get_BFGS (
            const float & stoch_grad_percentage = 1.f )  [static]
```

Creates a BFGS gradient descend solver (https://en.wikipedia.org/wiki/Broyden%E2%80%93Fletcher%E2%80%93Goldfarb%E2%80%93Shanno_algorithm)

Parameters

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

5.29.2.2 Get_fixed_step()

```
I_Trainer * Segugio::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:/Users/andre/Desktop/CRF/CRF/Header/Trainer.h
- C:/Users/andre/Desktop/CRF/CRF/Source/Trainer.cpp

5.30 Segugio::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:/Users/andre/Desktop/CRF/CRF/Source/Node.cpp

5.31 Segugio::info_neighbourhood Struct Reference

Classes

- struct info_neigh

Public Attributes

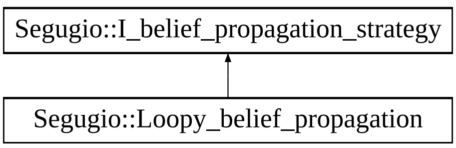
- size_t **Involved_var_pos**
- list< info_neigh > **Info**
- list< Potential ∗ > **Unary_potentials**

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

- C:/Users/andre/Desktop/CRF/CRF/Source/Node.cpp

5.32 Segugio::Loopy_belief_propagation Class Reference

Inheritance diagram for Segugio::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:

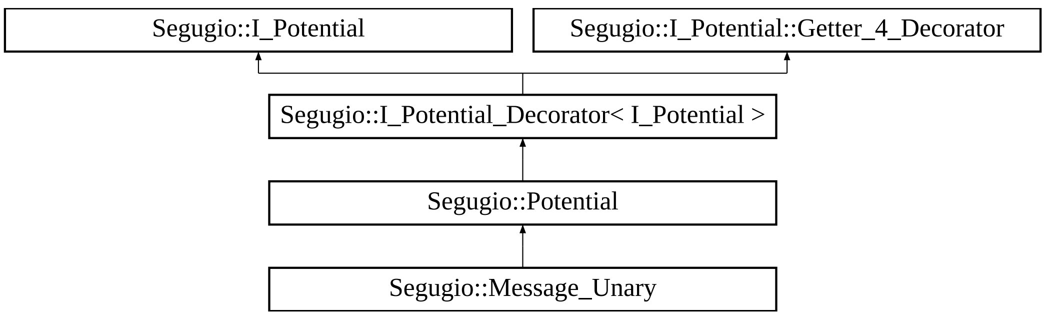
- C:/Users/andre/Desktop/CRF/CRF/Header/Belief_propagation.h
- C:/Users/andre/Desktop/CRF/CRF/Source/Belief_propagation.cpp

5.33 Segugio::Message_Unary Class Reference

This class is adopted by belief propagation algorithms. It is the message incoming to a node of the graph. Every node of a graph refers to a single Categorical variable. Internally it keeps track of the difference in time of the messages produced, in order to arrest loopy belief propagation.

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

Inheritance diagram for Segugio::Message_Unary:



Public Member Functions

- Message_Unary (Categoric_var *var_involved)

  Creates a Message with all 1 as values for the image.
- Message_Unary (Potential *binary_to_merge, const std::list< Potential * > &potential_to_merge, const bool &Sum_or_MAP=true)

  Firstly, all potential_to_merge are merged together using Potential::Potential(potential_to_merge, false) obtaining a merged potential. Secondly, the product of binary_to_merge and the merged potential is obtained. Finally the message is obtained by marginalizing from the second product, the variable of potential_to_merge, adopting a sum or a MAP. Exploited by message passing algorithms.
- Message_Unary (Potential *binary_to_merge, Categoric_var *var_to_marginalize, const bool &Sum_or_↩MAP=true)

  Same as Message_Unary::Message_Unary(Potential binary_to_merge, const std::list<Potential*>& potential_↩to_merge, const bool& Sum_or_MAP = true), but in the case potential_to_merge is empty.*
- void Update (float *diff_to_previous, Potential *binary_to_merge, const std::list< Potential * > &potential↩_to_merge, const bool &Sum_or_MAP=true)

  Adopted by loopy belief propagation.
- void Update (float *diff_to_previous, Potential *binary_to_merge, Categoric_var *var_to_marginalize, const bool &Sum_or_MAP=true)

  Adopted by loopy belief propagation.

Additional Inherited Members

5.33.1 Detailed Description

This class is adopted by belief propagation algorithms. It is the message incoming to a node of the graph. Every node of a graph refers to a single Categorical variable. Internally it keeps track of the difference in time of the messages produced, in order to arrest loopy belief propagation.

5.33.2 Constructor & Destructor Documentation

5.33.2.1 Message_Unary() [1/2]

```
Segugio::Message_Unary::Message_Unary (
            Categoric_var * var_involved )
```

Creates a Message with all 1 as values for the image.

Parameters

in	*var_involved*	the only variable in the domain

5.33.2.2 Message_Unary() [2/2]

```
Segugio::Message_Unary::Message_Unary (
            Potential * binary_to_merge,
            const std::list< Potential * > & potential_to_merge,
            const bool & Sum_or_MAP = true )
```

Firstly, all potential_to_merge are merged together using Potential::Potential(potential_to_merge, false) obtaining a merged potential. Secondly, the product of binary_to_merge and the merged potential is obtained. Finally the message is obtained by marginalizing from the second product, the variable of potential_to_merge, adopting a sum or a MAP. Exploited by message passing algorithms.

Parameters

in	*binary_to_merge*	binaty potential to consider
in	*potential_to_merge*	list of potentials to merge. The must be unary potentials

5.33.3 Member Function Documentation

5.33.3.1 Update() [1/2]

```
void Segugio::Message_Unary::Update (
            float * diff_to_previous,
            Potential * binary_to_merge,
            const std::list< Potential * > & potential_to_merge,
            const bool & Sum_or_MAP = true )
```

Adopted by loopy belief propagation.

Parameters

out	*diff_to_previous*	The difference with respect to the previous message camptation

5.33.3.2 Update() [2/2]

```
void Segugio::Message_Unary::Update (
            float * diff_to_previous,
            Potential * binary_to_merge,
            Categoric_var * var_to_marginalize,
            const bool & Sum_or_MAP = true )
```

Adopted by loopy belief propagation.

Parameters

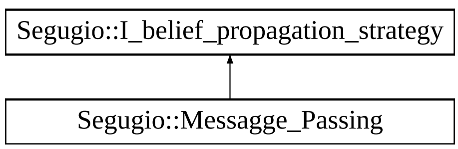
out	*diff_to_previous*	The difference with respect to the previous message camptation

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

- C:/Users/andre/Desktop/CRF/CRF/Header/Potential.h
- C:/Users/andre/Desktop/CRF/CRF/Source/Potential.cpp

5.34 Segugio::Messagge_Passing Class Reference

Inheritance diagram for Segugio::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:/Users/andre/Desktop/CRF/CRF/Header/Belief_propagation.h
- C:/Users/andre/Desktop/CRF/CRF/Source/Belief_propagation.cpp

5.35 Segugio::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:/Users/andre/Desktop/CRF/CRF/Header/Node.h
- C:/Users/andre/Desktop/CRF/CRF/Source/Node.cpp

5.36 Segugio::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:

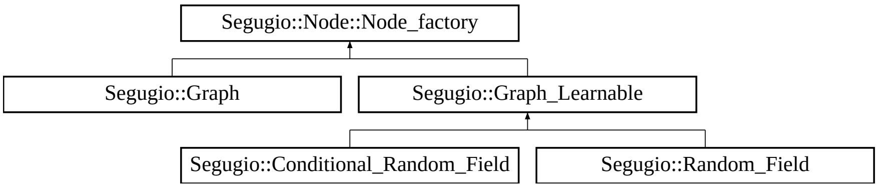
- C:/Users/andre/Desktop/CRF/CRF/Header/Node.h
- C:/Users/andre/Desktop/CRF/CRF/Source/Node.cpp

5.37 Segugio::Node::Node_factory Class Reference

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

```
#include <Node.h>
```

Inheritance diagram for Segugio::Node::Node_factory:



Classes

- struct _Baseline_4_Insertion
- struct _Pot_wrapper_4_Insertion
- 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.
- Categoric_var ∗ Find_Variable (Categoric_var ∗var_with_same_name)

  Returns a pointer to the variable in this graph with the same name of the variable passed as input.
- void Get_Actual_Hidden_Set (std::list< Categoric_var ∗ > ∗result)

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

  Returns the current set of observed variables.
- 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). .
- 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_Observation_Set_val (std::list< size_t > ∗result)

  Returns the attual values set observations. This function can be invokated after a call to void Set_Observation_Set_val(const std::list<size

- void Get_structure (std::list< const Potential ∗ > ∗structure)

  Returns the list of potentials constituting the net. Usefull for structural learning.

- 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)

Protected Member Functions

- **Node_factory** (const bool &use_cloning_Insert)
- void **Import_from_XML** (XML_reader ∗xml_data, const std::string &prefix_config_xml_file)
- void **Insert** (_Pot_wrapper_4_Insertion ∗element_to_add)
- void **Insert** (std::list< _Pot_wrapper_4_Insertion ∗ > &elements_to_add)
- virtual _Pot_wrapper_4_Insertion ∗ **Get_Inserter** (Potential_Exp_Shape ∗pot, const bool &weight_tunability)
- void **Insert** (Potential_Shape ∗pot)
- void **Insert** (Potential_Exp_Shape ∗pot, const bool &weight_tunability)
- Node ∗ **Find_Node** (const std::string &var_name)
- void Set_Observation_Set_var (const std::list< Categoric_var ∗ > &new_observed_vars)

  Set the values for the observations. Must call after calling Node_factory::Set_Observation_Set_val.

- void Set_Observation_Set_val (const std::list< size_t > &new_observed_vals)

  Set the observation set: which variables are treated like evidence when performing belief propagation.

- void **Belief_Propagation** (const bool &sum_or_MAP)
- size_t ∗ **Get_observed_val_in_case_is_in_observed_set** (Categoric_var ∗var)

5.37.1 Detailed Description

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

5.37.2 Member Function Documentation

5.37.2.1 Eval_Log_Energy_function()

```
void Segugio::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.

5.37.2.2 Find_Variable() [1/2]

```
Categoric_var * Segugio::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

5.37.2.3 Find_Variable() [2/2]

```
Categoric_var* Segugio::Node::Node_factory::Find_Variable (
            Categoric_var * var_with_same_name ) [inline]
```

Returns a pointer to the variable in this graph with the same name of the variable passed as input.

Returns NULL when the variable is not present in the graph

Parameters

in	*var_with_same_name*	variable having the same of name of the variable to search

5.37.2.4 Get_marginal_distribution()

```
void Segugio::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)

5.37.2.5 Gibbs_Sampling_on_Hidden_set()

```
void Segugio::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

5.37.2.6 MAP_on_Hidden_set()

```
void Segugio::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)

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

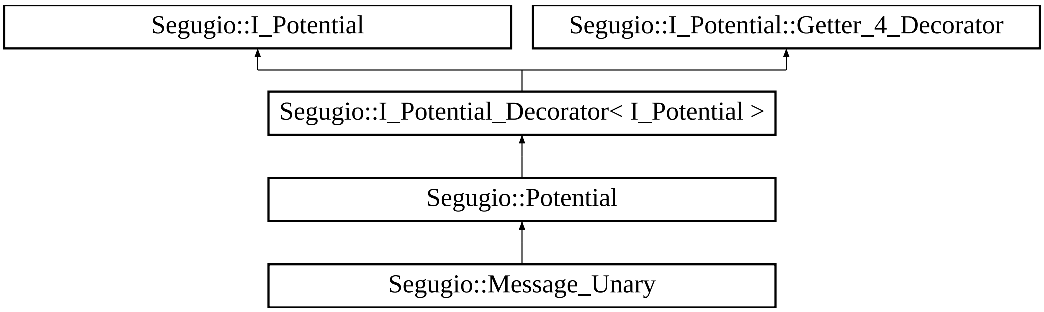
- C:/Users/andre/Desktop/CRF/CRF/Header/Node.h
- C:/Users/andre/Desktop/CRF/CRF/Source/Node.cpp

5.38 Segugio::Potential Class Reference

This class is mainly adopted for computing operations on potentials.

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

Inheritance diagram for Segugio::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.

Additional Inherited Members

5.38.1 Detailed Description

This class is mainly adopted for computing operations on potentials.

5.38.2 Constructor & Destructor Documentation

5.38.2.1 Potential() [1/4]

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

Parameters

in	*pot*	potential shape to wrap

5.38.2.2 Potential() [2/4]

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

Parameters

in	*pot*	exponential potential shape to wrap

5.38.2.3 Potential() [3/4]

```
Segugio::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

5.38.2.4 Potential() [4/4]

```
Segugio::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)

5.38.3 Member Function Documentation

5.38.3.1 Get_marginals()

```
void Segugio::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:

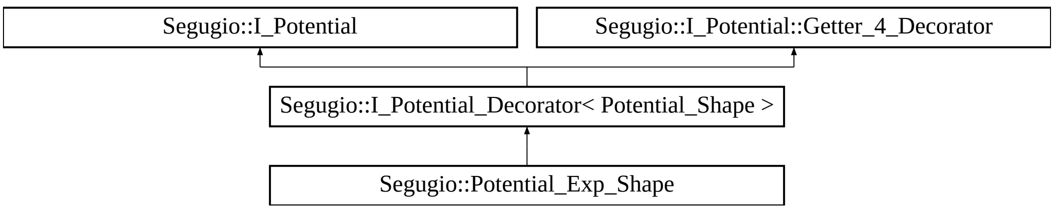
- C:/Users/andre/Desktop/CRF/CRF/Header/Potential.h
- C:/Users/andre/Desktop/CRF/CRF/Source/Potential.cpp

5.39 Segugio::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 Segugio::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)
- 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.

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

5.39.1 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)

5.39.2 Constructor & Destructor Documentation

5.39.2.1 Potential_Exp_Shape() [1/3]

```
Segugio::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

5.39.2.2 Potential_Exp_Shape() [2/3]

```
Segugio::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

5.39.2.3 Potential_Exp_Shape() [3/3]

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

Use this constructor for cloning an exponential 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

5.39.3 Member Function Documentation

5.39.3.1 Substitute_variables()

```
void Segugio::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

5.39.4 Member Data Documentation

5.39.4.1 Distribution

```
std::list<I_Distribution_value*> Segugio::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:

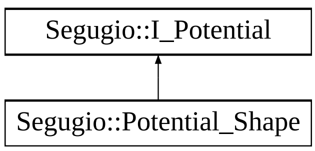
- C:/Users/andre/Desktop/CRF/CRF/Header/Potential.h
- C:/Users/andre/Desktop/CRF/CRF/Source/Potential.cpp

5.40 Segugio::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 Segugio::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)
- **Potential_Shape** (const Potential_Shape &to_copy)
- void Import (const std::string &file_to_read)
  For populating the image of the domain with the values reported in the textual file.
- 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.

Protected Member Functions

- void **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

5.40.1 Detailed Description

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

5.40.2 Constructor & Destructor Documentation

5.40.2.1 Potential_Shape() [1/4]

```
Segugio::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

5.40.2.2 Potential_Shape() [2/4]

```
Segugio::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

5.40.2.3 Potential_Shape() [3/4]

```
Segugio::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

5.40.2.4 Potential_Shape() [4/4]

```
Segugio::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

5.40.3 Member Function Documentation

5.40.3.1 Add_value()

```
void Segugio::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

5.40.3.2 Import()

```
void Segugio::Potential_Shape::Import (
            const std::string & file_to_read )
```

For populating the image of the domain with the values reported in the textual file.

Parameters

in	*file_to_read*	textual file to read containing the values for the image

5.40.3.3 Substitute_variables()

```
void Segugio::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:

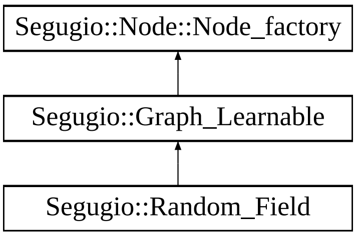
- C:/Users/andre/Desktop/CRF/CRF/Header/Potential.h
- C:/Users/andre/Desktop/CRF/CRF/Source/Potential.cpp

5.41 Segugio::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 Segugio::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_Observation_Set_var (const std::list< Categoric_var ∗ > &new_observed_vars)

    see Node::Node_factory::Set_Observation_Set_var(const std::list<Categoric_var∗>& new_observed_vars)
- void Set_Observation_Set_val (const std::list< size_t > &new_observed_vals)

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

Additional Inherited Members

5.41.1 Detailed Description

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

5.41.2 Constructor & Destructor Documentation

5.41.2.1 Random_Field() [1/3]

```
Segugio::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.

5.41.2.2 Random_Field() [2/3]

```
Segugio::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 2 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.

5.41.2.3 Random_Field() [3/3]

```
Segugio::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

5.41.3 Member Function Documentation

5.41.3.1 Insert() [1/2]

```
void Segugio::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.

5.41.3.2 Insert() [2/2]

```
void Segugio::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

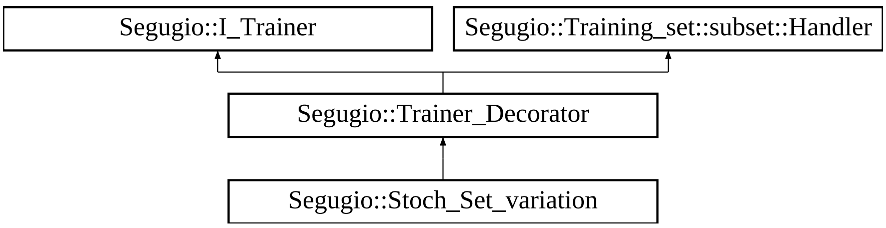
in	*vars_of_pot_whose_weight_is_to_share*	the list of varaibles involved in a potential already inserted whose weight is to share with the potential passed. They must be references to the variables actually wrapped into the model.

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

- C:/Users/andre/Desktop/CRF/CRF/Header/Graphical_model.h
- C:/Users/andre/Desktop/CRF/CRF/Source/Graphical_model.cpp

5.42 Segugio::Stoch_Set_variation Class Reference

Inheritance diagram for Segugio::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:/Users/andre/Desktop/CRF/CRF/Source/Trainer.cpp

5.43 Segugio::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)

5.43.1 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.

5.43.2 Constructor & Destructor Documentation

5.43.2.1 subset()

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

Parameters

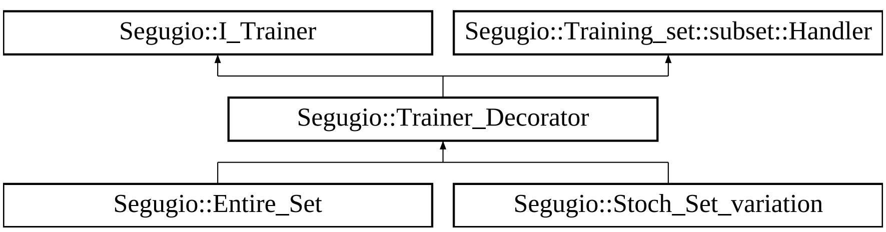
in	*set*	the training set from which this subset must be extracted
in	*size_percentage*	percentage to use for the extraction

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

- C:/Users/andre/Desktop/CRF/CRF/Header/Training_set.h
- C:/Users/andre/Desktop/CRF/CRF/Source/Training_set.cpp

5.44 Segugio::Trainer_Decorator Class Reference

Inheritance diagram for Segugio::Trainer_Decorator:



Public Member Functions

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

Protected Member Functions

- void **__check_tunable_are_present** (Graph_Learnable ∗model_to_train)

Protected Attributes

- Advancer_Concrete ∗ **Wrapped**

Additional Inherited Members

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

- C:/Users/andre/Desktop/CRF/CRF/Source/Trainer.cpp

5.45 Segugio::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.

5.45.1 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

5.45.2 Constructor & Destructor Documentation

5.45.2.1 Training_set() [1/2]

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

Parameters

in	*file_to_import*	file containing the set to import

5.45.2.2 Training_set() [2/2]

```
template<typename Array >
Segugio::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

5.45.3 Member Function Documentation

5.45.3.1 Print()

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

This training set is reprinted in the location specified.

Parameters

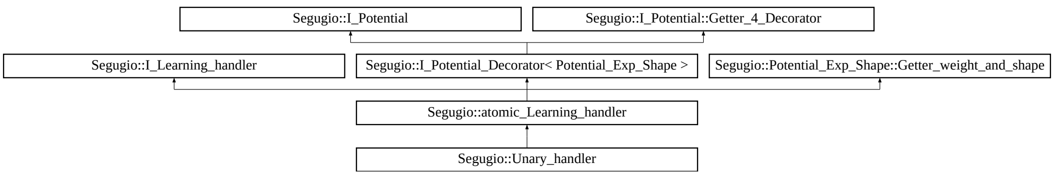
in	*file_name*	is the path of the file where the set must be printed

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

- C:/Users/andre/Desktop/CRF/CRF/Header/Training_set.h
- C:/Users/andre/Desktop/CRF/CRF/Source/Training_set.cpp

5.46 Segugio::Unary_handler Class Reference

Inheritance diagram for Segugio::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:/Users/andre/Desktop/CRF/CRF/Source/Graphical_model.cpp

5.47 Segugio::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:/Users/andre/Desktop/CRF/CRF/Header/Graphical_model.h
- C:/Users/andre/Desktop/CRF/CRF/Source/Graphical_model.cpp

Index