Hardware Implementation of Factor Graphs

eingereichtes
HAUPTSEMINAR
von

cand. ing. Thomas Templier

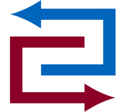
geb. am 12.11.1987
wohnhaft in:
Grasmeierstrasse 25
80805 München

Lehrstuhl für
STEUERUNGS- und REGELUNGSTECHNIK
Technische Universität München

Univ.-Prof. Dr.-Ing./Univ. Tokio Martin Buss
Univ.-Prof. Dr.-Ing. Sandra Hirche

Betreuer: Prof. Dr. sc. nat. Jörg Conradt
Beginn: 05.11.2010
Abgabe: 23.01.2011

TECHNISCHE UNIVERSITÄT MÜNCHEN
LEHRSTUHL FÜR STEUERUNGS- UND REGELUNGSTECHNIK
ORDINARIUS: UNIV.-PROF. DR.-ING./UNIV. TOKIO MARTIN BUSS
EXTRAORDINARIA: UNIV.-PROF. DR.-ING. SANDRA HIRCHE



18.10.2010

A D V A N C E D S E M I N A R
for
Thomas Templier, Mat.-Nr. 03277963

Hardware Implementations of Factor Graphs

Problem description:

Factor Graphs (FG) are graphical models that can be interpreted as distributed brain-inspired information processing systems. They are composed of processing nodes and message-passing "connections" between such nodes. Various computationally complex global problems can get decomposed in elementary computational operations that are performed in parallel on such graphs, to ultimately yield a consistent solution. The mathematical framework for regression and inference based on input data is well developed, turning such systems - in theory - into powerful Bayesian function approximators. However, FGs are not often applied to real-world problems, as (a) non-loopy FGs only solve "trivial" problems; and (b) loopy FGs require an large number of iterations before eventuelly convergencing into an equilibrium state on conventional computers.
Recently, several different approaches have been undertaken to investigate in specialized hardware solutions for computing FGs in real-time applications; examples of such are analog circuits for hearing aids or FPGA- or GPU- based customized systems for motor control. The task of this "Hauptseminar" will be to initially gain an understanding of FGs (review paper available) and afterwards review and summarize the large body of literature about hardware solutions for information processing in FGs.

Bibliography:

[1] Loeliger, Hans-Andrea. An Introduction to Factor Graphs. IEEE Signal Processing Magazine, January 2004: 28-41.
[2] Kschischang, Frank R, Brendan J Frey, and Hans-Andrea Loeliger. Factor Graphs and the Sum-Product Algorithm. IEEE Transactions On Information Theory 47, no. 2 (2001): 498-519.
[3] Damaj, I, J Hawkins, and A Abdallah. Mapping high level algorithms onto massively parallel reconfigurable hardware. ACS/IEEE International Conference on Computer Systems and Applications, 2003.
[4] Nedjah, N, and L de Macedo Mourelle. Stochastic reconfigurable hardware for neural networks. Euromicro Symposium on Digital System Design. IEEE, 2003. 438-442.

Supervisor: Jörg Conradt

(J. Conradt)
Professor

Contents

1 Introduction to Factor Graphs **5**
1.1 Graphical models 5
1.2 Factor Graphs 5
1.3 Example of block diagrams considered as factor graphs 6
1.4 Belief Propagation in Factor Graphs 8
1.5 Cycles in Factor Graphs 11
1.5.1 Cycle-free Factor Graphs 11
1.5.2 Factor Graphs with cycles 12
1.5.3 Variable stretching 13
1.6 Theoretical approach 14

2 Hardware implementation **17**
2.1 Analog processing with transistors for a node 17
2.1.1 Introduction and components 17
2.1.2 The multiplication matrix 18
2.1.3 Utilisation of the multiplication matrix 18
2.1.4 Simplifying the multiplication matrix 19
2.1.5 Using redundancy for accuracy 19
2.1.6 Example of a parity-check node 20
2.1.7 Results 20
2.2 An example of stochastic computing in a node 21
2.2.1 Short introduction to stochastic computing 21
2.2.2 Node implementation 22
2.2.3 Conclusion 24
2.3 General combinational stochastic logic 25
2.3.1 Stochastic coin flipper 25
2.3.2 Composition and abstraction 25
2.3.3 Parallelisation 26
2.4 Amelioration of belief propagation and message-passing schedules . . 27
2.4.1 Tile-based belief propagation 27
2.4.2 Message-passing schedules 28
2.5 Digital-circuit, analog-nodes 30

3 Summary **31**

List of Figures **33**

Bibliography **35**

Chapter 1

Introduction to Factor Graphs

This overview is essentially based on [Loe04] and the more detailed overview [KFL01], which gives a very good introduction to factor graphs.

1.1 Graphical models

Graphical models are very useful to engineers who aim to represent complex datas or algorithms. Well-known circuit digrams, trellis diagrams, block diagrams in control engineering or signal flow graphs are among the numerous existing graphical models. Areas that use graphical models are telecommunication and coding theory, image and sound processing, control engineering and automation or electronics. Over the past few decades, more and more similarity have been found between different algorithms associated to their graphical representation.The origins of factor graphs lie in coding theory and [Loe04] outlines the steps that have given birth to a general paradigm. It unifies the bulk of graphical algorithms which operate by message passing, that are found in numerous different areas.
That is why it is worth focusing our efforts on the very general factor graphs and their message passing algorithms which find application in vast domains.
Many well-known algorithms like the forward backward algorithm, the Viterbi algorithm in signal processing, the iterative turbo decoding algorithm, or even fast Fourier transform algorithms can be regarded as special cases of the sum-product algorithm.

1.2 Factor Graphs

We think that it is worth giving a brief introduction to factor graphs. By this mean, we will point out critical features that are essential to our problem of hardware implementation.
A factor graph is a graph that shows how a global function of many variables factors into a product of local functions.This point of view is very natural for the

computation of marginal functions in probability theory, which are the sums of the probabilities over some given variables. We take for example a function of five variables u, w, x, y, z that is the product of three different functions f_1, f_2, f_3 receiving inputs from the five variables mentioned.

$$f(u, w, x, y, z) = f_1(u, w, x) \cdot f_2(x, y, z) \cdot f_3(z) \tag{1.1}$$

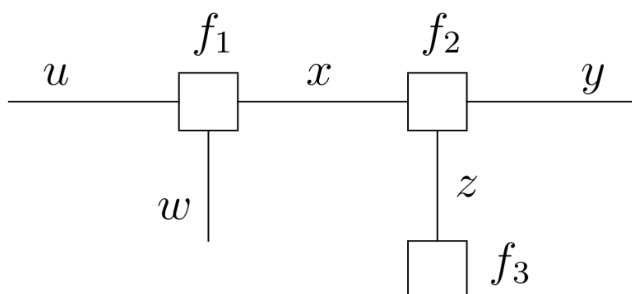


Figure 1.1: Forney-style Factor Graph representing the function f from equation 1.1

In Figure 1.1 the factor graph expressing this factorization is given with the Forney notation. A lot of different notations for factor graphs exist, but the Forney notation is particularly adapted for the consideration of general factor graphs.
As in Figure 1.1, a FG consists of edges and nodes. In the forney-style representation, a node represents a factor, and an edge represents a variable. A node (factor) is connected with an edge (variable) if and only if the factor depends on the variable represented by the edge. Concerning the unicity, a factor has a unique node, and a variable has a unique edge. In order to be able to express two functions that depend on a same variable, we can use an equality constraint node that practically duplicates a variable, see 1.3. Each factor can be considered as a local function, depending on the local variables represented by the edges that touch it. The product of the local functions is the global function. For example in Figure 1.1, the global function is f, and the local functions are f_1, f_2, f_3. A configuration is a particular assignment of values to all variables.
The common update rules for the graph are the sum product algorithm and the max sum algorithm.

1.3 Example of block diagrams considered as factor graphs

We give here two examples of well-known graphs that can be considered as factor graphs.

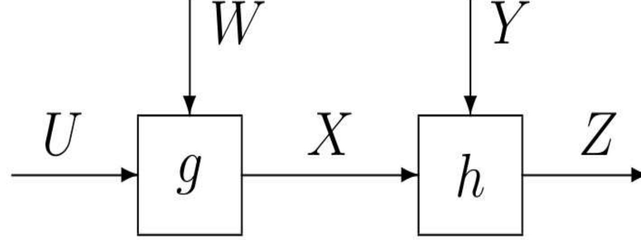


Figure 1.2: A classical elementary block diagram

The block diagram given in Figure 1.2 can be regarded as a factor graph. The two equations expressed by the block diagram are

$$\begin{cases} X = g(U, V) \\ Z = h(X, Y) \end{cases}$$

If we define the factors in terms of factor graph as $\delta(x - g(u, w))$ and $\delta(z - h(x, y))$, the global function is hence given by

$$f(u, w, x, y, z) = \delta(x - g(u, w)) \cdot \delta(z - h(x, y))$$

As we consider solely configurations that are consistent, i.e., configurations where local and global functions are non zero, this equation corresponds to the above system.

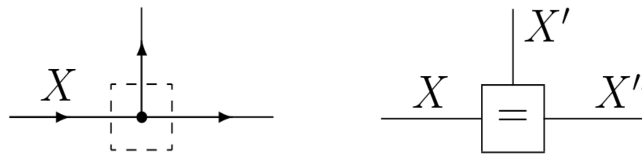


Figure 1.3: Branching point on a block diagram (left) and equality constraint node (right)

We lack then two factor graph units to enable the creation of an entire block diagram as a factor graph. The first one is the equality constraint node and corresponds to a branching point. Whith it we can circumvent the unicity of the pair edge-variable, and we can create different functions depending on the same variable. Figure 1.3 represents a branching point (left) on a block diagram, and the equality constraint node on a factor graph (right). The function at the node is

$$f_{=}(x, x', x'') = \delta(x - x') \cdot \delta(x - x'')$$

Once again, if the configuration in the factor graph is consistent, it corresponds to a branching point.
The second graph unit is the zero-sum constraint node. It represents a summation or soustraction on a block diagram. In Figure 1.4 the factor graph (left) and block diagram (right) versions are given. In this case, the function is

$$f_+(x, x', x'') = \delta(x + x' - x'')$$

and gives the "plus" on the block diagram if the configuration is consistent.
We see then that a general block diagram can be considered as a particular factor graph.

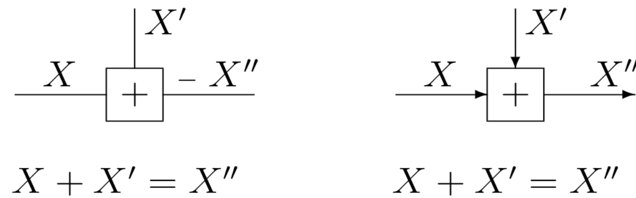


Figure 1.4: Zero-Sum constraint node (left) and branching point on a block diagram (right)

1.4 Belief Propagation in Factor Graphs

Let imagine that the factor graph shown in Figure 1.1 represents a bayesian network. Each node is a probability function and each edge is a probability variable. FG can indeed be regarded as structured probability distributions. The discrete probability mass function $f(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8)$ has the following expression

$$\begin{aligned} f(x_1, \ldots, x_8) &= \Big(f_1(x_1) f_2(x_2) f_3(x_1, x_2, x_3, x_4) \Big) \\ &\quad \cdot \Big(f_4(x_4, x_5, x_6) f_5(x_5) (f_6(x_6, x_7, x_8) f_7(x_7) \Big) \end{aligned} \quad (1.2)$$

With a probabilistic point of view, assume we want to calculate the marginal probability $p(x_4)$. Its expression is

$$p(x_4) = \sum_{x_1, x_2, x_3, x_5, x_6, x_7} f(x_1, \ldots, x_8). \quad (1.3)$$

It is the sum of all probabilities in every valid configuration.

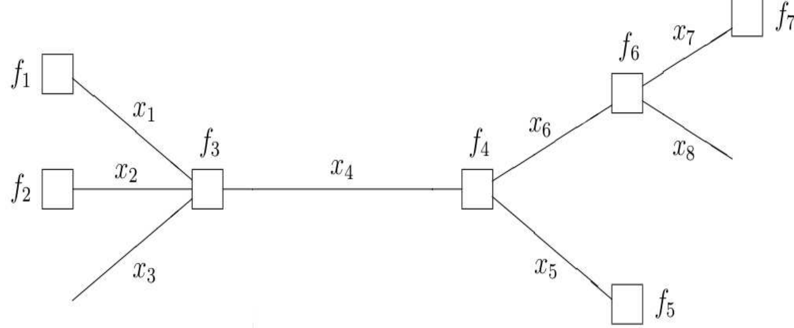


Figure 1.5: Factor Graph

Combining equations 1.2 and 1.3 and considering that f_1, f_2, f_3 depend only on $(x_1, \ldots, x_4)$ we have for the first step

$$\begin{aligned} p(x_4) &= \left(\sum_{x_1} \sum_{x_2} \sum_{x_3} f_1(x_1) f_2(x_2) f_3(x_1, x_2, x_3, x_4) \right) \\ &\cdot \left(\sum_{x_5} \sum_{x_6} \sum_{x_7} \sum_{x_8} f_4(x_4, x_5, x_6) f_5(x_5) (f_6(x_6, x_7, x_8) f_7(x_7) \right. \end{aligned} \tag{1.4}$$

And then finally we obtain the product

$$\begin{aligned} p(x_4) &= \left(\sum_{x_1} \sum_{x_2} \sum_{x_3} f_1(x_1) f_2(x_2) f_3(x_1, x_2, x_3, x_4) \right) \\ &\cdot \left(\sum_{x_5} \sum_{x_6} f_4(x_4, x_5, x_6) f_5(x_5) \left(\sum_{x_7} \sum_{x_8} (f_6(x_6, x_7, x_8) f_7(x_7) \right) \right) \end{aligned} \tag{1.5}$$

Now we identify three parts in this product : $\mu_{f_3 \to x_4}, \mu_{f_6 \to x_6}, \mu_{f_4 \to x_4}$ that can be seen in the following equation.

$$\begin{aligned} p(x_4) &= \underbrace{\left(\sum_{x_1} \sum_{x_2} \sum_{x_3} f_1(x_1) f_2(x_2) f_3(x_1, x_2, x_3, x_4) \right)}_{\mu_{f_3 \to x_4}} \\ &\cdot \underbrace{\left(\sum_{x_5} \sum_{x_6} f_4(x_4, x_5, x_6) f_5(x_5) \underbrace{\left(\sum_{x_7} \sum_{x_8} (f_6(x_6, x_7, x_8) f_7(x_7) \right)}_{\mu_{f_6 \to x_6}} \right)}_{\mu_{f_4 \to x_4}} \end{aligned} \tag{1.6}$$

For the interpretation, $\mu_{f_3 \to x_4}$ corresponds to the message sent by the node f_3. This message comes from left, and results of the consideration of all possible configurations of the set (x_1, x_2, x_3) with the given x_4. Similarly, $\mu_{f_6 \to x_6}$ is the message sent by the node f_6 computed with the messages coming from its right, i.e., with (x_7, x_8), and with x_6. These messages correspond to estimated marginal probabilities and are called beliefs. That is why the expression " belief propagation " is used. The ability to send messages can be interpreted as a process in which neighboring variables talk to each other, passing messages such as " I think that you belong in these states with these likelihoods ". And then, according to [Cou09], a high value of the message from node i to node j $m_{i,j}(x_j)$ means that the node i " believes" the marginal value $P(x_j)$ to be high.
Considering equation 1.6 on the graphical representation, it leads us to create boxes shown in Figure 1.6.

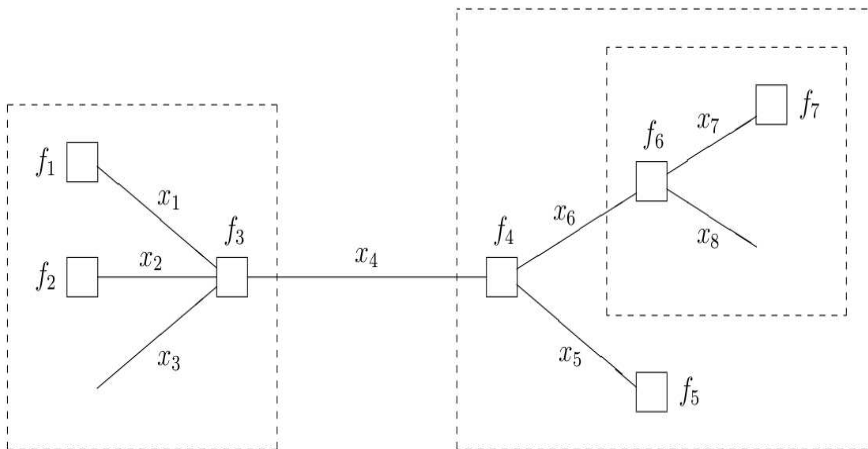


Figure 1.6: Factor Graph with encapsulating boxes

The final expression 1.6 is also interpreted as closing the dashed boxes by summarizing over their internal variables. The factor $\mu_{f_3 \to x_4}$ is the summary of the big dashed box left and is a function of x_4 only. We can then rewrite the final equation into the simplified one

$$p(x_4) = \mu_{f_3 \to x_4} \cdot \mu_{f_4 \to x_4} \tag{1.7}$$

The message $\mu_{f_3 \to x_4}$ is the summary of the subgraph to the left of X_4 and $\mu_{f_4 \to x_4}$ is the summary of the subgraph to the right of X_4. The precedent equation reflects the following fundamental rule in factor graphs, mentioned in [Mer05]. It enables after computation in the FG to obtain finally the value of a given variable on an edge.

The marginal for a given variable X corresponds to the multiplication of the two messages in opposite directions carried by the edge representing X.

The general calculation rule for a factor graph can now be imagined, given the shape of the final equation 1.6.

The message out of some node $g(x, y_1, \ldots, y_n)$ along the edge x is the function

$$\mu_{g \to x}(x) = \sum_{y_1} \cdots \sum_{y_n} g(x, y_1, \ldots, y_n) \mu_{y_1 \to g}(y_1) \cdots \mu_{y_n \to g}(y_n) \tag{1.8}$$

where $\mu_{y_i \to g}(y_i)$ is the message that arrives at g along the edge y_i.

Finally, we can show the messages traveling along the edges on Figure 1.7.

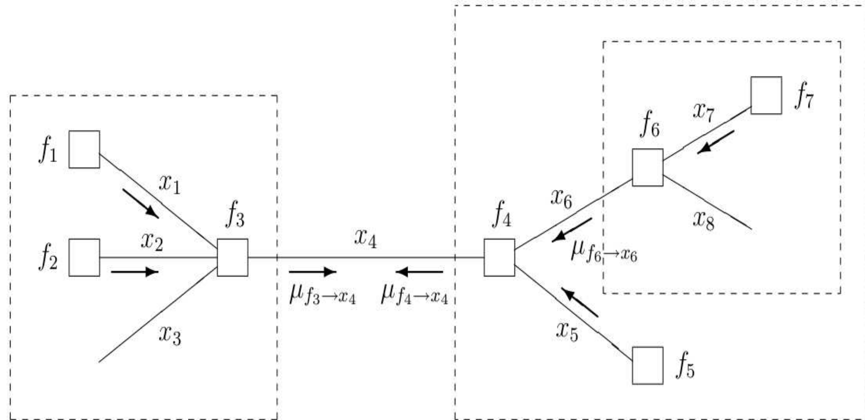


Figure 1.7: Factor Graph with boxes and traveling messages

1.5 Cycles in Factor Graphs

1.5.1 Cycle-free Factor Graphs

Cycle-free Factor Graphs, also called trees, are an easy case. To perform a computation on such graphs, the following steps must be underwent

- **Initialisation :** Each half edge present at the leaves of the graph is given a standard message, the same for all leaves.
- **Computation :** As long as a node has not received new messages from its connecting edges, it remains unchanged. As soon as it receives one new value

on at least one edge, it computes a new message and sends it along the remaining edges. The message is computed regarding Equation 2.1.

- **Result :** As stated in the rule mentioned above, the marginal of a given variable, is the product of the two messages carried out on the edge representing this variable.

An explicit scheme for this computation consists of only two rounds of calculations : a forward and a backward calculation. With cycle-free graphs, it is sure that the final state of the graph can be reached. Unfortunately, cycle-free graphs solely represent very simple algorithms and present hence a limited interest.

1.5.2 Factor Graphs with cycles

Factor Graphs with cycles will be systematically encountered in non-trivial algorithms. Implementing a neural network for the control of motion of a robot will certainly gives rise to graphs with cycles. Whereas the belief propagation algorithm in the cycle-free graph computes exact summaries, this will in general not be the case for graphs with cycles. Nothing ensures us that we can use the same initialisation and update rules as for cycle free FG and have a convergence of the computation. We also lack theoretical basics on cyclic graphs to decide wether the computation will converge or not, according to [Loe04]. It is the main reason why we try to design hardware implementations. Under the condition that it converges, hardware implementations will compute so quickly that a very good approximation of the final state can be reached. It is especially useful for real-time algorithms like coding-decoding algorithms, or automation algorithms.
We give the steps for a graph with cycle as we have done for cycle-free graphs.

- **Initialisation :** Each edge is initialized with a neutral message, and not juste the leaves, as with cycle-free graphs.
- **Computation :** The message is computed regarding Equation 2.1. The choice of a schedule is crucial in case of a FG with cycles.
- **Schedule :** In the second part of the report we will discuss more accurately these schedules.
  - Flooding schedule
  - Serial schedule
  - Two-way schedule
  - Synchronous
  - Asynchronous
- **Result :** As stated in the rule mentioned above, the marginal of a given variable, is the product of the two messages carried out on the edge representing this variable.

1.5.3 Variable stretching

The ultimate aim of the variable stretching is to transform a cyclic factor graph into a cycle-free factor graph. It could be a very good solution to avoid long calculations, as the computation on cycle-free graphs is almost trivial. Unfortunately, the uncycling process is very laborious and cannot be systematicaly employed on large cyclic factor graphs. We give here the principle of variable stretching, with help of [Mer05].

Considering the cyclic graph on Figure 1.8 (left), we give an elementary method to obtain a cycle-free graph (right). In this simple case, if we calculate the update

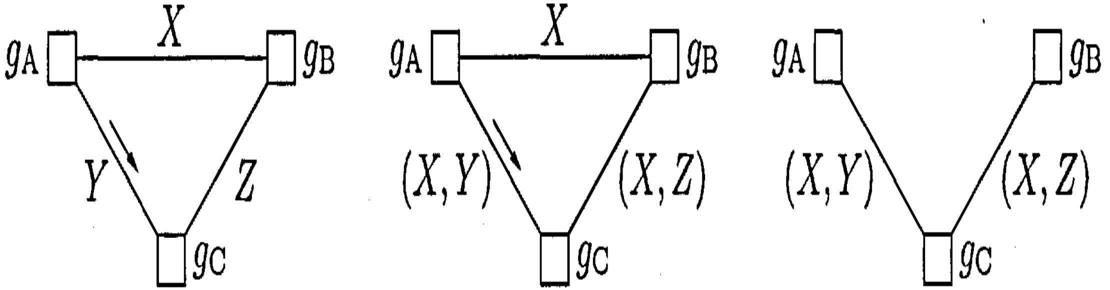


Figure 1.8: Method of stretching variables on an elementary factor graph

rule at the node g_A for the two first configurations of Figure 1.8, we have

$$\mu_{g\to z}(z) = \sum_x g_A(x,y) \cdot \mu_{x\to g}(x)$$
$$\mu_{g\to z}(z) = \sum_x g_A(x,x,y) \cdot \mu_{x\to g}(x).$$

We see now, that in the second case, we could delete the edge X and modify the node function g_A so that the two above expressions are equal. We obtain thus the third graph right on Figure 1.8, after removing the redundant edge X.
On this elementary example, we can imagine that by systematically stretching variables around cycles and then deleting a resulting redundant edge to break the cycle, it is possible to obtain a computation friendly cycle-free graph. The complexity of the local functions will not augment, but there is an increase in the complexity of the variable alphabets.
We also can imagine a selective variable stretching that removes only critical cycles. The question is to find the critical cycles. For the time being, no papers have been found on this subject. It is worth to mention that the dual hardware implementation (digital background circuitry, analog nodes) that we discuss later, would be optimal to use a selective variable stretching. Investigations on the transformation of cyclic factor graphs onto cycle-free graphs could break down an important barrier.

1.6 Theoretical approach

A first basic attempt to make connections between theoretical factor graphs with some hardware implementation has been done in [VL03]. The authors warn that the article does not have a practical aim. We have found this paper helpful as it gives some general rules that could be transposed to other implementation methods that are more likely to be practically designed.

Assume we have a factor graph representing a bayesian network with gaussian probabilities. Then according to [VL03], we have the following table in Figure 1.9that gives the correspondence between elementary units. [VL03] shows that there are two possible representations for the electrical networks, that are dual from each other, namely, the voltage representation and the current representation. In the first one, a variable corresponds to the voltage between two wires, whereas in the second one, it corresponds to the current in the wire. Using the table of Figure 1.9, we can translate the equality constraint node, the zero-sum constraint node, a multiplication node, and a gaussian distribution node. In case of vector variables,

	voltage mode	FFG	current mode
1	Y, X, Z	Y, X, $=$, Z $X = Y = Z$	Y, X, Z
2	Y, X, Z	Y, X, $+$, Z $X + Y = Z$	Y, X, Z
3	$1 : a$, X, Y	X, a, Y $Y = aX$	$a : 1$, X, Y
4	σ^2, m, X	X $e^{-\frac{(x-m)^2}{2\sigma^2}}$	m, X, σ^{-2}

Figure 1.9: Correspondence between the nodes in the factor graph and the components of the electrical network in the voltage (left) and current (right)versions

the elementary units given in Figure 1.9 can be extended and are given in [VL03] (vector addition, matrix multiplication).
In the article it is proven that the electrical network will settle in the unique global maximum of the global function.
This approach does not have a practical aim. Static electrical units are used in this model. Hence we can hardly imagine a big network with thousands of nodes modelling it (noise in resistances, DC-DC converters everywhere). We will now review some techniques for hardware implementation of factor graphs.

Chapter 2

Hardware implementation

We have seen so far that factor graphs can be used to implement a lot of algorithms. The principle of belief propagation is very general and is a foundation for numerous message-passing algorithms. The majority of the FG that are found by engineers in order to implement an algorithm are always cyclic graphs. On such graphs, there is no guarantee of convergence. If it converges, there is no guarantee that an acceptable approximate final state can be reached quickly. Implementing FG with analog components is a very promising approach. One can easily imagine that we could save a lot of time and power using an analog approach instead of a digital approach. The idea was first mentioned by [WLK96] to implement decoding algorithms of error correcting codes, that requires real-time computation. In this chapter we will see different approaches to design hardware implementation of factor graphs.

2.1 Analog processing with transistors for a node

2.1.1 Introduction and components

In this section we present the work of [LLH+98] and [FHAMS05] to design an analog processing node. We consider the case of discrete system variables, and assume that the node functions are binary. The circuits considered in this work are built up using transistors : they are pure transistor networks. Bipolar transistor and MOS transistors in weak inversion can be both used. They present the essential characteristic of an exponential current-voltage relation. Thus they are called translinearity elements. According to [LLH+98], translinear circuits are very well suited for circuit implementations of multitude of linear and non-linear arithmetic functions. A important feature that such circuits exhibit is that the computations carried out are fundamentally exact (neglecting second order effects) and do not rely on a linearization of the transistor characteristic. However we will see later that exact computations are not necessary. We notice also that these circuits show very low sensitivity to temperature variations.

2.1.2 The multiplication matrix

The multiplication matrix has been introduced in 1998 by Loeliger. This circuit is crucial for our purpose. It enables an easy computation of product of vectors. Assume we have two sets of input current $(I_{X,0}, \ldots, I_{X,M-1})$ and $(I_{Y,0}, \ldots, I_{Y,N-1})$. The output is the set of currents $(I_{0,0}, I_{0,1}, \ldots, I_{M-1,N-1})$. According to [Mer05] we have

$$I_{i,j} = \frac{1}{\sum_{i=0}^{M-1} I_{X,i}} \cdot I_{X,i} \cdot I_{X,j} \tag{2.1}$$

for $i = 0, 1, \ldots, M-1$ and $j = 0, 1, \ldots, N-1$. The circuit is given in Figure 2.1.

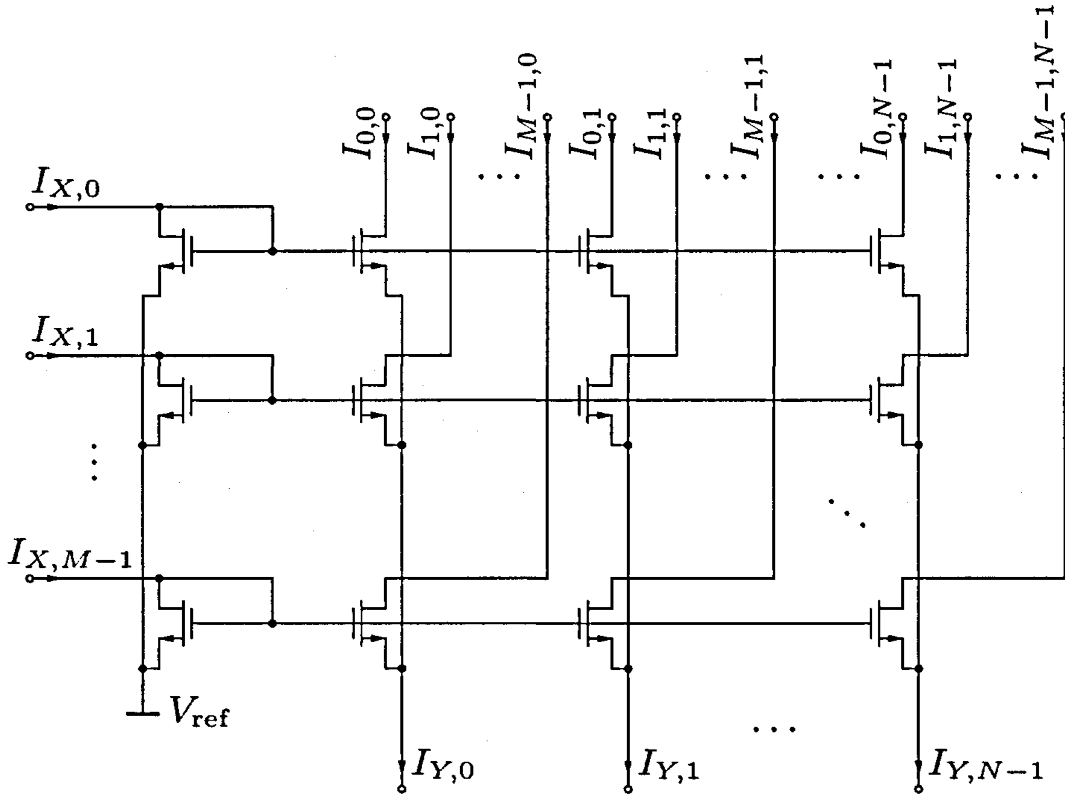


Figure 2.1: Multiplication matrix

2.1.3 Utilisation of the multiplication matrix

We recall the update rule given in the introduction part of the report.

$$\mu_{g \to x}(x) = \sum_{y_1} \cdots \sum_{y_n} g(x, y_1, \ldots, y_n) \mu_{y_1 \to g}(y_1) \cdots \mu_{y_n \to g}(y_n) \tag{2.2}$$

Computing products is the main calculation to do. Assume now we have to compute the product of $\mu_X(x)$ and $\mu_Y(y)$. We simply choose

$$I_{X,i} = I_X \cdot \mu_X(\epsilon_{X,i})$$
$$I_{Y,j} = I_Y \cdot \mu_Y(\epsilon_{Y,j})$$

with I_X and I_Y as scale factors for each equation. So the circuit computes the product of $\mu_X(x)$ and $\mu_Y(y)$. The choice of the scale factors is almost free. They must satisfy some conditions in order to let the transistor in the region of validity (weak inversion for MOS), explained in [Mer05].
This multiplication matrix is the elementary processing unit for products computing. An other circuit is shown in [Mer05] that enables also division, as it is sometimes required for other update rules. We do not relate this here, since the main idea stems from the basic multiplication matrix and has already been exposed.

2.1.4 Simplifying the multiplication matrix

In the case of dependent messages, for example if a same variable is shared by two neighboring functions, it is possible to simplify the multiplication matrix. This dependency forbids some configurations of the input variables, as they satisfy some conditions induced by their dependency. It means that some output currents calculated in the matrix are useless, since they are computed with non-valid input currents. It can thus lead us to delete lines or row from the multiplication matrix, based on the dependency of the input currents. It is formulated in [Mer05] by the following theorem.

Consider a multiplication matrix circuit. If none of the output currents originating from the ith row is used for subsequent processing, the entire row of transistors can be omitted. Similarly, if none of the output currents originating from the jth column is used for subsequent processing, the entire column of transistors can be omitted.

Remembering the process of stretching variables explained in the first part, we see that stretching variables produces graphs with dependencies for each cycle that has been removed. A combination of stretching variables that removes cycles and produces dependencies, with the matrix simplification exposed above could simplify tremendously the computation of factor graphs. No literature has been found on such a combination of these two methods.

2.1.5 Using redundancy for accuracy

If we choose not to employ the above mentioned method, in case of non-exact computation, we could use the redundancy to check the accuracy of the calculated outputs. Assume two overlapping messages $\mu_X(u, w)$ and $\mu_Y(w, v)$ that share the variable w

in common. If the two messages are consistent, then it is possible to obtain the message $\mu_W(w)$ by marginalizing either $\mu_X(u,w)$ or $\mu_Y(w,v)$. We could then define $\mu_W(w)$ as the average of the two marginalized variables. It is worth to mention that averaging is easily realisable, as stated in [Mer05], *App.C*.

2.1.6 Example of a parity-check node

We have explained the first part of the computation of the update rule that is the calculation of products. The second step is the summation. It is easily performed. Since variables are represented by currents, we just have to connect the necessary wires to create the sum, according to the Kirchhoff's law.
We present the case of a parity-check node. The function is defined as

$$f_{\oplus}(x,y,z) = \begin{cases} 1 & \text{if } x \oplus y \oplus z = 0, \\ 0 & \text{otherwise.} \end{cases}$$

Then using the update rule we have

$$\mu_Z(z) = \sum_x \sum_y f_{\oplus}(x,y,z) \cdot \mu_X(x) \cdot \mu_Y(y). \tag{2.3}$$

Calculating $\mu_Z(z)$ for $z = 0$ and $z = 1$ gives us

$$\begin{aligned} \mu_Z(0) &= \mu_X(0) \cdot \mu_Y(0) + \mu_X(1) \cdot \mu_Y(1) \\ \mu_Z(1) &= \mu_X(1) \cdot \mu_Y(0) + \mu_X(0) \cdot \mu_Y(1). \end{aligned} \tag{2.4}$$

Then $\mu_Z(0)$ and $\mu_Z(1)$ are the sum of two products. It corresponds to connect the two right outputs from the multiplcation matrix. The final circuit that computes the parity-check function for three variables is given in Figure 2.2.

2.1.7 Results

We mention here some results from [FHAMS05]. They applied the process mentioned above to implement two different error correcting codes. In such decoders, the iterations of the standard (discrete-time) decoding algorithm are replaced by the natural (continuous-time) settling behaviour of the transistor network. It is hoped, and partly corroborated by the results of [FHAMS05], that such analog decoders consume substantially less power (for a given speed) than a digital implementation of the corresponding iterative decoding algorithm.
A good characteristic that is good to mention is the decoding time, *i.e.*, the minimal settling time the decoder needs for attaining the minimal error rate. It corresponds to the better time step possible that does not induce errors. For the two tested decoders we have $t_{dec} = 10\,ms$ and $t_{dec} = 1\,ms$. The table given in Figure 2.3 summarizes important characteristics of the decoders.

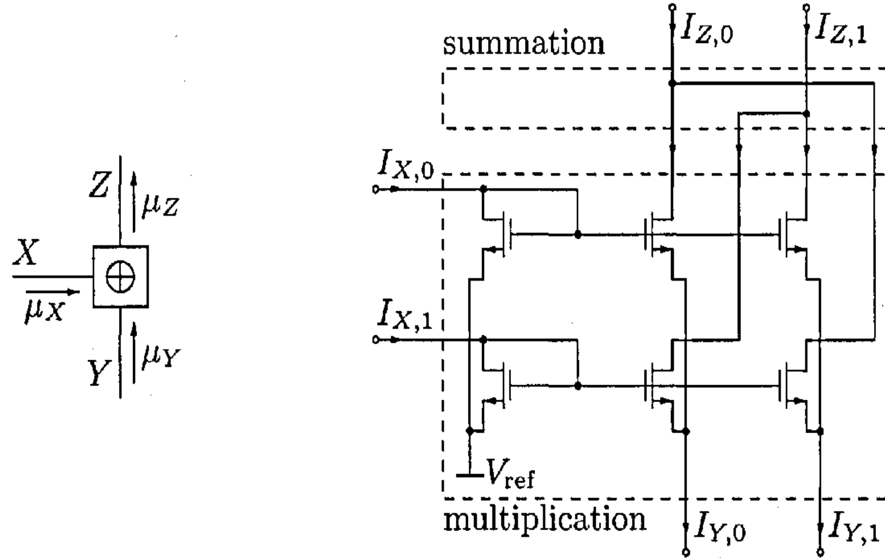


Figure 2.2: Implementation of a parity-check node

2.2 An example of stochastic computing in a node

In the previous section 2.1 we exposed works that aimed to increase computing performance in factor graphs by implementing analog nodes with transistors. An other way to improve the efficiency of calculations is to use “stochastic computing”. A detailed introduction to stochastic computations is given in [Gai67] and we outline here the basics. The principal amelioration given by stochastic computing versus exact computing is a huge saving on space.

2.2.1 Short introduction to stochastic computing

In a stochastic model, a number is represented by a long probabilistic bit-stream whose density of bits set to 1 is proportional to its numeric value. It means that instead of placing the exact value of a variable on a physical process (for instance a current in a wire, or a voltage between two wires), the value is hidden in the distribution of bits set to 1 transmitted successively on a channel. For example if we have this sequence on a wire

$$\underbrace{001001100010001000110100010010}_{30\,bits},$$

it corresponds to the value $\frac{10}{30} = \frac{1}{3}$ because there are ten 1 and a total of thirty bits. A disavantadge appears clearly : more clock cycles are needed to discover which value is hidden in a bit stream. However the increased number of clock cycles required to accomplish a given computation is compensated by the potential of massive parallelism enabled by the smaller circuit areas involved.

	(8,4,4) Hamming	(16,5,8) Reed-Muller
Technology:	0.25 μm (IBM6HP)	0.18 μm (IBM7HP)
Die size:	2×2 mm^2	2×2 mm^2
Active area:	0.5×0.7 mm^2	1×0.65 mm^2
nMOS size:	10 μm/ 1 μm	10 μm/ 1 μm
pMOS size:	30 μm/ 1 μm	30 μm/ 1 μm
V_{dd} (nominal):	1.8 V	1.8 V
V_{dd} (minimal):	0.7 V	0.9 V
$I_{\text{U}}^{\text{pad}}$ / $I_{\text{U}}^{\text{chip}}$:	1 μA/ 100 nA	1 μA/ 100 nA
P_{tot}:	55 μW (meas.)	55 μW (meas.)
P_{core}:	$<$ 5 μW (sim.)	$<$ 5 μW (sim.)
t_{dec}:	10 ms	1 ms
Energy per dec. info bit:	140 nJ (meas.)	11 nJ (meas.)

Figure 2.3: Some key data of the two tested decoders

2.2.2 Node implementation

In [NdMM03] a stochastic hardware architecture is presented for an artificial neuron in case of a feed-forward artificial neural networks. As explained in the introduction of the report, feed-forward networks, i.e., cycle-free graphs, are the easy case for computation in graphs. The important point in this paper is the use of stochastic computing within a node. All the products are computed at once in a single clock cycle, and it is possible thanks to the very limited area needed for stochastic computing elements.
In [NdMM03] it is mentioned that the choice of unipolar (variable $\in [0;1]$) or bipolar (variable $\in [-1;1]$) can have effects on the hardware implementation. Here we treat solely the bipolar case.

- **Three basic components :** Basic components are given in [NdMM03] : stochastic multiplier, stochastic adder, and stochastic subtracter. They are given in Figure 2.4. A simple trick is used and explained in the paper to produce normalized outputs from the adder and the subtracter.
- **Generating stochastic bit-streams from digital inputs and stochastic-**

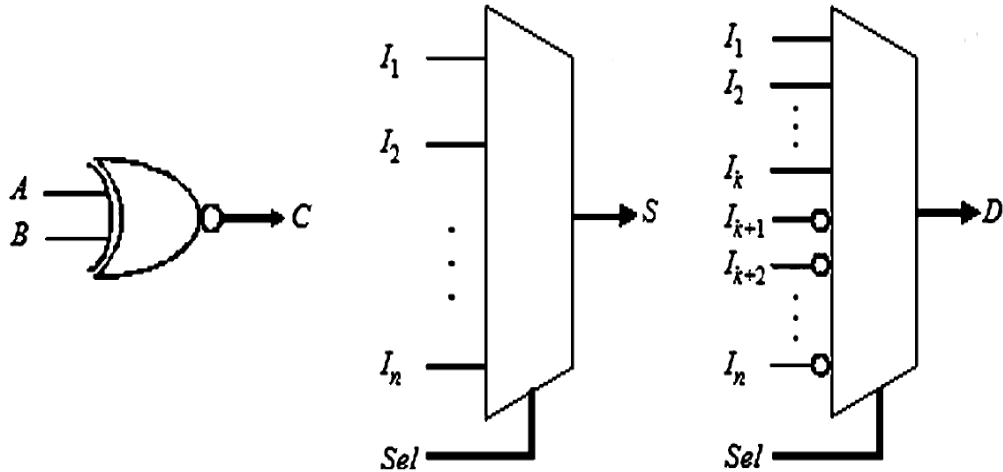


Figure 2.4: Basic commponents for bipolar stochastic computing : (left to right) multiplier, adder, subtracter

to-probability estimator: A circuit is given in [NdMM03] that is a digital-stochastic converter. Given a bipolar value $\in [0; 1]$, the circuit shown in Figure 2.5 produces a bit-stream with a density of bits equal to the given variable. We have to note that this circuit is just used at the beginning of the network.

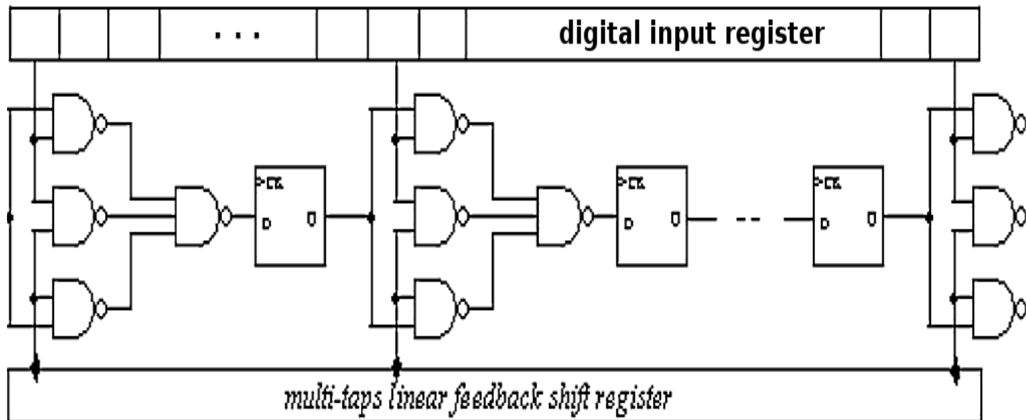


Figure 2.5: Digital-stochastic converter

The dual circuit stochastic-digital converter is used at the end of the circuit. It is applied for each neuron of the last layer. It simply consists of a bit-counter.

- **Source of pseudorandom digital noise :** Such sources are essential for stochastic circuits. There are two ways of implementation : Fibonacci and Galois.

- **Stochastic Neuron** As known for neural networks, a neuron with n inputs $(in_1, \cdots, in_n)$ and an activation A computes the following message

$$output = A\Big(\sum_{i=1}^{n} in_i \cdot w_i\Big) .$$

In the paper, an astutely designed state machine implements the activation function, that is a thresold activation function. A stochastic neuron is constitued of n AND gates that comput the n products, of a stochastic adder that calculates the sum of the weighted inputs, and an activation function.
The area needed for a neuron is hence clearly proportional to the number of inputs. The architecture of a bipolar neuron is given in Figure 2.6

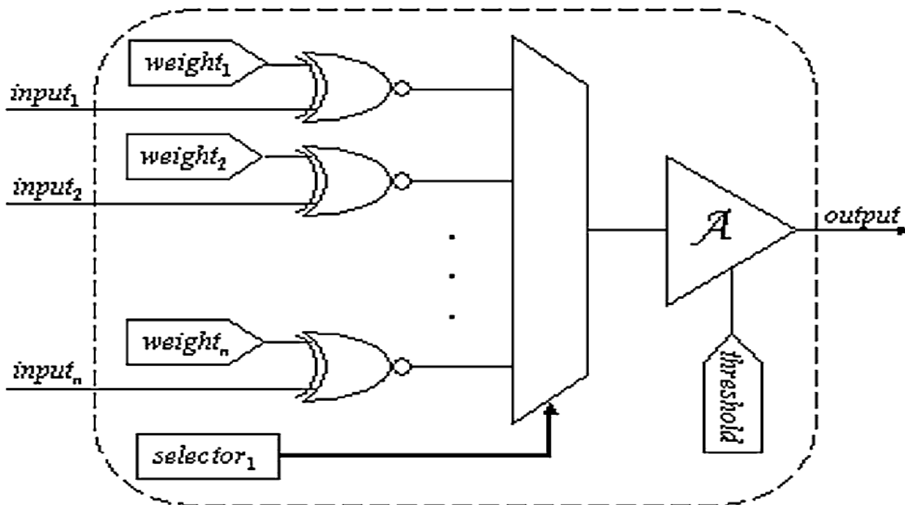


Figure 2.6: Architecture of a bipolar stochastic neuron

2.2.3 Conclusion

In a further study [NdMM07], a comparison is made between the above implementation and classic binary exact digital implementation. The binary version seem to be compute faster, but the area needed is huge compared to the area of stochastic neurons. On the overall, the *time* $\times$ *area* factor is clearly better for stochastical neurons, which is the most important parameter, as we want to deal with big networks in real-time applications. Comparison tables and graphs, can be found in [NdMM07].
The paper [NdMM03] exposes reconfigurable compact hardware implementation for stochastic neural networks. This implementation exhibits the advantage that it is an evolvable implementation. The digital background circuitry can be changed to treat different problems. In first approximation we can say that the nodes have a digital implementation, so that we can name this implementation “ digital-node,

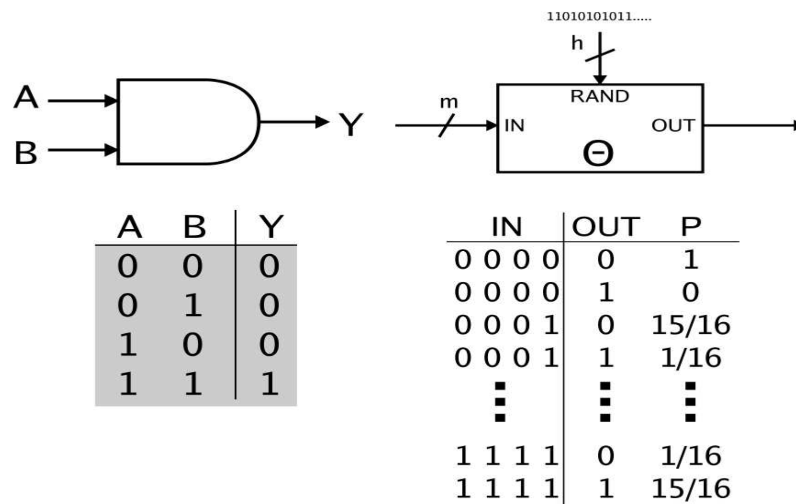
digital-circuit", or more precisely " stochastic-node, digital-circuit". We expose later an other promising evolvable hardware implementation that is an " analog-node, digital-circuit" implementation.

2.3 General combinational stochastic logic

The paper [MJT08] proposes a more general view of stochastic computing, and shows that stochastic logic is especially well suited to implement algorithms for probabilistic inference and nonlinear optimization. The paper cites Markoc chain and sequential Monte Carlo methods. Algorithms needing bayesian inferences are fundamentally stochastic, and the use of stochastic components at the hardware level will give better results than general-purpose computers.

2.3.1 Stochastic coin flipper

We give an example of elementary component given in [MJT08]. A classic boolean gate can be understood with its truth table. Similarly, a stochastic component can be regarded with its conditional probability table. Figure 2.7 give a classical boolean AND gate and a stochastic coin flipper.



A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

IN	OUT	P
0 0 0 0	0	1
0 0 0 0	1	0
0 0 0 1	0	15/16
0 0 0 1	1	1/16
⋮	⋮	⋮
1 1 1 1	0	1/16
1 1 1 1	1	15/16

Figure 2.7: A boolean AND gate and its table (left) and a stochastic coin flipper whith its conditional probability table (right)

2.3.2 Composition and abstraction

An essential feature presented in factor graphs, is the ability to close boxes around parts of the graph and the consider the created box as an unit with external inputs and outputs. The internal variables are no more visible. The similar feature is found

in stochastic elements. Combining two stochastic leads to a new stochastic element. Note that the former two stochastic elements had two different random bit streams, and that the new component is viewed whith just a stochastic bit stream. It is represented in Figure 2.8.

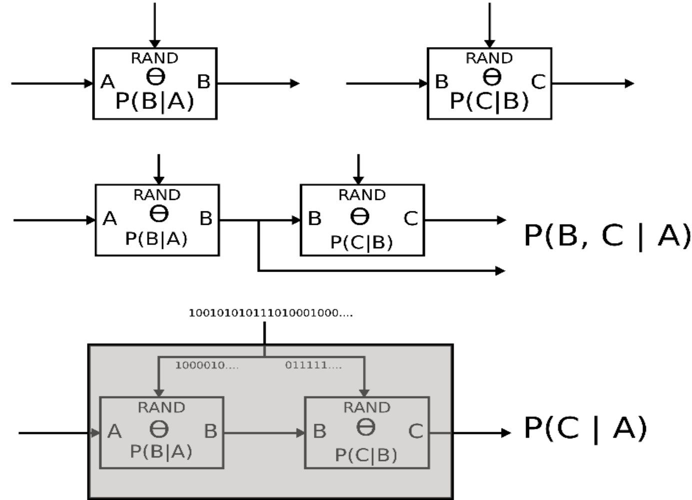


Figure 2.8: Composition and abstraction in stochastic elements

2.3.3 Parallelisation

In [MJT08], a method is proposed to achieve a high level of parallelisation in Gibbs decoders. We do not get deep into this field here, but note the method employed, that is specific to Markov random fields. Determining which nodes can be processed in parallel is in fact a coloring problem. We have to color the graph with as few colors as possible. Then nodes of same color can be calculated in parallel. Figure 2.9 gives two examples of coloring.

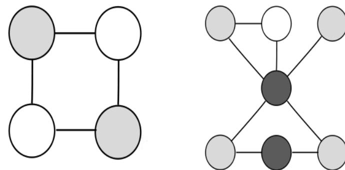


Figure 2.9: Two graphs with coloring schemes

2.4 Amelioration of belief propagation and message-passing schedules

A lot of different message schedules are known in graphs to compute the passing messages. The article [LCL+09] gives references of ameliorations to the algorithm of belief propagation :

- Energy Minimization Algorithms
- Iterated conditional modes (ICM)
- Graph cuts
- Max-product loopy belief propagation
- Tree-reweighted message passing.

2.4.1 Tile-based belief propagation

In [LCL+09], a method is explained that enables an amelioration of the classical belief propagation algorithm.
Many message-update schemes have been proposed to accelerate the convergence speed of computation. If we use the max-product algorithm (not presented in the introduction, but similar to the sum-product algorithm),there is a well-known schedule : messages are passed along the rows, that is, all messages travel in one direction, Then along columns. This well-known scheme is represented in Figure 2.10.
Given a factor graph, we split it in K different square tiles of size $B \times B$ $(S_1, \cdots, S_k)$. Classical schedules for messages passing lead, according to [LCL+09], to contruction methods that are sequential in nature, which cannot take advantage of modern parallel architectures. In this method we introduce parallelism in the belief propagation, by reducing the serial computations, that is, instead of calculating from node to node, we calculate messages from group of nodes to group of nodes. Instead of performing n iterations with messages passing on the whole graph, which require a lot of memory, we do the iterations from tile to tile. It gives a big advantage in terms of memory. All computed messages in a tile can be memorized on a buffer. The messages in the tile can be reused for updating each other many times. Because the number of these messages is much smaller than that of totally messages, they can be kept in the small but fast on-chip memory. The algorithm consists of the following steps.

At the iteration t of the whole process,

Compute the messages in tile S_1^t in I iterations
Save the "boundary messages" (messages that go out of the tile)

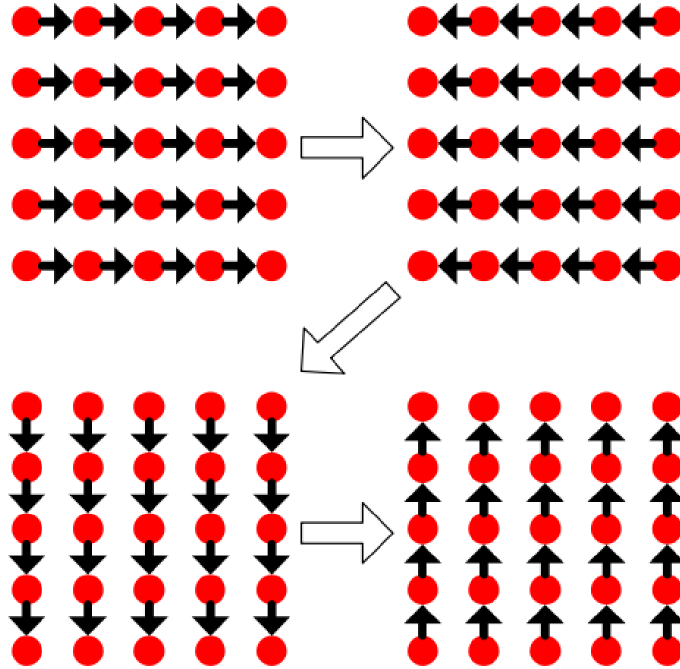


Figure 2.10: Right-left-up-down sweeping

Compute the messages in tile S_2^t *in* I *iterations, using the boundary messages from* $(S_1^t, S_3^{t-1}, \cdots, S_K^{t-1})$
$\vdots$
Compute the messages in tile S_k *in* I *iterations, using the boundary messages from* $(S_1^t, \cdots, S_{k-1}^t, S_{k+1}^{t-1}, \cdots, S_K^{t-1})$
Save the boundary messages

Figure 2.11 shows how a tile is computed, and Figure 2.12 shows how the sweeping is done.
[LCL+09] shows that this method is acurate and reaches convergence. [LCL+09] also makes a performance and cost analysis and shows that this method is a good amelioration of a normal belief propagation.

2.4.2 Message-passing schedules

Many message-passing schedules exist. In general, one needs experience to choose the right schedule for his algorithm. It depends on the architecture of the network, and its implementation. Flooding schedules are for example mor suited for high parallel implementations, whereas serial schedules should be used with serial implementations.

- **Serial schedule (or Two-Way schedule)** A node waits for each message to arrive before sending a new message

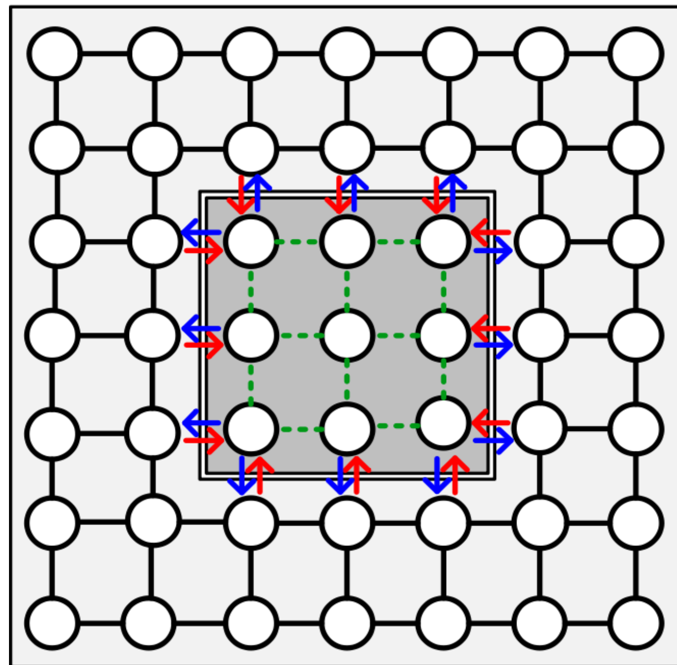


Figure 2.11: A tile receives messages from the boundaries

- **Flooding schedule** As explained in [KF98], an incoming message "activates" the node, which computes a new message with this incoming message and send it to all other edges. Messages may pass more than once in a given direction along an edge. This schedule consumes a lot more, as a lot more messages as needed are sent.

- **Hybrid schedule** When an incoming message arrives at a node, the node computes new messages but does not necessary send them (pending message). A given schedule could define a minimal number of incoming messages to wait for, before sending pending messages.

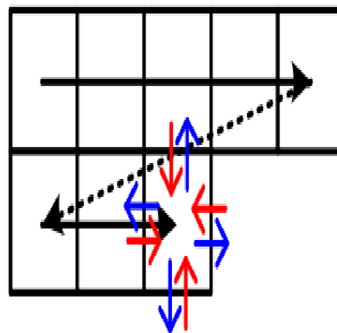


Figure 2.12: Raster scanning

2.5 Digital-circuit, analog-nodes

A promising approach combining advantages of methods presented precedently is to implement factor graphs with analog nodes, and a digital circuit. The advantages of analog nodes have been exposed in 2.1. Such nodes settle quasi instantaneously to the final state. The digital circuitry transforms the graph in an evolvable graph. Quasistationary weights can be given to the graph. This could be especially useful when the task solved by the algorithm changes. For instance, a robot catching an object and the same robot catching the same object with an obstacle. The change in the task leads to a new connectivity map in the graph, that can be instantaneously settled by the digital background circuitry.
New update rules should also be investigated to reduce the analog size of the nodes. [Con10] cites for example non linear sum and sum threshold algorithms that would be less time and space consuming. More investigations on this topic have to be done.

Chapter 3

Summary

Factor graphs are a general model that encompass numerous message-passing algorithms. When used on a general purpose computer for real time applications, they lack the necessary speed to be performant. Implementing these algorithms directly on hardware seems very ambitious, and convincing results have been obtained for fifteen years. Several trends have beeen given in this advanced seminar.

- As observed in 2.1, the sum-product algorithm can be mapped directly into simple analog transistor circuits.
- Nodes can be implemented with stochastic logic. It saves a lot of space on hardware.
- A method to develop is to build factor graphs on FPGA, with analog nodes and a digital background circuitry. It should combine also simplified versions of the sum-product algorithm
- Parallelization by the mean of ameliorations of the basic belief propagation algorithm.

Grasping the details of the papers concerning the harware implementation of factor graphs was sometimes difficult, because new techniques are explained in a specific application of factor graphs. For example [MJT08] was dealing with Monte Carlo algorithms, [KF98] with stereo matching in computer vision and [Mer05] with decoding algorithms.

List of Figures

1.1 Forney-style Factor Graph representing the function f from equation 1.1 6
1.2 A classical elementary block diagram 7
1.3 Branching point on a block diagram (left) and equality constraint node (right) 7
1.4 Zero-Sum constraint node (left) and branching point on a block diagram (right) 8
1.5 Factor Graph 9
1.6 Factor Graph with encapsulating boxes 10
1.7 Factor Graph with boxes and traveling messages 11
1.8 Method of stretching variables on an elementary factor graph 13
1.9 Correspondence between the nodes in the factor graph and the components of the elect

2.1 Multiplication matrix 18
2.2 Implementation of a parity-check node 21
2.3 Some key data of the two tested decoders 22
2.4 Basic commponents for bipolar stochastic computing : (left to right) multiplier, adder,
2.5 Digital-stochastic converter 23
2.6 Architecture of a bipolar stochastic neuron 24
2.7 A boolean AND gate and its table (left) and a stochastic coin flipper whith its conditio
2.8 Composition and abstraction in stochastic elements 26
2.9 Two graphs with coloring schemes 26
2.10 Right-left-up-down sweeping 28
2.11 A tile receives messages from the boundaries 29
2.12 Raster scanning 29

Bibliography

[AJ07] N. Ahmad and P. R. Jacob. Design of a simplified fuzzy inference engine using fpga. *Control Intell. Syst.*, 35:175–182, March 2007.

[Con10] Jorg Conradt. Digital hybrid digital-analog hardware for neuro-inspired graphical information processing. 2010.

[Cou09] James Coughlan. *A Tutorial Introduction to Belief Propagation*. PhD thesis, The Smith Kettlewell Eye Research Institute, 2009.

[DHA03] I. Damaj, J. Hawkins, and A. Abdallah. Mapping high level algorithms onto massively parallel reconfigurable hardware. In *International Conference of Computer Systems and Applications*, page 14 22, Tunisia, July 2003.

[FHAMS05] M. Frey, Loeliger H.-A., Patrick P Merkli, and P. Strebel. Two experimental analog decoders. *Proc. 2005 IEEJ Int.*, 2005.

[Gai67] B. R. Gaines. Stochastic computing. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 149–156, New York, NY, USA, 1967. ACM.

[KF98] Frank R. Kschischang and Brendan J. Frey. Iterative decoding of compound codes by probability propagation in graphical models. 1998.

[KFL01] Frank R. Kschischang, Brendan J. Frey, and Hans-Andrea Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*, 47:498–519, 2001.

[LCL+09] Chia-Kai Liang, Chao-Chung Cheng, Yen-Chieh Lai, Liang-Gee Chen, and H.H. Chen. Hardware-efficient belief propagation. *Computer Vision and Pattern Recognition, IEEE Computer Society Conference on*, 0:80–87, 2009.

[LLH+98] Hans Andrea Loeliger, Felix Lustenberger, Markus Helfenstein, Felix Tarkoy, Eth Zurich, and Endora Tech Ag. Probability propagation and decoding in analog vlsi. 1998.

[Loe04] H. Loeliger. An Introduction to factor graphs. *IEEE Signal Processing Magazine*, 21(1):28–41, January 2004.

[Mer05] Patrick P Merkli. *Message-passing algorithms and analog electronic circuits*. PhD thesis, Technische Wissenschaften ETH Zurich, Zurich, 2005.

[MJT08] Vikash K. Mansinghka, Eric M. Jonas, and Joshua B. Tenenbaum. Stochastic digital circuits for probabilistic inference. 2008.

[NdMM03] Nadia Nedjah and Luiza de Macedo Mourelle. Reconfigurable hardware architecture for compact and efficient stochastic neuron. In Jose Mira and Jose Alvarez, editors, *Artificial Neural Nets Problem Solving Methods*, volume 2687 of *Lecture Notes in Computer Science*, pages 1043–1043. Springer Berlin / Heidelberg, 2003. 10.1007/3-540-44869-13.

[NdMM07] Nadia Nedjah and Luiza de Macedo Mourelle. Reconfigurable hardware for neural networks: binary versus stochastic. *Neural Computing ; Applications*, 16:249–255, 2007. 10.1007/s00521-007-0086-x.

[Tou08] Marc Toussaint. Lecture notes: Factor graphs and belief propagation. 2008.

[VL03] P. Vontobel and H. Loeliger. On factor graphs and electrical networks, 2003.

[WLK96] Niclas Wiberg, Hans-Andrea Loeliger, and Ralf Kotter. Codes and iterative decoding on general graphs /. *European Transactions on Telecommunications*, 1996.