MASTER OF FUNDAMENTAL COMPUTER SCIENCE
*École Normale Supérieure de Lyon*
*Université Claude Bernard Lyon I*

Internship 2014–2015
Romain LABOLLE
Master 1

# Sudoku solving by factor graphs

Supervisors :
**Stephan Pfletschinger** `@ieee.org`
**Pau Closas** `@cttc.cat`
**Monica Navarro** `@cttc.cat`
Centre Tecnològic de Telecomunicacions de Catalunya (CTTC)
*Parc Mediterrani de la Tecnologia (PMT) - Building B4*
*Av. Carl Friedrich Gauss 7*
*08860 Castelldefels (Spain)*
`http://www.cttc.cat/`

# Contents

# Chapter 1

# Introduction

The goal of this internship was to study the Sum-Product Algorithm on Factor Graphs, an algorithm that can be used to express and solve many different kinds of problems. We especially focused on one example, the Sudoku solving problem, and studied different kinds of optimisations that could be made to the algorithm. Some of the optimisations we found can be them generalized to other problems.

In this document, we first introduce the different notions relative to Factor Graphs and the Sum-Product Algorithm, we then show how it can be applied to the Sudoku solving problem and we finally present different optimisations and the resulting performances.

# Chapter 2

# Factor Graphs

A Factor Graph (FG)[7] is a bipartite graph that can be used to describe the structure of the factorization of a function. For example, let us consider a function $f(x_1, x_2, x_3, x_4, x_5)$ that can be factorized as:

$$f(x_1, x_2, x_3, x_4, x_5) = f_1(x_1, x_2)f_2(x_2, x_3)f_3(x_4)f_4(x_4, x_5). \tag{2.1}$$

We can express the structure of this factorization by a FG. This is a bipartite graph with variable nodes for each variable $x_\nu$ on one side and factor nodes for each local function $f_\mu$ on the other side, with an edge between $x_\nu$ and $f_\mu$ if and only if $f_\mu$ is a function of $x_\nu$.

The FG corresponding to the example (2.1) is shown in figure 2.1. The typical representation uses squares for factor nodes and circles for variable nodes.

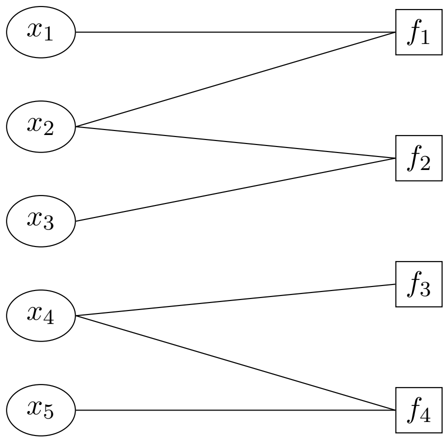


Figure 2.1: Example Factor Graph corresponding to the function in (2.1)

This type of problem representation is very useful in, for instance, large systems over which it might be difficult to make inference. Using FG theory and tools, we can divide a complex problem into small pieces that are easier to deal with by accounting for the variables' dependencies.

Furthermore, it allows for an iterative solution that reduces complex problems into feasible implementations.

## 2.1 Sum-Product Algorithm

In FGs, one is typically interested in making inferences about the marginal function of each variable node. This could be complex in general, but FG theory provide a powerful framework.

The Sum-Product Algorithm (SPA)[7] is a message-passing algorithm working on FGs. It is a generalization of algorithms used in many domains such as Low-Density-Parity-Check (LDPC) decoding, Belief Propagation on Bayesian networks, Kalman filtering, etc. There is no assumption about messages passed between the nodes and it can be, for example, sets of possible values, probability mass functions or probability density functions. In the following sections, we will mainly use messages representing probability distribution: sets of possible values or probability vector in discrete cases, parameters of a Gaussian distribution, ...

### 2.1.1 Message-passing rules

In order to compute the messages passed from one node to another we follow different rules described in [7]. A given message from a node $a$ to a node $b$ depends on messages received by $a$ from all its neighbors except $b$. An example is shown in figure 2.2.

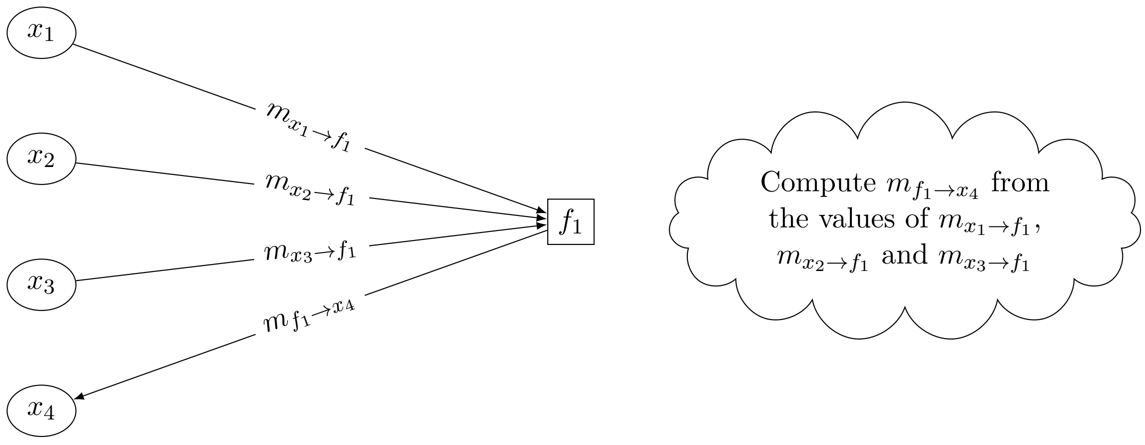

$x_1$
$x_2$
$x_3$
$x_4$
$m_{x_1\to f_1}$
$m_{x_2\to f_1}$
$m_{x_3\to f_1}$
$m_{f_1\to x_4}$
$f_1$
Compute $m_{f_1\to x_4}$ from the values of $m_{x_1\to f_1}$, $m_{x_2\to f_1}$ and $m_{x_3\to f_1}$

Figure 2.2: SPA: example message-passing on a FG, message generated from factor node $f_1$ to variable node $x_4$

#### Variable node to factor node messages

Variable node to factor node messages (VFMs) are the easiest to compute, it's simply the product of all other incoming messages:

$$m_{x\to f}(x) = \prod_{h\in\mathcal{N}(x)\backslash\{f\}} m_{h\to x}(x) \tag{2.2}$$

where $\mathcal{N}(x)$ represent the neighbors of $x$ in the FG.

#### Factor node to variable node messages

Factor node to variable node messages (FVMs) are more complex to compute:

$$m_{f\to x}(x) = \sum_{\sim\{x\}} f(X) \left( \prod_{y\in\mathcal{N}(f)\backslash\{x\}} m_{y\to f}(y) \right) \tag{2.3}$$

where $X = \mathcal{N}(f)$ represent the neighbors of $f$ in the FG and $\sum_{\sim\{x\}}$ is the summary operator described below.

### Termination rule

The final result of the algorithm for a given variable $x$ is called the marginalization of the function and is given by:

$$m(x) = \prod_{h \in \mathcal{N}(x)} m_{h \to x}(x) \tag{2.4}$$

If the messages being passed are sets of possible values or discrete probability distributions, we can use this marginal function to find an estimate of the corresponding variable, which is the ultimate goal in FGs.

The termination rule itself depend on the problem. In some cases, we can have a clear termination rule (for example in LDPC decoding, the algorithm terminate when all check nodes are satisfied), in other cases we either have to limit the number of iterations or consider that a message doesn't change anymore once the difference of its value at each iteration have reached a certain threshold.

### The summary operator

The summary operator $\sum_{\sim\{x_i\}} f(X)$ represent the summary of a function $f$ over all possible values of all variable $x_1, \dots, x_n$ except $x_i$. Depending on the definition domain of the function $f$, it can either be a sum $f$ is define over a discrete domain:

$$\sum_{\sim\{x_i\}} f(X) = \sum_{x_1} \cdots \sum_{x_{i-1}} \sum_{x_{i+1}} \cdots \sum_{x_n} f(x_1, \dots, x_{i-1}, x_i, x_{i+1}, \dots, x_n) \tag{2.5}$$

or an integral if $f$ is define over a continuous domain:

$$\sum_{\sim\{x_i\}} f(X) = \int_{D_1} \dots \int_{D_{i-1}} \int_{D_{i+1}} \dots \int_{D_n} f(x_1, \dots, x_{i-1}, x_i, x_{i+1}, \dots, x_n) \, \mathrm{d}x_n \dots \mathrm{d}x_{i+1} \, \mathrm{d}x_{i-1} \dots \mathrm{d}x_1 \tag{2.6}$$

where $\sum\limits_{x_k} = \sum\limits_{x_k \in D_k}$, $\int\limits_{D_k} = \int\limits_{x_k \in D_k}$ and $D_k$ is the domain of definition of the function for the $k$-th variable.

# Chapter 3

# Sudoku solving

## 3.1 Presentation of the problem

Lets consider a $q \times q$ grid subdivided into $q$ sub-grids of size $p \times p$ (with $q = p^2, p \in \mathbb{N}^*$). The goal of the Sudoku game is to fill the grid with numbers from 1 to $q$ so that each line, each row and each sub-grid contains exactly one of these numbers. In each instance of the game, some cells are already filled in so that there is a unique solution. For regular Sudoku $q = 9$, but to simplify the representation, examples will use $q = 4$.

We can define the constraint function $\mathcal{C} : \{1, \ldots, q\}^{q^2} \to \{0, 1\}$ for Sudokus that associate to a grid a value of 1 if it respects all the Sudoku rules and 0 otherwise. This function can be factorized into a product of $3 \cdot q$ local constraint functions depending only on $q$ variables each:

$$\mathcal{C}(x_1, \ldots, x_{q^2}) = c_1(x_1, \ldots, x_q) \cdot c_2(x_{q+1}, \ldots, x_{2q}) \cdots c_{3q}(\ldots) \tag{3.1}$$

The first $q$ functions represent the row constraints, then the column constraint and finally the $q$ last functions represent the sub-grid constraints. Those functions are all the same and return 1 if and only if all its input arguments are different. In other words, it checks if its input is a permutation of the set $\{1, \ldots, q\}$.

## 3.2 Solving Sudokus with the Sum-Product Algorithm

Since the constraint function can be factorized as in (3.1), it can be represented as a factor graph (see figure 3.1).

Then by applying the SPA over the FG, we deduce the marginal functions $m(x_i)$ and find the value of $x_i$ that yields $\mathcal{C}(x_1, \ldots, x_{q^2}) = 1$.

In this case, and to simplify the computation, the messages passed across the nodes will be sets of possible values, which can also be seen as probability vectors of size $q$ with all values being either 0 or 1 (with a normalization coefficient). Thus the sum of two messages is the union of the sets and the product is the intersection. We initialize the values with the known values of the grid to solve:

- $x_i = \{k\}$ if the cell $i$ containts $k$
- $x_i = \{1, \ldots, q\}$ if the cell $i$ is empty

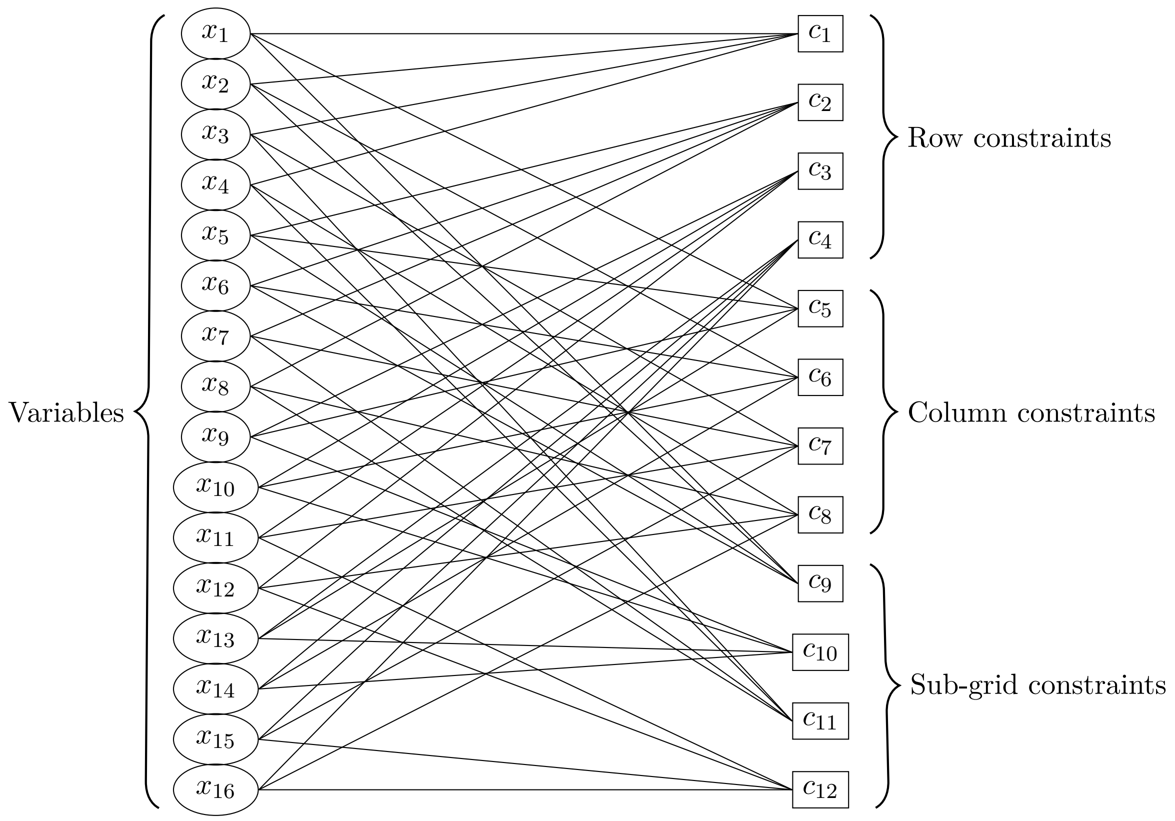


Figure 3.1: FG representation of the Sudoku constraint function $\mathcal{C}$ described in (3.1) for $q = 4$

## 3.3 Implementation

In order to test the algorithm, we have implemented a generic Python class to represent a FG and apply the SPA to it. We have then implemented different optimizations on the way and order in which messages are computed in order to achieve better performance, by simplifying the message computation and by reducing the number of computed messages.

Some of those optimization are specific to the given problem (but very similar optimization might be found for other problems), other are general and might be applied to other instances of the SPA

### 3.3.1 Original implementation

This is the most generic implementation, with almost no assumption on the nature of the FG. However, it is relatively slow, mostly due to the time taken by the FVMs computation. Indeed, in the case of Sudokus, the local function $c_i$ is non zero if its arguments are a valid permutation of $\{1, \ldots, q\}$. So in the FVM computation, the formula becomes:

$$m_{f\to x}(\mathbf{x}) = \sum_{\sim\{x\}} f(X) \left( \prod_{\mathbf{y}\in\mathcal{N}(f)\setminus\{x\}} m_{y\to f}(\mathbf{y}) \right) = \sum_{X\in\mathfrak{S}_x(\mathbf{x})} \left( \prod_{\mathbf{y}\in\mathcal{N}(f)\setminus\{x\}} m_{y\to f}(\mathbf{y}) \right), \tag{3.2}$$

where $\mathfrak{S}_x(\mathbf{x})$ is the set of all permutations of $\mathcal{N}(f) = \{1, \ldots, q\}$ in which the variable $x$ has the value $\mathbf{x}$. Since this involves permutation of all but one variable, we have the $\mathrm{Card}(\mathfrak{S}_x(\mathbf{x})) = (q-1)!$

Therefore, the complexity of the FVM computation in this case is $\mathcal{O}(q!)$, where $\mathcal{O}(\cdot)$ is the big O Landau notation.

We did not implemented the naive algorithm that compute the product even if $f(X) = 0$ since its complexity would have been $\mathcal{O}(q^q)$. Indeed, in the naive algorithm, the summary operator iterate over the $q$ possibilities for the $q-1$ variable and thus have to compute $q^{q-1}$ times a product of $q$ messages.

### 3.3.2 Faster message computation

The FVM computation can be simplified by using the following formula [4]:

$$m_{f\to x}(\mathbf{x}) = \{1,\ldots,q\} - \bigcup_n A_n \tag{3.3}$$

Where $A_n$ is any set such that

$$\exists \mathcal{J} \subset \{1,\ldots,q\} \text{ such that } \begin{cases} x \neq x_j, \forall j \in \mathcal{J} \\ A_n = \bigcup\limits_{j\in\mathcal{J}} m_{x_j\to f}(\mathbf{x_j}) \\ \text{Card}(\mathcal{J}) = \text{Card}(A_n) \end{cases} \tag{3.4}$$

In order to find all the $A_n$, we have to iterate over at most $2^{q-1}$ possible values of $\mathcal{J}$. We hence get a complexity of at most $\mathcal{O}(2^q)$.

Moreover if we subtract the $A_n$ as we compute them, we can improve the algorithm a little bit in some cases: if we obtain $m_{f\to x}(\mathbf{x}) = \emptyset$, we can stop searching for other $A_n$ as it will not change the result.

This approach is very close to the technique that is actually used by humans to solve Sudokus: if in a given row (resp. column and sub-grid) there are $k$ cells that can only accept a subset of $k$ values, these values can be removed from the possibilities of all other cells in the row (resp. column and sub-grid). The case $k = 1$ removes values already placed in the grid and the case $k = q-1$ complete a line with only 1 remaining cell.

This approach cannot be generalized to all instances of SPA, but it can actually be used on problems similar to this one where the messages are probabilities that are uniform on their support (and hence can be expressed as sets).

### 3.3.3 Message computation scheduling

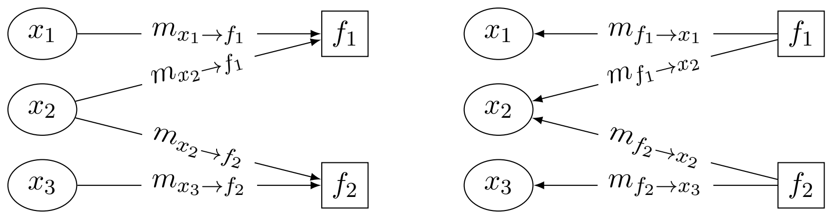
Messages can actually be computed in any given order and some scheduling might require less message computation in order to converge.

We have seen that VFM are fast to compute compared to FVM, thus we aim at minimizing the number of FVM computations in order to speed up our algorithm.

The scheduling we used for all the previous implementations is the naive one where we first compute all VFMs and the all FVMs, as shown in figure 3.2, and then repeat those two steps until all the messages are stabilized (no message was changed by those two steps).

#### Tracking VFM changes

The first improvement we have implemented is to track VFM changes in order to avoid computing twice a FVM with the same inputs.



Compute all VFMs first Then all FVMs

Figure 3.2: Example of naive scheduling on a simple FG

Indeed, especially towards the end of the algorithm when only a small portion of the graph is not stabilized, the naive scheduling was recalculating all the messages at each step even if only a few of them had actually to be computed.

### Focusing on FVMs

We then changed our scheduler to focus on the FVMs computation, propagating these messages immediately by computing the affected VFMs (see figure 3.3), in order to take into account the most recent updates for the next FVM computation. In this scheduler, the FVMs are always computed in the same order and we loop through them until the messages stabilize.

The idea here is still to try to reduce the number of FVMs computations, but this time by taking into account as much information as possible in each computation to make messages converge faster and reducing the number of iterations needed.

We reuse the previous optimization to avoid computing twice the same message. In order to do so, each time a VFM change, the destination factor node is marked as outdated for all its neighbors except the one sending the VFM, so that we know those FVMs will have to be recomputed.

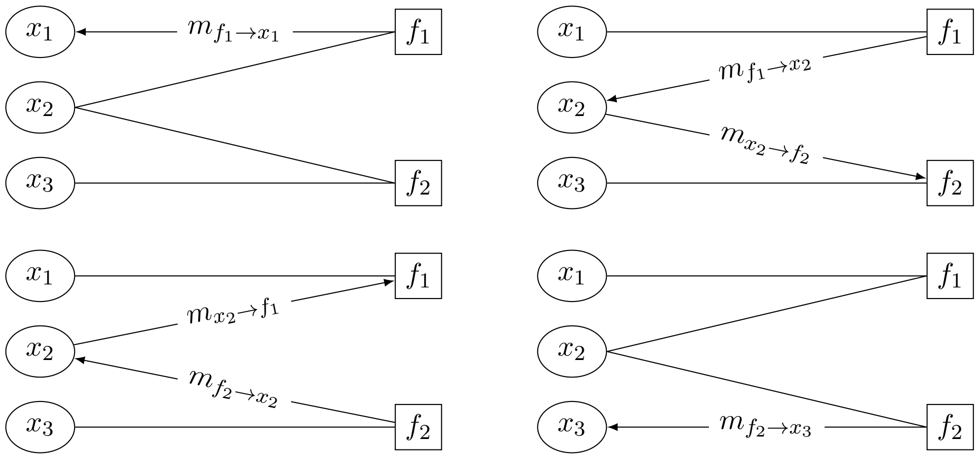


Figure 3.3: Example of FVM-focused scheduling on a simple FG

### Reordering the FVMs computation

By following this same idea of always integrating the most changes at each FVM computation, we implemented 3 different ways of choosing which FVM to compute at each step:

1. **Number of changed inputs:** we choose to propagate the FVM which are marked as outdated

by more of its variable nodes. If there are several candidates, we take one of them arbitrarily (the first one by index for example).

2. **Sum of the changes:** Instead of simply indicating 'outdated', the VFM computation results in a number representing how much it changed from its previous value. In our case, this number is the number of elements that have been removed from the set of possible values. Then we keep for each FVM the sum of all these number of changes and we choose the one with the highest value. We still break ties arbitrarily (e.g., by index).
3. **A mix of both:** We combine the two previous techniques: we first sort by number of changed inputs, then we break ties with the sum of changes and then arbitrarily (e.g., by index).

## 3.4 Results

In order to evaluate the efficiency of the different implementations and schedulers, we implemented an operation counter that was able to count the number of additions, multiplications and message computations in each direction. Hence, the resulting statistics doesn't depend on the machine running the tests. The input data sets was taken from online databases of known hard Sudokus for $9 \times 9$[2, 1] and $16 \times 16$[2] grids; the $4 \times 4$ grids was extracted from an online Sudoku generator[3].

### 3.4.1 Comparing different implementations

We compared the original implementation (Section 3.3.1), the 'set-formula' implementation (Section 3.3.2) and the set-formula implementation with message computation skipping (Section 3.3.3).

The original implementation could only be ran over small $4 \times 4$ Sudoku grids, any larger grid took too much resources to finish properly.

The results for the different grid sizes are shown in figure 3.4 (whiskers represent minimum and maximum values)

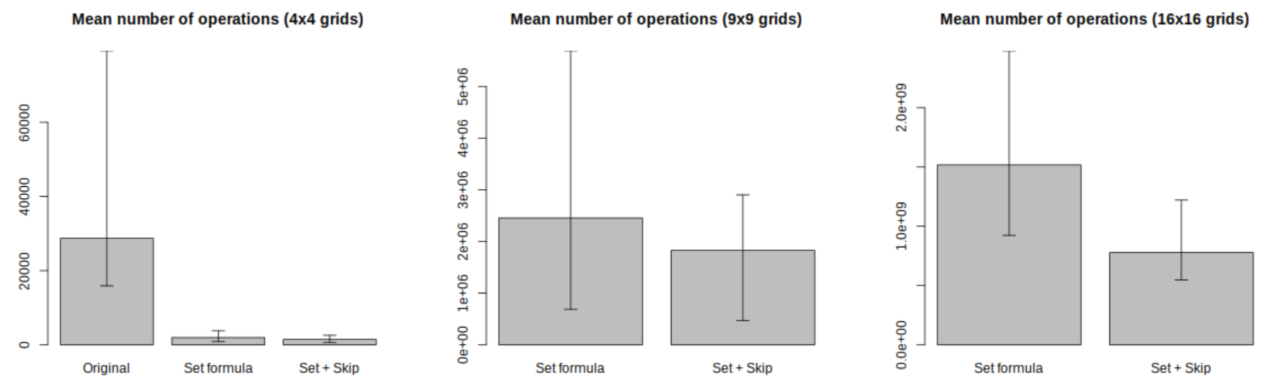


Figure 3.4: Comparison of the different implementations

Theses results confirm our expectations.

### 3.4.2 Comparing different schedulers

We now use our best implementation to compare the different message schedulers (Section 3.3.3). Results are shown in figure 3.5

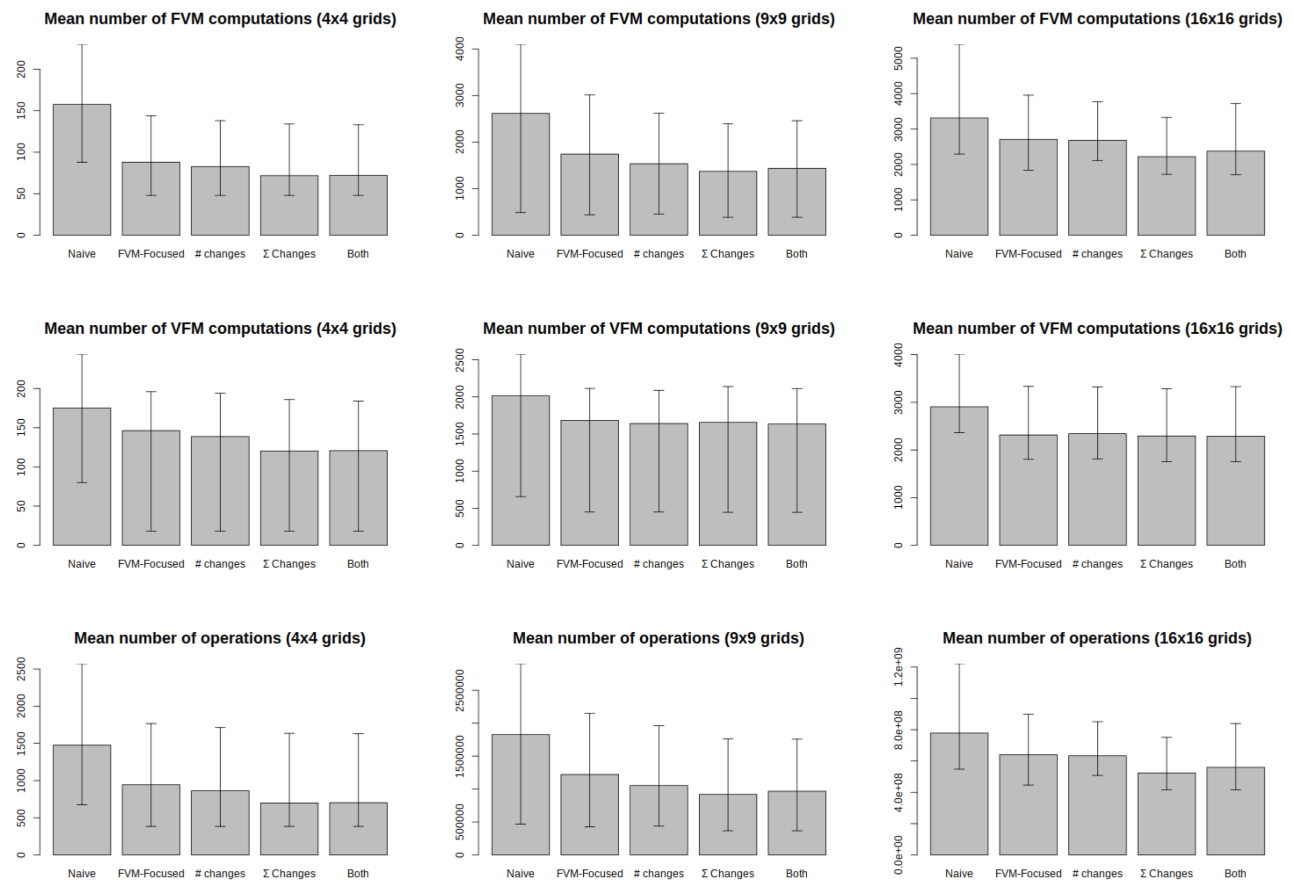


Figure 3.5: Comparison of the different schedulers

These results show that the FVM-focused approach provides better overall performances (in term of number of operations), but also that it does not even involve more VFM computations. Moreover, we can see that the most efficient scheduler is the one based on sum of all changes.

We can see for example that for the 16x16 grids, the "naive" scheduler takes an average of around 780 000 000 operations (additions + multiplications) to complete whereas the "Sum of changes" scheduler only takes about 525 000 000 operations, which represents a reduction of almost $1/3$ of the number of required operations.

# Chapter 4

# Conclusions

The general definition of the SPA leaves a lot of space to implementation choices, especially regarding the scheduling of the different message computations. And we were indeed able to come up with different implementations leading to different performances.

After analyzing the different results, it came out that we were right in assuming that we could achieve better performances by always computing messages with input that changed the most. Indeed, our best scheduler ("$\Sigma$ changes") is the one that take into account the most information from a message inputs.

This is an interesting result as it is not bound in any way to the problem we were looking at and can be applied to any other SPA instance.

## Resources

The source code of the implementation and of this document are available online at `http://sudoku.cat/`.

A live demo of the solver can be accessed via the telnet protocol at sudoku.cat on port 23 by running the command `telnet sudoku.cat`.

# Bibliography

[1] Gordon royle's minimum sudokus. `http://staffhome.ecm.uwa.edu.au/~00013890/sudokumin.php`. Accessed: 2015-07-30.

[2] Guenter stertenbrink and jean charles meyrignac top sudoku website. `http://magictour.free.fr/sudoku.htm`. Accessed: 2015-07-30.

[3] Sudoku generator. `http://www.sudokuweb.org/`. Accessed: 2015-07-30.

[4] Caroline Atkins and Jossy Sayir. Density evolution for sudoku codes on the erasure channel. In *Turbo Codes and Iterative Information Processing (ISTC), 2014 8th International Symposium on*, pages 233–237. IEEE, 2014.

[5] Andres I Vila Casado, Miguel Griot, Richard D Wesel, et al. Ldpc decoders with informed dynamic scheduling. *IEEE Transactions on communications*, 58(12):3470–3479, 2010.

[6] Frank R Kschischang. Codes defined on graphs. *Communications Magazine, IEEE*, 41(8):118–125, 2003.

[7] Frank R Kschischang, Brendan J Frey, and H-A Loeliger. Factor graphs and the sum-product algorithm. *Information Theory, IEEE Transactions on*, 47(2):498–519, 2001.

[8] H-A Loeliger. An introduction to factor graphs. *Signal Processing Magazine, IEEE*, 21(1):28–41, 2004.

[9] H-A Loeliger, Justin Dauwels, Junli Hu, Sascha Korl, Li Ping, and Frank R Kschischang. The factor graph approach to model-based signal processing. *Proceedings of the IEEE*, 95(6):1295–1322, 2007.

[10] Todd K Moon and Jacob H Gunther. Multiple constraint satisfaction by belief propagation: An example using sudoku. In *Adaptive and Learning Systems, 2006 IEEE Mountain Workshop on*, pages 122–126. IEEE, 2006.

[11] Jossy Sayir and Joned Sarwar. An investigation of sudoku-inspired non-linear codes with local constraints. *arXiv preprint arXiv:1504.03946*, 2015.