Application of Belief Propagation Algorithms on Factor Graphs: from Sudoku solving to LDPC decoding

Francesco Porcu

September 22, 2015

Contents

List of Figures **iii**

List of Tables **v**

List of Acronyms **vii**

List of Symobols and Operators **ix**

Introduction **1**

Outline 3

Introduzione **5**

Outline 7

1 Belief Propagation on Factor Graphs **9**

1.1 Introduction to Belief Propagation and Factor Graphs 9

1.2 Factor Graphs 9

1.3 Message Passing 12

2 Solving Sudoku using Belief Propagation Algorithms on Factor Graphs **17**

2.0.1 Historical notes 17

2.0.2 Related Works 18

2.1 Sudoku with Factor Graphs 18

2.1.1 Belief Propagation Formulation 22

2.2 Solving Sudoku using Moon Algorithm 25

2.2.1 Constraint to Variable Message 25

2.2.2 Variable to Constraint Message 28

2.3 Solving Sudoku using a different Algorithm 28

2.3.1 Program Analysis 29

2.3.2 Constraint Function: Methodologies For Solutions 30

2.3.3 Variable Function 37

2.3.4 Message-Passing Schedules 37

2.4 Conclusion and Results for Sudoku Sokving 42

2.4.1 Stopping set in Belief propagation 42

2.4.2 Equivalence of the Algorithms 43

2.4.3 Complexity of the Algorithms 45

2.4.4 Conclusions 48

3 Solving Low-Density Parity-Check Codes with Belief Propagation Algorithms on Binary Erasure Channel 51

3.1 Introduction to LDPC codes 51

3.2 Low Density Parity-check Code (LDPC) 52

3.2.1 Transmission Through a Gaussian Channel 54

3.2.2 Hard-decision Decoder 55

3.2.3 Hard-decision Encoder 56

3.3 LDPC Codes Over Binary Erasure Channels 57

3.4 Conclusion and Results for LDPC Codes 60

4 Conclusions and perspectives 69

Bibliography 71

Analytic Index 73

List of Figures

1.1 Different Examples of Factor Graphs. 11

1.2 Local substitutions that transform a rooted cycle-free factor graph to an expression tree for a marginal function at (a) a variable node and (b) a factor node. 12

1.3 A factor-graph fragment, showing the update rules of the sum-product algorithm. 13

2.1 Figures relative to a sudoku 4*4. 19

2.2 Sudoku Puzzle 9*9 20

2.3 Sudoku-Hex Puzzle 16*16 20

2.4 Tanner graph associated with Sudoku puzzle 9*9 21

2.5 Moon Algorithm for a Sudoku puzzle 9*9 27

2.6 Naked Single method for a Sudoku puzzle 9*9 31

2.7 Hidden Single method for a Sudoku puzzle 9*9 32

2.8 Naked Single method for a Sudoku puzzle 9*9 33

2.9 Different Check Node 1 and 2 40

2.10 Different Check Node 1 and 2 40

2.11 Stopping Set in Sudoku 43

2.12 Stopping Set in Sudoku 44

2.13 Invalid Permutation for A Sudoku 4*4. 44

2.14 Success solving rate of the algorithms depending on the level of sudokus. Evaluated using 30 sudokus of each level. 48

3.1 Belief propagation example code. 55

3.2 Binary channel. 58

3.3 Parity Equations for a LDPC code over BEC. 59

3.4 Additive white Gaussian noise channel AWGN 60

3.5 Graph of BER. 60

3.6 Graph of WER. 61

3.7 Range of Erasures. 61
3.8 Success Percentage. 62
3.9 BER and WER Vs. EbNo for a Threshold of 0.4. 62
3.10 BER and WER Vs. EbNo for a Threshold of 0.5. 63
3.11 BER and WER Vs. EbNo for a Threshold of 0.6. 63
3.12 BER and WER Vs. EbNo for a Threshold of 0.7. 64
3.13 BER and WER Vs. EbNo for a Threshold of 0.8. 64
3.14 BER and WER Vs. EbNo for a Threshold of 0.9. 65
3.15 BER and WER Vs. EbNo for a Threshold of 1.0. 65
3.16 BER and WER Vs. EbNo for a Threshold of 1.1. 66
3.17 BER and WER Vs. EbNo for a Threshold of 1.2. 66
3.18 BER and WER Vs. EbNo for a Threshold of 1.3. 67
3.19 BER and WER Vs. EbNo for a Threshold of 1.4. 67
3.20 BER and WER Vs. EbNo for a Threshold of 1.5. 68

List of Tables

2.1 Complexity at the Constraint Node. 24
2.2 Scheduling Average times. 42
2.3 Confront of the Time Duration. 47
2.4 Numbers for Sudoku Difficult. 48

3.1 Check nodes activities for Hard-Decision Decoder for code of Fig. 3.1 57
3.2 Message nodes decision for Hard-Decision Decoder for code of Fig. 3.1 57

List of Acronyms

AI	Artificial Intelligence
a.s.	almost surely
AS	Adaptive Schedule
AWGN	additive white Gaussian noise
BCJR	Bahl, Cocke, Jelinek and Raviv
BEC	Binary-input Erasure Channel
BAWGN	Binary additive white Gaussian noise
BSC	Binary Symmetric Channel
BER	bit error rate
BG	Bipartire graph
BP	Belief propagation
BPSK	Binary phase-shift keyed
CN	Constraint Nodes
CS	Configuration Space
CSP	Constraint Satisfaction Problem
EP	Erasure probability
EU	European Union
FG	Factor Graphs
GC	Gaussian channel
GM	Graphical Model
HD	Hard Decoder
HS	Hidden-single Algorithm
HW	Hamming weight
i.i.d.	independent and identically distributed
LDPC	Low-density Parity-check

MF	Marginal Function
MLCC	Multiple Local Combinatorial Constraint
MP	Max-product
MV	Message vector
NAS	Non Adaptive Schedule
NP	Nondeterministic Polynomial
NS	Naked-single algorithm
PCM	Parity check matrix
PP	Probabilistic Propagation
R	Real Numbers
RV	random variable
SD	Soft Decoder
SP	Sum-product
TC	Turbo Codes
TG	Tanner Graphs
US	United States
VA	Viterbi Algorithm
VN	Variable Nodes
WER	Word error rate

List of Symobols and Operators

cross product	$\times$
Hadamard (element-wise) product	$\circ$
floor function (rounding downward)	$\lfloor\cdot\rfloor$
absolute value	$\lvert\cdot\rvert$
vector norm	$\lvert\lvert\cdot\rvert\rvert$
conjugate transposition	$(\cdot)^H$
transposition	$(\cdot)^T$
Sum of the argument x	$\sum_x$
Product of the argument x	$\prod_x$
Set of real numbers	$\mathbb{R}$
Probability of erasure	ε
Energy per Bit	E_b
Energy per bit to noise power spectral density ratio	E_b/N_0
Energy per Coded Bit	E_c
Probability of erasure	ϵ
Permutations of cardinality i	c^i
Constraint function for the node m	C_m
Cells that partecipate in constraint m	N_m
Cells that partecipate in constraint m except for the cell n	N_{mn}
Probabilistic vector for the Variable n	p_n
Message from variable node n to constraint node m	q_{nm}
Decision valuebased upon the message received by the variable node n	q_n

Cell n of the factor Graph	S_n
Message vector from Constraint node m to variable node n	r_{mn}
Weight for the column c	w_c
Weight for the row r	w_r
Parity check for the constraint node m	z_m
Dimension of the Sudoku Puzzle	$N * N$
Variable nodes of a Factor Graph	$\{x_1, ..., x_{N*N}\}$
R-valued function of the variables x_i	$g(x_1, x_2, ...x_n)$
Marginal functions	$g_i(x_i)$
summary for x_i	$\sum_{\sim x_i}$
Variable x_i is function of the constraint node A	$f_A(x_i)$
Edge of the Factor Grapg	$e = \{x, f\}$
Message sent from node x to node f	$\mu_{x \to f}(x)$
Message sent from node f to node x	$\mu_{f \to x}(x)$
Set of neighbors of a given node v in a factor graph	$n(v)$
Marginal function for x obtained by multiplying together all the messages received by x	$\prod_{h \in n(x) \setminus \{f\}}$
Probability vector associated with cell S_n	$p_n = [P(S_n = 1)P(S_n = 2)...P(S_n = 9)]$
Probability that constraint C_m is satisfied when the cell S_n contains x	$P(C_m \mid S_n = x)$
Constraint nodes for variable n except m	$C_{m'},\ m' \in M_{nm}$
Variables nodes for the constraint node m except for n	$x_{n'}, n' \in N_{m,n}$
Rate of the code	$R = \dfrac{k}{n}$
Bit *1* sent from Constraint node i	$c_i \to 1$
Bit *1* received by Constraint node i	$1 \to c_i$
Bit sent from a variable node to the constraint i	$f_i \to 1$
Capacity of the binary Erasure Channel	$1 - \varepsilon$

Introduction

The *Belief Propagation* (BP) algorithm (Pearl 1988) is a sum-product message passing algorithm. The problem needs to be modeled by a *Bipartite Graph* called *Factor Graphs* (FG), which is a graph with two types of nodes : *Variable Nodes* to compute, and *Constraint Nodes* that represent the system properties. Then, by passing messages recursively between each connected nodes, the graph reaches a stationary (or nearly stationary) state where the variable nodes have the values that solve the system.

The presence of short cycles in the graph creates biases so that not every puzzle is solved by this method. However, all puzzles are at least partly solved by this method. The *Sudoku* application thus demonstrates the potential effectiveness of Belief Propagation algorithms on a general class of multiple constraint satisfaction problems.

Factor Graphs are a straightforward generalization of the *Tanner graphs* of Wiberg. Tanner introduced BG to describe families of codes which are generalizations of the *low-density parity-check* (LDPC) codes of Gallager [8].

The origins of Factor Graphs lie in coding theory, but they offer an attractive notation for a wide variety of *signal processing* problems. In particular, a large number of practical algorithms for a wide variety of detection and estimation problems can be derived as summary propagation algorithms. The algorithms derived in this way often include the best previously known algorithms as special cases or as obvious approximations.

Belief Propagation algorithms are also the means by which LDPC codes are decoded. In LDPC decoding, information about received bits that is implied collectively by the set of parity constraints is combined together in a (nearly) *Bayesian* way with information from the received data to provide information about the bits that were originally transmitted.

In applying Belief Propagation methods, the problem is mapped to a graph and messages representing Bayes probabilities are passed among the nodes of the graph.

Belief Propagation is *Bayesian optimal* for graphs without cycles, but suffers from *biases* for graphs with cycles. The graph associated with Sudoku, like the graphs for LDPC codes, does have cycles. Every node in the Sudoku graph lies on two cycles of length four. The biases introduced by these short cycles cause failure of the Belief Propagation method for more difficult puzzles due to the existence of *stopping-sets* where the algorithm shucked before the puzzle is completely solved.

The application reveals the possibility of applying Belief Propagation techniques to multiconstraint problems, at least to eliminate many of the possibilities, perhaps leaving the problem suficiently small that a global search may be possible.

The two main summary propagation algorithms are the *sum-product* (or Belief Propagation or *probability propagation*) algorithm and the *max-product* (or *min-sum*) algorithm, both of which have a long history.

In the context of error correcting codes, the sum-product algorithm was invented by Gallager as a decoding algorithm for LDPC codes; it is still the standard decoding algorithm for such codes. The full potential of LDPC codes was not yet realized at that time. Tanner explicitly introduced graphs to describe LDPC codes, generalized them (by replacing the parity checks with more general component codes), and introduced the min-sum algorithm.

A surprisingly wide variety of algorithms developed in the artificial intelligence, signal processing, and digital communications communities may be derived as specific instances of the sum-product algorithm, operating in an appropriately chosen factor graph.

In the first part of my thesis, we use the Belief Propagation paradigm to solve a problem with multiple local combinatorial constraints, namely, the popular Sudoku puzzle. The Belief Propagation method is very general and does not require any human insight or tricks, nor does it require building solution trees. It is thus potentially applicable to a broad variety of problems as a general tool. Easy Sudoku puzzles can be solved by simple elimination but difficult Sudoku puzzles are actually *NP-complete*.

In the second part, we use the BP paradigm to decode the LDPC codes over a *Binary Erasure Channel* (BEC) with the technique of the *Hard Decoder*. The belief propagation decoder for erasure channels operates by exchanging messages containing sets of possible bits. We use this section to show the analogy between SUDOKU puzzles and LDPC. Both can be represented by a factor graph, where the constraints for LDPC codes are linear, i.e. $\sum c_i * x_i = 0$, where the coefficients and sum are defined over a finite field, while for SUDOKU the constraints are non linear, requiring all variables in a constraint to have different values within a finite alphabet.

In this work we will implement different algorithms to solve the SUDOKU puzzle with Belief

Propagation; we will analyse the performances of every algorithm and we will present the advantages and the limitations of every approach. We will try to apply some improvements in order to have a better performance. Finally we will compare the SUDOKU puzzle algorithm with the LDPC algorithm, but just from a qualitative point of view, in order to see the common problems.

Outline

The remainder of this thesis is structured as follows.

In Chapter 1, we introduce the fundamental knowledges of Belief propagation. In particular, we outline the basic concepts of Belief Propagation, message passing, Factor Graph and Sum-product algorithm.

In Chapter 2 we introduce the mathematical notions about Sudoku and how we can associate this game with Belief Propagation and Factor Graph 2.1. It follow the explaination of the algorithm proposed by T.K.Moon in the Sec. 2.2, the explaination of my version of the algorithm in Sec. 2.3 and finally the conclusions with all the results and considerations about the Sudoku Solving algorithm in Sec. 2.4.

In Chapter 3 we talk about Low-Density Parity-Check Codes; first of all we introdece them from an historical point of view. In the Sec. 3.2 we can see an explaination more accurate about the algorithm and how it works. To follow, we find in Sec. 3.3 the mode of operation of the LDPC Codes over the Binary Erasure Channel that is the method debated in this chapter. Finally in Sec. 3.4 we can read the conclusions and the results for this argument.

To conclude in Chapter 4, we draw some final considerations for this thesis and the different studied algorithms.

Introduzione

L'algoritmo di *Belief Propagation* (BP) (Pearl 1988) è un algoritmo di *message passing.* Il problema viene modellato per mezzo di un *grafo Bipartito* chiamato *Factor Graph* (FG), che è un grafo con due tipi di nodi: *nodi variabile* e *nodi vincolo*; essi rappresentano le proprietà di sistema. Passando i messaggi ricorsivamente tra i nodi connessi, il grafo raggiunge uno stato stazionario (o quasi stazionario) dove i nodi variabile contengono i valori che risolvono il sistema.

La presenza di brevi cicli nel grafo crea distorsioni e non tutti i puzzle vengono risolti da questo metodo. Tuttavia, tutti i puzzle vengonoo almeno in parte semplificati con questo metodo. Con l'applicazione dell'algoritmo al *Sudoku* si vuole dimostrare la potenziale efficacia degli algoritmi di Belief Propagation in una classe pi generale di problemi di soddisfazione dei vincoli.

Il Factor Graph è una generalizzazione del *Tanner Graph* di Wiberg. Tanner ha introdotto i BG per descrivere famiglie di codici che sono generalizzazioni dei codici *Low-density parity-check* (LDPC) di Gallager [8].

Le origini dei Factor Graphs risiedono nella teoria dei codici, ma offrono una notazione interessante per un un'ampia varietà di problemi di elaborazione dei segnali. In particolare, un gran numero di algoritmi pratici per una grande varietà di problemi di rilevamento e di stima possono essere derivati; tra questi per esempio algoritmi di sintesi di propagazione. Gli algoritmi derivati in questo modo includono spesso i migliori algoritmi precedentemente noti come casi speciali o come evidenti approssimazioni.

Algoritmi di Belief Propagation sono anche i mezzi attraverso i quali i codici LDPC vengono decodificati. Durante la decodifica LDPC, le informazioni sui bit ricevuti che sono implicite nei vincoli di parità sono combinate insieme, in modo *Bayesiano* (o quasi), alle informazioni dai dati ricevuti per fornire informazioni sui bit che sono stati originariamente trasmessi.

Nell'applicazione dei metodi di Belief Propagation, il problema è associato a un grafo e i messaggi che rappresentano le probabilità *bayesiane* sono passati tra i nodi del grafo.

Belief Propagation è *bayesiano* ottimale per i grafi senza cicli, ma soffre di errori in quelli con cicli. Il grafo associato al Sudoku, come anche per i codici LDPC, ha vari cicli. Ogni nodo del grafo Sudoku si trova su due cicli di lunghezza *quattro*. Le distorsioni introdotte da queste brevi cicli causano il fallimento dell'algoritmo di Belief Propagation per puzzle più difficili a causa dell'esistenza di *stopping-set* dove l'algoritmo si blocca prima che il puzzle sia completamente risolto.

L'applicazione mostra la possibilità di applicare le tecniche di Belief Propagation a problemi *multiconstraint*, almeno per eliminare molte delle possibilità, lasciando il problema in una situazione più semplificata in cui una ricerca globale pu essere possibile.

I due algoritmi di sintesi di propagazione principali sono il *Sum-product* (detto anche algoritmo di Belief Propagation o di *Probability Propagation*) e l'algoritmo *max-product* (anche detto *min-sum*); entrambi i quali hanno una lunga storia.

Nel contesto degli algoritmi di *error correcting codes*, l'algoritmo di *sum-product* è stato inventato da Gallager come algoritmo di decodifica dei codici LDPC; ed ancora è lo standard di decodifica di tali codici. L'intero potenziale di codici LDPC non era ancora stato scoperto in quel momento. Tanner esplicitamente introdusse i grafi per descrivere i codici LDPC, li generalizzò (sostituendo i *parity check codes* con codici pi generale), introducendo anche l'algoritmo *min-sum*.

Una varietà sorprendente di algoritmi che venne sviluppata nell'ambito dell'intelligenza artificiale, elaborazione del segnale, e comunicazione digitale possono essere derivati come specifiche istanze dell'algoritmo sum-product, che operano in un Factor Graph opportunamente scelto.

Nella prima parte della mia tesi, mostrerò il paradigma di Belief Propagation di risolvere un problema con vincoli combinatoriali locali, vale a dire, il popolare puzzle di Sudoku.

Il metodo di Belief Propagation è molto generale e non richiede alcuna conoscenza o trucchi umani. è quindi potenzialmente applicabile ad un'ampia varietà di problemi, come strumento generale. I puzzle più semplici possono essere risolti con una semplice eliminazione logica ma quelli più difficili sono in realtà un problema *NP-completo.*

Nella seconda parte della mia tesi, usiamo il paradigma di Belief Propagation per decodificare i codici LDPC su un *Binary Erasure Channel* (BEC), con la tecnica del *Hard Decoder.* Il Belief Propagation decoder per gli *erasure channel* opera attraverso lo scambio di messaggi contenenti i set dei possibili bit.

Nella sezione successiva visualizzeremo l'analogia tra i puzzle Sudoku e LDPC. Entrambi possono essere rappresentati da un Factor Graph dove per i codici LDPC abbiamo vincoli lineari, vale a dire $sumC_i * x_i = 0$, dove i coefficienti e somma sono definiti su un campo finito, mentre per il Sudoku abbiamo dei vincoli non lineari, ossia le variabile appartenenti a un certo vincolo possono avere valori diversi all'interno di un alfabeto finito.

In questo lavoro si realizzeranno diversi algoritmi per risolvere il puzzle Sudoku con Belief Propagation; analizzeremo le prestazioni di ogni algoritmo e verranno presentati i vantaggi ed i limiti di ogni approccio. Cercheremo di applicare alcuni miglioramenti al fine di avere migliori prestazioni. Infine metteremo a confronto l'algoritmo usato nel Puzzle Sudoku con l'algoritmo usato sui codici LDPC, ma solo da un punto di vista qualitativo, al fine di vedere i problemi e i pregi comuni.

Outline

Questa tesi è strutturata come segue.

Nel capitolo 1, vi presentiamo le conoscenze fondamentali dell'algotitmo di Belief Propagation. In particolare, si delineano i concetti di base della Belief Propagation, il message-passing, i Factor Graph e l'algoritmo sum-product.
Nel capitolo 2 introduciamo le nozioni matematiche che stanno alla base del Sudoku e come possiamo associare questo gioco con la Belief Propagation e i Factor Graphs nella sezione 2.1.

Seguono la spiegazione dell'algoritmo proposto da T.K. Moon nella sezione 2.2, e della mia versione dell'algoritmo in 2.3. Per finire questo capitolo troviamo le conclusioni con tutti i risultati e le considerazioni circa l'algoritmo di *Sudoku Solving* in sezione 2.4.
Nel capitolo 3 si parla dei codici LDPC; prima di tutto li introdurremo da un punto di vista storico. Nella sezione 3.2 avremo una spiegazione più precisa circa il funzionamento di questi codici. A seguire, troviamo in 3.3 la modalità di funzionamento dei codici LDPC applicati ai *Binary Erasure Channels*, che è il metodo discusso in questo capitolo. Infine nel paragarfo 3.4 troveremo conclusioni e risultati di questo argomento.
Nel capitolo finale (4), leggeremo alcune considerazioni finali sul lavoro svolto in questa tesi e sui diversi algoritmi studiati.

Chapter 1

Belief Propagation on Factor Graphs

1.1 Introduction to Belief Propagation and Factor Graphs

A large variety of algorithms in coding, signal processing, and artificial intelligence may be viewed as instances of the *summary-product* algorithm (or *belief/probability propagation* algorithm), which operates by message passing in a graphical model.

The two main summary propagation algorithms are the *sum-product* (or belief propagation or probability propagation) algorithm and the *max-product* (or *min-sum*) algorithm.

In the context of error correcting codes, the sum-product algorithm was invented by Gallager [8] as a decoding algorithm for *low-density parity check* (LDPC) codes;

Tanner [15] explicitly introduced graphs to describe LDPC codes, generalized them (by replacing the parity checks with more general component codes), and introduced the *min-sum* algorithm.

The full power of iterative decoding was only realized by the breakthrough invention of turbo coding by Berrou *et al.* . [4], which was followed by the rediscovery of LDPC codes [10]. Wiberg *et al.* [16, 17] observed that the decoding of turbo codes and LDPC codes as well as the *Viterbi* and *BCJR* algorithms [3] are instances of one single algorithm, which operates by message passing in a generalized *Tanner graph*. The later introduction of factor graphs [7, 9] may be viewed as a further elaboration of the ideas by Wiberg *et al.* .

1.2 Factor Graphs

First of all, the belief propagation algorithm apply only to a factor graph. A *Factor Graph* is a *bipartite graph* representing the factorization of a function [18]. Graphs not only describe the codes, but, more important, they structure the operation of the sumproduct decoding algorithm (or one of many possible variations), which can be used for iterative decoding.

In probability theory and its applications, factor graphs are used to represent factorization of a *probability distribution function* , enabling efficient computations, such as the computation of *marginal distributions* through the *sum-product* algorithm. One of the important success stories of Factor Graphs and the *sum-product* algorithm is the decoding of capacity-approaching error-correcting codes, such as LDPC and *turbo codes*.

Let $x_1, x_2, ...x_n$, be a collection of variables, in which, for each $\{i, x_i\}$, takes on values in some (usually finite) domain (or alphabet) A_i. Let $g(x_1, x_2, ...x_n)$ be an $R-valued$ function of these variables, i.e., a function with domain $S = A_1 * A_2 * ... * A_n$ and codomain R. The domain of S is called the *configuration space* for the given collection of variables, and each element of S is a particular configuration of the variables, i.e., an assignment of a value to each variable.

The codomain R of g may in general be any semiring; however we will lose nothing essential by assuming that is the set of real numbers.

Assuming that summation in R is well defined, then associated with every function are *marginal functions* $g_i(x_i)$. For each $a \in A_i$, the value of $g_i(a)$ is obtained by summing the value of $g(x_1, x_2, ...x_n)$ over all configurations of the variables that have $x_i = a$.

This type of sum is central to this work and we now see a nonstandard notation to handle it: the *not-sum* or *summary.* Instead of indicating the variables being summed over, we indicate those variables not being summed over. For example, if h is a function of three variables, x_1, x_2, x_3, then the summary for x_2 is denoted by

$$\sum_{\sim x_2} h(x_1, x_2, x_3) := \sum_{x_1 \in A_1} \sum_{x_3 \in A_3} h(x_1, x_2, x_3) \tag{1.1}$$

In this notation we have

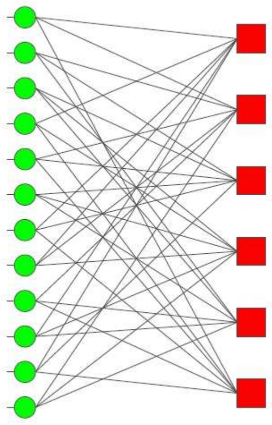
$$g_i(x_i) := \sum_{\sim x_i} g(x_1, ..., x_n) \tag{1.2}$$

i.e., the i-th *marginal function* associated with $g(x_1, x_2, ...x_n)$ is the summary for x_i of g.

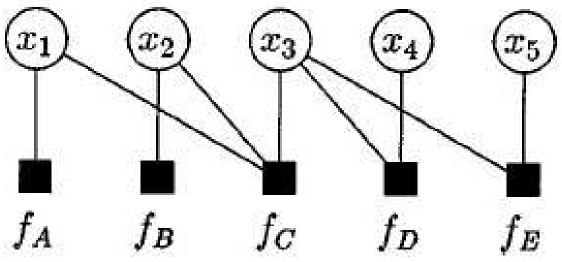
We are interested in developing efficient procedures for computing marginal functions that exploit the way in which the global function factors, using the distributive law to simplify the summations, and reuses intermediate values (partial sums). As we will see, such procedures can be expressed very naturally by use of a factor graph.

Suppose that $g(x_1, x_2, ...x_n)$ factors into a product of several *local functions*, each having some subset of $x_1, x_2, ...x_n$ as arguments;

i.e., suppose that



(a) *Factor Graph with Variable Nodes (green) and Constraint Nodes (red).*



(b) *A Factor Graph for the product* $f_A(x_1) * f_B(x_2) * f_C(x_1, x_2, x_3) * f_D(x_3, x_4) * f_E(x_3, x_5)$.

Figure 1.1: *Different Examples of Factor Graphs.*

$$g(x_1, x_2, ...x_n) = \prod_{j \in J} f_j(X_j) \tag{1.3}$$

where J is a discrete index set, X_j is a subset of $x_1, ...x_n$, and $f_j(X_j)$ is a function having the elements of X_j as arguments.

Definition: A factor graph is a bipartite graph that expresses the structure of the factorization 1.3. A factor graph has a variable node for each variable x_i, a factor node for each local function f_i, and an edge-connecting variable node x_i to factor node f_i if and only if x_i is an argument of f_i.

A Factor Graph is thus a standard *bipartite graphical representation* of a mathematical relation; in this case, the "*is an argument of*" relation between variables and local functions.

We can see an example of Factor Graph in the Figure **1.1 (a)**.

Example 1 (A Simple Factor Graph): Let $g(x_1, x_2, x_3, x_4, x_5)$ be a function of five variables, and suppose that g can be expressed as a product:

$$g(x_1, x_2, x_3, x_4, x_5) = f_A(x_1) * f_B(x_2) * f_C(x_1, x_2, x_3) * f_D(x_3, x_4) * f_E(x_3, x_5) \tag{1.4}$$

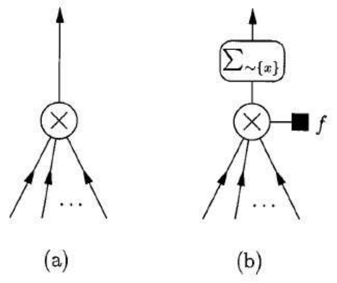


Figure 1.2: *Local substitutions that transform a rooted cycle-free factor graph to an expression tree for a marginal function at (a) a variable node and (b) a factor node.*

of five factors, so that ,$f_A(x_1) * f_B(x_2) * f_C(x_1, x_2, x_3) * f_D(x_3, x_4) * f_E(x_3, x_5)$. The Factor Graph that corresponds to 1.4 is shown in figure **1.1 (b)**.

The goal of the decomposition in two different types of nodes, is to divide a function, representing a complex system, which is difficult to compute, into smaller functions which only depend on a few variables and which can be more easily computed.

1.3 Message Passing

To better understand the *sum-product algorithm* we now see a *message-passing* algorithm: the *single-i sum-product* algorithm, since it computes, for a single value of i the marginal function $g_i(x_i)$ in a rooted cycle-free factor graph, with x_i taken as root vertex.

The computation begins at the leaves of the Factor Graph. Each leaf variable node sends a trivial *identity function* message to its parent, and each leaf factor node f sends a description of f to its parent. Each vertex waits for messages from all of its children before computing the message to be sent to its parent.

This computation is performed according to the transformation shown in Fig. 1.2; i.e., a variable node simply sends the product of messages received from its children, while a factor node with parent forms the product of with the messages received from its children, and then operates on the result with a $\sum\limits_{\sim x}$ summary operator. By a *product of messages* we mean an appropriate description of the (pointwise) product of the corresponding functions.

If the messages are parametrizations of the functions, then the resulting message is the parametrization of the product function, not (necessarily) literally the numerical product of the messages. Similarly, the summary operator is applied to the functions, not necessarily

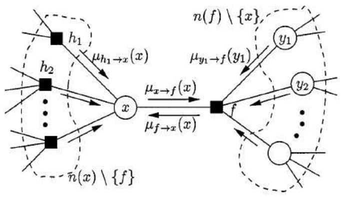


Figure 1.3: *A factor-graph fragment, showing the update rules of the sum-product algorithm.*

literally to the messages themselves.

The computation terminates at the root node x_i, where the marginal function $g_i(x_i)$ is obtained as the product of all messages received at x_i.

It is important to note that a message passed on the edge $\{x, f\}$, either from variable x to factor f, or vice versa, is a single-argument function of x, the variable associated with the given edge. This follows since, at every factor node, summary operations are always performed for the variable associated with the edge on which the message is passed. Likewise, at a variable node, all messages are functions of that variable, and so is any product of these messages.

The message passed on an edge during the operation of the single- sum-product algorithm can be interpreted as follows. If $e = \{x, f\}$ is an edge in the tree, where x is a variable node and f is a factor node, then the message passed on e during the operation of the sum-product algorithm is simply a summary for x of the product of the local functions descending from the vertex that originates the message.

In many circumstances, we may be interested in computing $g_i(x_i)$ for more than one value of i. Such a computation might be accomplished by applying the single- algorithm separately for each desired value of i, but this approach is unlikely to be efficient, since many of the sub-computations performed for different values of will be the same. Computation of $g_i(x_i)$ for all i simultaneously can be efficiently accomplished by essentially *overlaying* on a single FG all possible instances of the single algorithm. No particular vertex is taken as a root vertex, so there is no fixed parent/child relationship among neighboring vertices.

As in the single algorithm, message passing is initiated at the leaves. Each vertex remains idle until messages have arrived on all but one of the edges incident on v. Just as in the single-i algorithm, once these messages have arrived, v is able to compute a message to be sent on the one remaining edge to its neighbor (temporarily regarded as the parent), just as in the single-i algorithm, i.e., according to Fig. 1.2. Let us denote this temporary parent as vertex w. After sending a message to w, vertex v returns to the idle state, waiting for a return message to arrive from w. Once this message has arrived, the vertex is able to compute and send messages to each of its neighbors (other than w), each being regarded, in turn, as a parent.

The algorithm terminates once two messages have been passed over every edge, one in each direction. At variable node x_i, the product of all incoming messages is the marginal function $g_i(x_i)$, just as in the single algorithm. Since this algorithm operates by computing various sums and products, we refer to it as the *sum-product* algorithm.

The sum-product algorithm operates according to the following simple rule:

The message sent from a node v on an edge e is the product of the local function at v (or the unit function if v is a variable node) with all messages received at v on edges other *than e, summarized for the variable associated with e.*

Let $\mu_{x\to f}(x)$ denote the message sent from node x to node f in the operation of the *sum-product* algorithm, let $\mu_{f\to x}(x)$ denote the message sent from node f to node x. Also, let $n(v)$ denote the set of neighbors of a given node v in a factor graph.

Then, as illustrated in Fig. 1.3, the message computations performed by the *sum-product* algorithm may be expressed as follows:

Variable to Constraint function

$$\mu_{x\to f}(x) = \prod_{h\in n(x)\setminus\{f\}} \mu_{h\to x}(x) \tag{1.5}$$

Constraint to Variable function

$$\mu_{f\to f}(x) = \sum_{\sim x}(f(X) * \prod_{y\in n(f)\setminus\{x\}} \mu_{y\to f}(y)) \tag{1.6}$$

where $X = n(f)$ is the set of arguments of the function f. In words, the update rule to evaluate *marginal function* at a variable node x_i is obtained by multiplying together all of the messages received at x_i (as we can see in Eq. 1.5) because there is no local function to include, and the summary for of a product of functions of is simply that product.

On the other hand, the update rule at a local function node given by Eq. 1.6 in general involves nontrivial function multiplications, followed by an application of the summary operator.

In a *cycle-free* factor graph, an outgoing message at any vertex v may be computed and sent on an edge e as soon as all of the information needed to compute that message (messages that arrive at v on edges other than e) is available at v. It follows that message-passing is initiated at the leaf vertices, since such vertices have all the required information at the very start.

The algorithm terminates once every variable vertex has received a message from each of its neighbors. In a finite cycle-free factor graph with E edges, the termination condition is achieved in no more than $2E$ steps (i.e., it is never necessary to send more than two message over an edge, one in each direction). The following theorem is proved in [10].

Theorem 1 In a finite cycle-free factor graph representing a function $g(x_1, ..., x_n)$, the function $\mu(x_i)$ computed by the sum-product algorithm according to Eq. 1.6 is the marginal function for x_i.

Unfortunately, the presence of cycles in the graph results in indefinite propagation of messages, resulting in an iterative algorithm with no natural termination. Even with cycles in their factor graphs, *turbo codes* and *LDPC* codes achieve a performance with *sum-product decoding* that comes to within a fraction of a decibel of the *Shannon limit* in *binary-input additive white Gaussian noise* (BIAWGN) channels. Careful optimization of the graph structure associated with ensembles of irregular LDPC codes has led to capacity-achieving performance on the *binary erasure channel*, and to performance that is practically indistinguishable from the Shannon limit on BIAWGN channels.

Chapter 2

Solving Sudoku using Belief Propagation Algorithms on Factor Graphs

Sudoku is a popular number puzzle. It is composed by a grid (usually 9x9) with some cells already filled. Here, we model the puzzle as a probabilistic graphical model and we use the *sum-product* message passing [11] to solve the puzzle. In addition, we propose a different Sudoku solver algorithm and we will show that with this algorithm we have an improvement of the performances.

This is possible because the new algorithm is more specific for the Sudoku Puzzle, so it is better on the performances but we can't use it for other kind of problems. Vice versa the first algorithm is worse talking about performances but it can be applied in more problems of the Belief propagation on Factor Graphs.

This chapter contains an introduction to the sudoku and a brief chronology of Sudoku Puzzle, given in Sect. 2.0.1; the basic concepts of Belief Propagation, Factor Graphs and Sum-Product Algorithm associated with Sudoku Solving are outlined in Sects. 2.1. Finally, Sect. 2.2 and 2.3 provides all the research material used during the course of the work and the final results with their respective conclusions.

2.0.1 Historical notes

Sudoku is a popular puzzle printed daily in newspapers all over the world. The aim of the most popular form is to fill a $9 * 9$ matrix of cells with digits from 1 through 9. Sudoku fans get puzzles not only from daily newspapers. Bookstores sell books with Sudoku puzzles, web sites offer Sudoku problems and it's possible play Sudoku on own mobile.

The first occurrence was in **Dell Pencil Puzzles & Word Games** in 1979.

In Sudoku the cells are arranged in nine rows and nine columns; they have to be filled with the numbers 1 to 9. Initially some cells are already filled in with numbers. These cells are called *givens.* The players aim is to fill in the remaining cells in such a way, that each column, each row, and each $3 * 3$ sub-square contains no figure more than once. That means, each column, each row, and each $3 * 3$ sub-square contains a permutation of the numbers 1 to 9. A well formed Sudoku has a unique solution.

For short: **the numbers must be single**, what is roughly the meaning of the Japanese name Sudoku. (Actually, its not important that Sudoku deals with numbers, in fact, any set of nine different symbols would do.)

Sudoku is not only a nice way to make the time pass more quickly, it also gives rise to a lot of interesting mathematical problems. For example: How many arrangements of the numbers 1 to 9 on a $9 * 9$ field do exist, that are compatible with the Sudoku constrains? What is the smallest number of givens for that one can construct well formed Sudoku instances?

Many games similar to Sudoku exist such as Sudoku X, Nonomino Sudoku, Killer Sudoku, Hyper Sudoku, Greater-Than Sudoku, Kakuro, Futoshiki, and KenKen.

2.0.2 Related Works

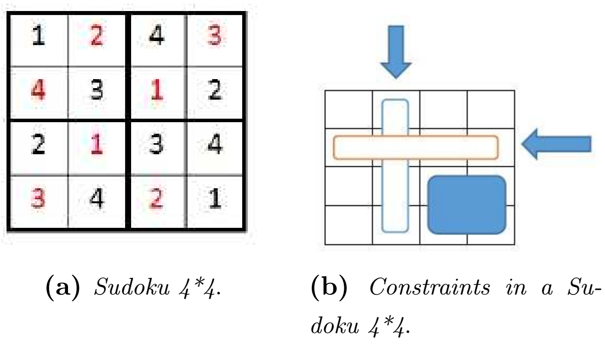
Sudoku puzzles provide an interesting playground for mathematics. It is typically treated as an instance of the *graph coloring problem* [5], or the quasi-group with holes problem [1]. In a recent work it was shown that Sudoku is NP-complete [19]. Also, Sudoku can be viewed as an LDPC decoding problem over an erasure channel [11]. Other industrial applications that can be modeled by Sudoku are shown in [6, 14].

There have been many reported algorithms for solving Sudoku. The seemingly most efficient is a search based solver presented by [12]. Although the algorithm can easily solve puzzles up to size 16x16, it fails on larger puzzles. The main reason for this is an exponential growth in the search space as the puzzle size increases.

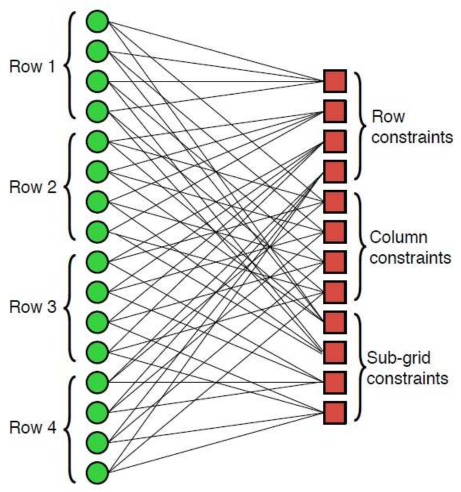
2.1 Sudoku with Factor Graphs

In this section we will describe how *Belief propagation* algorithms can be utilized to solve a *Sudoku puzzle*.

An $N * N$ Sudoku puzzle is a grid of cells partitioned into N smaller blocks of N elements. A solution to the puzzle involves filling in empty cells in the grid in a way that the numbers *1*



(a) *Sudoku 4*4.* (b) *Constraints in a Sudoku 4*4.*



(c) *Factor Graph for a Sudoku 4*4.*

Figure 2.1: *Figures relative to a sudoku 4*4.*

through N appear once on each row, each column, and each $\sqrt{N} * \sqrt{N}$ subgrid (all-different constraints).

We have $N * N$ cells to fill and we can associate the variable nodes of the Factor Graph with the $N * N$ entries, denoted by $\{x_1, ..., x_{N*N}\}$; for the Sudoku $4 * 4$ the last variable is x_{16}. Each variable can obtain a values from 1 to N.

In Figure 2.1 we can see: **(a)** an example of a $4 * 4$ Sudoku puzzle and its solution: the red numbers are the given numbers and the black are the numbers to be filled; in **(b)** we have the 3 different constraints in a Sudoku Puzzle; **(c)** show the Factor Graph for the Sudoku 4x4 with all the constraints. In the figures 2.2 and 2.3 we can see respectively an exampe of the classic Sudoku $9 * 9$ and the *Hexadecimal* Sudoku $16 * 16$.

The rule that each of the four numbers should appear exactly once in a row, column and sub-grid can be associate with a constraint node in the factor Graph; finally we have $N * 3$ total constraints: N for the rows, N for the columns and N for the sub-grid. We will

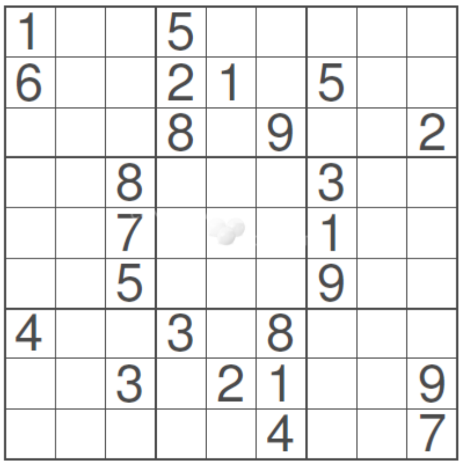


Figure 2.2: *Sudoku Puzzle 9*9*

denote the constraints as: C_m with $m \in \{1, ..., N * 3\}$.

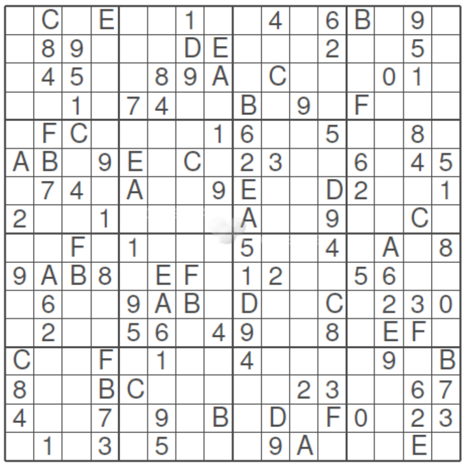


Figure 2.3: *Sudoku-Hex Puzzle 16*16*

Each constraint node is linked to N *cell nodes*, and the constraint function is that each cell node has a different value.

In order to start the algorithm, we need some cells already filled; this means that some variables have a unique value of the alphabet $\{1, ..., N\}$. There is a correlation between the number of given values and the difficulty level of the problem. At the moment, the smallest number of revealed values in a Sudoku puzzle that has a unique solution is *17*. There are no known *16-given* Sudoku examples that have a unique solution. Many examples of 17-given

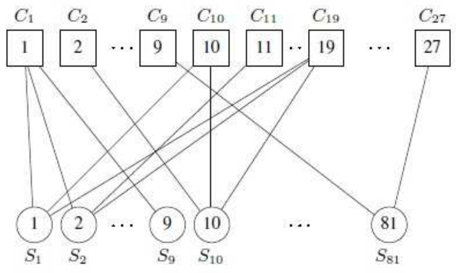


Figure 2.4: *Tanner graph associated with Sudoku puzzle 9*9*

Sudoku puzzles that has a unique solution were collected by Gordon Royle and can be found in his website [13].

The graph is initialized with the given number of the sudoku. Each Variable node contains the probability of taking each value, from 1 to N. If a cell has a given number X, then the probability of X is 1 and the probability of the other numbers is 0. Otherwise, if a cell does not have a given number, the probability of each value is $1/N$. The variable nodes send their probabilities to their constraint nodes. Then, each constraint node sends to its variable nodes the probability for them to take each value, knowing the probabilities sent by the $N-1$ other variable nodes to it. Then, each variable node sends again to each of his constraint node the probability of each state depending on the messages of its 2 other constraint nodes. And the computation keeps going until reaching a *stationary state* where the value of each cell node is the value that is the most probable taking its 3 constraint nodes into account.
A *constraint function* $C_m : \{1, ..., N\} \to 0, 1$ is defined as:

$$C_m(s_1, ..., s_N) = \begin{cases} 1 \; if \; \{s_1, ..., s_N\} \; are \; distinct \\ 0 \to otherwise \end{cases} \tag{2.1}$$

Let C_1 through C_N denote the constraints associated with the rows of the puzzle, C_{N+1} through C_{N*2} the constraints associated with the columns, and C_{N*2+1} through C_{N*3} the constraints associated with the $3*3$ sub-grid.

In figure 2.4 we can see the graph relative to a Sudoku Puzzle $9*9$. In the next matrix we can see the rapresentation of the *Tanner Graph* in form of matrix.

$$
Tanner\ Graph\ Matrix = \begin{bmatrix}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 \\
19 & 20 & 21 & 22 & 23 & 24 & 25 & 26 & 27 \\
28 & 29 & 30 & 31 & 32 & 33 & 34 & 35 & 36 \\
37 & 38 & 39 & 40 & 41 & 42 & 43 & 44 & 45 \\
46 & 47 & 48 & 49 & 50 & 51 & 52 & 53 & 54 \\
55 & 56 & 57 & 58 & 59 & 60 & 61 & 62 & 63 \\
64 & 65 & 66 & 67 & 68 & 69 & 70 & 71 & 72 \\
73 & 74 & 75 & 76 & 77 & 78 & 79 & 80 & 81 \\
1 & 10 & 19 & 28 & 37 & 46 & 55 & 64 & 73 \\
2 & 11 & 20 & 29 & 38 & 47 & 56 & 65 & 74 \\
3 & 12 & 21 & 30 & 39 & 48 & 57 & 66 & 75 \\
4 & 13 & 22 & 31 & 40 & 49 & 58 & 67 & 76 \\
5 & 14 & 23 & 32 & 41 & 50 & 59 & 68 & 77 \\
6 & 15 & 24 & 33 & 42 & 51 & 60 & 69 & 78 \\
7 & 16 & 25 & 34 & 43 & 52 & 61 & 70 & 79 \\
8 & 17 & 26 & 35 & 44 & 53 & 62 & 71 & 80 \\
9 & 18 & 27 & 36 & 45 & 54 & 63 & 72 & 81 \\
1 & 2 & 3 & 10 & 11 & 12 & 19 & 20 & 21 \\
4 & 5 & 6 & 13 & 14 & 15 & 22 & 23 & 24 \\
7 & 8 & 9 & 16 & 17 & 18 & 25 & 26 & 27 \\
28 & 29 & 30 & 37 & 38 & 39 & 46 & 47 & 48 \\
31 & 32 & 33 & 40 & 41 & 42 & 49 & 50 & 51 \\
34 & 35 & 36 & 43 & 44 & 45 & 52 & 53 & 54 \\
55 & 56 & 57 & 64 & 65 & 66 & 73 & 74 & 75 \\
58 & 59 & 60 & 67 & 68 & 69 & 76 & 77 & 78 \\
61 & 62 & 63 & 70 & 71 & 72 & 79 & 80 & 81
\end{bmatrix};
$$

We denote the set of indices of the cells (that is, the n values) that participate in constraint C_m by N_m, and the set of indices of the constraints (the m values) that associate with cell X_n by M_n.

For example:

$$N_{10} = \{1, 10, 19, 28, 37, 46, 55, 44, 73\}; \tag{2.2}$$

$$M_1 = \{1, 10, 19\}; \tag{2.3}$$

$$N_{10,19} = N_M \backslash n = \{1, 10, 28, 37, 46, 55, 44, 73\}; \tag{2.4}$$

These definitions have different means:

- We use the 2.2 to indicate the Variable nodes connected to the 10th Constraint node;
- We use the 2.3 to indicate the Constraint nodes connected to the 1st Variable node
- We use the 2.4 to indicate the Variable nodes connected to the 10th Constraint node, except for the 19th variable node.

2.1.1 Belief Propagation Formulation

In the *Belief Propagation* algorithm, the nodes in the *Tanner graph* send messages to each other, representing local information about the nodes. For the Sudoku puzzle, a constraint node sends a message about the probability that the constraint is satisfied, which it computes using information from the cell nodes about the probabilities of the cell contents.

A variable node, on the other hand, sends a message about the probabilities of the various cell contents, given information about the constraints associated with that cell. For a graph with cycles, nodes in the graph send information to each other until the messages converge, or until all constraints are satisfied, or until some maximum number of iterations is reached. We model the contents of the cells probabilistically. Let:

$$p_n = [P(S_n = 1)P(S_n = 2)...P(S_n = 9)] \tag{2.5}$$

be the *probability vector* associated with cell S_n. Cells which are specified initially, place all their probability mass on the specified value, while unspecified cells have probability uniformly distributed over possible outcome values. (The possible outcome values are obtained by eliminating values from consideration which would violate the three constraints associated with that cell. This is not strictly necessary; initial probabilities could be uniformly distributed over all nine possibilities. However, eliminating some contents based on constraints reduces the number of iterations of the algorithm.)
For example, a variable node relative to a cell initially filled with the number 3 send the following message

$$p_4 = e_3 = [0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0] \tag{2.6}$$

where e_3 is a vector of lenght nine with a single *1* at position $k = 3$ and zeros in other positions.

A variable node without a specified number, send for example the following message:

$$p_7 = \frac{1}{4}[0\ 0\ 1\ 0\ 1\ 0\ 1\ 1\ 0] \tag{2.7}$$

with a zero in position k if we can't have this number as possible value of the cell, *1* otherwise.

Belief propagation operates by sending probabilistic messages between adjacent nodes in the graph. The message that constraint node C_m sends to cell S_n is

$$r_{mn}(x) = P(C_m\ is\ satisfied \mid S_n = x)\ =\ P(C_m \mid S_n\ =\ x) \tag{2.8}$$

that is, the probability that constraint C_m is satisfied when the cell S_n contains x. The message from C_m to S_n is actually a probability vector:

SUDOKU	Alphabet Size q	Multiplications $(q-1)*q!$	Additions $(q!-1)$
MINI	4	72	23
CLASSIC	9	2903040	362879
HEX	16	$3.14*10^{14}$	$2.09*10^{13}$

Table 2.1: *Complexity at the Constraint Node.*

$$r_{mn} = [r_{mn}(1), r_{mn}(2), ..., r_{mn}(9)] \tag{2.9}$$

Each constraint node at each iteration requires the computation of this probability vector; With a *Brute Force* computation we have to evaluate: sum of $q!$ products of q factors. In the table 2.1 we can see the complexity of this computation.

The message that cell node S_n sends to constraint node C_m is

$$\begin{aligned} q_{mn}(x) &= P(S_n = x| \textit{ all the constraints except } C_m \textit{ involving } S_n \textit{ are satisfied }) \\ &= P(S_n = x|C_{m'},\ m' \in M_{nm}) \end{aligned} \tag{2.10}$$

that is, the probability that $S_n = x$ given that all of the constraints connected to S_n are satisfied, except the constraint to which the message is being sent. The *decision values* are based upon the message that cell node S_n obtains from all of the constraints,

$$\begin{aligned} q_n(x) &= P(S_n = x| \textit{ all the constraints involving } S_n \textit{ are satisfied }) \\ &= P(S_n = x|C_{m'},\ m' \in M_n) \end{aligned} \tag{2.11}$$

If there were no cycles in the graph, belief propagation theory asserts that, after a sufficiently large number of message passing steps, $q_n(x)$ would be the *Bayesian posterior probability*, incorporating information both from the prior probabilities and the evidence provided by the constraints. If there are cycles in the graph, then evidence recirculates around the graph, leading to potentially biased results. However, experience has shown that the results are usually still useful. The belief propagation rules are derived under certain assumptions of statistical independence. Strictly speaking, cycles in the graph lead to violation of these assumptions. However, the assumptions are approximately true, and lead to tractable, and useful, results.

2.2 Solving Sudoku using Moon Algorithm

In this section we analyze the work on Belief Propagation of Todd K. Moon and Jacob H. Gunther [11]. They found another way to express the classical formulation of Belief Propagation; with a particular approximation they create a different equation that can be used not only to solve the Sudoku puzzle but even in other fields. We will see that this formulation is really powerfull but to the other end is too much complex and slow.

2.2.1 Constraint to Variable Message

Moon start from the following equation:

$$\begin{aligned} r_{mn}(x) &= P(C_m|S_n = x) \\ &= \sum_{x_{n'}, n' \in N_{m,n}} P(C_m, \{S_{n'} = x_{n'}\}|S_n = x) \\ &= \sum_{x_{n'}, n' \in N_{m,n}} \Big[P(C_m|S_n = x, \{S_{n'}, n' \in Nm, n\}) * P(S_{n'}, n' \in Nm, n|S_n = x)\Big] \end{aligned} \tag{2.12}$$

Now, he invoke the assumption that the cells in the set $S_{n'}, n' \in N_{m,n}$ are independent. This is clearly not true, since cells associated with a constraint must have distinct contents; if $S_1 = 1$, it cannot be the case the $S_2 = 1$ also. However, following the spirit of the LDPC decoder we use that assumption. We thus have

$$r_{mn}(x) = \sum_{x_{n'}, n \in N_{m,n}} \Big(P(C_m|S_n = x, \{S_{n'}, n' \in N_{m,n}\}) * \prod_{l \in N_{m,n}} P(S_l = x_l|S_n = x)\Big) \tag{2.13}$$

We also note that $P(C_m|S_n = x, \{S_{n', n' \in N_{m,n}}\})$ is conditioned upon all of the cells connected to C_m. Constraint is C_m is then either satisfied or not, depending on the values of the arguments. Thus

$$P(C_m|S_n = x, \{S_{n'}, n' \in N_{m,n}\}) = \begin{cases} 1 \ \mathit{all}\ S_n\ \mathit{and}\ \{S_{n'}, n' \in N_{m,n}\}\ \mathit{are\ distinct} \\ 0 \ \ \mathit{otherwise} \end{cases} \tag{2.14}$$

We thus have

$$r_{mn}(x) = \sum_{\substack{\{x_{n'}, n' \in N_{m,n}\} \\ \{x, x_{n'}\}\ all\ unique}} \prod_{l \in N_{m,n}} P(S_l = x_l|S_n = x) \tag{2.15}$$

To formulate this as a belief propagation step, Moon invoke the approximation $P(S_l = x_l|S_n = x) = q_{ml}(x_l)$ the probability that cell S_l sends to constraint C_m. We thus obtain

$$r_{mn}(x) = \sum_{\substack{\{x_{n'}, n' \in N_{m,n}\} \\ \{x, x_{n'}\} \text{ all unique}}} \prod_{l \in N_{m,n}} q_{ml}(x_l) \tag{2.16}$$

Unfortunately, the sum is over a combinatorial set. However, when some of the cells in N_m are known, it reduces the size of the set. Furthermore, this is only a *local combinatorial complexity*, restricted to the cells involved in a constraint, and not over all the empty cells in the puzzle.

Let's see an example of this evauation: We considere a Sudoku $4*4$; If we have to evaluate the message $r_{m1} = [r_{m1}(1), r_{m1}(2), r_{m1}(3), r_{m1}(4)]$ for the 1th variable node, we need 4 steps in order to evaluate all the bits of the vector that we have to send. The formula for the 1st bit is:

$$\begin{aligned} r_{m1}(1) =& q_{m2}(2) * q_{m3}(3) * q_{m4}(4) + q_{m2}(2) * q_{m3}(4) * q_{m4}(3) + \\ & q_{m2}(3) * q_{m3}(2) * q_{m4}(4) + q_{m2}(3) * q_{m3}(4) * q_{m4}(2) + \\ & q_{m2}(4) * q_{m3}(3) * q_{m4}(2) + q_{m2}(4) * q_{m3}(2) * q_{m4}(3) \end{aligned} \tag{2.17}$$

where $r_{m1}(1)$ is given from the sum of all the possible permutation of the products of messages that a constraint receive from the other variable nodes.
In this formula we have to respect the condition

$$\substack{\{x_{n'}, n' \in N_{m,n}\} \\ \{x, x_{n'}\} \text{ all unique}} \tag{2.18}$$

that means that in each product we have to use all different probabilities among them and different from the probability that we evaluate in that moment. In a Sudoku $4*4$ we have *16* variables and each variable have *4* possible values; in each step we have to evaluate three sums from 1 to 4 in order to evaluate the value of a single probability. This means:

$$4^3 \textit{ sums } * \ 4 \textit{ values } * \ 16 \textit{ variables } = \ 4096 \textit{ total product to evaluate}$$

These products contains all the possible combinations, included the $X_{n'}$ equal among them; if we take only the valid products we have: $3! * 4 * 16 = 384$ *products*. In a Sudoku $4*4$ this number is very easy to evaluate; we have some problems in a Sudoku $9*9$ in which the total products are:

$$9^8\ sums\ *\ 9\ values\ *\ 81\ variables\ \approx\ 31\ billions\ total\ product\ to\ evaluate$$

And of these ones only $8!*9*81 \approx 29\ millions$ are valid.

In order to decrease this complexity, we generated some change to the program:

1. I evaluate only the messages $R_{mn}(x)$ for the unknown variables; the minimum number of known numbers for a Sudoku 9x9 is 17 then we have for the more complicated Sudoku:

   $$9^8 sums * 9 values * 64 variables \approx 24\ billions$$

2. For each cell, I evaluate only the probabilities relative to the numbers that are not present in the row, column or sub-grid.

E.G:

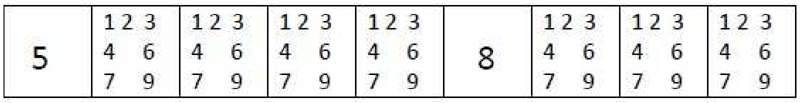
In the Figure 2.5 we can see that we know that the cells 1 and 6 have the known numbers 5 and 8, so I dont evaluate any value for these two cells; for the unknown cells I evaluate only the probabilities for the other numbers except for the 5 and the 8. For this simple example we have a reduction from:

$$9^8 sums * 9 values * 9 variables \approx 3\ billions$$

to:

$$9^8 sums * 7 values * 7 variables \approx 2\ billions$$

In a complete Sudoku the reduction is much more considerable.



5	1 2 3 4 6 7 9	1 2 3 4 6 7 9	1 2 3 4 6 7 9	1 2 3 4 6 7 9	8	1 2 3 4 6 7 9	1 2 3 4 6 7 9	1 2 3 4 6 7 9

Figure 2.5: *Moon Algorithm for a Sudoku puzzle 9*9*

2.2.2 Variable to Constraint Message

We derive $q_n(x)$; modifications to obtain $q_{mn}(x)$ are straightforward

$$\begin{aligned} q_n(x) &= P(S_n = x|\{C_m, m \in M_n\}) \\ &= \frac{P(S_n = x|\{C_m, m \in M_n\})}{P(\{C_m, m \in M_n\})} \\ &= \alpha P(\{C_m, m \in M_n\}|S_n = x) * P(S_n = x) \end{aligned} \tag{2.19}$$

where α is a *normalizing constant*. We assume independence again, then recognize $r_{mn}(x)$:

$$\begin{aligned} q_n(x) &= P(S_n = x) * \prod_{m \in M_n} P(C_m|S_n = x) \\ &= P(S_n = x) * \prod_{m \in M_n} r_{mn}(x) \end{aligned} \tag{2.20}$$

Similarly,

$$q_{mn}(x) = P(S_n = x) * \prod_{m' \in M_{n,m}} r_{m'n}(x) \tag{2.21}$$

The equation 2.21 means that each variable node send to one of its constraint nodes the product of the probabilities received from the other two constraint nodes.
At the same time, every variable node, evaluate a new *Probability Vector* in order to know if a new number has been found using all the messages received by the Constraint nodes; the formula is in the equation 2.22:

$$q_n(x) = P(S_n = x) * \prod_{m' \in M_n} r_{m'n}(x) \tag{2.22}$$

In operation, the Belief Propagation algorithm iterates between 2.16 and 2.21. However, for a cell whose contents are unambiguously known (such as the cells initially filled in) the *cell to constraint* message is simply the fixed probability vector.

2.3 Solving Sudoku using a different Algorithm

In this section we will explain the most important part of my work and how i reach my results and my conclusions.

2.3.1 Program Analysis

At the start of the program we have two matrix that represent the Sudoku and a matrix that represent the Tanner Graph on the strength of the Sudoku size N.

$$start\ grid = \begin{bmatrix} 3 & 1 & 4 & 2 \\ 4 & 2 & 1 & 3 \\ 2 & 4 & 3 & 1 \\ 1 & 3 & 2 & 4 \end{bmatrix};$$

In the start grid we have the complete Sudoku

$$default\ grid = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix};$$

In the default grid we have the starting disposition of the numbers; if we have 1 the cell is full at the start otherwise is empty.

$$Tanner\ Graph = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \\ 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \\ 1 & 2 & 5 & 6 \\ 3 & 4 & 7 & 8 \\ 9 & 10 & 13 & 14 \\ 11 & 12 & 15 & 16 \end{bmatrix};$$

The Tanner Graph matrix represent the link of the *Bipartite Tanner Graph* .

The second step is fill up the probabilities matrix; this is a matrix with size $\{N * N, N\}$: for the Sudoku $4 * 4$ I have a matrix $\{16, 4\}$. If a cell has a known number, the row relative to this one has only a one in the position of this number.

$$Probability\ Matrix = \begin{bmatrix} 0 & 0 & 1 & 0 \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ 0 & 0 & 0 & 1 \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ 0 & 1 & 0 & 0 \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ 1 & 0 & 0 & 0 \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \end{bmatrix};$$

E.G:

The cell S_1 at the start has a 3, so the relative row in the matrix is $P1 = \{0, 0, 1, 0\}$; The cell S_2 does not have e known number so his row at the start is $P2 = \{1, 1, 1, 1\}$;

When the constraint nodes evaluate the new probability, they have to send the value in the link relative at the right variable nodes. In this function we have $N * 3$ constraint nodes and each one send a message to his variables nodes, then we have $N * N * 3$ message to send through the links. In order to be sure that each variable node receive the right message in respect to the logical operation of the Tanner Graph, I have created a matrix of size $N * N * 3, N$ in which each variable node retrieve his messages. The constraint nodes in order to put the message in the correct link use this formula:

$$link = variable * 3 - \Big(3 - floor\Big(\Big(\frac{i-1}{N}\Big) + 1\Big)\Big);$$

where *link* is the entry in the matrix and i is the number of the constraint.

2.3.2 Constraint Function: Methodologies For Solutions

The following methodologies are applied in the constraints nodes; each of them after receiving the messages from the variable nodes, evaluate the new probability to send to the unknown cells using all these methods.

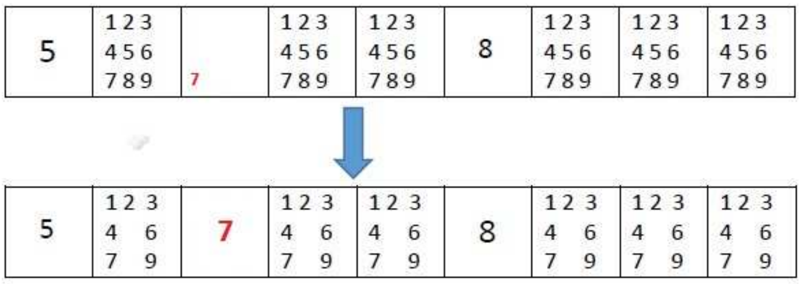


Figure 2.6: *Naked Single method for a Sudoku puzzle 9*9*

Naked Singles

This method requires that you can delete the content in cells. We start by writing in each free square all numbers allowed and not allowed, after eliminating the numbers already present in the row, column and sub-grid in the region to which the cell belongs. Then we examine the table in search of forced choices: in other words we will find a new number in a cell when this has only one possible candidate.

In order to implement this method the esed formula is:

$$r_{mn}(x) = \sim \prod_{\substack{n' \in N_m \\ \{if \ \sum_x q_{mn'}(x)=1\}}} q_{mn'}(x) \tag{2.23}$$

Let's see an example to better explain the algorithm:

The constraint node receive these messages,

- $S_1 = \{0, 0, 0, 0, 1, 0, 0, 0, 0\}$
- $S_2 = S_3 = S_4 = S_5 = S_6 = S_7 = S_9 = \{1, 1, 1, 1, 1, 1, 1, 1, 1\}$
- $S_8 = \{0, 0, 0, 0, 0, 0, 0, 1, 0\}$

The constraint node will send a message only to the unknown nodes; the new message is the product of the original message sent by the variable node and the negative value of the vector for the known cells: $S_2 * \sim S_1 * \sim S_8 = \{1, 1, 1, 1, 0, 1, 1, 0, 1\}$

In this way we remove from the cell the numbers that we already know. We can see an example of the algorithm in the Fig. 2.6.

Hidden Singles

This technique examine the arrangement of one of the numbers already appears twice in three regions in a row to check whether, in the third region where it is not present, in the line where it is not present, are prevented all other positions minus one, which then it must be the right one for that number.

In the Figure 2.7 there is an example for number 6: it is present already in two of the first three regions in column and then it must be present in the third region (the central one) in the remainder of the three columns (the first); here a box is already occupied (number 3) then controlling the orthogonal lines of the last two remaining boxes you find a line already occupied. The three 6 considered (in yellow), thus preventing the presence of other 6 in the empty boxes marked in purple. In the central region of the left remains one box "allowed" to 6 (highlighted in light green): and since there must be a 6 for each region, it is clear that the 6 of that region is right there.

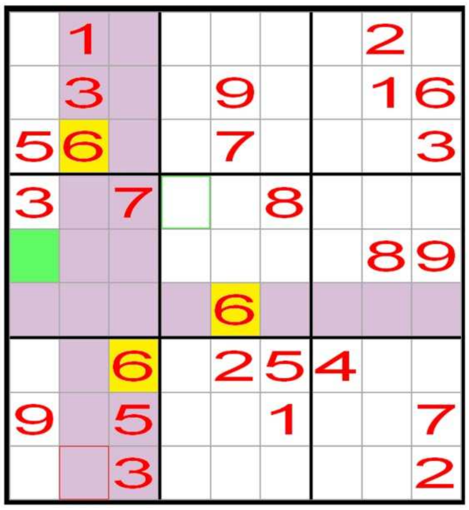


Figure 2.7: *Hidden Single method for a Sudoku puzzle 9*9*

In order to implement this method, the constraint nodes receive from the variable nodes their actual probabilities and evaluate a new value only by counting if a possible value appears once in the constraint. The formula implemented in this function is in the equation 2.24:

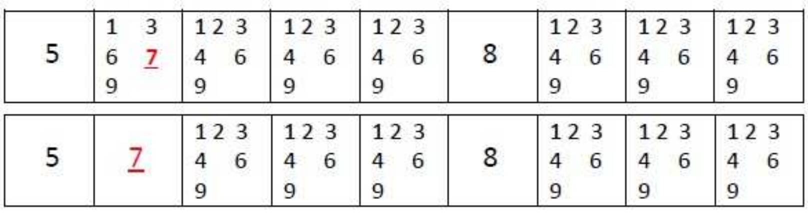
$$r_{mn}(x) = \begin{cases} 0 & \sum_{\substack{n' \in N_m \\ \{if \ \sum_x q_{mn'}(x) > 1\}}} q_{mn'}(x) > 0 \\ 1 & otherwise \end{cases} \tag{2.24}$$

Looking at the Fig. 2.8, the constraint node have to evaluate the new probability vector for S_2 and after received the probabilities from the other variable nodes that do not have a known number, they evaluate a new vector with all the possible entries:

$$Vector = \{0,0,0,0,1,0,1,1,0\} \tag{2.25}$$

When S_2 receive this new vector, it has only to multiply his actual probability to this one just received. The result can be a vector with all *zeros* and in this case the cell use the old probability vector or a vector with only a *one* corresponding to the right number.

$$\begin{aligned} New\ S_2 &= \{1,0,1,0,0,1,1,0,1\} * \{0,0,0,0,1,0,1,1,0\} = \\ &= \{0,0,0,0,0,0,1,0,0\} = 7 \end{aligned} \tag{2.26}$$



5	1 3 6 7 9	1 2 3 4 6 9	1 2 3 4 6 9	1 2 3 4 6 9	8	1 2 3 4 6 9	1 2 3 4 6 9	1 2 3 4 6 9

5	7	1 2 3 4 6 9	1 2 3 4 6 9	1 2 3 4 6 9	8	1 2 3 4 6 9	1 2 3 4 6 9	1 2 3 4 6 9

Figure 2.8: *Naked Single method for a Sudoku puzzle 9*9*

Pairs and Triplets: Tertium non datur

This technique is based on the assumption that within a group of n cells must exist exactly n numbers, hence from the corollary to the pragmatic choice is possible to reduce the number of candidates in the cells of the group.

1. If in a group of cells the same sequence of n candidates is present n times, then the candidates of this cells may be excluded from other cells.

   Let's see the following scheme of candidates for the cell:

   $$\{4,5\}\ \{4,7,9\}\ \{4,5\}\ \{7,9\}\ \{4,5,9,1\}$$

   in the example, only two boxes have the same sequence of two candidates $\{4,5\}$, we can then exclude those candidates from other fields, thus simplifying the possible solutions:

   $$\{4,5\}\ \{7,9\}\ \{4,5\}\ \{7,9\}\ \{9,\ 1\}$$

because 4 and 5 need to be in the two cells; if one of them was in a different box, would lead to a situation with an empty boxe. Now it's possible to repeat the algorithm with two other cells that have the sequence $\{7, 9\}$, then:

$$\{4,5\}\ \{7,9\}\ \{4,\ 5\}\ \{7,\ 9\}\ \{1\}$$

and we found a solution. The solution also applies to copresences triple, quadruple and so on:

$$\{4,5,7\}\ \{4,5,7\}\ \{4,5,7\}\ \{1,4,5,7,9\}\ \{1,4,5,7,9\}$$

the first three boxes all have the same candidates $\{4, 5, 7\}$ and these numbers can be only in these three boxes. Consequently, simplifying, we have:

$$\{4,5,7\}\ \{4,5,7\}\ \{4,5,7\}\ \{1,9\}\ \{1,9\}$$

2. If in a group the same n candidates are in exactly the same n sequences, then you can exclude other candidates from these cells.

   In the following example *5* and *9* appear only in the first and fourth cell, then

$$\{4,5,8,9\}\ \{2,3,4,6,8\}\ \{2,3,4,6,8\}\ \{2,3,4,5,9\}$$

   becomes

$$\{5,9\}\ \{2,3,4,6,8\}\ \{2,3,4,6,8\}\ \{5,9\}$$

In my program the used formula is:

- For Naked Pairs:

$$r_{mn}(x) = \begin{cases} \sim q_{mn}(x) \quad if \ \sum_x q_{mn'}(x) = 2 \ \& \\ \qquad\qquad \sum_x q_{mn'}(x) * q_{mn''}(x) = \sum_x q_{mn''}(x) > 0 \\ \qquad\qquad With \ \ n', n'' \in N_{mn} \\ 0 \ \ otherwise \end{cases} \tag{2.27}$$

- For Naked Triplets:

$$r_{mn}(x) = \begin{cases} \sim q_{mn}(x) \quad if \ \sum_x q_{mn'}(x) = 3 \\ \qquad \& \sum_x q_{mn'}(x) * q_{mn''}(x) = \sum_x q_{mn''}(x) > 0 \\ \qquad \& \sum_x q_{mn'}(x) * q_{mn'''}(x) = \sum_x q_{mn'''}(x) > 0 \\ \qquad With \ \ n', n'', n''' \in N_{mn} \\ 0 \quad otherwise \end{cases} \tag{2.28}$$

All Cardinality

All the functions previously analyzed can be group in a unique function. In this way the constraint node does not have to make different evaluations for each situation, but he can make the same evaluation in each cycle. Clearly the resulting function is more complicated and it need more time in order to evaluate the result.

This new formula allows the Constraint node to work in a more similar manner to the classic Belief Propagation Algorithm. This is because the Constraint node performs the same function at each iteration, instead to do different functions every cycle.

The rudiment that stay at the base of this function is similar to the function *Pairs and Triplets: Tertium non datur*; the only difference is that now we apply this thinking to all the possible cardinality. If we have a Sudoku with $N = 9$, when we are evaluating the message to send to a cell, we can evaluate all the cardinality starting from $N - 1 = 8$ until 1.

E.G:

If in X cells we have X values we can surely say that the last remaining value is in the cell which we are sending the message. If $X = N - 1$, we have the particular case of *Hidden Single* Values algorithm.

The formula used to implement this algorithm is:

$$r_{mn}(x) = \prod_{N-1}^{1} c^i(x) \tag{2.29}$$

where $c^i(x)$ are all the possible permutation of cardinality i of the message received by the other cells.

$$c^i(x) = \begin{cases} 0 & if \sum_{j=1}^{N-1} q_{mn'}(x) > 0 \ \& \ |\sum_{j=1}^{N-1} q_{mn}| = N - i, \ \ with \ n' \in N_{mn} \\ 1 & otherwise \end{cases} \tag{2.30}$$

Then, if we are using a Sudoku $N * N$ with $N = 9$, for the cardinality $N - 1 = 8$ we will have only one permutation of the remaining cells; for the cardinality $N - 2 = 7$ we will have 8 possible permutations (in each one we exclude one of the 8 cells) and so on until we arrive to the cardinality 1 in which we consider all the cells for themselves.

Let's see some example of this new check; we have different cases that can be treated in the same way. For example the following situation are equal:

$$\{1,2,3\} \ \{1,2,3\} \ \{1,2,3\} \ = \ \{1,2\} \ \{1,2,3\} \ \{1,2,3\}$$

$$\{1,2,3\} \ \{1,2,3\} \ \{1,2,3\} \ = \ \{1\} \ \{1,2\} \ \{1,2,3\}$$

If we continue with the algorithm, the last combination can be written as follow:

$$\{1\} \ \{1,2\} \ \{1,2,3\} \ = \ \{1\} \ \{2\} \ \{1,2,3\} \ = \ \{1\} \ \{2\} \ \{3\}$$

With the simple elimination of the known numbers. This system can be applied to many combinations of cardinality. We just have to check, as already told, if the same X numbers appear in X cells, without any restrictions.

Evaluation of complexity: In each cycle the constraints nodes have to evaluate all the messages for all the variable nodes belonging to them; so we have

$$N * 3 \ constraint \ node \ evaluation \ * \ N \ variable \ node \ evaluation \approx N^2 \ constraint \ evaluation.$$

Within the function we have to check all the cardinality; the simplier solution is to check each cardinality per time $\{N-1, N-2, ...1\}$; instead of do all this passages we can evaluate two cardinalities at the same time. In this way, when we are evaluating all the permutation of cardinality $N - X$, we can evaluate also the cardinality X and we are sure that we are checking for all the possible permutation of this last cardinality.

With this expedient is possible to decrease considerably the complexity of the function; especially for the heavier cardinality like 2 or 3 that have many permutation in the Sudoku $9 * 9$.

The resultant complexity for this check is N^5. So at the end of all the constraint node evaluation we have a complexity of N^7 approximatively. Really the complexity is lower[1] because the constraint nodes have to evaluate a new message only for the cells that dont have a known number. Because in a solvable Sudoku we need at least 17 known cells and each cell appear three times in the constraint nodes we have a considerable complexity reduction and the complexity continue to decrease gradually when we find new numbers.

2.3.3 Variable Function

This function describe the behavior of the variable nodes. They receive three different message from each constraint that they have to satisfy.

In a Sudoku $4 * 4$ then we have $16\ variables * 3\ messages = 48\ total\ messages$. For this reason the function has as argument a matrix of dimension "$N * N * 3$" that represent the link between variable and constraint nodes.

When the variable nodes receive these values they evaluate the new probabilities with the formula:

$$q_n(x) = \prod_{m \in M_n} r_{mn}(x)$$

It means that the new value is simply valuate as a product of the three messages received. If the result of this evaluation is a new number for the sudoku, the variable node is marked as *known* and the constraint nodes stop to evaluate its relative messages. Otherwise the variable node continue to send message to the constraint nodes and vice versa.

2.3.4 Message-Passing Schedules

A *message-passing schedule* in a factor graph is a specification of messages to be passed during a precise period. Obviously a wide variety of message-passing schedules are possible.

For example, the so-called *schedule flooding* calls for a message to pass in each direction over each edge at each period.

A schedule in which at most one message is passed anywhere in the graph at each period is called a *serial schedule*.

We will say that a vertex v has a message pending at an edge e if it has received any messages on edges other than e after the transmission of the most previous message on e.

[1]In a Sudoku $9 * 9$ we have a final complexity approximately of $N^7 = 4million$ iterations; instead of this results in all the done test we obtain an average iterations value of approximately 1 million with a maximum value of $1,8$ *millions*. The complexity is less than half of the expected value.

Such a message is pending since the messages more recently received can affect the message to be sent on e. The receipt of a message at v from an edge e will create pending messages at all other edges incident on v. Only pending messages need to be transmitted, since only pending messages can be different from the previous message sent on a given edge.

In a *cycle-free* factor graph, assuming a schedule in which only pending messages are transmitted, the sum-product algorithm will eventually halt in a state with no messages pending.

In a factor graph with *cycles*, however, it is impossible to reach a state with no messages pending, since the transmission of a message on any edge of a cycle from a node v will trigger a chain of pending messages that must return to v, triggering v to send another message on the same edge, and so on indefinitely.

In practice, all schedules are finite. For a finite schedule, the sum-product algorithm terminates by computing, for each x_i, the product of the most recent messages received at variable node x_i.

If x_i has no messages pending, then this computation is equivalent to the product of the messages sent and received on any single edge incident on x_i.

In order to improve the algorithm, instead to use the classic schedule (*flooding*) which send all the message from variable nodes to check nodes at the same period, we use a different schedule.

The schedules that we will see can be divided in two big groups: *No Adaptive* that follow an order without changing during the development of the algorithm and *Adaptive* change the order during the development in order to better handle the updates of the algorithm.

The first schedule that I tried is the classic one (flooding); all the messages in both directions from variable nodes to constraint nodes and from constraint nodes to variable nodes are sent together. The result of the simulations with the *flooding schedule* shows an average time necessary to complete the algorithm around at **2':05"**.

In the next Sections we will see the different kind of shchedules that we used in the simulations.

Non Adaptive Schedules

With the next schedules, during the algorithm, the order does not change.

1. *Linear Schedule (top-down)*: We send the messages starting from the first check node

until the last one. In this way we evaluate first all the check node relative to the Sudoku rows, then to the columns e finally to the sub-grids.

With this schedule we have an average time of: **1':02"**.

2. *Linear Schedule (bottom-up)*: We send the messages starting from the last check node until the first one. In this way we evaluate first all the check node relative to the Sudoku sub-grids, then to the columns e finally to the rows.

   With this schedule we have an average time of: **1':06"**.

3. *Alternate Schedule*: We send the messages starting from the first check node relative to the *row constraint* , but instead to send the next messages to the second check node, we send the messages to the first check node relative to the *column constraints* ; after that we send the new messages to the first nodes relative to the first check node relative to the *sub-grid constraints* . The algorithm start again from the second row constraint node and son on.

   With this schedule we have an average time of: **1':17"**.

Adaptive Schedules

1. *Min to Max (number possibilities)*:

   In this scheduling, before to start the algorithm, we have to order the check nodes, following a specific rule.

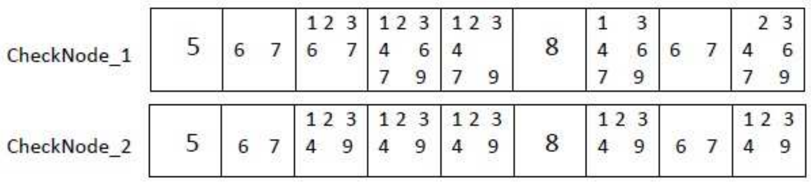
   As we can see in Fig. 2.9, the $Check\ Node$ 1 and $Check\ Node$ 2 receive nine messages with different possibilities. Every check node make the sum of the possible numbers and obtain the following results: $Check\ Node\ 1 = 36$ and $Check\ Node\ 2 = 31$.

   After that all the $N * 3$ check nodes evaluate the sum of the possible values of their cells, we can order them starting, in this schedule, from the lower to the higher. In this example, with only two check nodes, we start from the *check node* 2 and we continue with the *check node* 1.

   When we send the messages to the last check node, we provide to create another order looking at the new messages after this first cycle of the schedule. So after $N*3$ dispatches we need an other order for the check nodes.

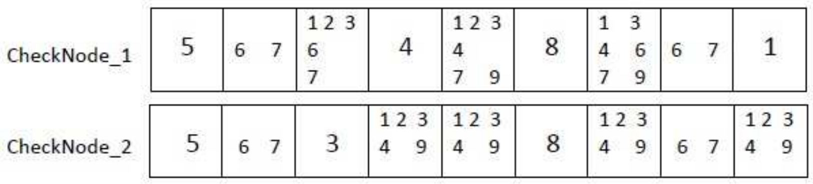
   With this schedule we have an average time of: **52"**.

2. *Max to Min (number possibilities)*: This scheduling is the opposite of the previous one. In this schedule, the check nodes are ordered starting from the higher to the lower.



CheckNode_1	5	6 7	1 2 3 6 7	1 2 3 4 6 7 9	1 2 3 4 7 9	8	1 3 4 6 7 9	6 7	2 3 4 6 7 9
CheckNode_2	5	6 7	1 2 3 4 9	1 2 3 4 9	1 2 3 4 9	8	1 2 3 4 9	6 7	1 2 3 4 9

Figure 2.9: *Different Check Node 1 and 2*



CheckNode_1	5	6 7	1 2 3 6 7	4	1 2 3 4 7 9	8	1 3 4 6 7 9	6 7	1
CheckNode_2	5	6 7	3	1 2 3 4 9	1 2 3 4 9	8	1 2 3 4 9	6 7	1 2 3 4 9

Figure 2.10: *Different Check Node 1 and 2*

Looking again the Fig. 2.9, this time we start from the *check node* 1 and we continue with the *check node* 2.

When we send the messages to the last check node, we provide for create another order looking at the new messages. So after $N * 3$ sent messages from Constraint to variable nodes we have to create an other order.

With this schedule we have an average time of: **1':09"**.

3. *Min to Max (known cells)*: In this scheduling, as in the two previous schedules, before to start the algorithm we have to order the check nodes, following a specific rule.

   E.G:

   After the evaluation of the known cells for all the variable nodes, we can order them starting, in this schedule, from the lower to the higher. In the example of Fig. 2.10, with only two check nodes, we start from the $Check\ Node$ 2 and we continue with the $Check\ Node$ 1 because they have rispectively 4 and 3 known cells.

   When we send the messages to the last check node, we provide to update the order looking at the new messages after this first cycle.

   With this schedule we have an average time of: **1':13"**.

4. *Max to Min (known cells)*: This scheduling is the opposite of the previous one. In this schedule, the check nodes are ordered starting from the higher to the lower.

Like in the others adaptive schedules, we provide to create another order looking at the new messages after every cycle.

With this schedule we have an average time of: **1':28"**.

5. *Maximal Residual (variable nodes metric)*: In this scheduling we consider the differences between the messages in the cycle i and the messages in the previous cycle $i-1$. In this version of the *maximal residual* we evaluate the metric in the variable nodes: we take the cell with the maximum residual and we send the messages to the three relative check nodes.

   With this schedule we have an average time of: **1':06"**.

6. *Maximal residual (check nodes metric)*: This scheduling is similar to the previous one: we consider the differences between the messages in the cycle i and the messages in the previous cycle $i-1$, but the only difference is that we evaluate the metric on the check node instead on the variable node. In every cycle we evaluate the differences between all the cells belonging to a check node; we take the check node with the maximum residual and we send first a message to that one.

   With this schedule we have an average time of: **1':04"**.

Final Comparison

After various simulations, we can say that all the different schedules are better than the algorithm without a schedule, because we have always a reduction of the average time which often corresponds to half of the average time without scheduling.

The best Schedule is the: *MinToMax (numbers possibilities)*. Whit this schedule we reduce the average time from **2':05"** to only **52"**, less than half time.

In Tab. 2.2 we can see the summary of all the average times. There are other possible adaptive schedules, for example using a semi-random order of the check nodes, but they dont improve considerably the performances. Sometimes, instead to improve, they make worse them.

The results of these simulations are obtained running simulations with different Sudoku. Every scheduler behave differently based on the Sudoku that we are trying to solve. This behavior can bring us to suppose that, with a high number of different Sudoku inputs, the average time of all the different scheduler become the same when the number of inputs increase.

TYPE	SCHEDULING	AVERAGE TIME
without	flooding	$2' : 05''$
No-adaptive	linear (top-down)	$1' : 02''$
	linear (bottom-up)	$1' : 06''$
	Alternate	$1' : 16''$
Adaptive	linear Min To Max (numbers)	$52''$
	linear Max To Min (numbers)	$1' : 09''$
	linear Min To Max (cells)	$1' : 13''$
	linear Max To Min (cells)	$1' : 28''$
	max residual variable	$1' : 06''$
	max residual check	$1' : 04''$

Table 2.2: *Scheduling Average times.*

Looking this table, we see that the best *Non Adaptive Scheduler* is the *Linear (top-down)*; the best *Adaptive Scheduler* is the Linear *Min to Max (number possibilities)*. The evaluation time is respectively **1':02"** for the first scheduler and **52"** for the second.

The improvement of the Adaptive Scheduler is not high enough to justify the use of this algorithm, so its better to use the Non Adaptive Scheduler, because it has a smaller complexity.

2.4 Conclusion and Results for Sudoku Sokving

2.4.1 Stopping set in Belief propagation

A *stopping-set* in a Sudoku problem is a set of random variables (cell) S such that, even if all the other cells are given (or correctly found), for each $x \in S$ there are at least two digits that satisfy all the constraints on x. In Fig. 2.11 we can see an example of this situation. We have a Sudoku problem with three solutions. The initially given numbers are marked by blue background, white cells form the solution backbone and solutions differ by the values in red cells.

We can have another possible stopping-set in the Sudoku when, with a simple cross-check of different constraint (row, column, sub-grid), it's possible to find new numbers and go on with the algorithm, but the Belief Propagation can not do that.

For example looking the figure 2.12 we can see taht we can have a 4 only in the cells $\{8, 4\}$

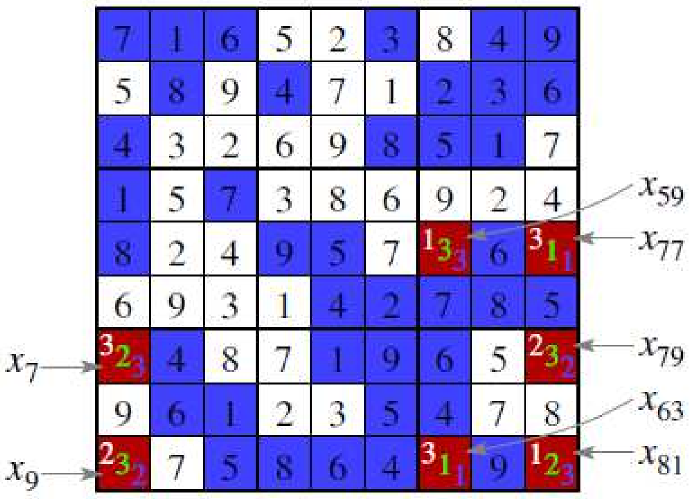


Figure 2.11: *Stopping Set in Sudoku*

and $\{9, 4\}$ and for sure we have another 4 in $\{5, 6\}$. Then, we are sure that in the cell $\{3, 5\}$ there is a 4. The problem is that for a player this is simple because he can cross-check the knowledge of the row $4, 5$ and 6. In the Belief Propagation this is not possible because every Constraint node work by itself and it does not know the data of the other Constraint nodes.

2.4.2 Equivalence of the Algorithms

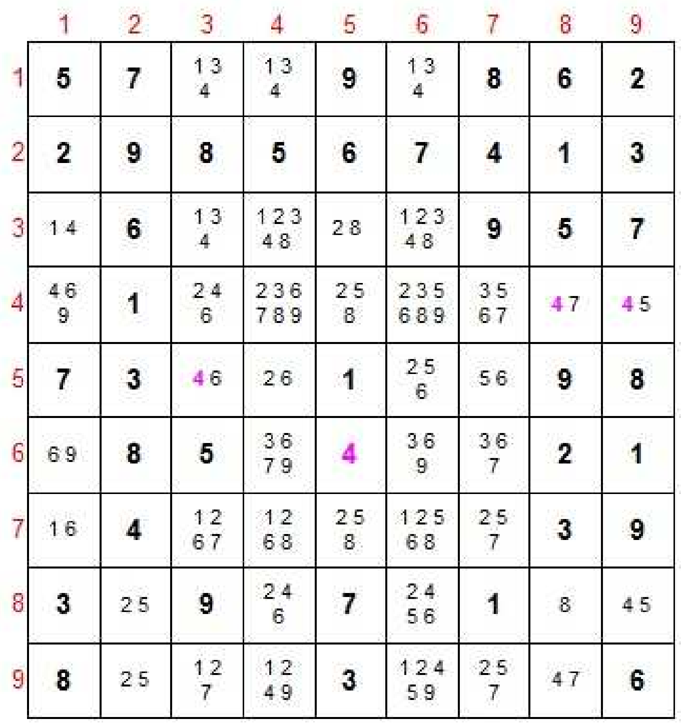
During all the simulations we tested the *Moon Algorithm* and my version of the algorithm for the Sudoku Solving; the first important thing was to prove that both the algorithm are identical. To do that I tried all the possible inputs that one Constraint Node can have during the algorithm and I compared the results of the two algorithms.

To better see how I did, let's see the example with the Sudoku $4 * 4$.

E.G.

Every Constraint node receive messages from 4 Variable nodes and after the evaluation they send the response. When they receive the messages, they take 3 messages from 3 variable nodes and they send the new evaluated probability using these messages to the $4th$ Variable node.

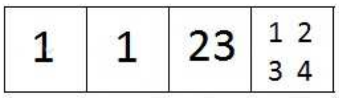
Every cell has 4 values so the Constraint node receive $4 * 3 = 12$ different values; then we have to evaluate all the valid permutation of this 12 values. The possible permutation are in total 2^{12} but not all are valid.



	1	2	3	4	5	6	7	8	9
1	**5**	**7**	1 3 4	1 3 4	**9**	1 3 4	**8**	**6**	**2**
2	**2**	**9**	**8**	**5**	**6**	**7**	**4**	**1**	**3**
3	1 4	**6**	1 3 4	1 2 3 4 8	2 8	1 2 3 4 8	**9**	**5**	**7**
4	4 6 9	**1**	2 4 6	2 3 6 7 8 9	2 5 8	2 3 5 6 8 9	3 5 6 7	4 7	4 5
5	**7**	**3**	4 6	2 6	**1**	2 5 6	5 6	**9**	**8**
6	6 9	**8**	**5**	3 6 7 9	**4**	3 6 9	3 6 7	**2**	**1**
7	1 6	**4**	1 2 6 7	1 2 6 8	2 5 8	1 2 5 6 8	2 5 7	**3**	**9**
8	**3**	2 5	**9**	2 4 6	**7**	2 4 5 6	**1**	8	4 5
9	**8**	2 5	1 2 7	1 2 4 9	**3**	1 2 4 5 9	2 5 7	4 7	**6**

Figure 2.12: *Stopping Set in Sudoku*

For example the following configuration is not possible



1	1	23	1 2 3 4

Figure 2.13: *Invalid Permutation for A Sudoku 4*4.*

because it's impossible to have in the same row two equal numbers (in this case the number "1").

So, instead to test $2^{12} = 4096$ possible permutation, this number is smaller: the real number is 2794. After the test we saw that the results of the two algorithms are the same not only for the 2794 valid inputs, but we arrive to 3125 equal results over 4096 inputs.

The rules of the Sudoku $4 * 4$ can be extended also for the Sudoku $9 * 9$ so we can claim that the *Moon Algorithm* and *my* version of the algorithm are identical.

To be more sure about this result we tried the test also with the sudoku $9 * 9$. In this case the Constraint node receive messages from 8 Variable nodes and send the result to the $9th$

node.

We have $9 * 8 = 72$ different inputs and 2^{72} total permutation. Obviously this number is impossible to evaluate and a lot of these are not valid. So we tested randomly different intervals of all the combinations and after the test all the results were the same.

2.4.3 Complexity of the Algorithms

After seen that the two algorithms are equal, we will see the complexity of both. I will show the complexity in terms of *number of iterations* and *Time Duration*.

Number of Iterations

- Moon Algorithm:

```
for (N*3)*N
.       for N
.       .       while (N-1)!
.       .       .       permutation();
.       .       .       for N
.       .       .               ....
.       .       .       end;
.       .       end;
.       end;
end;
```

  Where the function ***permutation()*** has a complexity equal to **O(n)**. The total complexity of this Constraint Function is $N^4 * (N-1)!$. In a Sudoku $4 * 4$ it means only 1536 in every Constraint node and this is acceptable; we have a problem in the sudoku $9 * 9$ where the complexity in every Function node is: $9^4 * 8! = 2380855680 \approx 2^{31}$ and we understand that this evaluation for 27 nodes in every cycle is really hard and it need too much time.

  After many simulations I reached an average number of iterations per esecution, approximately equal to 800 *millions*.

- My algorithm:

```
        for (N*3)*N
        .       for (N-1)*N + N
```

```
.       .       ...
.       end;
.       for N
.       .       for N
.       .               ...
.       .       end;
.       .       for (N-2)*N
.       .               ...
.       .       end;
.       .       for (N-2)
.       .       .       for N
.       .       .               ...
.       .       .       end;
.       .               for (N-3)*N
.       .       .               ...
.       .       .       end;
.       .       .       for (N-3)
.       .       .       .       for N
.       .       .       .               ...
.       .       .       .       end;
.       .       .       .       for (N-4)*N
.       .       .       .               ...
.       .       .       .       end;
.       .       .       .       for (N-4)
.       .       .       .       .       for N
.       .       .       .       .               ...
.       .       .       .       .       end;
.       .       .       .       end;
.       .       .       end;
.       .       end;
.       end;
end;
```

we can summarize the previous code in the following reduced.

```
for (N*3)*N
.       for N
.       .       for N
.       .       .       for N
.       .       .       .       for N^{2}
```

Algorithm	Scheduling	Time Duration						
My	No	$32''$	$64''$	$25''$	$30''$	$46''$	$44''$	$76''$
	Yes	$19''$	$34''$	$16''$	$17''$	$25''$	$23''$	$41''$
Moon	No	$68' : 12''$	$86' : 11''$	$71' : 36''$	$72' : 42''$	$63' : 37''$	$63' : 26''$	$98' : 51''$
	Yes							

Table 2.3: *Confront of the Time Duration.*

```
.         .         .         .         .                   ...
.         .         .         .         end;
.         .         .         end;
.         .         end;
.         end;
end;
```

The total complexity is $O(n^7)$. In a Sudoku $4 * 4$ we have 16384 iteration per node every cycle.

This result is worse than the Moon Algorithm, but if we see the Sudoku $9 * 9$ we have only $4782969 \approx 2^{22}$ iterations against the 2^{31} for the Moon Algorithm.

After many simulations I reached an average number of iterations per esecution, approximately equal to 1.2 *millions*.

Duration

Obviously the number of iterations affect also the duration of all the program. In the table 2.3 we can see with different Sudoku the times needed by the two algorithms with and without scheduling.

The first observation is that with or without scheduling the Moon algorithm is not comparable with my version of the algorithm in both cases.

Another observation is that in the table does not appear the time with scheduling for the Moon Algorithm. This is because, the time is really high; indeed with scheduling all the algorithms have more iterations than without; for the Moon Algorithm this is very negative because every cycle has a lot of iterations; so instead to improve, the performances decrease.

Final Results

This algorithm allows to solve sudokus of moderate difficulty in less than 30 rounds. In the Fig. 2.14 we can see the percentage of Sudoku solved with the two algorithms.

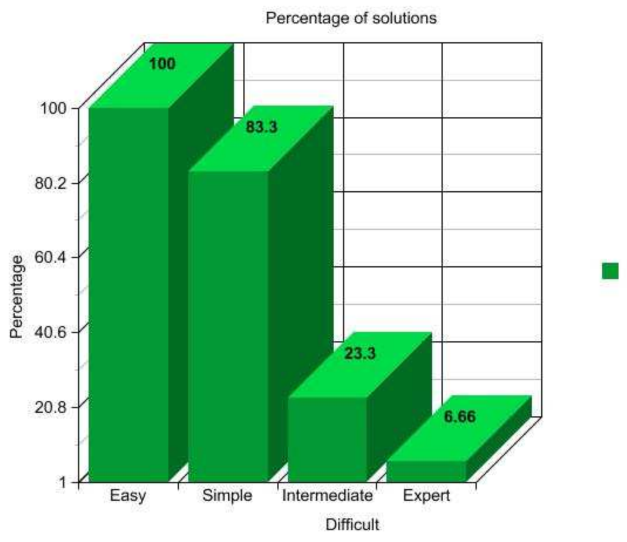


Figure 2.14: *Success solving rate of the algorithms depending on the level of sudokus. Evaluated using 30 sudokus of each level.*

difficulty	Min Numbers	Max Numbers
expert	17	20
intermediate	21	24
simple	25	28
easy	29	32

Table 2.4: *Numbers for Sudoku Difficult.*

These results are obtained using four different kind of sudokus: difficult, intermediat, simple, easy; Every Sudoku difficult has a different range of starting numbers; this range is explained in Table 2.4. If there more *givens*, message passing is equivalent to *logical deduction* and the Sudoku Solving lose his sense.

2.4.4 Conclusions

The *Belief Propagation* method is exact on graphs with no cycles. However, the graph associated with Sudoku has many short cycles in it. In fact, every cell is in four cycles of girth four. There are two of the following form: "*cell* $\rightarrow$ *row constraint* $\rightarrow$ *cell on row* $\rightarrow$ *box constraint* $\rightarrow$ *cell*" one for each of the two other cells on the row in a box, and "*cell* $\rightarrow$ *column constraint* $\rightarrow$ *cell on column* $\rightarrow$ *box constraint* $\rightarrow$ *cell*" one for each of the two other cells on the column in a box. These many short cycles will definitely bias the results of the message passing algorithm. What results is that not every puzzle is solvable by this Message Passing technique.

When the Sudoku complexity is higher we have a few *givens* and it follow that in the graph there are many cycles that prevent us to reach the solution. When the *givens* number increase many Variable nodes are known, so they do not partecipate to the algorithm and it follow that we have many cropped cycles: this is why is simplier to reach a solution.

In solving the puzzle, several iterations of elimination were computed: the possible contents of each cell were eliminated based on the constraints the cell associated with. This reduced the number of Belief Propagation iterations, and acts according to how a human would begin solving the puzzle. Following this simple elimination, the Belief Propagation proceeds. As computation proceeds, as a probability vector emerges that places all of its mass on a single cell, a *hard decision* is declared, establishing the contents of a cell. (Filled cells are important because they reduce the computationally complexity of the sum in 2.16 and 2.29.)

We can finish saying that Belief Propagation is really powerfull because of its semplicity and can be used in the *multiple constraint satisfaction problems* in order to completely find a solution or just to find a part of it but greatly decrease the complexity of the problem.

Chapter 3

Solving Low-Density Parity-Check Codes with Belief Propagation Algorithms on Binary Erasure Channel

3.1 Introduction to LDPC codes

Low-density parity-check (LDPC) codes is an *error correcting code* used in the *noisy communication channels* to reduce the probability of loss of information. With LDPC, this probability can be reduced to as small as desired, thus the data transmission rate can be as close to *Shannon Limit*.

LDPC was developed by Robert Gallager in his doctoral dissertation at MIT in 1960 [8].

Due to the limitation in computational effort in implementing the coder and decoder for such codes, LDPC was ignored for almost 30 years. During that long period, the only notable work done on the subject was due to R. Michael Tanner [15] where he generalized LDPC codes and introduced a graphical representation of the codes later called *Tanner Graph*.

Since 1993, with the invention of *turbo codes*, researchers switched their focus to finding low complexity code which can approach *Shannon channel capacity* . LDPC was reinvented with the work of Mackay [10] and Luby [2].

The powerful capabilities of LDPC codes have led to their recent inclusion in several standards, such as IEEE 802.16, IEEE 802.20, IEEE 802.3 and DBV-RS2.

On the negative side, LDPC codes have a significantly higher encode complexity, being generically quadratic in the code dimension, although this can be reduced somewhat. Also, decoding may require many more iterations than *turbo decoding*, which has implications for latency.

In the next section, I discuss how to decode an LDPC using belief propagation algorithm with the *hard-decision decoder* over a *Binary Erasure Channel* (BEC).

3.2 Low Density Parity-check Code (LDPC)

LDPC Codes are named that way because they are defined by a *sparse parity check matrix*. A parity check matrix is a matrix that define the allowed *codewords* of a system using the fact that the product of all allowed *codewords* and only them by this matrix gives a zero vector.

The matrix is said to be sparse because it contains a lot of zeros, and that is why these codes are called *Low density.*

Any linear code has a *bipartite graph* and a *parity-check matrix representation.* But not all linear code has a *sparse* representation.

A $N * M$ matrix is sparse if the number of ones in any row (the row weight w_r, and the number of ones in any column (the column weight w_c), is much less than the dimension ($w_r << M, w_c << N$). The sparse property of LDPC gives rise to its algorithmic advantages. An LDPC code is said to be regular if w_c is constant for every column, w_r is constant for every row and $w_r = w_c * \frac{N}{M}$. An LDPC which is not regular is called irregular .

The sum-product is used for the *iterative decoding algorithm*; there are two derivations of this algorithm: *hard-decision* and *soft-decision* schemes. In the *soft-decision scheme* as in the *Sudoku Solving Algorithm* every node send a vector message with the probability to have all the different values. In the *hard-decidion scheme* every node send only one value (usually a bit 0 or 1).

Of course there are two kinds of nodes: the *variable nodes*, which contain the value of the *codeword*, and the *constraint nodes*, which represent the system. All the variable nodes are initialized with the received value. Then they send their value to the constraint nodes that are connected to it. The constraint nodes compute the parity using all the inputs except one, and send to the remaining variable node the value it should have so that the parity is respected.

The constraint node do this for all its neighbors. Then, the variable nodes update their value according to the messages of the constraint nodes, and it starts over. As the graph may not be cycle-free, the algorithm cannot stop after one message exchanged on each edge in each direction.

The algorithm stops when the messages has reached a fixed point, when they are the same from an iteration to another. The problem is that when there are cycles, the messages may oscillate and converge to a wrong value. The result is the values of the variables nodes at the end of the iterative process.

Since the parity check matrices are generally not in systemic form, in the next explaination we will have the symbol A to represent parity check matrices, reserving the symbol H for parity check matrices in systematic form.

A *message vector* m is a $K * 1$ vector; a *codeword* is a $Nx1$ vector. The *generator matrix* G is $N * K$ and the parity check matrix A is $(N - K) * N$, such that $HG = 0$.

We denote the rows of a parity check matrix as:

$$\mathbf{A} = \begin{bmatrix} A_1^T \\ A_2^T \\ . \\ . \\ A_M^T \end{bmatrix};$$

The equation $a_i^T c = 0$ is said to be a *linear parity-check constraint* on the codeword c. We use the notation $z_m = a_m^T c$ and call z_m a *parity check* or, more simply, a *check*.

Using Gaussian elimination with column pivoting as necessary (with binary arithmetic) determine an $M * M$ matrix A_p^{-1} so that

$$H = A_p^{-1} * A = [I \ A_2]. \tag{3.1}$$

(If such a matrix A , does not exist, then A is rank deficient, $r = rank(A) < M$. In this case, form H by truncating the linearly dependent rows from $A_p^{-1} * A$. The corresponding code has $R = \dfrac{K}{N} > \dfrac{N-M}{N}$, so it is a higher rate code than the dimensions of A would suggest). Having found H, form

$$\mathbf{G} = \begin{bmatrix} A_2 \\ I \end{bmatrix};$$

Then $HG = 0$, so $A_p HG = AG = 0$, so G is a *generator matrix* for A. While A may be sparse, neither the *systematic generator* G nor H is necessarily sparse.

Finally, if the message is

$$\mathbf{m} = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \end{bmatrix};$$

then the codeword will be

$$\mathbf{c} = \mathbf{Gm} = \begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \\ 1 \\ 2 \\ 0 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

If no bit is flipped during transmission, in other words, $y = c$. Then the *syndrome vector* is

$$\mathbf{z} = \mathbf{Hy} = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

If, for example, the 6^{th} bit is flipped, then

$$\mathbf{z} = \mathbf{Hy} = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}.$$

Reading z from the bottom up (higher position first), we see the flipped bit is indeed 6 (110).

3.2.1 Transmission Through a Gaussian Channel

The decoder for codewords is transmitted through an *additive white Gaussian noise (AWGN) channel* .

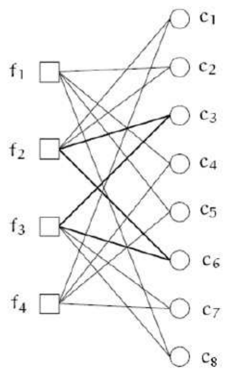


Figure 3.1: *Belief propagation example code.*

When a codeword is transmitted through an AWGN channel, the binary vector c is first mapped into a transmitted signal vector t. A *binary phase-shift keyed* (BPSK) signal constellation is employed, so that the signal $a = \sqrt{E_c}$ represents the bit 1 and the signal $-a$ represents the bit 0. The *energy per message* bit E_b is related to the energy per transmitted coded bit E_c, by $E_c = RE_b$, where $R = k/n$ is the *rate* of the code.

The transmitted signal vector t has elements $t_n = (2c_n - 1)a$. This signal vector passes through a channel and sometimes some bits are erasered during the transmission.

3.2.2 Hard-decision Decoder

In Figure 3.1 we can see an example of Factor Graph for LDPC Code, its corresponding paritycheck matrix is:

$$\mathbf{H} = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \end{bmatrix};$$

An error free codeword of H is

$$c = [1\ 0\ 0\ 1\ 0\ 1\ 0\ 1]^T.$$

Suppose we receive

$$y = [1\ 1\ 0\ 1\ 0\ 1\ 0\ 1]^T$$

So c_2 was flipped. The algorithm work as follow:

1. In the first step, all message nodes send a message to their connected check nodes. In this case, the message is the bit they believe to be correct for them. For example, message node c_2 receives a *1* ($y_2 = 1$), so it sends a message containing *1* to check nodes f_1 and f_2. Table 3.1 illustrates this step.

2. In the second step, every check nodes calculate a response to their connected message nodes using the messages they receive from step 1. The response message in this case is the value (0 *or* 1) that the check node believes the message node has based on the information of other message nodes connected to that check node. This response is calculated using the parity-check equations which force all message nodes connect to a particular check node to sum to 0 (*mod* 2). In Table 3.1, check node f_1 receives *1* from c_4, *0* from c_5, *1* from c_8 thus it believes c_2 has *0* $(1 + 0 + 1 + 0 = 0)$, and sends that information back to c_2.

   At this point, if all the equations at all check nodes are satisfied, meaning the values that the check nodes calculate match the values they receive, the algorithm terminates. If not, we move on to step 3.

3. In this step, the message nodes use the messages they get from the check nodes to decide if the bit at their position is a *0* or a *1* by majority rule. The message nodes then send this hard-decision to their connected check nodes. Table 3.2 illustrates this step. To make it clear, let us look at message node c_2. It receives *2* zeros from check nodes f_1 and f_2. Together with what it already has $y_2 = 1$, it decides that its real value is *0*. It then sends this information back to check nodes f_1 and f_2.

4. Repeat step 2 until either exit at step 2 or a certain number of iterations has been passed.

3.2.3 Hard-decision Encoder

If the *generator matrix* G of a linear block code is known then encoding can be done using equation $c = Gm$. The cost (number of operations) of this method depends on the *Hamming weights* (number of ones) of the basis vectors of G. If the vectors are dense, the cost of encoding using this method is proportional to n^2. This cost becomes linear with n if G is sparse.

check nodes	activities				
f_1	receive	$c_2 \to 1$	$c_4 \to 1$	$c_5 \to 0$	$c_8 \to 1$
	send	$0 \to c_2$	$0 \to c_4$	$1 \to c_5$	$0 \to c_8$
f_2	receive	$c_1 \to 1$	$c_2 \to 1$	$c_3 \to 0$	$c_6 \to 1$
	send	$0 \to c_1$	$0 \to c_2$	$1 \to c_3$	$0 \to c_6$
f_3	receive	$c_3 \to 1$	$c_6 \to 1$	$c_7 \to 0$	$c_8 \to 1$
	send	$0 \to c_3$	$1 \to c_6$	$0 \to c_7$	$1 \to c_8$
f_4	receive	$c_1 \to 1$	$c_4 \to 1$	$c_5 \to 0$	$c_7 \to 0$
	send	$1 \to c_1$	$1 \to c_4$	$0 \to c_5$	$0 \to c_7$

Table 3.1: *Check nodes activities for Hard-Decision Decoder for code of Fig. 3.1*

message nodes	y_i	messages from check nodes		decision
c_1	1	$f_2 \to 0$	$f_4 \to 1$	1
c_2	1	$f_1 \to 0$	$f_2 \to 0$	0
c_3	0	$f_2 \to 1$	$f_3 \to 0$	0
c_4	1	$f_1 \to 0$	$f_4 \to 1$	1
c_5	0	$f_1 \to 1$	$f_4 \to 0$	0
c_6	1	$f_2 \to 0$	$f_3 \to 1$	1
c_7	0	$f_3 \to 0$	$f_4 \to 0$	0
c_8	1	$f_1 \to 1$	$f_3 \to 1$	1

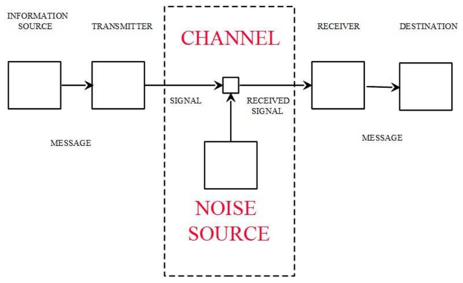
Table 3.2: *Message nodes decision for Hard-Decision Decoder for code of Fig. 3.1*

However, LDPC is given by the null space of a sparse *parity-check matrix H*. It is unlikely that the generator matrix G will also be sparse. Therefore the straightforward method of encoding LDPC would require number of operations proportional to n^2. This is too slow for most practical applications.

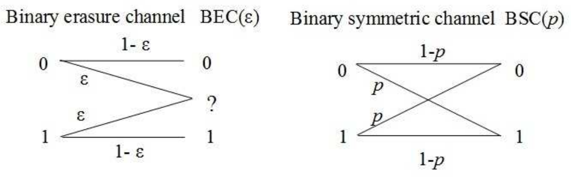
3.3 LDPC Codes Over Binary Erasure Channels

In this section we will see how to use the LDPC codes in a *Noisy Communication Channel* like in Fig. 3.2 **(a)**; there are two different kind of channels: the *Binary Symmetric Channel* (BSC) and the *Binary Erasure Channel* (BEC). In Fig 3.2 **(b)** we can see the two different types of channels.

A *Binary Channel* is so-called because it can transmit only one of two symbols (usually 0



(a) *A Noisy Communication System.*


Binary erasure channel BEC(ε)
0
0
1- ε
ε
ε
?
1
1
1- ε
Binary symmetric channel BSC(p)
0
0
1-p
p
p
1
1
1-p

(b) *BEC and BSC.*

Figure 3.2: *Binary channel.*

and 1). (A *non-binary channel* would be capable of transmitting more than two symbols, possibly even an infinite number of choices.) The channel is not perfect and sometimes the bit gets *erased*; that is, the bit gets scrambled so the receiver has no idea what the bit was.

The BEC is, in a sense, *error-free.* Unlike the *Binary Symmetric Channel*, when the receiver gets a bit, it is 100% certain that the bit is correct. The only confusion arises when the bit is erased.

A *Binary Erasure Channel* with *erasure probability* p is a channel with binary input, ternary output, and probability of erasure p (Fig. 3.2 **(b)**). That is, let X be the transmitted random variable with alphabet $\{0, 1\}$. Let Y be the received variable with alphabet $\{0, 1, e\}$, where e is the *erasure symbol*. Then, the channel is characterized by the conditional probabilities: the received bit is correct with probability $1 - \varepsilon$ or is incorrect with probability ε. "$1 - p$" is the *capacity* of the BEC.

The work of the Constrain nodes in a Factor Graph relative to a LDPC code on BEC is really simple. Every constraint node receive v bits from the variable nodes; with this it evaluate the *Parity Equation* to see if every variable node ha sthe correct bit. If the result of the equation is 0 then the algorithm can stop. In fig. 3.3 we can see an example of a factor Graph with the Constraint nodes and the relative equations that everyone has to satisfy to complete the algorithm.

If during the transmission a bit is lost the Constraint node take the $v - 1$ bits of the other Variable nodes and evaluate the *Parity Equation* without the last bit. Looking the $Constrainnode1$ in Fig. 3.3 we suppose that the bit 3 is lost; we will have the resultant equation 3.2:

$$x_1 + x_2 + x_4 + x_6 + x_8 + x_{10} = ? \tag{3.2}$$

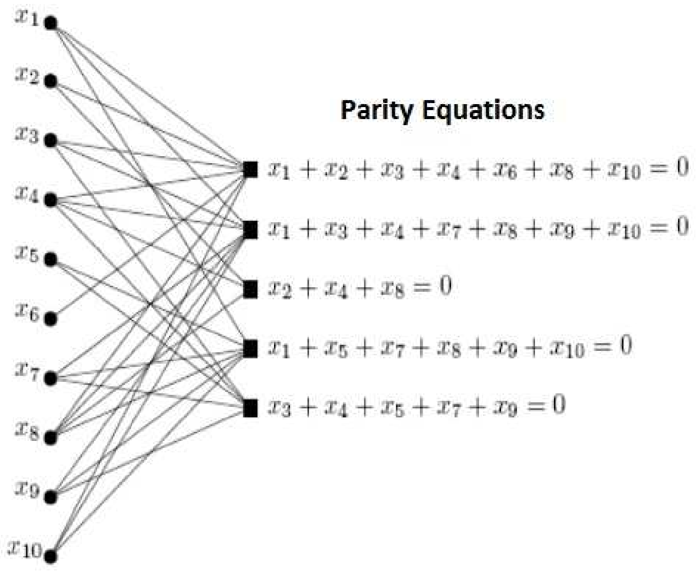

Parity Equations
$x_1 + x_2 + x_3 + x_4 + x_6 + x_8 + x_{10} = 0$
$x_1 + x_3 + x_4 + x_7 + x_8 + x_9 + x_{10} = 0$
$x_2 + x_4 + x_8 = 0$
$x_1 + x_5 + x_7 + x_8 + x_9 + x_{10} = 0$
$x_3 + x_4 + x_5 + x_7 + x_9 = 0$
x_1
x_2
x_3
x_4
x_5
x_6
x_7
x_8
x_9
x_{10}

Figure 3.3: *Parity Equations for a LDPC code over BEC.*

If the result of the equation is 1 then the Constraint node knows that the variable node 3 has to be 1 and vice versa if the result of the equation is 0. This is because the result of the equation with all the the variables has to be 0.
The performance of the LDPC code over BEC depends on the result of the algorithm: it fulfils the precise codeword, it founds a *legal codeword* but not the right one, it does not converges to a solution (a Stopping Set is found).

The Stopping Set situation can happen when a constraint node has more than one unknown bits and it's impossible to solve the Parity Equation.

In order to implement my programm, I use a different kind of channel: the *Additive white Gaussian noise channel* (AWGN). In this channel a noise is introducted to the signal. The bit after the transmission follow a *Gaussian distribution* and they are distributed like in Fig. 3.4.

In my algorithm we have the paramether *threshold* α. Instead to use a probability to erase the bits, we delete all the numbers inside the interval $\{-\alpha, \alpha\}$ taht correspond to erase the bit that during the transmission loose too much power. In this way the probability of *erasure* increase when α increase and vice versa.

In the Section 3.4 we can see the results of the simulations.

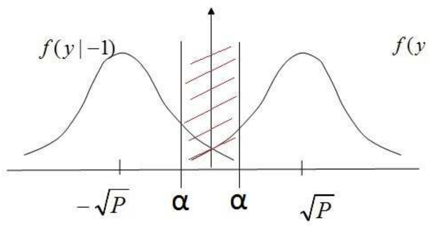
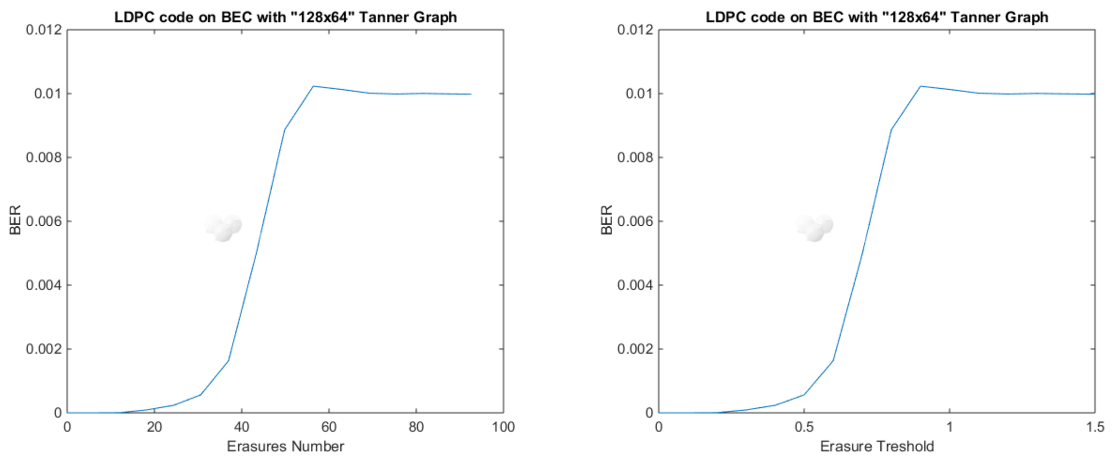


Figure 3.4: *Additive white Gaussian noise channel AWGN*



(a) *BER Vs. Erasered Numbers.* (b) *BER Vs. Erasered Threshold.*

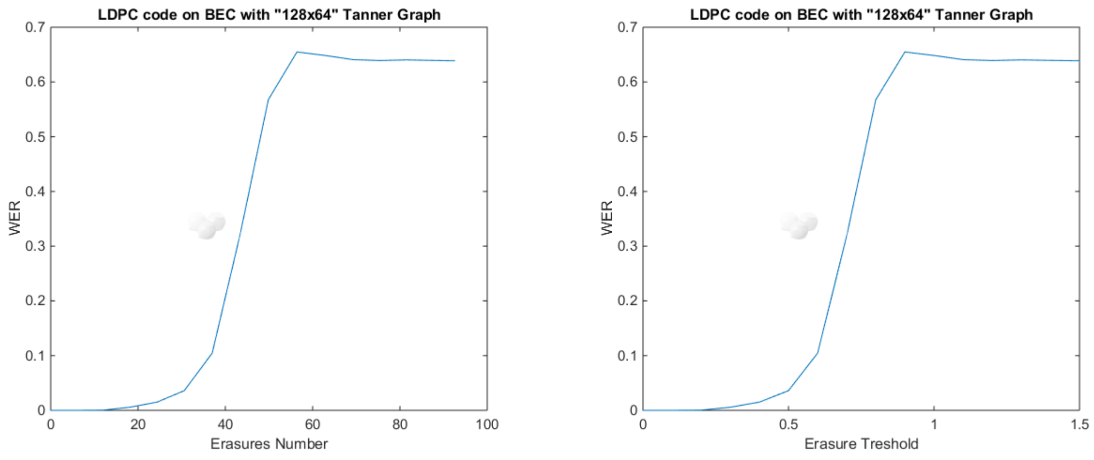
Figure 3.5: *Graph of BER.*

3.4 Conclusion and Results for LDPC Codes

This section present all the results of the simulations. All the results have been obtained with a Parity Check Matrix of dimension $124 * 64$. The Threshold α start from 0 (no erasures) until 1.5. The data that we analyse are: *Bit Error Rate* (BER) and *Word Error Rate* (WER). These data are compared with the *erasure number* and the *erasure threshold.*

Looking the Figures 3.5 and 3.6 we can immediately see that the BER and the WER is null until an Erasures Number of 12 that corresponds to a Threshold value $\alpha = 0.19$. In the interval $\{0, 0.19\}$ the algorithm stop always with success and the right codeword is always found.

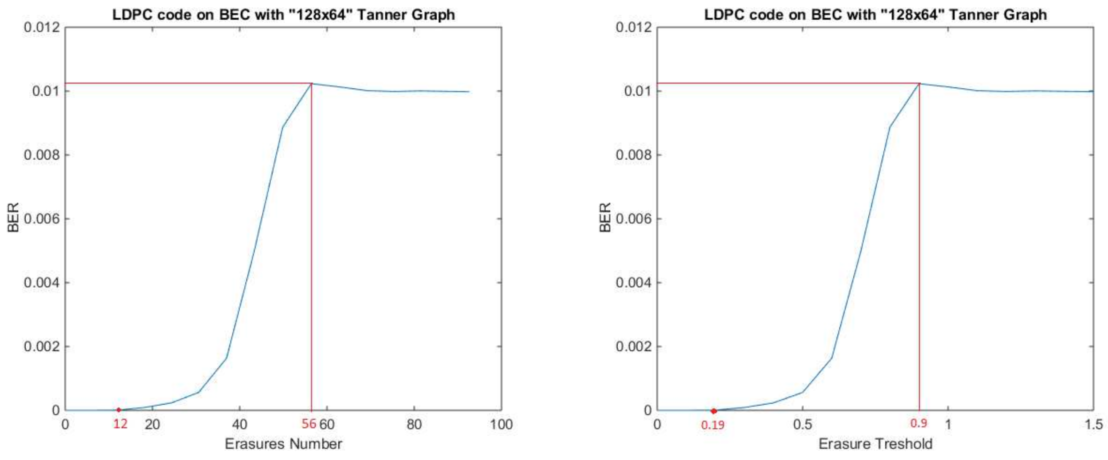
After this Threshold the algorithm is not more precise and we start to have some incorrect results until we arrive to a Threshold of $\alpha = 0.9$ in which the value of the Bit Error rate (BER) and Word Error Rate (WER) are the biggest. After that the BER and WER value is stable around a value of respectively $BER = 0.01$ and $WER = 0.64$.



(a) *WER Vs. Erasered Numbers.* **(b)** *WER Vs. Erasered Threshold.*

Figure 3.6: *Graph of WER.*

In the Fig. 3.7 we can better see these intervals and the values.

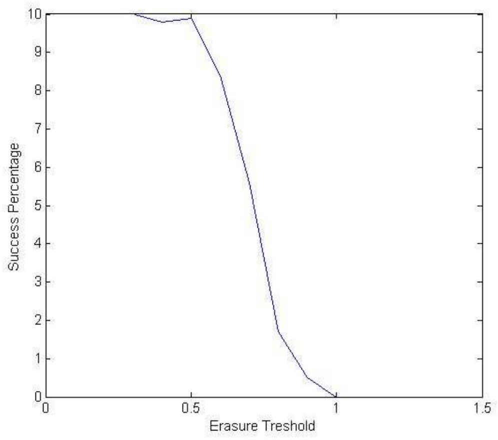


(a) *BER Vs. Erasered Numbers.* **(b)** *BER Vs. Erasered Threshold.*

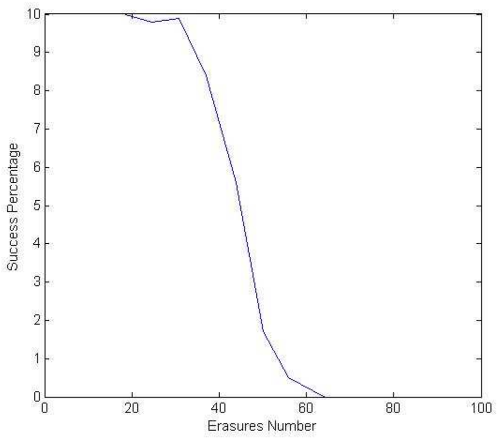
Figure 3.7: *Range of Erasures.*

In the Fig. 3.8 we can see the percentage of success during all the simulations in rapport with the erasures number **(a)** and the Threshold **(b)**.

The rest of the figures show how the BER, rapported with the EbNo, changes when the Threshold change. The figures go from a value of the threshold of $\alpha = 0.4$ until a value $\alpha = 1.5$.

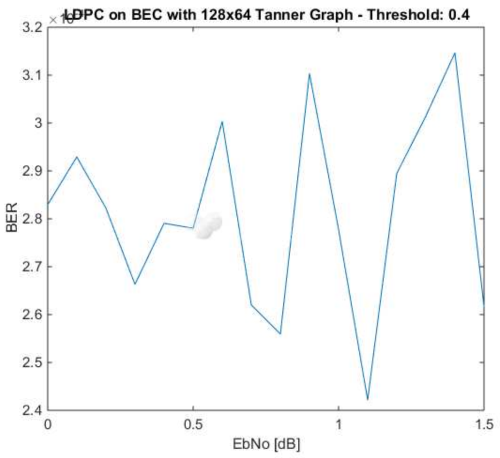


(a) *Success Percentage Vs. Erasered Numbers.*

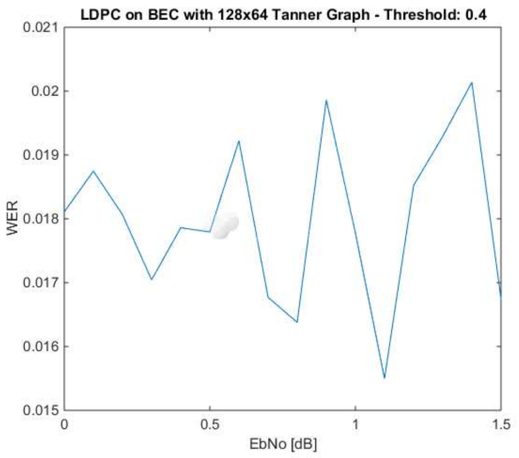


(b) *Success Percentage Vs. Erasered Threshold.*

Figure 3.8: *Success Percentage.*

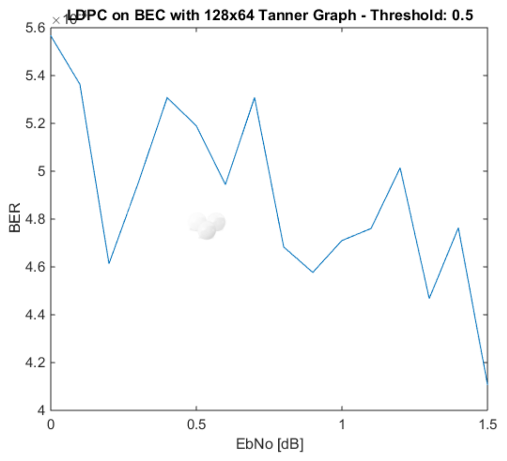


(a) *BER Vs. EbNo.*

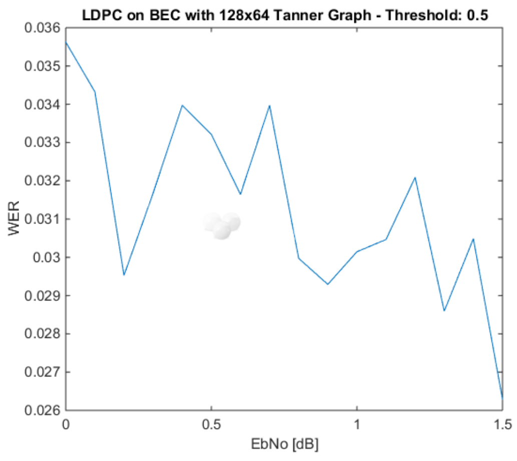


(b) *WER Vs. EbNo.*

Figure 3.9: *BER and WER Vs. EbNo for a Threshold of 0.4.*

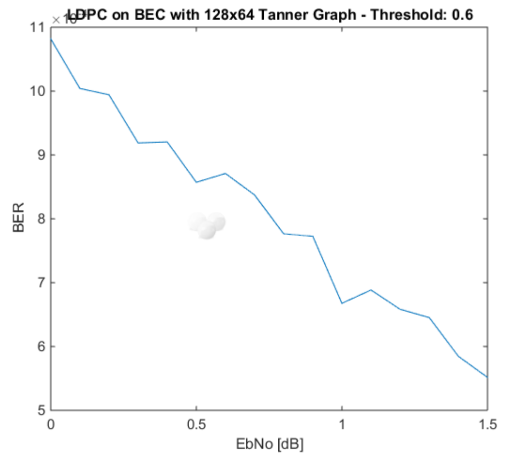


(a) *BER Vs. EbNo.*

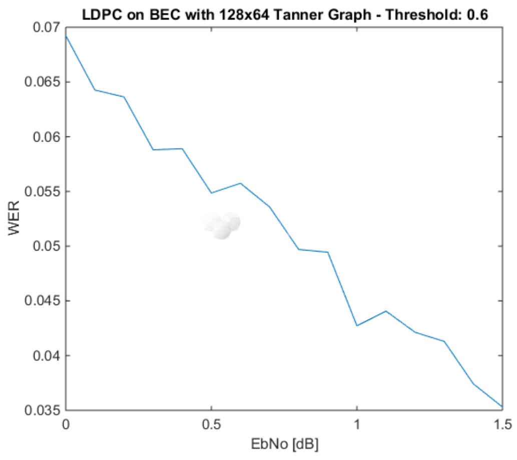


(b) *WER Vs. EbNo.*

Figure 3.10: *BER and WER Vs. EbNo for a Threshold of 0.5.*

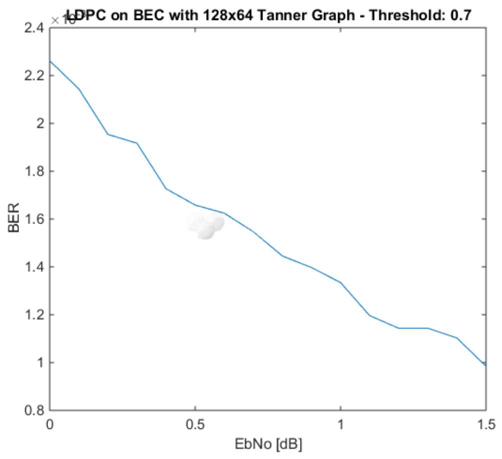


(a) *BER Vs. EbNo.*

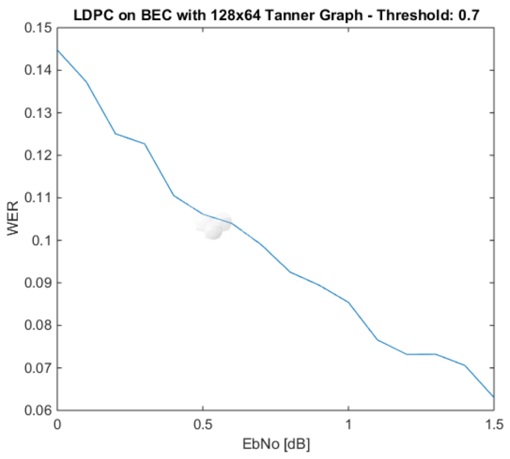


(b) *WER Vs. EbNo.*

Figure 3.11: *BER and WER Vs. EbNo for a Threshold of 0.6.*

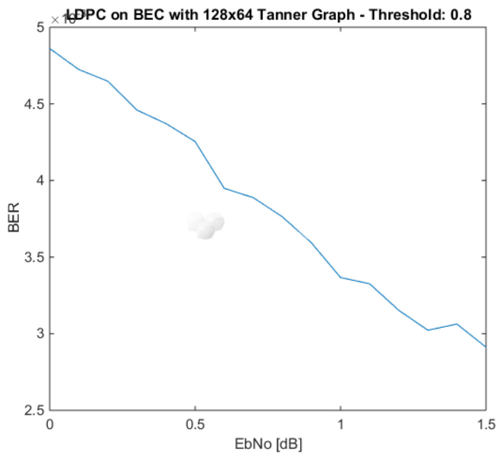


(a) *BER Vs. EbNo.*

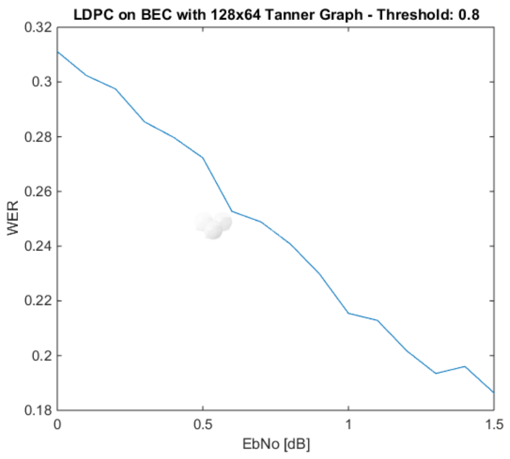


(b) *WER Vs. EbNo.*

Figure 3.12: *BER and WER Vs. EbNo for a Threshold of 0.7.*

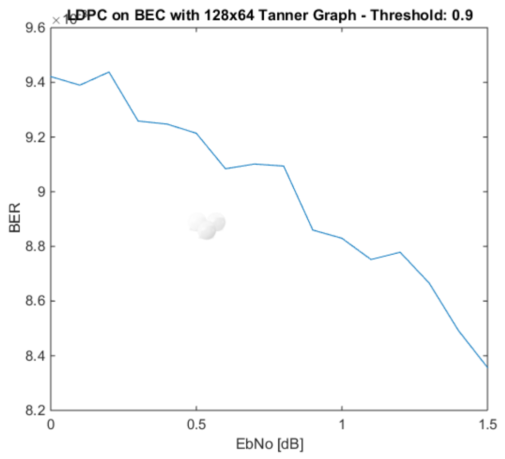


(a) *BER Vs. EbNo.*

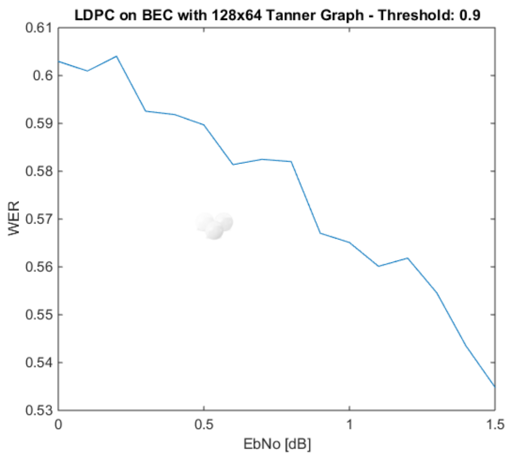


(b) *WER Vs. EbNo.*

Figure 3.13: *BER and WER Vs. EbNo for a Threshold of 0.8.*

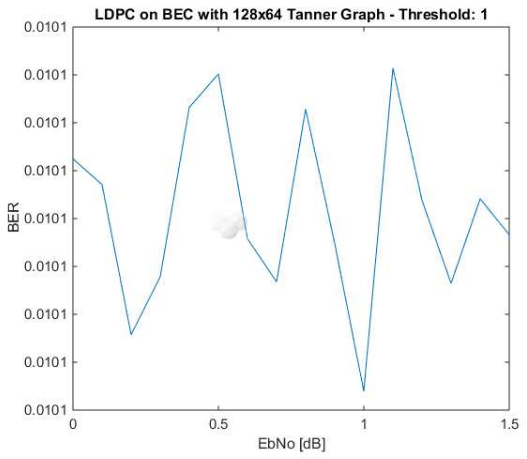


(a) *BER Vs. EbNo.*

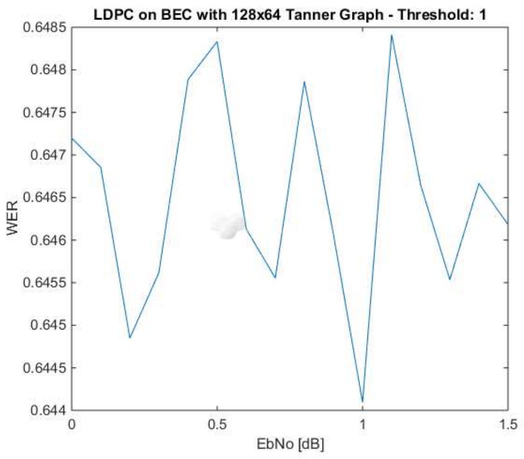


(b) *WER Vs. EbNo.*

Figure 3.14: *BER and WER Vs. EbNo for a Threshold of 0.9.*

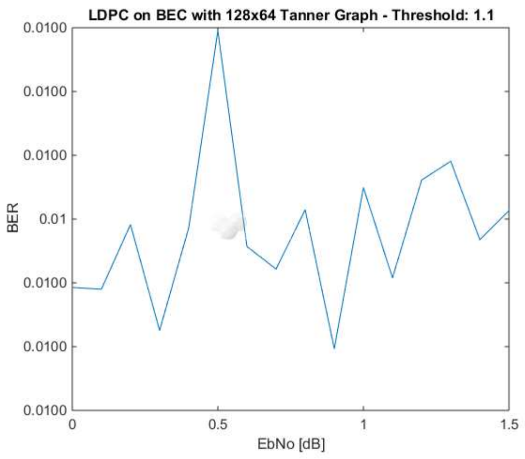


(a) *BER Vs. EbNo.*

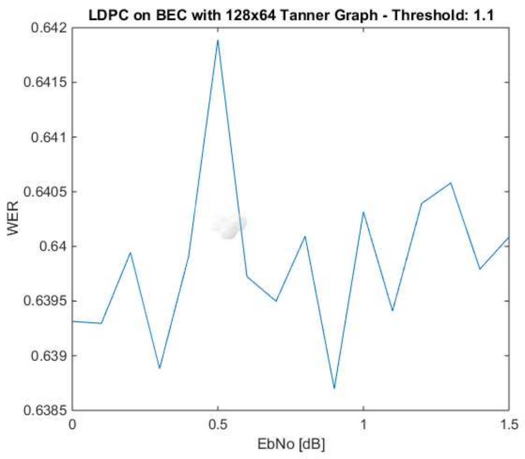


(b) *WER Vs. EbNo.*

Figure 3.15: *BER and WER Vs. EbNo for a Threshold of 1.0.*

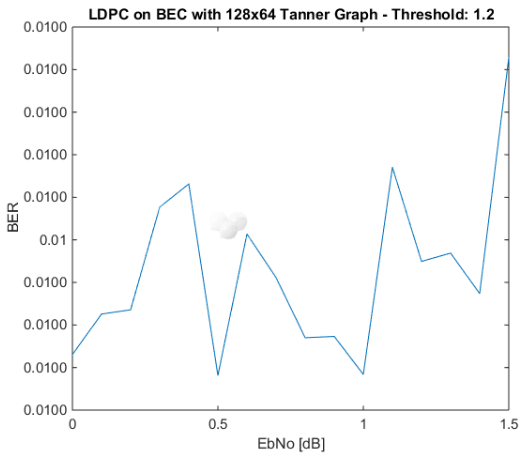


(a) *BER Vs. EbNo.*

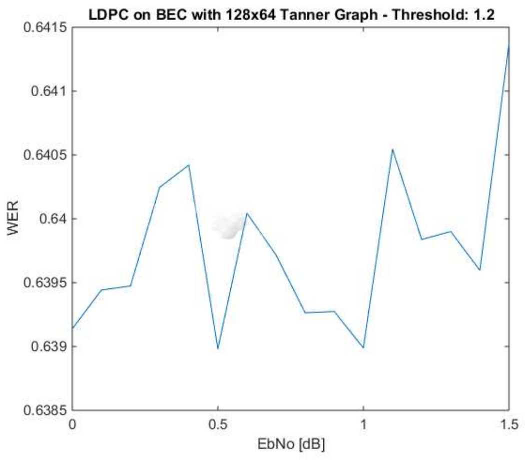


(b) *WER Vs. EbNo.*

Figure 3.16: *BER and WER Vs. EbNo for a Threshold of 1.1.*

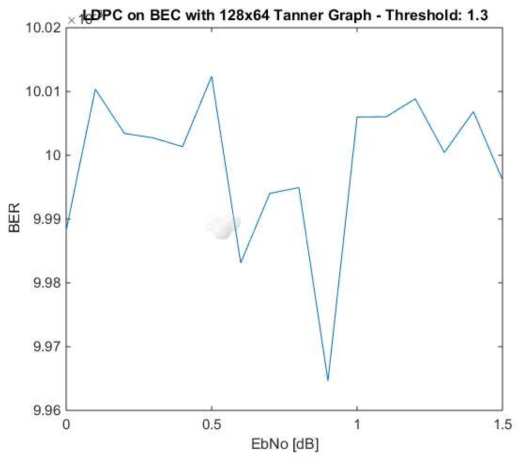


(a) *BER Vs. EbNo.*

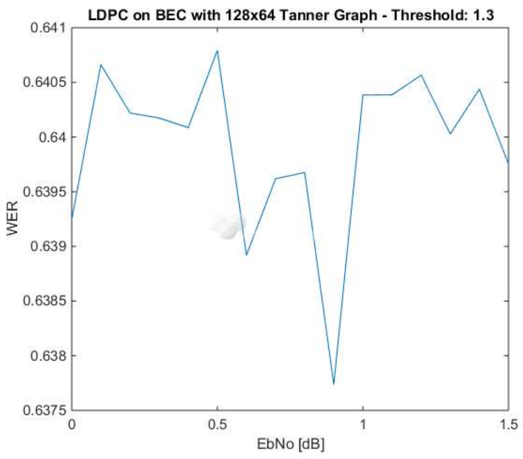


(b) *WER Vs. EbNo.*

Figure 3.17: *BER and WER Vs. EbNo for a Threshold of 1.2.*

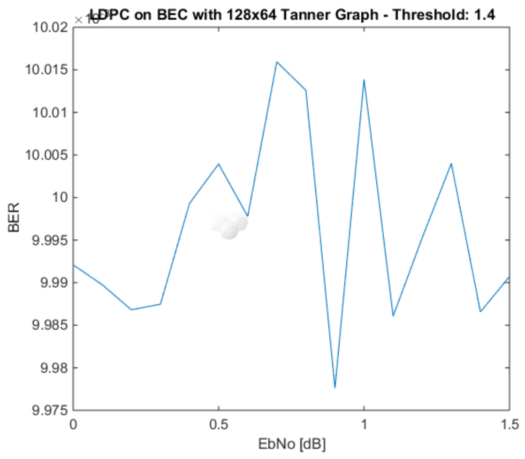


(a) *BER Vs. EbNo.*

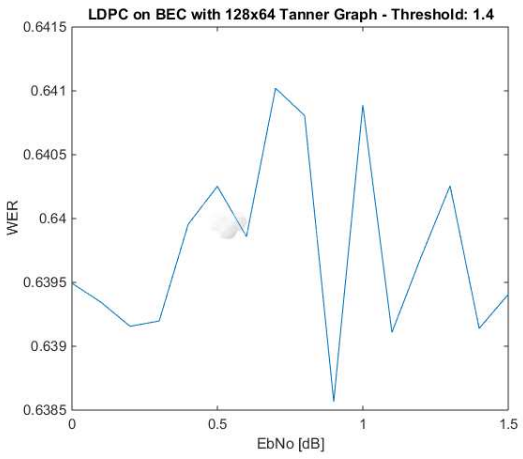


(b) *WER Vs. EbNo.*

Figure 3.18: *BER and WER Vs. EbNo for a Threshold of 1.3.*

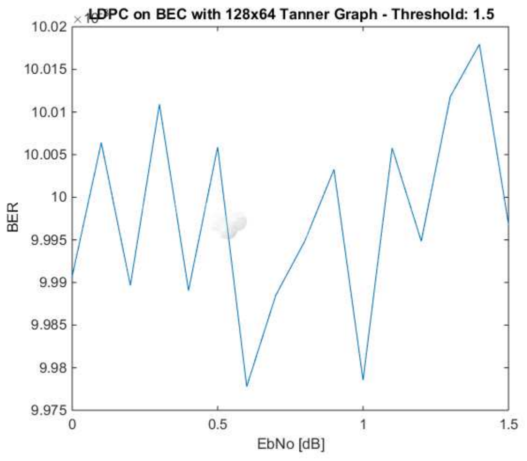


(a) *BER Vs. EbNo.*

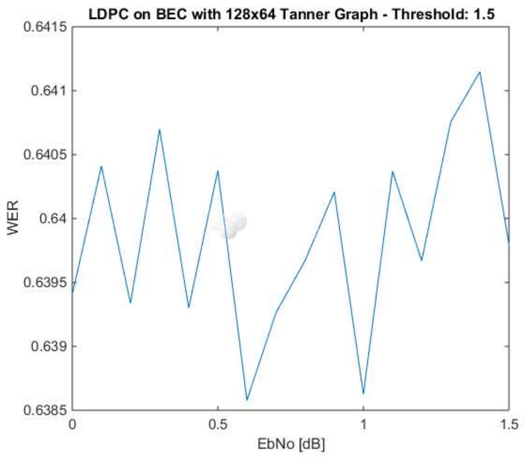


(b) *WER Vs. EbNo.*

Figure 3.19: *BER and WER Vs. EbNo for a Threshold of 1.4.*



(a) *BER Vs. EbNo.*



(b) *WER Vs. EbNo.*

Figure 3.20: *BER and WER Vs. EbNo for a Threshold of 1.5.*

Chapter 4

Conclusions and perspectives

Factor graphs provide a natural graphical description of the factorization of a global function into a product of local functions. They can be applied in a wide range of application areas.

Codes defined on graphs and decoded using the sum-product algorithm (or one of many possible variations) appear to be a solution to the problem of approaching fundamental limits in communication with practical decoding complexity.

The sum-product algorithm may be applied to arbitrary factor graphs, cycle-free or not. In the cycle-free finite case, we have shown that the sum-product algorithm may be used to compute function summaries exactly.

In other applications, e.g., in decoding of LDPC codes, solving Sudoku Puzzle or in general in graph with cycles, it is not. In the latter case, a successful strategy has been simply to apply the sum-product algorithm without regard to the cycles.

The MP paradigm is straightforward to apply to some problems with multiple constraints, with solutions obtained over discrete sets. The computational complexity is localized to each constraint. Cycles lead to failures in some cases, due to biases in the MP process.

The Belief Propagation Algorithm can be used in several fields. Everything depends on the faculty to express the problem in term of factor graph. As shown in this thess, Belief Propagation is not always the most accurate or optimal algorithm to solve a problem. Nevertheless, it is often used for its reduced complexity.

The failure of the Sum-product algorithm when applied to Sudoku similarly to LDPC over BEC is due to the existence of stopping-sets where we the algorithm shucked before the puzzle is completely solved.

This Stopping Sets in both cases are situations that Belief Propagation can not evaluate because constraint nodes can not communicate between them; but maybe, as we saw in Sec. 2.4.1, with different algorithms are very simple problems to evaluate and solve.

We can also see that we have some similar situation in Sudoku Sokving and in LDPC Decoding:

- In the Sudoku Solving we have a percentage of success of the BP algorithm when the starting numbers are between 29 and 32; in the LDPC Decoding th percentage is the same when the erasures number is between $\{0, 0.19\}$.
- Gradually the percentage of success decrease, respectively, in the Sudoku Solving when the starting numbers decrease and when the erasures number increase in LDPC Decoding.
- Finally in the last phase the percentage of success is really low, almost zero. We can see that in Fig. 2.14 in Sec. 2.4.3 for the Sudoku Solving and in Fig. 3.8 in Sec. 3.4 for the LDPC Decoding.

To conclude we can say that LDPC Decoding and Sudoku Solving have a similar behavior. The success is based on the starting complexity of the problem and the problems for both are more or less the same. This happens because when the complexity is bigger, there are more cycles in the graph and we find more stopping-sets; if the complexity decrease, many Variable Nodes are known, they do not take part anymore in the algoritm and we can say that they are cropped from the graph. In this way, many cycles are removed and consequently the Stopping-set disappear and the complexity decrease.

Belief Propagation with all its algorithms is very powerfull for their simplicity of implementation, but at the same time, they have limits due to the presence of these cycles in the Factor Graph and Stopping-Sets during the solution.

The best thing, is to use it to simplify the problem and if is not possible to find the solution, it is possible to support it with other kinds of algorithms more powerfull that can better work when the complexity of the problem is considerably reduced.

Bibliography

[1] D. Achlioptas, C. Gomes, H. Kautz, and B. Selman, “Generating satisfiable problem instances,” in *In Proceedings of the Seventeenth National Conference on Artificial Intelligence*, AAAI-00, Ed., 2000.

[2] N. Alon and M. Luby, “A linear time erasureresilient code with nearly optimal recovery,” *IEEE Trans. Inform. Theory*, vol. 47, pp. 623–656, February 2001.

[3] L. R. Bahl, J. Cocke, F. Jelinek, and J. Raviv, “Optimal decoding of linear codes for minimizing symbol error rate,” *IEEE Trans. Inform. Theory*, vol. 20, pp. 284–287, March 1974.

[4] C. Berrou, A. Glavieux, and P. Thitimajshima, “Near shannon limit error correcting coding and decoding: Turbo codes,” *IEEE Int. Conf. Commun.*, pp. 1064–1070, May 1993.

[5] J. Bondy and U. Murty, “Graph theory with applications,” 1982.

[6] M. Dincabs and H. Simonis, “Apache - a constraint based, automated stand allone allocation system,” *Advanced Software Technology in Air Transport*, pp. 267–282, 1991.

[7] B. J. Frey, F. R. Kschischang, H. A. Loeliger, and N. Wiberg, Eds., *Factor graphs and algorithms*, vol. Proc. 35th Allerton Conf. on Communications, Control, and Computing, September-October 1997.

[8] R. G. Gallager, *Low-Density Parity-Check Codes.* MA: M.I.T. Press, 1963.

[9] F. R. Kschischang, B. J. Frey, and H. A. Loeliger, “Factor graphs and the sum product algorithm,” *IEEE Trans. Inform. Theory*, vol. 47, no. 3, pp. 498–519, February 2001.

[10] D. MacKay, “Good error-correcting codes based on very sparse matrices,” *IEEE Trans. Inform. Theory*, vol. 45, no. 2, pp. 399–431, March 1999.

[11] T. K. Moon and J. H. Gunther, "Multiple constraint satisfaction by belief propagation: An example using sudoku," *IEEE Mountain Workshop on Adaptive and Learning Systems*, pp. 122–126, 2006.

[12] P. Norvig. Solving every sudoku puzzle. [Online]. Available: http://norvig.com/sudoku.html

[13] G. Royle. (2005) Gordon royle. [Online]. Available: http://www.csse.uwa.edu.au/gordon/sudokupat.php

[14] H. Simonis, "Building industrial applications with constraint programming," *Constraints in computational logics: theory and applications*, pp. 271–309, 2001.

[15] R. M. Tanner, "A recursive approach to low complexity codes," *IEEE Trans. Inform. Theory*, vol. 27, pp. 533–547, September 1981.

[16] N. Wiberg, "Codes and decoding on general graphs," 1996.

[17] N. Wiberg, H. A. Loeliger, and R. K. Otter, "Codes and iterative decoding on general graphs," *Europ. Trans. Telecommunications*, vol. 6, pp. 513–525, 1995.

[18] Wikipedia. (2015) Factor graph. [Online]. Available: https://en.wikipedia.org/wiki/Factor_graph

[19] T. Yato and T. Seta, "Complexity and completeness of finding another solution and its application to puzzles," *IEICE Trans Fundam Electron Commun Comput Sci E86-A*, vol. 5, pp. 1052–1060, 2003.

Index

A

additive white Gaussian noise channel, 54

B

bayesian
- optimal, 1
- posterior probability, 24

BCJR, 9

belief propagation, 1, 9, 22

binary
- erasure channel, 2, 15, 52
- symmetric channel, 57

binary phase-shift keyed, 55

binary-input additive white Gaussian noise, 15

bipartite graph, 1, 9, 29, 52

bit error rate, 60

brute force, 24

C

cardinality, 35

channel capacity, 58

code rate, 55

codewords, 52

constraint
- column, 39
- function, 21
- linear parity-check, 53
- row, 39
- sub-grid, 39

D

decision values, 24

E

energy per message, 55

erasure probability, 58

erasure symbol, 58

error correcting code, 51

F

factor graph, 1, 9

G

Gaussian distribution, 59

generator matrix, 53

H

hamming weights, 56

hard
- decision, 49
- decoder, 2

hidden singles, 32

I

identity function, 12
iterative decoding algorithm, 52

L

LDPC
  irregular, 52
  regular, 52
local functions, 10
low-density parity-check, 1, 9, 51

M

marginal
  distributions, 10
  function, 10
max-product, 2, 9
message-passing, 12, 17
min-sum, 9
moon algorithm, 43

N

naked
  pairs & triplets, 33
  singles, 31
node
  cell, 20
  check, 38
  constraint, 1, 19
  variable, 1, 11, 19
noisy communication channel, 51
non-binary channel, 58
normalizing constant, 28
NP-complete, 2

P

parity equation, 58
probability
  distribution function, 10
  propagation, 2, 6, 9
  vector, 23

S

schedule
  adaptive, 38
  alternate, 39
  flooding, 37
  linear, 38
  message-passing, 37
  no-adaptive, 38
  serial, 37
shannon
  channel capacity, 51
  limit, 15, 51
soft-decision, 52
sparse parity-check matrix, 52
stationary state, 21
stopping-set, 2, 42
sudoku, 1
  Puzzle, 18
sum-product, 2, 9, 14, 17
  decoding, 15
summary-product, 9
syndrome vector, 54
systematic generator, 53

T

tanner graph, 1, 9, 22, 51
threshold, 59
turbo
  code, 51
  decoding, 51
turbo code, 10

W

word error rate, 60