A Method of Lowering the Complexity of the Sum-Product Algorithm Using Graph Transformations

by

Siarhei Yermakou

A thesis submitted to the Faculty of Graduate and
Postdoctoral Affairs
in partial fulfillment of the requirements for the degree of

Master of Applied Science

in

Electrical and Computer Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada, K1S 5B6

September 2011

The undersigned recommend to
the Faculty of Graduate and Postdoctoral Affairs
acceptance of the thesis

A Method of Lowering the Complexity of the Sum-Product Algorithm Using Graph Transformations

submitted by
Siarhei Yermakou

in partial fulfillment of the requirements for the degree of
Master of Applied Science in Electrical and Computer Engineering

_____

Professor Howard Schwartz, Chair,
Department of Systems and Computer Engineering

_____

Professor Amir H. Banihashemi, Thesis Supervisor

Carleton University
September 2011

Abstract

We investigate the effect of factor graph transformations on the complexity of the sum-product algorithm. Our work clearly shows that it is possible to lower the number of operations required by the sum-product algorithm in its original form. For some graphs, it is even possible to convert the graph to the cycle-free form and simultaneously reduce the number of operations.

We applied the transformations to the applications of factor graphs in Joint DNA Base Calling, decoding of the Hamming (7,4) code and Link Loss Monitoring in Wireless Sensor Networks. In these applications, for considered models, we successfully lowered the number of operations by 25%, 65% and 48%, respectively. On an example of the Hamming(7,4) code we demonstrated that the transformations may also improve the bit error rate performance of a decoder.

Table of Contents

List of Tables

List of Figures

List of Algorithms

List of Symbols and Acronyms

List of Acronyms

Acronym	Definition
BAWGN	Binary Additive Ahite Gaussian Noise
BP	Belief Propagation
FG	Factor Graph
GDL	Generalized Distributive Law of Aji et.al [2]
i.e.	Latin id est. that is.
i.i.d.	independent and identically distributed
ISI	Intersymbol interference
LDPC	Low Density Parity Check Codes
LHS	left hand side (of an equation)

LLR	Log Likelihood Ratio
MAP	Maximum a posteriori probability
MPF	Marginalize Product Function
pdf	Probability Density Function
PMF	Probability Mass Function
r.v.	Random Variable
RHS	right Hand Side (of an equation)
SPA	Sum-Product Algorithm
TOA	Time of Arrival

List of Symbols

Symbol	Definition
A_v	Number of additions required to compute messages on all edges of node v
C_v	complexity of update of a node v defined as number of additions and multiplications required by SPA in order to update messages on all edges connected to the node v

$d(v)$	degree of a node
$diam(G)$	diameter of a graph
$dist(v_1, v_2)$	shortest path between nodes v_1 and v_2
e	an edge of a graph
E	set of edges
F	global function (usually represented by a FG)
f	local function
FG	factor graph
fn	function node
FN	set of function nodes in FG
G	graph
m	mean of a Gaussian r.v.
$m(x)$	marginals of global function over all variables except x
M_v	number of multiplications required to compute messages on all edges connected to a node v
μ	message
$\mu_{x_i \to f_j}$	message sent by a variable node x_i to a function node f_j

$\mu_{f_j \to x_i}$	message sent by a function node f_j to a variable node x_i
$N_x(v)$	neighbors of node v that can be reached by path of length x
$O(N)$	Order of complexity
Q_i^{vn}	cardinality of the domain of a variable node i
Q_j^{fn}	cardinality of the domain of a function node j
q_i	number of values taken by a discrete variable x_i
S	set of indices of variables
S_j^{fn}	set of indices variables of associated with a function node j
S_i^{vn}	set of indices variables of associated with a variable node i
v	node of a graph
VN	set of variable nodes in FG
vn	variable node
X_S	set of variables with indices from S
$\{\dots\}$	set
$\lvert\bullet\rvert$	cardinality of a vector, domain of a variable of set

Acknowledgments

I would like to express my sincere gratitude to my supervisor Dr. Amir H. Banihashemi for his continuous support. His support, belief in my work, encouragement and ability to ask "the right questions" allowed me to progress and complete this thesis. I also would like to express my appreciation to Mehdi, Anosheh, Sina and other members of our research group for their help, ideas and inspirational presentations. I wish to express gratitude to Dr. Yongyi Mao from the University of Ottawa for teaching a very interesting course on graphical models which inspired my curiosity in this topic of research. I cannot thank Heather enough for her support and understanding during the writing of this thesis.

Also a special thank you to Don Marlin and Jean Ouellet from BTI Systems who encouraged me and gave me a great deal of flexibility in my working schedule.

It is hard to mention everyone who supported me and helped me in writing this thesis. I am very grateful to you all.

Dedication

I wish to dedicate this work to my grandparents, who survived hunger, terrible war and then worked so hard to build peace. You are always in my thoughts and none of this would be possible without your support, inspiration and encouragement to pursue my dreams.

Chapter 1

Introduction

1.1 Background and Outline

Factor graphs and the sum-product algorithm [3–5], as a general framework to represent and analyze systems, have attracted a lot of interest from the research community. The factor graph framework has been applied to many problems such as decoding of error correcting codes, probabilistic inference, pattern recognition, data fusion, constrained optimization, and many others. The key to the success of the factor graph-based approach is that it enables the development of computationally efficient algorithms for a wide variety of applications. Often, the computational cost of the solutions derived based on factor graphs is much lower and/or other performance metrics are much better compared to the other approaches used to solve the same problem. For example, it has been shown that Maximum-Likelihood (ML) decoding of Low-Density Parity-Check codes (LDPC) is NP-

complete [6]. The method based on factor graphs and the sum-product algorithm allows close to ML decoding of the codes with complexity only $O(N)$ where N is the block length of the code. In addition to producing excellent results, the factor graph-based approach is well structured and easily understandable.

Abstractly speaking, a factor graph is a graphical representation of the structure of a problem to be solved. A factor graph comprises a set of nodes and a set of edges that connect the nodes. We refer to a particular configuration of nodes and edges as the topology of a graph. There are two types of nodes in a factor graph: function and variable nodes. In a factor graph, edges can connect nodes of different types only, i.e., a function node can be connected to any variable node but not to other function nodes. The sum-product algorithm operates on a factor graph by passing messages on the edges of the graph and finds a solution to the problem represented by the graph. Depending on the structure of a factor graph, the sum-product algorithm can yield an exact or approximate solution. It has been shown that if the underlying graph is cycle-free, then the solution found by the sum-product algorithm is exact [3, 4]. Conversely, if a graph has cycles, the sum-product algorithm yields an approximate solution. Most of the graphs encountered in real-life problems have cycles and for some of these graphs the sum-product algorithm may not converge or may converge to a wrong solution [7, 8]. It appears that the presence of a large number of short cycles can be especially harmful for the convergence and accuracy [9]. A lot of work has been done with respect to the investigation of the convergence properties of the algorithm [7, 8, 10–14] as well as to improving them, see, e.g., [15–17].

The authors in [3] introduce a set of transformations of a factor graph. These transformations include joining nodes of a graph, removing edges or nodes, etc. It has been shown that a graph with cycles can always be converted to the cycle-free form using the transformations. The authors in [3] observe that the transformation may significantly increase the complexity of the sum-product computations on the transformed nodes.

The issue of complexity is ever-important for the implementation of algorithms. The primary focus of the existing work on the complexity of the sum-product algorithm [18–21] and the references therein is the simplification and approximation of the operations of the algorithm on the nodes of a graph in the case where the variables represented by the nodes are binary. To the best of our knowledge, there is no reference in the literature that comprehensively explores the effect of factor graph transformations on the complexity of the sum-product algorithm in its general form when underlying variables are not binary. The objective of this thesis is to fill this gap. We show that the transformations may considerably decrease the complexity of the sum-product algorithm for certain graph topologies.

We define complexity as the number of arithmetic operations required to compute the messages on the edges of a factor graph. Following common practice, we refer to the computation of messages on some or all edges connected to a node as "update of the node". We notice that while a factor graph transformation increases the complexity of the update of transformed nodes, the number of operations required to update the nodes *adjacent* to the transformed nodes may decrease. It

appears that an increase in the required number of operations on the transformed part of the graph can be compensated by a decrease in the number of operations required to update the rest of the graph. In other words, the complexity can be "shifted" in order to find the graph that requires the sum-product algorithm with the least complexity.

The gain promised by the presented method depends on the structure of a factor graph. One case stands out in particular: the transformations can be beneficial if the graph is dense and has many short cycles. The same case is problematic in terms of the convergence of the sum-product algorithm. In some cases, applying our method achieves a two-fold gain: short cycles from a graph are removed, or the graph is converted to a cycle-free form, with a simultaneous reduction in the number of arithmetic operations required by the sum-product algorithm.

Examples of the method of lowering the complexity using graph transformations presented in this thesis include the following factor graph applications: i) Joint DNA Base Calling [1], ii) decoding of the Hamming(7,4) code and iii) Link Loss Monitoring in Wireless Sensor Networks [22]. In application i) we are able to convert the graph to the cycle free form and reduce complexity by 25% at the same time. In the case of the Hamming (7,4) code we are able to convert the graph to the cycle-free form and reduce the number of operations by 51%. The complexity of finding the ML estimates of the transmitted bits on the transformed graph is less compared to the complexity of a single iteration of the sum-product on the original graph with cycles. The transformation not only decreased complexity but also improved the bit error rate of the decoder. In the application iii) we were

able to lower the operation count by 48%. In this thesis, we also propose a greedy depth-N search algorithm that optimizes the complexity of the sum-product algorithm by transforming a factor graph.

1.2 Objectives

The objectives of this thesis are the following:

1. To review the complexity of the sum-product algorithm in its generalized form.

2. To examine the complexity of the algorithm under a set of factor graph transformations such as clustering of variable and function nodes, and stretching transformations.

3. To explore the prospect of reducing the complexity of the generalized sum product algorithm under transformations of factor graphs.

4. To consider practical applications where the transformation of a factor graph lowers the complexity of the sum-product algorithm and to evaluate the effect of the transformations in these applications.

5. To develop a practical algorithm that reduces the complexity of the sum-product algorithm by transforming a factor graph.

1.3 Organization of the Thesis

The remainder of this thesis is organized in the following way. In Chapter 2, we review the framework of factor graphs and the sum product algorithm. In Section

2.1, we introduce the sum-product algorithm in its general form for discrete variables and briefly review aspects of the implementation of the algorithm, message scheduling, and convergence properties. In Section 2.2, the concept of commutative semirings and its application to the generalized sum-product algorithm is introduced. In Section 2.3, we consider the case where the variables represented by a factor graph are continuous random variables and in particular Gaussian. In Section 2.4, we review factor graph transformations as they are introduced in [3]. In Section 2.5, we review the existing literature related to the complexity of the sum-product algorithm.

In Chapter 3, we explore the complexity of the sum-product algorithm. In Sections 3.1 we introduce our definition of complexity. In Section 3.2 on several simple examples we show that the complexity of the sum-product algorithm can be lowered by transforming a factor graph. In Sections 3.3 we present the method of efficient message computations and derive expressions for the number of operations required to update a variable and function nodes. In Section 3.4 we show example of optimization of number of operations on a part of a graph.

In Chapter 4, we apply our method of lowering the complexity to the following applications: i) Joint DNA Base-Calling [1] (Section 4.1) , ii) decoding of the Hamming(7,4) code (Section 4.2), and iii) Wireless Link Loss Monitoring [22] (Section 4.3). In Chapter 5, we present a recursive greedy algorithm which performs a depth N search on a graph and finds a sub-optimal solution to the complexity optimization problem.

We conclude in Chapter 6 with list of contributions summary and future work.

Chapter 2

Review of the framework of Factor Graphs and the Sum-Product Algorithm

2.1 Factor Graphs and the Sum-Product Algorithm

In this chapter, we discuss the framework of factor graphs and the sum-product algorithm and their generalizations. Factor graphs and the sum-product algorithm were discussed in detail in [3, 4]. Many efficient algorithms developed in a variety of areas of mathematics, engineering and computer sciences may be described as cases of the sum-product algorithm. It appears that a large variety of efficient and

sometimes quite complex algorithms utilize the simple distributive law:

$$A \cdot C + B \cdot C = (A + B) \cdot C$$

We note that the left-hand-side (LHS) of the equation requires three operations while the right-hand-side (RHS) requires only two. Factor graphs and the sum-product algorithm is a general framework that efficiently solves a class of computational problems known as "marginalize product of functions" (MPF) problems [2, 3] by effectively utilizing the distributive law on a commutative semiring. In this framework, we have a function of many variables called "global function" denoted by F that can be factorized as a product of a number of functions called "local functions" denoted by f. We say that a function is parameterized by a set of variables if the function depends on the variables in the set. Alternatively, we say that the variables form the domain of a function. We use the lower case letters such as x_i to denote a single variable and the upper case letters such as X to denote a set of variables. From this point onwards, we denote subsets of variable indices associated with local functions by S_j, where j is the index of a local function, i.e., if a local function is $f_3(x_1, x_5, x_6)$, then $S_3 = \{1, 5, 6\}$, and we write $f_3(X_{S_3})$. If a global function F is parameterized by the set $X = \{x_1, \ldots, x_n\}$, and can be represented as the product of k functions $f_i, i \in \{1, \ldots, k\}$, we may write:

$$F(X) = \prod_{i=1}^{k} f_i(X_{S_i}) \tag{2.1}$$

In many applications, we wish to find marginals of a global function, i.e., the

sum of $F(X)$ over all but a single variable $x \in X$. The problem appears to be trivial but may have a prohibitive computational cost if solved in the direct way. For example, if a set S has 1000 binary variables (which is modest for some applications) then to find marginals over a variable one needs to perform $2 * (2^{999} - 1)$ summations which is prohibitive.[1] By applying the sum-product algorithm we may be able to reduce this number dramatically.

We start by introducing a few definitions from the graph theory [24] that will be helpful throughout this thesis:

Definition 1 *A* graph $G = (V, E)$ *is described by a set of vertices* $V = \{v_1, \ldots, v_k\}$ *and set of edges* $E = \{e_1, \ldots, e_l\}$. *An* edge *connects a pair of vertices* v_i *and* v_j *and can be denoted by* e_{v_i,v_j}. *In this thesis we consider undirected graphs so that the beginning and end vertices of an edge can be interchanged.*

Definition 2 *A* path *is a sequence of edges* $e_{v_i,v_j} \to e_{v_j,v_n} \to e_{v_n,v_m}, \ldots$ *such that for all edges in the path except for the last edge the end of an edge is also the beginning of the next edge. The* length of a path *is the number of edges in the sequence. A path is called* a simple path *if every edge appears in the sequence once.*

Definition 3 *A* cycle *is a path of length 3 or more that begins and ends at the same vertex. An edge which is not part of a cycle but joins two vertices which are part of the cycle is called a* chord.

[1] A similar example is given in [23].

Definition 4 *The* degree *of a vertex v_i which we denote by $d(v_i)$ is the number of edges connected to the vertex. Alternatively, we may say that the degree is the number of neighbors of the vertex that can be reached by the path of length one $N_1(v)$. A node of degree one is called a leaf.*

Definition 5 *A graph is called* r-partite *if the set of nodes V admits partition into r classes such that every edge has its ends in different classes, in other words vertices in the same partition class are not adjacent. A "2-partite" graph is called bipartite [24].*

A factor graph is a graphical representation of the factorization of a global function. The graph has one set of nodes corresponding to the local functions and another set of nodes representing variables. In the factor graph framework these nodes are called function and variable nodes, respectively. For the moment, we assume that there is a unique variable node representing each variable and a unique function node corresponding to each local function [2]. There is an edge connecting a variable node to a function node if and only if the variable node is part of the domain of the corresponding local function. An example of a global function which is factorized as a product of local functions and the factor graph representing this factorization is presented in Figure 2.1.

The degree of a variable node x_i, $d(x_i)$ is the number of local functions that have the variable x_i as a parameter. The degree of a function node f_j, $d(f_j)$ is equal to the number of variables involved in the local function j. The sum-product

[2]This will not be the case after we apply factor graph transformations

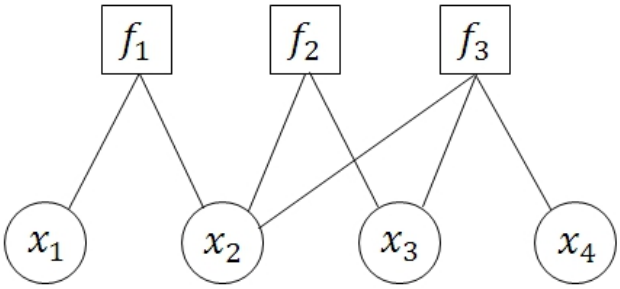


Figure 2.1: The factor graph representing the factorization $F(x_1,x_2,x_3,x_4) = f_1(x_1,x_2)f_2(x_2,x_3)f_3(x_2,x_3,x_4)$

algorithm operates on a factor graph by passing messages between nodes of a factor graph. We may say that a variable x_i is associated with an edge e of a factor graph if the edge is connected to the variable node corresponding to the variable x_i. Messages sent to and from the node x_i have the variable x_i as a parameter. Assuming that the variable x_i is discrete and has a finite alphabet with q_i values, then messages sent to and from the variable node x_i will have q_i values.

The sum-product update rules [3] are defined for the function and variable nodes. According to the rules, messages sent by a node (or outgoing messages) are computed from messages received by the node (or incoming messages) and the values of the local functions. The message(s) sent by a node on one step of the algorithm is received by the node's neighbor(s) on the next step of the algorithm.[3] We often refer to the process of computing outgoing messages from a node as "the update of a node".

In the following discussion, we will use $\mu_{x_i \to f_j}$ to denote a message sent from a variable node x_i to the function node f_j and $\mu_{f_j \to x_i}$ to denote a message from a

[3]Here we use "step" in a broad sense as a single part of a sequence of computations.

function node f_j to a variable node x_i. We use a backslash "\" to denote exclusion from a set, as in $\{x_1, x_5, x_6\} \backslash x_5 = \{x_1, x_6\}$. According to the update rule [3], the message sent from a variable node x_i to a function node f_j is computed as:

$$\mu_{x_i \to f_j}(x_i) = \prod_{f_k \in N(x_i) \backslash f_j} \mu_{f_k \to x_i}(x_i) \tag{2.2}$$

where $N(x_i)$ is the set of neighbors of the variable node x_i and $\mu_{f_k \to x_i}(x_i)$ is the message sent by a function node f_k and received by the variable node x_i on the previous step of the algorithm. In other words, an outgoing message sent by a variable node is the product of the incoming messages on edges other than the edge to which the message will be sent. A message sent by a function node f_j to a variable node x_i is computed as [3]:

$$\mu_{f_j \to x_i}(x_i) = \sum_{X_{S_j} \backslash x_i} f_j(X_{S_j}) \prod_{x_k \in N(f_j) \backslash x_i} \mu_{x_k \to f_j}(x_k) \tag{2.3}$$

where $N(f_j)$ is the set of neighbors of the function node f_j and $\mu_{x_k \to f_j}(x_k)$ is the message sent by a variable node x_k and received by the function node f_j on the previous step of the algorithm. In other words, a message sent by a function node is the product of the incoming messages on all edges except for the edge where the outgoing message will be sent multiplied by the local function and marginalized over the variable associated with the edge where the message will be sent. In the case where the variables are continuous, the summation in (2.3) is replaced by integration. We will consider this case in Section 2.3.

In order to send the message on an edge, a node of degree $d(v) > 1$ has to re-

ceive incoming messages on the rest of the edges. Nodes of degree one can send an outgoing message without receiving an incoming message. The values of the outgoing message of a function node[4] of degree one are the values of the local function. The values of the message from a variable node of degree one is equal to unity. Messages on a graph represent local information or in the case of probabilistic inference, "local beliefs".[5] As nodes exchange messages the information is spread in the graph. In order to compute exact marginals over a variable, the node that represents the variable has to receive the information from all parts of the graph.

Definition 6 *A graph is a connected tree if and only if there is a unique path between any two variables in the graph. By definition, if a graph is a tree then it has no cycles.*

Definition 7 *The distance between nodes x and y* $\mathbf{dist}(x,y)$ *is the shortest path between the nodes. The diameter of a graph* $\mathbf{diam}(G)$ *is the longest distance between any two vertices in G.*

The order in which the sum-product algorithm computes the messages sent by the nodes of a graph is called a *schedule*. Various schedules are possible; a schedule may depend on the presence of cycles in a graph and the type of problem being solved. We differentiate between single-vertex and multiple-vertices problems [2]. In a single-vertex problem we need to compute the marginals of a

[4]Nodes of degree one are commonly called leaf nodes.

[5]The other name of the sum-product algorithm is "belief propagation" or BP.

global function over a single variable. In a multiple-vertices problem we need to find marginals over more than one variable. For example, if our global function is $F(x_1,x_2,x_3)$, then to compute $m_1(x_1) = \sum_{x_2,x_3} F(x_1,x_2,x_3)$ would be a single-vertex problem while to compute the marginals $m_1(x_1)$, $m_2(x_2)$, and $m_3(x_3)$ would be an all-vertices problem.

For a single-vertex problem on a cycle-free graph, the algorithm may start at the leaf nodes and at each step, update the nodes that receive at least $d(v)-1$ incoming messages. In the case of the single-vertex problem, we need to compute the outgoing messages for a node only on the edge that belongs to the unique path from the node to the node that contains the variable of interest. The exact marginals at the node x_i can be computed after $\max_{v \in G} \mathbf{dist}(x_i, v)$ rounds[6] of updates.

For the multiple-vertices problem on a cycle-free graph, we may also start at the leaf nodes and on each step, compute outgoing messages of the nodes that receive at least $d(v)-1$ incoming messages. This time however, we need to update outgoing messages on all edges. We will be able to compute exact marginals at the variable nodes after $\mathbf{diam}(G)$ rounds of updates.

For a graph with cycles, the schedule applied above results in a deadlock because the nodes involved in the cycle(s) will never receive $d(v)-1$ messages and will never be able to compute outgoing messages. In order to resolve this, we may use the flooding schedule [3]: at the beginning of the algorithm all messages are initialized to some values (often unity) and on each step of the algorithm all

[6]We define a round as part of a sequence of computations in which all nodes of either type (variable or function) that received at least $d(v)-1$ messages compute outgoing messages.

nodes compute outgoing messages.[7] The algorithm may terminate after selected convergence criteria have been reached. For example, convergence can be concluded once the difference between the values of messages on the edges on two consecutive iterations is below a certain threshold. Convergence can also be concluded based on other criteria, for example, in the case of decoding a linear code, convergence may be declared after all parity checks are satisfied. The algorithm may also terminate after a certain fixed number of iterations.

After the algorithm has finished the message updates (or converged), the value of the marginals of global function over variable x_i can be computed at the variable node associated with variable x_i as:

$$m(x_i) = \prod_{f_k \in N(i)} \mu_{f_k \to x_i}(x_i) \tag{2.4}$$

i.e., the marginal is the product of all incoming messages. The marginal represents the global function marginalized over all variables but x_i:

$$m(x_i) = \sum_{X \setminus x_i} F(X) \tag{2.5}$$

The values of the marginals over a set of variables associated with a local function node can be computed as:

$$m(X_{S_j}) = f_j(X_{S_j}) \prod_{x_i \in N(j)} \mu_{x_i \to f_j}(x_i) \tag{2.6}$$

[7] To be exact, a node needs to compute outgoing message only if any of the values of the incoming messages are changed.

In other words, the marginal is the product of all incoming messages and the local function. The marginal represents the global function summed over all of the variables except the variables in the set X_{S_j}:

$$m(X_{S_j}) = \sum_{X \setminus X_{S_j}} F(X) \tag{2.7}$$

In some cases, we may need to compute joint marginals over a set of variables that does not belong to a single local function. For example, consider a case where we have a global function which is a probability mass function $P(x_1, x_2, x_3, \dots)$ and we need to compute the marginal distribution $p(x_1, x_2)$. If there is a local function which depends on both variables x_1 and x_2 then we can compute the marginal using the expression (2.6). If no local function includes both x_1 and x_2 then we may proceed in the following way [25, p. 37]:

1. Compute the marginal $p_2(x_2)$ in the standard way;

2. Consider $p_2(x_2)$ as given evidence, fix the values of the messages coming from the node x_2 and compute conditional $p(x_1|x_2)$ by re-running the sum-product algorithm on the graph again.

3. Compute $p(x_1, x_2) = p(x_1|x_2)p_2(x_2)$.

Alternatively, using factor graph transformations described in the Section 2.4, we may create a node that includes both variables x_1 and x_2 and compute $p(x_1, x_2)$ using the regular approach.

If a factor graph is cycle-free the marginals computed by the sum-product algorithm are exact. In the case where a graph has cycles, it has been shown that:

- The marginals are approximate and the algorithm is not guaranteed to converge. Good results and convergence however, have been observed for many practical applications, as seen in [7].

- The values of the marginals are computed up to a scale factor [26]. [8]

A lot of work has been devoted to the investigation of the convergence properties of the sum-product algorithm on a graph with cycles [7, 8, 10, 13, 14]. The accuracy of the approximation of the marginals by the sum-product depends on the structure of the graph and the strength of influence along the cycles.[9] The algorithm is less likely to converge on the graphs with many short cycles and strong influence/coupling between the states of variables involved in the cycles. There is a connection between the convergence rate and accuracy, i.e., if the sum-product algorithm is converging slowly then the accuracy of marginals is likely to be poor [8]. It has been shown that the algorithm converges to the points which correspond

[8]For example, in the case when a global function is a pmf $P(x_1,\ldots,x_n)$ and the variables $X=\{x_1,\ldots,x_n\}$ are binary then for a cycle-free graph the marginal at a node x_i is $m_i(x_i=0)=\sum_{X\setminus x_i}P(x_1,\ldots,x_i=0,\ldots,x_n)$ and $m_i(x_i=1)=\sum_{X\setminus x_i}P(x_1,\ldots,x_i=1,\ldots,x_n)$. In the case of graph with cycles even when the algorithm converges to the correct marginals, the result at the node x_i is $m_i(x_i=0)=C\sum_{X\setminus x_i}P(x_1,\ldots,x_i=0,\ldots,x_n)$ and $m_i(x_i=1)=C\sum_{X\setminus x_i}P(x_1,\ldots,x_i=1,\ldots,x_n)$ where C is an arbitrary constant.

[9]Let assume the SP algorithm converged on a graph with a cycle which include n nodes $v_1,v_2,\ldots,v_n$. We may change the values of the message sent by the node v_1 to the node v_2 by some amount and sequentially update the nodes $v_2,\ldots,v_n$. Lastly, we re-compute the message $v_n\rightarrow v_1$ as well as the value of the marginal at the node v_1. If the value of the marginal did not change by much we may say that the influence along the cycle is "weak" and that the SP algorithm on the graph with such a cycle is more likely to behave as if the graph is cycle-free, i.e., converge to the correct marginal. Note that this is our "intuition" since we did not specify the value of "some amount of change".

to the stationary points of Bethe approximation of free energy [12] and that the convergence depends on the uniqueness of the stationary points. There are ways to induce convergence, for example by introducing scaling factors on edges of a graph [16, 27, 28].

A schedule has influence on convergence, accuracy, and complexity of the sum-product algorithm [29–32]. For example, it has been shown that the sum-product algorithm under the serial schedule where nodes are updated in sequence tends to converge more quickly.

In order to minimize the operation counts it is desirable to perform as few message updates as possible. From this point of view, a cycle-free graph has a definite advantage since each message has to be updated only once. For a graph with cycles, the more iterations required for convergence, the higher the complexity of the algorithm, if all other parameters are the same. We note that elimination of short cycles, which our method often uses to reduce "complexity per iterations", is also likely to have positive results on the convergence of the algorithm. This however, is case specific and is yet to be proven.

2.2 Generalization of the Sum-Product Algorithm on an arbitrary commutative semiring

The versatility and wide variety of applications of the sum-product algorithm can be partially explained by the fact that the algorithm can be generalized on an arbitrary *commutative semiring*[2, 3, 33].

Definition 8 *A commutative semiring is a set with defined operations* $\oplus$ *and* $\otimes$

such that:

- *The operation $\oplus$ (which is equivalent to the summation in the sets of $\mathbb{R}$, $\mathbb{C}$, $\mathbb{Z}$) is associative and commutative and additive element "0" exists such that $a \oplus 0 = a$.*

- *The operation $\otimes$ (which is the equivalent of multiplication in the sets of $\mathbb{R}$, $\mathbb{C}$, $\mathbb{Z}$) is associative and commutative and there is a multiplicative unity "1" such that $a \otimes 1 = a$.*

- *the distributive law holds, i.e., $(a \otimes b) \oplus (a \otimes c) = a \otimes (b \oplus c)$*

A few examples of the semirings include:

- The set of $\mathbb{R}$ with ordinary additions and multiplications forms a semiring. The well-known form of the distributive law for this semiring is expressed as:
  $A \cdot C + B \cdot C = (A + B) \cdot C$

- The *min-sum semiring* is formed by the set of $\mathbb{R}$ with the operations $\oplus$ and $\otimes$ represented by *MIN* and "+" respectively. The distributive law in this case is:
  $MIN(A + C, B + C) = C + MIN(A, B)$

- The *boolean semiring* is formed by the set of $\{TRUE, FALSE\}$ with the operations $\oplus$ and $\otimes$ represented by the boolean operations "OR" and "AND". The distributive law in this case is:
  (A AND B) OR (A AND C) = A AND (B OR C)

A table of several semirings is given in [2, Table. 1].

The sum-product algorithm is based on the distributive law [2, 3, 33] and can be applied to any problem where we have a global function over elements of semiring which can be represented as a generalized product of smaller functions and where we need to find the generalized marginals over subsets of variables. By "generalized product" and "generalized marginals" we are refereing to the expressions where ordinary multiplications and summations are replaced by the operations $\oplus$ and $\otimes$ respectively. When the sum-product algorithm is applied to a problem defined over a commutative semiring, it operates "as usual" with exception that the operations addition and multiplication in the update rules for the variable and function nodes (2.2) and 2.3) are replaced by the semiring operations $\oplus$ and $\otimes$. The values of the messages and local functions in this case are the elements of the semiring.

2.3 The Sum-Product Algorithm in the case of continuous variables

In the discussion above, the variables of the global function were discrete with the values taken from finite alphabets. Now we briefly consider the case where some or all of the variables are continuous. In this case, the messages on the edges of a factor graph are functions of the continuous variables. The messages sent by a function node f_j to a variable node x_i node is expressed as:

$$\mu_{f_j \to x_i}(x_i) = \int_{X_{S_j} \setminus x_i} f_j(X_{S_j}) \prod_{x_k \in N(f_j) \setminus x_i} \mu_{f_j \leftarrow x_k}(x_k) dX_{S_j} \setminus x_i \tag{2.8}$$

where the integral is evaluated over all the variables in X_{S_j} but x_i. The expression is similar to the update rule of a function node for the case of discrete variables (2.3) with the exception that the summation is replaced by the integration. The messages sent from a variable node to a function node can be found using the expression identical to the discrete case (2.2). However, this time the outgoing messages are products of functions of a continuous variable.

A major application of factor graphs is the task of probabilistic inference. In the probabilistic framework, the local functions represent probability distributions and the messages on the edges are marginal distributions. The computation of the integral in the update of a function node (2.8) depends on the type of local function. In the important case where the distributions are Gaussian, the probability distribution can be completely described by its mean and variance. Therefore, messages sent on the edges of a factor graph can have just two values: mean and variance of the distribution. The probability density function of the Gaussian distribution is expressed as [34]:

$$f(x) = \frac{\exp\left\{-\frac{(x-\mu_x)^2}{2\sigma_x^2}\right\}}{\sqrt{2\pi\sigma_x^2}} \tag{2.9}$$

In the case of a probabilistic framework, we are interested in the values of the distributions up to a scale factor [26]. The message sent by a variable node x_i is a product of the incoming messages which are distributions (2.9) over common variable x_i. The product of two distributions over x_i with variances σ_1^2, σ_2^2 and

means μ_1, μ_2 is [35, p. 11]:

$$f_1(x_i)f_2(x_i) \propto \exp\left\{-\left(\frac{(x_i-\mu_1)^2}{2\sigma_1^2}+\frac{(x_i-\mu_2)^2}{2\sigma_2^2}\right)\right\} = \exp\left\{-\frac{(x-\mu_P)^2}{2\sigma_P^2}\right\}$$

where, the mean and μ_P and variance σ_P^2 of the product-distribution are:

$$\sigma_P^2 = \frac{\sigma_1^2\sigma_2^2}{\sigma_1^2+\sigma_2^2} \tag{2.10}$$

$$\mu_P = \sigma_P^2\left(\frac{\mu_1}{\sigma_1^2}+\frac{\mu_2}{\sigma_2^2}\right) \tag{2.11}$$

Applying (2.10) and (2.11) in a chain we can evaluate the product of any number of Gaussian distributions. We can simplify the equations for the case of a product of several distributions if, instead of the variances, we use precisions $P = \frac{1}{\sigma^2}$. The precision of the product distribution (2.10) is $P_P = P_1 + P_2$. The precision $P_{x_i \to f_j}$ of the message sent by a variable node x_i to a function node f_j is expressed as [35]:

$$P_{x_i \to f_j} = \sum_{f_k \in N(x_i)\setminus f_j} P_{x_i \leftarrow f_k} \tag{2.12}$$

where $P_{x_i \leftarrow f_k}$ is the precision sent by a function node f_k to the node x_i. The mean $m_{x_i \to f_j}$ sent by a variable node x_i to a function node f_j is [35]:

$$m_{x_i \to f_j} = P_{x_i \to f_j}^{-1}\left(\sum_{f_k \in N(x_i)\setminus f_j} P_{x_i \leftarrow f_k} m_{x_i \leftarrow f_k}\right) \tag{2.13}$$

where $m_{x_i \leftarrow f_k}$ is the mean sent by a function node f_k to the node x_i.

Now we are going to consider the update of a function node. The expression (2.8) has the product of distributions over different variables, i.e., the variables that are connected to the function node. In our case, the distributions are Gaussian. A two-dimensional jointly Gaussian distribution is expressed as [34]:

$$f(x,y) = \frac{\exp\left\{\frac{-1}{2(1-\rho_{xy}^2)}\left(\frac{(x-\mu_x)^2}{\sigma_x^2} - \frac{2\rho_{xy}(x-\mu_x)(y-\mu_y)}{\sigma_x\sigma_y} + \frac{(y-\mu_y)^2}{\sigma_y^2}\right)\right\}}{2\pi\sigma_x\sigma_y\sqrt{1-\rho_{xy}^2}} \tag{2.14}$$

where ρ_{xy} is the correlation coefficient between x and y. Assuming that the messages from nodes x_i and x_j are Gaussian distributions $f_{x_i}(x_i)$ and $f_{x_j}(x_j)$ with means μ_{x_i} and μ_{x_j} and variances $\sigma_{x_i}^2$, $\sigma_{x_j}^2$, then the product of the distributions is:

$$f_1(x_i)f_2(x_j) \propto \exp\left\{-\left(\frac{(x_i-\mu_{x_i})^2}{2\sigma_{x_i}^2} + \frac{(x_j-\mu_{x_j})^2}{2\sigma_{x_j}^2}\right)\right\} \tag{2.15}$$

By comparing (2.15) with (2.14) one can see that up to a scale factor the product is a two dimensional Gaussian distribution with correlation coefficient 0. Therefore, the product of the messages in (2.8) is a Gaussian distribution. The multidimensional Gaussian distribution $f_{\mathbf{x}}(\mathbf{x})$ over variables $\mathbf{x} = x_1, \ldots, x_n$ is expressed as [34]:

$$f_{\mathbf{x}}(\mathbf{x}) = \frac{\exp\left\{-\frac{1}{2}(\mathbf{x}-\mathbf{m})^T V^{-1}(\mathbf{x}-\mathbf{m})\right\}}{(2\pi)^{n/2}|V|^{1/2}} \tag{2.16}$$

where $\mathbf{m}$ is the mean vector of $f_{\mathbf{x}}(\mathbf{x})$ and V is the covariance matrix of $f_{\mathbf{x}}(\mathbf{x})$.

Assuming that the product under the integral (2.8) can be represented as multidimensional Gaussian distribution $f_{\mathbf{x}}(\mathbf{x})$(2.16) then the integral can be evaluated using *Gaussian Max/Int theorem* [26, Appendix 1, Theorem 4]. Suppose we wish to compute the message to the node x_i, i.e., integrate the distribution over all variables but the variable x_i, then:

$$f_{x_i}(x_i) = \int_{-\infty}^{+\infty} f_{\mathbf{x}}(\mathbf{x}) d(\mathbf{x}\backslash x_i) \propto \max_{\mathbf{x}\backslash x_i} \left\{ \exp\left(-\frac{1}{2}(\mathbf{x}-\mathbf{m})^T V^{-1}(\mathbf{x}-\mathbf{m}) \right) \right\}$$
$$= exp\left\{ -\frac{1}{2} \min_{\mathbf{x}\backslash x_i} \left((\mathbf{x}-\mathbf{m})^T V^{-1}(\mathbf{x}-\mathbf{m}) \right) \right\} \quad (2.17)$$

Following [26], let $W = V^{-1}$ be the inverse of the covariance matrix V and w_{ii} be the diagonal element in the row i of W. Also by $W_{\mathbf{x}\backslash x_i}$ we denote the matrix W *without* the column and row i (so $W_{\mathbf{x}\backslash x_i}$ is $n-1 \times n-1$ matrix), by W_{x_i} the row i of W without the element w_{ii} (so W_{x_i} is a vector of $n-1$ values), and by m_{x_i} the element i of mean-vector $\mathbf{m}$. Then the minimum under the exponent in (2.17) is [26, Appendix 1, Theorem 5]:

$$\min_{\mathbf{x}\backslash x_i} \left((\mathbf{x}-\mathbf{m})^T W (\mathbf{x}-\mathbf{m}) \right) = (x_i - m_{x_i})^2 (w_{ii} - W_{x_i} W^{-1}_{\mathbf{x}\backslash x_i} W_{x_i}) \quad (2.18)$$

and the value that minimizes (2.18) is:

$$\arg\min_{\mathbf{x}\backslash x_i} \left((\mathbf{x}-\mathbf{m})^T W (\mathbf{x}-\mathbf{m}) \right) = \mathbf{m}_{\mathbf{x}\backslash x_i} - W^{-1}_{\mathbf{x}\backslash x_i} W_{x_i} (x_i - m_{x_i}) \quad (2.19)$$

where $\mathbf{m}_{\mathbf{x}\setminus x_i}$ is the vector of means of the variables $\mathbf{x}$ without the element m_{x_i}. By comparing (2.18) to (2.9) one can see that the mean of the outgoing message is m_{x_i} and that the precision $P_{x_i} = w_{ii} - W_{x_i} W_{\mathbf{x}\setminus x_i}^{-1} W_{x_i}$.

Once the sum-product algorithm has converged, one can find the mean and precision of the marginal distribution over variable x_i as:

$$P_{x_i} = \sum_{f_k \in N(x_i)} P_{x_i \leftarrow f_k} \tag{2.20}$$

$$m_{x_i} = P_{x_i}^{-1} \left(\sum_{f_k \in N(x_i)} P_{x_i \leftarrow f_k} m_{x_i \leftarrow f_k} \right) \tag{2.21}$$

The task of probabilistic inference consists of determining distributions or other parameters of some "hidden" phenomena from the parameters that can be observed directly. It is obvious that, in order to infer something the latter must provide information about the former. Examples of such tasks include determining a transmitted sequence from the received sequence in a communication system or inferring a diagnosis based on observations of symptoms in a patient by a software for medical diagnostics. Given a set of observations A, set of unknowns B, and the distribution $P(A|B)$, the value of B which maximizes the likelihood of observing the configuration A is $\hat{B}_{ML} = \underset{B}{argmax} P(A|B)$ and is known as the *Maximum Likelihood* (ML) estimate of B. Using the Bayes rule $P(B|A) = P(A|B)P(B)/P(A)$ the most likely value of B given A is known as *Maximum a Posteriori Probability* (MAP) estimate of B and can be expressed as $\hat{B}_{MAP} = \underset{B}{argmax} P(A|B)P(B)$. If a

priori distribution of B is uniform then the ML estimation corresponds to the MAP estimation. In the case where the distribution $P(A|B)$ is Gaussian then the most likely values of B given A are the values of the mean-vector $\mathbf{m}$ of the distribution $P(A|B)$ so that $\hat{B}_{ML/MAP} = \mathbf{m}$. In this regard the sum-product algorithm on a factor graph where distributions of all variables are Gaussian and the updates of the function nodes preserve Gaussianity has an important property: if the algorithm converges then the means of the marginal distributions are guaranteed to be correct, even in the case where the graph has cycles [8, 26]. This implies that the MAP estimate of the Gaussian variable on a factor graph is correct.

In the case where densities are non-Gaussian and the integral in (2.8) has no closed form, we can use one of the methods below [26]:

- use quantization to convert continuous variables to discrete variables
- approximate the variables as Gaussian or as a mixture of Gaussian densities
- represent message $\mu(x)$ as a single point $\hat{x}$, which may be viewed as a temporary or final estimate of x. [10]
- use some other methods which are listed in [26] and [36]

[10] This means that in all equations such as joint probability distributions we replace the variable x with its estimate $\hat{x}$. The messages toward the node x does not need to be computed since we assumed that x is "known". Equivalently we may fix the mean of messages to $m_x = \hat{x}$ and precision $P_x = \infty$ or some large value.

2.4 Factor Graph Transformations

Factor graph transformations have been introduced in [3]. However, the concept has been known earlier, as seen in node clustering in [37]. The transformations were mainly considered the means that allows converting a graph with cycles to the cycle-free form. We reiterate that a factor graph is essentially a representation of the way how a global function factorizes as a product of local functions. It is always possible to modify this factorization and in turn, modify the factor graph. Below, we will review the modifications of the factorization that correspond to the clustering of function and variable nodes, stretching variables, and adding and removing nodes and edges in a graph. In all of these transformations, the factorization of the global function is changed but the values and the domain of the function remain unchanged.

In the discussion below we focus on the case where the variables are drawn from discrete finite alphabets and the local functions are represented in the form of tables. Let f_i and f_j be two local functions with the domains X_{S_i} and X_{S_j}. It is always possible to replace the product of f_i and f_j by $f_k\left(X_{S_i} \cup X_{S_j}\right) = f_i\left(X_{S_i}\right) f_j\left(X_{S_j}\right)$. From the factor graph perspective, this is equivalent to removing the nodes f_i and f_j from the graph and replacing them with a single node corresponding to f_k. Effectively, nodes f_i and f_j have been clustered. The domain of the node f_k is the union of the domains of the original nodes, and the degree of the node f_k is $deg(f_k) \geq max(deg(f_i), deg(f_j))$. The increase of the degree and cardinality of the domain indicates a possible significant increase of complexity of the process-

ing of node f_k compared to the original nodes. By repeating this procedure we can join any number of function nodes.

Similar to the function nodes, any number of the variable nodes can be clustered. This can be viewed as a substitution of the original variables, for example x_i and x_j, by the third variable x_k with the domain $|x_k| = |x_i| \times |x_j|$. To keep the global function unchanged we need to modify the tables of the local functions in such a way that the values of the functions remain unchanged for the values of x_k that correspond to the values of x_i and x_j. From the factor graph perspective, this transformation corresponds to removing the nodes represented by x_i and x_j from the graph and replacing them with the single node x_k. Note that the degree and the cardinality of the domain of the node x_k have increased as compared to the degrees and cardinalities of the original nodes. This may result in a higher computational cost of the update of the node. We can repeat the procedure and cluster any number of the variable nodes. Since x_i, x_j, and x_k are just labels, we may (and usually do) retain the reference to the original labels, i.e., for joined nodes we usually use joint labels such as $x_i x_j$. We emphasize however, the fact that the composite node, resulted from joining simple nodes, has the domain $|x_i| \times |x_j|$ and that there is an apparent correspondence between the values of the variable $x_i x_j$ and the values of the original variables x_i and x_j. An example of clustering transformation can be found in [3, Figure 20(b)].

Now, since we have arrived to the point where multiple variables can be assigned to a single variable node we need to update our notation. By S_i^{vn} we denote the set of indices of variables associated with a variable node i and by $X_{S_i^{vn}}$ we

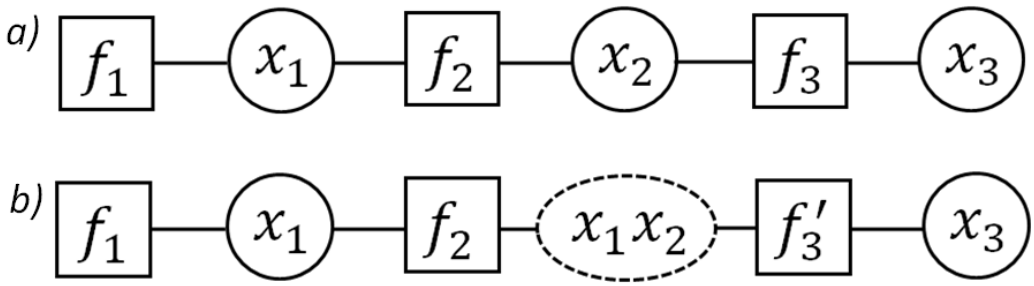
denote the set of variables with indices S_i^{vn} . We will also use the notation S_i^{fn} to differentiate between sets corresponding to function nodes.

In a factor graph, the set of neighbors $N_1(x_i)$ of a variable node x_i represents the area of the graph where the information about the dependency of the variable x_i is presented in its explicit form. In other parts of the graph, the dependency on the variable is marginalized out. It has been suggested in [3] that it is possible to extend the area where the dependency on a variable is presented in an un-marginalized form. This transformation corresponds to "stretching" of the domain of a variable.

Let the nodes corresponding to variables x_i and x_j be connected by a path of length 2. As in the clustering of variable nodes, we can expand the domain of the variable x_j by including all possible permutations of the variables x_i and x_j, i.e., we replace the node x_j with the node labeled as $x_i x_j$ and the size of the domain $|x_i| \times |x_j|$. We update the local functions which have x_j as a parameter in such a way that the tables for the values $x_i x_j$ corresponding to the values of x_j , remain unchanged. We effectively stretched variable x_i to the variable x_j. An example of the stretching transformation is presented in Figure 2.2. The factor graph in Figure 2.22a corresponds to the factorization:

$$F(x_1, x_2, x_3) = f_1(x_1) f_2(x_1, x_2) f_3(x_2, x_3) \tag{2.22}$$

Now we "stretch" the variable x_1 to the node x_2. The transformation is shown in



c)

x_2	x_3	f_3
0	0	b_1
0	1	b_2
1	0	b_3
1	1	b_4

d)

x_1	x_2	x_3	f_3'
0	0	0	b_1
0	0	1	b_2
0	1	0	b_3
0	1	1	b_4
1	0	0	b_1
1	0	1	b_2
1	1	0	b_3
1	1	1	b_4

Figure 2.2: Example of the stretching transformation:
a) The FG corresponding to the factorization $F(x_1,x_2,x_3) = f_1(x_1) f_2(x_1,x_2) f_3(x_2,x_3)$.
b) The FG with the variable x_1 "stretched" to the variable x_2.
c) The table of the function f_3 on original FG.
d) The table of the function f_3' after the transformation.

Figure 2.2b and corresponds to the global function:

$$F'(x_1,x_2,x_3) = f_1(x_1) f_2(x_1,x_2) f_3'(x_1,x_2,x_3) \tag{2.23}$$

The table of f_2 is unaffected by the transformation, while the table of the values of f_3 now includes all permutations of the variables of x_1, x_2, and x_3. Figure 2.2c and 2.2d represent the tables of the values of the local functions f_3 before and

after the transformation, respectively. We note that the values of f_3' are equal to the values of f_3 for the same assignment of the variables x_1, x_2. We also note that now, the variable x_1 is not marginalized out during the update of node f_2; instead, it will be summed out during the computation of the messages from f_3 to x_3 so that the marginals at the node x_3 remain unchanged. In general, a variable can be stretched to an arbitrary connected part of a graph.

It is possible to remove the nodes and edges from a graph. An edge can be removed if removing the edge does not change the domains of the function nodes. A variable node can be removed if all the edges connected to the node can be removed. When removing edges, we must ensure that the *running intersection property* is preserved.

Definition 9 *We say that a graph satisfies the running intersection property if for any nodes A and B with the domains X_A and X_B and any variable $x_i \in X_A \cup X_B$, a path in the graph exists such that the variable x_i is part of the domains of every node in the path. This is equivalent to the nodes with the variable x_i in their domains forming a connected sub-graph.*

The running intersection property ensures consistency of information about the dependency on a variable in different parts of a graph.

For example, consider the factor graph in Figure 2.3a. We can stretch the variable x_1 to the variables x_2 and x_3. The edge $x_1 - f_1$ can then be removed since the variable x_1 is present in the domain of the node x_1x_3 connected to the node f_1. The edge $x_1 - f_2$ can be removed as well since the variable x_1 is present

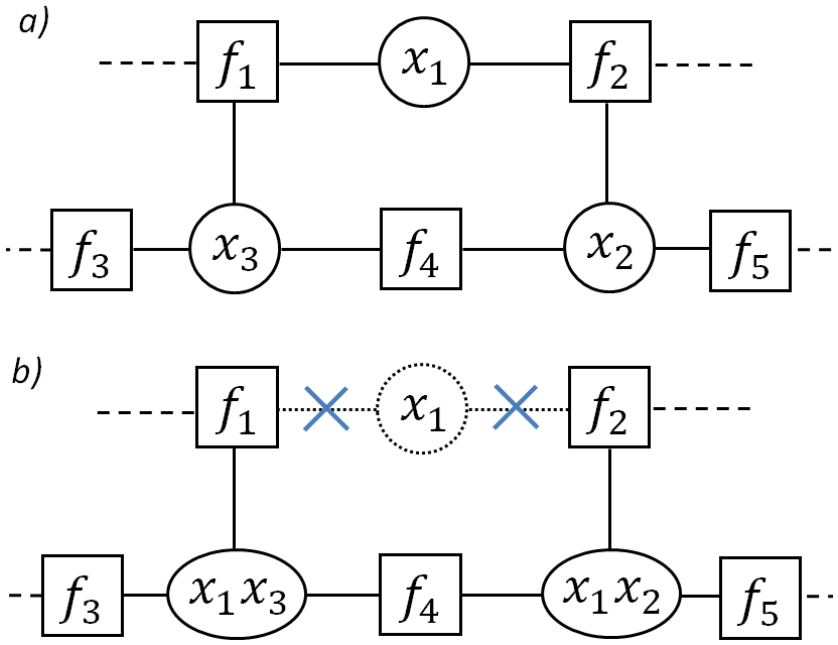


Figure 2.3: Example of removing edges and nodes: a) A FG with a cycle of length 6. b) The factor graph where the variable x_1 has been stretched to the variables x_2 and x_3. The edges $x_1 - f_1$, $x_1 - f_2$ and node x_1 can be removed from the graph.

in the domain of the node x_1x_2 connected to the node f_2. The node x_1 itself can then be removed since it has no connected edges. Removing the edges and the node in this case preserved the running intersection property since the subgraph formed by the nodes with the variable x_1 (f_1, x_1x_2, f_4, x_1x_3, and f_2) in their domains is a connected subgraph. This transformation removed the cycle $x_1 \to f_2 \to x_2 \to f_4 \to x_3 \to f_1$ from the graph.

It is possible to introduce new function nodes to a graph. We can do this by multiplying the product representing the factorization of a global function by a local function with the values equal to unity. The local function may include

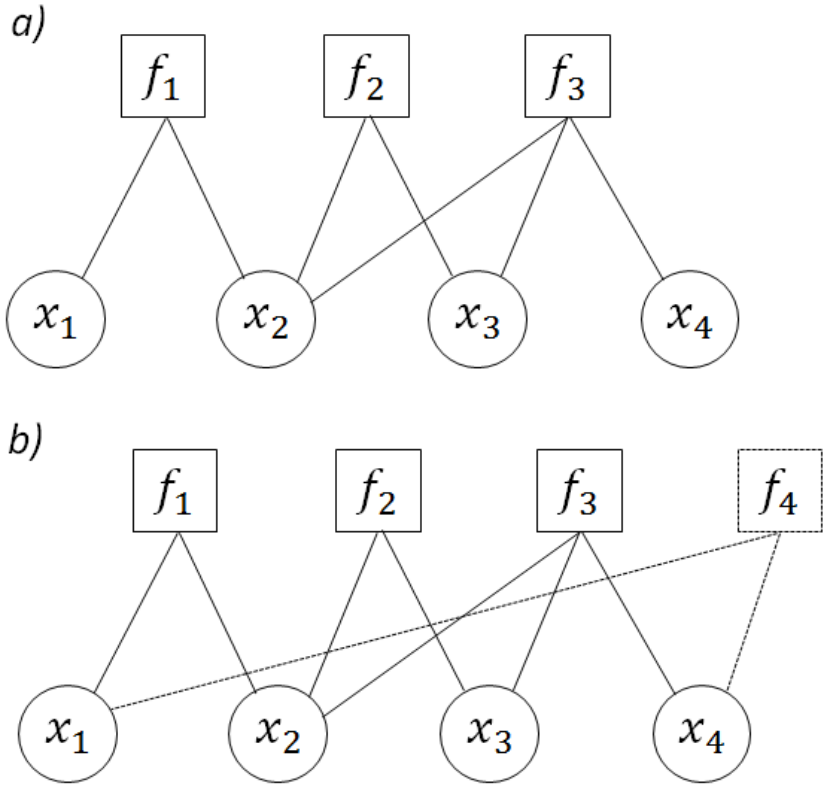


Figure 2.4: Example of introducing new nodes: a) FG corresponding to the factorization $F(x_1,x_2,x_3,x_4) = f_1(x_1,x_2)f_2(x_2,x_3)f_3(x_2,x_3,x_4)$. b) The node $f_4(x_1,x_4) = 1, \forall x_1,x_4$ has been introduced in the factor graph, this does not change the values of the global function

any of the variables from the domain of the global function. This transformation can be handy in the case where we need to find marginals over a set of variables which is not part of the domain of the existing local function or variable nodes. For example, consider the global function:

$$F(x_1,x_2,x_3,x_4) = f_1(x_1,x_2)f_2(x_2,x_3)f_3(x_2,x_3,x_4)$$

and assume that we need to find the marginal $m(x_1,x_4) = \sum_{x_2,x_3} F(x_1,x_2,x_3,x_4)$. The factor graph corresponding to this factorization is depicted in Figure 2.4a. There is no straightforward way to find $m(x_1,x_4)$ in the original graph. However, we may introduce the auxiliary local function $f_4(x_1,x_4) = 1, \forall x_1,x_4$, then the new factorization of the global function becomes (in Figure 2.4b):

$$F'(x_1,x_2,x_3,x_4) = f_1(x_1,x_2)f_2(x_2,x_3)f_3(x_2,x_3,x_4)f_4(x_1,x_4)$$

and assuming that the sum-product algorithm converged on the modified graph, $m(x_1,x_4)$ can be found in the node $f_4(x_1,x_4)$ using the expression (2.6). Note that the values of the global function in this transformation remain unchanged.

2.5 Literature on the effect of Factor Graph transformations on the complexity of the Sum-Product algorithm

The emphasis of our work is on the effects of factor graph transformations on the complexity of the sum-product algorithm. There is little existing literature which addresses this topic. The primary focus of the existing literature on the complexity of the sum-product algorithm has been the development of various efficient implementations of the updates of variable and function nodes for the specific case of decoding error corrections codes [18–21, 38, 39]. These references however, do not explore the effects of transformations on the complexity of the sum-product algorithm.

The main reference for our work is the renowned paper of Kschischang et al. [3]. In this publication, the authors observed that the cardinality of the domain of a composite variable node created by the clustering of variable nodes is the multiplication of the cardinalities of the domains of the original nodes. This, according to the authors "can imply a substantial cost increase in computational complexity of the sum-product algorithm". The local domain of a composite function node created by clustering function nodes f_i and f_j with the domains X_{S_i} and X_{S_j} is $X_{S_i} \cup X_{S_j}$ which "can imply a substantial cost increase in computational complexity of the sum-product algorithm; however, clustering functions do not increase the complexity of the variables."

The authors of [3] also noted that "it is always possible to transform a factor graph with cycles into a cycle-free factor graph, but at the expense of increasing the complexity of the local functions and/or the domains of the variables."

In the next chapter, we will present examples of factor graphs where the transformations lower the complexity of the sum-product algorithm and investigate in detail the issue of the effect of transformations on the complexity of the sum-product algorithm. We also show that eliminating cycles from a graph sometimes leads to the lowering of the complexity of the sum-product algorithm.

Chapter 3

A method of lowering the complexity of the sum-product algorithm using graph transformations

3.1 Definitions

In this chapter we discuss the complexity of the sum-product algorithm in relation to the factor graph transformations described in Section 2.4. We will focus on the discrete case with the sum-product update rules defined by (2.2) and (2.3). We define complexity C as the number of ordinary or semiring additions and multiplications required to find the marginals of a global function using the sum-product algorithm. We consider the all-vertices problem, i.e., the marginals have to be computed at each variable node, and assume that the flooding schedule is used so

that during iterations the outgoing messages are updated on all edges connected to a node. The dependency of the complexity on the convergence speed or on the number of iterations required for convergence of the algorithm is not considered. In other words, we are mostly concerned with the "complexity per iteration." Let A denote the number of additions, and M denote the number of multiplications. For a factor graph with the set of variable nodes VN and the set of function nodes FN, we define the complexity of the sum-product algorithm as:

$$C_G = \sum_{\forall i \in VN} M_i^{vn} + \sum_{\forall j \in FN} \left(A_j^{fn} + M_j^{fn} + C_j^{lf} \right) \tag{3.1}$$

where M_i^{vn} is the number of multiplications required for the update of a variable node i, A_j^{fn} and M_j^{fn} are the number of additions and multiplications required for the update of a function node j, and C_j^{lf} is the complexity of the evaluation of a local function j. From this point onwards, we assume that the local functions are presented in the form of tables so $C_j^{lf} = 0$. The expression (3.1) represents the number of additions and multiplications necessary to compute messages on each edge in a graph. In the case where the flooding schedule is used, this number C_G is equivalent to the number of operations required by a single iteration of the sum-product algorithm.[1]

The parameters that define the complexity of the update of a node are the node's degree and the cardinalities of the variables involved in the node's domain.

[1] In (3.1) we actually did not include the operation of computing the marginals at the variable nodes vn_i upon completion of iterations. This however, as we are going to show below, requires much fewer operations compared to the updates of the nodes.

By q_i we denote the number of values taken by the variable x_i. By Q_i^{vn} and Q_j^{fn} we denote the cardinalities of domains of the variable node i and function node j, respectively. On a factor graph in the "original" form there is a one-to-one correspondence between the variables and variable nodes so that $Q_i^{vn} = q_i$. However, this may not be the case after we apply factor graph transformations and there can be more than a single variable associated with a variable node. In this case, the cardinality of the domain of a variable node is:

$$Q_i^{vn} = \prod_{\forall k \in S_i^{vn}} q_k \tag{3.2}$$

where, following the notation introduced in Section 2.4, by S_i^{vn} we denote the set of indices of the variables associated with a variable node i. For a function node j the cardinality of the domain is:

$$Q_j^{fn} = \prod_{\forall k \in S_j^{fn}} q_k \tag{3.3}$$

where by S_j^{fn} we denote the set of indices of the variables associated with a function i.

In this Chapter we consider the most general form of the sum-product algorithm where the messages on the edges connected to a variable node i have Q_i^{vn} values. In other words we do not take into account the simplifications which may arise in some particular applications. Some of such simplifications are considered in Chapter 4.

There are other important metrics of efficiency of an algorithm such as memory requirements, ability to implement the computations in parallel, and others. In this thesis we focus primarily on the number of operations as a measure of the complexity. We briefly discuss the memory requirements of the sum-product algorithm in Sections 3.3.3 and throughout Chapter 4 .

3.2 Examples of factor graphs where the transformations lower the complexity of the sum-product algorithm

In this section we present several examples of the factor graphs where the transformations introduced in Section 2.4 lower the number of operations required by the sum-product algorithm. These examples serve as motivation for our work. To the best of our knowledge, the observation that factor graph transformations may lower the complexity of the sum-product algorithm has not appeared in published literature before.

Intuitively, the complexity of the sum-product algorithm expressed as the number of arithmetic operations necessary to update every node in a graph under the flooding schedule depends on the number of nodes in a graph, degrees and the size of the domains of the nodes. We will review the complexity of the update of variable and function nodes in detail in Section 3.3. Meanwhile, we assume that the complexity is a strictly increasing function of the degree and cardinality of the domain of a node. In [3] Kschischang et al. observed that the factor graph transformations increase the cardinality of the domains of transformed nodes. We

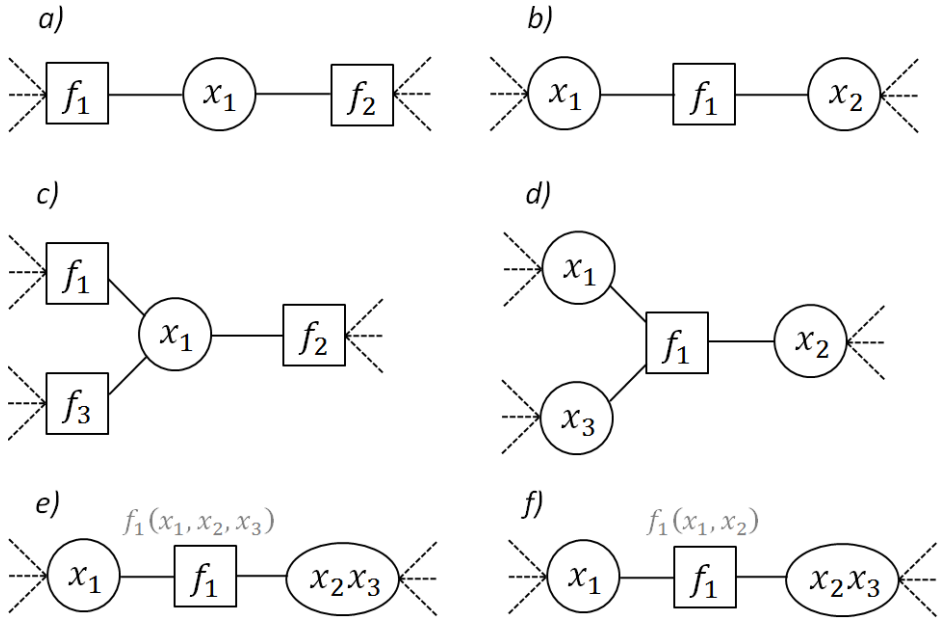


Figure 3.1: Example of simple nodes of degree 2 and 3: a) VN degree 2. b) FN degree 2. c) VN degree 3. d) FN degree 3. e) FN degree 2 with 3 variables f) FN degree 2 with 3 variables and a local function which depends on two variables

note that the transformations (such as clustering) may also decrease the number of nodes in a graph and the degrees of the nodes. In other words, there are two opposite effects: one is the increase of the complexity due to the increase of cardinalities and in some cases degrees of the nodes, and the other effect is the lowering of the complexity due to the decrease of the number of the nodes in a graph and in some cases degrees of the nodes. Whether a particular transformation increases or decreases the total number of operations depends on the topology of the graph and the cardinalities of the domains.

We begin our examples by determining the number of operations necessary to update simple nodes of degree 2 and 3 in Figure 3.1. For the sake of simplicity

we assume that all variables are binary. In a general case of binary variables the messages sent on edges of a factor graph have two values. Following the notation introduced earlier, by $\mu_{x_i \to f_j}$ and $\mu_{f_j \to x_i}$ we denote the messages sent by a variable node x_i to a function node f_j and by a function node f_j to a variable node x_i, respectively. In order to compute the outgoing messages of the variable and function nodes we use the expressions (2.2) and (2.3), respectively. In all cases we assume that the flooding schedule is used so that the messages have to be updated on all edges. The number of operations necessary to update the nodes in Figure 3.1 is the following:

- A variable node of degree 2 in Figure 3.1a. The messages from the variable node x_1 to the function nodes f_1 and f_2 (outgoing messages) are:

$$\mu_{x_1 \to f_1}(x_1) = \mu_{f_2 \to x_1}(x_1)$$
$$\mu_{x_1 \to f_2}(x_1) = \mu_{f_1 \to x_1}(x_1)$$

  In other words, the outgoing messages are simply the values of incoming messages and therefore, a variable node of degree 2 does not require any operations.

- A function node of degree 2 which is presented in Figure 3.1b. The message from the function node f_1 to the variable nodes x_1 for the values of $x_1 = 0$

and $x_1 = 1$ is:

$$\begin{aligned}
\mu_{f_1 \to x_1}(x_1 = 0) &= f_1(x_1 = 0, x_2 = 0) \cdot \mu_{x_2 \to f_1}(x_2 = 0) \\
&\quad + f_1(x_1 = 0, x_2 = 1) \cdot \mu_{x_2 \to f_1}(x_2 = 1) \\
\mu_{f_1 \to x_1}(x_1 = 1) &= f_1(x_1 = 1, x_2 = 0) \cdot \mu_{x_2 \to f_1}(x_2 = 0) \\
&\quad + f_1(x_1 = 1, x_2 = 1) \cdot \mu_{x_2 \to f_1}(x_2 = 1)
\end{aligned}$$

The computation of the message $\mu_{f_1 \to x_1}(x_1)$ requires 4 multiplications and 2 additions. Likewise, the computation of the message $\mu_{f_1 \to x_2}(x_2)$ requires 4 multiplications and 2 additions. The total number of arithmetic operations necessary to update the function node therefore is 8 multiplications and 4 additions.

- A variable node of degree 3 in Figure 3.1c. The message from the variable node x_1 to the function node f_1 (outgoing messages) is computed as:

$$\begin{aligned}
\mu_{x_1 \to f_1}(x_1 = 0) &= \mu_{f_2 \to x_1}(x_1 = 0) \cdot \mu_{f_3 \to x_1}(x_1 = 0) \\
\mu_{x_1 \to f_1}(x_1 = 1) &= \mu_{f_2 \to x_1}(x_1 = 1) \cdot \mu_{f_3 \to x_1}(x_1 = 1)
\end{aligned}$$

Hence, 2 multiplications are required to update the messages on each of the edges and in the case of binary variables, 6 multiplications are necessary to update the node. For the non-binary case when the cardinality of the domain of a variable x_1 is q (x_1 takes q values), the messages sent to and from the node will have q values and the total number of multiplications necessary

to update the node is $3q$.

- A function node of degree 3 in Figure 3.1d. The message from the function node f_1 to the variable nodes x_1 for the values of $x_1 = 0$ and $x_1 = 1$ is:

$$\begin{aligned}\mu_{f_1\to x_1}(x_1=0) = {} & f_1(x_1=0,x_2=0,x_3=0)\cdot\mu_{x_2\to f_1}(x_2=0)\cdot\mu_{x_3\to f_1}(x_3=0)\\ & + f_1(x_1=0,x_2=1,x_3=0)\cdot\mu_{x_2\to f_1}(x_2=1)\cdot\mu_{x_3\to f_1}(x_3=0)\\ & + f_1(x_1=0,x_2=0,x_3=1)\cdot\mu_{x_2\to f_1}(x_2=0)\cdot\mu_{x_3\to f_1}(x_3=1)\\ & + f_1(x_1=0,x_2=1,x_3=1)\cdot\mu_{x_2\to f_1}(x_2=1)\cdot\mu_{x_3\to f_1}(x_3=1)\end{aligned}$$

$$\begin{aligned}\mu_{f_1\to x_1}(x_1=1) = {} & f_1(x_1=1,x_2=0,x_3=0)\cdot\mu_{x_2\to f_1}(x_2=0)\cdot\mu_{x_3\to f_1}(x_3=0)\\ & + f_1(x_1=1,x_2=1,x_3=0)\cdot\mu_{x_2\to f_1}(x_2=1)\cdot\mu_{x_3\to f_1}(x_3=0)\\ & + f_1(x_1=1,x_2=0,x_3=1)\cdot\mu_{x_2\to f_1}(x_2=0)\cdot\mu_{x_3\to f_1}(x_3=1)\\ & + f_1(x_1=1,x_2=1,x_3=1)\cdot\mu_{x_2\to f_1}(x_2=1)\cdot\mu_{x_3\to f_1}(x_3=1)\end{aligned}$$

  The products of the incoming messages $\mu_{x_2\to f_1}(x_2)\cdot\mu_{x_3\to f_1}(x_3)$ are the same for both $\mu_{f_1\to x_1}(x_1=0)$ and $\mu_{f_1\to x_1}(x_1=1)$. The products can be computed once, saved and used for computing both $\mu_{f_1\to x_1}(x_1=0)$ and $\mu_{f_1\to x_1}(x_1=1)$. Therefore, the computation of the message on an edge can be done with 12 multiplications and 6 additions. In total, 36 multiplications and 18 additions are necessary to update the node.

- A function node of degree 2 with 3 binary variables in Figure 3.1e. Such a

node may be formed by clustering of variable nodes connected to a function node of degree 3. In this case, the messages sent from f_1 to x_2x_3 and from x_2x_3 to f_1 have 4 values. The message from the node f_1 to the node x_1 can be computed using 8 multiplications and 6 additions:

$$\begin{aligned}\mu_{f_1\to x_1}(x_1=0) = {} & f_1(x_1=0,x_2=0,x_3=0)\cdot\mu_{x_2x_3\to f_1}(x_2=0,x_3=0)\\ & + f_1(x_1=0,x_2=1,x_3=0)\cdot\mu_{x_2x_3\to f_1}(x_2=1,x_3=0)\\ & + f_1(x_1=0,x_2=0,x_3=1)\cdot\mu_{x_2x_3\to f_1}(x_2=0,x_3=1)\\ & + f_1(x_1=0,x_2=1,x_3=1)\cdot\mu_{x_2x_3\to f_1}(x_2=1,x_3=1)\end{aligned}$$

$$\begin{aligned}\mu_{f_1\to x_1}(x_1=1) = {} & f_1(x_1=1,x_2=0,x_3=0)\cdot\mu_{x_2x_3\to f_1}(x_2=0,x_3=0)\\ & + f_1(x_1=1,x_2=1,x_3=0)\cdot\mu_{x_2x_3\to f_1}(x_2=1,x_3=0)\\ & + f_1(x_1=1,x_2=0,x_3=1)\cdot\mu_{x_2x_3\to f_1}(x_2=0,x_3=1)\\ & + f_1(x_1=1,x_2=1,x_3=1)\cdot\mu_{x_2x_3\to f_1}(x_2=1,x_3=1)\end{aligned}$$

The update of the message from f_1 to x_2x_3 requires 8 multiplications and 4 additions:

$$\begin{aligned}\mu_{f_1\to x_2x_3}(x_2=0,x_3=0) = {} & f_1(x_1=0,x_2=0,x_3=0)\cdot\mu_{x_1\to f_1}(x_1=0)\\ & + f_1(x_1=1,x_2=0,x_3=0)\cdot\mu_{x_1\to f_1}(x_1=1)\\ \mu_{f_1\to x_2x_3}(x_2=1,x_3=0) = {} & f_1(x_1=0,x_2=1,x_3=0)\cdot\mu_{x_1\to f_1}(x_1=0)\\ & + f_1(x_1=1,x_2=1,x_3=0)\cdot\mu_{x_1\to f_1}(x_1=1)\end{aligned}$$

$$\mu_{f_1\to x_2x_3}(x_2=0,x_3=1) = f_1(x_1=0,x_2=0,x_3=1)\cdot\mu_{x_1\to f_1}(x_1=0)$$
$$+ f_1(x_1=1,x_2=0,x_3=1)\cdot\mu_{x_1\to f_1}(x_1=1)$$
$$\mu_{f_1\to x_2x_3}(x_2=1,x_3=1) = f_1(x_1=0,x_2=1,x_3=1)\cdot\mu_{x_1\to f_1}(x_1=0)$$
$$+ f_1(x_1=1,x_2=1,x_3=1)\cdot\mu_{x_1\to f_1}(x_1=1)$$

In total, number 16 multiplications and 10 additions are required to update the node.

- A function node of degree 2 with 3 binary variables in Figure 3.1f. In this case, the function node f_1 is connected to the node x_2x_3 but *the local function* $f_1(x_1,x_2)$ *does not depend on* x_3. As we will see in the examples below, this configuration appears after the stretching transformation or after clustering of variable nodes.[2] The fact that the local function does not depend on one of the variables allows us to save some of the operations compared

[2]Formally, to stay within the formalism of the factor graphs in the transformations we expand the domain of the local function and replace $f_1(x_1,x_2)$ with $f_1'(x_1,x_2,x_3)$, however it is clear that $f_1'(x_1,x_2,x_3) = f_1(x_1,x_2), \forall x_3$.

to the case in Figure 3.1e. The message from the node f_1 to the node x_1 is:

$$\begin{aligned}
\mu_{f_1\to x_1}(x_1=0) &= f_1'(x_1=0,x_2=0,x_3=0)\cdot\mu_{x_2x_3\to f_1}(x_2=0,x_3=0)\\
&\quad + f_1'(x_1=0,x_2=1,x_3=0)\cdot\mu_{x_2x_3\to f_1}(x_2=1,x_3=0)\\
&\quad + f_1'(x_1=0,x_2=0,x_3=1)\cdot\mu_{x_2x_3\to f_1}(x_2=0,x_3=1)\\
&\quad + f_1'(x_1=0,x_2=1,x_3=1)\cdot\mu_{x_2x_3\to f_1}(x_2=1,x_3=1)\\
&= f_1(x_1=0,x_2=0)\left(\mu_{x_2x_3\to f_1}(x_2=0,x_3=0)+\mu_{x_2x_3\to f_1}(x_2=0,x_3=1)\right)\\
&+ f_1(x_1=0,x_2=1)\left(\mu_{x_2x_3\to f_1}(x_2=1,x_3=0)+\mu_{x_2x_3\to f_1}(x_2=1,x_3=1)\right)
\end{aligned}$$

Similarly,

$$\begin{aligned}
&\mu_{f_1\to x_1}(x_1=1) =\\
&= f_1(x_1=1,x_2=0)\left(\mu_{x_2x_3\to f_1}(x_2=0,x_3=0)+\mu_{x_2x_3\to f_1}(x_2=0,x_3=1)\right)\\
&+ f_1(x_1=1,x_2=1)\left(\mu_{x_2x_3\to f_1}(x_2=1,x_3=0)+\mu_{x_2x_3\to f_1}(x_2=1,x_3=1)\right)
\end{aligned}$$

Therefore, the message can be computed with 4 multiplications and 4 additions. In the case of the message from f_1 to x_2x_3 only the values $\mu_{f_1\to x_2x_3}(x_2=0,x_3=0)$ and $\mu_{f_1\to x_2x_3}(x_2=1,x_3=0)$ have to be computed since $\mu_{f_1\to x_2x_3}(x_2=0,x_3=1)=\mu_{f_1\to x_2x_3}(x_2=0,x_3=0)$ and $\mu_{f_1\to x_2x_3}(x_2=1,x_3=$

$1) = \mu_{f_1 \to x_2 x_3}(x_2 = 1, x_3 = 0)$. The message is:

$$\mu_{f_1 \to x_2 x_3}(x_2 = 0, x_3 = 0) = f_1'(x_1 = 0, x_2 = 0, x_3 = 0) \cdot \mu_{x_1 \to f_1}(x_1 = 0)$$
$$+ f_1'(x_1 = 1, x_2 = 0, x_3 = 0) \cdot \mu_{x_1 \to f_1}(x_1 = 1)$$
$$\mu_{f_1 \to x_2 x_3}(x_2 = 1, x_3 = 0) = f_1'(x_1 = 0, x_2 = 1, x_3 = 0) \cdot \mu_{x_1 \to f_1}(x_1 = 0)$$
$$+ f_1'(x_1 = 1, x_2 = 1, x_3 = 0) \cdot \mu_{x_1 \to f_1}(x_1 = 1)$$
$$\mu_{f_1 \to x_2 x_3}(x_2 = 0, x_3 = 1) = f_1'(x_1 = 0, x_2 = 0, x_3 = 1) \cdot \mu_{x_1 \to f_1}(x_1 = 0)$$
$$+ f_1'(x_1 = 1, x_2 = 0, x_3 = 1) \cdot \mu_{x_1 \to f_1}(x_1 = 1) = \mu_{f_1 \to x_2 x_3}(x_2 = 0, x_3 = 0)$$
$$\mu_{f_1 \to x_2 x_3}(x_2 = 1, x_3 = 1) = f_1'(x_1 = 0, x_2 = 1, x_3 = 1) \cdot \mu_{x_1 \to f_1}(x_1 = 0)$$
$$+ f_1'(x_1 = 1, x_2 = 1, x_3 = 1) \cdot \mu_{x_1 \to f_1}(x_1 = 1) = \mu_{f_1 \to x_2 x_3}(x_2 = 1, x_3 = 0)$$

The update of such a node in total requires 8 multiplications and 6 additions. It is just 2 more additions compared to the update of a function node of degree 2 with 2 binary variables. These 2 extra additions "spend" on marginalizing out the variable x_3 from the message $\mu_{x_2 x_3 \to f_1}$.

An important observation from the example 3.1e is that in the case where a local function does not depend on one (or some) of the variables, the variable can be marginalized out prior to participation in any local computations. [3] We will use this observation throughout this thesis.

The summary of the number of operations necessary to update the nodes of degree 2 and 3 in the case binary variables is presented in Table 3.1

[3] An additional condition for a variable to be marginalized out is that the variable appears on a single edge only.

Node	Number of operations			Comments
	Multiplications	Additions	Total	
VN degree 2	0	0	0	
FN degree 2	8	4	12	
VN degree 3	6	0	6	$3q$ for a node with cardinality q
FN degree 3	36	18	54	
FN degree 2 with 3 variables	10	16	26	
FN degree 2 with 3 variables	8	6	14	The local function does not depend on one of the variables

Table 3.1: The summary of the number of operations required to update the variable (VN) and function (FN) nodes of degree 2 and 3.

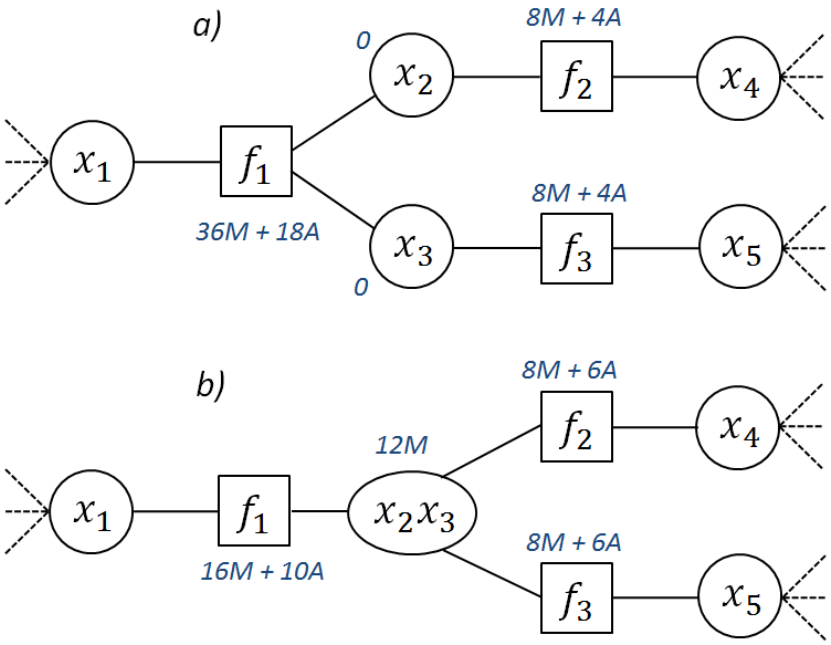


Figure 3.2: An example of a factor graph where clustering of variable nodes lowers the complexity of the sum product algorithm. The number of operations for each node is shown as $\mathbf{X}M + \mathbf{Y}A$, where X is the number of additions and Y is the number of multiplications.

Now we will present three examples of transformations and show that they lead to lowering of the complexity of the sum-product algorithm.

Example #1: The complexity of the update of the simple graph in Figure 3.2 can be lowered by clustering the variable nodes x_2 and x_3. The number of operations necessary to update each of the nodes is shown in Figure 3.2a in the form $\mathbf{X}M + \mathbf{Y}A$ where X and Y are the number of multiplications and additions, respectively. Assuming that all variables are binary and taking into account the number of operations in Table 3.1, one can see that 52 multiplications and 26 additions are required to update the sub-graph $\{f_1, x_2, x_3, f_2, f_3\}$ in Figure 3.2a. By clustering the nodes x_2 and x_3 as shown in Figure 3.2b we can lower the operation count to 44 multiplications and 22 additions, i.e., save 12 operations. Note that the nodes, except for the nodes $\{f_1, x_2, x_3, f_2, f_3\}$, are not affected by the transformation.

On the original graph the nodes x_1 and x_2 do not need any operations, while on the transformed graph the composite node x_1x_2 requires 12 multiplications. Therefore, as it has been indicated in [3], the clustering increased the number of operations necessary to update the nodes x_1 and x_2. However, the degrees of the node f_1 decreased from 3 to 2 and this lead to the lowering of *the total* number of operations necessary to update this part of the graph. As we shall see later, the higher the degree and cardinality of the domain of the node f_1, the higher the gain (number of saved operations) achieved by such a transformation.

Example #2: Consider the factor graph in Figure 3.3a. Here we are interested in the complexity of the update of the sub-graph $\{x_1, f_2, f_3, x_2\}$. The sub-graph is a cycle of length 4. Applying the number of operations from the Table 3.1

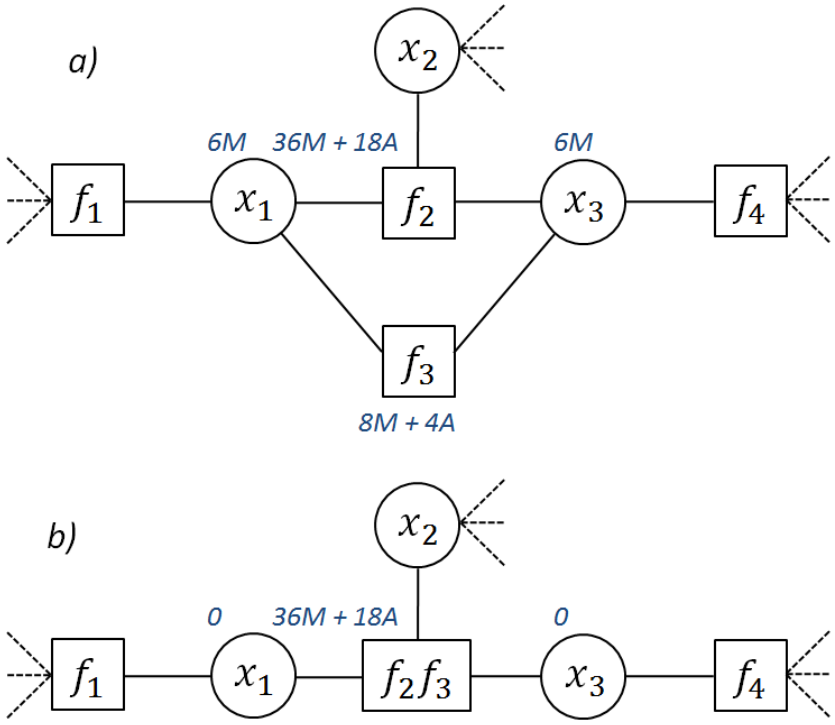


Figure 3.3: An example of a factor graph where the clustering of function nodes lowers the complexity of the sum product algorithm.

one can see that the update of the sub-graph requires 56 multiplications and 22 additions. In this graph clustering the nodes f_2 and f_3 as shown in Figure 3.3b lowers the operations count. Computing the 8-valued table $f_2(x_1,x_2,x_3)\cdot f_3(x,x_3)$ can be done with 8 multiplications. In the cases where the computation of the table $f_2\cdot f_3$ can be done "off-line"[4] the total number of operations necessary to update the sub-graph $\{x_1, f_2f_3, x_2\}$ in 3.3b is 38 and 18 additions. In other words, by clustering the nodes we can save 12 multiplications and 4 additions.[5] The

[4]By "offline" we mean that it is not done at the time of execution of the sum-product algorithm.

[5]Even in the case where we need to compute $f_2\cdot f_3$ at the time of execution we save 4 multiplications and 4 additions.

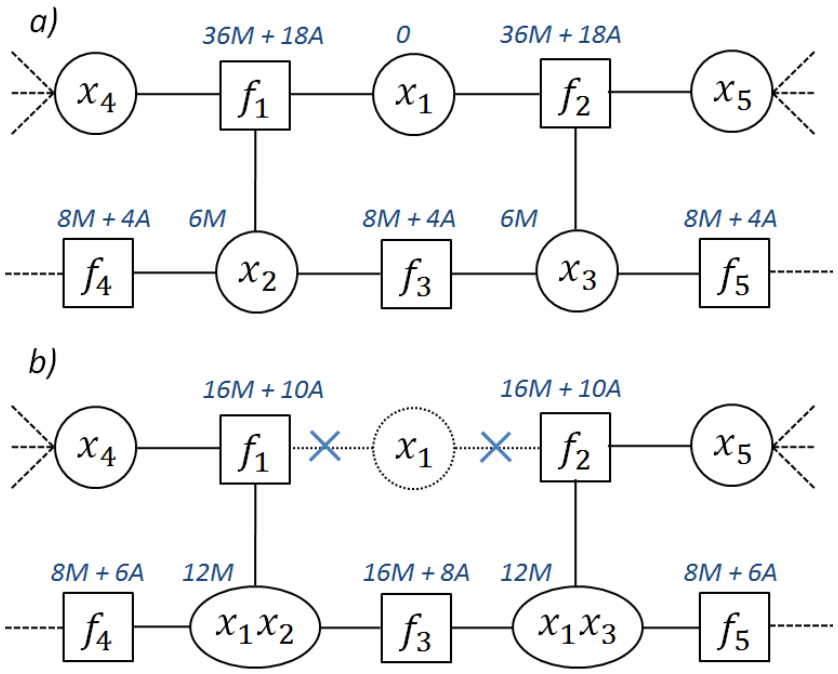


Figure 3.4: An example of a factor graph where stretching transformation lowers the complexity of the sum product algorithm.

decrease in the operation count comes from the lowering of the degrees of the nodes x_1 and x_2 and from the fact that the node f_3 has been eliminated from the graph. Also note that the cycle was removed from the graph and the number of operations was decreased at the same time.

Example #3: The factor graph in Figure 3.4 is an example of a graph where the stretching transformation and removing edges lowers the required number of operations. The graph has a cycle of length 6 and we are interested in the complexity of the update of the sub-graph $\{f_1,\dots f_5,x_1\dots,x_3\}$. Again, taking into account the number of operations in Table 3.1, one may determine that the update of the sub-graph in Figure 3.4a requires 100 multiplications and 48 additions. The

cycle can be eliminated from the graph using the stretching transformation.[6] At first, we "stretch" the variable x_1 along the path $f_1 \to x_2 \to f_3 \to x_3$. The transformation makes the edges $x_1 - f_1$ and $x_1 - f_2$ redundant since the nodes f_1 and f_2 are connected to the nodes x_1x_2 and x_1x_3 which now have the variable x_1 in their domains. Finally, the node x_1 can be removed since after removing the edges the node has degree 0. It can be seen from the number of operations in Figure 3.4b that such transformations not only eliminated the cycle from the graph but also lowered the number of operations necessary to update the sub-graph to 80 multiplications and 40 additions.

The examples above show that the number of operations can be lowered by transforming a factor graph. In order to consider the cases of nodes of degrees higher than 3, we need to determine the complexity of the update of the nodes of arbitrary degrees.

3.3 The complexity of nodes updates

3.3.1 The complexity of the update of a variable node

We remind the reader that the outgoing message of a variable node vn_i[7] is expressed as:

$$\mu_{vn_i \to fn_j}(X_{S_i^{vn}}) = \prod_{fn_k \in N(vn_i) \setminus fn_j} \mu_{fn_k \to vn_i}(X_{S_i^{vn}}) \tag{3.4}$$

[6] This transformation was described in Section 2.4.

[7] Here we denote a variable node by vn_i and function node by fn_j.

In order to evaluate the message, we need to compute $d(vn_i)$[8] products of $d(vn_i) - 1$ incoming messages - the products of all but a single incoming message. The messages have to be computed for each of Q_i^{vn} values of the domain of the node vn_i. Computing the products can be done in several ways. The direct approach, where the messages are computed independently, requires $d(vn_i)(d(vn_i) - 2)Q_i^{vn}$ multiplications.[9] We can also compute the product of all the messages, then divide the product by a single message for each edge. This would require $(d(vn_i) - 1)Q_i^{vn}$ multiplications and $d(vn_i)Q_i^{vn}$ divisions. Divisions however, do not need to be part of the semiring and may not be allowed. Another consideration is that, the division is computationally expensive [40] compared to additions and multiplications. For a node of a large degree we may disregard the influence of a single message and approximate it with the product of all the messages. This approach requires only $(d(vn_i) - 1)Q_i^{vn}$ multiplications but may introduce a large error in the case of small degree nodes or in the case where the values of one or more messages are much lower or higher compared to the values of the rest of the messages.

Below, we consider a simple and efficient way of computing the product originally proposed by Aji et al. [2]. Assuming that a variable node vn_i has n edges and received n incoming messages $\mu_1, \mu_2, \ldots, \mu_n$. At first we compute $n-1$ partial products $A_1, A_2, \ldots, A_{n-1}$ and $n-1$ partial products $B_n, B_{n-1}, \ldots, B_2$. The term A_i

[8] $d(vn_i)$ is degree of the node vn_i.

[9] Note that $n-1$ multiplications are required in order to compute the product of n values.

is the product of incoming messages $\mu_1 \cdots \mu_i$:

$$\begin{aligned} A_1 &= \mu_1 \\ A_2 &= A_1 \cdot \mu_2 \\ A_3 &= A_2 \cdot \mu_3 \\ &\vdots \\ A_{n-1} &= A_{n-2} \cdot \mu_{n-1} \end{aligned} \tag{3.5}$$

The term B_i is the product of the messages $\mu_i \cdot \cdots \cdot \mu_n$

$$\begin{aligned} B_n &= \mu_n, \\ B_{n-1} &= B_n \cdot \mu_{n-1} \\ B_{n-2} &= B_{n-1} \cdot \mu_{n-2} \\ &\vdots \\ B_2 &= B_3 \cdot \mu_2 \end{aligned} \tag{3.6}$$

The computation of the partial products A and B requires $2(n-2)$ multiplications. Now the outgoing message on edge i can be simply computed as $\mu_i^{Out} = A_{i-1} \cdot B_{i+1}$. Taking into account that $\mu_1^{Out} = B_1$ and $\mu_n^{Out} = A_{n-1}$ the total number of multiplications necessary to compute all n products of $n-1$ incoming messages is $3(n-2)$. The message has to be computed for each of Q_i^{vn} values of the domain of the node vn_i. Therefore, the total number of multiplications necessary to update

the node in this way is:

$$M_i^{vn} = 3 \cdot Q_i^{vn} \cdot (d(vn_i) - 2) \tag{3.7}$$

Upon convergence of the algorithm utilizing the terms A_{n-1}, we can compute the final result which is the marginal of the global function over variable $X_{S_i}^{vn}$ expression (2.4) with additional Q_i^{vn} multiplications:

$$m(X_{S_i^{vn}}) = A_{n-1}(X_{S_i^{vn}}) \cdot \mu_n(X_{S_i^{vn}})$$

Now we compare the effectiveness of this approach compared to the "straightforward approach" where the messages on each of the edges are computed independently. For example, in the case of a node of degree 10, the message on each edge is the product of 9 values and requires 8 multiplications. In total, the "straightforward approach" requires 80 multiplications for each of Q_i^{vn} values. Using the approach described above the number of multiplications is only 24.

The method described above, can also be applied in the case of the sum-product algorithm generalized on an arbitrary commutative semiring (see Section 2.2). For example, in the "min-sum" semiring the number of operations in expression (3.7) represents the number of additions.

3.3.2 The complexity of the update of a function node

We remind the reader that the outgoing message of a function node fn_i is expressed as:

$$\mu_{fn_j \to vn_i}(X_{S_i^{vn}}) = \sum_{X_{S_j^{fn}} \setminus X_{S_i^{vn}}} F_j(X_{S_j^{fn}}) \prod_{vn_k \in N(fn_j) \setminus vn_i} \mu_{vn_k \to fn_j}(X_{S_k^{vn}}) \qquad (3.8)$$

where $F_j(X_{S_j})$ in general can be a product of several local functions. We can evaluate the expression (3.8) in the way similar to the computation of a variable node, i.e., at first compute the partial products of the incoming messages $A_i = \mu_1 \cdot \mu_2 \cdots \mu_i$ and $B_j = \mu_n \cdot \mu_{n-1} \cdots \mu_j$ and then compute the outgoing message for an edge k as $A_{k-1} \cdot B_{k+1}$. Compared to the computation of the messages in a variable node the messages in the product in RHS of expression (3.8) are parameterized by different sets of variables $X_{S_i^{vn}}$ and this can be explored to efficiently compute the partial products A and B.

It is convenient to explain the proposed method of the message computation with an example. Consider the function node of degree 5 in Figure 3.5a. For the sake of simplicity we assume that the variables are binary. We can then compute the term A using the computational tree in Figure 3.5b. The term $A_1(x_1) = \mu_1(x_1)$[10] has 2 values corresponding to the incoming messages $\mu_1(x_1 = 0)$ and $\mu(x_1 = 1)$. The term $A_2(x_1, x_2)$ has 4 values corresponding to all of the permuta-

[10]Hereforth, we add the domain of variables to the notation of the product A and B

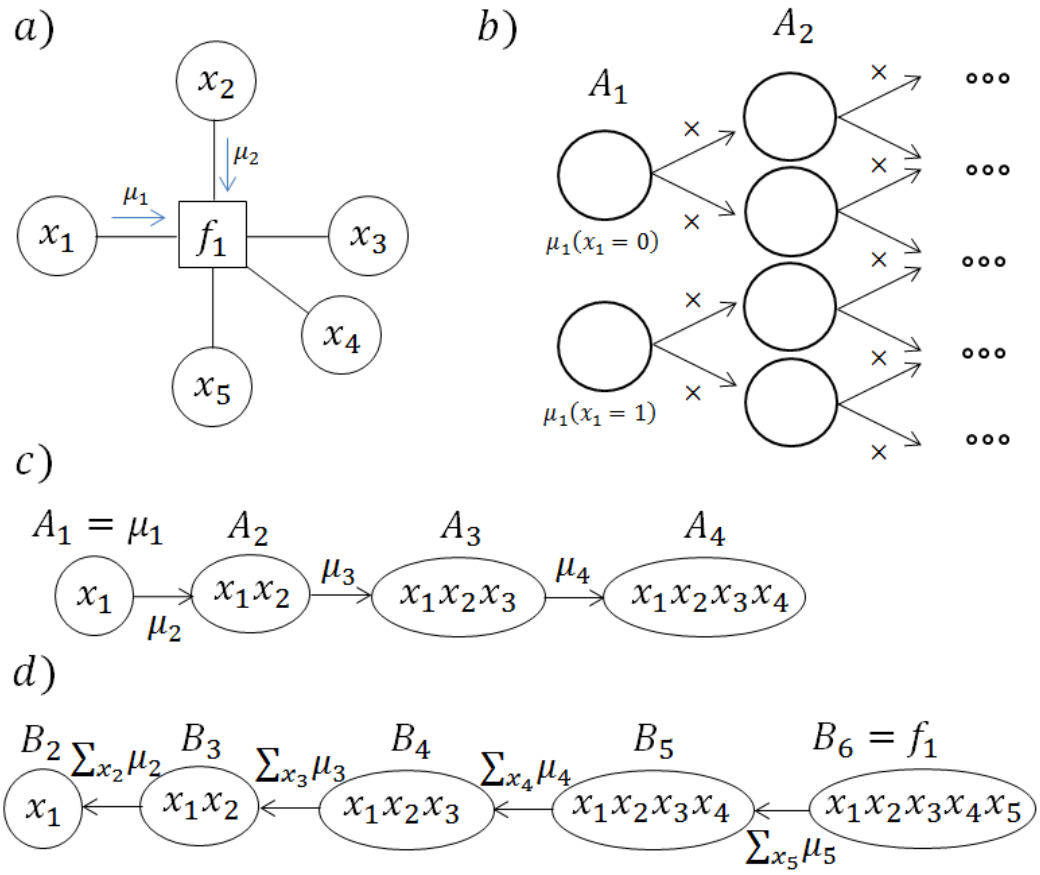


Figure 3.5: The explanations of the update of a function node of degree 5.

tions of the variables x_1 and x_2 and can be computed with 4 multiplications:

$$A_2(x_1 = 0, x_2 = 0) = A_1(x_1 = 0)\mu_2(x_2 = 0)$$

$$A_2(x_1 = 0, x_2 = 1) = A_1(x_1 = 0)\mu_2(x_2 = 1)$$

$$A_2(x_1 = 1, x_2 = 0) = A_1(x_1 = 1)\mu_2(x_2 = 0)$$

$$A_2(x_1 = 1, x_2 = 1) = A_1(x_1 = 1)\mu_2(x_2 = 1)$$

Similarly, $A_3(x_1,x_2,x_3)$ and $A_4(x_1,x_2,x_3,x_4)$ takes 8 and 16 values and requires 8 and 16 multiplications. The computation of the terms A is schematically presented in Figure 3.5c.

We incorporate the multiplication by the local function f_1 in the term B. So let $B_6 = f_1(x_1,x_2,x_3,x_4,x_5)$. The outgoing message to the variable x_5 is $\mu_5^{Out}(x_5) = \sum_{\{x_1,x_2,x_3,x_4\}} A_4(x_1,x_2,x_3,x_4) \cdot B_6 = f_1(x_1,x_2,x_3,x_4,x_5)$. The product under the summation has 32 values and requires 32 multiplications. The message $\mu_5^{Out}(x_5)$ has two values each of which is the summation of 16 values. Therefore, the marginalization requires $2(16-1) = 30$ additions. Now we notice that the variable x_5 is not needed for the computation of the messages to the nodes $x_4,\ldots,x_1$ and it can be marginalized out from the terms $B_5 \ldots B_2$. Taking this observation into account the term B_5 can then be computed as $B_5(x_1,x_2,x_3,x_4) = \sum_{x_5} \mu_5(x_5) B_6 = f_1(x_1,x_2,x_3,x_4,x_5)$. The term B_5 which takes 16 values, requires 36 multiplications and 16 additions. As we proceed and compute B_4, B_3 and B_2 using the same logic we marginalize out the variables x_4, x_3 and x_2. The process of computation of the terms B is schematically presented in Figure 3.5d.

The outgoing messages are computed as follows:

$$\begin{aligned}
\mu_1^{Out}(x_1) &= B_2(x_1) \\
\mu_2^{Out}(x_2) &= \sum_{x_1} A_1(x_1) B_3(x_1,x_2) \\
\mu_3^{Out}(x_3) &= \sum_{x_1,x_2} A_2(x_1,x_2) B_4(x_1,x_2,x_3)
\end{aligned}$$

Term	$A_1(x_1)$	$A_2(x_1,x_2)$	$A_3(x_1,x_2,x_3)$	$A_4(x_1,x_2,x_3,x_4)$	NA	Total
Size	2	4	8	16	NA	NA
Multipl.	0	4	8	16	NA	28
Term	$B_2(x_1)$	$B_3(x_1,x_2)$	$B_4(x_1,x_2,x_3)$	$B_5(x_1,x_2,x_3,x_4)$	$B_6 = f_1$	Total
Size	2	4	8	16	32	NA
Multipl.	4	8	16	32	0	60
Additions	2	4	8	16	0	30
Term	$\mu_1^{Out}(x_1)$	$\mu_2^{Out}(x_2)$	$mu_3^{Out}(x_3)$	$\mu_4^{Out}(x_4)$	$\mu_5(x_5)$	Total
Size	2	2	2	2	2	NA
Multipl.	0	4	8	16	32	60
Additions	0	2	6	14	30	52
Totals	Multiplications: 148 Additions:80 Total operations:228					

Table 3.2: The number of operations required to compute A, B and μ^{Out}

$$\mu_4^{Out}(x_4) = \sum_{x_1,x_2,x_3} A_3(x_1,x_2,x_3)B_5(x_1,x_2,x_3,x_4)$$
$$\mu_5^{Out}(x_5) = \sum_{x_1,x_2,x_3,x_4} A_4(x_1,x_2,x_3,x_4)B_6(x_1,x_2,x_3,x_4,x_5)$$

The tables of the number of operations necessary to compute the terms A, B and the outgoing message are presented in Figure 3.5.

The reader may wonder whether the method of updating a function node, as described above, is efficient. We may evaluate the computational advantage of the method with an example of the update of a function node of degree 10. Assuming that the domains of all 10 variable nodes include a single binary variable (so that the cardinality of the domain of the local function is 1024) then updating the node in the "straightforward way" requires:

1. Computing 1024 products of 9 values (all but 1 incoming messages) for each of 10 edges. This can be done with 24576 multiplications.

2. Marginalizing the 1024 products in order to compute two values of the outgoing message for each of 10 edges. This would require 10220 additions.

In Section 3.4 we will determine that the to update a node of degree 10 using the approach described above requires 5108 multiplications and 3048 additions. Hence, compared to the straightforward approach the method described above provides 75% reduction in the operations counts.

In a general, the domains of variables may have different cardinalities. In this case in order to achieve the lowest operation count it is preferable to add the variables with higher cardinalities to the computations as late as possible and marginalize them out as early as possible. This can be easily done by arranging the messages in order of increasing cardinalities of the domains. For example, if the incoming messages $\mu_1(x_1)$, $\mu_2(x_2)$ and $\mu_3(x_3)$ and the variables x_1, x_2 and x_3 have cardinalities 4,8 and 2, respectively then in order to achieve a lower count of operations we need to compute the partial terms as:

$$\begin{aligned} A_1(x_3) &= \mu_3(x_3) \\ A_2(x_1,x_3) &= A_1(x_3)\mu_1(x_1) \\ B_3(x_1,x_3) &= \sum_{x_2} f(x_1,x_2x_3)\mu_2(x_2) \\ B_2(x_3) &= \sum_{x_1} B_3(x_1,x_3)\mu_1(x_1) \end{aligned}$$

We also reiterate the observation of Section 3.2: if a variable appears on a single edge connected to a function node and the local function does not depend on the variable, then the variable can be marginalized out from the domain of the message

prior to performing any computations at the function node. So if the message is $\mu_1(x_1,x_2)$ the local function f_i does not depend on the variable x_2, then x_2 can be marginalized out from $\mu_1(x_1,x_2)$ and in the following computations the message can be treated as if it depends on the variable x_1 only.

Now we are going to derive the expression for the number of operations necessary to update a function node. Assuming we have the number of non-overlapping domains with cardinalities $q_1,q_2,\ldots,q_n$, then the computation of the partial products $A_1,A_2,\ldots,A_{n-1}$ would require $q_1q_2+q_1q_2q_3+\cdots+q_1\cdots q_{n-1}$ multiplications which can be expressed as:

$$M_A=\sum_{i=2}^{n-1}\prod_{j=1}^{i}q_j \tag{3.9}$$

Similarly, computation of the terms B requires:

$$M_B=\sum_{i=2}^{n}\prod_{j=1}^{i}q_j \tag{3.10}$$

The terms B also requires:

$$A_B=\sum_{i=2}^{n}(q_i-1)\prod_{j=1}^{i-1}q_j \tag{3.11}$$

additions. The logic behind the formula above is that in order to evaluate the term B_i for $i\in\{2,\ldots,n\}$ we need to compute $q_1\cdots q_{i-1}$ sums of q_i number which require $(q_1\cdots q_{i-1})(q_i-1)$ additions. Computing the outgoing messages (multi-

plications of the terms $A_{i-1}B_{i+1}$) requires:

$$M_\mu = \sum_{i=2}^{n} \prod_{j=1}^{i} q_j \tag{3.12}$$

multiplications. Finally, the marginalization of extra variables from the term $A_{i-1}B_{i+1}$ requires computing of q_i sums of $q_1 \cdot q_2 \cdots q_{i-1}$ numbers which requires:

$$A_\mu \sum_{i=2}^{n-1} q_i \left(\prod_{j=1}^{i-1} q_j - 1 \right) \tag{3.13}$$

additions. The total number of multiplications necessary to update a node therefore, can be expressed as:

$$M^{fn} = 3 \cdot \sum_{i=2}^{n} \prod_{j=1}^{i} q_j - \prod_{j=1}^{n} q_j \tag{3.14}$$

where the last term is due to the fact that in the expression (3.9) the summation is up to $n-1$ while in (3.10) and (3.12) the summation is up to n. The number of additions nessesary to update a function node is:

$$A^{fn} = \sum_{i=2}^{n} (q_i - 1) \prod_{j=1}^{i-1} q_j + \sum_{i=2}^{n-1} q_i \left(\prod_{j=1}^{i-1} q_j - 1 \right) \tag{3.15}$$

The methods that we have used to evaluate the operation counts holds when the sum-product algorithm is generalized on an arbitrary semiring (see Section 2.2). In this case the number of multiplications in (3.14) reflects the number of semiring multiplications and the number of additions in (3.15) corresponds to the

number of semiring additions. For example, in the case of "min-sum" semiring the number in (3.15) is the number of operations "min" and the number in (3.14) is the number of additions.

3.3.3 Memory Requirements

Now we are going to discuss the memory requirements of the sum-product algorithm. The algorithm requires storing the messages on the edges and the values of local functions. We define U as memory unit that represents the number of bits necessary to store a value of a message on an edge or a value of a local function. The exact value of U depends on the accuracy and performance requirements of a particular implementation and could be in the range of 2-32 bits. Some temporary variables are also necessary for processing at the nodes, e.g., for computing the terms A and B as discussed above. These temporary variables could be shared between the nodes and therefore, take an insignificant amount of memory compared to the amount of memory required to store the messages. The amount of memory S required by the sum-product algorithm which operates on a factor graph with the set of variable nodes VN and the set of function nodes FN, can be expressed as:

$$S = \underbrace{\sum_{i \in VN} d(vn_i) \cdot Q_i^{vn} \cdot U}_{\text{memory to store the messages}} + \underbrace{\sum_{j \in FN} Q_j^{fn} \cdot U}_{\text{memory to store the values of local functions}} \tag{3.16}$$

3.4 An example of complexity optimization

Up to this point we have defined the number of operations (additions and multiplications) necessary to update a node of an arbitrary degree. For a variable node the number of multiplications is expressed by (3.7) and for a function node the number of multiplications and additions is expressed by (3.14) and (3.15), respectively. Now we can consider a general factor graph with aim of finding the realization which leads to the lowest complexity of the sum-product algorithm.

Consider an example of a node of degree 10 presented in Figure 3.6a. In this example all variable nodes adjacent to f_1 have degree 2. In other words, each of the variable nodes $x_1, x_2, \ldots x_{10}$ is connected to f_1 and one of the other 10 function nodes $f_2, f_3, \ldots f_{11}$ (note that only the nodes f_2 and f_3 are shown in Figure 3.6a). We may consider various configurations of clustering the variable nodes. For example, in the pairwise clustering of the nodes the node x_1 is clustered with the node x_2, the node x_3 with the node x_4 and so on. After this transformation the function node f_1 will have degree 5 while 5 clustered variable nodes will have degree 3 as shown in Figure 3.6b. We will also consider other ways of clustering the nodes, such as 3 clusters with 3, 3 and 4 variable nodes as well as 2 clusters of 5 variable nodes. Number of possible configurations of clustering is large but intuitively we wish that clustered nodes will have the same or similar degrees.

We update the nodes as described in Section 3.3.2. Tables 3.3 and 3.4 represent the number of operations necessary to compute the partial products A and B as well as the outgoing messages in the case of the original graph in Figure 3.6a and

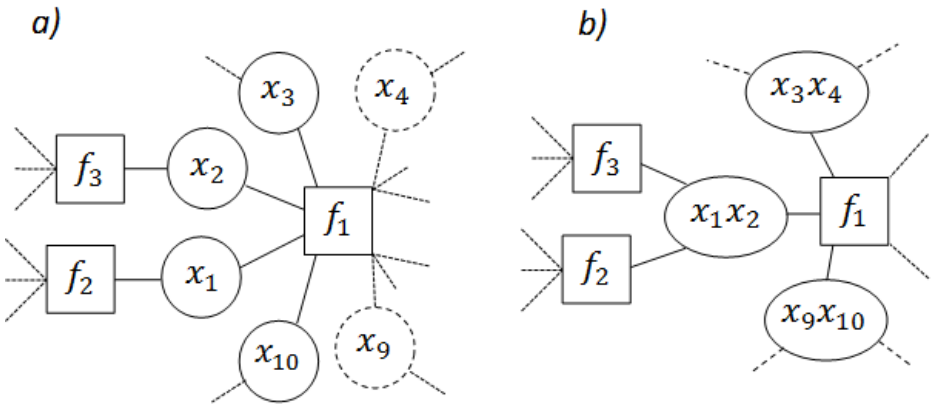


Figure 3.6: a) A function node of degree 10 with 10 adjacent variable nodes of degree 2. b) Pairwise clustering of the variable nodes.

in the case of the graph with clustered nodes in Figure 3.6b. All variables are assumed to be binary. As one can see from the number of operations presented in the tables, the pairwise clustering of the nodes lowered the total number of operations required to update the node f_1 from 8156 to 5420.

After the transformation in Figure 3.6b we need to take into account the complexity of the update of the variable nodes and the complexity introduced to the function nodes, *except* for the node f_1. Each of the variable nodes x_1x_2, x_3x_4 and so on now has degree 3 and cardinality of the domain 4. Therefore, from (3.7), 12 multiplications are necessary to update a single node and 60 multiplications are necessary to update all 5 clustered nodes. After the clustering, the message sent to the nodes $f_2, f_3, \ldots, f_{11}$ has 2 variables in their domain. The local functions however, do not depend on one of the variables in the domains of the messages. This case has been discussed in Section 3.2. In such a case, the variables which are not the part of the local function can be marginalized out prior to perform-

Table 3.3: The number of operations required to update a node of degree 10 in Figure 3.6a

Note: All messages have cardinality of 2.

Terms A

Term A	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}	Total
Size A	2	4	8	16	32	64	128	256	512	NA	NA
Multiplications	0	4	8	16	32	64	128	256	512	NA	1020

Terms B

Term B	B_2	B_3	B_4	B_5	B_6	B_7	B_8	B_9	B_{10}	$B_{11} = f_1$	Total
Size B	2	4	8	16	32	64	128	256	512	1024	NA
Multiplications.	4	8	16	32	64	128	256	512	1024	NA	2044
Additions	2	4	8	16	32	64	128	256	512	NA	1022

Outgoing messages

Messages	μ_1	μ_2	μ_3	μ_4	μ_5	μ_6	μ_7	μ_8	μ_9	μ_{10}	Total
Multiplications.	0	4	8	16	32	64	128	256	512	1024	2044
Additions	0	2	6	14	30	62	126	254	510	1022	2026

Total multiplications: 5108 Total additions: 3048 Total operations: 8156

ing operations at the function nodes. Taking this into account each of the nodes $f_2, f_3, \ldots, f_{11}$ has to perform 2 extra additions which results in 20 additional additions for all 10 function nodes. In total, the number of operations necessary to update this part of the graph has been lowered from 8156 to 5500.

Using a similar approach, we computed the number of operations for other clustering configurations. The total number of operations for various cases of clustering the nodes is presented in Table 3.5. Note that among the explored

Table 3.4: The number of operations required to update a node of degree 5 with the variable nodes clustered "in pairs" in Figure 3.6b

Note: All messages have cardinality of 4

Terms A

Term A	A_1	A_2	A_3	A_4	A_5	Total
Size A	4	16	64	256	NA	NA
Multiplications	4	16	64	256	NA	336

Terms B

Term B	B_2	B_3	B_4	B_5	$B_6 = f_1$	Total
Size B	4	16	64	256	1024	NA
Multiplications	16	64	256	1024	NA	1360
Additions	12	48	192	768	NA	1020

Outgoing messages

Messages	μ_1	μ_2	μ_3	μ_4	μ_5	Total
Multiplications	0	16	64	256	1024	1360
Additions	0	12	60	252	1020	1344

Total multiplications: 3056 Total additions: 2364 Total operations: 5420

combinations, clustering the nodes in the combination 3+3+4 (2 nodes with 3 variables and 1 node with 4 variables) results in the lowest operation count. By clustering the nodes in this way we are able to lower the operation count from 8156 to 4652 or by 43%.

Now assume that the variable nodes adjacent to the node f_1 have degree 10. In other words, each of the variable nodes is connected to 10 nodes and there are

Table 3.5: The number of operations required to update a node of degree 10 in Figure 3.6a under various clustering configurations of the variable nodes. The variables are binary.

Configuration	Number of variable nodes	Degrees of the variable nodes	Operations				
			Update of the node f_1		Update of the variable nodes	Complexity added to the other function nodes	Total
			Multiplications	Additions	Multiplications	Additions	
Original configuration	10	2	5108	3048	0	0	8156
VN clustered in "pairs"	5	3	3056	2364	60	20	5500
3 clusters with 3,3 and 4 variables	3	4,4, and 5	2240	2080	240	92	4652
2 clusters of 5 nodes	2	6	2048	1984	768	300	5100
1 cluster of 10 variables	1	11	0	0	27648	10220	37868

90 function nodes besides the nodes f_1. The number of operations for this case is presented in Table 3.6. This time the minimum number of operations is reached in the configuration with the nodes clustered in pairs. The lowest count is 6620 which is a 23% reduction. Note that we can lower the operation count even in the case where the adjacent variable nodes have higher degrees.

These examples of local optimization looks interesting but the reader perhaps wonders how to find the global optimal solution, i.e., the graph which guarantees to provide the minimal complexity of the sum-product algorithm. In this thesis, we do not offer a solution to this non-trivial problem. In fact, we suspect that this problem is NP-complete. The greedy algorithm that attempts the optimization for a graph as a whole is presented in Chapter 5.

3.5 Summary

In this chapter we showed that the number of operations required by the sum-product algorithm can be lowered by transforming a factor graph. While the transformations increase the number of operations of transformed nodes, they may lower the complexity of the nodes which are adjacent to the transformed nodes. In the next chapter we will discuss several practical applications of the method of lowering the complexity.

Table 3.6: The number of operations required to update a node of degree 10 that has 10 adjacent variable nodes of degree 10. The variables are binary.

Configuration	Number of the variable nodes	Degrees of the variable nodes	Operations				
			Update of the node f_1		Update of the variable nodes	Complexity added to the other function nodes	Total
			Multiplications	Additions	Multiplications	Additions	
Original configuration	10	10	5108	3048	480	0	8636
VN clustered in "pairs"	5	19	3056	2364	1020	180	6620
3 clusters with 3,3 and 4 variables	3	28,28, and 37	2240	2080	2928	828	8076
2 clusters of 5 nodes	2	46	2048	1984	8448	2700	15180
1 cluster of 10 variables	1	91	0	0	911360	91980	1003340

Chapter 4

Examples of practical applications

In Chapter 3 we demonstrated that the complexity of the sum-product algorithm can be lowered by transforming a factor graph. The objective of this chapter is to show that the transformations can be used to lower the complexity of the sum-product algorithm in practical applications. As examples, we consider applications of the sum-product algorithm in Joint DNA Base-Calling [1], Wireless Link Loss Monitoring in Sensor Networks [22] and decoding of the Hamming (7,4) code [41]. Since our main focus is on the lowering of the complexity, our reviews of the original publications are brief and in no way comprehensive; we refer interested readers to the original publications for a more comprehensive treatment of the subjects.

4.1 Joint Base-Calling of Two DNA Sequences

In this section we apply factor graph transformations to lower the complexity of the sum product algorithm in the model of joint DNA Base-Calling [1]. We give only a brief introduction to the background and the method from the original paper. We have chosen this publication as our example because of its practical significance and because of the structure of the factor graph presented in the paper. The factor graph has many short cycles which suggest the possibility of lowering the operation counts using transformations of the graph.

DNA is the carrier of genetic information in all known biological life forms except for viruses. A DNA molecule consists of two long polymers, each formed from units called nucleotides. Each nucleotide is made up, in part, of one of four bases: Adenine (A), Cytosine (C), Guanine (G), and Thymine (T). Genetic information in DNA is "encoded" in a sequence of these bases in a DNA molecule. The length of a genome sequence is millions of nucleotides. Two DNA molecules form a paired "double-helix" structure, i.e., bases from one molecule form a connection with the bases from another molecule. The chemistry of DNA allows only Adenine-Thymine and Cytosine-Guanine base pairs. The double structure of DNA allows replications of DNA during cell divisions. During replication, the double-helix splits into two single DNA molecules each of which is then complemented to the "full" double molecule with complementary bases. Determining the sequence of the bases from a given DNA sample is a matter of paramount importance for many disciplines of biological science, medicine and others.

The most common method for DNA sequencing dates back to the work of F. Sanger [42]. The technique utilized in [1] is referred to as "whole-genome shotgun sequencing using dye-termination based electrophoresis." The process starts with a random division of DNA sample into fragments, with a subsequent independent reading of each fragment. The division is required since the maximum length of the sequence that can be reliably read using the method does not exceed thousands of bases. The process of division and sequencing of a single DNA sample is repeated several times, i.e., several passes are made and in each pass the molecule is randomly divided into fragments. At the final stage, the DNA is "reassembled" from the sequenced fragments based on the overlap between the fragments from different "passes". It requires more than 30 minutes to sequence a fragment of 600-800 bases long and 8-12 passes in order for the final reassembly of DNA to occur. Considering the fact that DNA consists of millions of bases, this process is time consuming and expensive.

The sequencing of each fragment is performed as follows. The fragment is replicated and during the replication process, multiple shorter fragments are formed according to the sequence of bases in the original fragment. The growth of each replicated fragment is terminated after a special molecule that contains fluorescent dye, called dye terminator, attaches to the end of the fragment. There are four types of dye terminators and each type is fluorescent with its own color of light. Each type can be attached to only one of four bases (A, C, G or T). The process is random and as a result, there are many fragments of different lengths in the solution. Each fragment is a shorter copy original fragment. Fragment of the

same length are terminated by the same type of dye depending on the type of the base at the "tail" of the original fragment.

During separation, stage fragments travel in a medium under the influence of an electric field. Since the charge of the DNA depends on the length of the fragment, the length determines the speed of the fragment in a medium. Fragments of the same length arrive to the detector at approximately the same time. At the detector, florescent dyes are excited with a laser and then the color of the dyes is read. As a result, a sequence of pulses of four colors of light is generated at the detector, with each sequence of pulses corresponding to a base at the end of the sequence of a certain length. The light of each color is sampled with a speed of approximately 12 samples per pulse period (in the case of the equipment used by the authors).

During the data processing stage, the captured data is de-noised and other distortions such as slowly varying period and light intensity are corrected. After the data processing, the bases are detected. The maximum length of the sequence that can be successfully detected comes from the fact that with longer sequences, the resolution of the method decreases and the light pulses overlap more and more until they become undistinguishable. An example of a DNA trace at the beginning and end of the sequence is presented in Figure 4.1a.

The novelty of the approach of the paper [1] is that the authors proposed sequencing two DNA fragments in a single run. The approach could potentially increase the efficiency by not only doubling the speed but also, by reducing at the same time, the cost of DNA sequencing. The method works by mixing two

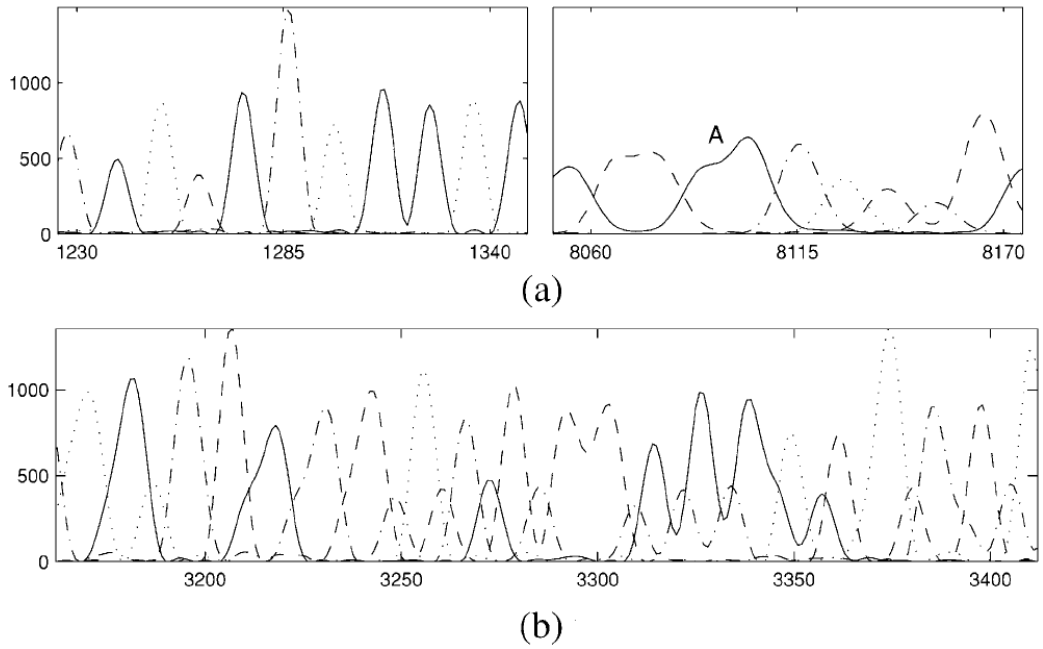


Figure 4.1: Example of a DNA trace [1] in the case of a single (a) and joint sequencing of two samples (b). Lines of different types present light of four colors corresponding to four DNA bases

DNA samples with different concentrations in the replication stage. As a result, during the detection phase, an overlap of the two sequences of pulses is created, with each sequence corresponding to its own DNA sample. The sequences can be separated based on the difference in amplitude and the time of arrival (TOA) of the pulses. The amplitude of the pulses from each of the sequences depends on the concentrations of DNA material from the two samples. According to the authors, the practically achievable difference between the amplitudes is approximately 2. Noise floor and saturation prevent any further possible increase of the difference in amplitude. Figure 4.1b represents an example of a DNA trace in the case of Joint DNA Base Calling. The sequence with larger amplitude is refereed as major

and the other sequence is refereed as minor.

A received sampled sequence can be described with the time series:

$$y[t] = \sum_{i=1}^{N_1} \alpha_{1i} p_i (t - \tau_{1i}) \underline{x}_{1i} + \sum_{j=1}^{N_2} \alpha_{2j} p_j \left(t - \tau_{2j}\right) \underline{x}_{2j} + \underline{e}(t) \tag{4.1}$$

where we adopted the notation of [1] with a vector of variables denoted by underlining. The variables in the equation are defined as follows.

- $y[t]$ is the discrete time sample of the vector of intensity of four light colors for $1 < t < K$ with K corresponding to the total number of samples.[1] The authors in [1] use equipment that samples the signal at approximately 12 samples per pulse period. The length of the sequence therefore is $K = 12 \cdot (max(N_1, N_2) + 1)$.

- N_1 and N_2 are the lengths of two DNA fragments being identified.

- α_{1i} and α_{2j} is amplitude of pulse i and j from sequences 1 and 2, respectively.

- p_i and p_j are shapes of pulses i and j of sequences 1 and 2, respectively.

- τ_{1i} and τ_{2j} are the times of arrival of pulses i and j of sequences 1 and 2, respectively.

[1] There are 4 colors of light which can be viewed as "orthogonal dimensions" of the signal. In this regard, a pulse in a sequence can appear in a single dimension only. At a certain time t in a certain color there can be 4 cases: 1) no pulse in either sequence 2) a pulse of the major sequence 3) a pulse of the minor sequence 4) pulses in both sequences. At each sampling point t the vector $\underline{y}[t]$ is the vector of 4 values corresponding to the intensities of light of 4 colors that are equivalent to 4 bases.

- $\underline{x}_{1i}$ and $\underline{x}_{2j}$ are the vectors of the four symbols $\{1000, 0100, 0010, 0001\}$ corresponding to the DNA bases.

- $\underline{e}(t)$ - is the vector of additive noise for each color.

The objective of the sequencing is to estimate the sequence of the symbols $\underline{x}_{1i}$ and $\underline{x}_{2j}$. The statistics of the parameters are:

- α_{1i} and α_{2j} are [1] "independent and identically distributed (i.i.d.) with Gamma distribution, where the right tail is larger". The definition of Gamma distribution can be found, for example, in [34]. The average amplitude of the prevalent sequence is approximately 750 units and its spread is approximately 200, while the average amplitude of the minor sequence is half the amplitude of the major sequence. There is significant overlap between the areas of the pdfs of the sequences. Therefore, the pulses cannot be identified from their amplitudes only. For example, a pulse with amplitude 500 units may belong to either of the sequences.

- Pulse shapes p_i and p_j are estimated from the data. The width of a pulse is increasing with the number of the pulse in the sequence, as can be seen in Figure 4.1a by comparing the trace on the left, which represents the beginning of a sequence, to the trace on the right, which corresponds to the end of the sequence.

- Pulses "times of arrival" τ_{1i} and τ_{2j} can be described by the first order

Markov chain:

$$f(\tau_{l,i+1}|\tau_{l,i}) = f_{\Delta\tau}(\tau_{l,i+1} - \tau_{l,i}) \qquad l \in \{1,2\} \tag{4.2}$$

where the mean of $f_{\Delta\tau}$ corresponds to slowly varying peak spacing (approximately known) and variance of approximately 0.8 samples. The authors do not specify the distribution of $f_{\Delta\tau}$.

In general, there is no alignment between the pulses from sequence 1 and sequence 2. The pulses within a sequence however, are spaced approximately 12 samples apart. The authors in [1] note that the most difficult case in terms of separation is when the pulses from two sequences overlap and the amplitudes of the sequences are close.

The factor graph corresponding to the model is depicted in Figure 4.2. The variable nodes in the top and bottom rows correspond to the parameters of the sequences 1 and 2, respectively. y_i represents a set of samples associated with a peak i. $f(y_i|*)$ is the probabilities of observed samples y_i given a configuration of the variables α_{1i}, α_{2i}, τ_{1i}, τ_{2j}, $\underline{x}_{1i}$ and $\underline{x}_{2i}$ as well as the variables α, τ and $\underline{x}$ from the pulses $i-1$ and $i+1$. Cross-edges between $f(y_i|*)$ nodes and the variables from the neighboring pulses ($i-1$ and $i+1$) reflect the interference between the pulses. The MAP estimate of the base type $\underline{x}_{1i}$ and $\underline{x}_{2j}$ in sequence 1 and 2 can be inferred from the graph using the sum-product algorithm. Assuming

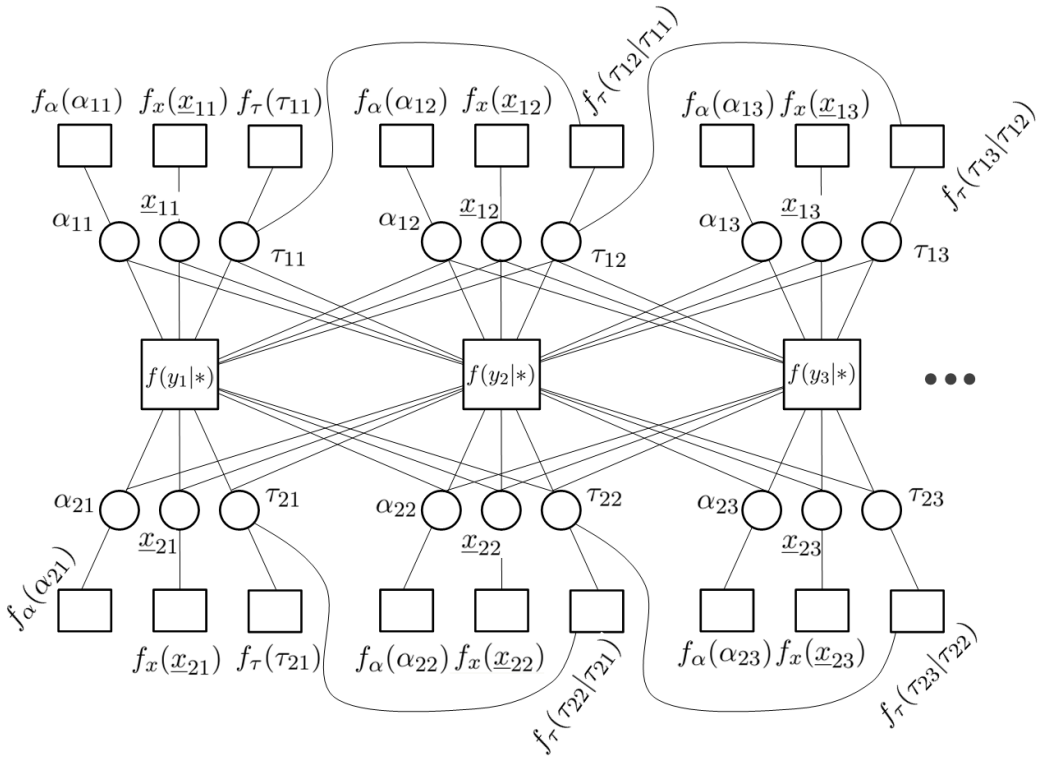


Figure 4.2: Factor graph corresponding to the model of Joint DNA Base-Calling. Function nodes f_α, f_x, and f_τ are a priori probability of the variables α, x and τ , respectively. * - represent the set of variables of the variable nodes connected to a function node. $f(y_i|*)$ - is the probability of sample y_i given the configurations of the variables.

$\underline{\theta} = \{\underline{x}_1, \underline{\alpha}_1, \underline{\tau}_1, \underline{x}_2, \underline{\alpha}_2, \underline{\tau}_2\}$ MAP estimate of the parameters is:

$$\underline{\hat{\theta}} = \arg\max_{\underline{\theta}} f(\underline{\theta}|y) = \arg\max_{\underline{\theta}} \left(\log f(y|\underline{\theta}) + \log(\underline{\theta})\right) \tag{4.3}$$

where we are interested in finding the estimates of $\underline{x}_1$ and $\underline{x}_2$. A priori distributions

of the parameters is:

$$\log(\underline{\theta}) = \sum_{l=1}^{2}\sum_{i=1}^{N_l}\{\log f_\alpha(\alpha_{li}) + \log f_{x_l}(x_{li})\} + \log f_\tau(\underline{\tau}_1, \underline{\tau}_1) \tag{4.4}$$

Assuming zero mean Gaussian additive noise with variance σ^2 which can be estimated, the log likelihood of observed data is:

$$\log f(\underline{y}|\underline{\theta}) = \frac{1}{2\sigma^2}\sum_t \left(y(t) - \sum_{i=1}^{N_1} \alpha_{1i}\hat{p}_{1i}(t-\tau_{1i})\underline{x}_{1i} - \sum_{j=1}^{N_2} \alpha_{2j}\hat{p}_{2j}(t-\tau_{2j})\underline{x}_{2i} \right)^2 + c \tag{4.5}$$

where c is a constant that does not affect maximization.

The sum-product algorithm can be implemented on the graph in the following way[2]

- First of all, the distribution f_α is continuous and in order to apply the sum-product algorithm in the discrete form we need to approximate the distribution with a discrete PMF. The number of values of the discrete PMF $\hat{f}_\alpha$ determines the cardinalities of the domains of the variables α. [3]

- Partition the sets of samples into possibly overlapping sets of points y_i for $i = 1, \ldots N$, associated with pulses of sequences 1 and 2, where N is the estimated number of pulses (perhaps $MAX(N_1, N_2)$). This assumes an approximate estimation of the locations (times of arrival) of pulses $\hat{\tau}_{1i}$ and $\hat{\tau}_{2i}$.

[2]The authors of [1] noted that the implementation of the sum-product algorithm on the graph in Figure 4.2 is prohibitively complex and did not describe the implementation of the algorithm. Here we describe our understanding of implementation of the algorithm.

[3]The effect of such approximations and required value q_α has to be studied but this is outside the scope of this work.

We assume that such an estimation can be done since the period of pulses in the sequence is approximately known. The period is approximately 12 samples on the equipment that the authors of [1] use. So at least we can estimate τ with 12 samples accuracy. The accuracy of the estimation determines the cardinality of the variables τ. For example, if the time of arrival of a pulse i is assumed to be $t_i \pm 2$ then cardinality of the domain of the variable τ_i is 5 (τ_i can be $t_i-2, t_i-1, t_i, t_i+1, t_i+2$).

- The variables $\underline{x}_{1i}$ and $\underline{x}_{1i}$ have cardinality 4 corresponding to 4 colors of light or to four bases in positions i of the sequences. The length of the vectors is 4 but only the combinations $\{1000, 0100, 0010, 0001\}$ are allowed since a pulse in a given sequence at time t can appear in a single color only.

- Compute the probabilities $f(y_i|*)$ for each configuration of the variables α_{1i}, α_{2i}, τ_{2i}, τ_{3i}, $\underline{x}_1$ and $\underline{x}_2$ for each pulse using (4.5).

- Initialize the message passing algorithms on the graph in Figure 4.2. The messages from the variable nodes can be initialized to unity.

- Perform propagation on the graph using the update equations for the variable and function nodes (2.2) and (2.3), respectively.

- After the algorithm converged, for each pulse i at the nodes $\underline{x}_{1i}$ and $\underline{x}_{2i}$ compute the marginals using the expression (2.4). The marginals represent the global function (4.3) (the function under $\arg\max$) marginalized over all variables but $\underline{x}_{1i}$ (at the node $\underline{x}_{1i}$) and $\underline{x}_{2i}$ (at the node $\underline{x}_{1i}$). In other words,

the marginals up to a scale factor are the marginal distributions of $\underline{x}_{1i}$ and $\underline{x}_{1i}$ given the observations $\underline{y}$.

- For each i select the index corresponding to the largest value of the marginals $\underline{x}_{1i}$ and $\underline{x}_{2i}$ as the decoded colour/DNA base. Selected indices are the MAP estimates of the bases $\hat{x}_{1i}$ and $\hat{x}_{2i}$.

The authors note that belief propagation of the graph in Figure 4.2 is prohibitively complex and proceed to develop a method where they at first, determine the location of peaks using deconvolution and then assign detected peaks to one of the sequences.

Here we will diverge from the original paper and apply the method of lowering the complexity of the sum-product algorithm using graph transformations. At first we determine the complexity of the algorithm in the original graph in Figure 4.2 then we transform the graph in order to lower the complexity.

We reiterate our assumptions that the distribution $f_\alpha(\alpha)$ is approximated with a discrete distribution and that the locations of the peaks $\underline{\tau_1}$ and $\underline{\tau_2}$ can be estimated with a certain accuracy. These assumptions allow us to implement the sum-product algorithm in the discrete form.

The variables x_{1i} and x_{2i} have cardinality $q_x = 4$, i.e., the variables take 4 values corresponding to 4 bases in position i of major and minor sequences. By q_α and q_τ we denote the cardinalities of the variables α and τ, respectively. The nodes $f(y_i|*)$ for $i > 1$ in the graph in Figure 4.2 have degree 18 which is quite large. The cardinality of the domain of the nodes is $Q_{fn} = 4^6 \cdot q_\tau^6 \cdot q_\alpha^6$ (the node

includes 6 variables of each type: α, τ and $\underline{x}$). The number of multiplications and summations necessary for the update of the node can be evaluated using (3.14), (3.15). As an example, let $q_\tau = 4$ and $q_\alpha = 4$ so that all domains have cardinality 4. We update the function node using the method described in Section 3.3.2, i.e., at first evaluate the partial products of the messages A_i and B_j for $i \in 1 \dots 17$ and $j \in 2 \dots 19$ and then compute the outgoing message to an edge k as $\sum A_{k-1} B_{k+1}$, where the summation is performed over all variables that are not present on edge k. The number of operations necessary to compute the terms $A_1, A_2, \dots, A_{17}$, $B_2, B_3, \dots, B_{19}$ and the outgoing messages is presented in Table 4.1. The reader can see that update of each node $f(y_1|*)$ requires $2.06 \cdot 10^{11}$ multiplications, $1.60 \cdot 10^{11}$ additions and $3.67 \cdot 10^{11}$ operations in total.

Regarding the other nodes in the graph, all nodes x, α and τ have degree 4 (for $i > 1$) and the number of multiplications necessary to update the nodes can be computed using expression 3.7. Considering the cardinality of the domains from our example above ($q_\tau = 4$ and $q_\alpha = 4$) the update of each of the nodes requires 24 multiplications. The nodes f_α and f_x have degree 1 and do not require any operations. The nodes f_τ have the domain $\{\tau_i, \tau_{i+1}\}$ with cardinality 16 (q_τ^2) and degree 2 which results in 32 multiplications and 24 additions. Hence, the complexity of the update of the graph in Figure 4.2 is dominated by the complexity of the update of the nodes $f(x_i|*)$.

Now we apply the factor graph transformations in order to reduce the complexity. It is apparent that it is desirable to reduce the degree of the function nodes $f(y_i|*)$ in Figure 4.2. This can be accomplished by joining the variables nodes.

Table 4.1: Computation of the number of operations necessary for update of the node $f(y_i|*)$

Note: All messages have cardinality 4, the degree of the node $f(y_i|*)$ is 18.

Terms A

Term A	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}	A_{11}	A_{12}
Size A	4	16	64	256	1024	4096	$1.64 \cdot 10^4$	$6.55 \cdot 10^4$	$2.62 \cdot 10^5$	$1.05 \cdot 10^6$	$4.19 \cdot 10^6$	$1.68 \cdot 10^7$
Multipl.	0	16	64	256	1024	4096	$1.64 \cdot 10^4$	$6.55 \cdot 10^4$	$2.62 \cdot 10^5$	$1.05 \cdot 10^6$	$4.19 \cdot 10^6$	$1.68 \cdot 10^7$

A_{13}	A_{14}	A_{15}	A_{16}	A_{17}	Total
$6.71 \cdot 10^7$	$2.68 \cdot 10^8$	$1.07 \cdot 10^9$	$4.29 \cdot 10^9$	$1.72 \cdot 10^{10}$	NA
$6.71 \cdot 10^7$	$2.68 \cdot 10^8$	$1.07 \cdot 10^9$	$4.29 \cdot 10^9$	$1.72 \cdot 10^{10}$	$2.29 \cdot 10^{10}$

Terms B

Term B	B_2	B_3	B_4	B_5	B_6	B_7	B_8	B_9	B_{10}	B_{11}	B_{12}	B_{13}
Size B	4	16	64	256	1024	4096	$1.64 \cdot 10^4$	$6.55 \cdot 10^4$	$2.62 \cdot 10^5$	$1.05 \cdot 10^6$	$4.19 \cdot 10^6$	$1.68 \cdot 10^7$
Multipl.	16	64	256	1024	4096	$1.64 \cdot 10^4$	$6.55 \cdot 10^4$	$2.62 \cdot 10^5$	$1.05 \cdot 10^6$	$4.19 \cdot 10^6$	$1.68 \cdot 10^7$	$6.71 \cdot 10^7$
Additions	12	48	192	768	3072	$1.23 \cdot 10^4$	$4.92 \cdot 10^4$	$1.97 \cdot 10^5$	$7.86 \cdot 10^5$	$3.15 \cdot 10^6$	$1.26 \cdot 10^7$	$5.03 \cdot 10^7$

B_{14}	B_{15}	B_{16}	B_{17}	B_{18}	$B_{19} = f_i$	Total
$6.71 \cdot 10^7$	$2.68 \cdot 10^8$	$1.07 \cdot 10^9$	$4.29 \cdot 10^9$	$1.72 \cdot 10^{10}$	$6.87 \cdot 10^{10}$	NA
$2.68 \cdot 10^8$	$1.07 \cdot 10^9$	$4.29 \cdot 10^9$	$1.72 \cdot 10^{10}$	$6.87 \cdot 10^{10}$	0	$9.16 \cdot 10^{10}$
$2.01 \cdot 10^8$	$8.05 \cdot 10^8$	$3.22 \cdot 10^9$	$1.29 \cdot 10^{10}$	$5.15 \cdot 10^{10}$	0	$6.87 \cdot 10^{10}$

Outgoing messages

Messages	μ_1	μ_2	μ_3	μ_4	μ_5	μ_6	μ_7	μ_8	μ_9	μ_{10}	μ_{11}	μ_{12}
Multipl.	0	16	64	256	1024	4096	$1.64 \cdot 10^4$	$6.55 \cdot 10^4$	$2.62 \cdot 10^5$	$1.05 \cdot 10^6$	$4.19 \cdot 10^6$	$1.68 \cdot 10^7$
Additions	0	12	60	252	1020	4092	$1.64 \cdot 10^4$	$6.55 \cdot 10^4$	$2.62 \cdot 10^5$	$1.05 \cdot 10^6$	$4.19 \cdot 10^6$	$1.68 \cdot 10^7$

μ_{13}	μ_{14}	μ_{15}	μ_{16}	μ_{17}	μ_{18}	Total
$6.71 \cdot 10^7$	$2.68 \cdot 10^8$	$1.07 \cdot 10^9$	$4.29 \cdot 10^9$	$1.72 \cdot 10^{10}$	$6.87 \cdot 10^{10}$	$9.16 \cdot 10^{10}$
$6.71 \cdot 10^7$	$2.68 \cdot 10^8$	$1.07 \cdot 10^9$	$4.29 \cdot 10^9$	$1.72 \cdot 10^{10}$	$6.87 \cdot 10^{10}$	$9.16 \cdot 10^{10}$

Total multiplications: $2.06 \cdot 10^{11}$ Total additions: $1.60 \cdot 10^{11}$ Total operations: $3.67 \cdot 10^{11}$

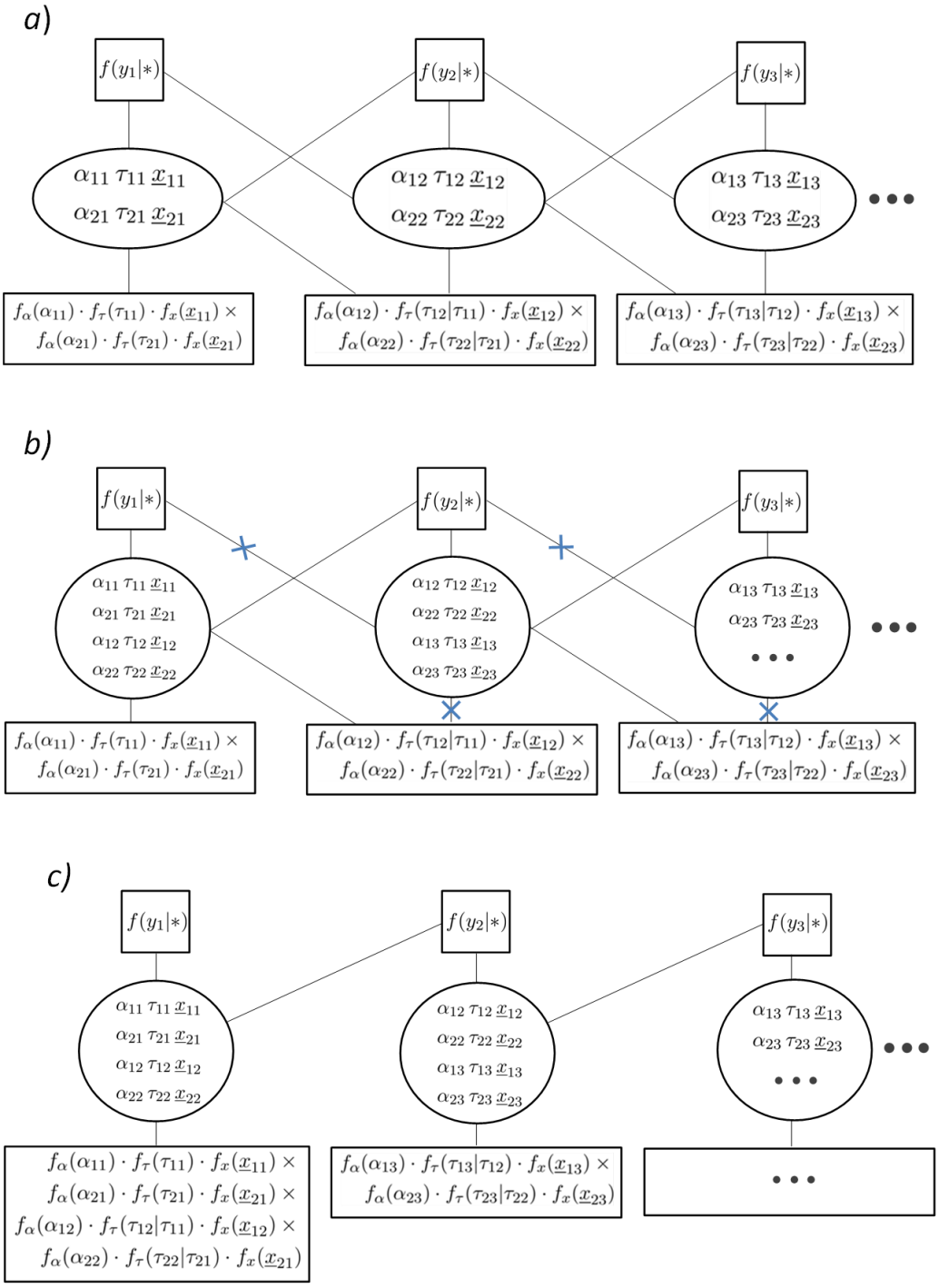


Figure 4.3: Transformed graph of Joint-DNA Base Calling.

Figure 4.3a shows a graph where we at first, joined all variable nodes related to the same pulse number: x_{1i}, x_{2i}, α_{1i}, α_{2i}, τ_{1i} and τ_{2i}). Then we joined the function nodes that represent a priori probabilities. As a result the degrees of the nodes $f(y_i|*)$ for $i > 1$ is lowered to 3. We may proceed further and transform the graph to the cycle-free form. To do this we stretch the variables α_{1i}, τ_{1i}, $\underline{x}_{1i}$, α_{2i}, τ_{2i}, and $\underline{x}_{2i}$ to the node with the variables from the previous pulse, i.e., pulse $i-1$. This transformation is presented in Figure 4.3b. The edges between the nodes $f(y_i|*)$ and the variable node "on the right" (the node that represent the variables associated with the pulse $i+1$) can now be removed. This became possible since the variable nodes that used to have the variables related to pulse i now also include the variables related to the pulse $i+1$. For the same reason one of the edges connected to the function node with a priori probabilities can be removed. We removed the edges indicated in Figure 4.3b with crosses and arrived to the cycle-free representation in Figure 4.3c.[4].

It is important to note that these transformations did not change the domains of $f(y_i|*)$ and the cardinality of the domains is still $Q_{fn} = 4^6 \cdot q_\tau^6 \cdot q_\alpha^6$. The degree of the nodes has been reduced from 18 to 2. The cardinalities of the messages in the graph are now $Q_\mu = 4^4 \cdot q_\tau^4 \cdot q_\alpha^4$ (the message includes the information about 4 variables of each type). The function nodes can be updated in the following way, assuming the node $f(y_i|*)$ received a message on the edge from the nodes "on the left", then the message is parameterized by the variables from the pulses $i-1$

[4]In the figure we also clustered the function nodes representing a priory probabilities of the pulses 1 and 2

and i. The values of the message is multiplied by the local function $f(y_i|*)$. This requires Q_{fn} multiplications. The product is parameterized by the variables from the pulses $i-1$, i and $i+1$. The message sent is computed by marginalizing out the variables that correspond to the pulse $i-1$ so that the message is parameterized by the variables related to the pulses i and $i+1$. The marginalization requires $Q_\mu(Q_{fn}/Q_\mu - 1)$ additions (the message has Q_μ values and each value is sum of Q_{fn}/Q_μ values of the products). The message sent from the node to the left is computed in the same way. In total, the update of the node $f(y_i|*)$ requires $2Q_{fn}$ multiplications and $2Q_\mu(Q_{fn}/Q_\mu - 1)$ additions. For example, applying the cardinalities of the domains from the example above ($q_\alpha = 4$ and $q_\tau = 4$) we determined that $Q_{fn} = 6.87 \cdot 10^{10}$, $Q_{mu} = 1.68 \cdot 10^7$ and the update of the node requires $1.37 \cdot 10^{10}$ multiplications and $1.37 \cdot 10^{10}$ additions. Compared to the original case the total number of operations necessary to update the node has been lowered from $3.67 \cdot 10^{11}$ to $2.74 \cdot 10^{11}$ which constitutes a reduction of 25%.[5].

The complexities of the update of the variable nodes have increased. The variable nodes have degree 3 but the messages to the nodes with a priori probabilities do not need to be computed. The message sent "to the right" (to the $f(y_{i+1}|*)$) is the message received from the node $f(y_i|*)$ multiplied by the message with a priori probabilities. Similarly, the message to the node $f(y_i|*)$ is the message from the node $f(y_{i+1}|*)$) multiplied by a priori probabilities. Hence, the update of

[5]This actually shows that the method of the update of a function node proposed in Section 3.3.2 is very effective. The update of a node of degree 18 requires just 33% more operations compared to the update of a node of degree 2! If we used the "straightforward method" of the update of the node then the increase in the number of operations would be more than 20 times.

the node requires $2Q_\mu$ multiplications. For our example, with $q_\alpha = 4$ and $q_\tau = 4$ the number of multiplications is $3.36 \cdot 10^7$ which is still small compared to the complexity of the function node.

In order to compute the marginals and MAP estimates of $\underline{x}_{1i}$ and $\underline{x}_{2i}$ we have to multiply all incoming messages (from the nodes $f(y_{i+1}|*)$, $f(y_i|*)$ and the node with a priori probabilities) and marginalize the product for all variables but $\underline{x}_{1i}$ and $\underline{x}_{2i}$. This requires $2Q_\mu$ multiplications and approximately Q_μ additions which is again much less compared to the number of operations necessary to update the function nodes.

We will now discuss the memory requirements of the sum-product algorithm applied to the original and transformed graphs. As has been noted in the Section 3.3.3 the sum-product algorithm requires memory to store the tables of local functions and the message. The number of units[6] necessary to store the function $f(y_i|*)$ is Q_{fn}. This value did not change since the domains of the local function were not changed by the transformation. On the original graph the amount of memory necessary to store the messages is negligible compared to the amount of memory necessary to store the local functions. For example, there are 26 edges originating from the nodes associated with the pulse i and considering the case where $q_\alpha = 4$, $q_\tau = 4$ the amount of the memory required to store the messages is 104 units which is negligible compared to $Q_{fn} = 6.87 \cdot 10^{10}$. In the transformed graph the variable nodes have the degree 3 so that $3Q_\mu = 5.03 \cdot 10^7$ units necessary

[6]We assume that 1 unit is the number of bits necessary to store a single value of message or local function.

to store the message which is still small compared to Q_{fn}.

To conclude, by transforming the factor graph corresponding to the Joint DNA Base Calling we succeeded in not only converting the graph to the cycle-free form but also in lowering the number of operations required by the sum-product algorithm. We showed that the number of operations necessary to find MAP estimates of the DNA bases on the graph in the cycle-free form in Figure 4.3c is approximately 25% less compared to the number of operations necessary for *a single iteration* of the sum-product algorithm on the graph with cycles in Figure 4.2. The cycle-free implementation has a clear advantage since each node needs to be updated only once and the marginals (MAP estimates) inferred by the sum-product algorithm are exact. As for the exact number of operations necessary to perform the propagation, even on the cycle-free realization, the number of operations remains rather large for practical implementation.

4.2 Decoding of the (7,4) Hamming Code

Decoding of the linear error-correcting codes is an important application of the sum-product algorithm. In this section, we apply the method of lowering the complexity of the sum-product algorithm using factor graph transformations to the framework of iterative decoding. As an example of a code we use the (7,4) Hamming code, see for example [41].

Channel coding is an indispensable part of a digital communication system. A linear code is defined either by a generator matrix G or by a parity check matrix H. The codeword c is a vector of symbols such that $Hc = 0$. On the transmitter side a

stream of bits is split into blocks, encoded, mapped into channel symbols and then transmitted over a channel. By "channel" we refer to the transmission medium which is subject to noise, distortion and other impediments. On the receiver side the decoder has to estimate the transmitted codeword from noisy observations of transmitted symbols.

Let x_i be a binary data symbol $x_i = \{0,1\}$ which is mapped into the channel symbols $s_i = \{-\sqrt{E_b}, +\sqrt{E_b}\}$ and transmitted over a channel. By E_b we denote the energy per bit and by y_i we denote the output of the matched filter at the receiver which corresponds to the transmission of x_i. One of the most commonly used models of a channel is Additive White Gaussian Noise (AWGN) channel. In the AWGN channel model $y_i = x_i + n_i$, where n_i is a sample of zero-mean white Gaussian noise with variance σ^2. The probability that y_i is received given that x_i is transmitted is expressed as:

$$p(y_i|x_i) = \frac{\exp\left\{-\frac{(y_i - s_i)^2}{2\sigma^2}\right\}}{\sqrt{2\pi}\sigma} \tag{4.6}$$

where s_i corresponds to the transmitted data symbol x_i. The quantity $\frac{E_b}{2\sigma^2} = \frac{E_b}{N_0}$ is referred to as Signal to Noise Ratio (SNR) of a digital communication system and N_0 is the power spectral density of White Gaussian Noise.

Assuming that a binary stream $(x_1, \ldots, x_n)$ of length n has been transmitted over a AWGN memoryless channel then the likelihood the outputs $(y_1, \ldots, y_n)$

can be expressed as:

$$p(y_1,\ldots,y_n|x_1,\ldots,x_n) = \prod_{i=1}^{n} p(y_i|x_i) \tag{4.7}$$

where $p(y_i|x_i)$ is expressed as (4.6).

In the case where the channel coding is used, the sequence $(x_1,\ldots,x_n)$ represents a codeword. Since a row of H multiplied by a codeword must yield 0, the rows of the matrix define a set of constraints (parity checks) imposed on the codeword symbols x_i. For a code of length n we may express the constraint corresponding to a row j by a parity check function $C_j(X_j)$, where X_j is a set of the transmitted symbols which participate in the parity check j. For given values of the variables X_j the function $C_j(X_j)$ is equal to 1 if the parity check is satisfied and 0 otherwise.

With the assumptions that the channel is memoryless the Maximum Likelihood (ML) bit decoding of a symbol x_i of a codeword can be expressed as:

$$\hat{x_i} = \underset{x_i}{\operatorname{argmax}} \sum_{X\setminus x_i} \left(\prod_{j=1}^{n-k} C_j(X_j) \prod_{i=1}^{n} p(y_i|x_i) \right) \tag{4.8}$$

where n is the length of a codeword, k is the number of encoded information bits, $X\setminus x_i$ is the set of all symbols $(x_1,\ldots,x_n)$ except for the symbol x_i. The expression (4.8) is an instance of a MPF problem and can be solved using the sum-product algorithm. The global function in this case is the product under the $\sum$ in (4.8). The factor graph corresponding to the global function is also referred to as Tanner

graph.

The variable nodes of the Tanner graph correspond to the symbols x_i of a codeword. The function nodes correspond to the parity check function $C_j(X_j)$ as well as conditional probabilities $p(y_i|x_i)$. A symbol x_i participates in the parity check C_j only if there is a non-zero element in the position i of a row j of the H matrix. This property is reflected in the Tanner graph by the check node C_j being connected to the variable node x_i only if there is a non-zero element in position i of row j. In general, the symbols and elements of H can be non-binary, but for this discussion we make the assumption that the symbols are binary, so each symbol is 0 or 1.

The (7,4) Hamming code (see for example [41]) can be defined by the parity check matrix:

$$H = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{bmatrix} \tag{4.9}$$

The global function corresponding to the code is expressed as:

$$\begin{aligned} G(x_1,\ldots,x_7) = {} & C_1(x_1,x_4,x_6,x_7)C_1(x_2,x_4,x_5,x_6)C_1(x_3,x_5,x_6,x_7) \times \\ & p(y_1|x_1)p(y_2|x_2)p(y_3|x_3)p(y_4|x_4)p(y_5|x_5)p(y_6|x_6)p(y_7|x_7) \end{aligned} \tag{4.10}$$

The terms C_1, C_2 and C_3 represent parity check corresponding to the rows 1,2,3 of

the matrix (4.9). For example, $C_1(x_1, x_4, x_6, x_7)$ is defined as:

$$C_1(x_1, x_4, x_6, x_7) = \begin{cases} 1, \text{ if } x_1 \oplus x_4 \oplus x_6 \oplus x_7 = 0; \\ 0, \text{ otherwise} \end{cases} \tag{4.11}$$

where $\oplus$ is the operation of modulo-2 addition.

The factor graph corresponding to the code is depicted in Figure 4.4. The function nodes $p_1, \ldots, p_7$ on the graph represent conditional probabilities $p(y_1|x_1), \ldots, p(y_7|x_7)$ of transmitted bits given the noisy observations of received bits.

The decoding on a Tanner graph can be implemented in several ways. In one possible implementations, messages on edges connected to a node x_i represent probabilities of value of the symbols x_i. This implementation is commonly referred to as the implementation in the probability domain. In the binary case the messages $p(x_i = 0) + p(x_i = 1) = 1$ and only a single value such as $p(x = 1)$ can be sent on an edge. The message sent by a variable node x_i to a function (parity check) node C_j is expressed as:

$$\mu_{x_i \to C_j} = \frac{\prod_{C_k \in N(x_i) \backslash C_j} \left(\mu_{C_k \to x_i}\right)}{\prod_{C_k \in N(x_i) \backslash C_j} \left(\mu_{C_k \to x_i}\right) + \prod_{C_k \in N(x_i) \backslash C_j} \left(1 - \mu_{C_k \to x_i}\right)} \tag{4.12}$$

where $N(x_i) \backslash C_j$ is the set of neighbors of a variable node x_i except for the node C_j and $\mu_{C_k \to x_i}$ is a message sent by a node C_k and received by the node x_i. The denominator of (4.14) ensures that the probabilities of $p(x = 1)$ and $p(x = 0)$ adds up to unity.

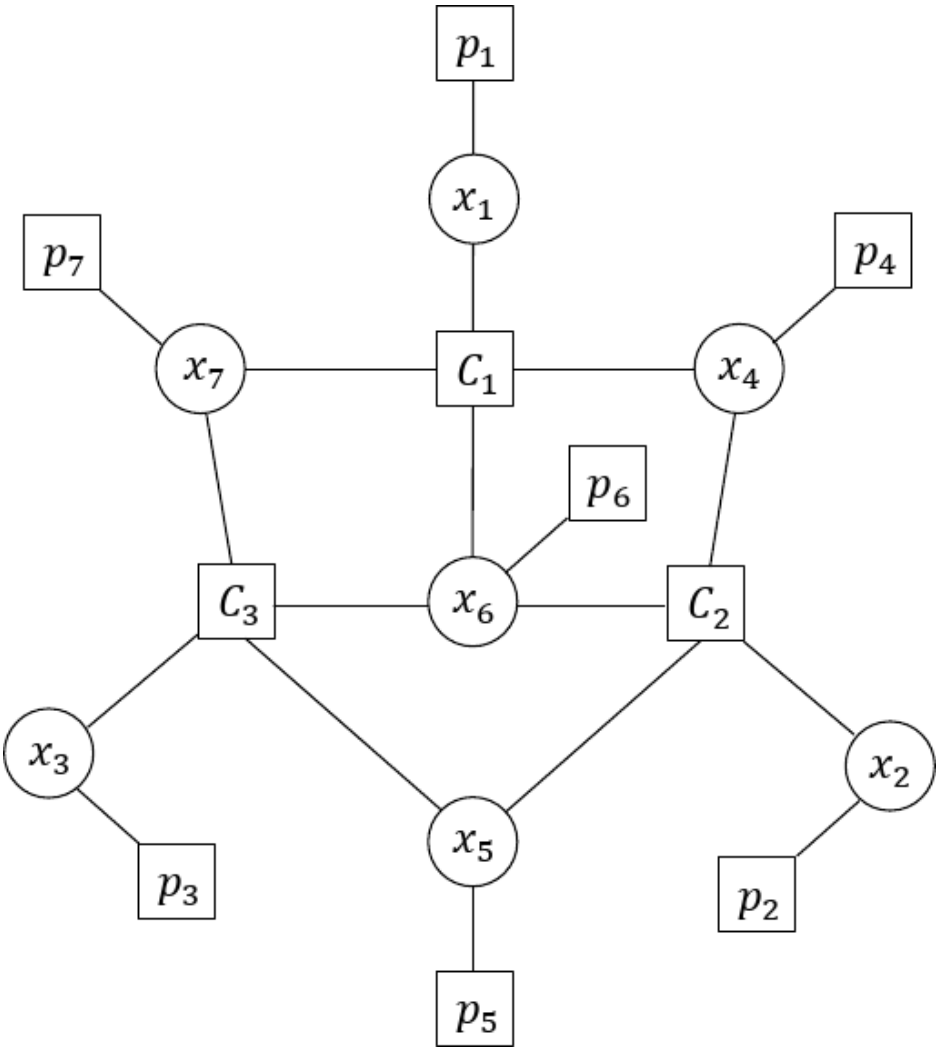


Figure 4.4: Factor graph of the (7,4) Hamming code.

In the numerator and denominator of (4.14) we need to evaluate $d(x_i)$[7] products of all but one incoming messages. This can be done using the method described in Section 3.3.1 and requires $3(d(x_i)-2)$ multiplications. Evaluation of (4.14) requires $6(d(x_i)-2)$ multiplications, $d(x_i)$ subtractions, 1 division and 1 addition.

The messages from a parity check node C_j to a variable node x_i is expressed as:

$$\mu_{C_k \to x_i} = \sum_{X_j \backslash x_i} C_j(X_j) \prod_{x_k \in N(C_j) \backslash x_i} p(x_k) \tag{4.13}$$

where $p(x_k) = \mu_{x_k \to C_j}$ for $x_k = 1$ and $p(x_k) = 1 - \mu_{x_k \to C_j}$ for $x_k = 0$. For x_i to be equal to 1 the number of 1s in the vector of the variables in $X_j \backslash x_i$ has to be odd. Hence, the equation (4.13) represents the sum of the probabilities of the vectors of the variables $X_j \backslash x_i$ which has an odd number of 1s. For example, if C_j includes $\{x_1, x_2, x_3, x_4\}$ and we wish to compute the message to x_3 then the probability that $x_3 = 1$ is:

$$\begin{aligned} p(x_3 = 1) = {} & p(x_1 = 1, x_2 = 0, x_4 = 0) + p(x_1 = 0, x_2 = 1, x_4 = 0) \\ & + p(x_1 = 0, x_2 = 0, x_4 = 1) + p(x_1 = 1, x_2 = 1, x_4 = 1) \end{aligned}$$

The algorithm may start by initializing all messages from the x_i to nodes C_j to the values of conditional probabilities $p(y_i | x_i = 1)$. Then the algorithm proceeds

[7] $d(x_i)$ denotes the degree of the node x_i

by updating the check nodes and the variable nodes using the expressions (4.14) and (4.13). The messages towards the nodes with the conditional probabilities $p(y_i|x_i)$ do not need to be updated. Assuming that the algorithm converged, the marginal probability of x_i is computed as:

$$m_{x_i} = \frac{\prod_{C_k \in N(x_i)} \left(\mu_{C_k \to x_i}\right)}{\prod_{C_k \in N(x_i)} \left(\mu_{C_k \to x_i}\right) + \prod_{C_k \in N(x_i)} \left(1 - \mu_{C_k \to x_i}\right)} \tag{4.14}$$

If the value $m_{x_i} < 0.5$ then the ML decision on the bit x_i value is 0, otherwise it is 1. The symbols decoded in this way may or may not form a valid codeword since it is ML bit decoding. If the symbols do not form a codeword then the decoding has failed.

In the other possible implementation of the sum-product algorithm, the messages represent Log-Likelihood-Ratios (LLR) of probabilities. The LLR of a symbol x_i is expressed as $LLR(x_i) = \ln(\frac{p(y_i|x_i=0)}{p(y_i|x_i=1)})$. The implementation in the LLR domain, compared to the sum-product algorithm in its original form, is numerically more stable and requires fewer quantization bits (see for example [43] and reference therein). The sum-product algorithm in the LLR domain can be implemented in several ways including the use of the tangent hyperbolic function and the Jacobian logarithm [20]. In the case of the LLR domain, the update rule for a

check node C_j is defined as:

$$\mu_{C_j \to x_i} = \left(\prod_{k \in N(i) \setminus j} \text{sign}(\mu_{x_k \to C_j}) \right) \times 2 \tanh^{-1} \left(\prod_{k \in N(i) \setminus j} \tanh \left(\frac{|\mu_{x_k \to C_j}|}{2} \right) \right) \tag{4.15}$$

where the function $sign(x) = 1$ if $x > 0$ and -1 otherwise.

The update of a check node of degree d_v as expressed by (4.15) requires d_v divisions by 2, d_v evaluations of tanh and $\tanh^{-1}$ functions and $3(d_v - 2) + d_v$ multiplications.[8] There are serval ways to implement the tanh and $\tanh^{-1}$ functions in hardware including piecewise linear and non-linear approximations and lookup tables, see for example [44] and references therein. The expression (4.15) can also be represented as 2D lookup table [20].

The update of a variable node x_i in LLR domain is defined as:

$$\mu_{x_i \to C_j} = \sum_{k \in N(i) \setminus j} \mu_{C_k \to x_i} \tag{4.16}$$

The update of a variable node of degree d_v requires $3(d_v - 2)$ additions.

Now we will consider the complexity of the sum-product algorithm in the case of decoding of the Hamming (7,4) code. At first we consider the sum-product in its original form, as described by the equations (4.14) and (4.14), then we discuss the implementations of the algorithm in LLR domain.

Consider the graph in Figure 4.4. The messages to the nodes $p_1, \ldots, p_7$ rep-

[8] Here we use the same approach as in Section 3.3.1.

resent the conditional probabilities $p(y_1|x_1),\ldots,p(y_7|x_7)$ and do not need to be computed. We assume that the messages represent the probabilities that the bits are equal to 1.

The variable nodes x_1, x_2 and x_4 have degree 2 and do not require operations. The node x_4, x_5 and x_7 have degree 3. We will determine the number of operations necessary to update the nodes on the example of the node x_4. The message sent by the node x_4 to the node C_2 is computed as:

$$\mu_{x_4\to C_1} = \frac{\mu_{C_1\to x_4} p(y_4|x_4=1)}{\mu_{C_1\to x_4} p(y_4|x_4=1) + (1-\mu_{C_1\to x_4})\, p(y_4|x_4=0)}$$

The message from x_4 to the node C_1 is computed in similar way. The update of the node requires 4 multiplications, 2 subtractions, 2 additions and 2 divisions, which is 10 operations in total. The update of the nodes x_5 and x_7 is performed in the same way and requires the same number of operations.

The node x_6 has degree 4. The message from the node to the node C_1 is computed as:

$$\mu_{x_6\to C_1} = \frac{\mu_{C_2\to x_6}\mu_{C_3\to x_6} p(y_6|x_6=1)}{\mu_{C_2\to x_6}\mu_{C_3\to x_6} p(y_6|x_6=1) + \left(1-\mu_{C_2\to x_6}\right)\left(1-\mu_{C_3\to x_6}\right) p(y_6|x_6=0)}$$

The messages to the nodes C_2 and C_3 are computed in similar way. Some of the terms in the expressions for the messages sent to the nodes C_1, C_2 and C_3 are the same:

1. The terms $1-\mu_{C_1\to x_6}$, $1-\mu_{C_2\to x_6}$ and $1-\mu_{C_3\to x_6}$

2. The terms $\mu_{C_3 \to x_6} p(y_6|x_6 = 1)$ and $\left(1 - \mu_{C_3 \to x_6}\right) p(y_6|x_6 = 0)$

These terms can be computed once. Taking this into account, the update of the node x_6 requires 10 multiplications, 3 additions, 3 subtractions, 3 divisions which in total is 19 operations.

Now we will consider the complexity of the update of the function nodes C_1, C_2 and C_3. The message from the node C_1 to the node x_1 is computed as:

$$\mu_{C_1 \to x_1} = (1 - \mu_{x_4 \to C_1})\, \mu_{x_6 \to C_1} \mu_{x_7 \to C_1} + \mu_{x_4 \to C_1} \left(1 - \mu_{x_6 \to C_1}\right) \mu_{x_7 \to C_1}$$
$$+ \mu_{x_4 \to C_1} \mu_{x_6 \to C_1} (1 - \mu_{x_7 \to C_1}) + (1 - \mu_{x_4 \to C_1}) \left(1 - \mu_{x_6 \to C_1}\right) (1 - \mu_{x_7 \to C_1})$$

The message is the sum of probabilities of configurations of the variables x_4, x_6 and x_7 which correspond to $x_1 = 1$. The messages sent towards the nodes x_4, x_6 and x_7 are computed in a similar way. The terms $1 - \mu$ for four edges can be computed once. The messages to x_1 and x_4 include the same terms which can be computed once: $\left(1 - \mu_{x_6 \to C_1}\right) \mu_{x_7 \to C_1}$, $\mu_{x_6 \to C_1} (1 - \mu_{x_7 \to C_1})$, $\mu_{x_6 \to C_1} \mu_{x_7 \to C_1}$ and $\left(1 - \mu_{x_6 \to C_1}\right) \left(1 - \mu_{x_6 \to C_1}\right)$. Similarly, the terms $(1 - \mu_{x_1 \to C_1})\, \mu_{x_4 \to C_1}$, $\mu_{x_1 \to C_1} (1 - \mu_{x_4 \to C_1})$, $\mu_{x_1 \to C_1} \mu_{x_4 \to C_1}$ and $(1 - \mu_{x_1 \to C_1})(1 - \mu_{x_4 \to C_1})$ are the same for the messages to the nodes x_6 and x_7. Taking this into consideration, 24 multiplications, 12 additions and 4 subtractions or 40 operations in total are required to update the node. The nodes C_2 and C_3 are updated in a similar fashion and require the same number of operations.

We conclude that a single iteration of the sum-product algorithm in the graph in Figure 4.4 requires in total 169 operations: 94 multiplications, 45 additions,

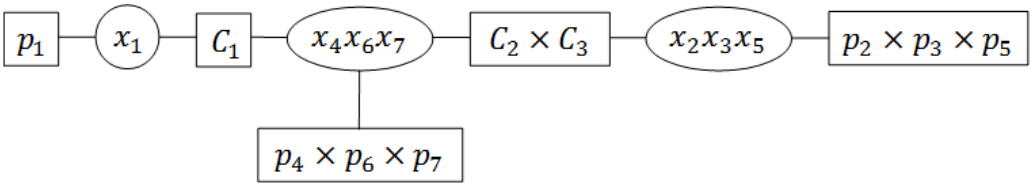


Figure 4.5: One of the possible transformations of the factor graph corresponding to the Hamming code.

21 subtractions, 9 divisions. The detailed operation counts are presented in the Table 4.2. Considering that in average approximately 10 iterations are necessary for convergence of the sum-product algorithm, the average number of operations necessary to decode a codeword on the graph in Figure 4.4 is 1690.

Now consider the factor graph presented in Figure 4.5. The graph is formed from the graph in Figure 4.4 by clustering the following nodes:

1. C_2 and C_3
2. x_4, x_6, and x_7
3. x_2, x_3, and x_5
4. p_4, p_6, and p_7
5. p_2, p_3 and p_5

On the transformed graph the messages sent to and from the nodes $x_4x_6x_7$ and $x_2x_3x_5$ are the joint probabilities of three included variables. Only 7 values of the probabilities are independent since the values have to sum to unity. Hence we could have messages with 7 values. It is more efficient however, to have an 8-valued message, since otherwise, every time we update the nodes, we would have

Table 4.2: Number of operations required to update the graphs of the Hamming(7,4) code in the original and the cycle-free forms

Factor graph in the original form in Figure 4.4

Nodes	Number of nodes	Number of operations per iteration								
		Multiplications		Additions		Subtractions		Divisions		Total
		Per node	Total	Per node	Total	Per node	Total	Per node	Total	
x_1, x_2 and x_3	3	0	0	0	0	0	0	0	0	0
x_7, x_4 and x_5	3	4	12	2	6	2	6	2	6	30
x_6	1	10	10	3	3	3	3	3	3	19
C_1, C_2 and C_3	3	24	72	12	36	4	12	0	0	120
Total for nodes	10	94		45		21		9		**169**

Factor graph in the transformed form in Figure 4.5

Nodes	Number of nodes	Number of operations iteration								
		Multiplications		Additions		Subtractions		Divisions		Total
		Per node	Total	Per node	Total	Per node	Total	Per node	Total	
x_1	1	0	0	0	0	0	0	0	0	0
C_1	1	0	0	3	3	1	1	0	0	3
$x_4x_6x_7$	1	16	16	14	14	0	0	16	16	46
C_2C_3	1	0	0	8	8	0	0	0	0	8
$x_2x_3x_5$	1	0	0	0	0	0	0	0	0	0
Total for nodes	7	16		25		1		16		**58**

Reduction of the number of operations per iteration $\frac{169-58}{169}100\% = 65.5\%$
Additional operations which are necessary to decode a codeword on the transformed graph:
1) Finding joint probabilities $p_4\,p_6\,p_7$ and $p_2\,p_3\,p_5$ requires 24 multiplications.
2) Finding marginal bit probabilities for the bits $x_2,\ldots,x_7$ at the nodes $x_2x_3x_5$ and $x_4x_6x_7$ requires 60 operations:
24 multiplications and 36 additions.
Considering 10 iterations, 1690 operations are required in order to decode a codeword on the graph in Figure 4.4.
The total number of operations necessary to decode a codeword on the graph in Figure 4.5 is only 142.

to compute 8th value by adding 7 values and subtracting the result from unity which takes extra 8 operations. Hence, we consider the case where the messages sent to and from the nodes $x_4 x_6 x_7$ and $x_2 x_3 x_5$ have 8 values. Note that the update of the variable nodes of degree 2 which are x_1 and $x_2 x_3 x_5$ requires no operations.

The parity check node C_1 defines allowed configurations of variables, for example, if $x_1 = 1$ then four configurations of the variables $x_4 x_6 x_7$ are valid: 100, 010, 001, 111. Similarly, if $x_4 = 0$, $x_6 = 0$, $x_7 = 0$, then clearly $x_1 = 0$.

The message sent from the node C_1 to the variable node x_1 may still have a single value. The value is the sum of probabilities of the configurations of the variables $x_4 x_6 x_7$ that correspond to the $x_1 = 1$:

$$\begin{aligned}\mu_{C_1 \to x_1}(x_1 = 1) = \mu_{x_4 x_6 x_7 \to C_1}(x_4 = 1, x_6 = 0, x_7 = 0) \\ + \mu_{x_4 x_6 x_7 \to C_1}(x_4 = 0, x_6 = 1, x_7 = 0) \\ + \mu_{x_4 x_6 x_7 \to C_1}(x_4 = 0, x_6 = 0, x_7 = 1) \\ + \mu_{x_4 x_6 x_7 \to C_1}(x_4 = 1, x_6 = 1, x_7 = 1)\end{aligned}$$

The message from the node C_1 to the node $x_4 x_6 x_7$ represents the probability of the value of the variable x_1 which corresponds to the configuration of variables $x_4 x_6 x_7$. For example, the configuration $x_4 = 0, x_6 = 0, x_7 = 0$ corresponds to the $x_1 = 0$ and the configuration $x_4 = 1, x_6 = 0, x_7 = 0$ corresponds to the $x = 1$. So

we have:

$$\begin{aligned}
\mu_{C_1\to x_4x_6x_7}(x_4=0,x_6=0,x_7=0) &= 1-\mu_{x_1\to C_1}(x_1=1) = p(y_1|x_1=0)\\
\mu_{C_1\to x_4x_6x_7}(x_4=1,x_6=0,x_7=0) &= \quad \mu_{x_1\to C_1}(x_1=1) \quad = p(y_1|x_1=1)\\
&\vdots\\
\mu_{C_1\to x_4x_6x_7}(x_4=1,x_6=1,x_7=1) &= \quad \mu_{x_1\to C_1}(x_1=1) \quad = p(y_1|x_1=1)
\end{aligned}$$

Therefore, the update of the node C_1 requires 3 additions and 1 subtraction.

It is clear that the message sent to and from the node $x_4x_6x_7$ depends on the variables x_4, x_6, and x_7. In order to make the notation more concise, from this point onward, we will not repeat the names of the variables in the message notation, e.g., instead of $\mu_{C_1\to x_4x_6x_7}(x_4=1,x_6=1,x_7=1)$ we will write $\mu_{C_1\to x_4x_6x_7}(111)$. In order to update the node $x_4x_6x_7$ we need to compute two 8-valued messages $\mu x_4x_6x_7 \to C_1$ and $\mu x_4x_6x_7 \to C_2C_3$. The messages are computed in a way similar to the binary case with exception that now we have 8 values. The message $\mu_{x_4x_6x_7\to C_2C3}$ is expressed as:

$$\begin{aligned}
\mu_{x_4x_6x_7\to C_2C_3}(000) &= \frac{\mu_{C_1\to x_4x_6x_7}(000)p(y_4,y_6,y_7|x_4=0,x_6=0,x_7=0)}{\sum_{x_4,x_6,x_7}\mu_{C_1\to x_4x_6x_7}(x_4,x_6,x_7)p(y_4,y_6,y_7|x_4,x_6,x_7)}\\
\mu_{x_4x_6x_7\to C_2C_3}(100) &= \frac{\mu_{C_1\to x_4x_6x_7}(100)p(y_4,y_6,y_7|x_4=1,x_6=0,x_7=0)}{\sum_{x_4,x_6,x_7}\mu_{C_1\to x_4x_6x_7}(x_4,x_6,x_7)p(y_4,y_6,y_7|x_4,x_6,x_7)}\\
&\vdots\\
\mu_{x_4x_6x_7\to C_2C_3}(111) &= \frac{\mu_{C_1\to x_4x_6x_7}(111)p(y_4,y_6,y_7|x_4=1,x_6=1,x_7=1)}{\sum_{x_4,x_6,x_7}\mu_{C_1\to x_4x_6x_7}(x_4,x_6,x_7)p(y_4,y_6,y_7|x_4,x_6,x_7)} \qquad (4.17)
\end{aligned}$$

Considering our assumption that the channel is memoryless, the 8-valued conditional probability function $p(y_4, y_6, y_7|x_4, x_6, x_7)$ is evaluated as the product of the probabilities $p(y_4|x_4)p(y_6|x_6)p(y_7|x_7)$.[9] Using the approach described in Section 3.3.2 the product can be evaluated with 12 multiplications.

The denominator of (4.17), which ensures that the sum of the probabilities is equal to 1, is same for all configurations of the variables. The denominator is the sum of 8 terms $\mu_{C_1 \to x_4 x_6 x_7} p(y_4, y_6, y_7|x_4, x_6, x_7)$ and requires 8 multiplications and 7 additions. Moreover, the products of the messages with the conditional probabilities $\mu_{C_1 \to x_4 x_6 x_7} p(y_4, y_6, y_7|x_4, x_6, x_7)$ are the same in the numerator and denominator and can be evaluated a single time. In total, the evaluation of 8 values of the message $\mu_{x_4 x_6 x_7 \to C_2 C_3}$ requires 8 multiplications, 7 additions, and 8 divisions. The message $\mu_{x_4 x_6 x_7 \to C_1}$ can be computed in a similar way. The message to the node $p_4\, p_6\, p_7$ does not need to be computed. Therefore, the update of the node $x_4 x_6 x_7$ requires a total of 46 operations: 16 multiplications, 14 additions and 16 divisions.

The message $\mu_{C_2 C_3 \to x_2 x_3 x_5}$ is the sum of the probabilities of the configurations of the variables x_4, x_6, x_7 which, for a given configuration of variables x_2, x_3, x_5, satisfy both parity checks $C_2(x_2, x_4, x_5, x_6)$ and $C_3(x_3, x_5, x_6, x_7)$. For example, in the case where $x_2 = 0$, $x_3 = 0$, $x_5 = 0$ the configurations $x_4 = 0, x_6 = 0, x_7 = 0$ and $x_4 = 1, x_6 = 1, x_7 = 1$ satisfy both parity checks so $\mu_{C_2 C 3 \to x_2 x_3 x_5}(000) = \mu_{x_4 x_6 x_7 \to C_2 C_3}(000) + \mu_{x_4 x_6 x_7 \to C_2 C_3}(111)$. Similarly, for the other values of x_2, x_3,

[9] We assume that the probabilities $p(y_1|x_1), \ldots, p(y_7|x_7)$ have to be evaluated for both cases, on original and on transformed graphs. Hence, we do not consider the complexity associated with computation of the probabilities.

x_5 we obtain:

$$\mu_{C_2C_3\to x_2x_3x_5}(100) = \mu_{x_4x_6x_7\to C_2C_3}(100) + \mu_{x_4x_6x_7\to C_2C_3}(011)$$

$$\mu_{C_2C_3\to x_2x_3x_5}(010) = \mu_{x_4x_6x_7\to C_2C_3}(110) + \mu_{x_4x_6x_7\to C_2C_3}(001)$$

$$\mu_{C_2C_3\to x_2x_3x_5}(110) = \mu_{x_4x_6x_7\to C_2C_3}(010) + \mu_{x_4x_6x_7\to C_2C_3}(101)$$

$$\mu_{C_2C_3\to x_2x_3x_5}(001) = \mu_{x_4x_6x_7\to C_2C_3}(010) + \mu_{x_4x_6x_7\to C_2C_3}(101) = \mu_{C_2C_3\to x_2x_3x_5}(110)$$

$$\mu_{C_2C_3\to x_2x_3x_5}(101) = \mu_{x_4x_6x_7\to C_2C_3}(110) + \mu_{x_4x_6x_7\to C_2C_3}(001) = \mu_{C_2C_3\to x_2x_3x_5}(010)$$

$$\mu_{C_2C_3\to x_2x_3x_5}(011) = \mu_{x_4x_6x_7\to C_2C_3}(100) + \mu_{x_4x_6x_7\to C_2C_3}(011) = \mu_{C_2C_3\to x_2x_3x_5}(100)$$

$$\mu_{C_2C_3\to x_2x_3x_5}(111) = \mu_{x_4x_6x_7\to C_2C_3}(000) + \mu_{x_4x_6x_7\to C_2C_3}(111) = \mu_{C_2C_3\to x_2x_3x_5}(000)$$

As the reader can see the latter four messages are equal to the former four messages. The message from $\mu_{C_2C_3\to x_4x_6x_7}$ is computed in a similar way and the update of the node C_2C_3 requires only 8 additions.

The summary of the number of operation necessary to update the nodes of the graph in Figure 4.5 is presented in the Table 4.2. In total, the update of the nodes requires 16 multiplications, 25 additions, 1 subtraction and 16 divisions, which constitutes a total of 58 operations. As it has been shown above, the update of the nodes on the graph prior to the transformations requires 169 operations. Therefore, by transforming the graph we decreased the number of operations necessary for a single iteration from 169 to 58 which constitutes a reduction of 65.5%.

On the transformed in Figure 4.5 additional operations are required in order to find the joint probabilities $p(y_2,y_3,y_5|x_2,x_3,x_5)$ and $p(y_4,y_6,y_7|x_4,x_6,x_7)$ as

well as in order to obtain the marginal probabilities of the bits $x_2, x_3, \ldots, x_7$. As discussed above, the computation of each of the joint probabilities requires 12 multiplications.

The marginal probabilities of the bits $x_2, x_3, \ldots, x_7$ are computed as follows. The product of the messages $\mu_{C_2C_3 \to x_4x_6x_7}$, $\mu_{C_1 \to x_4x_6x_7}$ and conditional probability $p(y_4, y_6, y_7 | x_4, x_6, x_7)$ is proportional to the joint a posteriori probability $p(x_4, x_6, x_7)$. Evaluation of the product requires 16 multiplications. The values of the variables x_4, x_6, and x_7 that correspond to the maximal joint pmf $p(x_4, x_6, x_7)$ correspond to *joint* ML decoding of the bits x_4, x_6, and x_7. In order to find the ML values of a single bit we have to marginalize out the rest of the variables from the joint pmf. For example, the probability of x_4 is computed by marginalizing out the variables x_6, and x_7:

$$\begin{aligned} p(x_4 = 0) &= p(x_4 = 0, x_6 = 0, x_7 = 0) + p(x_4 = 0, x_6 = 1, x_7 = 0) \\ &\quad + p(x_4 = 0, x_6 = 0, x_7 = 1) + p(x_4 = 0, x_6 = 1, x_7 = 1) \\ p(x_4 = 1) &= p(x_4 = 1, x_6 = 0, x_7 = 0) + p(x_4 = 1, x_6 = 1, x_7 = 0) \\ &\quad + p(x_4 = 1, x_6 = 0, x_7 = 1) + p(x_4 = 1, x_6 = 1, x_7 = 1) \end{aligned}$$

If $p(x_4 = 0) > p(x_4 = 1)$ then the decoded value of x_4 is 0, otherwise 1. Therefore, we conclude that obtaining ML values for all three bits at the node $x_4 x_6 x_7$, requires 16 multiplications and 18 additions. Using similar approach, with 8 multiplications and 18 additions we can find the ML values of the bits x_2, x_3 and x_5, at the node $x_2 x_3 x_5$.

In total, decoding the Hamming (7,4) code on the graph in Figure 4.5 requires 142 operations. This number includes the 24 operations necessary to find joint conditional probabilities $p(y_2, y_3, y_5|x_2, x_3, x_5)$ and $p(y_4, y_6, y_7|x_4, x_6, x_7)$, 58 operations required to update the nodes in the graph and 60 operations necessary to find the marginal of single bits. As it has been noted above, in average 1690 operations is required to decode a codeword on the original graph. Hence, by transforming the graph we were able to achieve more than 10 fold reduction in the count of operations necessary to decode the code.

We also wish to include a remark of one of the reviewers of this thesis. For such a short code as Hamming(7,4) code the direct ML decoding may have even less complexity than decoding using the sum-product algorithm on the transformed graph. However, for longer codes direct approach to ML decoding will be unpractical while decoding using sum-product algorithm may have reasonable complexity which can be reduced using graph transformations.

We performed Monte-Carlo simulations of Bit Error Rate (BER) and Word Error Rate (WER) performance of the sum-product (SP) and Maximum Likelihood (ML) decoders of the Hamming (7,4) code for binary AWGN channel. The results of the simulations in the linear domain are presented in Figure 4.7. The sum-product decoding on the factor graph with cycles and on the cycle-free graph was implemented as described above. The data for BER and WER curves in the case of decoding on the factor graph with cycles are provided by Sina Tolouei [45].

The decoder in the case of the ML word decoding functions as follows:

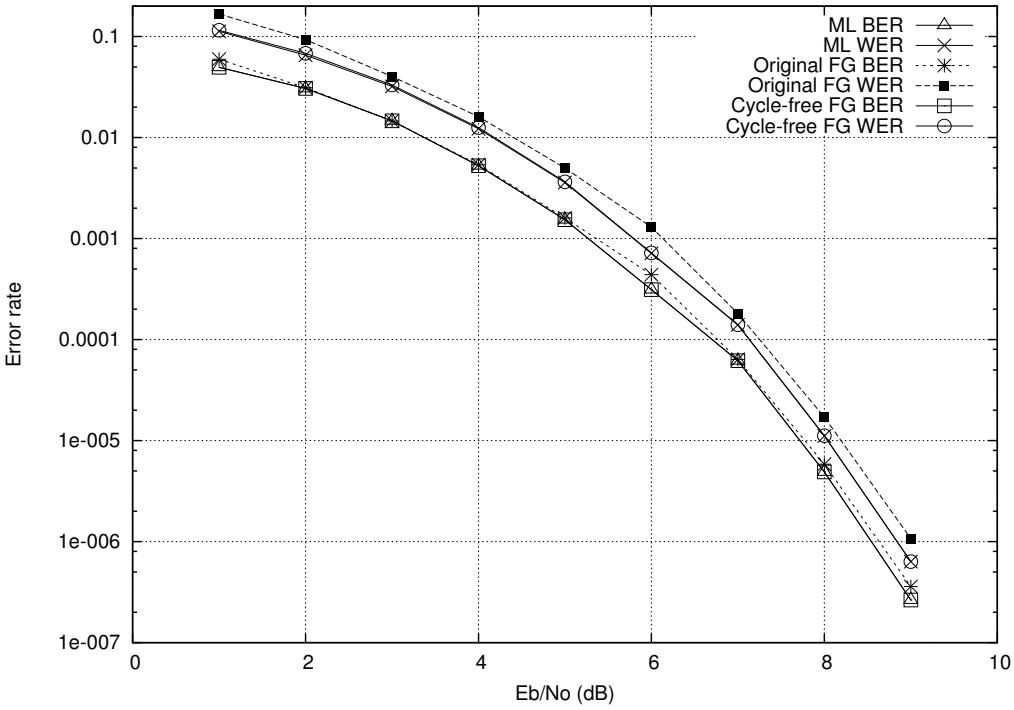


Figure 4.6: Bit Error Rate (BER) and Word Error Rate (WER) of the Hamming (7,4) code in the case of the original graph with cycles and in the case of the transformed graph.

1. Compute conditional probabilities $p(y_i|x_i)$ of each of the 7 bits x_i using the expression (4.6).

2. Compute the probability of each of the 16 codewords using the expression (4.7).

3. Select the codeword with the maximal probability for the decoded codeword.

4. Compare the selected codeword with the transmitted codeword and count a

word error if the codewords are not the same.

The decoder in the case of ML bit decoding operates as follows:

1. Compute the conditional probabilities $p(y_i|x_i)$ using the expression (4.6).

2. Compute the probability of each of the 16 codewords using the expression (4.7).

3. For each of the bits, compute the probability of a bit i equal to 0 as the sum of the probabilities of the codewords that have 0 in position i. Likewise, compute the probability of a bit i equal to 1 as the sum of the probabilities of the codewords which have 1 in position i.

4. Selected the value of bit i with the higher probability as the decoded bit.

5. Compare the decoded bit values to the transmitted bit and count a bit error if the bits are not the same.

The values of the messages and the probabilities were represented with high precision (C++ type double). In the simulations, the energy per bit Eb was scaled by $4/7$ compared to the uncoded case so that the energy per block remains unchanged. The Eb/No in the graph represent SNR in the case of transmission without coding. 500 bit errors were counted for each SNR values.

The reader can see that the decoding on the cycle-free graph corresponds to the ML decoding. In fact, the curves for the ML decoding and decoding on the cycle free-factor graph overlap. This result is expected since it is known that

the sum-product algorithm on a cycle-free graph is guaranteed to be exact. The sum-product decoder on the graph with cycles has slightly worse performance compared to the performance of the decoder on the cycle- free graph. This effect is more apparent on the WER curves.

Now we are going to consider the implementations of the sum-product algorithm on the transformed graphs in Figure 4.5 with messages represented in LLR domain. On the transformed graph the messages to and from the nodes $x_3\,x_5\,x_7$ and $x_2\,x_4\,x_6$ have 8 values and the problem is similar to the problem encountered in the case of decoding non-binary codes. Below, we consider the approach proposed by Wymeersch et al. [43]. This approach has been applied to the log-domain decoding of LDPC codes defined over Galois fields. The approach utilizes the Jacobi logarithm:

$$\max^* \triangleq \ln(e^{x_1} + e^{x_2}) \tag{4.18}$$

The function $\max^*$ can be expressed as:

$$\max^*(x_1, x_2) = \max(x_1, x_2) + \ln(1 + e^{-|x_1 - x_2|}) \tag{4.19}$$

The operation $\max^*(x_1, x_2)$ can be computed using maximization corrected by the $\ln(1 + e^{-|x_1 - x_2|})$ term. The correction term can be represented "without any performance loss" [43] [10] as a small look-up table. Therefore, the evaluation of (4.19) requires one comparison, one addition and one table lookup. The logarithm

[10] This statement applies to LDPC codes over GF(q) and has not been verified for our case.

of the sum of more than two exponential functions can be computed recursively, i.e., $\max^*(x_1,x_2,x_3) = \max^*(\max^*(x_1,x_2),x_3)$.

We define a 7-valued log-likelihood vector $L_{\{2,4,6\}}$ as the fraction of the probabilities of $p(x_4,x_6,x_7)$ for the values of the variables $x_4,x_6,x_7 = \{000,100,010,110,001,101,011\}$ over the probability $p(x_4=1,x_6=1,x_7=1)$:

$$\begin{aligned} L_{\{4,6,7\}}(000) &\triangleq \ln\left(\frac{p(x_4=0,x_6=0,x_7=0)}{p(x_4=1,x_6=1,x_7=1)}\right) \\ L_{\{4,6,7\}}(100) &\triangleq \ln\left(\frac{p(x_4=1,x_6=0,x_7=0)}{p(x_4=1,x_6=1,x_7=1)}\right) \\ &\vdots \\ L_{\{4,6,7\}}(011) &\triangleq \ln\left(\frac{p(x_4=0,x_6=1,x_7=1)}{p(x_4=1,x_6=1,x_7=1)}\right) \end{aligned} \tag{4.20}$$

Similarly, we define a 7-valued log-likelihood vector $L_{\{2,3,5\}}$ as the fraction of the probabilities of $p(x_2,x_3,x_5)$ over $p(x_2=1,x_3=1,x_5=1)$:

$$\begin{aligned} L_{\{2,3,5\}}(000) &\triangleq \ln\left(\frac{p(x_2=0,x_3=0,x_5=0)}{p(x_2=1,x_3=1,x_5=1)}\right) \\ L_{\{2,3,5\}}(100) &\triangleq \ln\left(\frac{p(x_2=1,x_3=0,x_5=0)}{p(x_2=1,x_3=1,x_5=1)}\right) \\ &\vdots \\ L_{\{2,3,5\}}(011) &\triangleq \ln\left(\frac{p(x_2=0,x_3=1,x_5=1)}{p(x_2=1,x_3=1,x_5=1)}\right) \end{aligned} \tag{4.21}$$

Now we will define the update rules for the variable and function nodes. The message sent by the node p_1 to the node x_1 defined as $L_1 \triangleq \ln\left(\frac{p(y_1|x_1=0)}{p(y_1|x_1=1)}\right)$. The node x_1 again just passes the message "through" and does not require any operations.

Recall that C_1 defines a valid value of x_1 which corresponds to the combination of the variables x_4, x_6, and x_7. For example, if $x_4 = 0, x_6 = 0, x_7 = 0$, then x_1 can only be 0, or if $x_4 = 1, x_6 = 0, x_7 = 0$, then x_1 is 1 and so on. For the LLR $L_{\{4,6,7\}}(000) = \ln\left(\frac{p(x_4=0,x_6=0,x_7=0)}{p(x_4=1,x_6=1,x_7=1)}\right)$ the probability in the numerator corresponds to $x_1 = 0$ and the probability in the denominator corresponds to $x_1 = 1$. Hence the message $\mu_{C_1 \to x_4 x_6 x_7}$ in LLR domain is expressed as:

$$\mu_{C_1 \to x_4 x_6 x_7} = \ln\left(\frac{p(x_1 = 0)}{p(x_1 = 1)}\right) = \mu x_1 \to C_1.$$

In the case of the $L_{\{4,6,7\}}(100) = \ln\left(\frac{p(x_4=1,x_6=0,x_7=0)}{p(x_4=1,x_6=1,x_7=1)}\right)$ the probabilities in both the numerator and denominator corresponds to $x_1 = 1$ and $L_{\{4,6,7\}}(100) = \ln\left(\frac{p(x_1=1)}{p(x_1=1)}\right) = 0$. We may conclude that the values of the message sent by C_1 towards $x_4 x_6 x_7$ are either $\mu x_1 \to C_1$ or 0 depending on whether the configuration of the variables in the numerator of $L_{\{4,6,7\}}$ corresponds to $x = 1$ or $x_1 = 0$. The vector of the values of the message sent from C_1 to $x_4 x_6 x_7$ is expressed as:

$$\mu_{C_1 \to x_4 x_6 x_7} = \{\overset{000}{\mu x_1 \to C_1}, \overset{001}{0}, \overset{010}{0}, \overset{011}{\mu x_1 \to C_1}, \overset{100}{0}, \overset{101}{\mu x_1 \to C_1}, \overset{110}{\mu x_1 \to C_1}\} \quad (4.22)$$

where we have denoted the configuration of the variables $x_4 x_6 x_7$ by the upper index.

For the equations that follow we need to shorten our notations and from this point onwards, instead of $p(x_4 = 0, x_6 = 0, x_7 = 0)$ we write $p_{\{4,6,7\}}(000)$, i.e., we moved the variables indices in the subscript and the values of the variables

represented in the form of a vector in the brackets.

The message from C_1 to the node x_1 can be expressed as:

$$\mu_{C_1 \to x_1} = \ln \left(\frac{p_{\{4,6,7\}}(000) + p_{\{4,6,7\}}(011) + p_{\{4,6,7\}}(101) + p_{\{4,6,7\}}(110)}{p_{\{4,6,7\}}(001) + p_{\{4,6,7\}}(010) + p_{\{4,6,7\}}(100) + p_{\{4,6,6\}}(111)} \right) \tag{4.23}$$

$$= \ln(e^{L_{\{4,6,7\}}(000)} + e^{L_{\{4,6,7\}}(011)} + e^{L_{\{4,6,7\}}(101)} + e^{L_{\{4,6,7\}}(110)})$$

$$- \ln(e^{L_{\{4,6,7\}}(001)} + e^{L_{\{4,6,7\}}(010)} + e^{L_{\{4,6,7\}}(100)} + e^{0}) \tag{4.24}$$

where in the numerator of (4.23) we have the sum of the pmfs $p(x_4, x_6, x_7)$ corresponding to the configurations of the variables $x_4 x_6 x_7$ such that C_1 is satisfied with $x_1 = 0$. The denominator of (4.24) has the sum of probabilities of the configurations that satisfy the parity check with $x_1 = 1$. We obtained (4.24) from (4.23) by dividing both the numerator and the denominator in (4.23) by $p_{\{4,6,7\}}(111)$ and using the fact that $e^{L(x_4=i,x_6=j,x_7=k)} \triangleq \frac{p(x_4=i,x_6=j,x_7=k)}{p(x_4=1,x_6=1,x_7=1)}$.

The evaluation of the equation (4.24) requires 6 computations of the terms in the form $\ln(e^{x_1} + e^{x_2})$.[11] Using the approach of Wymeersch et al. [43] this can be done with 6 comparisons, 6 additions and 6 table lookups. Besides the evaluation of the ln terms the expression (4.24) also requires one subtraction.

The message sent from the node $x_4 x_6 x_7$ to the node $C_2 C_3$ is sum of the message $\mu_{C_1 \to x_4 x_6 x_7}$ and the conditional probabilities in LLR domain. The message is

[11] Recall that the term $\ln(e^{x_1} + e^{x_2} + e^{x_3})$ can be computed recursively, so that $\ln(e^{x_1} + e^{x_2} + e^{x_3}) = \ln(e^{x_1} + \ln(e^{x_2} + e^{x_3}))$.

computed as follows:

$$\begin{aligned}
\mu_{x_4x_6x_7\to C_2C_3}(000) &= \mu_{C_1\to x_4x_6x_7}(000) + \ln\left(\tfrac{p(y_4,y_6,y_7|x_4=0,x_6=0,y_7=0)}{p(y_4,y_6,y_7|x_4=1,x_6=1,y_7=1)}\right)\\
\mu_{x_4x_6x_7\to C_2C_3}(100) &= \mu_{C_1\to x_4x_6x_7}(100) + \ln\left(\tfrac{p(y_4,y_6,y_7|x_4=1,x_6=0,y_7=0)}{p(y_4,y_6,y_7|x_4=1,x_6=1,y_7=1)}\right)\\
&\vdots\\
\mu_{x_4x_6x_7\to C_2C_3}(011) &= \mu_{C_1\to x_4x_6x_7}(011) + \ln\left(\tfrac{p(y_4,y_6,y_7|x_4=1,x_6=1,y_7=1)}{p(y_4,y_6,y_7|x_4=0,x_6=1,y_7=1)}\right)
\end{aligned} \tag{4.25}$$

The message in other direction (from $x_4x_6x_7$ to the node C_1) is computed in the same way i.e., the message is sum of the message $\mu_{C_2C_3\to x_4x_6x_7}$ and the conditional probabilities in LLR domain. Therefore, the update of the node $x_4x_6x_7$ requires 14 additions.

The joint LLR probabilities are computed as:

$$\begin{aligned}
&\ln\left(\frac{p(y_4,y_6,y_7|x_4=0,x_6=0,y_7=0)}{p(y_4,y_6,y_7|x_4=1,x_6=1,y_7=1)}\right) = \ln\left(\frac{p(y_4|x_4=0)}{p(y_4|x_4=1)}\right)\\
&+\ln\left(\frac{p(y_6|x_6=0)}{p(y_6|x_6=1)}\right)+\ln\left(\frac{p(x_7|x_7=0)}{p(y_7|x_7=1)}\right) = L_4+L_6+L_7
\end{aligned} \tag{4.26}$$

$$\begin{aligned}
&\ln\left(\frac{p(y_4,y_6,y_7|x_4=1,x_6=0,y_7=0)}{p(y_4,y_6,y_7|x_4=1,x_6=1,y_7=1)}\right) = \ln\left(\frac{p(y_4|x_4=1)}{p(y_4|x_4=1)}\right)\\
&+\ln\left(\frac{p(y_6|x_6=0)}{p(y_6|x_6=0)}\right)+\ln\left(\frac{p(x_7|x_7=0)}{p(y_7|x_7=1)}\right) = L_6+L_7
\end{aligned} \tag{4.27}$$

$$\vdots$$

$$\begin{aligned}
&\ln\left(\frac{p(y_4,y_6,y_7|x_4=0,x_6=1,y_7=1)}{p(y_4,y_6,y_7|x_4=1,x_6=1,y_7=1)}\right) = \ln\left(\frac{p(y_4|x_4=0)}{p(y_4|x_4=1)}\right)\\
&+\ln\left(\frac{p(y_6|x_6=1)}{p(y_6|x_6=1)}\right)+\ln\left(\frac{p(x_7|x_7=1)}{p(y_7|x_7=1)}\right) = L_4
\end{aligned} \tag{4.28}$$

where L_4, L_6 L_7 are the conditional probabilities of x_4, x_6 and x_7 in LLR domain

and we considered that the terms which have probability of $x = 1$ in the numerator are 0, e.g. $\ln\left(\frac{p(y_4|x_4=1)}{p(y_4|x_4=1)}\right) = 0$. The evaluation of the joint LLR requires 6 additions.

The parity checks C_2 and C_3 define the values of the variables x_4, x_6 and x_7 which are valid for the given values of the variables x_2, x_3 and x_5. For example, if $x_2 = 0$, $x_3 = 0$ and $x_5 = 0$ then two configurations $x_4 = 0$, $x_6 = 0$, $x_7 = 0$ and $x_4 = 1$, $x_6 = 1$, $x_7 = 1$ satisfy the parity checks. Similarly, if $x_2 = 1$, $x_3 = 0$ and $x_5 = 0$ then, in order to satisfy the parity checks, the values of x_4, x_6 and x_7 have to be either 100 or 011. Hence, if we only consider the probabilities of x_4, x_6 and x_7 then the likelihood of the configuration $x_2 = 0$, $x_3 = 0$ and $x_5 = 0$ is: $p_{\{2,3,5\}}(000) = p_{\{4,6,7\}}(000) + p_{\{4,6,7\}}(111)$.

The LLR value sent from the node C_2C_3 to the node $x_2x_3x_5$ for the values of the variables 100 is:[12]

$$\mu_{C_1C_2 \to x_2x_3x_5}(100) = \ln\left(\frac{p_{\{2,3,5\}}(100)}{p_{\{2,3,5\}}(111)}\right) = \ln\left(\frac{p_{\{4,6,7\}}(100) + p_{\{4,6,7\}}(011)}{p_{\{4,6,7\}}(000) + p_{\{4,6,7\}}(111)}\right) =$$
$$\ln\left(e^{\mu_{x_4x_6x_7 \to C_2C_3}(100)} + e^{\mu_{x_4x_6x_7 \to C_2C_3}(011)}\right) - \ln\left(e^{\mu_{x_4x_6x_7 \to C_2C_3}(000)} + e^{0}\right) \qquad (4.29)$$

The expression can be evaluated using the approach described above with 2 comparisons, 2 additions, 2 table lookups and one subtraction. The LLR messages for

[12] Please note that by definition the LLR corresponding to the values 111 is 0.

the other values of x_2, x_3 and x_5 can be evaluated in a similar way:

$$\mu_{C_1C_2\to x_2x_3x_5}(000) = \ln\left(\frac{p_{\{2,3,5\}}(000)}{p_{\{2,3,5\}}(111)}\right) = \ln\left(\frac{p_{\{4,6,7\}}(000)+p_{\{4,6,7\}}(111)}{p_{\{4,6,7\}}(000)+p_{\{4,6,7\}}(111)}\right) = 0$$

$$\mu_{C_1C_2\to x_2x_3x_5}(010) = \ln\left(\frac{p_{\{4,6,7\}}(110)+p_{\{4,6,7\}}(001)}{p_{\{4,6,7\}}(000)+p_{\{4,6,7\}}(111)}\right) =$$

$$\ln\left(e^{\mu_{x_4x_6x_7\to C_2C_3}(110)}+e^{\mu_{x_4x_6x_7\to C_2C_3}(001)}\right) - \ln\left(e^{\mu_{x_4x_6x_7\to C_2C_3}(000)}+e^{0}\right)$$

$$\mu_{C_1C_2\to x_2x_3x_5}(110) = \ln\left(\frac{p_{\{4,6,7\}}(101)+p_{\{4,6,7\}}(010)}{p_{\{4,6,7\}}(000)+p_{\{4,6,7\}}(111)}\right) =$$

$$\ln\left(e^{\mu_{x_4x_6x_7\to C_2C_3}(101)}+e^{\mu_{x_4x_6x_7\to C_2C_3}(010)}\right) - \ln\left(e^{\mu_{x_4x_6x_7\to C_2C_3}(000)}+e^{0}\right)$$

$$\mu_{C_1C_2\to x_2x_3x_5}(001) = \ln\left(\frac{p_{\{4,6,7\}}(101)+p_{\{4,6,7\}}(010)}{p_{\{4,6,7\}}(000)+p_{\{4,6,7\}}(111)}\right) = \mu_{C_1C_2\to x_2x_3x_5}(110)$$

$$\mu_{C_1C_2\to x_2x_3x_5}(101) = \ln\left(\frac{p_{\{4,6,7\}}(110)+p_{\{4,6,7\}}(001)}{p_{\{4,6,7\}}(000)+p_{\{4,6,7\}}(111)}\right) = \mu_{C_1C_2\to x_2x_3x_5}(010)$$

$$\mu_{C_1C_2\to x_2x_3x_5}(011) = \ln\left(\frac{p_{\{4,6,7\}}(100)+p_{\{4,6,7\}}(011)}{p_{\{4,6,7\}}(000)+p_{\{4,6,7\}}(111)}\right) = \mu_{C_1C_2\to x_2x_3x_5}(100) \quad (4.30)$$

Note that the message values for the vectors 001, 101 and 011 are equal to the message values for the vectors 011, 010 and 100, respectively. Also the denominator is the same for all of the messages and can be evaluated once. The message $\mu_{C_1C_2\to x_4x_6x_7}$ is computed in a similar way. The update of the node C_2C_3 requires 8 comparisons, 8 additions, 8 table lookups and 6 subtractions.

The summary of the number of operations necessary to update the nodes in LLR domain on original and transformed graphs is presented in Table 4.3. In total, a single iteration of the sum-product algorithm on the transformed graph in Figure 4.5 requires 63 operations: 28 additions, 7 subtractions, 14 table lookups

and 14 comparisons. On the graph prior to the transformations only 26 operations are required for a single iteration of the sum-product algorithm. However, direct comparison of the total number of operations may not give a fair comparison of the complexity of the algorithms since different operations have different latencies and memory requirements. For example, additions and subtractions may require less resources (this depends on the hardware used) as compared to the table lookups. The numbers of the table lookups are approximately the same for the original and transformed graphs.

On the transformed graph, additional complexity is associated with computing the marginals over single bits which are required for ML decision on the bits values. As usual, the probabilities at the variable nodes are the products (or in LLR domain sum) of all incoming messages. For example, at the node $x_2\,x_3\,x_5$ the probability is the sum of the message $\mu_{C_1\,C_2\to x_2\,x_3\,x_5}$ and the message $\mu_{p_2\,p_3\,p_5\to x_2\,x_3\,x_5}$. The former term is the LLR of the conditional probability $p(y_2,y_3,y_5|x_2,x_3,x_5)$. Evaluation of the sum requires 7 additions. In order to find the LLR value of a single bit (which is necessary to decide whether the bit 0 or 1) we may marginalize out the other variables. For example, for LLR of bit x_2 we have:

$$L_{x_2} = \ln\left(\frac{p_{\{2,3,5\}}(000)+p_{\{2,3,5\}}(001)+p_{\{2,3,5\}}(010)+p_{\{2,3,5\}}(011)}{p_{\{2,3,5\}}(100)+p_{\{2,3,5\}}(101)+p_{\{2,3,5\}}(110)+p_{\{3,5,7\}}(111)}\right) =$$
$$\ln\left(e^{L_{\{2,3,5\}}(000)}+e^{L_{\{2,3,5\}}(001)}+e^{L_{\{2,3,5\}}(010)}+e^{L_{\{2,3,5\}}(011)}\right)$$
$$-\ln\left(e^{L_{\{2,4,6\}}(100)}+e^{L_{\{2,4,6\}}(101)}+e^{L_{\{2,4,6\}}(110)}+e^{0}\right)$$

Hence, 6 comparisons, 6 additions and 6 table lookups and 1 subtraction are re-

Table 4.3: Number of operations required to update the graphs of the Hamming(7,4) code in the original and the cycle-free forms with decoding performed in LLR domain.

Factor graph in the original form in Figure 4.4

Nodes	Number of nodes	Number of operations per iteration				
		Additions		2D lookups		Total
		Per node	Total	Per node	Total	
x_1, x_2 and x_3	3	0	0	0	0	0
x_7, x_4 and x_5	3	2	6	0	0	6
x_6	1	5	5	0	0	5
C_1, C_2 and C_3	3	0	0	5	15	15
Total for nodes	10	11		15		**26**

Considering 10 iterations, the total number of operations necessary to decode a codeword is 260. Assumptions:
1) The updates of the nodes C_1, C_2 and C_3 are defined by expression (4.15) and performed as search in $2D$ lookup table [20].
2) The messages towards the nodes x_1, x_2, x_3 and $p_1, p_2, \ldots, p_7$ are not updated during iterations.

Factor graph in the transformed form in Figure 4.5

Nodes	Number of nodes	Number of operations iteration								
		Additions		Subtractions		Comparisons		Lookups		Total
		Per node	Total	Per node	Total	Per node	Total	Per node	Total	
x_1	1	0	0	0	0	0	0	0	0	0
C_1	1	6	6	1	1	6	6	6	6	19
$x_4 x_6 x_7$	1	14	14	0	0	0	0	0	0	14
$C_2 C_3$	1	8	8	6	6	8	8	8	8	30
$x_2 x_3 x_5$	1	0	0	0	0	0	0	0	0	0
Total for nodes	5	28		7		14		14		**63**

Additional operations which are necessary to decode a codeword on the transformed graph:
1) Finding joint probabilities $p_4\, p_6\, p_7$ and $p_2\, p_3\, p_5$ requires 12 additions. 2) Finding marginal bit probabilities for the bits $x_2, \ldots, x_7$ at the nodes $x_2 x_3 x_5$ and $x_4 x_6 x_7$ requires 114 operations in total: 36 additions, 6 subtractions 36 comparisons and 36 lookups.
The total number of operations necessary to decode a codeword on the transformed graph is 189.

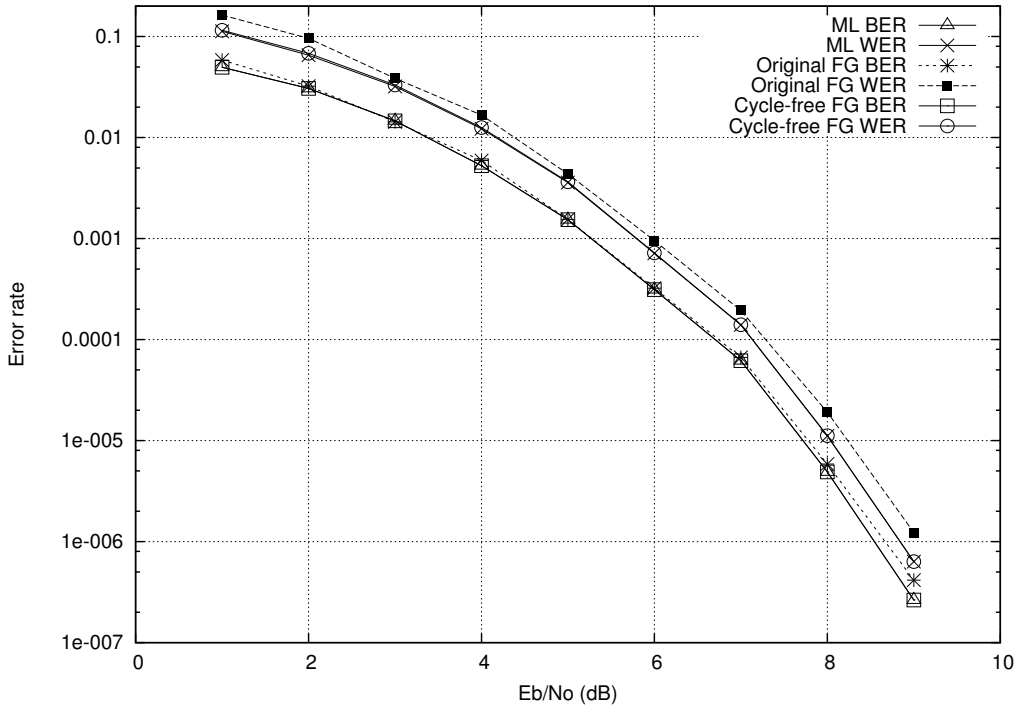


Figure 4.7: Performance of ML and SP decoder implemented in LLR domain on the original factor graph with cycles in Figure 4.4 and on the transformed cycle-free graph in Figure 4.5.

quired for taking the decision on values of each of the variables $x_2, \ldots, x_7$.

The results of Monte-Carlo simulations of Bit Error Rate (BER) and Word Error Rate (WER) of the decoders of the Hamming code (7,4) implemented in LLR domain is represented in Figure 4.7. The decoders in LLR domain on the cycle-free graph was implemented in the method described above with exception that we evaluated the tangent hyperbolic functions in expressions (4.15) as well as exponential and logarithmic functions in the expressions (4.19-4.29) exactly, using corresponding functions of C++. In other words, the approximations of the

functions were not implemented. The simulation data for the decoder on the graph with cycles is provided by Sina Tolouei [45]. Similar to the case of decoding in the probability domain, the performance of the SP decoder on the cycle- free graph corresponds to the performance of the ML decoder. The SP decoder on the graph with cycles has slightly worse performance compared to the performance of the decoder on the cycle-free graph.

Most practical codes are much longer than the Hamming (7,4) code and the H matrices and factor graphs of the codes are therefore, much larger. It might not be possible to convert the graphs to a cycle-free form while maintaining a reasonable computational complexity. We may however suggest that if a factor graph of a code has a sub-graph similar to the graph of the Hamming (7,4) code, then the sub-graph can be transformed as described above and this may lower the overall complexity of the belief propagation decoding algorithm.

4.3 A Factor Graph Approach to Link Loss Monitoring in Wireless Sensor Networks

In this section we apply factor graph transformations and lower the complexity of an application of factor graphs in monitoring losses in wireless sensor networks [22]. Wireless sensor networks are comprised of a large number of sensors that take measurements from the surrounding environment. A sensor is an autonomous device which normally operates under strict power and computational complexity constraints. In order to optimize power efficiency, sensors usually use relaying to transmit information to the destination, which is usually a single data collect-

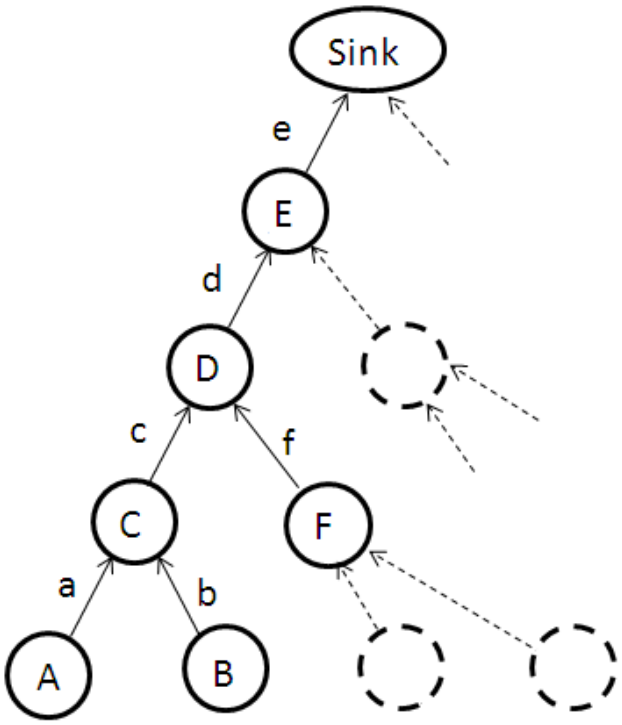


Figure 4.8: An example of communications in a sensor network. The capital letters denote sensors, the lower case letters denote links. The nodes A and B use the nodes C, D and E as relays.

ing node in the network. Sensors are prone to failure and various impediments of wireless media so that losses of information often occur. Knowledge of link loss rates is essential for developing "good" routing protocols for wireless sensor networks.

A factor graph approach to inferring and monitoring link loss rates has been proposed by Mao et al. [22]. The method allows continuous monitoring of link losses with minimal complexity. The authors utilized a model of a sensor network with data aggregation and relaying. In this model a node receives informa-

tion (packets) from other nodes, incorporates the information into a single packet which is then sent further through the network. The network can be represented by a reverse multicast tree. A node in the network has several branches and one connection to the "trunk" which is the next sensor in the chosen routing path to the data collection node or sink. The sink receives packets from all sensors-branches. Loss of a packet is inferred if the packet does not arrive from a branch in an allocated period of time. The task of network tomography [22] is to infer the link loss rates from the statistic of received and lost packets. Figure 4.8 represents a model of a sensor network under consideration. The nodes A and B use the nodes C, D and E as relays. A packet sent by the node C is an aggregation of the packets from nodes A, B and C. The authors of [22] used a rather abstract notion of a "packet" and do not focus of the physical layer of the transmission. It is assumed that the nodes have sufficient bandwidth to relay the information from its child nodes.

In the text below, we closely follow the notation of [22]. By e we denote an edge or link (such as $a, b, c, \ldots$ in Figure 4.8) and by w we denote an path in the network (such as $A \rightarrow Sink$ or $B \rightarrow Sink$ in Figure 4.8). The authors of [22] use capital letters to denote sets. E denotes the set of all edges and W the set of all paths from the sensors to the sink in a sensor network.

The state of a link e is represented by a Bernoulli random variable x_e [34] which takes value 1 (good state) and value 0 (bad state) with probability α and $1-\alpha$, respectively. The distribution of the state of a link e is denoted by $B(x_e, \alpha_e)$. The state of a path w which is comprised from several links is denoted as x_w. $x_e^{(i)}$ and $x_w^{(i)}$ denote the states of a link and a path during the transmission of packet i,

respectively. If any of the links in a path are in a "bad" state then the state of the path is considered as "bad" which is expressed as:

$$x_w^{(i)} = \underset{e \in w}{\otimes} x_e^{(i)} \tag{4.31}$$

where $\otimes$, denotes binary AND operation. The sets X_E^i and X_W^i denote the states of all edges and all paths in the network in the time of transmission of packet i.[13] For example, in the network in Figure 4.8 $X_E^{(i)} = \{x_a^{(i)}, x_b^{(i)}, x_c^{(i)}, \dots\}$ and $X_W^{(i)} = \{x_{A\to Sink}^{(i)}, x_{B\to Sink}^{(i)}, \dots\}$.

Following [22] let $C(x)$ be a Boolean proposition of some variable or vector x, then an indicator function δ is defined as:

$$\delta\left[C(x)\right] = \begin{cases} 1, \text{ if } C(x) \text{ holds} \\ 0, \text{otherwise} \end{cases} \tag{4.32}$$

The probability mass function of path states given link state at the time of transmission of a packet is i is a deterministic function, which is just an extension of (4.31) for the all paths $w \in W$:

$$P_{W|E}(X_W^{(i)}|X_E^{(i)}) = \prod_{w \in W} \delta\left[x_w^{(i)} = \underset{e \in w}{\otimes} x_e^{(i)}\right] \tag{4.33}$$

The set α_E denotes loss rates for all links in the network, e.g. $\alpha_E = \{\alpha_a, \alpha_b, \alpha_c, \dots\}$

[13] Perhaps in this context i can be viewed as a period where all sensors transmit a single packet with number i plus the time sufficient for the packets to propagate trough network and reach the sink.

in Figure 4.8. Assuming that the losses are independent on the links, the joint PMF of all links in the network $B_E(X_E, \alpha_E)$ is expressed as:

$$B_E(X_E, \alpha_E) = \prod_{e \in E} B(x_e, \alpha_e) \tag{4.34}$$

The sets of the link and path losses over transmissions of n packets $1, 2, \ldots, n$ is denoted by $X_E^{(1,n)}$ and $X_W^{(1,n)}$, respectively.

Now, using introduced notation we reiterate our objective. Given the set of observations of path states $X_W^{(1,n)}$ over transmission of the packets $1, 2, \ldots, n$ we aim to infer the link loss rates α_E.

Assuming that α_E stays constant during transmission of the packets $1, 2, \ldots, n$, then the joint PMF of α_E, $X_W^{(1,n)}$, $X_E^{(1,n)}$ during transmissions $1, 2, \ldots, n$ is expressed as [22, eq. 2]:

$$P\left[\alpha_E, X_E^{(1,n)}, X_W^{(1,n)}\right] \propto \prod_{i=1}^{n} B_E(X_E^{(i)}, \alpha_E) P_{W|E}(X_W^{(i)} | X_E^{(i)}) \tag{4.35}$$

and the objective is for every $e \in E$ find $\hat{\alpha}_e$ which maximizes:

$$\begin{aligned} P\left[\alpha_e | X_W^{(1,n)}\right] &\propto \sum_{\sim \alpha_e} P\left[\alpha_E, X_E^{(1,n)}, X_W^{(1,n)}\right] \\ &\propto \sum_{\sim \alpha_e} \prod_{i=1}^{n} B_E(X_E^{(i)}, \alpha_E) P_{W|E}(X_W^{(i)} | X_E^{(i)}) \end{aligned}$$

where $X_W^{(1,n)}$ and $X_W^{(i)}$ (observations of path states) are given. As one can see it is an instance MPF problem which can be solved by the sum-product algorithm.The

global function is:

$$G(\alpha_E, X_E) = \prod_{i=1}^{n} B_E(X_E^{(i)}, \alpha_E) P_{W|E}(X_W^{(i)} | X_E^{(i)})$$
$$= \prod_{i=1}^{n} B_E(X_E^{(i)}, \alpha_E) \prod_{w \in W} \delta \left[x_w^{(i)} = \underset{e \in w}{\oplus} x_e^{(i)} \right] \tag{4.36}$$

For the transmission of a single packet i in the network in Figure 4.8 the global function is represented as:

$$G^{(i)}(\alpha_E^{(i)}, X_E^{(i)}) = B(x_a^{(i)}, \alpha_a^{(i)}) B(x_a^{(i)}, \alpha_a^{(i)}) \cdots B(x_f^{(i)}, \alpha_f^{(i)}) \times$$
$$\delta_A[x_{\{a,c,d,e\}}^{(i)} = x_a^{(i)} \otimes x_c^{(i)} \otimes x_d^{(i)} \otimes x_e^{(i)}] \times$$
$$\delta_B[x_{\{b,c,d,e\}}^{(i)} = x_b^{(i)} \otimes x_c^{(i)} \otimes x_d^{(i)} \otimes x_e^{(i)}] \times$$
$$\delta_F[x_{\{f,d,e\}}^{(i)} = x_f^{(i)} \otimes x_d^{(i)} \otimes x_e^{(i)}] \tag{4.37}$$

where we only considered the paths from the sensors A, B and F ($w_A = \{a,c,d,e\}$, $w_B = \{a,c,d,e\}$ and $w_F = \{f,d,e\}$). The factor graph corresponding to the global function 4.37 is presented in Figure 4.9.

The bottom row of the graph represents the observation of path states by the sink node, i.e., $x_w = 1$ if the sink received the packet from the path w or $x_w = 0$ otherwise. The variable nodes in the middle row correspond to the link states. The top row nodes are estimates of the parameters α (loss rates, or rather success rates) of Bernoulli distributions for each link.

The sum-product operates on the factor graph such as shown in Figure 4.9. Our goal is to acquire the marginals of α. The messages towards the vertices of

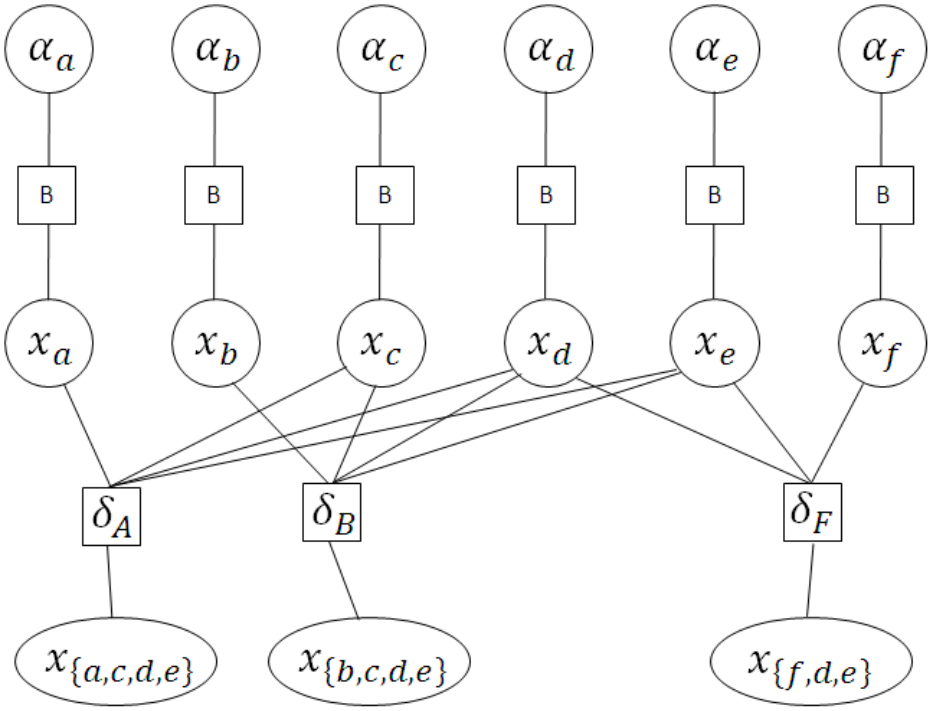


Figure 4.9: An example of a factor graph for the sensor network in Figure 4.8. The graph represents a single round of transmission.

path states such as $x_{\{f,d,e\}}$ do not need to be updated. The nodes α are leaf-nodes and therefore the messages towards the nodes need to be updated only once, upon the completion of the iterations. Following the notation of the authors of [22], from this point onward, by $\mu_{w\to e}$ we denote a message from a node δ_w to the node x_e. Likewise, by $\mu_{e\to w}$ we denote a message from the node x_e to the node δ_w. By $N(w)$ and $N(e)$ we denote the sets of the neighbors of the node δ_w and x_e, respectively. The message values on the edges connected to the link state node x_e are the probabilities of the link being in "good" and "bad" states. The messages may have a single value since they represent probabilities and have to sum to unity.

The value which is chosen as the message value in the algorithm is the probability that the link is in "good" state.

During the initialization stage of the algorithm messages from the nodes x_e to the nodes δ_w are set to 0.5 (which means that it is equally likely that the links are in good and bad states). During the propagation stage also referred as the iterative stage of the algorithm, the link nodes x_e and the path nodes δ_w exchange messages. A message from a path state node δ_w to a link state x_e node is expressed as [22, eq. (6)]:

$$\mu_{w\to e} = \begin{cases} 1, \text{ if } x_w^{(i)} = 1 \\ \dfrac{1 - \prod\limits_{e' \in N(w)\setminus\{e\}} \mu_{e'\to w}}{2 - \prod\limits_{e' \in N(w)\setminus\{e\}} \mu_{e'\to w}}, \text{ if } x_w^{(i)} = 0 \end{cases} \tag{4.38}$$

where $N(w)\setminus\{e\}$ denotes the set of neighbors of the node δ_w except for the node e and node B. The message is the belief of a node δ_w that the node x_e is in "good" state. The first condition of (4.38) represents simple logic: if a path is in "good" state then all links are in "good" state with probability 1.

The second condition is the probability that the transmission over link e is successful given that the path is failed $p($ link e is good$|$ path w is bad$)$ which using the Bayes rule is expressed as:

$$p(\text{ link e is good}|\text{ path w is bad}) = \frac{p(\text{ link e is good}|\text{ path w is bad})p(\text{link e is good})}{p(\text{path w is good})}$$

The expression is equal to the second clause of (4.38) given that a priori $p(\text{link e is good}) = 0.5$.

Note that if at least one of the messages from the edges, except for the edge e, is 0 then the messages to the node x_e is $\mu_{w\to e} = 0.5$, which means that this path cannot contribute more information about the state of x_e since it is known that some other links are failed. If the path state $x_w = 1$ (the path is "good") then the node does not require any operations.

The message from a variable node x_e to a function node δ_w is expressed as [22, eq. 7]:

$$\mu_{e\to w} = \frac{\prod\limits_{w'\in N(e)\setminus\{w\}} \mu_{w'\to e}}{\prod\limits_{w'\in N(e)\setminus\{w\}} \mu_{w'\to e} + \prod\limits_{w'\in N(e)\setminus\{w\}} (1-\mu_{w'\to e})} \tag{4.39}$$

where $N(e)\setminus\{w\}$ denotes a set of node neighbors of the node x_e except for the node δ_w. The numerator of 4.39 is the belief of all nodes $\delta \in N(e)\setminus\{w\}$ that the link x_e is in "good" state, and the denominator has sum of probabilities of the link being in "good" and in "bad" states, that ensures that the probability sums to unity.

Upon the completion of the iterations the link nodes x_e compute the probabilities that they are in "good" states which are the estimates of α_e for the transmission of the current packet [22, eq. 8]:.

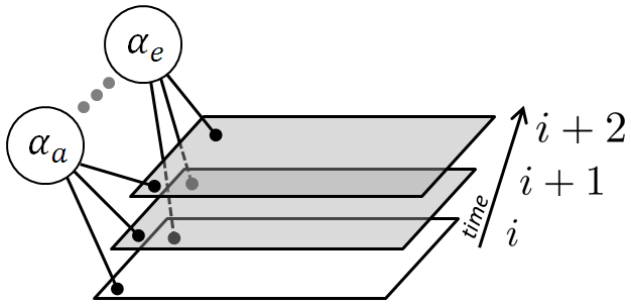


Figure 4.10: Representation of the algorithm that estimates α over multiple rounds of transmission. Each layer corresponds to the graph at a single time instant in Figure 4.9.

$$\mu_{e\to w} = \frac{\prod\limits_{w'\in N(e)} \mu_{w'\to e}}{\prod\limits_{w'\in N(e)} \mu_{w'\to e} + \prod\limits_{w'\in N(e)} (1-\mu_{w'\to e})} \tag{4.40}$$

The schematic representation of the graph for the estimation of $\hat{\alpha}_E$ over the time period where packets i, $i+1$ and $i+2$ are transmitted is presented in Figure 4.10. Three surfaces on the picture represent factor graphs for a single round of transmissions (i, $i+1$ and $i+2$), i.e., each of the surfaces is a factor graph such as the graph in Figure 4.9. On each of the surfaces the algorithm performs the estimation of $\hat{\alpha}_E^{(i)}$ (estimation α_E during the transmission i) as it has been described above. The layers are connected via the variable nodes α since it is assumed that α stays constant over the period of transmission of multiple packets. After the completion of estimation of $\hat{\alpha}_E^{(i)}$ the algorithm combines the current estimate with estimates from the past. By combining, we refer to multiplications of $\hat{\alpha}_E^{(i)} \cdot \hat{\alpha}_E^{(i-1)} \cdot \hat{\alpha}_E^{(i-2)} \ldots$ or perhaps, to multiplication of the weighted estimates $\hat{\alpha}_E^{(i)}$

which would allow the algorithm to add more value to the most current estimates and eventually "forget" the past estimates. This approach has low complexity since only the most recent observations need to be processed on the factor graph, i.e., only the graph on the top surface in Figure 4.9 is updated.

Now we will evaluate the complexity of the algorithm in terms of the operation counts and lower the complexity of the algorithm using transformations of the graph. First of all, we determine the complexity of the algorithm on the original graph. The products in the equations above can be computed using the method described in Section 3.3.1, so that the number of multiplications necessary for their computation is $3(d(v)-2)$. To simplify the estimation of the count of operations, we assume that the link losses are the same i.e., α for all links. While this assumption may not be realistic in practice, it may still provide an insight into the complexity of the algorithm.

The update of the path states nodes δ_w is expressed as (4.38). By $d(\delta_w)$ we denote the degree of the node δ_w. With the probability $\alpha^{(d(\delta_w)-1)}$, a path is in "good" state[14] and no operations are needed since the first condition in (4.38) is involved and all messages are equal to 1. With the probability $1-\alpha^{(d(\delta_w)-1)}$ the path is in "bad" state so that second second condition in (4.38) is involved.[15] Therefore, the evaluation of the second clause in the expression (4.38)) requires:

[14]We reiterate our assumption that link losses are independent. If the probability of success of Bernoulli trial (which in our case, is a transmission over a single link in a path) is p then the probability of success of n independent trials is p^n (which is in our case the probability that all links in a path are in "good" state). Then the probability that at least 1 of the trails (transmissions) fails is $1-p^n$.

[15]Note that 1 is subtracted from $d(\delta_w)$ since there is one edge towards the path node x_w and the messages on the edge do not need to be updated or considered.

- $3(d(\delta_w)-3)$ multiplications to compute the products of the incoming messages
- $2(\delta_w-1)$ subtractions to evaluate the numerator and denominator of the expression
- δ_w-1 divisions to evaluate the fraction

In total, when the second clause in (4.38) is involved, the update of the node δ_w requires $5d(\delta_w)-12$ operations. A node of degree less than 3 does not need any operations since is this case the state of a path determines the state of a link.

Now we will determine the number of operations necessary to update the link state nodes x_e as expressed by (4.39). The messages to the leaf nodes B need to be computed only once, upon completion of the estimations. The messages from the nodes B are a priori estimates of α_e which assumed to be 0.5 and therefore, can be canceled from the denominator in (4.39). The update of the link state nodes x_e requires:

- $d(x_e)-1$ subtractions, in order to evaluate the terms $1-\mu_{w\to e}$ in the second part of the denominator
- $6(d(x_e)-3)$ multiplications in order to evaluate the products of the incoming messages $\mu_{w\to e}$ as well as the products of $1-\mu_{w\to e}$ in the second part of the denominator
- $d(x_e)-1$ additions in order to evaluate the denominator
- $d(x_e)-1$ divisions in order to evaluate the fraction

The total number of operations, which we assume has to be done for every iteration, in $9d(x_e) - 21$. The node of degree 2 does not need any operations.

Due to the probabilistic nature of the update, we need to define average complexity as the average number of operations per iteration that the sink node has to perform over a period of time with a large number of transmitted packets. If with probability P_1 and P_2 a node has to perform X and Y operations, respectively then the average number of operations will be $XP_1 + YP_2$. In our case, if the link w is failed then the complexity consists of the complexity of the update the nodes δ_w and x_e. When the path is good we assume that only the nodes x_e are updated.

Now, as an example, we are going to consider the factor graph in Figure 4.9. The nodes δ_A, δ_B and δ_F in the graph have the degrees 5, 5 and 4, respectively. Taking into account the operation count for the nodes δ defined above, the nodes δ_A and δ_B each with probability $1 - \alpha^4$ need 8 operations and the node δ_F with probability $1 - \alpha^3$ needs 3 operations.[16] In total, the delta nodes require in average $19 - 16\alpha^4 - 3\alpha^3$ operations. The link states nodes x_a, x_b and x_f in Figure 4.9 have degree 2 and do not require any operations. The remaining 3 link state nodes x_c, x_d and x_e) have degrees 3, 4 and 4, respectively. Taking into account the number of operations for a link state node defined above, the variable nodes require 36 operations per iteration. In total, the average number of operations per an iteration in the graph in Figure 4.9 is:

$$C_O = 55 - 16\alpha^4 - 3\alpha^3 \tag{4.41}$$

[16] We assume that the links $A \rightarrow Sink$, $A \rightarrow Sink$, and $F \rightarrow Sink$ have failure probabilities $1 - \alpha^4$, $1 - \alpha^4$ and $1 - \alpha^3$, respectively.

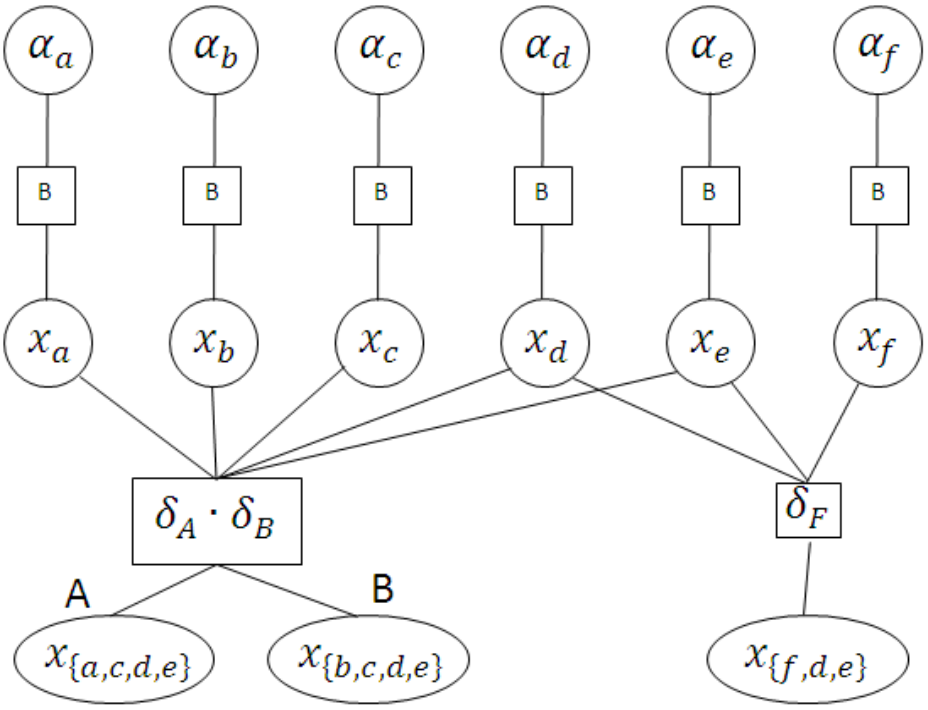


Figure 4.11: Factor graph of wireless sensor network with joined δ nodes.

As an example for the $\alpha = 0.95$, the number of operations is 39.4.

In this application, for the nodes of the same degree, the complexity of the update of the link nodes x_e is higher compared to the complexity of the update of δ_w nodes. Hence, we may attempt to reduce the degrees of variable nodes x_e by clustering the function nodes δ_w. Also we note that the function nodes δ_A and δ_B have 3 variable nodes in common and that they differ by a single variable only. By joining the nodes δ_A and δ_B we can reduce the degrees of the link nodes x_c, x_d and x_e as shown in Figure 4.11.

We observe that if the transmission $A \rightarrow Sink$ is successful and $B \rightarrow Sink$ is failed then we know that only the link b has failed. Similarly, if $A \rightarrow Sink$ is failed

but $B \rightarrow Sink$ is successful, then we know that only the link a has failed. These observations can be incorporated in the logic of clustered node $\delta_A \cdot \delta_B$ and the update rule for the node can be expressed as:

$$\mu_{w\rightarrow e} = \begin{cases} 1, \text{ if (A) AND (B)} \\ \mu_{w\rightarrow x_b} = 0, \mu_{w\rightarrow N(w)\setminus x_b} = 1, \text{if (A) AND (NOT (B))} \\ \mu_{w\rightarrow x_a} = 0, \mu_{w\rightarrow N(w)\setminus x_a} = 1, \text{if (NOT(A)) AND (B)} \\ \text{see the expressions (4.43) and (4.44) below, if (NOT (A)) AND (NOT (B))} \end{cases} \tag{4.42}$$

where A and B denote Boolean variables indicating whether the packet from corresponding node was received by the sink. The clauses in the expression (4.42) have the following meaning.

1. *(A) AND (B)*: both paths from A $\rightarrow$ Sink and from B $\rightarrow$ Sink are in "good" state and all links $\{a,b,c,d,e\}$ are in good state with probability 1. Therefore, 1 is sent to all link nodes x_a, x_b, x_c, x_d and x_e.

2. *(A) AND (NOT (B))*: the packet from A is received but the packet from B is lost. The link b is in "bad" state with probability 1 and the other links are in "good" state with probability 1. Therefore, the message sent to x_b is 0 and the messages sent to the nodes x_a, x_c, x_d and x_e are 1.

3. *(NOT(A)) AND (B)*: the packet from A is lost but the packet from B is received. The state of the link a is "bad" with probability 1 and the other links are in "good" state with probability 1. Therefore, the message sent to x_a is

0 and the messages sent to the nodes x_b, x_c, x_d and x_e are 1.

4. *(NOT(A)) AND NOT(B)*: the packets from both A and B are lost. This indicates that either both links a and b are failed or that one of the links $\{c,d,e\}$ is failed. The probability of this is:

$$P_{AB}^{f} = (1-\mu_{c\to AB}\mu_{d\to AB}\mu_{e\to AB}) + (1-\mu_{a\to AB})(1-\mu_{b\to AB}) \\ -(1-\mu_{c\to AB}\mu_{d\to AB}\mu_{e\to AB})(1-\mu_{a\to AB})(1-\mu_{b\to AB})$$

where the first term is the probability of a failure of any of the links c, d, and e, the second term is the probability of failure of both links a and b and the last term is the joint probability that links a and b and any of the links c, d, and e are failed at the same time. The probability that the transmission over the link a is successful given, that the transmission over the path failed (which is the message to the node x_a), can be evaluated as:

$$\mu_{AB\to a} = \frac{1-\mu_{c\to AB}\mu_{d\to AB}\mu_{e\to AB}}{2(1-\mu_{c\to AB}\mu_{d\to AB}\mu_{e\to AB})+1-\mu_{b\to AB}-(1-\mu_{c\to AB}\mu_{d\to AB}\mu_{e\to AB})(1-\mu_{b\to AB})} \tag{4.43}$$

where we used Bayes rule and considered a priori $P(\text{link a failed}) = 0.5$. The message to the node x_b is computed in similar fashion with the exception that $\mu_{b\to AB}$ in the equation (4.43) is replaced by $\mu_{a\to AB}$. The message

to the node x_c is evaluated as:

$$\mu_{AB\to c} = \frac{P_1}{2P_2} \tag{4.44}$$

where the term P_1 is the probability of failure of both paths A and B given that link c is in good state:

$$P_1 = (1-\mu_{d\to AB}\mu_{e\to AB}) + (1-\mu_{a\to AB})(1-\mu_{b\to AB})$$
$$-(1-\mu_{d\to AB}\mu_{e\to AB})(1-\mu_{a\to AB})(1-\mu_{b\to AB})$$

and the $P_2 =$ the probability of path failure considering a priori probability of failure of the link c is equal to 0.5:

$$P_2 = (1-0.5\mu_{d\to AB}\mu_{e\to AB}) + (1-\mu_{a\to AB})(1-\mu_{b\to AB})$$
$$-(1-0.5\mu_{d\to AB}\mu_{e\to AB})(1-\mu_{a\to AB})(1-\mu_{b\to AB})$$

The messages towards the nodes x_d and x_e are computed in a similar way with exception that in the first case, $\mu_{d\to AB}$ is replaced by $\mu_{c\to AB}$ and in the second case, $\mu_{e\to AB}$ is replaced by $\mu_{c\to AB}$.

In order to update the joint node $\delta_A\delta_B$ we need to:

- compute the terms $1-\mu$ for incoming messages from the nodes $\{x_a, x_b, x_c, x_d, x_e\}$; this requires 5 subtractions
- compute the products

1. $(1-\mu_{a\to AB})(1-\mu_{b\to AB})$
2. $0.5(1-\mu_{a\to AB})$
3. $0.5(1-\mu_{b\to AB})$

which requires 3 multiplications

- compute the terms

1. $\mu_{d\to AB}\mu_{e\to AB}$
2. $\mu_{c\to AB}\mu_{e\to AB}$
3. $\mu_{c\to AB}\mu_{d\to AB}$
4. $0.5\mu_{d\to AB}\mu_{e\to AB}$
5. $0.5\mu_{c\to AB}\mu_{e\to AB}$
6. $0.5\mu_{c\to AB}\mu_{d\to AB}$
7. $\mu_{d\to AB}\mu_{d\to AB}\mu_{e\to AB}$

which requires 7 multiplications

- compute the terms which are 1 minus the terms above; this requires 7 subtractions.
- compute the messages to x_a and x_b which, using the terms computed above, requires 4 multiplications, 2 additions, 2 subtractions and 2 divisions.
- compute the messages to x_c, x_d and x_e which requires 9 multiplications, 6 additions, 6 subtraction and 3 divisions.

The update of the node $\delta_A \delta_B$ requires a total of 56 operations: 23 multiplications, 8 additions, 20 subtractions and 5 divisions.

The node δ_F is not affected by the transformation and still requires 3 operations with probability $1-\alpha^3$. After the transformations only the link nodes x_e and x_d require any operations since the other nodes have degrees less than 3. Taking into account the number of operations defined above, the nodes x_e and x_d require 12 operations (both nodes have degree 3). Considering our assumption that all links have the same probability of success α, the number of operations required to update the graph in Figure 4.11 is expressed as:

$$C_T = 12 + 56\left(1-\alpha^3+(1-\alpha)^2-(1-\alpha^3)(1-\alpha)^2\right)+3(1-\alpha^3) \qquad (4.45)$$

For the $\alpha = 0.95$ the number of operations as defined by (4.45) is equal to 20.5. Therefore, by clustering the nodes δ_A and δ_B under the assumptions that the α is the same for all links and $\alpha = 0.95$, we lowered the average number of operations required for a single iteration of the sum-product algorithm from 39.4 to 20.5, which constitutes a reduction of 48%.

For a larger network the clustering of δ nodes could be more beneficial if the paths share more links. For example, in the case where two paths $w1$ and $w2$ consist of 10 links, $w_1 = \{x_1, x_3, x_4, \ldots x_{11}\}$ and $w_2 = \{x_2, x_3, x_4, \ldots x_{11}\}$, i.e., the paths share 9 links and differ by 1 link, by clustering the nodes δ_{w1} and δ_{w2} we can lower the degrees of 9 nodes $\{x_3, x_4, \ldots x_{11}\}$.

4.4 Summary

In this chapter, we discussed several practical applications of the method of lowering the complexity of the sum-product algorithm using graph transformations. For the applications of the factor graphs in Joint DNA Base-Calling, decoding of the Hamming (7,4) code and Link Loss Monitoring in Wireless Sensor Networks we were able to lower the complexity of the sum-product algorithm by 25%, 50% and 48%, respectively. We also showed that in the case of Joint DNA Base-Calling and decoding of the Hamming (7,4) code, converting the graph to the cycle-free form lowered the complexity of the sum-product algorithm. It is quite remarkable that the number of operations required to find the marginals on a cycle-free graph in these applications is less compared to the number of operations required for a single iteration of the sum-product algorithm on the graph with cycles.

The examples in this chapter clearly show that, contrary to the widespread belief, converting a graph to a cycle-free form does not always lead to a dramatic increase in complexity of belief propagation. Since the algorithm is exact in a cycle-free graph, it may be desirable to convert a graph to a cycle-free form even at the expense of a moderate increase in complexity.

Chapter 5

The depth N greedy search algorithm

In Chapter 4 we applied the factor graph transformations and reduced the complexity of the sum-product algorithm in several applications. The factor graphs that we considered were small or had repeated structures which allowed us to devise the transformations manually. In the majority of applications however, factor graphs have thousands of nodes and often random or pseudo-random structures. Therefore, we need an algorithm that can be applied to a general factor graph. The objective of the algorithm is to find the topology of the graph that corresponds to the least complex sum-product algorithm. This can be expressed as:

$$T_{min} = \arg\min_{T} \sum_{\forall vn \in G} C_{vn} + \sum_{\forall fn \in G} C_{fn} \tag{5.1}$$

where T_{min} is the transformation that results in the least complex sum-product algorithm, T is the set of all possible transformations, C_{vn} and C_{fn} are the number of operations required to update a variable node vn and a function node fn, respectively. There are a number of ways to approach this problem. Below, we will attack the problem directly but not necessarily in an optimal way. We develop a recursive greedy algorithm that, given a factor graph, attempts to find the solution to the optimization problem.

The complexity in expression (5.1) is used as the goal function for the optimization in this chapter. It has been noted by one of the reviewers of this thesis that different operations may have different latencies and complexities of the implementations in hardware. In order to take this into account, we may include in the goal function (5.1) complexity coefficients for each of the involved operations. The coefficients should correspond to the complexity of the implementations of the corresponding operations. In addition, as it has been noted a number of times, there are several benefits of having a graph in a cycle-free form. Therefore, if the cycle-free form has been achieved, it has to be detected by the algorithm and preference has to be given to graph in the cycle-free form.

A factor graph can be described by an adjacency matrix [46], which we denote by S. The rows of the matrix correspond to the local functions and the columns correspond to the variables. An element a_{ji} is 1 if and only if variable i is part of the domain of a function j, otherwise the element $a_{ji} = 0$. An example of an adjacency matrix with a corresponding factor graph is shown in Figure 5.1a. Given the degrees and sizes of the domains of each node in a graph, it is possible

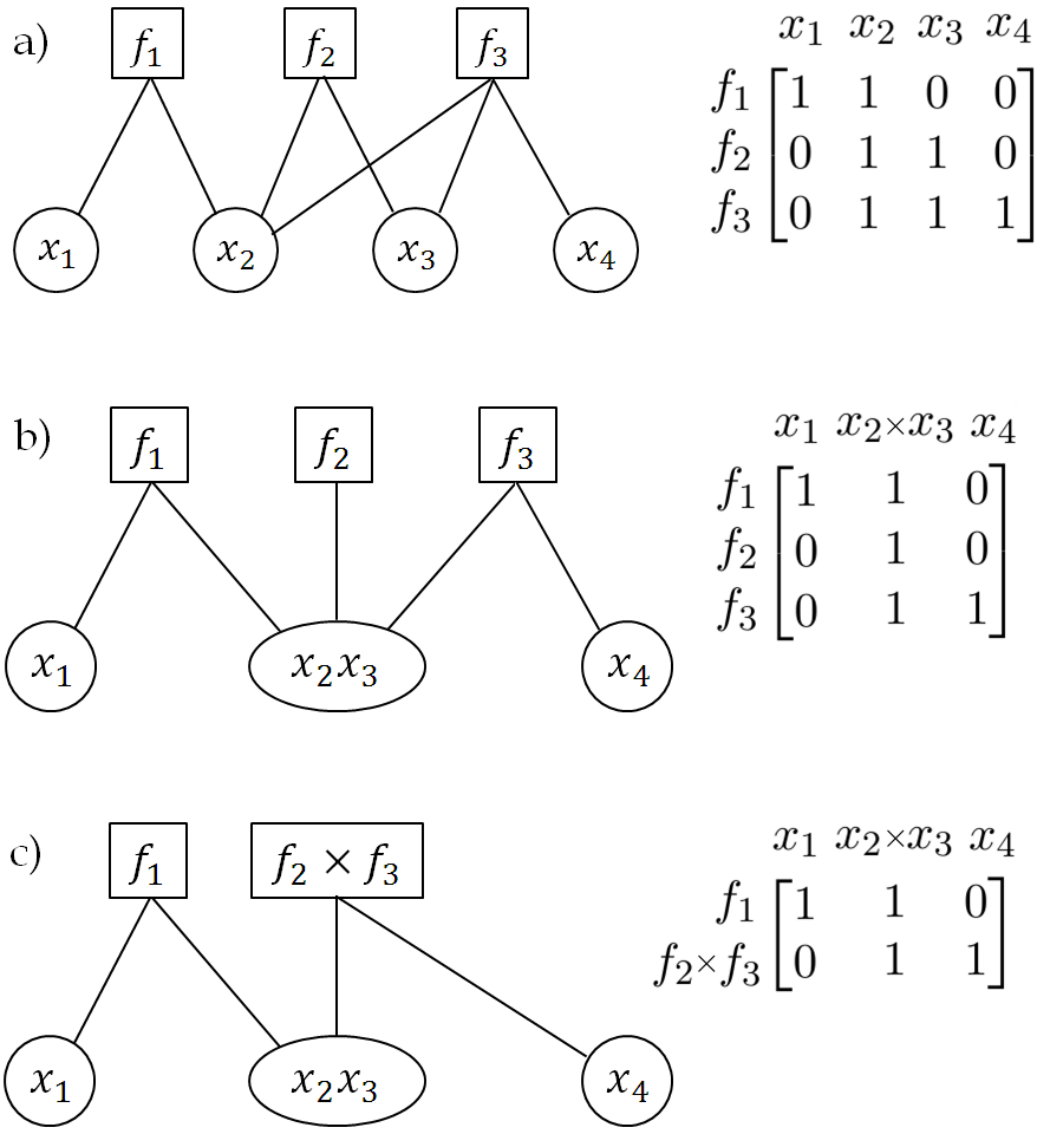


Figure 5.1: Example of a factor graph and the adjacency matrix for the graph(a). Effects of factor graph transformations on the adjacency matrix of the graph: clustering of variable nodes (b), clustering of function nodes(c).

to compute the operation counts required by the sum-product algorithm using the formulas defined in Section 3.3 equations (3.7), (3.14), and (3.15). The degree of a variable node i is the number of 1s in the column i of adjacency matrix S while the degree of a function node j is the number of 1s in the row j. An algorithm can compute the operation count by applying the above-mentioned formulas for each row and each column of S. The pseudo-code of the procedure is:

Input parameters: graph adjacency matrix S and vector of sizes of nodes domains Vd

Output: The number of operations C required by a single iteration of the sum-product algorithm on a given factor graph under the assumption that the flooding schedule is applied. Now we will show that the transformations of a factor graph

Algorithm 1 The function that computes the number of operations required by the sum-product algorithm

```
function INTEGER COMPUTESPCOMPLEXITY(S , Vd)
    C=0
    for each row of S (function node) do                    ▷ for all FN
        Compute A and M from (3.15, 3.14)
        C=C+A+M
    end for

    for each column of S (variable node) do                 ▷ for all VN
        Compute M from (3.7)
        C=C+M
    end for
     return C
end function
```

defined in Section 2.4 correspond to the operations with rows and columns of the matrix S. The transformations result in multiple variables and functions being as-

sociated with the nodes of a graph. This information has to be stored, perhaps in the form of a vector of vectors (one vector for each variable node and one for each function node).

- The clustering of variable nodes corresponds to replacing the columns corresponding to joined variables by a single column, which is formed from the original column using binary OR operation. The domain of new variable node (column) is the multiplication of the domains of variables included in the joined node. An example of such a transformation is shown in Figure 5.1b.

- As a result of the clustering of the function nodes, the original rows are replaced with the row formed by performing the OR operation on the original row. An example of such a transformation is shown in Figure 5.1c.

- To "stretch" a variable, the algorithm needs to add an element to the vector representing the variables associated with a node. The size of the domain of a variable node also needs to be multiplied by the size of the domain of the stretched variable.

- Removing an edge from a graph is represented by setting an element of the matrix to 0.

- Removing nodes corresponds to dropping a column from the matrix.

In order to conclude that an edge connecting a variable node i to a function node j is redundant, the algorithm has to verify that removing the edge does not change

the domain of the function node j which can be done by comparing the domains of the nodes connected to the function node j. The algorithm must also verify that after the transformation the graph still satisfies the running intersection property (Section 2.4). This can be accomplished using a graph search algorithm, such as a breadth-first or a depth-first algorithm (see for example [46]). A node can be removed if all edges connected to the node can be removed; in other words, if the node is represented by a 0-column.

As we have seen in Chapters 3 and 4, it may be desirable to stretch the nodes along the cycle, and then to break the cycle by removing one of the edges. In order to do this, we need to identify all the cycles that are shorter than a specified length. This also can be done using the breadth-first search algorithm [46].

The goal of the algorithm is to systematically explore the graph in an attempt to determine the transformations which lead to the minimal operation count. We say that in a factor graph a variable node j is adjacent to the other variable node i if the node j can be reached from the node i by a path of length 2. Similarly, a function node k is adjacent to the function node j if the node k can be reached from the node j by a path of length 2. The pseudo code of the depth 1 search algorithm is the following:

Input parameters: graph adjacency matrix S, vector of domain sizes Vd, length of the longest explored cycle L

Output: The lowest operation count achievable by a single transformation, the transformation that achieves this count

Algorithm 2 The algorithm determines depth 1 transformations which leads to the minimal operation count

```
function INTEGER DEPTH1SEARCH(S, Vd, L)
    Minimal complexity= ComputeSPComplexity(S , Vd)
    for each column of S (variable node i) do      ▷ Clustering of variable nodes
        for each variable node j adjacent to the node i do
            Join i and j, compute new S' and Vd'
            complexity = ComputeSPComplexity(S' , Vd')
            if complexity < Minimal complexity then
                Minimal complexity= complexity
                Save the transformation as the new candidate
            end if
        end for
    end for
    for each row of S (function node j) do          ▷ Clustering of function nodes
        for each function node k adjacent to the node j do
            Join k and j, compute new S' and Vd'
            complexity = ComputeSPComplexity(S' , Vd')
            if complexity< Minimal complexity then
                Minimal complexity= complexity
                Save the transformation as the new candidate
            end if
        end for
    end for
```

```
    Find all cycles of length L or less                    ▷ Checking cycles
    for each cycle of length L or less do
        for each node in the cycle do
            Stretch the variables along the cycle, break the cycle, compute new
S' and Vd'
            complexity =ComputeSPComplexity(S' , Vd')
            if complexity < Minimal complexity then
                Minimal complexity= complexity
                Save the transformation as the new candidate
            end if
        end for
    end for
     return Minimal complexity and the transformation candidate
end function
```

The following is the pseudo-code of the program that optimizes the complexity of the sum-product algorithm. The program essentially calls **Depth1Search(S, Vd, L)** in **Repeat Until** cycle till **Depth1Search** cannot find a transformation that reduces the operation count.

The depth 1 search algorithm may however get stuck since it is possible that a transformation, which originally increases the complexity, leads to other transformations that will reduce the complexity. To avoid this problem, we propose another algorithm which at first explores all possible transformations up to depth

Algorithm 3 OptimizeSumProduct - The algorithm optimizes complexity of the sum-product algorithm using factor graph tranfomations

```
repeat
    Current complexity = ComputeComplexity(S , Vd)
    NewMinOperationsCount=Depth1Search(S, Vd, L)
    if NewMinOperationsCount < Current operation count then
        Transform the graph, compute new S, Vd
    end if
until Current complexity > NewMinOperationsCount
```

N and then selects the next transformation which leads to the greatest overall decrease in complexity. The pseudo-code of such a search algorithm is given below. The only difference with **Depth1Search** is that there are recursive calls within each **for** block.

Input parameters: graph adjacency matrix S, vector of domain sizes Vd, length of longest explored cycle L, depth of the search N.

Output: complexity C, The lowest operation count achievable by N transformations, the transformations that achieve this count

Algorithm 4 The algorithm determines depth N transformations which lead to the minimal operation count

function INTEGER DEPTHNSEARCH(S, Vd, L, N)

Minimal complexity= ComputeComplexity(S , Vd)

for each column of S (variable node *i*) **do** ▷ Clustering variable nodes

for each variable node j adjacent to variable node *i* **do**

Join *i* and *j*, compute new S' and Vd'

if N=0 **then**

Complexity = ComputeComplexity(S' , Vd')

else ▷ **Recursive call to self**

Complexity = DepthNSearch(S', Vd', L, N-1)

end if

if complexity $<$ Minimal complexity **then**

Minimal complexity=complexity

Save the transformations as the sequence of transformations-candidates

end if

end for

end for

for each row of S (function node *j*) **do** ▷ Clustering function nodes

for each function node k adjacent to function node *j* **do**

Join *k* and *j*, compute new S' and Vd'

```
            if N=0 then
                Complexity = ComputeComplexity(S' , Vd')
            else                                          ▷ Recursive call to self
                Complexity = DepthNSearch(S', Vd', L, N-1)
            end if
            if complexity < Minimal complexity then
                Minimal complexity= complexity
                Save the transformations as the sequence of transformations-
candidates
            end if
        end for
    end for
    Find all cycles of length L or less                   ▷ Checking cycles
    for each cycle of length L or less do
        for each node in the cycle do
            Stretch the variables along the cycle, break the cycle, compute new
S' and Vd'
            if N=0 then
                Complexity = ComputeComplexity(S' , Vd')
            else                                          ▷ Recursive call to self
                Complexity = DepthNSearch(S', Vd', L, N-1)
            end if
```

if complexity $<$ Minimal complexity **then**

Minimal complexity= complexity

Save the transformations as the sequence of transformations-candidates

end if

end for

end for

return Minimal complexity and transformation candidate

end function

The algorithm that optimizes the complexity of the sum-product algorithm using DepthNSearch function is the same as ComputeSPComplexity except that Depth1Search function is replaced with DepthNSearch function.

There is no guarantee that the algorithms presented above will find the optimal graph. The algorithm however, may work in many cases. It will find all of the transformations that we applied manually in Chapter 4.

The complexity of the algorithm increases exponentially with increases in the depth of the search. The optimization however, has to be performed only once during the system design. It also appears that in many cases, the search of depth 2-4 may be enough for the algorithm to proceed without getting stuck. In fact, for all of the examples presented throughout this thesis, the depth 1 search will find the solutions.

The drawback of the algorithm is that during the different steps it explores the

same sequence of transformations in successive steps of the search over and over again. In order to improve the algorithm, one may attempt to reuse the information about already explored paths.

The other way to approach the problem would be to recursively partition the graph into smaller sub-graphs and to try to optimize each of the sub-graphs independently.[1] Then the algorithm may explore a large number of possible graph partitions with the hope that the optimal transformation on each step will be identifiable and will belong to a single sub-graph.

[1]For example, it may work by partitioning a graph to 2 sub-graphs, then each of the subgraphs on 2 sub-graphs again and so on till arrive to a sub-graph where optimal transforation can be found or known

Chapter 6

Conclusions

6.1 The summary of contributions

The main contributions of this thesis are the following:

1. A novel approach aimed at increasing the efficiency of the sum-product and belief propagation algorithms using graph transformations has been introduced.

2. It has been shown that, at least for some applications, the approach may significantly lower the number of operations required by the sum-product algorithm.

3. The applicability of the method of lowering complexity to different practical applications of factor graphs has been demonstrated.

4. An algorithm that optimizes the complexity of the sum-product algorithm

has been developed.

5. An efficient way of performing the sum-product nodes computations for a function node have been proposed.

6. It has been demonstrated that converting a graph to a cycle-free form does not always significantly increase the number of operations required by the sum-product algorithm. In some cases converting a graph to the cycle-free form may even decrease the complexity of the algorithm.

6.2 Conclusions and suggestions for future research

In this thesis we showed that the complexity of the sum-product algorithm in its original form is not optimal under all circumstances. We introduced a novel framework for lowering the complexity of the algorithm using graph transformations. We demonstrated that the approach can significantly lower the number of operations required to find marginals using the sum-product algorithm. The proposed approach is general and can be applied to a variety of applications of the sum-product algorithm. As examples, we used transformations and successfully lowered the complexity in applications of the sum-product algorithm in Joint DNA Base-Calling, decoding of the Hamming(7,4) code and Wireless Link Monitoring in Wireless Sensor Networks. On considered models transformation of the graphs lowered the number of arithmetic operations required for a single iteration of the sum-product algorithm by 25-65%. Remarkably, in the cases of the Joint DNA Base-Calling and the decoding of the Hamming(7,4) code, the number of oper-

ations required to find the marginals on the cycle-free graph is less compared to the number of operations required to perform a single iteration on the graphs with cycles. In the case of the Hamming (7,4) code, the decoding on the transformed cycle-free graph requires approximately 10 times fewer operations compared to the decoding on the original graph with cycles.

Considering that the concept of graph transformations is general we expect that the transformations can also be applied to lower the complexity in related graphical models such as Bayesian Networks and Markov random fields.

In practical settings, given a factorization of a global function, one would be interested in determining the minimal achievable number of operations required to find the marginals. The other question is how to devise the algorithm that achieve this number. Further research is needed to answer this questions.

The method of lowering the complexity removes short cycles from a graph which would likely to improve the convergence and accuracy of the sum-product algorithm as it has been seen on the example of the Hamming(7,4) code. This effect is yet to be investigated in future research.

Ultimately, by utilizing the approach presented in this thesis the designers of real-life applications may be able to reduce the computational time and lower the power consumption of power constrained devices. In addition, it may be possible to implement more comprehensive models that take into account a greater number of factors and that correspond better to real world phenomena.

References

[1] X. Shi, D. Lun, M. Medard, R. Kotter, J. Meldrim, and A. Barry, "Joint base-calling of two DNA sequences with factor graphs," *IEEE Trans. Inf. Theory*, vol. 56, pp. 724 –733, Feb. 2010.

[2] S. Aji and R. McEliece, "The generalized distributive law," *IEEE Trans. Inf. Theory*, vol. 46, pp. 325–343, Mar. 2000.

[3] F. Kschischang, B. Frey, and H.-A. Loeliger, "Factor graphs and the sum-product algorithm," *IEEE Trans. Inf. Theory*, vol. 47, pp. 498–519, Feb. 2001.

[4] B. J. Frey, F. R. Kschischang, H. andrea Loeliger, and N. Wiberg, "Factor graphs and algorithms," in *Proc. of the 35th Annu. Allerton Conf. on Communications, Control and Computing Allerton*, pp. 666–680, 1998.

[5] N. Wiberg, H.-A. Loeliger, and R. Kotter, "Codes and iterative decoding on general graphs," in *Proc. IEEE Symp. Information Theory,*, p. 468, Sept. 1995.

[6] E. Berlekamp, R. McEliece, and H. van Tilborg, "On the inherent intractability of certain coding problems," *IEEE Trans. Inf. Theory*, vol. 24, pp. 384 – 386, May 1978.

[7] K. P. Murphy, Y. Weiss, and M. I. Jordan, "Loopy belief propagation for approximate inference: An empirical study," in *Proceedings of Uncertainty in AI*, pp. 467–475, 1999.

[8] Y. Weiss, "Correctness of local probability propagation in graphical models with loops," *Neural Comput.*, vol. 12, pp. 1–41, January 2000.

[9] Y. Mao and A. Banihashemi, "A heuristic search for good low-density parity-check codes at short block lengths," in *Proc. IEEE International Conference on Communications,ICC 2001.*, vol. 1, pp. 41–44 vol.1, jun 2001.

[10] S. C. Tatikonda and M. I. Jordan, "Loopy belief propagation and gibbs measures," in *Uncertainty in Artificial Intelligence*, pp. 493–500, Morgan Kaufmann, 2002.

[11] M. J. Wainwright, T. S. Jaakkola, and A. S. Willsky, "A new class of upper bounds on the log partition function," in *Uncertainty in Artificial Intelligence*, pp. 536–543, 2002.

[12] J. Yedidia, W. Freeman, and Y. Weiss, "Constructing free-energy approximations and generalized belief propagation algorithms," *IEEE Trans. Inf. Theory*, vol. 51, pp. 2282–2312, July 2005.

[13] A. T. Ihler, J. W. F. Iii, A. S. Willsky, and M. Chickering, "Loopy belief propagation: Convergence and effects of message errors," *Journal of Machine Learning Research*, vol. 6, pp. 905–936, 2005.

[14] J. Mooij and H. Kappen, "Sufficient conditions for convergence of the sum-product algorithm," *IEEE Trans. Inf. Theory*, vol. 53, pp. 4422 –4437, Dec. 2007.

[15] J. M. Mooij and H. J. Kappen, "Loop corrections for approximate inference on factor graphs," *Journal of Machine Learning Research*, vol. 8, pp. 1113–1143, May 2007.

[16] T. Roosta, M. Wainwright, and S. Sastry, "Convergence analysis of reweighted sum-product algorithms," *IEEE Trans. Signal Process.*, vol. 56, pp. 4293 –4305, Sept. 2008.

[17] J. K. Johnson, D. Bickson, and D. Dolev, "Fixing convergence of Gaussian belief propagation," in *Proc. IEEE International Symposium on Information Theory*, ISIT'09, pp. 1674–1678, 2009.

[18] M. Fossorier, M. Mihaljevic, and H. Imai, "Reduced complexity iterative decoding of low-density parity check codes based on belief propagation," *IEEE Trans. Commun.*, vol. 47, pp. 673 –680, May 1999.

[19] L. Ping and W. Leung, "Decoding low density parity check codes with finite quantization bits," *IEEE Commun. Lett.*, vol. 4, pp. 62–64, Feb. 2000.

[20] J. Chen, A. Dholakia, E. Eleftheriou, M. Fossorier, and X.-Y. Hu, "Reduced-complexity decoding of LDPC codes," *IEEE Trans. Commun.*, vol. 53, pp. 1288–1299, Aug. 2005.

[21] J. Zhao, F. Zarkeshvari, and A. Banihashemi, "On implementation of min-sum algorithm and its modifications for decoding low-density Parity-check (LDPC) codes," *IEEE Trans. Commun.*, vol. 53, pp. 549 – 554, Apr. 2005.

[22] Y. Mao, F. Kschischang, B. Li, and S. Pasupathy, "A factor graph approach to link loss monitoring in wireless sensor networks," *IEEE J. Sel. Areas Commun.*, vol. 23, pp. 820–829, Apr. 2005.

[23] "Belief propagation." http://en.wikipedia.org/wiki/Belief_propagation.

[24] R. Diestel, *Graph Theory*, vol. 173 of *Graduate Texts in Mathematics*. Springer-Verlag, Heidelberg, third ed., 2005.

[25] B. J. Frey, *Graphical models for machine learning and digital communication*. Cambridge, MA, USA: MIT Press, 1998.

[26] H.-A. Loeliger, J. Dauwels, S. K. J. Hu, L. Ping, and F. Kschischang, "The factor graph approach to model-based signal processing," *Proc. of the IEEE*, vol. 95, pp. 1295–1322, June 2007.

[27] M. Wainwright, T. Jaakkola, and A. Willsky, "A new class of upper bounds on the log partition function," *IEEE Trans. Inf. Theory*, vol. 51, pp. 2313–2335, July 2005.

[28] M. Yazdani, S. Hemati, and A. Banihashemi, "Improving belief propagation on graphs with cycles," *IEEE Commun. Lett.*, vol. 8, pp. 57 – 59, Jan. 2004.

[29] Y. Mao and A. Banihashemi, "Decoding low-density parity-check codes with probabilistic scheduling," *IEEE Commun. Lett.*, vol. 5, pp. 414 –416, Oct. 2001.

[30] H. Xiao and A. Banihashemi, "Graph-based message-passing schedules for decoding LDPC codes," *IEEE Trans. Commun.*, vol. 52, pp. 2098–2105, Dec. 2004.

[31] N. Taga and S. Mase, "On the convergence of loopy belief propagation algorithm for different update rules," *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, vol. E89-A, pp. 575–582, Feb. 2006.

[32] G. Elidan, "Residual belief propagation: Informed scheduling for asynchronous message passing," in *Proc. 22d Conf. Uncertainty in Artificial Intelligence*, (Cabridge, USA), 2006.

[33] Y. Mao, "Lecture notes: ELG5131 Graphical Models and Their Applications,University of Ottawa, SITE ," 2006.

[34] A. Leon-Garcia, *Probability and Random Processes for Electrical Engineering*. Addison-Wesley, 1994.

[35] D. Bickson, *Gaussian Belief Propagation: Theory and Application*. PhD thesis, Hebrew University of Jerusalem, July 2009.

[36] E. B. Sudderth, A. T. Ihler, M. Isard, W. T. Freeman, and A. S. Willsky, "Nonparametric belief propagation," *Commun. ACM*, vol. 53, pp. 95–103, Oct. 2010.

[37] J. Pearl, *Probabilistic Reasoning in Intelligent Systems : Networks of Plausible Inference*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1988.

[38] R. Gallager, "Low-density parity-check codes," *IEEE Trans. Inf. Theory*, vol. 8, pp. 21–28, Jan. 1962.

[39] D. MacKay, "Good error-correcting codes based on very sparse matrices," *IEEE Trans. Inf. Theory*, vol. 45, pp. 399 –431, Mar. 1999.

[40] N. Moller and T. Granlund, "Improved division by invariant integers," *IEEE Trans. Comput.*, vol. 60, pp. 165 –175, feb. 2011.

[41] T. K. Moon, *Error Correction Coding: Mathematical Methods and Algorithms*. Wiley-Interscience, 2005.

[42] F. Sanger, S. Nicklen, and A. Coulson, "DNA sequencing with chain-terminating inhibitors," *Proceedings of the National Academy of Sciences of the United States of America (PNAS)*, vol. 74, pp. 5463–5467, 1977.

[43] H. Wymeersch, H. Steendam, and M. Moeneclaey, "Log-domain decoding of LDPC codes over GF(q)," in *Proc. IEEE International Conference on Communications*, vol. 2, pp. 772–776, June 2004.

[44] A. Namin, K. Leboeuf, R. Muscedere, H. Wu, and M. Ahmadi, "Efficient hardware implementation of the hyperbolic tangent sigmoid function," in *IEEE International Symposium on Circuits and Systems*, pp. 2117 –2120, may 2009.

[45] T. Sina, "Performance of BP decoder of the Hamming(7,4) code." Private communications.

[46] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd ed., 2001.