POLITECNICO DI TORINO

Master degree in
Physics of Complex Systems

Lossy compression and other hard optimization problems: a statistical physics approach

SUPERVISOR
prof. Alfredo Braunstein
Politecnico di Torino - DISAT

CANDIDATE
Stefano Crotti

CO-SUPERVISOR
prof. Alessandro Barenghi
Politecnico di Milano - DEI

Academic year 2019/2020

Abstract

The task of data compression originally arises in the field of communications, where messages from a source must be delivered to a receiver in the most compact form, possibly with loss of information. Besides the countless practical applications in engineering, data compression is interesting enough to be studied on its own, its foundations being rooted in probability theory, optimization and other related disciplines which are in turn deeply linked to statistical physics. The class of problems analyzed here can be described as follows: given a random bit string of fixed length, design a compressing algorithm that casts it into a shorter string and a de-compressing algorithm that is able to reconstruct the original message up to some error.

A cunning approach from Information Theory addresses the challenge by setting up a constrained optimization problem. It just so happens that such problem is completely equivalent to that of finding the ground state of an Ising model defined on a graph that encodes the system of constraints. This opens up the way for a variety of well-known tools from statistical physics to be used in order to find an approximate solution that works in polynomial time, in our case the Belief Propagation algorithm. An efficient computer implementation is presented along with the results and a proof of exactness of the algorithm under some conditions is given.

From the practical point of view, it was found that better performances are achieved by working with strings of numbers taking values in the so-called Galois Field $\mathbb{GF}(q)$, which can be thought of as a generalization of bits, taking values in $0, ..., q-1$. From the theoretical point of view, the most relevant observation is that a small modification of the graph's structure can lead to a remarkable increase in performance. There is good reason to think that this fact, together with the rest of our method, may be applied to other hard problems in the constrained optimization family.

Contents

1 Statistical physics and optimization **4**
- 1.1 The Ising model . . . 5
- 1.2 The two faces of GF(2) . . . 6
- 1.3 Representing constraints with factor graphs . . . 8
- 1.4 Examples . . . 11
  - 1.4.1 Min-cut and ferromagnetic Ising ground state . . . 11
  - 1.4.2 p-XORSAT formulas and p-spin Ising . . . 13

2 Lossy compression as constrained optimization **17**
- 2.1 Problem framing . . . 17
- 2.2 Taking inspiration from LDPC codes . . . 18
  - 2.2.1 Channel coding . . . 19
  - 2.2.2 Parity-check codes . . . 20
  - 2.2.3 Codewords as points in space . . . 22
  - 2.2.4 Low-Density Parity-Check codes . . . 24
- 2.3 A compression scheme . . . 25
  - 2.3.1 Encoding - part one . . . 26
  - 2.3.2 Encoding - part two . . . 27
  - 2.3.3 Decoding . . . 27

3 Methods **28**
- 3.1 Irregular graphs . . . 28
  - 3.1.1 Theory . . . 28
  - 3.1.2 Graph construction . . . 30
  - 3.1.3 Choice of lambda and rho . . . 31
- 3.2 Belief Propagation . . . 33
  - 3.2.1 The BP equations . . . 33
  - 3.2.2 Max-sum: the zero-temperature version of BP . . . 35
  - 3.2.3 Max-sum for our compression scheme . . . 36
  - 3.2.4 Complexity . . . 37
- 3.3 A proof of exactness . . . 37
  - 3.3.1 Computation trees . . . 38
  - 3.3.2 Statement and proof . . . 39
- 3.4 Reduced factor graphs and energy landscape . . . 41
- 3.5 Moving to GF(q) . . . 42
- 3.6 Efficient implementation of BP . . . 43
  - 3.6.1 Convolutions . . . 43
  - 3.6.2 BP and MS with convolutions . . . 45

4 Experimental results and conclusions **48**
4.1 Implementation details and notation 48
4.1.1 Reinforced Belief Propagation 48
4.1.2 Symmetry-breaking fields 49
4.1.3 Message updates 49
4.1.4 Multiple trials 50
4.1.5 Damping 50
4.1.6 Convergence criteria 50
4.1.7 Performance assessment 51
4.2 Results with graph reduction 51
4.3 Results with reinforcement 52
4.4 Comparison of field orders 52
4.5 Conclusions 54

Acknowledgements **56**

Bibliography **57**

Chapter 1

Statistical physics and optimization

Nature is constantly optimizing: the spherical shape of bubbles, the trajectories of motion, the ordered structure of crystalline materials, the hexagonal form of honeycombs. These are all examples of a system organizing itself so as to optimize some cost function. Formulations of physics such as Lagrangian and Hamiltonian mechanics rely on the well-known principle of *least* action, suggesting that the fundamentals of motion itself are ultimately based on an optimization process.

Minimization and maximization problems often appear explicitly in the context of equilibrium statistical physics, where typically a system settles in a configuration which is overwhelmingly more likely than all others. Such state with maximum probability is equivalent to that of minimum free energy, as is clear from the grand canonical ensemble formulation

$$\begin{aligned} P(\mathbf{s}) &\propto \exp\left[-\beta F(\mathbf{s})\right] \\ &= \exp\left\{-\beta\left[H(\mathbf{s}) - k_B T S(\mathbf{s})\right]\right\} \end{aligned} \tag{1.1}$$

where **s** indicates a macro-state, i.e. the set of all configurations satisfying some constraint, and the entropy S is a logarithmic measure of how large such set is. In the limit of **s** including only a single micro-state, we have $S(\mathbf{s}) = 0$ and we find the well-known Boltzmann distribution

$$P(\mathbf{s}) \propto \exp\left[-\beta H(\mathbf{s})\right] \tag{1.2}$$

For more than 150 years now, statistical physicists have been putting effort into this and other optimization problems arising when studying magnetic systems, classic and quantum gases, liquid/gas transitions and many more. More recently, optimization problems showed up as core ingredients in areas such as operations research, statistics and learning. Despite the more modern flavor, in many of these cases there can be found a tight correspondence with the "old" puzzles from physics. Exploring in deep the reasons for the existence of such links would probably require a lifetime's work: here we will limit ourselves to mentioning some mathematical facts that make the connection possible and providing a few examples. Even the reader who is not fascinated by the above mentioned connections will hopefully appreciate how useful it is, when dealing with a task connected to several areas of science, to be able to draw tools from any of them. This work is also a modest attempt at suggesting the need for scientists who can move with ease through and between different disciplines.

Before diving into the features of the statistical physics - constrained optimization parallel, it is useful to establish some common background by briefly reviewing what has been called the "favorite toy-world of the statistical physicist" [MacKay, 2014]: the Ising model.

1.1 The Ising model

An Ising system consists of a set of N degrees of freedom, called "spins", which can take value in $\{-1, +1\}$, interacting according to some given energy function. The name spins is reminiscent of the fact that the model was born to describe the behavior of large collections of magnetic dipoles, although it can be applied to a variety of different situations.

The general form for the energy function, called H as in Hamiltonian, is

$$-H(\boldsymbol{\sigma}) = \sum_{1 \le i < j \le N} J_{ij} \sigma_i \sigma_j + \sum_{i=1}^{N} h_i \sigma_i \tag{1.3}$$

where $\boldsymbol{\sigma} \equiv \sigma_1, \sigma_2, ..., \sigma_N$ is the configuration of all spin variables, $J_{ij} \in \mathbb{R}$ are called the *couplings* between spins i and j and $h_i \in \mathbb{R}$ are called *external fields*. Pairs of spins with large (absolute value) coupling tend to influence each other: if $J_{ij} > 0$, it is energetically convenient for the two to take the same value, vice-versa if $J_{ij} < 0$. $J_{ij} = 0$ means that spins i and j do not influence each other; an alternative way of communicating the energy function is to specify only the non-zero couplings. In this case, spins that do interact are called *neighbors*. A coupling is said to be *satisfied* if sign $(J_{ij}) = \sigma_i \sigma_j$, otherwise it is said to be *frustrated*. At the same time, the second term in (1.3) gives lower energy when spins are aligned with their field components.

We can already notice that there is ground for potential conflict: couplings from some neighbors can encourage a spin to be in state $+1$ while others to be in state -1, and the same goes for the field. Imagine the poor spin being shouted discordant orders from all directions: what will it do? It turns out that there are cases where this question is extremely hard to answer, to the point that Ising systems are sometimes used as a tool for the study of complexity itself (see for instance [Stein and Newman, 2013]).

As is often the case in physics, the energy function is all one needs to know in order to characterize the system's behavior and "solve" it. The system at equilibrium at inverse temperature $\beta = \frac{1}{k_B T}$ is found in state $\boldsymbol{\sigma}$ with probability $P(\boldsymbol{\sigma}|\beta)$

$$P(\boldsymbol{\sigma}|\beta) = \frac{\exp\left[-\beta H(\boldsymbol{\sigma})\right]}{Z(\beta)} \tag{1.4}$$

where

$$Z(\beta) = \sum_{\sigma_1 = \pm 1} \sum_{\sigma_2 = \pm 1} \cdots \sum_{\sigma_N = \pm 1} \exp\left[-\beta H(\boldsymbol{\sigma})\right] \tag{1.5}$$

is the temperature-dependent normalization constant, called partition function. The summations in (1.5) are intentionally made explicit to stress the fact that, unless H has some "nice" form, the computation of $Z(\beta)$ requires a number of operations scaling as 2^N. Since typically statistical mechanics deals with very large systems ($N \to \infty$), this can represent a challenge.

The power of the Ising model comes from the fact that, starting from a fairly simple expression for the interaction between microscopic variables, one can perform some calculations and extract useful macroscopic length scale properties. For example, one could be interested in the magnetization $M(\boldsymbol{\sigma}) = \sum_i \sigma_i$, which is a measure of the net balance between $+1$ and -1 throughout the whole system. The expectation value of the magnetization is given by a weighted average over all states

$$\langle M(\boldsymbol{\sigma}) \rangle = \sum_{\boldsymbol{\sigma}} M(\boldsymbol{\sigma}) P(\boldsymbol{\sigma}|\beta) \tag{1.6}$$

which again, in principle, suffers the same exponential computational cost as the partition function.

Let us explore what happens at the two extreme temperature limits: the simplest one is $\beta \to 0$, or equivalently $T \to \infty$. In this case the distribution in (1.4) becomes uniform and each state has the same probability.

More interesting is the case $\beta \to \infty$, or $T \to 0$. We have

$$P(\boldsymbol{\sigma}|\beta \to \infty) = \lim_{\beta \to \infty} \frac{\exp\left[-\beta H(\boldsymbol{\sigma})\right]}{\sum_{\boldsymbol{\sigma}} \exp\left[-\beta H(\boldsymbol{\sigma})\right]} \tag{1.7}$$

The summation at denominator is dominated by its largest term(s), corresponding to the lowest energy: if we call $\Sigma^* = \arg\min_{\boldsymbol{\sigma}} H(\boldsymbol{\sigma})$, then

$$H(\boldsymbol{\sigma}^*) < H(\boldsymbol{\sigma}), \quad \forall\, \boldsymbol{\sigma}^* \in \Sigma^*, \boldsymbol{\sigma} \notin \Sigma^* \tag{1.8}$$

and, therefore

$$\exp\left[-\beta H(\boldsymbol{\sigma^*})\right] \gg \exp\left[-\beta H(\boldsymbol{\sigma})\right] \tag{1.9}$$

The probability finally becomes

$$P(\boldsymbol{\sigma}|\beta \to \infty) = \frac{\mathbb{1}\left[\boldsymbol{\sigma} \in \Sigma^*\right]}{|\Sigma^*|} \tag{1.10}$$

$|\Sigma^*|$ being the cardinality of the set Σ^*, i.e. the number of configuration that yield to the minimum energy value. In physics lingo, $H(\boldsymbol{\sigma}^*)$ is called the ground state and $|\Sigma^*|$ its degeneracy. Equation (1.10) tells us that lowering the temperature to zero corresponds to forcing the system to one of the states with minimum energy, each of them coming up with uniform probability. Of course, if the ground state is unique, we will find the system in that state with probability 1.

Finally, an Ising model where the couplings are all non-negative $J_{ij} \geq 0$ is called *ferromagnetic*. Conversely, if $J_{ij} \leq 0\, \forall\, i, j$, it is called *anti-ferromagnetic*.

As it was mentioned earlier, despite having been conceived for the study of magnetic materials, the Ising model has found vast applications in many different areas of physics: DNA stretching [Nelson, 2004] and melting [Badasyan et al., 2010], liquid/gas transition [Lee and Yang, 1952], superfluidity in Helium mixtures [Blume et al., 1971] are just a few examples. Such extreme versatility is probably due to the simplicity of the model (eq. (1.3) is not complicated after all) that nevertheless leads to great predictive power on complex phenomena as the ones just cited. It sounds plausible then, that such generality might be exploited in fields outside physics!

1.2 The two faces of $\mathbb{GF}(2)$

Let us now describe a mathematical fact which is a key element in our discussion. Binary variables play a fundamental role in (at least) two contexts: spin systems, as described in the previous section, and the digital world, where information is encoded in binary digits (bits). In the following, bit variables taking value in $\{0, 1\}$ will be indicated by Latin letters (x in this section) and spins by the Greek letter σ.

Let us focus on bits: mathematically speaking, the set $\{0, 1\}$, endowed with multiplication and sum operations, constitutes the finite field $\mathbb{GF}(2)$. Multiplication is performed by the AND operator, while the sum is intended modulo 2 and without carry, which is

&	0	1
0	0	0
1	0	1

⊕	0	1
0	0	1
1	1	0

Figure 1.1: Tables for the AND and XOR operations

what the XOR operator does. We will denote multiplication with no symbol or "$\cdot$", just as it is customary with real numbers, and XOR sums with "$\oplus$".

We can define vector spaces over $\mathbb{GF}(2)$ and do linear algebra. A vector **x** is a list of n components in $\mathbb{GF}(2)$, vector sum is defined as bitwise XOR

$$\mathbf{x} + \mathbf{y} = \begin{bmatrix} x_1 \oplus y_1 & x_2 \oplus y_2 & \cdots & x_n \oplus y_n \end{bmatrix} \tag{1.11}$$

and multiplication by a scalar performs an AND operation on all components

$$0\mathbf{x} = \mathbf{0} \quad 1\mathbf{x} = \mathbf{x} \tag{1.12}$$

Since multiplication by a scalar either gives zero or leaves a vector unchanged, taking linear combination of vectors simply amounts to picking or not each one of them and then summing together. This means that the AND operation applied to vectors is so trivial that it hardly deserves the status of operation.

Let us note a few additional facts which follow from the above definitions:

(i) The sum of two vectors is 1 for the components that differ in the two addends and 0 where they are equal. In this sense we might say that vector sum is the indicator function of the differences between the summands

(ii) The zero vector always appears in a vector space (this is true in general, not just for $\mathbb{GF}(2)$)

(iii) $\mathbf{x} + \mathbf{x} = \mathbf{0}$ for any vector **x**.

Turning now our attention back to spins, if we take the multiplication table for the set $\{-1, +1\}$ in fig. 1.2, we see that it looks just like the one for the XOR operation in fig. 1.1.

For the sake of formality, we can introduce a counterpart for the AND operator as well, although this will not be used:

$$\sigma_1 \,\&\, \sigma_2 = \begin{cases} +1 & \text{if } \sigma_1 = \sigma_2 = 1 \\ -1 & \text{otherwise} \end{cases}$$

We can conclude that the field composed by the set $\{-1, +1\}$ endowed with spin multiplication as sum and "&" as multiplication is isomorphic to $\mathbb{GF}(2)$.

•	1	-1
1	1	-1
-1	-1	1

Figure 1.2: Table for spin multiplication

The mapping from one representation to the other can be done in an infinite amount of different ways. Two common examples are

$$\sigma = 2x - 1 \qquad \Longleftrightarrow \qquad x = \frac{1-\sigma}{2} \tag{1.13}$$

and

$$\sigma = (-1)^x \qquad \Longleftrightarrow \qquad x = \log_{-1} \sigma \tag{1.14}$$

where $\log_{-1}$ is the inverse of $(-1)^{(\cdot)}$ on the appropriate domain. Notice that in the first case -1 corresponds to 0 and $+1$ to 1, vice-versa in the second.

The main message from this section is that we can do linear algebra with binary variables, for example we can look for solutions to linear systems of equations. Basic rules of matrix calculus can be exploited, notably Gaussian elimination for system inversion, with due modifications that account for the somewhat special algebra underlying $\mathbb{GF}(2)$. All of this can be done interchangeably for bits or spins.

1.3 Representing constraints with factor graphs

A factor graph is a graphical model that represents the mutual interactions among variables in a system (see [Loeliger, 2004] for a more detailed introduction). Although potentially more general, factor graphs are mostly used in probabilistic applications, where they are sometimes called *Markov random fields*. The idea is that, if a bunch of variables are all connected together in some complex way, but so that each one is interacting with a small number of others, it may be possible to efficiently compute marginal probability distributions and averages. This is made possible thanks to a certain notion of independence (Markov blanket property) implied by factorization.

Say we have a multivariate discrete probability distribution which can be factorized, i.e. written as the product of factors

$$p(x_1, x_2, ..., x_N) = \prod_{f \in F} p_f(\mathbf{x_f}) \tag{1.15}$$

where F is the set of factors and $\mathbf{x_f}$ is a short-hand notation for the set of variables which appear in factor f. Energy functions in statistical physics are suitable for this representation as most of them are written as a sum of terms, which becomes a product once raised

at exponent to give a probability. Take for instance the Ising model

$$p(\sigma_1, \sigma_2, ..., \sigma_N) \propto \exp\left[\beta\left(\sum_{1\le i<j\le N} J_{ij}\sigma_i\sigma_j + \sum_{i=1}^{N} h_i\sigma_i\right)\right] \tag{1.16}$$

$$\propto \prod_{1\le i<j\le N} \exp\left(\beta J_{ij}\sigma_i\sigma_j\right) \prod_{i=1}^{N} \exp\left(\beta h_i\sigma_i\right) \tag{1.17}$$

Here we have a factor for each couple of interacting spins (first term) plus an additional one for each spin (second term).

The relations between variables and factors are best represented by means of a graphical model. Given a set of nodes $I = \{i : i = 1, 2, ..., N\}$ corresponding to variables, another set of nodes $F = \{f : f = 1, 2, ..., M\}$ corresponding to factors and a set of undirected edges $E = \{(i, f) : i \in I, f \in F\}$, we define a factor graph to be the bipartite graph $\mathcal{G} = (V = (I, F), E)$. As notation for a graph description, we use V to indicate the set of vertices, or nodes, and E to indicate the set of edges.

The factor graph for a simple Ising model is depicted in figure 1.3.

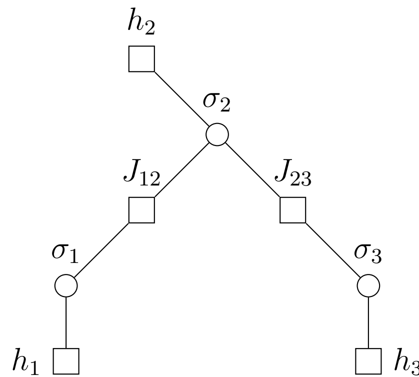


Figure 1.3: Factor graph for an Ising model defined on three variables σ_1, σ_2 and σ_3 with energy function $H(\sigma_1, \sigma_2, \sigma_3) = h_1\sigma_1 + h_2\sigma_2 + h_3\sigma_3 + J_{12}\sigma_1\sigma_2 + J_{23}\sigma_2\sigma_3$. Here we are slightly abusing notation by indicating variables and factors not with ordinal indices as it was stated in the definition of factor graphs, but with hopefully more meaningful names

Notice how the graph topology already specifies which pairs of spins are coupled by a $J \neq 0$ and which are not: J_{13} is implicitly equal to zero, which reflects on the absence of the corresponding factor.

In the special case where the distribution is factorized in such a way that the corresponding factor graph is a tree (it has no cycles), then marginals $p_i(x_i)$ and averages $\sum_{\mathbf{x}} A(\mathbf{x})p(\mathbf{x})$ can be computed exactly in polynomial time in N, while the cost for a naïve calculation would scale as q^N if q is the number of values each variable can take. Similarly, one can look for values of x_i which maximize probabilities, marginal or joint, and this can also be done in polynomial time on trees. In case the graph is not a tree, one can however resort to approximations.

At this point, a fair question would be: "does this have anything to do with optimization? And with bits?". The answer is yes to both questions: consider a linear system in involving a sparse matrix $H \in \mathbb{R}^{M,N}$ and a known vector $\mathbf{y} \in \mathbb{R}^M$. We look for a solution

x of

$$H\mathbf{x} = \mathbf{y} \tag{1.18}$$

Each of the M equations looks like

$$\sum_{n=1}^{N} h_{mn} x_n = y_m, \quad m = 1, 2, ..., M \tag{1.19}$$

where h_{mn} are the entries of H. The only non-zero terms at LHS are the ones with $h_{mn} \neq 0$, whose number decreases with the sparseness of H. Thus, we can re-write (1.19) as

$$\sum_{\substack{n=1,\dots,N \\ h_{mn} \neq 0}} h_{mn} x_n = y_m \tag{1.20}$$

Take an example with $M = 3, N = 4$, with the second equation highlighted, to stress that not all x variables appear:

$$\begin{bmatrix} h_{11} & 0 & 0 & h_{14} \\ h_{21} & h_{22} & 0 & h_{24} \\ 0 & h_{32} & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

The system of equations is

$$\begin{cases} h_{11}x_1 + h_{14}x_4 & = y_1 \\ h_{21}x_1 + h_{22}x_2 + h_{24}x_4 & = y_2 \\ h_{32}x_2 & = y_3 \end{cases}$$

Here the matrix topology is defining which variables appear in which constraints, just as the graph did before! Indeed, we can build a factor graph that encodes the system's structure, as shown in figure 1.4

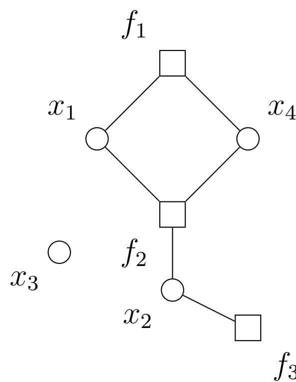


Figure 1.4: Factor graph for the example in (1.3). Factors f_1, f_2, f_3 represent equations $1, 2, 3$. Variable x_3 does not appear in any constraint, therefore it is disconnected from the rest of the graph.

Representing systems of constraints with factor graphs facilitates probabilistic approaches to the solution, both exact and approximate. Notable examples are found in the family of

compressed sensing algorithms, which look for solutions of undetermined linear systems conjoint with a set of additional constraints that typically involve one variable each. Such constraints can be easily incorporated in a factor graph representation: they behave just like Ising external fields.

As explained in chapters 2 and 3, data compression can be formulated in terms of a homogeneous linear system (on $\mathbb{GF}(2)$) together with a set of additional constraints and then solved by looking for the maximum of the joint probability over all variables.

1.4 Examples

Establishing a systematic and generally applicable mapping between problems from physics and optimization has probably never been done and is certainly beyond the writer's abilities. After all, physics itself has not yet been proven to be wholly adaptable to general principles and sometimes a different approach must be taken for each particular situation.

Here we present two instances of problems where a physics-optimization mapping is essential in finding a smart solution. The first one is in the $\mathcal{P}$ complexity class, meaning it can be solved in polynomial time; the second is $\mathcal{NP}$-hard.

1.4.1 Min-cut and ferromagnetic Ising ground state

This section is heavily inspired by Alfredo Braunstein's course notes from 2019.

We previously mentioned that an Ising model is said to be ferromagnetic if all couplings J_{ij} are non-negative. In the case where the fields h_i are all zero, there is no frustration in the system, meaning no conflict between constraints. There are two ground states: the one with all $\sigma_i = +1$ and the one with all $\sigma_i = -1$. In presence of non-zero fields, the situation is slightly more complicated, but still the ground state can be found in polynomial time by means of a mapping on a min-cut problem.

The min-cut problem is stated as follows: consider a graph $\mathcal{G} = (V, E)$, a *capacity* function $c : E \to \mathbb{R}_+$, a source vertex $s \in V$ and a sink vertex $t \in V$. Find a *cut* made of two disjoint subsets of V, S and T such that $S \cup T = V$, $s \in S$, $t \in T$ and

$$C(S,T) := \sum_{i \in S, j \in T} c(i,j) \tag{1.21}$$

is minimal. In words, this amounts to finding a cut through the graph that divides it into two parts, one containing the source and one containing the sink, in a way that minimizes the sum of the capacities of the cut edges. Min-cut is a paradigmatic problem in the area of linear programming; for a detailed introduction, see [Wolsey and Nemhauser, 1999, part 1.3].

Before giving the mathematical details of the mapping, let us informally outline the reasons why it works. Because of the ferromagnetic property, all pairs of spins are encouraged to point in the same direction to minimize the energy. In opposition to this, node-dependent fields are trying to align their spins to their direction, so whenever two neighbors have discordant fields, a conflict arises. It turns out that the best way to solve the trade-off is for spins to organize in clusters of same sign. Then the loss in energy of $2J_{ij}$ occurring whenever $\sigma_i \neq \sigma_i, (i,j) \in E$ is minimized by limiting the frontier between clusters of different sign. The above described process is, up to a couple of additional

details, equivalent to finding the minimum cut on the graph where the Ising model is defined and the capacities are related to fields and coupling.

More rigorously: given an Ising model on a graph $\mathcal{G} = (V, E)$ with energy

$$-H(\boldsymbol{\sigma}) = \sum_{1 \leq i < j \leq N} J_{ij}\sigma_i\sigma_j + \sum_{i=1}^{N} h_i\sigma_i, \quad J_{ij} \geq 0 \; \forall \; (i,j) \in E \tag{1.22}$$

build a capacity net $\{\mathcal{G}' = (V', E' = V' \times V'), c : E' \to \mathbb{R}_+\}$ as

$$\begin{aligned} V' &= V \cup \{s, t\} \\ c(i,j) &= J_{ij} && \text{if } i \in V, j \in V \\ c(s,j) &= h_j && \text{if } h_j > 0 \\ c(i,t) &= -h_i && \text{if } h_i < 0 \\ c(i,j) &= 0 && \text{otherwise} \end{aligned} \tag{1.23}$$

and consider the mapping $\boldsymbol{\sigma} \longleftrightarrow S, T$

$$\begin{aligned} S &= \{i \in V : \sigma_i = +1\} \cup \{s\} \\ T &= \{j \in V : \sigma_j = -1\} \cup \{t\} \end{aligned} \tag{1.24}$$

The capacity of the S, T cut is

$$C(S,T) = \sum_{i \in S, j \in T} c(i,j) \tag{1.25}$$

$$= \sum_{\substack{i \in S \setminus \{s\} \\ j \in T \setminus \{t\}}} c(i,j) + \sum_{j \in T} c(s,j) + \sum_{i \in S} c(i,t) \tag{1.26}$$

$$= \sum_{\substack{i<j \\ \sigma_i = -\sigma_j}} J_{ij} + \sum_{\substack{\sigma_j < 0 \\ h_j > 0}} h_j + \sum_{\substack{\sigma_i > 0 \\ h_j < 0}} -h_j \tag{1.27}$$

$$= \sum_{\substack{i<j \\ \sigma_i\sigma_j < 0}} J_{ij} + \sum_{\substack{i \in V \\ \sigma_i h_i < 0}} |h_j| \tag{1.28}$$

$$= \sum_{i<j} J_{ij} \, \mathbb{1}\left[\sigma_i\sigma_j < 0\right] + \sum_{i \in V} |h_i| \, \mathbb{1}\left[\sigma_i h_i < 0\right] \tag{1.29}$$

Using the identity

$$\mathbb{1}\left[x < 0\right] = \frac{|x| - x}{2|x|} \tag{1.30}$$

and noting that $|\sigma_i| = 1$, we have

$$C(S,T) = \sum_{i<j} J_{ij} \frac{1 - \sigma_i\sigma_j}{2} + \sum_{i \in V} |h_i| \frac{|h_i| - \sigma_i h_i}{2} \tag{1.31}$$

$$= \frac{1}{2}\left(-\sum_{i<j} J_{ij}\sigma_i\sigma_j - \sum_{i=1}^{N} h_i\sigma_i\right) + \text{const.} \tag{1.32}$$

Therefore, finally,

$$\arg\min_{S,T} C(S,T) \longleftrightarrow \arg\min_{\boldsymbol{\sigma}} H(\boldsymbol{\sigma}) \tag{1.33}$$

Once the mapping is performed, the problem of finding the minimum cut can by solved in polynomial time using the Ford-Fulkerson algorithm [Ford and Fulkerson, 1956].

1.4.2 p-XORSAT formulas and p-spin Ising

This example reports the content of the remarkable work by Mézard, Ricci-Tersenghi and Zecchina [Mézard et al., 2003].

The p-XORSAT problem, where SAT stands for "satisfaction" or "satisfiability", consists in finding an assignment for a set of binary variables $\{x_i\}_{i=1}^n$ such that they satisfy a list of given XOR clauses, also called parity checks. Each of the m checks looks like

$$x_{i_1^f} \oplus x_{i_2^f} \oplus ... \oplus x_{i_p^f} = y_f, \quad \forall f = 1, 2, .., m \tag{1.34}$$

where p is the fixed number of variables in each clause. An instance of p-XORSAT is completely identified by the indices $\{i_j^f : j = 1, 2, ..., p,\ f = 1, 2, ..., m\}$, specifying which x variables are involved in each constraint, and by the constant terms $\{y_f\}_{f=1}^m$, also in $\mathbb{GF}(2)$.

An example with $n = 8, m = 2, p = 3$:

$$(x_3 \oplus x_5 \oplus x_6) = 1$$
$$(x_1 \oplus x_5 \oplus x_8) = 0$$

It is not hard to realize that what we are doing is nothing but solving a linear system on $\mathbb{GF}(2)$ like (1.20) in the special case where the non-zero coefficients are all equal to 1 and appear in a fixed number p in each row. To determine whether the system has at least one solution, one can use Gaussian elimination, which runs in polynomial time (in n, m and p). If the system is found to be non-solvable, one can still be interested in the "least wrong" assignment, i.e. the one minimizing the number of unsatisfied constraints. This, however, happens to be (for $p > 2$) a $\mathcal{NP}$-hard problem: so far no algorithm has been able to solve it with 100% certainty in polynomial time. The number of unsatisfied checks is given by

$$\sum_{f=1,...,m} x_{i_1^f} \oplus x_{i_2^f} \oplus ... \oplus x_{i_p^f} \oplus y_f$$

since the summation argument gives 0 if the check is satisfied, 1 otherwise.

We saw that to each linear system modulo 2 there is an associated factor graph. The one for the previous example is shown in fig. 1.5.

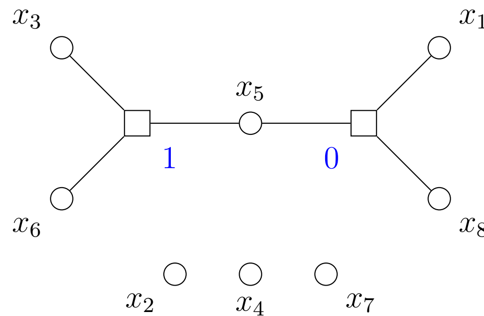


Figure 1.5: Factor graph for the example in (1.4.2). For each factor, the relative constant term is reported in blue

The next step, you guessed it, is to map p-XORSAT on an Ising model. Using mapping (1.14), we define a set of spin variables

$$\sigma_i = (-1)^{x_i}, \quad i = 1, 2, ..., n \tag{1.35}$$

and couplings

$$J_f = (-1)^{y_f}, \quad f = 1, 2, ..., m \tag{1.36}$$

Now a generic constraint f is satisfied if

$$\sigma_{i_1^f} \sigma_{i_2^f} \cdots \sigma_{i_p^f} = J_f \tag{1.37}$$

or, equivalently

$$\sigma_{i_1^f} \sigma_{i_2^f} \cdots \sigma_{i_p^f} J_f = 1 \tag{1.38}$$

We can define a Hamiltonian which counts the number of unsatisfied constraints

$$H(\boldsymbol{\sigma}) = \frac{1}{2} \sum_{f=1,\dots,m} \Big(1 - \sigma_{i_1^f} \sigma_{i_2^f} \cdots \sigma_{i_p^f} J_f \Big) \tag{1.39}$$

and look for the optimal configuration(s)

$$\boldsymbol{\sigma}^* \in \arg\min_{\boldsymbol{\sigma}} H(\boldsymbol{\sigma}) \tag{1.40}$$

If $H(\boldsymbol{\sigma}^*) = 0$, then there exists an assignment that solves the system. When finding the optimal assignment, we do not care about the $\frac{1}{2}$ factor not the constant $\sum_{f=1,\dots,m} 1 = m$. Therefore, for our purposes,

$$-H(\boldsymbol{\sigma}) = \sum_{f=1,\dots,m} \sigma_{i_1^f} \sigma_{i_2^f} \cdots \sigma_{i_p^f} J_f \tag{1.41}$$

This energy function looks vaguely like a zero-field version of (1.3), although not quite exactly. Here variables interact in p-tuples, whereas the standard Ising energy accounts only for pairwise couplings. An appropriate generalization is the p-spin Ising model, which takes into account many-body interactions

$$-H(\boldsymbol{\sigma}) = \sum_{f=1,\dots,m} \sigma_{i_1^f} \sigma_{i_2^f} \cdots \sigma_{i_p^f} J_f + \sum_{i=1}^{N} h_i \sigma_i \tag{1.42}$$

where in general $J_f \in \mathbb{R}, h_i \in \mathbb{R}$.

On a factor graph, p-wise interactions are represented by a factor node with p incident edges. Equivalently, one can generalize the concept of graph edge, which usually connects two nodes, to that of *hyperedge*, which connects an arbitrary number. In this case the graph is called a *hypergraph*.

Heuristics that work in polynomial time can be used to find approximate solutions to this hard optimization problem. One of the most convenient, due to its good trade-off between performance and ease of implementation, is Belief Propagation (BP) (more details in section 3.2), a well-known technique from statistical physics.

Rather than studying single instances of p-XORSAT, it is more interesting to look at ensembles of randomly-extracted ones, in the "thermodynamic" limit $n \to \infty, m \to \infty, \gamma = m/n \in [0, 1]$. Such instances are obtained by uniformly sampling p indices from the set $\{1, ..., n\}$ without replacement, m times. The components of $\mathbf{y}$ are also picked uniformly in $\{0, 1\}$. This ensemble features interesting properties, most notably a phase transition in the ratio of satisfiable instances to varying of the *density* γ.

At a certain value γ_c ($\gamma_c = 0.818469$ for $p = 3$), the ratio of solvable instances in the ensemble drops from 1 to 0, exponentially sharply as the system size grows to infinity.

(a) Factor node (b) Hyperedge

Figure 1.6: Factor node and hyperedge representation of a 3-way interaction. Hyperedges are represented with shaded polygons, in this case a triangle

This is called the SAT/UNSAT transition. If we were to make statistical mechanics considerations, we could say that γ is the state variable, the ratio of solvable instances is the order parameter and what happens at $\gamma = \gamma_c$ is a first order phase transition.

The ground state energy profile will be flat at zero for $\gamma < \gamma_c$, since instances in that regime are solvable with probability that tends to 1 as n tends to infinity. However, if one simulates the ensemble's behavior by extracting random instances of large enough size and looks for the ground state energy using BP, the result is what is shown in fig. 1.7.

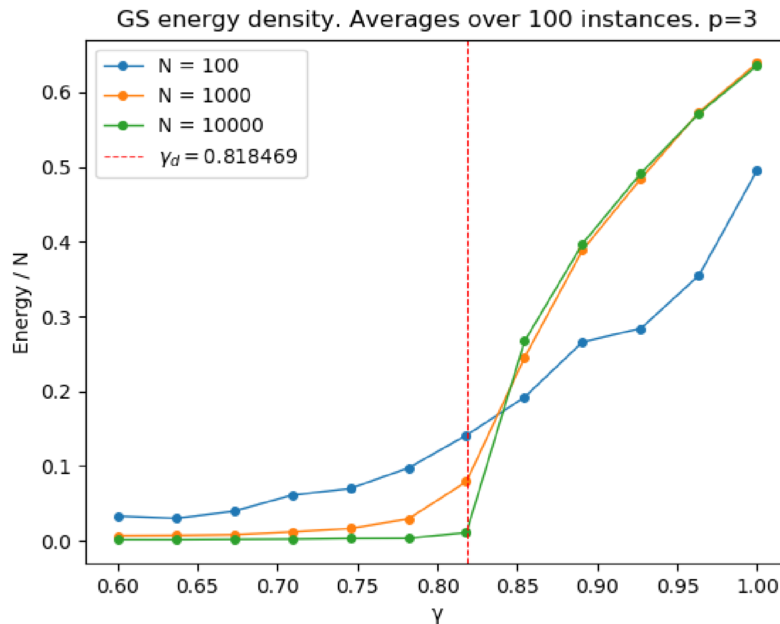


Figure 1.7: Ground state energy density approximated with BP vs. γ. The energy starts increasing before the expected value γ_c. Picture from `https://github.com/stecrotti/xorsat`

The energy is zero for small values of γ, but starts increasing at some $\gamma_d < \gamma_c$ where something called a *dynamic transition* happens. It turns out the reason for this unexpected behavior resides in how solutions are organized in the phase space $\{0, 1\}^n$: for $\gamma < \gamma_d$, all solutions are concentrated in one cluster, meaning that each one can be reached from any

other with a finite number of spin flips. Therefore BP, which is a *local* message-passing algorithm, is able to find solutions. Instead, for $\gamma_d < \gamma < \gamma_c$, systems are still solvable, but solutions are scattered into an extensive (with respect to N) number of clusters, therefore more sophisticated techniques such as Survey Propagation [Braunstein et al., 2005] are required in order to approximate the actual behavior.

Looking at the solutions of combinatorics problems represented in some space can give interesting points of view, as it will hopefully become clear in the next chapters.

Chapter 2

Lossy compression as constrained optimization

2.1 Problem framing

Given an input vector $\mathbf{y} \in \{0,1\}^n$, the goal is to generate a shorter vector $\mathbf{x} \in \{0,1\}^k$, $k < n$, from which an estimate $\hat{\mathbf{y}}$ for $\mathbf{y}$ can be recovered up to a certain distortion. The problem of finding a metric for the reconstruction error translates to that of finding some discrepancy measure between two vectors, in our case restricted to bit vectors. A natural choice is the Hamming distance

$$d_H(\mathbf{y}, \hat{\mathbf{y}}) = \sum_{i=1}^{n} y_i \oplus \hat{y}_i \tag{2.1}$$

which computes the number of differences between two vectors $\mathbf{y}$ and $\hat{\mathbf{y}}$, or alternatively the (minimum) number of flips needed to transform one into the other. Sometimes it is more handy to get rid of the dependency on the length n by working with the average Hamming distance, or the fraction of bits that differ

$$\overline{d_H}(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{n} \sum_{i=1}^{n} y_i \oplus \hat{y}_i. \tag{2.2}$$

We can now define the distortion to be the expected average Hamming distance between the original vector and the reconstructed one

$$D = \mathbb{E}\left[\overline{d_H}(\mathbf{y}, \hat{\mathbf{y}})\right]. \tag{2.3}$$

It is common practice to characterize the goodness of a compression scheme by considering another parameter in addition to the distortion: the compression rate

$$R = \frac{k}{n} \tag{2.4}$$

The rate simply quantifies how much the original length has been squished to obtain the compressed vector $\mathbf{x}$. The tuple R, D is a proper tool to establish the cost-effectiveness of a compression. But what values are to be considered good?

We can't proceed any further in answering this question without giving some information about the source that generated the input string. Since the pioneering work of Shannon [Shannon, 1948], many efforts have been devoted by information theorists to

exploiting the probability distribution of a source for efficient data compression. In general, as the source becomes more predictable, it gets easier and easier to achieve small or even zero distortions. For example, imagine that input bits coming from a distribution that outputs 1 with probability $p_0 \ll 1$ and 0 with probability $1 - p_0$. The fact that the source distribution is so unbalanced can be exploited effectively using, for example, arithmetic coding [MacKay, 2003].

However, here we are interested in the case where the source probability distribution is not accessible or very hard to characterize because, for example, it varies with time. To represent all these situations, here we look at the case of a symmetric source of independent bits. Formally,

$$y_i \sim \text{Bernoulli}\left(\frac{1}{2}\right) \quad \forall i = 1, ..., n. \tag{2.5}$$

This is a limit case in the sense that $\frac{1}{2}$ gives the maximum entropy for a binary source, i.e. the minimum predictability.

For a binary symmetric source, there exists a theoretical statement against which to compare results [Shannon, 1959], called the rate-distortion bound

$$R(D) = 1 - H(D), \quad D \in \left[0, \frac{1}{2}\right] \tag{2.6}$$

where $H(D)$ is the binary entropy $H(D) = D \log_2\left(\frac{1}{D}\right) + (1 - D)\log_2\left(\frac{1}{1-D}\right)$. It does not make sense to talk about distortions greater than $\frac{1}{2}$, since that is the average reconstruction error one would get by discarding the input and randomly extracting another vector to use as output. A similar reasoning can be done jointly on R, D by observing that a naïve compression technique could be: take $\mathbf{x}$ to be the first k bits of $\mathbf{y}$ and then append $n - k$ random values to form $\hat{\mathbf{y}}$. k bits will obviously be reconstructed correctly, while there is a $\frac{1}{2}$ chance of error for each remaining bit, giving

$$\begin{aligned} D &= \frac{1}{2}\frac{n-k}{n} = \frac{1}{2}\left(1 - \frac{k}{n}\right) \\ R &= \frac{k}{n} \end{aligned} \tag{2.7}$$

and finally

$$R = 1 - 2D. \tag{2.8}$$

Shannon's theoretical bound and the "naïve compression" line will be used from now on as a comparison when assessing the goodness of results. A picture is shown in fig. 2.1.

2.2 Taking inspiration from LDPC codes

It is interesting at this point to notice that the problem of data compression, or *source coding* in the information theory lingo, is somewhat dual to that of *channel coding* (often called *error correction*). Channel coding is perhaps *the* fundamental topic in information theory, and it faces the issue of ensuring a reliable communication of data over a noisy channel. This section takes a detour into channel coding and the technique of Low-Density Parity-Check (LDPC) codes, hopefully motivating and explaining the origin of the compression strategy that is being presented later in this work.

For a more detailed introduction to channel coding and LDPC codes see [MacKay, 2003, parts II, VI]

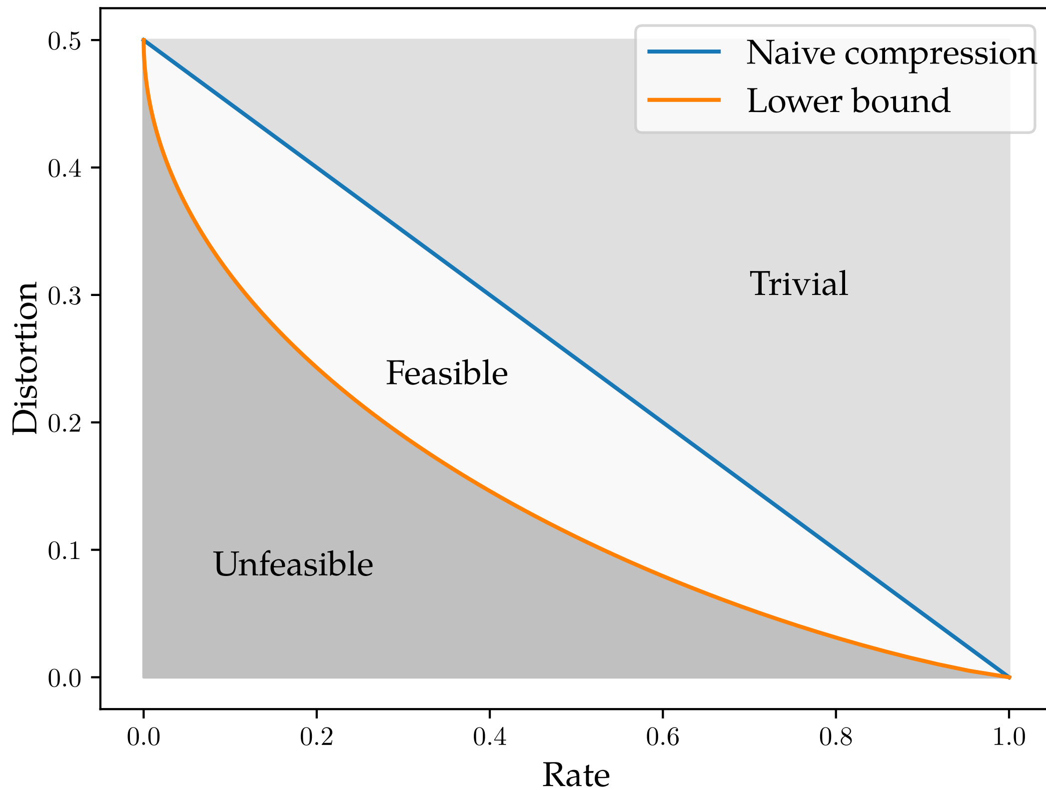


Figure 2.1: The (Rate, Distortion) space of compression codes

2.2.1 Channel coding

Suppose you have a source that generates some data $\mathbf{s}$, in our case the usual string of bits, and these are then transmitted over a channel which might corrupt the information, namely flip some of the bits. The channel output $\mathbf{r}$ is finally collected by the receiver, which will get the original message, affected by some error.

The idea behind source coding is to add some redundancy to the message before transmitting it in a clever way so that the receiver can detect and correct as many errors as possible. The input $\mathbf{s}$ is therefore transformed according to some strategy called a "code" (hence the name channel coding) into some other string $\mathbf{t}$ to be transmitted over the channel. The receiver then tries to infer the original message by means of a decoding algorithm paired to the encoding one and makes a guess which typically maximizes some probability $p(\mathbf{s}|\mathbf{r})$ of $\mathbf{s}$ having been sent given that $\mathbf{r}$ was received.

As for the case of compression, the ability of a coding strategy to maximize the reconstruction capability cannot be the only feature that one takes into account. Take as an example the following code: for each bit in $\mathbf{s}$, replicate it 100 times when constructing $\mathbf{t}$. For instance, the string $[0, 1, 1]$ will be encoded as

$$[0, 1, 1] \quad \rightarrow \quad [\underbrace{0, ..., 0}_{\text{100 times}}, \underbrace{1, ..., 1}_{\text{100 times}}, \underbrace{1, ..., 1}_{\text{100 times}}]$$

Noise might flip some of the bits but it is clear that this strategy is robust: the receiver can look at 100 bits at a time and guess the original value based on which digit appears most frequently. If a block contains say 80 '1's and 20 '0's, the receiver will guess 1. An error will

be made only if more than half of the transmitted bits were flipped during transmission, which is very unlikely if the noise is reasonably small.

However, what prevents us to add repetitions at will is that communication becomes less and less efficient as the channel gets crowded with redundant information, namely the *rate* at which data are transmitted drops. Ultimately, in analogy to what happens with compression, the performance of a channel coding algorithm should be assessed based on both the reconstruction fidelity and the transmission rate.

2.2.2 Parity-check codes

A more sophisticated family of approaches falls under the name of parity-check codes. The idea is to append to the message $\mathbf{s} \in \{0,1\}^n$, p extra bits containing information about combinations of the bits in $\mathbf{s}$.

$$\mathbf{s} = \begin{bmatrix} s_1 \\ s_2 \\ \vdots \\ s_n \end{bmatrix} \quad \mathbf{t} = \begin{bmatrix} s_1 \\ s_2 \\ \vdots \\ s_n \\ t_{n+1}(s_1, ..., s_n) \\ \vdots \\ t_{n+p}(s_1, ..., s_n) \end{bmatrix}$$

The receiver, which shares knowledge about this process, will then check if those extra bits indeed correspond to the right combinations of source bits and any mismatch is labeled as an error. The simplest combination of source bits, which is also responsible for the origin of the name "parity check", is the sum of all of them

$$t_{n+1} = \bigoplus_{i=1}^{n} s_i \tag{2.9}$$

In this case, the single extra bit is called a parity bit because it equals 0 if $\mathbf{s}$ contains an even number of '1's and 1 otherwise. The set of p equations relative to the chosen combinations are called "check equations" or simply "checks". The extra bits are called "check bits", while the ones coming from the original message are called "information bits". The total length $n + p$ is the code's "block length".

The receiver can notice that there was some corruption on the channel if in the received vector $\mathbf{r}$ one or more parity equations are not satisfied. This information, however, is not enough on its own. Take the previous example with a single parity bit: what if 2 or 4 or any even number of source bit get corrupted? The sum will be insensitive to such changes and no error will be raised. We need to increase the number and the complexity of the parity equations in such a way to maximize the receiver's ability to spot and correct errors. Moreover, say one or more constraints are not satisfied: is it because some of the source bits were corrupted, or maybe the parity bit itself was? If there is no hint on where the error (or errors) was generated, there will be no hope to correct it.

In facing these issues, it is useful to begin by introducing some definitions. A code is called "linear" if parity bits are obtained through linear combinations of elements in $\mathbf{s}$. In

this case, **t** is obtained by matrix multiplication as

$$\begin{bmatrix} t_1 \\ t_2 \\ \vdots \\ t_{n+p} \end{bmatrix} = G^T \begin{bmatrix} s_1 \\ \vdots \\ s_n \end{bmatrix} \tag{2.10}$$

where $G \in \{0,1\}^{n,n+p}$ is called generator matrix and p is the number of parity bits. This process is called "encoding".

In our example with just one parity bit where the last element in **t** is the sum of **s**, we have

$$G^T = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & \ddots & \\ & & & 1 \\ 1 & 1 & \cdots & 1 \end{bmatrix} \tag{2.11}$$

Once we restrict ourselves to linear transformations, and since we are working with numbers in base 2, there are actually not many operations left allowed! The coefficients of such linear combinations can only be 0 or 1, which means that any check bit is obtained as the parity of a certain subset of the information bits

$$t_{n+j} = \bigoplus_{i \subset \{1,\dots,n\}} s_i, \quad \text{for some } j \in 1,\dots,p \tag{2.12}$$

It is important to stress that the p extra bits are redundant: each one of the 2^n possible source vectors gets mapped onto one **t** vector. Therefore, the number of valid **t** vectors is still 2^n, even though there are in principle 2^{n+p} vectors of length $n+p$.

Message **t** then travels through the channel where it suffers some perturbation after which it turns into **r**. The receiver starts the decoding phase by a parity-check operation that consists in controlling that all parity relations are satisfied. For linear codes this is again performed through multiplication of the received vector **r** with a matrix H, paired to G. The resulting vector **z** is called syndrome

$$\begin{bmatrix} z_1 \\ \vdots \\ z_p \end{bmatrix} = H \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_{n+p} \end{bmatrix} \tag{2.13}$$

and $H \in \{0,1\}^{p,n+p}$ is called parity-check matrix. For our simple example H is just a row vector of $n+1$ '1's

$$H = \begin{bmatrix} 1 & \cdots & 1 \end{bmatrix} \tag{2.14}$$

We say that parity is fulfilled if

$$\mathbf{z} = H\mathbf{r} = \mathbf{0} \tag{2.15}$$

which means that **r** satisfies all the parity constraints. This might be happening because no corruption happened, meaning that **r** is equal to the transmitted vector **t**, which satisfies parity by construction. Or, the message might have been perturbed in the channel in such

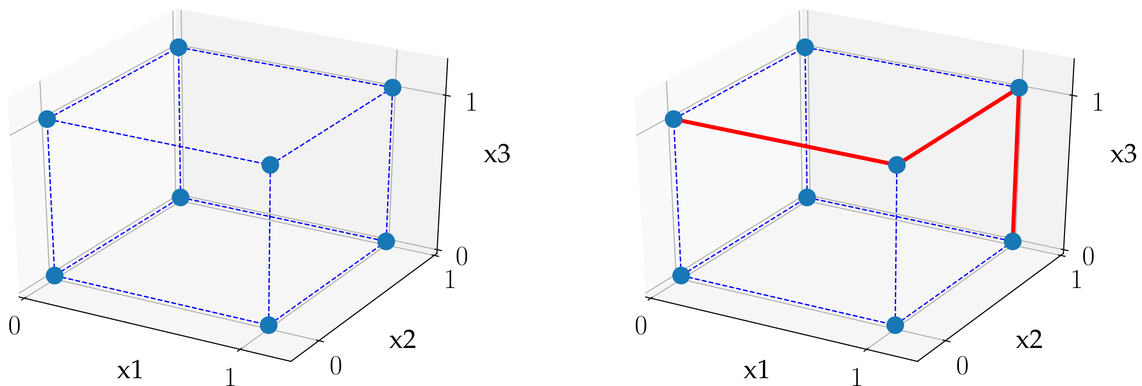
a way that all errors canceled each other exactly, so that **r** happens to fall in the set of possible transmitted vectors, although it is different from the one that was sent.

The vectors that satisfy all parity constraints are given a special name: codewords. Stated alternatively, we define codewords to be the elements of the nullspace (or kernel) of H. If H is full-rank, than such space will have dimension $n + p - p = n$, which is consistent with the fact that input vectors have length n. Again, G and H are built so that by construction every transmitted vector is a codeword, while in general received vectors will not, due to noise in the channel. In the following, we will assume that G and H are paired in the correct way and H is full rank.

In more complex codes, parity bits are not necessarily appended at the end of the source signal (which would imply G^T and H to contain an identity block). Codewords may be built as any linear function of the information bits.

2.2.3 Codewords as points in space

To understand the principles behind LDPC codes it can be useful to adopt a new point of view when looking at messages. We can think of a bit string, or vector, of length n as a point living in an n-dimensional space where each coordinate takes value in $\{0, 1\}$. Another way to state the same is that all bit vectors with a certain fixed length can be identified with the vertices of a hypercube with dimension equal to the vector length. Clearly, the number of possible vectors is 2^n, just like the number of vertices of the hypercube.



(a) Vertices of a hypercube of dimension 3 (a cube)

(b) Path of length 3 connecting $[1, 1, 0]$ to $[0, 0, 1]$

Figure 2.2

Each point has got n nearest-neighbors ($n = 3$ for the example in fig. 2.2), that is points that are reachable by a straight path of length 1. Such paths give us a notion of how close or far apart two points are in this space. We define the distance between any two vertices as the (minimum) number of "nearest-neighbor" steps needed in order to go from one to the other.

Notice that taking a step of length one as described above is equivalent to flipping a bit in the vector seen in string form, since one component changes its value from 0 to 1 or vice-versa. The analogy is completed by the observation that the distance that we just defined is nothing but the Hamming distance (2.1). This means that the Hamming

distance between two bit strings of same length corresponds to the number of hypercube edges traversed by the shortest path composed of nearest-neighbor hops.

Let us visualize the process of parity generation and check described in 2.2.3 in space. For the sake of representation, in the following all points are drawn in two-dimensional spaces, but should be imagined as living on the above mentioned hypercube vertices.

We said previously that source vectors fill the space of all possible vectors of length n, in the sense that any vector of such length might have been originated by the source. Let us call this set S: formally,

$$S = \{\mathbf{s} : \mathbf{s} \in \{0, 1\}^n\}, \quad |S| = 2^n$$

Each of these vectors will have a counterpart represented on $n + p$ bits, living in the set of codewords T

$$T = \{\mathbf{t} : \mathbf{t} = G^T\mathbf{s}, \mathbf{s} \in \{0, 1\}^n\}, \quad |T| = 2^n$$

or, equivalently

$$T = \{\mathbf{t} : H\mathbf{t} = 0, \mathbf{t} \in \{0, 1\}^{n+p}\}, \quad |T| = 2^n$$

where G and H are a pair of suitable generator and parity-check matrices. T is in turn a subset of all possible vectors of length $n + p$, which is the space where received vectors live. Let us call it R

$$R = \{\mathbf{r} : \mathbf{r} \in \{0, 1\}^{n+p}\}, \quad |R| = 2^{n+p}$$

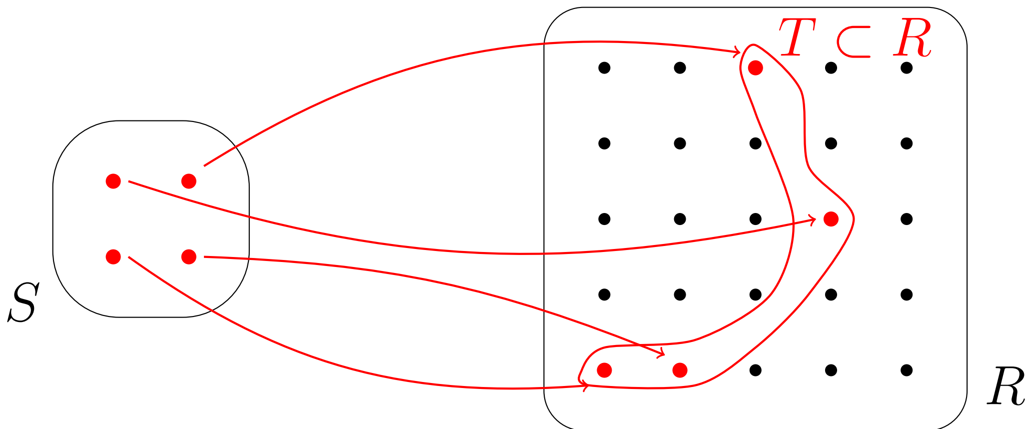


Figure 2.3: Encoding of source vectors from S to T. Red dots indicate codewords, black dots indicate generic received vectors

The effect on **t** of the channel noise is to flip some of the bits. Under some reasonable assumptions on the noise, the probability that a number of bits get flipped decreases with the number itself (This is easy to show for example in the case of Binary Symmetric Channel [MacKay, 2003], where bits get flipped independently with a fixed probability). Translated to the space point of view, this means that the received vector will be somewhere in R, likely somewhere close to **t**, since the probability of many nearest-neighbor hops (corresponding to bit flips) is low.

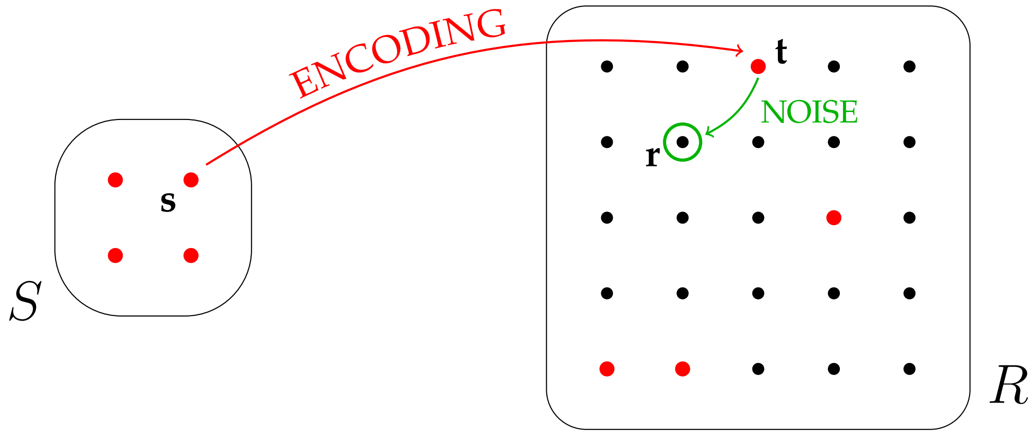


Figure 2.4: Effect of channel noise on the transmitted vector: the number of bit flips is small (just 1). Red dots indicate codewords, black dots indicate generic received vectors

Based on these visualizations, here is a proposition for an efficient coding strategy: build the encoded messages $\mathbf{t} \in T$ to be as spread out as possible, so that there are many non-codewords (black dots) between any pair of codewords (red dots). A small perturbation will place the received vector on some black dot somewhere close to **t** but typically still sufficiently far from any other codeword. Build the decoder so that, given a received vector **r**, the guess for the transmitted message is the closest codeword

$$\mathbf{t}^* = \arg\min_{\mathbf{t} \in T} d_H(\mathbf{t}, \mathbf{r}). \tag{2.16}$$

Once $\mathbf{t}^*$ is found, the estimate for the source vector is straightforward, since the mapping from S to T is one-to-one.

The operations needed for the strategy just described are indeed very effective but perhaps more tricky than they seem when it comes to practical implementation. The reason is twofold:

1. The problem of generating a set of "spread out" codewords is not trivial. One could think about maximizing the sum of distances between all pairs but that sounds like a nasty optimization task. Instead, the solution provided by Shannon, which is proved to be the optimal one, exploits some statistical properties of hypercubic spaces as their dimension grows very large. In other words, such optimal performance can be reached asymptotically as $n \to \infty$, $p \to \infty$, p/n finite.

2. In general, finding the minimum in (2.16) requires an extensive search over all possible $\mathbf{t} \in T$ in order to be carried out exactly. The computational cost grows as 2^n, which soon becomes prohibitive if we wish to work with high dimensions, therefore some kind of approximation is needed.

Long codes and a polynomial-time approximation are the key ingredient to Low-Density Parity-Check codes, first introduced by Gallager in the early '60 [Gallager, 1962].

2.2.4 Low-Density Parity-Check codes

Low-Density Parity-Check (LDPC) codes are parity-check codes of large block length whose defining matrix H is sparse, meaning it contains many 1's and few 0's, hence the

low density. In their first version, LDPC matrices were "regular": constructed so that each check involved a fixed number k of digits and each digit appeared in a fixed (smaller) number j of checks. Later, it was found that performances can be improved by choosing irregular codes, where j and k are not fixed (see for example [Luby et al., 2001]). For H to be sparse, both j and k must be small, typically less than 10.

It was proved that optimal codes are achievable with matrices H whose entries are all extracted randomly, therefore not sparse. However, the advantage brought by sparseness in terms of decoding feasibility is large enough to compensate for such lack of optimality.

Gallager pointed out that, since the properties of single codes are hard to analyze, it is wiser to consider an ensemble of such codes constituted by the set of some permutations of the 1's in the parity-check matrix (details in [Gallager, 1963]). He used the fact that, for a code picked at random from the ensemble, the Hamming distance between the two nearest codewords is a non-zero fraction of the block length, with probability that goes to 1 as the block length grows. Sets of codewords built in this way indeed meet the previously mentioned requirement of being nicely spread-out.

We can rewrite (2.16) in the form of a constrained optimization problem using the definition for the set of codewords T

$$\begin{aligned} \mathbf{t}^* &= \arg\min_{\mathbf{t}:H\mathbf{t}=0} d_H(\mathbf{t}, \mathbf{r}) \\ &= \arg\min_{\bigoplus_{j=1}^{n+p} h_{ij} t_j} \sum_{i=1}^{n+p} t_i \oplus r_i \end{aligned} \tag{2.17}$$

where h_{ij} are the entries of H.

It is worth emphasizing that (2.17) is the core equation of this whole work: it constitutes the hardest step in the compression procedure, it is explicitly stated as an optimization problem, it involves $\mathbb{GF}(2)$ operations and it features a parity-check matrix.

The fact that each parity constraint involves only a small number of digits, paves the way for local algorithms to heuristically look for a solution. The one described by Gallager, perhaps at that time without even his knowledge, is nothing but the Belief Propagation approach well-known in the statistical mechanics community. A key step in the process of finding an approximate solution is to work not directly on matrix H but on the corresponding factor graph (see section 1.3 for details about the link between linear systems and factor graphs).

As all the ingredients for the channel coding - data compression analogy have now been laid down, we can go back to the latter to study it in more detail.

2.3 A compression scheme

While channel codes expand the source vector into a higher-dimensional space, the aim of compression is the opposite: reduce the number of digits. The reconstruction step, vice-versa, brings the dimension back to the original one, i.e. reduction or expansion for channel and source coding respectively. This suggests that decoding strategies for channel coding, namely LDPC codes, can be exploited to design the encoding stage of a compressor. The compression strategy that is being described here is the one presented in [Braunstein et al., 2011].

Let us call $\mathbf{s} \in \{0, 1\}^n$ the source vector coming from a memory-less binary symmetric source. The idea is to pick a subset of the 2^n possible source vectors as "representatives"

for all the rest: each time an input is presented, the encoder does not forward it but its representative is sent instead. Say that there are $2^k, k < n$ representatives: in order to communicate any of them, we do not need n bits anymore. It is enough to send a number in $\{0, ..., 2^k - 1\}$ that identifies it and such value can be represented on just k bits, which will constitute the transmitted message $\mathbf{t} \in \{0,1\}^k$. The 2^k representatives expressed in their *long* form, i.e. still embedded in the n-dimensional space, are called, in analogy to what was said before, codewords.

Finally, the decoder has the easiest job: it suffices to look up which codeword corresponds to $\mathbf{t}$, that will be the guess $\hat{\mathbf{s}}$ for what the source vector was.

It is clear that the described strategy implies a loss of information, namely in the stage where the input is replaced by a codeword, since clearly more than one input will (at least on average) correspond to the same codeword. It should be no surprise that in this case compression has to be lossy, since it is well known that a memory-less symmetric source emits digits with maximum entropy (1 bit/symbol).

2.3.1 Encoding - part one

What is needed at this point is a good set of codewords together with the mapping from the source vector space. The analogy with the decoding phase of a LDPC code suggests a similar strategy: map each input vector onto the nearest of the codewords, which have been picked as spread out as possible. In this way, reconstruction error will correspond to the distance between source vector and codeword which is always kept fairly small.

As we have seen in 2.2.4, sets of codewords with the above mentioned property can be obtained as the solutions of homogeneous linear system modulo 2 defined as

$$C = \{\mathbf{c} : H\mathbf{c} = 0, \mathbf{c} \in \{0,1\}^n\}$$

where $H \in \{0,1\}^{n-k,n}$ is called a parity-check matrix. Again, we will in general assume H to be full-rank so that $dim(C) = k$. The size of H determines the compression rate, which is the ratio of lengths between original and compressed vector

$$R = \frac{k}{n} \tag{2.18}$$

In complete analogy to what happens for LDPC codes (compare (2.17)), the encoding step is performed by finding the nearest codeword to the source vector

$$\begin{aligned} \mathbf{c}^* &= \arg\min_{\mathbf{c}:H\mathbf{c}=0} d_H(\mathbf{c}, \mathbf{s}) \\ &= \arg\min_{\bigoplus_{j=1}^{n} h_{ij}c_j = 0} \sum_{i=1}^{n} c_i \oplus s_i \end{aligned} \tag{2.19}$$

where h_{ij} are the entries of H. If H is sparse, the minimization can be attempted via local algorithms of polynomial complexity such as Belief Propagation.

Building H and solving for the minimum are far from being trivial problems; the latter in particular is computationally characterized as *hard*, a feature shared with many various similar tasks in constrained optimization. The study of an efficient solving algorithm is the essential content of the rest of this work from chapter 3 on, while details on how to build H are given in section 3.1.

2.3.2 Encoding - part two

Suppose somehow **t** was found, the difficult part is over! Since the space of codewords C is obtained as a kernel, it is by definition a linear vector space. Despite being embedded in a n-dimensional space, we saw that it has dimension $k < n$, therefore one can find k basis vectors and express any $\mathbf{c} \in C$ as a linear combination of those. For any choice of **c**, the coefficient of such expansion will constitute a list of k bits which works as a unique identifier: a perfect choice for the compressed vector **t**.

2.3.3 Decoding

The decoder must share knowledge about which basis was chosen for C. Once this requirement is met, it is enough to re-expand **t** on the basis into $\mathbf{c}^*$

$$\mathbf{c}^* = \bigoplus_{i=1}^{k} t_i \mathbf{b_i} \tag{2.20}$$

where $\{\mathbf{b_i}\}_{i=1}^{k}$ are the n-dimensional basis vectors.

Chapter 3

Methods

This chapter describes in detail the methods used and the corresponding code implementation.

In section 3.1, a family of factor graphs known as Tanner graphs is presented: as mentioned in the previous chapter, they are used to build parity-check matrices. Then, in section 3.2, the Belief Propagation (BP) equations are stated, in their finite and infinite temperature version. For the latter case, an original proof is given in section 3.3 for the algorithm exactness under some conditions. Some details of the proof suggest that removing some factors from the graph to expose more leaves might help the performance of BP. The matter is explored in section 3.4.

It was found in the context of LDPC codes that it can be advantageous to generalize the field on which variables live from $\mathbb{GF}(2)$ to $\mathbb{GF}(q)$: in section 3.5 we describe how the same strategy can be applied to data compression. Finally, in section 3.6, we describe how BP can be efficiently implemented by means of convolutions, which are briefly reviewed together with their infinite temperature limit.

All code is written in Julia Language.

3.1 Irregular graphs

It was mentioned earlier that the encoding step of our compression method is based on sparse Low-Density Parity-Check (LDPC) matrices. This section provides a description of a strategy for building matrices that are good for both channel coding and source coding. In the following, the term "code" is used to generically indicate any of the two.

The idea is to build a factor graph according to some recipe and then once again exploit the correspondence with linear systems of constraints to obtain a parity-check matrix. Once the optimization problem (2.17) is expressed in terms of a factor graph, we can resort to message-passing solvers, namely the Belief Propagation algorithm.

See [Loeliger, 2004] for an introduction on factor graphs and section 1.3 for a description of their relation with systems of constraints.

In the literature, factor graphs used for error correction are sometimes called Tanner graphs.

3.1.1 Theory

Both channel coding and data compression reach their best performances when codewords are as spread out as possible. Since codewords are the solutions of the homoge-

neous linear system defined by a matrix H

$$C = \{\mathbf{c} : H\mathbf{c} = 0, \mathbf{c} \in \{0,1\}^n, H \in \{0,1\}^{n-k,n}\} \tag{3.1}$$

building a good code eventually boils down to picking a good H. It was shown [Shannon, 1957] that optimal performance is achievable, for large n and k, by extracting each entry of H uniformly at random. Unfortunately this makes decoding (2.17) (respectively encoding (2.19)) for error correction (respectively data compression) a difficult task to tackle. At least so far, the most successful algorithms, LDPC being the most notable example, have been based on sparse matrices, which allow for local message-passing decoding (encoding) algorithms. The advantage brought by computational tractability compensates for the lack of optimality.

Randomness still helps, also in the case of sparse parity-check matrices. In his original paper [Gallager, 1962], Gallager proposed to use matrices with a certain number k of 1's in each row and a fixed number j of 1's in each column. Codes with fixed j and k are called *regular*. For fixed j, k and block length n, one can define an ensemble of matrices, whose statistical properties can be studied. Gallager proved that the minimum distance between two codewords is greater than a finite fraction of n, with probability that tends to 1 as the size increases. There is no evidence of non-random codes with the same property.

The correspondence between systems of constraints and factor graphs, which is mostly useful to exploit message-passing optimization algorithms, can also be employed to build parity-check matrices: it turns out to be more handy to randomly extract graph edges rather than matrix elements. The number k of 1's in each row is nothing but the number of variables involved in each check, j is the number of checks any variable appears in. In graph theory jargon, j and k are the *degrees* of variable and check nodes, respectively. An example of regular parity-check code is shown in figure 3.1. The factor graph representation emphasizes the bipartition between variable and check nodes.

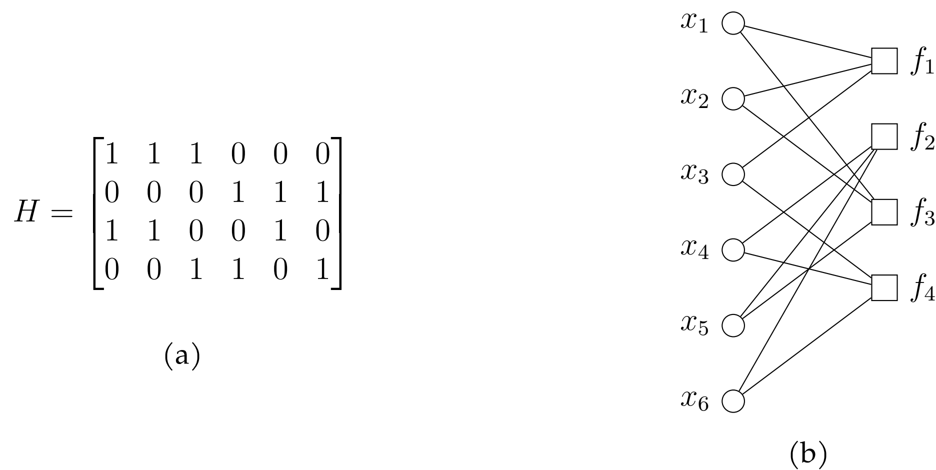


Figure 3.1: Regular parity-check code with $n = 6$, $j = 2$, $k = 3$ in matrix and factor graph form.

Factor graphs underlying regular codes are called *regular* graphs. It was found [Luby et al., 2001] that better performances can be achieved by letting j and k vary according to some distribution, thus obtaining and *irregular* graph.

There are several ways of specifying an ensemble of irregular graphs. Here we will adopt the one in [Luby et al., 2001], which is based on edge degrees. Let us define a vector $\boldsymbol{\lambda} = (\lambda_1, \lambda_2, ..., \lambda_{d_V})$ where λ_i is the fraction of edges incident on a variable node of degree i

and d_V is the maximum of such degrees. Analogously, we define $\boldsymbol{\rho} = (\rho_1, \rho_2, ..., \rho_{d_F})$ with ρ_j being the fraction of edges incident on a check node of degree j and d_F the maximum degree. The above definitions imply a few identities, involving n, the number of variables, m, the number of checks, and $|E|$, the number of edges

$$\sum_i \lambda_i = \sum_j \rho_j = 1 \tag{3.2}$$

$$\sum_i \frac{\lambda_i}{i}|E| = n \tag{3.3}$$

$$\sum_j \frac{\rho_j}{j}|E| = m \tag{3.4}$$

The compression rate $R = \frac{n-m}{n}$ is also determined once $\boldsymbol{\lambda}$, $\boldsymbol{\rho}$ and $|E|$ are given.

It can admittedly be uncomfortable to work with edge degrees instead of the usual node degrees, but it will turn out to be useful in the graph construction. However it is relatively simple to switch to node degrees by noting that

$$\frac{\lambda_i}{i}|E| \tag{3.5}$$

is the number of variable nodes with degree i and

$$\frac{\rho_j}{j}|E| \tag{3.6}$$

is the number of check nodes with degree j.

As an example, if one has

$$\boldsymbol{\lambda} = \begin{bmatrix}\lambda_1 & \lambda_2 & \lambda_3\end{bmatrix} = \begin{bmatrix}\frac{1}{2} & \frac{1}{4} & \frac{1}{4}\end{bmatrix}$$

this means that half of the variables have degree 1, namely they are leaves, $\frac{1}{4}$ have degree 2 and another $\frac{1}{4}$ have degree 3.

Optimal choices for $\boldsymbol{\lambda}$ and $\boldsymbol{\rho}$ depend on the context.

3.1.2 Graph construction

The recipe given in [Luby et al., 2001], it is not the only possibility, for constructing a bipartite graph given $\boldsymbol{\lambda}$, $\boldsymbol{\rho}$ and the number of edges $|E|$, goes as follows

1. Visualize the bipartite graph with n variable nodes on the left and m check nodes on the right
2. Make two copies of the edge set E' and E'' and number the edges in both from 1 to $|E|$
3. Assign each edge in E' to a left node in a way that respects $\boldsymbol{\lambda}$. Do the same with E'' and the right nodes, respecting $\boldsymbol{\rho}$. This can be done in any non-random way, for example assigning blocks of contiguous edges to the same node

At this point, each variable node (left) is assigned to an edge and so is every check node (right). All that is left to do is to identify each edge in E' to one in E'' in such a way that the final result is a uniform draw from the space of all possible graphs in the $(\boldsymbol{\lambda}, \boldsymbol{\rho}, |E|)$ ensemble.

4. Permute the edges in E'' via a uniform random permutation $\pi : \{1, 2, ..., |E|\} \rightarrow \{1, 2, ..., |E|\}$
5. Put E' and E'' in correspondence by assigning i in E' to $\pi(i)$ in E'': connect the variable assigned to i to the check assigned to $\pi(i)$ at point 3.

It must be pointed out that this procedure can lead to the formation of multi-edges, if the same pair variable-check is connected more than once. Multi-edges can simply be discarded, leading to a slight change in the final $\boldsymbol{\lambda}$, $\boldsymbol{\rho}$ and $|E|$, with no harmful consequences. Alternatively, one can repeat the random extraction, or a part of it, until no multi-edges are found.

3.1.3 Choice of $\boldsymbol{\lambda}$ and $\boldsymbol{\rho}$

Concerning the choice of $\boldsymbol{\lambda}$ and $\boldsymbol{\rho}$, for the purpose of data compression it can be useful to work with only node variables of degree 1 or 2 by setting $\lambda_i = 0 \, \forall \, i > 2$. In this way, each variable is involved in at most two checks, a fact which is crucial for proving the theoretical results in section 3.3. Let us start with $\boldsymbol{\lambda} = \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \end{bmatrix}$: all variables have degree two. Degree-one variables will possibly appear due to multi-edge discard or the factor-removing procedure described later in section 3.4.

The problem of choosing $\boldsymbol{\rho}$ and $|E|$ given the number of variables n and the rate (which in turn determines the number of factors m) implies a good deal of freedom. However, the above mentioned constraints (3.2), (3.3) and (3.4) must be met, together with the fact that most of the quantities in question are forced to be positive integers: number of nodes, degrees, etc. Here we present a practical strategy to produce valid $\boldsymbol{\rho}$ and $|E|$, based on the simplifying assumption that the distribution $\boldsymbol{\rho}$ is non-zero only for two contiguous indices r and $r + 1$.

First, let us explicit (3.3) for our choice of $\boldsymbol{\lambda}$. We get

$$\frac{|E|}{2} = n \tag{3.7}$$

which fixes the number of edges to $|E| = 2n$. Next, if we call $N_j = \rho_j |E|$, we have that, from (3.4),

$$m = \frac{N_r}{r} + \frac{N_{r+1}}{r+1} \tag{3.8}$$

and, from (3.2),

$$N_r + N_{r+1} = |E| \tag{3.9}$$

Substituting back in (3.8), we have

$$m = \frac{N_r}{r} + \frac{|E| - N_r}{r+1} \tag{3.10}$$

$$= N_r \left(\frac{1}{r\,(r+1)} \right) + \frac{|E|}{r+1} \tag{3.11}$$

Therefore, we get that for N_r

$$N_r = r\,(r+1) \left(m - \frac{|E|}{r+1} \right) \tag{3.12}$$

$$= r\,[(r+1)\,m - |E|] \tag{3.13}$$

which automatically fulfills $N_r \in \mathbb{Z}$. The only thing to ensure is that it is positive, namely

$$(r+1)\, m - |E| \geq 0 \tag{3.14}$$

which constitutes a lower bound on r.

For N_{r+1},

$$\begin{aligned} N_{r+1} &= |E| - N_r & (3.15) \\ &= (r+1)\,(|E| - rm) & (3.16) \end{aligned}$$

Similarly as before, we must enforce

$$|E| - rm \geq 0 \tag{3.17}$$

which constitutes an upper bound on r. Intersecting (3.14) and (3.17), we get

$$\frac{|E|}{m} - 1 \leq r \leq \frac{|E|}{m} \tag{3.18}$$

Finally, we combine this with the fact that r must be an integer by picking

$$r = \left\lceil \frac{|E|}{m} - 1 \right\rceil \tag{3.19}$$

where $\lceil\rceil$ is the ceiling function that rounds up to the nearest integer. There is one caveat: if n is an integer multiple of m, then $N_r = 0$ and all factor nodes have degree $r+1$.

Once the right value for r is found, one can compute N_r and N_{r+1} to obtain $\boldsymbol{\rho}$. A simple computer program is shown in algorithm 1

```
1   function generate_polyn(n::Int, m::Int)
2
3       # Lambda and number of edges
4       lambda = [0.0, 1.0]
5       nedges = 2*n
6       # Find the right r
7       r = Int(ceil(nedges/m - 1))
8       # Initialize and fill rho
9       rho = zeros(r+1)
10      rho[r] = r*((r+1)*m - nedges)
11      rho[r+1] = (r+1)*(nedges - r*m)
12      # Normalize
13      rho ./= nedges
14
15      return nedges, lambda, rho
16  end
```

Algorithm 1: Generate $\boldsymbol{\lambda}$, $\boldsymbol{\rho}$ and $|E|$

3.2 Belief Propagation

The Belief Propagation (BP) algorithm is a method that allows efficient computations on factor graph probabilistic models for discrete variables. It belongs to the broad class of mean-field methods in statistical physics, which basically consist in approximating intractable probability distributions with simpler ones. In statistical physics, BP is also known under the name *cavity method* and is found to be essentially equivalent to the Bethe-Peierls approximation. In the coding community, BP is often called *sum-product algorithm*.

In statistical physics, when one is presented a model, usually in the form of an energy function, the aim is to *solve* it. This means finding a way to compute partition functions, marginal probabilities (often called simply marginals) over variables, observable averages and possibly minima/maxima of some quantity. All of the mentioned operations in principle require, in order to be computed exactly, a number of operations that scales exponentially with the system size. This is prohibitive since typically one is interested in large systems.

For certain simple instances, however, one can proceed analytically and find a closed-form solution to the model. If we take the Ising model, this is possible for trivial graph topologies and uniform fields and couplings: the one-dimensional chain, the completely disconnected model (another way to say paramagnetic) model and the fully connected ferromagnetic one.

For slightly more complicated structures, analytic calculations are not possible, but results can be obtained computationally with a cost that scales polynomially with the system size. This is the case for Ising models defined on tree graphs, i.e. factor graphs with no cycles. One can recursively find marginals starting from the tree leaves and express the full joint distribution as a product of marginals.

The basic idea is that in a tree, due to the loop-free property, every branch interacts with the rest of the graph through a single edge. Therefore, one can first consider how variables influence each other within the branch, then extract some net information about the whole branch's behavior and communicate it to the rest of the system. Technically this is done by computing some probability distribution which can be looked at as being a *message* exchanged between different areas of the graph. Proceeding recursively in a "dynamic programming" fashion, one can eventually find the BP equations. We will use the results without giving a proof other than the intuitive explanation above. However, a rigorous derivation can be found in many textbooks, like [Mezard and Montanari, 2009, Chapter 14].

3.2.1 The BP equations

Consider a probability distribution $p(x_1, x_2, ..., x_N)$ on a tree-like factor graph $\mathcal{G} = (V = (I, F), E)$, where variable nodes are indicated by indices $v = 1, 2, ..., n$ and factors by $f = 1, 2, ..., m$. Each variable x_v can take values in a discrete alphabet, say $\{-1, +1\}$. Let us call ∂f the set of variable nodes neighbors to factor f and similarly ∂v the set of factors neighbors to variable v. Let us call $\boldsymbol{x}_f$ the set of variables appearing in factor f and $\boldsymbol{x}_{f\setminus v}$ the variables corresponding to nodes in $\partial f \setminus \{v\}$.

The joint probability distribution of variables $\{x_v\}$ is then given by

$$p(x_1, x_2, ..., x_N) \propto \prod_{f \in F} \Psi_f(\boldsymbol{x}_f) \tag{3.20}$$

where Ψ_f is the Boltzmann weight of factor f. Equivalently, we can look at the energy

function

$$H(x_1, x_2, ..., x_N) = \sum_{f \in F} H_f(\boldsymbol{x}_f) \tag{3.21}$$

with $\Psi_f(\boldsymbol{x}_f) = \exp\left[-\beta H_f(\boldsymbol{x}_f)\right]$.

For each pair (v, f) of variable-factor nodes, we define a *message* $\mu_{fv}(x_v)$ from factor to variable and another $m_{vf}(x_v)$ from variable to factor. These are, after proper normalization, univariate probability distributions on spin x_v. The set of all messages satisfy the Belief Propagation equations

$$\begin{cases} \mu_{fv}(x_v) \propto \sum\limits_{\boldsymbol{x}_{f \setminus \{v\}}} \Psi_f\left(\boldsymbol{x}_{f \setminus \{v\}}, x_v\right) \prod\limits_{v' \in \partial f \setminus \{v\}} \mu_{v'f}\left(x_{v'}\right) \\ \mu_{vf}(x_v) \propto \prod\limits_{f' \in \partial v \setminus \{f\}} \mu_{vf'}(x_v) \end{cases} \tag{3.22}$$

Messages can be computed starting from the leaves, where two things can happen. The leaf might be a factor f, connected to a single variable v. This means that v is the only variable involved in f and therefore $\Psi_f \equiv \Psi_f(x_v)$, implying $\mu_{fv}(x_v) \propto \Psi_f(x_v)$. Otherwise, if the leaf is a variable, it will only be connected to some factor f. We get $\mu_{vf}(x_v) = \prod_{\emptyset} = 1$, which, once appropriately normalized, gives a uniform distribution. At this point, messages for the rest of the graph can be computed.

Once one has got the messages, marginal distributions for each variable can be computed as

$$b_v(x_v) \propto \prod_{f \in \partial v} \mu_{fv}(x_v) \tag{3.23}$$

These marginals are called *beliefs*. From messages, it is also possible to derive the full distributions and compute averages, although we will not make use of that.

It often happens that the energy looks like

$$H(x_1, x_2, ..., x_N) = \sum_{f \in F} H_f(\boldsymbol{x}_f) + \sum_{v \in I} H_v(x_v) \tag{3.24}$$

with factors linking different variables and factors that involve a single variable. It is the case for the Ising model, where there is one external field for each variable. (We are abusing notation a little, because earlier f used to indicate a generic factor while now just the ones involving more than one variable. The distinction is hopefully clear from the context).

When the Hamiltonian takes such form, the BP equations can be written as

$$\begin{cases} \mu_{fv}(x_v) \propto \sum\limits_{\boldsymbol{x}_{f \setminus \{v\}}} \Psi_f\left(\boldsymbol{x}_{f \setminus \{v\}}, x_v\right) \prod\limits_{v' \in \partial f \setminus \{v\}} \mu_{v'f}\left(x_{v'}\right) \\ \mu_{vf}(x_v) \propto \exp\left[-\beta H_v(x_v)\right] \prod\limits_{f' \in \partial v \setminus \{f\}} \mu_{vf'}(x_v) \end{cases} \tag{3.25}$$

with the local field $H_v(x_v)$ only appearing in the second equation.

The whole discussion above works only for tree graphs: beliefs b_v *are* the marginals distributions

$$b_v(x_v) \propto \sum_{x_{v'} : v' \neq v} p(x_1, x_2, ..., x_n) \tag{3.26}$$

which would in general require $\mathcal{O}(2^{n-1})$ operations to be computed.

On a graph with cycles one can nevertheless define messages and compute beliefs, although they will not be guaranteed to correspond to the true marginals anymore. In

this case, (3.22) becomes a fixed-point equation that updates messages iteratively, starting from some initial condition

$$\boldsymbol{\mu}^{(t+1)} = F\left(\boldsymbol{\mu}^{(t)}\right) \tag{3.27}$$

where $\boldsymbol{\mu} = [\{\mu_{fv}\}, \{\mu_{vf}\}]$ is a vector containing all messages. The fixed-point equation is typically repeated a number of times until a fixed point is possibly reached, meaning that for some t_{converg},

$$\boldsymbol{\mu}^{(t+1)} = \boldsymbol{\mu}^{(t)}, \quad \forall t > t_{\text{converg}} \tag{3.28}$$

Used in this way, BP becomes a heuristic, kind of a shot in the dark, since there is no assurance that the outcome is correct. It has proven to work well on graphs that "look locally like" a tree, in the sense that they contain large loops. However we still do not have a complete understanding of why and how BP works well in the loopy case. Sections 3.3 and 3.4 are meant to contribute to the research in this direction.

There are some cases when some facts about possible fixed points are guaranteed by Banach's theorem. Loosely speaking, the theorem states that, if under function F any pair of points come closer to each other, then there exists a unique fixed point that can be reached by iterative applications of F. The condition in question is easy to check albeit rarely met.

3.2.2 Max-sum: the zero-temperature version of BP

Taking the usual path in statistical physics, we can look for ground states of the energy function (3.21), which correspond to maxima of the probability (3.20), by taking a $\beta \to \infty$ limit. It is not important if the problem is formulated in a way that β has no meaning, like in data compression: it will eventually cancel out.

Let us perform a change of variables

$$\begin{cases} \nu_{vf}(x_v) & := \frac{1}{\beta}\log \mu_{vf}(x_v) \\ \nu_{fv}(x_v) & := \frac{1}{\beta}\log \mu_{fv}(x_v) \end{cases} \tag{3.29}$$

Now the first BP equation becomes

$$\nu_{fv}(x_v) = \frac{1}{\beta}\log\left\{\sum_{\boldsymbol{x}_{f\setminus\{v\}}} \exp\left[-\beta H_f\left(\boldsymbol{x}_{f\setminus\{v\}}, x_v\right) + \sum_{v'\in\partial f\setminus\{v\}} \beta\nu_{v'f}\left(x_{v'}\right)\right]\right\} + \text{const} \tag{3.30}$$

Thanks to the saddle point theorem

$$= \max_{\boldsymbol{x}_{f\setminus\{v\}}}\left\{-H_f\left(\boldsymbol{x}_{f\setminus\{v\}}, x_v\right) + \nu_{v'f}\left(x_{v'}\right)\right\} + \text{const} \tag{3.31}$$

The second equation becomes

$$\nu_{vf}\left(x_v\right) = \sum_{f'\in\partial v\setminus\{f\}} \nu_{vf'}(x_v) + \text{const} \tag{3.32}$$

The counterparts of beliefs in the new domain are

$$\nu_v\left(x_v\right) = \frac{1}{\beta}\log p(x_v) = \sum_{f\in\partial v} \nu_{fv}(x_v) + \text{const} \tag{3.33}$$

The last two equations are called the Max-Sum (MS) equations and are again exact on trees.

Notice that if the ground state is unique, than the marginal for each variable will be an indicator function on a single value. The belief in log space will look like

$$\nu_v(x_v) = \begin{cases} 0 & x_v = x_v^* \\ -\infty & \text{otherwise} \end{cases} \tag{3.34}$$

and the ground state configuration is found by simply picking x_v^* for each v. This is not true in case the ground state is degenerate. In order to force the non-degeneracy, one can add small random noise by means of single-variable factors that break the symmetry.

3.2.3 Max-sum for our compression scheme

The purpose of the whole discussion about BP and MS is to provide a method that efficiently solves the minimization problem at the core of the compression strategy, which we re-state here

$$\mathbf{c}^* = \underset{\bigoplus_{j=1}^{n} h_{fv} c_v = 0}{\arg\min} \sum_{v=1}^{n} c_v \oplus s_v \tag{3.35}$$

where h_{fv} are the elements of the parity-check matrix H and $\mathbf{s}$ is the source vector so be compressed. The notation v for variables and f for checks is reminiscent of the factor graph underlying H.

We can re-organize the constraint under arg min to get the argument in the form of an energy

$$H(\mathbf{c}) = \sum_{f=1,\ldots,m} H_f(\mathbf{c_f}) + \sum_{v \in I} H_v(c_v) \tag{3.36}$$

with

$$H_f(\mathbf{c_f}) = \begin{cases} 0 & \text{if } \bigoplus_{j=1}^{n} h_{fv} c_v = 0 \\ -\infty & \text{otherwise} \end{cases} \tag{3.37}$$

$$H_v(c_v) = c_v \oplus s_v \tag{3.38}$$

The Max-sum equations for the compression problem look like

$$\begin{cases} \nu_{fv}(c_v) = \max\limits_{\mathbf{c}_{f\setminus\{v\}}} \left\{ -H_f\left(\mathbf{c}_{f\setminus\{v\}}, x_v\right) + \nu_{v'f}\left(c_{v'}\right) \right\} \\ \nu_{vf}\left(c_v\right) = -c_v \oplus s_v + \sum\limits_{f' \in \partial v \setminus \{f\}} \nu_{f'v}(c_v) \end{cases} \tag{3.39}$$

Since configurations with non-satisfied checks will never "win" the maximum in the first equations, we might as well consider only the configurations that do satisfy all of them

$$\begin{cases} \nu_{fv}(c_v) = \max\limits_{\mathrm{Conf}_{(v,f)}(c_v)} \nu_{v'f}\left(c_{v'}\right) \\ \nu_{vf}\left(c_v\right) = -d_H(c_v, s_v) + \sum\limits_{f' \in \partial v \setminus \{f\}} \nu_{f'v}(c_v) \end{cases} \tag{3.40}$$

where $\mathrm{Conf}_{(v,f)}(c_v)$ is a short-hand notation for

$$\mathrm{Conf}_{(v,f)}(c_v) = \left\{ \mathbf{c}_{v'} : \sum_{v' \in \partial f \setminus v} h_{fv'} c_{v'} + h_{fv} a = 0 \right\} \quad (3.41)$$

and we substituted $-c_v \oplus s_v$ with the Hamming distance notation.

Beliefs are given by

$$\nu_v(c_v) = -d_H(y_v, c_v) + \sum_{f \in \partial v} \nu_{fv}(c_v) \quad (3.42)$$

Finally, our guess for the closest codeword is given by the decision variables

$$g_v = \max_{x_v} b_v(x_v) \quad \forall v \in V \quad (3.43)$$

It should be noted that one can weight $d_H(c_v, s_v)$ by a factor L in order to modulate the relative importance of constraints and external fields in the energy function. This makes sense when using BP, it is irrelevant for MS.

The graph underlying the parity-check matrix H is, unfortunately, not going to be a tree, so MS will have to be used as a heuristic. The reasons why trees do not make good codes are explained and proved, for error-correcting codes, in [Etzion et al., 1999].

3.2.4 Complexity

A closer look to (3.22) and (3.40) shows that we have been cheating a little when saying that BP runs in polynomial time. The sum (or max) in the first equation runs over $\mathbf{c}_{f \setminus \{v\}}$: all the possible configurations of $p-1$ variables, p being the number of variables appearing in check f, which can each take q values. This means that the computational cost of a single BP iteration scales as $\mathcal{O}(|E|q^{p-1})$, where $|E|$ is, as always, the number of edges. In our case, since $|E| = 2n$, we have $\mathcal{O}(nq^{p-1})$. Although p is usually small and highly-connected graphs are not suited for BP anyway, this is clearly not polynomial.

In the case of p-spin Ising models, one can proceed analytically and find a closed for expression which involves tanh and sign functions for BP and MS respectively. Here, a single factor $\to$ variable message update for an edge takes $\mathcal{O}(p-1)$.

In general one cannot expect to be always that lucky. However, when factors are enforcing "rigid" constraints, as in our case (the passage stressed in going from (3.39) to (3.40)), dynamic programming can be exploited to reduce the cost by means of convolutions. Details are given in section 3.6, after having introduced the generalization to higher order finite fields.

3.3 A proof of exactness

This section contains an original proof of the fact that, for a class of constrained optimization problems with parity-check-like constraints and objective function separable over variables, if Max-sum converges, then it has converged to the optimum. Lossy compression is a particular case.

Key hypothesis are:

- Variables live in $\mathbb{GF}(2)$

- Variable nodes in the factor graph all have degree 2 or less
- Possible degeneracy on the ground state were lifted so that the minimum is unique

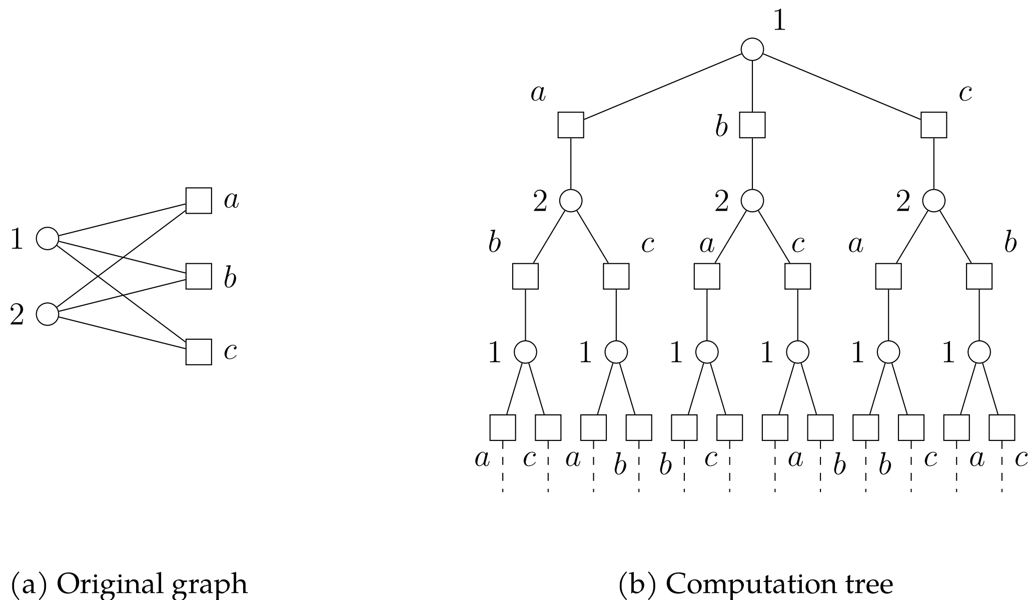
It is quite a lengthy proof, admittedly not easy to read, as it often happens for algorithms on graphs. The main tool being used is a computation tree, whose features are reviewed here, before stating the proof.

3.3.1 Computation trees

A computation tree is a loop-free graph obtained from a loopy one. Many proofs involving BP for combinatorial problems make use of this technique [Bayati et al., 2008, Weiss and Freeman, 2000, Frey and Koetter, 2000]. Here we will follow the notation in [Bayati et al., 2008]. In principle, computation trees can be built for any graph, although here we will focus on factor graphs.

Given a factor graph $\mathcal{G} = ((V, F), E)$, the idea is to pick a variable node $v_0 \in V$ to be the root and then unroll $\mathcal{G}$ around it, respecting the connectivity of each node. Starting from the root, the first level of the computation tree is formed by the root's neighbors. The second level is made of the first level nodes' neighbors, except the root. The third level is made of the second level nodes' neighbors, except the ones already connected from above. Proceeding in this way for a number k of levels produces the level-k computation tree with root v_0 $T^k_{v_0}$.

The concept is best explained through an example (fig. 3.2).



(a) Original graph

(b) Computation tree

Figure 3.2: A graph and its level-5 computation tree rooted at variable 1. Variables are indicated by numbers and factors by letters. The dashed lines at the bottom in (b) indicate that the tree can grow to arbitrary depths

Nodes in the original graphs have multiple counterparts in the computation tree. Sometimes it is useful to refer to those counterparts with the same name as the respective nodes in the original graph, although this is not rigorous notation.

3.3.2 Statement and proof

Define a factor graph $\mathcal{G} = ((V, F), E)$, where $V = \{v | v \in 1, ..., n\}$ are variable nodes, $F = \{f | f = 1, ..., m\}$ are factor nodes and edges E are defined by the non-zero element of the parity-check matrix $\mathbf{H} \in \{0, 1\}^{m,n}$. Variable nodes have degree at most 2.

Define the problem of finding the minimum of some objective function that we will call energy E using an analogy from physics, expressed as

$$\min_{\mathbf{y}:\mathbf{Hy}=\mathbf{0}} E(\mathbf{y}) = \min_{\mathbf{y}:\mathbf{Hy}=\mathbf{0}} \sum_{v=1}^{n} f_v(y_v) \tag{3.44}$$

with $f_v : \mathbb{GF}(2) \to \mathbb{R}$ for $v = 1, ..., n$.

Claim If there is a unique solution, then any run of the Max-Sum algorithm that converges to a fixed point, will have converged to such solution.

In particular, in the case of data compression we have

$$f_v(y_v) = y_v \oplus y_{0_v} \tag{3.45}$$

and the energy is the Hamming distance between $\mathbf{y}$ and $\mathbf{y_0}$.

Proof Suppose MS was run on $\mathcal{G}$, until convergence of messages to a fixed point. Call $\{m_{fv}\}_{\substack{v \in V \\ f \in F}}$, $\{b_v\}_{v \in V}$ respectively the messages and beliefs and let

$$g_v = \max_{y_v} b_v(y_v) \quad \forall v \in V \tag{3.46}$$

be the decision variables.

Pick a node v_0 and build the computation tree $T^k_{v_0}$ obtained by unrolling the original graph around v_0 until there are at least r counterparts of each vertex in V and so that all leaves are variable nodes; v_0 will be the root. It is not important what the depth k is, as much as the number of times each variable is repeated.

Now place messages $\{m_{fv}\}$ on the edges of $T^k_{v_0}$ that correspond to the original edges $\{(f, v)\}$ on $\mathcal{G}$. Attach to all nodes that are leaves in the computation tree but were not leaves in the original graph, a fictitious factor that sends a message equal to that flowing out of the leaf. By construction, decision variables on the computation tree are now equal to the ones in the original graph.

Consider now the optimization problem with the same structure of the original one but defined on the computation tree. Messages on $T^k_{v_0}$ constitute a solution for the MS equations: they are naturally fulfilled in the inner part of the tree and imposed on the leaves by the fictitious factors. Since MS is exact on trees, the assignment $\{g_v\}$ (replicated for each of the counterparts in $T^k_{v_0}$ of each variable in $\mathcal{G}$), is an exact solution for the problem defined on the computation tree.

Now call the optimum for the original problem

$$\{y^*_v\} = \arg\min_{\mathbf{y}:\mathbf{Hy}=\mathbf{0}} E(\mathbf{y}). \tag{3.47}$$

Suppose (absurd) that decision variable g_{v_0} for the root is different from its value in the optimal assignment, $y^*_{v_0}$. Namely, $g_{v_0} = \overline{y^*_{v_0}}$, where $\overline{x}$ is the complement of x under $\mathbb{GF}(2)$. We show that it is always possible to find a path $\mathcal{P}$ on $T^k_{v_0}$ such that complementing every

variable along such path results in an improvement in the objective function of the problem defined on the tree. This contradicts what was proven about $\{g_v\}$ being an optimum.

The key idea is the following: suppose two vectors y_1, y_2 are both solutions of $H\mathbf{y} = \mathbf{0}$ but differ in a component, which in this case is the root v_0. If v_0 is disconnected from the rest of the graph, y_1 and y_2 can have all the other bit equal to each other and be two solutions. If v_0 has degree 1, then the single factor attached to it must have at least another incident variable with value different in y_1 and y_2 in order for the parity check to be satisfied. If v_0 has degree 2, then the above must be true for both factor neighbors. With these observation in mind, let us move to the explicit construction of path $\mathcal{P}$.

To construct path $\mathcal{P}$, start from the root v_0, pick any of its factor neighbors, which by construction are at most two, and do the following: look at all the variable nodes incident on the factor and pick one for which the decision variable disagrees with the optimal solution. There will always be at least one in order for the parity-check to be satisfied, as explained before. Include that variable in the path and move on to its other factor if there is any. The process is halted when either a leaf is encountered and the path ends, or the root is found again.

In case the path ends on a leaf, go back to the root and repeat the process in the other direction, or end the path on the root if the root is a leaf. In case the path ends in a cycle, carry on extending the path repeating the same choice of variables to be included, until the leaves of the computation tree are reached. Let us stress that the resulting path $\mathcal{P}$ only touches variables which have different values in the solutions on $\mathcal{G}$ and $T^k_{v_0}$.

At this point, the path stemming on both sides from the root can either end on a "true" leaf of $T^k_{v_0}$ (one that corresponds to a leaf also on $\mathcal{G}$) or continue until the bottom of the tree, where the fictitious factors are. We prove our claim in the worst case, where both branches of the path go all the way down, the others follow easily.

Call $\mathcal{P}'$ the projection of $\mathcal{P}$ on $\mathcal{G}$: it will not be in general a path, but might contain cycles. Again thanks to the fact that no parity check can be left unsatisfied, if $\mathcal{P}'$ is a path, then its endpoints must be leaves of $\mathcal{G}$. Call E_0 the energy of the optimal configuration $\{y^*_v\}$ and $E_0 + \epsilon$ the energy of the first non-optimal one. Further, call $y_{\mathcal{P}}$ and $y_{\mathcal{P}'}$ the indicator functions of paths $\mathcal{P}$ on $T^k_{v_0}$ and $\mathcal{P}'$ on $\mathcal{G}$ respectively.

For sure, since $\mathbf{y}^*$ gives a minimum, a transformation $\mathbf{y}^* \oplus y_{\mathcal{P}'}$ that complements the variables touched by $\mathcal{P}'$ gives a positive shift in energy

$$E\left(\mathbf{y}^* \oplus y_{\mathcal{P}'}\right) \geq E_0 + \epsilon \tag{3.48}$$

Because the energy function (3.44) is a sum over functions of single nodes, the shift in energy is only due to the flip of variables in $\mathcal{P}'$. After the flip, all the touched variables assumed the value they have on $T^k_{v_0}$. But this means that complementing them on $T^k_{v_0}$ would reverse the shift, thus lowering the energy of the problem defined there by at least ϵ for each repetition of $\mathcal{P}'$, at least in the bulk.

If $\mathcal{P}'$ is a path, then the same negative shift in energy happens along $\mathcal{P}$ on $T^k_{v_0}$, finding a better optimum than $\{g_v\}$ and thus contradicting the starting point. If instead, $\mathcal{P}$ goes down all the way to the fictitious factors, the improvement gets multiplied times the number of repetitions of $\mathcal{P}'$, although it might in principle by outbalanced by the change in energy due to the interaction of the one or two leaves $\mathcal{P}$ ends on with their fictitious factors. An upper bound for this changes is given by the maximum absolute difference in messages on the tree

$$m_{\max} = \underset{(v,f)\in\mathcal{P}'}{\arg\max} \, |m_{fv}(0) - m_{fv}(1)|. \tag{3.49}$$

Since there is no limit to the tree's depth, it suffices to repeat the path enough times to be sure that energy will decrease. This amounts to choosing k so that

$$k\epsilon > m_{max} \tag{3.50}$$

Namely,

$$k > \frac{m_{\max}}{\epsilon} \tag{3.51}$$

The same argument can be repeated for all v_0's for which the solution on the computation tree differs from the presumed optimal one.

This completes the proof.

One never needs to actually build the computation tree, it is enough to ensure that it exists. In addition to this, unfortunately the proof assumes convergence has happened, which is not trivial at all. However, (3.51) states something important: the depth of the computation tree is bounded by a quantity that increases as the energy shift between ground state and first excited state becomes smaller. This fact suggests that MS might have an especially hard time finding the true minimum when this is not well isolated from the other sub-optimal solutions.

Perhaps more importantly, in case the path that was build does not go down to the fictitious factors, the proof works without having to unroll the computation tree at all! This situation corresponds to $\mathcal{P}'$ being a simple path connecting two leaves. Although this is heuristic reasoning, the strong hint is towards the fact that having a large number of leaves available in $\mathcal{G}$ might improve the results.

3.4 Reduced factor graphs and energy landscape

Interestingly, the proof just presented gives a hint for the explainability of BP as a heuristic technique. What seems to be suggested is that a graph with many leaves can bring advantages in term of convergence probability and perhaps also distortion.

Discarding for a moment the minimization and looking only at the problem of satisfying the parity checks, leaves represent "free" variables, in the sense that they can be adjusted to accommodate whatever values of the neighboring bulk variables. It would make sense if the message-passing iterations worked somehow like the path-building process described in the previous section: one can imagine waves of spin (bit) flips that expand throughout the graph, at each factor tweaking a new variable so that parity stays fulfilled (of course this works well provided that variables all have degree ≤ 2). In a graph with no leaves, such process is bound to end up trapped in loops, thus reducing the chances of finding a solution. If there are leaves available, instead, they act as an outlet for the unresolved conflicts, adapting smoothly to being either a 0 or a 1.

Exposing some leaves can be done by either creating the graph with $\lambda_1 > 0$ in the first place or, after having created it with all variables of degree 2, removing a number b of factors picked at random, together with the connections they had. The latter strategy, the one we used, is equivalent to erasing a row from the parity-check matrix H. With each removed factor one gets rid of a constraint and the number of codewords doubles, causing the rate to increase by $1/n$, which is negligible for $n \to \infty$. The corresponding improvement in distortion (see details in section 4.2) is remarkable.

b-reduction perturbs the space of codewords, which most likely are not as evenly spread anymore. Therefore we expect distortion to decrease the more checks are removed

but only up to a certain point, where the advantages in convergence are overcome by the fact that the graph is not a *good* one anymore.

Besides affecting the graph topology, reduction also modifies the energy landscape of the problem, meaning the shape of the n-variable function (2.19) (or, stated alternatively, (3.24)) that we are trying to minimize. We can picture the landscape as an n-dimensional grid of infinitely high bars alternating with a minority of low ones corresponding to codewords, which all get a null energy contribution from the fact that they satisfy all parity checks. Each low bar is as low as the corresponding codeword is near the source vector and the lowest one corresponds to the ground state, i.e. the closest codeword.

1-reduction amounts to doubling the number of low bars, thus statistically increasing the availability of other minima in the proximity of any codeword. This might be advantageous for an algorithm like BP which, starting from a configuration, tries some local changes hoping to find a better solution.

3.5 Moving to $\mathbb{GF}(q)$

The non-optimality of LDPC matrices, as we saw, is necessary in order to allow the problem of finding the closest codeword to be tractable, both in a channel coding and source coding context. A clever strategy to overcome such limitation has been proposed by Davey and McKay [Davey and MacKay, 1998]: work with a generalization of bits. As it was mentioned in chapter 1, binary numbers are a particular case of a finite (or Galois) field $\mathbb{GF}(q)$. The term *finite* expresses the fact that such fields contain a finite amount of numbers, unlike, for example, the integers $\mathbb{Z}$. In particular, generalization from $\mathbb{GF}(2)$ can be done with little effort to fields with $q = 2^k, k > 1$.

The set of elements in $\mathbb{GF}(2^k)$ is

$$\{0, 1, 2, ..., 2^k - 1\}$$

for which $q = 2$ ($k = 1$) is a valid sub-case. A notorious example are hexadecimal numbers, which correspond to $\mathbb{GF}(2^4)$.

The algebra works as follows: take $x_1, x_2 \in \mathbb{GF}(2^k)$ and "unwrap" them as binary digits. Example with $\mathbb{GF}(8)$:

$$\begin{aligned} x_1 = 6 \quad &\longleftrightarrow \quad 100 \\ x_2 = 3 \quad &\longleftrightarrow \quad 011 \end{aligned}$$

The sum of x_1 and x_2 is given by the bitwise XOR

$$x_1 + x_2 = 7 \quad \longleftrightarrow \quad 100 \oplus 011 = 111$$

Again, we have that that each number is the summation inverse of itself.

Multiplication is more complicated because it involves polynomial representations. Regardless of how it is performed, one can consult a table and store all the possible results in a matrix once and for all. For $\mathbb{GF}(8)$ the table is reported in tab. 3.1.

It can be useful to also store multiplication inverses, meaning the values whose product gives 1. For $\mathbb{GF}(8)$, inverses are shown in tab. 3.2. As is the case with real numbers, 0 has no well-defined inverse.

The workflow for data compression goes as follows: given a bit string to compress, transform it into a $\mathbb{GF}(q)$ vector, perform the compression using a parity-check matrix H also taking values in $\mathbb{GF}(q)$, transform back to $\mathbb{GF}(2)$ and compute the distortion.

×	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	3	1	7	5
3	0	3	6	5	7	4	1	2
4	0	4	3	7	6	2	5	1
5	0	5	1	4	2	7	3	6
6	0	6	7	1	5	3	2	4
7	0	7	5	2	1	6	4	3

Table 3.1: Multiplication table for $\mathbb{GF}(8)$

1	2	3	4	5	6	7
1	5	6	7	2	3	4

Table 3.2: Multiplication inverses for $\mathbb{GF}(8)$

To go from a string of bits to a string of $\mathbb{GF}(2^k)$ values, one simply groups together every k bits and interprets them as binary numbers. For example, with $k = 3$

$$[011101000010] \rightarrow [\underbrace{011}_{3}\underbrace{101}_{5}\underbrace{000}_{0}\underbrace{010}_{2}] \rightarrow [3, 5, 0, 2]$$

If the bit vector has length n, then the resulting one has length n/k. Of course vector sizes must be adjusted as to produce integer values. Matrix H is constructed by means of the usual factor graph, this time choosing a coefficient h_{fv} sampled uniformly from $\{1, ..., q-1\}$ for each edge and coefficient zero for non-existing edges [Braunstein et al., 2011]. Once MS has finished running, the resulting vector is re-cast into bits simply by representing values in binary and concatenating. It can then be compared with the source vector to compute the distortion.

The reasons why moving to $\mathbb{GF}(q)$ gives an improvement in performance are not quite clear, although well supported by empirical evidence for channel coding [Davey and MacKay, 1998, Declercq and Fossorier, 2005] and source coding [Braunstein et al., 2011].

3.6 Efficient implementation of BP

The implementation of factor→variable messages is in principle done by summing over all possible configurations of the factor's neighbors. When the number of those configurations is limited by some constraint that is simple enough to treat, messages can be computed efficiently by means of a convolution operation. In our case, the constraints are the parity checks.

Here we quickly review what a convolution is and then describe the application to BP on $\mathbb{GF}(2^k)$.

3.6.1 Convolutions

This section is heavily inspired by Alfredo Braunstein's course notes from 2019.

Convolutions, perhaps best known as a signal processing tool, play an important role in efficient solutions to problems in combinatorics. More precisely, they are often used in the context of Dynamic Programming.

```
function conv(f1,f2)
    q1 = lenght(f1)
    q2 = length(f2)
    f3 = zeros(q1+q2)
    for x1 in 1:q1
        for x2 in 1:q2
            f3[x1+x2] += f1[x1] + f2[x2]
        end
    end
    return f3
end
```

Algorithm 2: Convolution of two functions. Domains are shifted with respect to the definition that was just given because array indices in Julia start at 1

Here we will treat the case of convolutions of functions with discrete, finite domains. Let $f_1 : \{0,1,...,q_1-1\} \to \mathbb{R}$ and $f_2 : \{0,1,...,q_2-1\} \to \mathbb{R}$ be a pair of discrete functions. The convolution of f_1 and f_1 is a third function f_3

$$f_3(k) = (f_1 \circledast f_2)(k) = \sum_{x_1+x_2=k} f_1(x_1) f_2(x_2) \tag{3.52}$$

where $x_1+x_2 = k$ implicitly means $\{x_1, x_2 \mid x_1 \in \{0,1,...,q_1-1\}, x_2 \in \{0,1,...,q_2-1\}, x_1+x_2 = k\}$

The domain of f_3 is $\{0,1,...,q_1+q_2-2\}$ and the cost to compute it is $\mathcal{O}(q_1 q_2)$, as is clear from the double for loop in algorithm 2. In analogy with what is done in signal processing, one can perform a Fast Fourier Transform to reduce the cost for a single convolution to $\mathcal{O}(q \log q)$, where we supposed for simplicity that $q_1 = q_2 = q$.

Convolution turns out to be associative, therefore one can generalize it from two to an arbitrary number of functions

$$\left(\circledast_{i=1}^{n} f_i\right)(k) = \sum_{\sum_i x_i = k} \prod_{i=1}^{n} f_i(x_i) \tag{3.53}$$

The order in which pairs of functions are convoluted (which does not matter, thanks to associativity) can be picked in a smart "divide and conquer" fashion, in order to optimize the overall number of operations.

Constraint $\sum_i x_i = k$ in (3.53), which is a simple sum, can be generalized to a weighted linear combination $\sum_i w_i x_i = k$, $w_i \in \mathbb{Z}^+$. Moreover, the sum operation itself can be generalized to, say, a sum over $\mathbb{GF}(q)$. The next section contains an example of how to include both these two steps.

In full generality, a weighted convolution looks like

$$\left(\circledast_{i=1}^{n} f_i\right)(k) = \sum_{\sum_i w_i x_i = k} \prod_{i=1}^{n} f_i(x_i) \tag{3.54}$$

One can define a "zero-temperature" limit for convolutions. Say $f_1(x_1) = e^{\beta h_1(x_1)}$, $f_2(x_2) = e^{\beta h_2(x_2)}$, then define

$$e^{\beta h_3(x_3)} := (f_1 \circledast f_2)(k) = \sum_{x_1+x_2=k} e^{\beta[h_1(x_1) h_2(x_2)]} \tag{3.55}$$

When $\beta \to \infty$, the sum concentrates on the summand that gives the largest argument at exponent, namely

$$e^{\beta h_3(x_3)} = \max_{x_1+x_2=k} e^{\beta[h_1(x_1)h_2(x_2)]} \tag{3.56}$$

and finally

$$h_3(x_3) = \max_{x_1+x_2=k} \{h_1(x_1)h_2(x_2)\} \tag{3.57}$$

The $\beta \to \infty$ limit is an interesting point of view, although not an essential step. One can directly define the $(\max, +)$-convolution as

$$f_3(k) = (f_1 \mathbin{\bigcirc\!\!\!\!\vee} f_2)(k) = \max_{x_1+x_2=k} f_1(x_1) + f_2(x_2) \tag{3.58}$$

An implementation is shown in algorithm 3.

```
1  function msc(f1,f2)
2     q1 = lenght(f1)
3     q2 = length(f2)
4     f3 = fill(-Inf, q1+q2)
5     for x1 in 1:q1
6        for x2 in 1:q2
7           f3[x1+x2] = max(f3[x1+x2], f1[x1] + f2[x2])
8        end
9     end
10    return f3
11 end
```

Algorithm 3: $(\max, +)$-convolution of two functions. Domains are shifted with respect to the definition that was just given because array indices in Julia start at 1

Just like before, one can generalize to weighted sums and more than two functions

$$\left(\bigvee_{i=1}^{n} f_i \right)(k) = \max_{\sum_i w_i x_i = k} \sum_{i=1}^{n} f_i(x_i) \tag{3.59}$$

which has the same form as the MS equations for data compression (3.40).

3.6.2 BP and MS with convolutions

We use the notation from section 3.2.3 where variables are indicated with a letter c for codeword and $\mu_v^1(c_v) := e^{-\beta d_H(c_v, s_v)}$ is the external fields that favors codewords close to the source vector **s**. We have that, at fixed f, v, c_v, the first BP equation is

$$\mu_{fv}(c_v) \propto \sum_{\text{Conf}_{(v,f)}(c_v)} \prod_{v' \in \partial f \setminus \{v\}} \mu_{v'f}(c_{v'}) \tag{3.60}$$

The set $\text{Conf}_{(v,f)}(c_v)$, is given by

$$\left\{ \boldsymbol{c}_{v'} : \sum_{v' \in M(f) \setminus v} h_{fv'} c_{v'} + h_{fv} c_v = 0 \right\} = \left\{ \boldsymbol{c}_{v'} : \sum_{v' \in M(f) \setminus v} h_{fv'} c_{v'} = -h_{fv} c_v \right\} \tag{3.61}$$

$$= \left\{ \boldsymbol{c}_{v'} : \sum_{v' \in M(f) \setminus v} h_{fv'} c_{v'} = h_{fv} c_v \right\} \tag{3.62}$$

$$= \left\{ \boldsymbol{c}_{v'} : \sum_{v' \in M(f) \setminus v} (h_{fv})^{-1} h_{fv'} c_{v'} = c_v \right\} \tag{3.63}$$

where all summations, multiplications and inverses involving variables c_v are intended on $\mathbb{GF}(q)$.

In order to recover a convolution in a more comfortable form, define new variables

$$\tilde{c}_{v'} := (h_{fv})^{-1} h_{fv'} c_{v'} \tag{3.64}$$

and functions $\tilde{\mu}_{v'f}(\tilde{c}_{v'})$ so that $\tilde{\mu}_{v'f}(\tilde{c}_{v'}) = \mu_{v'f}(c_{v'})$. The way to accomplish this is to impose

$$\tilde{\mu}_{v'f}(\tilde{c}_{v'}) := \mu_{v'f}\Big((h_{fv'})^{-1} h_{fv} \tilde{c}_{v'} \Big) \tag{3.65}$$

The equation now is a convolution in a familiar form

$$\mu_{fv}(c_v) \propto \sum_{\left\{ \tilde{\boldsymbol{c}}_{v'} : \sum_{v' \in M(f) \setminus v} \tilde{c}_{v'} = c_v \right\}} \prod_{v' \in \partial f \setminus \{v\}} \tilde{\mu}_{v'f}(\tilde{c}_{v'}) \tag{3.66}$$

Explicitly,

$$\mu_{fv}(c_v) \propto \sum_{\left\{ \tilde{\boldsymbol{c}}_{v'} : \sum_{v' \in M(f) \setminus v} \tilde{c}_{v'} = c_v \right\}} \prod_{v' \in \partial f \setminus \{v\}} \mu_{v'f}\Big((h_{fv'})^{-1} h_{fv} c_{v'} \Big) \tag{3.67}$$

Messages from variable to factor are simpler and do not involve convolutions.

We can repeat an analogous process for the Max-sum equations. Define

$$\nu_{vf}(c_v) := \frac{1}{\beta} \log \mu_{vf}(c_v) \tag{3.68}$$

Now (3.60) becomes

$$\mu_{fv}(c_v) \propto \sum_{\text{Conf}_{(v,f)}(c_v)} \prod_{v' \in \partial f \setminus \{v\}} e^{\beta \nu_{v'f}(c_{v'})} \tag{3.69}$$

$$= \sum_{\text{Conf}_{(v,f)}(c_v)} \exp\left(\beta \sum_{v' \in \partial f \setminus \{v\}} \nu_{v'f}(c_{v'}) \right) \tag{3.70}$$

$$= \max_{\text{Conf}_{(v,f)}(c_v)} \exp\left(\beta \sum_{v' \in \partial f \setminus \{v\}} \nu_{v'f}(c_{v'}) \right) \tag{3.71}$$

$$= \exp\left(\beta \max_{\text{Conf}_{(v,f)}(c_v)} \sum_{v' \in \partial f \setminus \{v\}} \nu_{v'f}(c_{v'}) \right) \tag{3.72}$$

Substitute back (3.68)

$$\nu_{fv}(c_v) = \frac{1}{\beta} \log \exp \left(\beta \max_{\mathrm{Conf}_{(v,f)}(c_v)} \sum_{v' \in \partial f \setminus \{v\}} \nu_{v'f}(c_{v'}) \right) + \mathrm{const} \tag{3.73}$$

$$= \max_{\mathrm{Conf}_{(v,f)}(c_v)} \sum_{v' \in \partial f \setminus \{v\}} \nu_{v'f}(c_{v'}) + \mathrm{const} \tag{3.74}$$

which is a weighted $(\max, +)$-convolution. It can be reduced to an unweighted one following the same steps as above.

Chapter 4

Experimental results and conclusions

4.1 Implementation details and notation

BP, being a heuristic with (in general) no assurance of convergence, has got its tricks, empiric rules of thumb and parameters which must be tuned by trial-and-error in the absence of more precise indications from the theory. Here we present the ones that were employed in our work, together with the notation that is used later to describe the outcome of experiments.

4.1.1 Reinforced Belief Propagation

The most prominent problem that is the scarce rate of convergence. BP not converging can be brought back to the presence of multiple conflicting constraints, none of them sufficiently more "convincing" than the others. As a result, the algorithm keeps bouncing, changing its mind often, never settling on one solution. Usually, or at least for the problem of data compression, it would be preferable to have a solution which is not exactly optimal, rather than having no solution at all. Therefore, one can push the algorithm into making a decision by *reinforcing* beliefs obtained from previous iterations, therefore adding some memory to the procedure.

The resulting modified algorithm, called *Reinforced* Belief Propagation, adds the following step to the computation of beliefs (3.23)

$$b_v^{(t)}(x_v) \propto \left[b_v^{(t-1)}(x_v)\right]^{\gamma t} \prod_{f \in \partial v} \mu_{fv}^{(t)}(x_v) \tag{4.1}$$

where t is the iteration index and γ is the reinforcement parameter for which a good value should be found. Analogously in log-domain, for MS

$$\nu_v^{(t)}(x_v) = \left[\nu_v^{(t-1)}(x_v)\right] \gamma t + \sum_{f \in \partial v} \nu_{fv}^{t}(x_v) + \text{const} \tag{4.2}$$

As a rule of thumb, BP should converge in a number of iterations of the order of $1/\gamma$.

Logically, reinforcement will worsen the performance in distortion for converged instances, because it means caring about reaching a solution with non-infinite energy, with less regard about the solution's quality. However, whenever the algorithm does not converge, it is fair to consider a distortion of 0.5: since there is no hint on a codeword that might be good, we can imagine to be sending the zero vector or any other randomly picked codeword. Both choices give an expected error of 0.5.

In light of this consideration, it can definitely be a good idea to use reinforcement and increase the chances of convergence, even if it means obtaining a slightly larger distortion for the converged instances.

Unfortunately, the extension of BP to the reinforced version does not meet the hypothesis of the theorem in section 3.3 and therefore no claims can be made on the solution's optimality.

4.1.2 Symmetry-breaking fields

As it was mentioned earlier, the estimates for marginals at ground state provided by MS are useful as long as such minimum energy state is unique. In the case of data compression, such condition fails to be met in case there are two or more codewords equally distant from the source vector. From a practical perspective, we do not care which one is chosen, since they all give the same energy, hence the same distortion.

A simple way to avoid possible degeneracies is to add a small random noise to the external fields

$$h_v\left(c_v\right) = d_H\left(c_v, s_v\right) + \eta, \quad \eta \sim \mathcal{N}(0, \sigma^2) \tag{4.3}$$

where $\mathbf{s}$ is the source vector and σ controls the noise level. In this way the minimization problem (2.19) becomes

$$\mathbf{c}^* = \underset{\mathbf{c}:H\mathbf{c}=0}{\arg\min} \sum_{v=1}^{n} h_v(c_v) \tag{4.4}$$

One thing to be careful about is the magnitude of σ, which should be on a different scale than the Hamming distance, so the two do not interfere: external field should only make equally-distant codewords distinguishable among themselves. In the experiments, we set $\sigma = 10^{-4}$.

4.1.3 Message updates

From the way the BP equations are stated as a fixed-point system, message updates should be done in parallel, with each message at time $t+1$ being a function of messages at time t. With this approach the order in which messages are picked to be updated is irrelevant for the final result.

It might be convenient, instead, to proceed in a "serial" fashion, with messages updated in random order at each iteration. This is common practice for numerical fixed-point equation solvers. Say for example that

$$\mu_{fv} = F_{fv}\left(\mu_{v_1 f}, \mu_{v_2 f}, \mu_{v_3 f}\right),$$

v_1, v_2, v_3 being the neighbors of f. If any of the messages at RHS came before μ_{fv} in the ordering, say $\mu_{v_2 f}$ and $\mu_{v_3 f}$, their updated version will be used:

$$\mu_{fv}^{(t+1)} = F_{fv}\left(\mu_{v_1 f}^{t}, \mu_{v_2 f}^{(t+1)}, \mu_{v_3 f}^{(t+1)}\right)$$

Mixing up old and new messages in this way is empirically found to help convergence and clearly both serial and parallel updates have the same fixed points.

4.1.4 Multiple trials

Before the first BP iteration, messages are randomly set to initial values and the same goes for the symmetry-breaking component added to the external fields. It might be the case that such starting point influence the algorithm's trajectory in phase space, ultimately determining if and where convergence happens.

Following the idea in [Braunstein et al., 2011], here BP was set to re-start with a new random initialization in case convergence was not reached after some $maxiter$ iterations. This is repeated for a maximum of T_{max} times. In our experiments, we usually set T_{max} to around $3-6$.

4.1.5 Damping

Damping is a technique often used for iterative solving of fixed-point equations. While normally the update step at iteration t is

$$\boldsymbol{x}^{(t+1)} \leftarrow F\left(\boldsymbol{x}^{(t)}\right) \tag{4.5}$$

with damping this becomes

$$\boldsymbol{x}^{(t+1)} \leftarrow \alpha\boldsymbol{x}^{(t)} + (1-\alpha)\, F\left(\boldsymbol{x}^{(t)}\right) \tag{4.6}$$

with $\alpha \in [0,1]$ the *damping factor*. The "new" vector $F\left(\boldsymbol{x}^{(t)}\right)$ goes through a weighted average with the "old" one, with α ruling the mixture proportions. This strategy is known to help convergence in cases where iterations hop back and forth around the fixed point.

How and why the process works is evident for simple F's while ours is quite a complex one. Thus, we proceeded tentatively to see whether damping could help improve the ratio of converged instances or the number of iterations needed to reach a fixed point. Unfortunately, none of the two facts were proven to be true by the simulations and we kept on working without damping.

4.1.6 Convergence criteria

The BP equations, being a fixed-point system, are considered to have reached convergence once messages do not vary anymore from one iteration to the next. Numerically, this condition is tested by checking that the maximum absolute variation is lower than some tolerance

$$\max_{(f,v)\in E} \left\| \mu_{fv}^{(t)} - \mu_{fv}^{(t-1)} \right\| < tol. \tag{4.7}$$

We call this convergence criterion *convergence of messages*. Incidentally, this is the criterion that must be used to perform empirical observation regarding the statement proved in section 3.3. The heuristic expedients described previously, however, claim the need for new ways to assert that BP has converged.

When reinforcement is used, messages will eventually diverge instead of converging. In this case a sensible thing to do is establish that the algorithm has converged if the decision variables (3.43) do not change for at least $nmin$ consecutive iterations

$$g_v^{(t)} = g_v^{(t-1)} = \dots = g_v^{(t-nmin)} \tag{4.8}$$

We call this criterion *convergence of decision variables*.

Lastly, it was found during the experiments that MS spends the initial iterations on a non-codeword, then as soon as it lands on one, it converges (in messages) immediately. Therefore, a third criterion can be introduced which states that convergence has been reached whenever

$$H\mathbf{g} = 0 \tag{4.9}$$

meaning the decision variables form a codeword. In the following, this will be referred to as *parity convergence*. This criterion has the advantage of not relying on any arbitrary threshold, unlike the ones before.

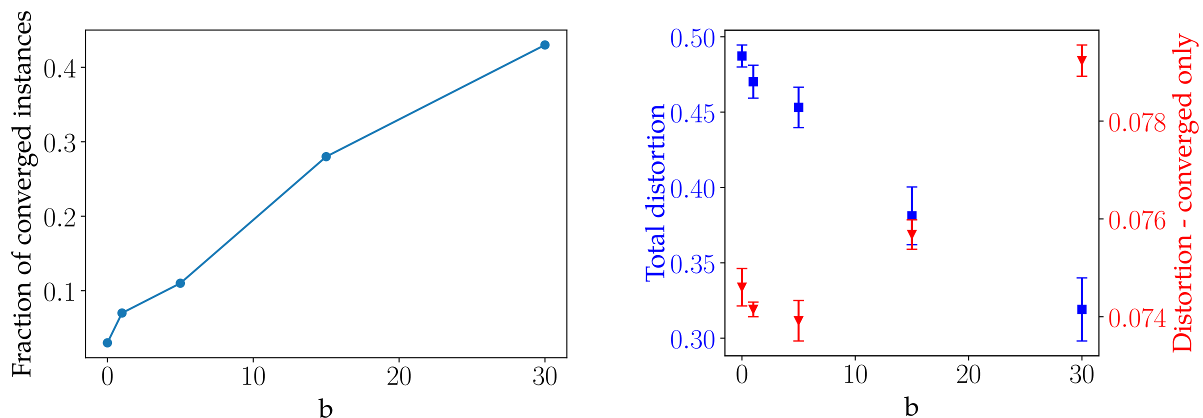
4.1.7 Performance assessment

The goodness of a certain setup is assessed by comparison the empirical distortions with the rate-distortion and naïve compression curves. Points representing experiments are plotted on a rate-distortion plane like the one in fig. 2.1. Whenever MS did not converge to a solution satisfying all parity checks, we counted a distortion of 0.5, since in this case one cannot do any better than choosing a random codeword.

The code block length is given as $n = ...$, which corresponds to the number of bits. For $q = 2$ this value coincides with the number of graph variable nodes $|V|$, while in general $|V| = \frac{n}{\log_2(q)}$. When comparing different field orders, we do so keeping n fixed.

4.2 Results with graph reduction

Without further hesitation, let us start laying out the empirical results, starting from the effects of b-reduction. With $q = 2, n = 840, R = 0.7, T_{max} = 3$, we can let the number b of removed factors vary. This is done by creating factors in excess and then removing them such that the rate stays fixed. Results for the average distortion are showed in figure 4.1.



(a) Fraction of converged instances vs. number of removed factors for $b = [1, 2, 5, 15, 30]$.

(b) Average distortion vs. number of removed factors. Left axis (blue squares): total distortion, 0.5 for instances that did not converge. Right axis (red triangles): distortion for converged instances only

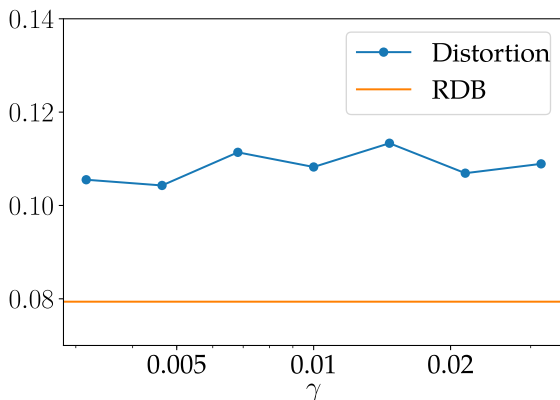
Figure 4.1: Results for $q = 2, n = 840, R = 0.7, T_{max} = 3, \gamma = 0$, average over 100 random {graph, source vector} instances, convergence criterion: parity.

It is immediately clear from fig. 4.1a that, at equal rate, the number of instances that converge increases steadily with b. From fig. 4.1b, left axis, we see that, as an expected consequence, the total distortion decreases, due to the fact that there are less unconverged instances to contribute with a 0.5.

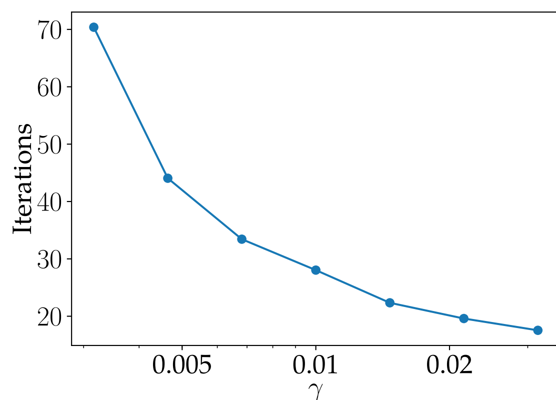
A fair question would be whether reducing the graph can make the distortion decrease *per se* and the answer, as shown in fig. 4.1b, right axis, is yes. At least for small values of b, also the distortion for only the instances that satisfy all parity checks, decreases. As b increases further, the properties of the graph get too perturbed from the original ones and the distortion goes up again.

4.3 Results with reinforcement

Reinforcement allows a remarkable speed-up in convergence. Figure 4.2 shows the effects of varying the reinforcement parameter γ.



(a) Average distortion vs. γ, with rate-distortion bound as a reference



(b) Average number of iterations needed for convergence

Figure 4.2: Results for $q = 64, n = 420, R = 0.6, T_{max} = 5, b = |V|/30$, average over 200 random {graph, source vector} instances, convergence criterion: parity.

The number of iterations needed to reach convergence is considerably reduced, while the average distortion does not seem to be affected too much, perhaps a slight upward trend. As was the case with b-reduction, we can take a look at the average distortion given by converged instances only (fig. 4.3). Time for convergence is reduced, at the cost of worsening the "pure" distortion, i.e. the one computed without the 0.5 contribution from unconverged instances.

An optimal value for γ should be chosen depending on the rate and the code block length. Something around $\gamma = 10^{-2}$ seems to work well as a first guess.

4.4 Comparison of field orders

Finally, let us show the improvements brought by increasing the field order to $q > 2$ (fig. 4.4). As q increases, data points get closer and closer to the rate-distortion bound.

One could wonder whether the reasons for the remarkable improvement brought by moving to higher-order fields rely in the algebra of $\mathbb{GF}(q)$, in the way numbers are com-

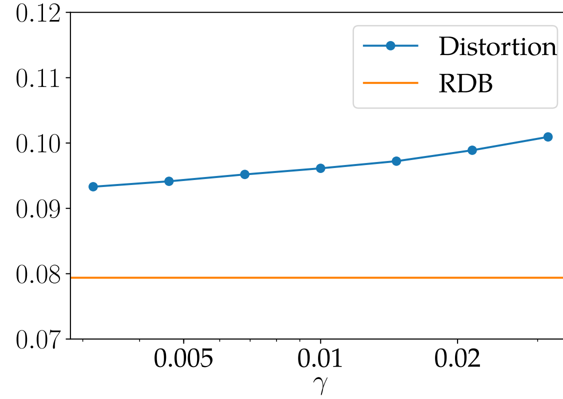


Figure 4.3: Average distortion for instances that converged. $q = 64, n = 420, R = 0.6, T_{max} = 5, b = |V|/30$, average over 200 random {graph, source vector} instances, convergence criterion: parity.

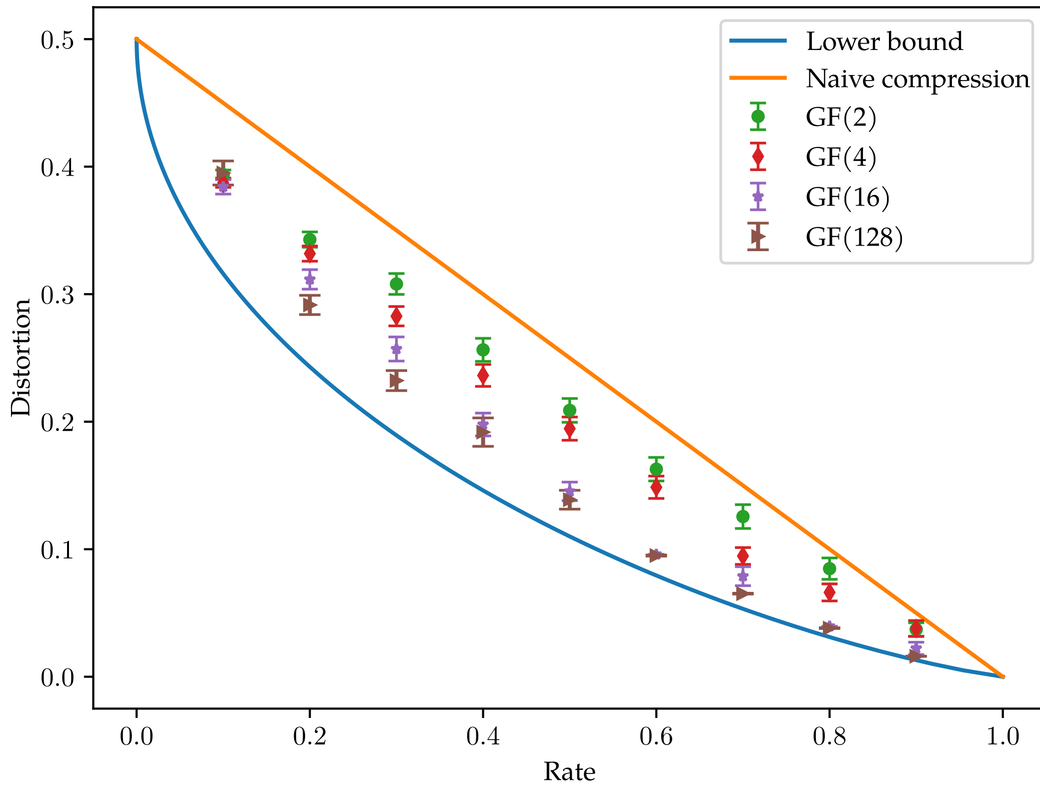


Figure 4.4: Average distortion for increasing values of q. $n = 4200, T_{max} = 5, b = |V|/30, \gamma = 5 \cdot 10^{-3}$, average over 100 random {graph, source vector} instances, convergence criterion: parity.

bined through multiplications and additions. Interestingly enough, the answer seems to be... not necessarily.

A field is a set of elements with a series of nice properties which include associativity and commutativity of multiplication. However, nothing prevents us from inventing a new multiplication operation different from the one defining $\mathbb{GF}(q)$ and using it in the algorithm. Taking $q = 8$ as an example, we can randomly permute the rows of the original

multiplication table 3.1 (excluding the entries that impose $0x = 0 \, \forall x$) and get table 4.1

$\times$	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	7	6	1	4	5	2	3
2	0	4	1	5	7	6	3	2
3	0	7	6	4	5	1	3	2
4	0	1	5	2	7	4	6	3
5	0	2	6	7	4	1	3	5
6	0	6	4	1	7	3	2	5
7	0	1	7	5	2	3	6	4

Table 4.1: Table for an arbitrary multiplication operation on a set with $q = 8$ elements

Imposing that multiplication is performed in the order *row* $\cdot$ *column*, we can immediately see that the new product is non-commutative (the matrix is not symmetric) and also non-associative:

$$3 \cdot (4 \cdot 5) = 5$$
$$(3 \cdot 4) \cdot 5 = 1$$

By carefully re-writing the calculations in section 3.6 without ever using associativity, one can run the compression algorithm with a new arbitrary product extracted randomly each time a new graph instance is built. The result, shown in figure 4.5 is, quite surprisingly, if not equivalent at least comparable to what we would get from the standard $\mathbb{GF}(q)$ product with the same random sequence of graphs and input vectors.

For rates below 0.6 arbitrary multiplication performs slightly worse than $\mathbb{GF}(q)$, for $R = 0.7$ it works even better, for $R = 0.8, 0.9$ the results are equivalent.

This is not much of a constructive conclusion, since we still do not know why higher order "fields" work better than binary variables, however it seems sensible to guess that the algebra of $\mathbb{GF}(q)$ has little to do with it. A possible explanation is that, while with $q = 2$ multiplication was trivial, this is not true anymore for $q > 2$. Elements h_{fv} of the parity check matrix now play a role in the BP equations that is more than selecting or not a certain variable, as was the case for bits. If we look at q-ary numbers in their binary representation, ultimately, the effect of multiplication, the $\mathbb{GF}(q)$ one or any other, is that those bits somehow get mixed. It may be that it is not important how that happens, whether it is following rigorous algebraic rules or a random table, but only the fact that those bits actually get mixed up.

It would be interesting to check whether analogous results can be found for LDPC codes.

4.5 Conclusions

In this work we expanded the results in [Braunstein et al., 2011] by using the Max-Sum algorithm instead of plain Belief Propagation and provided a mathematical proof that, when using binary variables, in case of convergence the result is the theoretical optimum. The effort was made keeping in mind the bigger goal of explainability of heuristic techniques in the context of constrained optimization, with a focus on the variable space and energy landscape conformations.

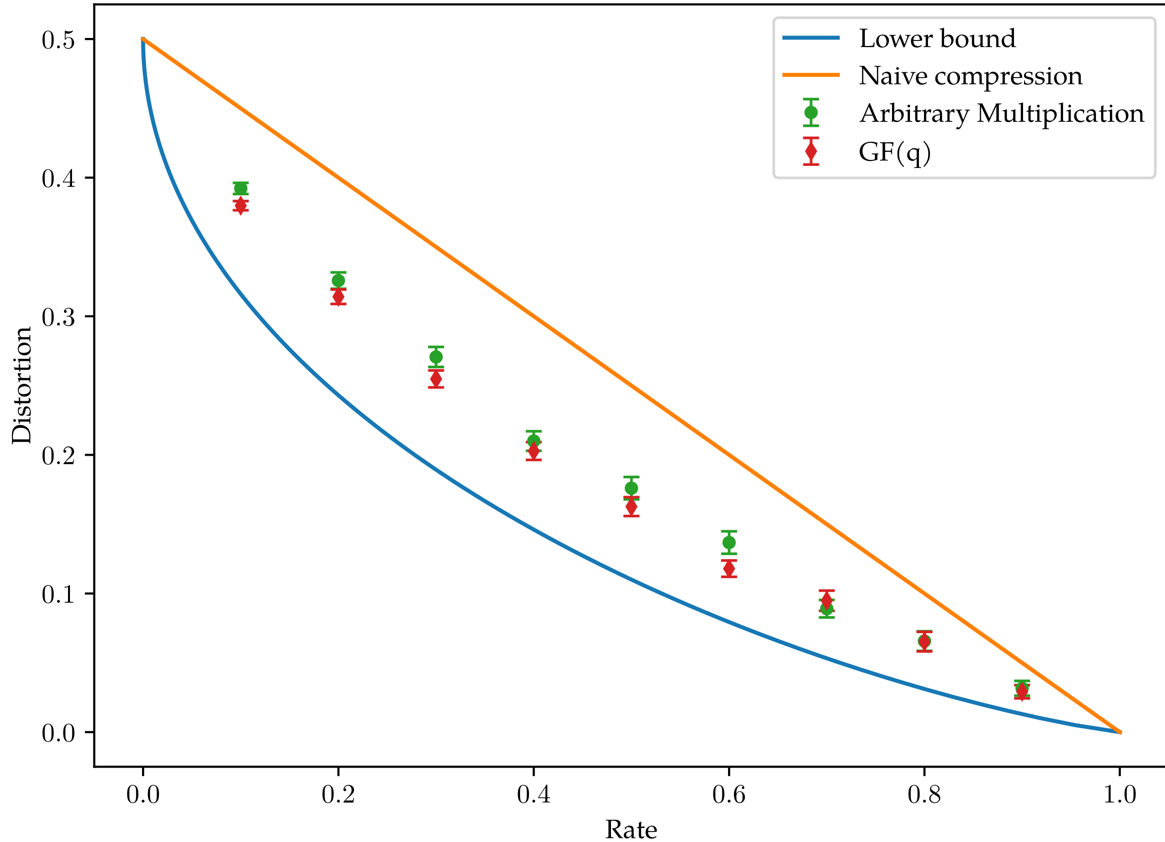


Figure 4.5: Comparison of average distortions obtained with random product tables and standard $\mathbb{GF}(q)$ multiplication. $q = 8, n = 2520, T_{max} = 4, b = 28, \gamma = 5 \cdot 10^{-3}$, average over 200 random {graph, source vector} instances, convergence criterion: parity.

Improvements coming from graph reduction, which were previously found for BP, were confirmed to be considerable also when using MS. The underlying reasons for such improvements remain unclear, although there are strong suggestions, reinforced by the reasoning about the computation tree involved in the proof, that exposing graph leaves facilitates BP in its task of resolving conflicting constraints. Graphs in which variables are binary and the corresponding nodes in the factor graphs have degree ≤ 2 are particularly suitable for this interpretation.

The strategy of generalizing binary variables to q-ary ones, already popular in the information theory community, has been confirmed to work well also for our compression scheme. In addition to this, we showed how the algebra ruling operations among these variables might not need to be that of $\mathbb{GF}(q)$ as it is usually done, but other options, like random multiplication tables, can be explored.

The connections with statistical physics, reviewed in the first chapter, allowed for seamless shifts between different representations of the same problem and provided consolidated tools such as the BP algorithm itself. Moreover, the spectrum of potential applications of our method crosses the boundaries of information theory, expanding to problems in the spin glass family.

As a future development, it would be interesting to understand more about the role of graph reduction in the modification of the energy landscape and in relation to the performances of BP, as well as the implications of working with higher-order fields.

Acknowledgements

Alfredo Braunstein, for offering me a thesis topic he was enthusiast about and for guiding me through it.
Giovanni Catania, for his help and the friday morning calls during lockdown.
Adrian, for the countless discussions about maths and physics and for inspiring me to be more Feynmanesque.
Uncle Franz, for his help with the abstract.

Bibliography

[Badasyan et al., 2010] Badasyan, A., Giacometti, A., Mamasakhlisov, Y. S., Morozov, V., and Benight, A. S. (2010). Microscopic formulation of the zimm-bragg model for the helix-coil transition. *Physical Review E*, 81(2):021921.

[Bayati et al., 2008] Bayati, M., Shah, D., and Sharma, M. (2008). Max-product for maximum weight matching: Convergence, correctness, and lp duality. *IEEE Transactions on Information Theory*, 54(3):1241–1251.

[Blume et al., 1971] Blume, M., Emery, V. J., and Griffiths, R. B. (1971). Ising model for the λ transition and phase separation in he^3-he^4 mixtures. *Phys. Rev. A*, 4:1071–1077.

[Braunstein et al., 2011] Braunstein, A., Kayhan, F., and Zecchina, R. (2011). Efficient data compression from statistical physics of codes over finite fields. *Physical Review E*, 84(5):051111.

[Braunstein et al., 2005] Braunstein, A., Mézard, M., and Zecchina, R. (2005). Survey propagation: An algorithm for satisfiability. *Random Structures & Algorithms*, 27(2):201–226.

[Davey and MacKay, 1998] Davey, M. C. and MacKay, D. J. (1998). Low density parity check codes over gf (q). In *1998 Information Theory Workshop (Cat. No. 98EX131)*, pages 70–71. IEEE.

[Declercq and Fossorier, 2005] Declercq, D. and Fossorier, M. (2005). Extended minsum algorithm for decoding ldpc codes over gf(/sub q/. In *Proceedings. International Symposium on Information Theory, 2005. ISIT 2005.*, pages 464–468.

[Etzion et al., 1999] Etzion, T., Trachtenberg, A., and Vardy, A. (1999). Which codes have cycle-free tanner graphs? *IEEE Transactions on Information Theory*, 45(6):2173–2181.

[Ford and Fulkerson, 1956] Ford, L. R. and Fulkerson, D. R. (1956). Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404.

[Frey and Koetter, 2000] Frey, B. J. and Koetter, R. (2000). Exact inference using the attenuated max-product algorithm. *Advanced mean field methods: Theory and Practice*, pages 213–228.

[Gallager, 1962] Gallager, R. (1962). Low-density parity-check codes. *IRE Transactions on information theory*, 8(1):21–28.

[Gallager, 1963] Gallager, R. G. (1963). *Low-Density Parity-Check Codes, Monograph.* MIT press.

[Lee and Yang, 1952] Lee, T. D. and Yang, C. N. (1952). Statistical theory of equations of state and phase transitions. ii. lattice gas and ising model. *Phys. Rev.*, 87:410–419.

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

[Luby et al., 2001] Luby, M. G., Mitzenmacher, M., Shokrollahi, M. A., and Spielman, D. A. (2001). Improved low-density parity-check codes using irregular graphs. *IEEE Transactions on information Theory*, 47(2):585–598.

[MacKay, 2003] MacKay, D. J. (2003). *Information theory, inference and learning algorithms*. Cambridge university press.

[MacKay, 2014] MacKay, D. J. (2014). Lecture 13: Approximating probability distributions (iii): Monte carlo methods (ii): Slice sampling. `https://youtu.be/Qr6tg9oLGTA?t=5701`.

[Mezard and Montanari, 2009] Mezard, M. and Montanari, A. (2009). *Information, physics, and computation*. Oxford University Press.

[Mézard et al., 2003] Mézard, M., Ricci-Tersenghi, F., and Zecchina, R. (2003). Two solutions to diluted p-spin models and xorsat problems. *Journal of Statistical Physics*, 111(3-4):505–533.

[Nelson, 2004] Nelson, P. (2004). *Biological physics*. WH Freeman New York.

[Shannon, 1948] Shannon, C. E. (1948). A mathematical theory of communication. *Bell system technical journal*, 27(3):379–423.

[Shannon, 1957] Shannon, C. E. (1957). Certain results in coding theory for noisy channels. *Information and control*, 1(1):6–25.

[Shannon, 1959] Shannon, C. E. (1959). Coding theorems for a discrete source with a fidelity criterion. *IRE Nat. Conv. Rec*, 4(142-163):1.

[Stein and Newman, 2013] Stein, D. L. and Newman, C. M. (2013). *Spin glasses and complexity*, volume 4. Princeton University Press.

[Weiss and Freeman, 2000] Weiss, Y. and Freeman, W. T. (2000). Correctness of belief propagation in gaussian graphical models of arbitrary topology. In *Advances in neural information processing systems*, pages 673–679.

[Wolsey and Nemhauser, 1999] Wolsey, L. A. and Nemhauser, G. L. (1999). *Integer and combinatorial optimization*, volume 55. John Wiley & Sons.