Formalization of Error-correcting Codes: from Hamming to Modern Coding Theory

Reynald Affeldt\textsuperscript{1} and Jacques Garrigue\textsuperscript{2}

\textsuperscript{1} National Institute of Advanced Industrial Science and Technology
\textsuperscript{2} Nagoya University

\textbf{Abstract.} By adding redundancy to transmitted data, error-correcting codes (ECCs) make it possible to communicate reliably over noisy channels. Minimizing redundancy and coding/decoding time has driven much research, culminating with Low-Density Parity-Check (LDPC) codes. At first sight, ECCs may be considered as a trusted piece of computer systems because classical results are well-understood. But ECCs are also performance-critical so that new hardware calls for new implementations whose testing is always an issue. Moreover, research about ECCs is still flourishing with papers of ever-growing complexity. With the ambition to provide means for implementers to perform verification and for researchers to firmly assess recent advances, we have been developing a formalization of ECCs using the SSReflect extension of the Coq proof-assistant. We report on the formalization of linear ECCs, duly illustrated with a theory about the celebrated Hamming codes and the verification of the sum-product algorithm for decoding the more recent LDPC codes.

1 Introduction

Error-correcting codes (ECCs) add redundancy to transmitted data to ensure reliable communication over noisy channels. Low-Density Parity-Check (LDPC) codes are ECCs discovered in 1960 by Gallager; they were not thoroughly studied until they were showed in the nineties to deliver good performance in practice. Since then, LDPC codes have found their way into modern devices such as hard-disk storage, wifi communications, mobile phones, etc. and have motivated a new body of works known as modern coding theory.

Implementations of ECCs cannot be crystallized as a generic library that can be deemed correct because extensively tested. Because ECCs are performance-critical, new implementations are required to take advantage of the latest hardware, so that testing is always an issue. Also, research (in particular about LDPC codes) is so active that correctness guarantees for cutting-edge ECCs are scattered in scientific publications of ever-growing complexity.

A formalization of ECCs could help implementers and researchers. First, it would first make possible verification of concrete implementations. Today, an implementer willing to perform formal verification should first provide a formal specification of what ECCs are supposed to achieve. To compare, this is more difficult than verification of cryptographic functions whose specification requires little infrastructure when they rely on number theory (e.g., [1]).
However, the formalization of ECCs is a difficult undertaking. They rely on a vast body of mathematics: probabilities, graphs, linear algebra, etc. Teaching material is rarely (if ever) structured as algebra textbooks; monographs and research papers do not provide details for the non-expert reader. It is therefore no wonder that researchers are seeking for means to firmly assess the correctness of their pencil-and-paper proofs: our work is part of such a project.

Still, there is previous work that we can take advantage of to formalize ECCs. The SSREFLECT/MATHCOMP library [8] provides big operators to formalize combinatorial results, a substantial formalization of linear algebra, and tools to reason about graphs. The formalization of the foundational theorems of information theory [2] provides us with basic definitions about channels and probabilities. Last, we are fortunate enough to have colleagues, expert in ECCs, who provided us with details about linear ECCs and LDPC codes [9, Chapters 7 and 9].

To the best of our knowledge, our effort is the first attempt at a systematic formalization of ECCs inside a proof-assistant. In Sect. 3, we formalize basic results about linear ECCs. Already at this point, some effort was spent in augmenting textbook definitions with their implicit assumptions. In Sect. 4, we formalize Hamming codes. In particular, we provide a concrete encoder and decoder and express the error rate in terms of a closed formula. In Sect. 5, we formalize the key properties of the sum-product algorithm, the reference algorithm for efficient decoding of LDPC codes. Finally, in Sect. 6, we apply our formalization to the verification of a concrete implementation of the sum-product algorithm, making our work the first formal verification of a decoding algorithm for an advanced class of ECCs.

2 Premises on Information Theory and Probabilities

2.1 Channels and Codes in Information Theory

We recall basic definitions from [2].

The most generic definition of a code is as a channel code: a pair of encoder/decoder functions. Messages to be encoded are taken from a finite type \( M \). Encoded messages (codewords) are represented by row vectors over a finite alphabet \( A \) (denoted by \( 'rV[A]_n \) in MATHCOMP). The decoder (that may fail) is fed with the outputs of a noisy channel that are also represented by row vectors (possibly over a different alphabet \( B \)):

\[
\text{Definition } \text{encT} := \{ \text{ffun } M \rightarrow 'rV[A]_n \}.
\]

\[
\text{Definition } \text{decT} := \{ \text{ffun } 'rV[B]_n \rightarrow \text{option } M \}.
\]

\[
\text{Record code := mkCode } \{ \text{enc : encT ; dec : decT } \}.
\]

A (discrete) noisy channel is modeled as a stochastic matrix that we formalized as a function from the input alphabet \( A \) to probability distributions over the output alphabet \( B \):

\[
\text{Notation } '\mathcal{CH}(A, B)' := (A \rightarrow \text{dist } B).
\]

dist is the type of probability distributions. It is essentially a function from some finite type to non-negative reals whose outputs sum to 1 (the big operator \( \sum_{(x \in P)} f \) comes from MATHCOMP):

\[3\] Grants-in-Aid for Scientific Research, Project Number: 25289118.
Record dist := mkDist {
  pmf :> A → R+ : (* → R+ is a notation *)
  pmf1 : Σ_(a in A) pmf a = 1 }.

Given a distribution P, the probability of an event (represented by a finite set of elements: type {set A} in MATHCOMP) is formalized as follows:

Definition Pr P (E : {set A}) := Σ_(a in E) P a.

Communication of n characters is thought of as happening over the n\textsuperscript{th} extension of the channel, defined as a function from input vectors to distributions of output vectors:

Notation "CH\textsubscript{n}(A, B)" := ( _rV[A] \_n → {dist \_rV[B] \_n} ).

We deal with discrete memoryless channels (DMCs), where the output probability of a character does not depend on preceding inputs. The DMC corresponding to a channel W is defined by associating to an input vector x a distribution of output vectors with the following probability mass function:

Definition f x : _rV \_n) := ∏_(i < n) W(x /_i) (y /_i).

where x /_i represents the i\textsuperscript{th} element of the vector x. The notation W^n(y | x) (W^n(y|x) in pencil-and-paper proofs) is the probability for the DMC of W that an input x (of length n) is output as y.

Finally, the quality of a code c for a given channel W is measured by its error rate (notation: \(\bar{e}_{cha}(W, c)\)), that is defined by the average probability of errors:

Definition CodeErrRate (W : CH_1(A, B)) c := 1 / INR # M | ∑_(m in M) ErrRateCond W c m.

\(W^n(y | x)\) is the distribution of outputs corresponding to the codeword \(\text{enc} c m\) of a message m sent other the DMC of W. \[\{set y | \text{dec} c y \neq \text{Some m}\}\] is the (finite) set of outputs that do not decode to m. INR injects naturals into reals.

2.2 Aposteriori Probability

Probabilities are used to specify the correctness of probabilistic decoders such as the sum-product algorithm (see Sect. 5).

We first formalize the notion of aposteriori probability: the probability that an input was sent knowing that some output was received. It is defined via the Bayes rule from the probability that an output was received knowing that some input was sent. For an input distribution P and a channel W, the aposteriori probability of an input x given the output y is:

\[P^W(x|y) := \frac{P(x)W^n(y|x)}{\sum_{x' \in A^*} P(x')W^n(y|x')}\]

We formalize aposteriori probabilities with the following probability mass function:

Definition den := Σ_(x in _rV \_n) P x * W \_n (y \mid x).

Definition f x := P x * W \_n (y \mid x) / den.
This probability is well-defined when the denominator is not zero. This is more than a technical hindrance: it expresses the natural condition that, since \( y \) was received, then necessarily a suitable \( x \) (i.e., such that \( P_x \neq 0 \) and \( W^n(y \mid x) \neq 0 \)) was sent beforehand. The denominator being non-zero is thus equivalent to the receivable condition:

**Definition receivable** \( y \) := \( \exists x. (P_x \neq 0) \land (W^n(y \mid x) \neq 0) \).

In Coq, we denote aposteriori probabilities by \( P'^W H(x \mid y) \) where \( H \) is a proof that \( y \) is receivable.

The goal of the sum-product algorithm is to find the input bits that maximizes the aposteriori probabilities. The probability that the \( n \)th bit of the input \( x \) is set to \( b \) (0 or 1) given the output \( y \) is defined by the marginal aposteriori probability (\( K \) is chosen so that it is indeed a probability):

\[
P^W_m(b \mid y) := K \sum_{x \in \mathbb{F}_2^n} P^W(x \mid y)
\]

In Coq, we will denote this probability by \( P'_{n0} H(x \mid y) \) where \( H \) is the proof that \( y \) is receivable. See [3] for complete formal definitions.

### 3 A Formal Setting for Linear ECCs

Linear ECCs are about bit-vectors, i.e., vectors over \( \mathbb{F}_2 \) (we use the notation \( 'rV[\mathbb{F}_2\_n] \) from MATHCOMP). Their properties are mostly discussed in terms of Hamming weight (the number of 1 bits) or of Hamming distance (the number of bits that are different).

In Coq, we provide a function \( wH \) for the Hamming weight, from which we derive the Hamming distance:

**Definition dH** \( n(x y : 'rV_n) := wH(x - y) \).

### 3.1 Linear ECCs as Sets of Codewords

The simplest definition of a linear ECC is as a set of codewords closed by addition (\( n \) is called the length of the code):

**Record lcode0 n := mkLcode0 {**
  **codewords :> set 'rV[\mathbb{F}_2\_n] ;**
  **lclosed : addr_closed codewords }.**

In practice, a linear ECC is defined as the kernel of a parity check matrix (PCM), i.e., the matrix whose rows correspond to the checksum equations that codewords fulfill (\( m \) is multiplication and \( ^T \) is transposition of matrices):

**Definition syndrome** \( (H : 'M[\mathbb{F}_2\_n](m, n)) (y : 'rV_n) := H * m y^T.**

**Definition kernel** \( H := [set c \mid syndrome H c = 0 ] \).

Since the kernel of the PCM is closed by addition, it defines a linear ECC:

**Lemma kernel_add** \( H : addr_closed (kernel H). Proof. ... Qed.**

**Definition lcode0_kernel** \( H := mkLcode0 (kernel_add H).**

When \( H \) is a \( m \times n \) matrix, \( k = n - m \) is called the dimension of the code.

A code is trivial when it is reduced to the singleton with the null vector:
Definition not_trivial := ∃ cw, (cw ∈ C) ∧ (cw ≠ 0).

When a linear ECC is not trivial (proof non_trivial_code below), one can define the minimum distance between any two codewords, or, equivalently, the minimum weight of non-zero codewords, using SSREFLECT’s xchoose and arg_min functions:

Definition non_0_codeword := xchoose C_not_trivial.
Definition min_wH_codeword :=
arg_min non_0_codeword [pred cw in C | wH cw ≠ 0] wH.
Definition d_min := wH min_wH_codeword.

The minimum distance $d_{\text{min}}$ defines in particular the number of errors $\left\lfloor \frac{d_{\text{min}}-1}{2} \right\rfloor$ that one can correct using minimum distance decoding (see Sect. 3.3):

Definition mdd_err_cor :=
if odd d_min then (d_min - 1)/2 else (d_min - 2)/2.

3.2 Linear ECCs with Coding and Decoding Functions

In practice, a linear ECC is not only a set of codewords but also a pair of coding and decoding functions to be used in conjunction with a channel as illustrated by Fig. 1 [5, p. 16]. We combine the definition of a linear ECC as a set of codewords (Sect. 3.1) and as a pair of encoding and decoding functions (i.e., a channel code—Sect. 2.1) with the hypotheses that (1) the encoder is injective and (2) the image of the encoder is a subset of the codewords:

Record lcode n k : Type := mkLcode {
  lcode0_of => lcode0 n ;
  enc_dec := code 'F_2 'F_2 'rV['F_2]_k n ;
  enc_inj : injective (enc enc_dec) ;
  enc_img : enc_img_in_code lcode0_of (enc enc_dec) }.

enc_img_in_code is the hypothesis that the image of the messages (here, 'rV['F_2]_k) by the encoder (enc enc_dec) is included in the set of codewords (here, lcode0_of).

As indicated by Fig. 1, the decoder is decomposed into (1) a function that repairs the received output and (2) a function that discards the redundancy bits:
Record lcode1 n k := mkLcode1 { lcode_of := lcode n k; repair := repairT n; discard := discardT n k; dec_is_repair_discard : → dec lcode_of = [ffun y ⇒ omap discard (repair y)]; enc_discard_is_id : cancel_on lcode_of (enc lcode_of) discard }. enc_discard_is_id is a proof that discard followed by encoding enc is the identity over the domain lcode_of.

Example The r-repetition code encodes one bit by replicating it r times. It has therefore only two codewords: 00···0 and 11···1 (r times). The PCM can be defined as \( H = A \| 1 \) where \( A \) is a column vector of \( r - 1 \) 1’s, and the corresponding encoder is the matrix multiplication by \( G = 1 \| (-A)^T \). More generally, let \( A \) be a \((n-k) \times k\) matrix, \( H = A \| 1 \) and \( G = 1 \| (-A)^T \). Then \( H \) is the PCM of a \((n,k)\)-code with the (injective) encoding function \( x \mapsto x \times G \). Such a linear ECC is said to be in systematic form.

3.3 The Variety of Decoding Procedures

There are various strategies to decode the channel output. Minimum distance decoding chooses the closest codeword in terms of Hamming distance. When such a decoder decodes an output \( y \) to a message \( m \), then there is no other message \( m' \) whose encoding is closer to \( y \):

Definition minimum_distance_decoding := ∀ y m, (dec c) y = Some m → ∀ m', dH ((enc c) m) y ≤ dH ((enc c) m') y.

We can now formalize the first interesting theorem about linear ECCs [11, Theorem 2, p. 10] that shows that a minimum distance decoder can correct mdd_err_cor (defined in Sect. 3.1) errors:

Lemma encode_decode m y : (dec c) y \neq None → dH ((enc c) m) y ≤ mdd_err_cor C_not_trivial → (dec c) y = Some m.

For example, a repetition code can decode \((r - 1)/2\) errors (with \( r \) odd) since minimum distance decoding can be performed by majority vote (see [3] for formal proofs):

Definition majority_vote r (s : seq F_2) : option F_2 := let cnt := num_occ 1 s in if r./2 < cnt then Some 1 else if (r./2 = cnt) \& \& odd r then None else Some 0.

Maximum likelihood (ML) decoding decodes to the message that is the most likely to have been encoded according to the definition of the channel. More precisely, for an encoder \( f \), an ML decoder \( \phi \) is such that \( W^n(y|f(\phi(y))) = \max_{m \in M} W^n(y|f(m)) \):

Definition maximum_likelihood_decoding := ∀ m, (dec c) y = Some m \& \& \& \exists n, (enc c) m = \\backslash \\text{max}_y (m' \in M) W^n (y | (enc c) m').
The assumption receivable \( WP y \) says that we consider outputs with non-zero probability (same assumption as in Sect. 2.2). The assumption support \((\text{enc} \ c)\) says that only codewords can be input. Textbooks do not make these assumptions explicit but they are essential to complete formal proofs.

ML decoding is desirable because it achieves the smallest error rate among all the possible decoders [3, lemma ML_smallest_err_rate]. Still, it is possible to achieve ML decoding via minimum distance decoding. This is for example the case with a binary symmetric channel (that inputs and outputs bits) with error probability \( p < \frac{1}{2} \). Formally, for a code \( c \) with at least one codeword:

\[
\text{Lemma MD_implies_ML : } p < \frac{1}{2} \Rightarrow \text{minimum_distance_decoding } \Rightarrow \text{maximum_likelihood_decoding } W c P.
\]

Maximum a posteriori probability (MAP) decoding decodes to messages that maximize the a posteriori probability (see Sect. 2.2). MAP decoding is desirable because it achieves ML decoding [3, lemma MAP_implies_ML]. Maximum posterior marginal (MPM) decoding is similar to MAP decoding: it decodes to messages such that each bit maximizes the marginal a posteriori probability. The sum-product algorithm of Sect. 5 achieves MPM decoding.

4 Formalization of Hamming Codes and Their Properties

We formalize Hamming codes. In particular, we show that the well-known decoding procedure for Hamming code is actually a minimum distance decoding and that the error rate can be stated as a closed formula.

**Formal Definition** Hamming codes are \((n = 2^m - 1, k = 2^m - m - 1)\) linear ECCs, i.e., one adds \( m \) extra bits for error checking. The codewords are defined by the PCM whose columns are the binary representations of the \( 2^m - 1 \) non-null words of length \( m \). For a concrete illustration, here follows the PCM of the \((7,4)\)-Hamming code:

\[
\text{hamH}_{7,4} = \begin{pmatrix}
0 & 0 & 0 & 1 & 1 & 1 & 1 \\
0 & 1 & 1 & 0 & 0 & 1 & 1 \\
1 & 0 & 1 & 0 & 1 & 0 & 1
\end{pmatrix}
\]

Formally, for any \( m \), we define the PCM using a function \( \text{nat2bin_cV} \) that builds column vectors with the binary representation of natural numbers (e.g., for the matrix \( H \) above, \( \text{nat2bin_cV} 3 \ 1 \) returns the first column vector, \( \text{nat2bin_cV} 3 \ 2 \) the second, etc.):

\[
\text{Definition hamH} := \backslash \text{matrix}(i < m, j < n) (\text{nat2bin_cV} m j+1 i 0).
\]

**Definition** \( \text{hamC} : \text{lcode0 } n := \text{lcode0_kernel hamH} \).

**Minimum Distance** The minimum distance of Hamming codes is 3, and therefore, by minimum distance decoding. Hamming codes can correct 1-bit errors (by the lemma encode_decode of Sect. 3.3). The fact that the minimum distance is 3 is proved by showing that there are no codewords of weights 1 and 2 (by analysis of \( H \)) while there is a codeword of weight 3 \((7 \times 2^{n-3} = (1110 \cdots 0)_2\), where \( n = 2^m - 1 \) is the length of the code). Hamming codes are therefore not trivial and their minimum distance is 3:
Lemma hamming_not_trivial : not_trivial hamC.
Lemma minimum_distance_is_3 : d_min hamming_not_trivial = 3.

Minimum Distance Decoding The procedure of decoding for Hamming codes is well-known. To decode the output $y$, compute its syndrome: if it is non-zero, then it is the binary representation of the index of the bit to flip back. The function `ham_detect` computes the index $i$ of the bit to correct and prepare a vector to repair the error. The function `ham_repair` fixes the error by adding this vector:

Definition ham_detect y :=
  let i := bin2nat_cV (syndrome hamH y) in
  if i is 0 then 0 else nat2bin_rV n (2 ^ (n - i)).
Definition ham_repair : decT _ _ m := ffun y =>
  let ret := y + ham_detect y in
  if syndrome hamH ret = 0 then Some ret else None.

Let `ham_scde` be a linear ECC using the `ham_repair` function. We can show that it implements minimum distance decoding:

Lemma hamming_MD : minimum_distance_decoding ham_scde.
It is therefore an ML decoding (by the lemma MD_implies_ML from Sect. 3.3).

The Encoding and Discard Functions We now complete the formalization of Hamming codes by providing an encoding and discard functions (as in Fig. 1). Modulo permutation of the columns, the PCM of Hamming codes can be transformed into systematic form $sysH = sysA||1$ (as explained in the example about repetition codes in Sect. 3.2). This provides us with a generating matrix $sysG = 1||(-sysA)^T$. For illustration in the case of the $(7,4)$-Hamming code:

$$sysH_{7,4} = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}$$  

$$sysG_{7,4} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

Let `sysH_perm` be the column permutation that turns $H$ into $sysH$. The parity check and generating matrices in systematic form are formalized as follows:

Definition sysH : 'M[ 'F_2 ]_(m, n) := col_perm sysH_perm hamH.
Definition sysG :=
  castmx (eref1, subnK (m_len m')) (row_mx 1%:M (-sysA)^T).
(castmx is a cast that deals with dependent types.) The discard function in systematic form is obvious:

Definition sysDiscard : 'M[ 'F_2 ]_(n - m, n) :=
  castmx (eref1, subnK (m_len m')) (row_mx 1%:M 0).

Using the column permutation `sysH_perm` the other way around, we can produce the discard function and the generating matrix corresponding to the original `hamH`:

Definition ham_discard := col_perm sysH_perm^-1 sysDiscard.
Definition hamG := col_perm sysH_perm^-1 sysG.
Coupled with the `ham_repair` function above, `hamG` and `ham_discard` provides us with a complete definition of Hamming encoders and decoders:

```plaintext
Definition ham_channel_code := mkCode [ffun t => t * m hamG] [ffun x => omap ham_discard (ham_repair _ x)].
```

**Error Rate** Finally, we show, in the case of a binary symmetric channel $W$, that the error rate (see Sect. 2.1) of Hamming codes can be expressed as a closed formula:

```plaintext
Lemma hamming_error_rate : p < 1/2 -> echa (W, ham_channel_code) = 1 - ((1 - p)^n - INR n * p + ((1 - p)^n - 1)).
```

The existence of codes with arbitrary small error rates is the main result of Shannon’s theorems. But Shannon’s proofs are not constructive. Our formalization of Hamming codes with a closed formula for their error rate provides us with a concrete candidate for the codes Shannon was anticipating.

5 Formalization of the Properties of Sum-product Decoding

The sum-product algorithm provides efficient decoding for LDPC codes. It decides each decoded bit using an estimation of its marginal aposteriori probability that is obtained by propagating “messages” in a graph corresponding to the PCM. We explain those graphs in Sect. 5.1, the summary notation used to specify the sum-product algorithm in Sect. 5.2, and the main properties of the sum-product algorithm in Sect. 5.3.

5.1 Parity Check Matrix as Tanner Graphs

The vertices of a Tanner graph correspond to the rows and columns of a parity-check matrix $H$ with an edge between $m$ and $n$ when $H_{m,n} = 1$. Rows are called function nodes and columns are called variables nodes. By construction, a Tanner graph is bipartite.

Sets of successor nodes and subgraphs of Tanner graphs appear as indices of big operators in the definitions and proofs of the sum-product algorithm.

Let $g$ be a graph (formalized by a binary relation) and $m$ and $n$ be two connected vertices. The subgraph rooted at the edge $m$–$n$ is the set of vertices reachable from $m$ without passing through $n$:

```plaintext
Definition except n := [rel x y | g x y ∧ (x ≠ n) ∧ (y ≠ n)].
Definition subgraph m n := [set v | g n m ∧ connect (except n) m v].
```

For Tanner graphs, we distinguish successors and subgraphs of variable nodes and of function nodes. We denote the successors of the function (resp. variable) node $m0$ (resp. $n0$) by $'V$ $m0$ (resp. $F$ $n0$). We denote the functions nodes of the subgraph rooted at edge $m0$–$n0$ by $'F$(m0, n0). Similarly, we denote the variables nodes of the subgraph rooted at edge $m0$–$n0$ (to which we add $n0$) by $'V$(m0, n0). Fig. 2 provides an explanatory illustration, see [3] for complete definitions.

It will be important to distinguish acyclic Tanner graphs:

```plaintext
Definition acyclic g := ∀ l, 2 < size l → ¬ path.ucycle g l.
```

Technically, we will need to establish partition properties when proving the properties of the sum-product decoding algorithm (see Sect. 5.3 for a concrete example).
5.2 The Summary Notation

The summary notation $\sum_{n_0} \cdots$ is a special summation used in modern coding theory [12]. It is a sum over a set of vectors, say $[x_0, x_1; \ldots; x_{n-1}]$, whose $n_0$th element is fixed. The precise set of varying $x_i$ is implicit but can be inferred by looking at the $x_i$ appearing below the summation sign. This notation saves the practitioner “from a flood of notation” [12, p. 49], for example by writing steps such as:

$$\prod_{m_0 \in F(m_0)} \sum_{n_0} \cdots = \sum_{n_0} \prod_{m_0 \in F(m_0)} \cdots,$$

the reader being trusted to understand that both summary notations sum over different sets. This is difficult to achieve in a proof-assistant unless ones reflects most syntax.

The alternative summary notation $\sum_{(i \neq n_0)} \cdots$ makes the bound variables explicit (e.g., [9]). They are the vectors of length $|s|$ obtained by picking up from the vector $x$ only the elements at index $i \in s$, the contents at index $n_0$ being fixed to some value:

$$\prod_{m_0 \in F(m_0)} \sum_{x \in V(m_0, n_0)} \cdots = \sum_{n_0} \prod_{m_0 \in F(m_0)} \cdots$$

This summary notation suggests to work with vectors $x$ of varying sizes, which can be an issue when the size of vectors appears in dependent types (tuples or row vectors in MATHCOMP). Moreover, this notation is not clear about the contents at index $n_0$.

In our formal development, we work with vectors of size $n$ but we look only at a subset of their indices; for indices at which the contents is fixed, we provide a default vector $d$ to look at. Using big operators, this amounts to sum over all vectors $t$ of size $n$ such that $t$ agrees with $d$ outside of a set of indices $s$:

**Definition**: summary \((s : \{\text{set } i, n\}) (d x : x V[A] n) : = \forall i, (i \in s) \Rightarrow (x / i = d / i))\).

**Notation**: $\sum_{(x \# s \_ \_ d \_ \_ e)} := (\sum_{x \mid \text{summary } s d x} e)$

Fig. 2. Successors and subtrees in an acyclic Tanner graph
Thanks to this encoding, we can benefit from MATHCOMP lemmas about big operators to prove the properties of the sum-product decoding in Sect. 5.3.

Alternatively, the summary notation \( \sum_{(x \neq a)} e \) can also be thought as \( \sum_{x_1 \in F_1} \cdots \sum_{x_m \in F_m} e \) \( d[x_1 := x_1] \cdots [x_m := x_m] \) where \( d[i := a] \) represented the vector \( d \) with index \( i \) updated with \( a \) (\( \text{enum} s \) being the list \( [s_1,s_2,\ldots,s_m] \)).

**Definition summary_fold**(s : 
\[s \in \{ \text{set } 'I_n \}\}] d e :=
foldr \( \text{fun} \ n0 \ F \ t = \sum_{(b \ in \ 'F,2)} F \ (t \ ['n0 := b]) \) e \( \text{enum} s \) d.

This is equivalent \( \sum_{(x \neq a)} e = \text{summary} \_ \text{fold} a d e \) but we found it easier to use \text{summary} \_ \text{fold} to prove our implementation of the sum-product algorithm in Sect. 6.

### 5.3 Properties of the Sum-product Decoding

**Correctness of the Estimation** Let us consider a channel \( W \), an output \( y \) we want to decode, and let us denote by \( x \) the input codeword we are looking for. When decoding, we are concerned with evaluating \( P_n^W(x|y) \). In the following, we show that it is proportional to the following quantity:

\[
P_n^W(x|y) \propto W(y_n|x) \prod_{m0 \in F(n0)} \alpha_{m0,n0}(x)
\]

\( \alpha_{m0,n0}(x) \) is a value (see below) computed using information from the subtree rooted at the edge \( m0 \_ n0 \) (we assume that the Tanner graph is acyclic).

We now provide the formal statement. Let \( \mathcal{W} \) be a channel. Let \( \mathcal{H} \) be a \( m \times n \) PCM such that the corresponding Tanner graph is acyclic (hypothesis \text{acyclic} \_ \text{graph}(\text{tanner} \_ \text{rel} \mathcal{H})) where \text{tanner} \_ \text{rel} turns a PCM into the corresponding Tanner graph). Let \( y \) be the message to decode. We assume that it is receivable (hypothesis \text{receive} \_ y below, see Sect. 2.2). Finally, let \( a \) be the vector used in the summary notation. Then the posteriori probability \( P_n^W(x|y) \) can be evaluated by a closed formula:

**Lemma** estimation\_correctness \( (d : 'r \mathcal{V},n) \ n0 : \)
\[\text{let } x := d / \_ n0 \text{ in let } P := 'U \text{\_ not\_empty } \text{in}
\]
\[P : 'n0 \cdots 'w , \text{Hy} (x \ y) = \text{Kmpp} \text{Hy} * \text{Kpp} 'w \ y = \_ 'w x (y / \_ n0) * \text{INR} (m0 \text{in } 'F n0) \alpha m0 n0 d .
\]

\( \text{Kmpp} \) and \( \text{Kpp} \) are normalization constants (see [3]). \( P \) is a uniform distribution. The distribution \'U \text{\_ not\_empty} \) of codewords has following probability mass function:
\( cw \_ \mapsto 1/|C| \) if \( cw \in C \) and 0 otherwise. The definition of \( \alpha \) messages uses the summary notation: it is a sum over vectors \( x \) projected on the set \'V(m0, n0) : \n0 \) of products (containing products over function nodes belonging to \'F(m0, n0)). Formally:

**Definition** \( \alpha m0 n0 d := \sum_{(x \ # 'V(m0, n0) : \n0 , d)} \)
\[ \text{INR} (\delta ('V m1) x) .
\]

where \( \delta s t \) performs checksum checks:

**Definition** \( \delta n (s : \{ \text{set } 'I_n \}\}) (v : 'r \mathcal{V}[ 'F,2],n) :=
\[ (\text{\_ \_ \_ } + \% R/Zp0) \_ \_ n0 \text{ in } s \) v / \_ nu = Zp0 .
Let us comment about two technical aspects of the proof of estimation_correctness. The first one is the need to instrument Tanner graphs with partition lemmas to be able to decompose big sums/prods. See the next paragraph on lemma message_computation for a concrete example. The second one is the main motivation for using the summary notation. We need to make big sums commute with big prods in equalities like:

\[
\prod_{m0 \text{ in } \mathcal{F}(m0, n0)} \sum_{x \# \mathcal{V}(m0, n0) : \mathcal{n0, d}} \cdots = \\
\sum_{x \# \mathcal{setT} : \mathcal{n0, d}} \prod_{m0 \text{ in } \mathcal{F}(n0)} \cdots
\]

Such steps amount to apply the MATHCOMP lemma big_distr_big_dep together with technical reindexing. This is one of our contributions to provide lemmas for such steps.

**Computation of Messages** The property above provides a way to evaluate \( P_{m0}^W(x|y) \) but not an efficient algorithm because the computation of the \( \alpha_{m0,n0}(x) \) messages are about the subgraph rooted at the edge \( m0-n0 \). The second property that we formalize introduces \( \beta \) messages such that \( \alpha \)'s (resp. \( \beta \)'s) can be computed from neighboring \( \beta \)'s (resp. \( \alpha \)'s). This is illustrated by Fig. 3 whose meaning will be made clearer in Sect. 6.

We define \( \beta \) messages using \( \alpha \) messages as follows:

**Definition** \( \beta \ m0 \ n0 \ (d : \mathcal{rV}_n) := \) \( W \ (d \ / \ n0) \ (y \ / \ n0) \ast \prod_{m1 \ in \ \mathcal{F}(n0)} \alpha \ m1 \ n0 \ d. \)

We prove that \( \alpha \)'s can be computed using \( \beta \)'s by the following formula (we assume the same setting as for the lemma estimation_correctness):

**Lemma** message_computation \( m0 \ n0 \ d : n0 \in \mathcal{V} m0 \rightarrow \) 
\( \alpha \ m0 \ n0 \ d = \sum_{x \# \mathcal{V} m0 \ : \ m0, d} \mathcal{INR} (\delta \ (\mathcal{V} m0 \ : \ x) \ast \prod_{n1 \ in \ \mathcal{V} m0 \ : \ n0} \beta \ n1 \ m0 \ x. \)

This proof is technically more involved than the lemma estimation_correctness but relies on similar ideas: partitions of Tanner graphs to split big sums/prods and computations of big sums and big prods using the summary notation. Let us perform the first proof step for illustration. It consists in turning the inner product of \( \alpha \) messages \( \prod_{m1 \ in \ \mathcal{F}(m0, n0) : \ m0} \mathcal{INR} (\delta \ (\mathcal{V} m1 \ : \ x)) \) into:

\[
\prod_{n1 \ in \ \mathcal{V} m0 \ : \ n0} \prod_{m1 \ in \ \mathcal{F}(n1, m0)} \mathcal{INR} (\delta \ (\mathcal{V} m1 \ : \ x))
\]

This is a consequence of the fact that \( \mathcal{F}(m0, n0) : \ m0 \) can be partitioned (when \( H \) is acyclic) into smaller \( \mathcal{F}(n1, m0) \) where \( n1 \) is a successor of \( m0 \) and \( m1 \) is a successor of \( n1 \), i.e., according to the following partition:

**Definition** Fgraph_part_Fgraph \( m0 \ n0 : \{\text{set } \{\text{I}_m\}\} := \) 
\( \{\text{fun } n1 \rightarrow \bigcup_{m1 \ in \ \mathcal{F}(n1, m0)} \mathcal{F}(m1, n1)\} \ast \mathcal{INR} (\delta \ (\mathcal{V} m0 \ : \ n0)). \)

Once Fgraph_part_Fgraph m0 n0 has been shown to cover \( \mathcal{F}(m0, n0) : \ n0 \) with pairwise disjoint sets, this step essentially amounts to use the lemmas big_trivIset and big_imset from MATHCOMP. See [3, tanner_partition.v] for related lemmas.

6 Implementation and Verification of Sum-product Decoding

An implementation of sum-product decoding takes as input a Tanner graph and an output \( y \), and computes for all variable nodes, that represent a bit of the decoded codeword,
the opposite direction (starting from the root that time). Fig. 3 illustrates this.

Concretely, we provide Coq functions to build the tree, compute the messages, and extract the estimations, and prove formally that the results indeed agree with the definitions from Sect. 5.3.

**Definition of the tree** Function nodes and variables nodes share the same data structure, and are just distinguished by their kind.

Definition R2 := (R × R)%type.

Inductive kind : Set := kf | kv.

Fixpoint negk k := match k with kf ⇒ kv | kv ⇒ kf end.

Inductive tag : kind → Set := Func : tag kf | Var : R2 → tag kv.

Inductive tn_tree (k : kind) (U D : Type) : Type :=
Node { node_id : id; node_tag : tag k;
  children : seq (tn_tree (negk k) U D);
  up : U; down : D }.

This tree is statically bipartite, thanks to the switching of the kind for the children. Additionally, in each variable node, node_tag is expected to contain the channel probability for this bit to be 0 or 1, i.e., the pair \( (W(0|y_n), W(1|y_n)) \). The up and down fields are to be filled with the values of \( \alpha \) and \( \beta \) messages (according to the kind), going to the parent node for \( \text{up} \), and coming from it for \( \text{down} \). Here again we will use pairs of the 0 and 1 cases. Note that the values of messages need not be normalized.

**Computation of \( \alpha \) and \( \beta \) messages** The function \( \alpha, \beta \) takes as input the tag of the source node of the message, and the messages received from neighboring nodes, ex-
cluding the destination, and computes either \( \alpha \) or \( \beta \), according to the tag. Thanks to this function, the remainder of the algorithm keeps a perfect symmetry between variable and function nodes.

**Definition** \( \alpha_{\text{op}} \) (\( \text{out} \ \text{inp} : \mathbb{R}^2 \)) :=
\[
\text{let} \quad (o, o') := \text{out} \quad \text{in} \quad \text{let} \quad (i, i') := \text{inp} \quad \text{in} \\
(o \cdot i + o' \cdot i', o \cdot i' + o' \cdot i) .
\]

**Definition** \( \beta_{\text{op}} \) (\( \text{out} \ \text{inp} : \mathbb{R}^2 \)) :=
\[
\text{let} \quad (o, o') := \text{out} \quad \text{in} \quad \text{let} \quad (i, i') := \text{inp} \quad \text{in} \\
(o \cdot i', o' \cdot i) .
\]

**Definition** \( \alpha_\beta k \) (\( t : \text{tag} k \)) : seq \( \mathbb{R}^2 \) \( \rightarrow \) \( \mathbb{R}^2 \) :=
\[
\text{match} \ t \ \text{with} \\
\mid \text{Func} \Rightarrow \text{foldr} \ \alpha_{\text{op}} (1, 0) \\
\mid \text{Var} \ v \Rightarrow \text{foldl} \ \beta_{\text{op}} v
\end{align*}
\]

The definition for \( \beta \) is clear enough: assuming that \( v \) contains the channel probabilities for the corresponding bit, it suffices to compute the product of these probabilities with the incoming messages. For \( \alpha \), starting from the message_computation lemma, we remark that assuming a bit to be 0 leaves the parity unchanged, while assuming it to be 1 switches the parities. This way, the sum-of-products can be computed as an iterated product, using \( \alpha_{\text{op}} \). This optimization is described in [10, Sect. 5-E]. We will of course need to prove that these definitions compute the same \( \alpha \)'s and \( \beta \)'s as in Sect. 5.3.

**Propagation of messages** \( \text{sumprod_up} \) and \( \text{sumprod_down} \) compute respectively the contents of the up and down fields.

**Fixpoint** \( \text{sumprod_up} \) (\( n : \text{tn_tree} \ k \ \text{unit} \ \text{unit} \)) :
\( \text{tn_tree} \ k \ \mathbb{R}^2 \ \text{unit} \) :=
\[
\text{let} \ l := \text{map} \ \text{sumprod_up} \ (\text{children} \ n) \ \text{in} \\
\text{let} \ \text{out} := \alpha_\beta \ (\text{node_tag} \ n) \ (\text{map} \ \text{up} \ l) \ \text{in} \\
\text{Node} \ (\text{node_id} \ n) \ (\text{node_tag} \ n) \ l \ \text{out} \ \text{tt} .
\]

**Fixpoint** \( \text{seqs_but1} \) (\( a \ b : \text{seq} \ \mathbb{R}^2 \)) :=
\[
\text{if} \ b \ \text{is} \ h : : t \ \text{then} \ (a++t) : : \text{seqs_but1} \ (\text{rcons} \ a \ h) \ t \ \text{else} \ [::].
\]

**Fixpoint** \( \text{sumprod_down} \) (\( n : \text{tn_tree} \ k \ \mathbb{R}^2 \ \text{unit} \)) (\( \text{down} : \text{option} \ \mathbb{R}^2 \)) :
\( \text{tn_tree} \ k \ \mathbb{R}^2 \ \mathbb{R}^2 \) :=
\[
\text{let} \ \text{args} := \text{seqs_but1} \ \text{arg0} \ (\text{map} \ \text{up} \ (\text{children} \ n)) \ \text{in} \\
\text{let} \ \text{funs} := \ [\text{seq} \ (\text{fun} \ a \Rightarrow \text{sumprod_down} \ n' \ (\text{Some} \ (\alpha_\beta \ (\text{node_tag} \ n) \ a))) \\
\mid \ n' \leftarrow \text{children} \ n] \ \text{in} \\
\text{let} \ \text{children} := \text{apply_seq} \ \text{funs} \ \text{args} \ \text{in} \\
\text{Node} \ (\text{node_id} \ n) \ (\text{node_tag} \ n) \ \text{children} \ \text{up} \ n \ \text{down} .
\]

**Definition** \( \text{sumprod_all} \) (\( k \ n := \text{sumprod_down} \ (@\text{sumprod_up} \ k \ n) \ \text{None} \). The \text{down} argument is None for the root of the tree, or the message coming from the parent node otherwise. \text{apply_seq} applies a list of functions to a list of arguments. This is a workaround to allow defining \( \text{sumprod_down} \) as a fixpoint.

**Building the tree** A parity-check matrix \( H \) and the probability distribution \( rW \) for each bit (computed from the output \( y \) and the channel \( W \)) is turned into a \( \text{tn_tree} \), using the function \( \text{build_tree} \), and fed to the above algorithm.
Variables \((W : \mathcal{W}(\text{F}_2, B))\) \((y : \text{rV}[B]_n)\).
Let \(rW n0 := (W 0 (y / n0), W 1 (y / n0))\).
Let \(\text{computed_tree} := \text{sumprod_all (build_tree H rW (k:=kv) ord0)}\).

**Extraction of estimations** We finally recover normalized estimations from the tree.

**Definition** normalize \((p : \mathbb{R}^2) :=\)
\[
\text{let } (p0, p1) := p \text{ in } (p0 / (p0 + p1), p1 / (p0 + p1)).
\]

**Fixpoint** estimation \(\{k\} (n : \text{tn_tree k R2 R2}) :=\)
\[
\text{let } l := \text{flatten (map estimation (children n)) in }
\text{match node_tag n with}
\begin{cases} 
\text{Func} & \Rightarrow l \\
\text{Var v} & \Rightarrow (\text{node_id n, normalize } \beta_{op} \text{(up n) (down n)}) : : l 
\end{cases}
\]

**Correctness** The correctness of the algorithm above consists in showing that the estimations computed are the intended a posteriori probabilities:

Let \(\text{estimations} := \text{estimation computed_tree}\).

**Definition** esti_spec \(n0 (x : \text{rV}_n) :=\)
\[
(U \rightarrow \text{not_empty}) \rightarrow n0 \rightarrow W, \text{Hy} (x / n0 | y).
\]

**Definition** estimation_spec := uniq [seq p.1 | p ← estimations] \(\land\)
\[
\forall n0, (\text{inr n0, p01 (esti_spec n0) n0}) \in \text{estimations}.
\]

where \(p01 f n0\) applies \(f\) to a vector whose \(n0_{th}\) bit is set to 0 and 1.

**Theorem** estimation_ok in [3, ldpc_algo_proof.v] provides a proof of this fact. As key steps, it uses the lemmas messages_computation and estimation_correctness from Sect. 5.3.

**Concrete codes** All proofs of probabilistic sum-product decoding assume the Tanner graph to be acyclic [10]. In practice codes based on acyclic graphs are rare and not very efficient [7]. We tested our implementation with one of them [3, sumprod_test.ml].

In the general case where the Tanner graph contains cycles, an alternative algorithm computes the messages repeatedly, propagating them in the graph until the corrected word satisfies the parity checks, failing if the result is not reached in a fixed number of iterations [10, Sect. 5]. While this works well in practice, there is no proof of correctness for the cyclic case. Based on our implementation, one could also build a tree approximating the graph, by unfolding it to a finite depth.

7 Related Work

Coding theory has been considered as an application of the interface between the Isabelle proof-assistant and the Sumit computer algebra system [4]. In order to take advantage of the computer algebra system, proofs are restricted to a certain code length. Though the mathematical background about polynomials has been formally verified, results about coding theory are only asserted. In comparison, we formally verify much more (generic) lemmas. Yet, for example when proving that certain bitstrings are codewords, we found ourselves performing formal proofs close to symbolic computation. With this respect, we may be able in a near future to take advantage of extensions of the MATHCOMP library that provide computation [6].
8 Conclusion

In this paper, we have proved the main properties of Hamming codes and sum-product decoding. It is interesting to contrast the two approaches, respectively known as classical and modern coding theory.

For Hamming codes, we could provide an implementation of minimal-distance decoding, and prove that it indeed realizes maximum likelihood decoding, i.e., the best possible form of decoding.

For sum-product decoding, which provides the basis for LDPC codes, one can only prove that the sum-product algorithm allows to implement MPM decoding. However, this is two steps away from maximum likelihood: the proof is only valid for acyclic Tanner graphs, while interesting codes contain cycles, and MPM is an approximation of MAP decoding, with only the latter providing maximum likelihood. Yet, this “extrapolation” methodology does work: sum-product decoding of LDPC codes is empirically close to maximum likelihood, and performs very well in practice.

Acknowledgments Taku Asai, Kazuhiko Sakaguchi, Yuto Takahashi, and Takanfumi Saikawa contributed to the formalization. The formalization of modern coding theory is a collaboration with Manabu Hagiwara, Kenta Kasai, Shigeaki Kuzuoka, and Ryosuke Obi.

References

A Overview of the Formalization

Table 1 indicates where to find in our formalization [3] the files relevant to the definitions and lemmas discussed in the paper. As explained in Sect. 2.1, this formalization relies on previous work on information theory [2] regarding the formalization of probabilities, channel codes, channels, etc. We have not listed up files that provide MATH-COMP with minor extensions: big operator for maximum over reals (Rbigop_max.v), additional definitions about row vectors (ssralg_ext.v), etc. There are also several files with minor results (cyclic_code.v, mceliece.v, etc.) or in a work-of-progress state that we have not referred to in this paper.

<table>
<thead>
<tr>
<th>file name</th>
<th>contents</th>
<th>l.o.c.</th>
</tr>
</thead>
<tbody>
<tr>
<td>f2.v</td>
<td>Lemmas about $F_2$</td>
<td>215</td>
</tr>
<tr>
<td>pproba.v</td>
<td>A posteriory probability (Sect. 2.2)</td>
<td>151</td>
</tr>
<tr>
<td>natbin.v</td>
<td>Naturals as bit-vectors (such as nat2bin_cV of Sect. 4)</td>
<td>618</td>
</tr>
<tr>
<td>linear_code.v</td>
<td>Formalization of linear codes (Sect. 3)</td>
<td>648</td>
</tr>
<tr>
<td>repcode.v</td>
<td>Example: repetition codes (see Sections 3.2 and 3.3)</td>
<td>418</td>
</tr>
<tr>
<td>decoding.v</td>
<td>Specifications of decoders (Sect. 3.3)</td>
<td>309</td>
</tr>
<tr>
<td>hamming.v</td>
<td>Hamming weight and Hamming distance and lemmas (Sect. 4)</td>
<td>889</td>
</tr>
<tr>
<td>hamming_code.v</td>
<td>Theory of Hamming codes (Sect. 4)</td>
<td>1227</td>
</tr>
<tr>
<td>subgraph_partition.v</td>
<td>Subgraphs and generic partition properties (Sect. 5.1)</td>
<td>1590</td>
</tr>
<tr>
<td>tanner.v</td>
<td>Tanner graphs: definitions and notations (Sect. 5.1)</td>
<td>217</td>
</tr>
<tr>
<td>tanner_partition.v</td>
<td>Partition properties of Tanner graphs (Sect. 5.1)</td>
<td>1257</td>
</tr>
<tr>
<td>summary.v</td>
<td>Summary notations (Sect. 5.2)</td>
<td>458</td>
</tr>
<tr>
<td>summary_tanner.v</td>
<td>Properties of the summary notation (see Sect. 5.2 for example)</td>
<td>797</td>
</tr>
<tr>
<td>checksum.v</td>
<td>Properties of the $\delta$ function (see Sect. 5.3)</td>
<td>219</td>
</tr>
<tr>
<td>ldpc.v</td>
<td>Properties of sum-product decoding (Sect. 5.3)</td>
<td>1127</td>
</tr>
<tr>
<td>ldpc_algo.v</td>
<td>Implementation of the sum-product algorithm (Sect. 6)</td>
<td>321</td>
</tr>
<tr>
<td>ldpc_algo_proof.v</td>
<td>Verification of the sum-product implementation (Sect. 6)</td>
<td>2456</td>
</tr>
<tr>
<td>sumprod_test.ml</td>
<td>OCaml code for testing, partly extracted (Sect. 6)</td>
<td>143</td>
</tr>
</tbody>
</table>

Total (Coq only): 12917

Table 1. Overview of the formalization [3]