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

Reynald Affeldt[1] and Jacques Garrigue[2]

[1] National Institute of Advanced Industrial Science and Technology
[2] Nagoya University

Abstract. By adding redundancy to transmitted data, error-correcting codes (ECCs) make it possible to communicate reliably over noisy channels. Minimizing redundancy and (de)coding time has driven much research, culminating with Low-Density Parity-Check (LDPC) codes. At first sight, ECCs may be considered as a trustful 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. In order 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 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 R. G. Gallager; they were not thoroughly studied until they were shown 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, 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 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. In comparison, this is more difficult than verification of cryptographic functions whose specification requires little infrastructure when they rely on number theory (e.g., [1]).

[*] This work was presented at ITP 2015 (http://www.inf.kcl.ac.uk/staff/urbanc/itp-2015)

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: prose definitions that look incomplete without the accompanying examples, algorithms written in prose, hypotheses about the model that appear during the course of proofs, etc. 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 standard 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 first recall basic definitions from [2].

The most generic definition of a code is as a *channel code*: a pair of encoder/decoder functions with a finite type `M` for the message pieces to be encoded. Encoded message pieces (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[3] alphabet `B`):

```
Definition encT := {ffun M → 'rV[A]_n}.
Definition decT := {ffun 'rV[B]_n → option M}.
Record code := mkCode { 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`:

```
Notation "𝒞ℋ₁(A, B)" := (A → dist B).
```

[3] `A` and `B` are different for example in the case of the *binary erasure channel* that replaces some bits with an *erasure*.

`dist` is the type of probability distributions. They are essentially functions from some finite type to non-negative reals whose outputs sum to 1 (the big operator `\sum_(x in P) f x` comes from MATHCOMP):

```
Record dist (A : finType) := mkDist {
  pmf :> A → R+ ; (* "→ R+" is a notation *)
  pmf1 : \sum_(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}) := \sum_(a in E) P a.
```

Communication of n characters is thought of as happening over the n^{th} extension of the channel, defined as a function from input vectors to distributions of output vectors (`{dist T}` is a notation for the type of distributions; it hides a function that checks whether `T` is a finite type):

```
Notation "𝒞ℋn(A, B)" := ('rV[A]_n → {dist 'rV[B]_n}).
```

In this paper, we deal with *discrete memoryless channels* (DMCs). It means that the output probability of a character does not depend on preceding inputs. In this case, the definition of the n^{th} extension of a channel `W` boils down to a probability mass function that associates to an input vector `x` the following distribution of output vectors:

```
Definition f (y : 'rV_n) := \prod_(i < n) W (x /_ i) (y /_ i).
```

where `x /_ i` represents the `i`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 ErrRateCond (W :𝒞ℋ1(A, B)) c m :=
  Pr (W ^ n (| enc c m)) [set y | dec c y ≠ Some m].
Definition CodeErrRate (W :𝒞ℋ1(A, B)) c :=
  1 / INR #| M | * \sum_(m in M) ErrRateCond W c m.
```

`W ^ n (| enc c m)` is the distribution of outputs corresponding to the codeword `enc c m` of a message `m` sent other the DMC of `W`. `[set y | dec c y ≠Some m]` is the 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:

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

We formalize aposteriori probabilities with the following probability mass function:

```
Definition den := ∑_(x in 'rV_n) P x * W ^ n (y | x).
Definition f x := P x * W ^ n (y | 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 | x)` $\neq 0$) was sent beforehand. The denominator being non-zero is thus equivalent to the `receivable` condition:

```
Definition receivable y := [∃ x, (P x ≠ 0) ∧ (W ^ n (y | x) ≠ 0)].
```

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

Finally, the probability that the n_0^{th} bit of the input 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):

$$P^W_{n_0}(b|y) := K \sum_{x \in \mathbb{F}_2^n \; x_{n_0} = b} P^W(x|y)$$

In Coq, we will denote this probability by `P '_ n0 `^^ W , H (b | 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 `'F_2` 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['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['F_2]_(m, n)) (y : 'rV_n) := H *m y^T.
Definition kernel H := [set c | 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).
```

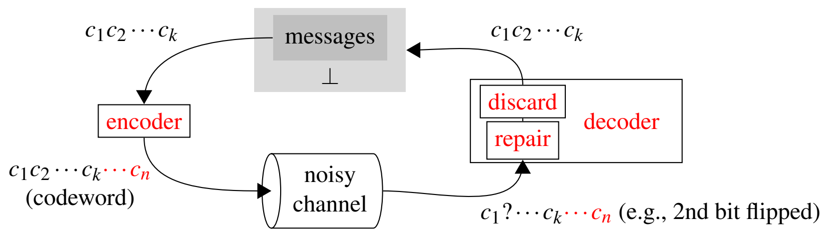

Fig. 1. The setting of error-correcting codes

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 `C` is not trivial (proof `C_not_trivial` 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 $d_{\min}$ defines in particular the number of errors $\lfloor\frac{d_{\min}-1}{2}\rfloor$ that one can correct using minimum distance decoding (see Sect. 3.3):

```
Definition mdd_err_cor := (d_min-1)/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 (see 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) its image 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`). Note that `k` $\leq$ `n` can be derived from the injectivity of the encoder.

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 ; (* 'rV['F_2]_n → option ('rV['F_2]_n) *)
  discard : discardT n k ; (* 'rV['F_2]_n → 'rV['F_2]_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 two codewords: $00\cdots0$ and $11\cdots1$ (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* (details in [3]).

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, p. 10] that shows that a minimum distance decoder can correct `mdd_err_cor` (see Sect. 3.1) errors:

```
Lemma encode_decode m y : (dec C) y ≠ 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, a ML decoder ϕ is such that $W^n(y|f(\phi(y))) = \max_{m\in M} W^n(y|f(m))$:

```
Definition maximum_likelihood_decoding :=
  support (enc c) → ∀ y, receivable W P y →
  ∃ m, (dec c) y = Some m ∧
    W ^ n (y | (enc c) m) = \rmax_(m' in M) W ^ n (y | (enc c) m').
```

The assumption `receivable W P y` says that we consider outputs with non-zero probability (see Sect. 2.2). The assumption `support (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:

```
Lemma MD_implies_ML : p < 1/2 → minimum_distance_decoding c →
  (∀ y, (dec c) y ≠ None) → maximum_likelihood_decoding W c P.
```

Maximum aposteriori probability (MAP) decoding decodes to messages that maximize the aposteriori 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 aposteriori 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:

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

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

```
Definition hamH := \matrix_(i < m, j < n) (nat2bin_cV m j+1 i 0).
Definition hamC : lcode0 n := 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 ($7 \times 2^{n-3} = (1110\cdots0)_2$). 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_scode` 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_scode.
```

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 the 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:

$$sysH_{7,4} = \left(\begin{array}{cccc||ccc} 0&1&1&1&1&0&0 \\ 1&0&1&1&0&1&0 \\ 1&1&0&1&0&0&1 \end{array}\right) \qquad sysG_{7,4} = \left(\begin{array}{cccc||ccc} 1&0&0&0&0&1&1 \\ 0&1&0&0&1&0&1 \\ 0&0&1&0&1&1&0 \\ 0&0&0&1&1&1&1 \end{array}\right)$$

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 (erefl, 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 (erefl, 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:

```
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:

```
Lemma hamming_error_rate : p < 1/2 → ē_cha(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.

5 Formalization of the Properties of Sum-product Decoding

The sum-product algorithm provides efficient decoding for LDPC codes. It computes for each bit its marginal aposteriori probability by propagating probabilities in a graph corresponding to the PCM. We explain those graphs in Sect. 5.1, the summary operator 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 *variable 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`:

```
Variables (V : finType) (g : rel V).
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 function nodes of the subgraph rooted at edge `m0`–`n0` by `‘F(m0, n0)`. Similarly, we denote the variable 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:

```
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).

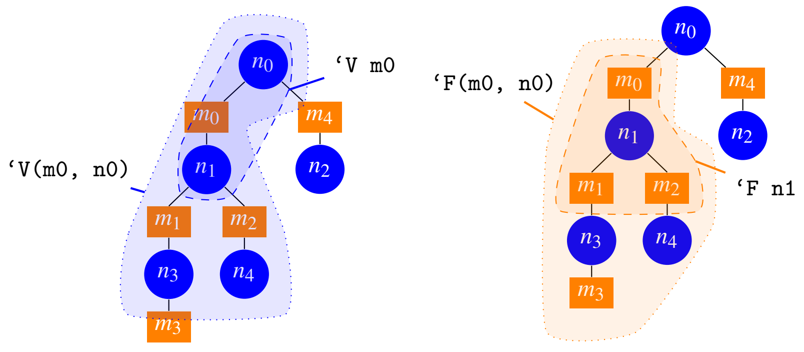

Fig. 2. Successors and subtrees in an acyclic Tanner graph

5.2 The Summary Operator

Pencil-and-paper proofs in modern coding theory [12] make use of a special summation called the *summary operator* [10]. It is denoted by $\sum_{\sim s}$ and indicates the variables *not* being summed over. This operator saves the practitioner "from a flood of notation" [12, p. 49], for example by writing steps such as:

$$\prod_{m_0 \in F(n_0)} \sum_{\sim\{n_0\}} \cdots = \sum_{\sim\{n_0\}} \prod_{m_0 \in F(n_0)} \cdots, \qquad (1)$$

the reader being trusted to understand that both operators sum over different sets.

We formalize the summary operator as a sum over vectors `x` such that `x /_ i` is fixed using a default vector `d` when `i ∉ s` and write `∑_(x # s, d)` instead of $\sum_{\sim s}$:

```
Definition summary (s : {set 'I_n}) (d x : 'rV[A]_n) :=
  [∀ i, (i ∈ ~: s) ⇒ (x /_ i = d /_ i)].
Notation "∑_ ( x '#' s ',' d ) e" := (∑_( x | summary s d x ) e)
```

Indeed, $\sum_{\sim s}$ can be understood as a sum over vectors $[x_0; x_1; \ldots; x_{n-1}]$ such that x_i is fixed when $i \in s$. We found it difficult to recover the terseness of the pencil-and-paper summary operator in a proof-assistant. First, the precise set of varying x_j is implicit; it can be inferred by looking at the x_j appearing below the summation sign but this is difficult to achieve unless one reflects most syntax. Second, it suggests working 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). Last, it is not clear about the values of x_i when $i \in s$.

In contrast, our formalization makes clear, for example, that in equation (1) the first summary operator sums over $V(m_0, n_0) \backslash \{n_0\}$ while the second one sums over $[1, \ldots, n] \backslash \{n_0\}$ (see Sect. 5.3 for the formalization). More importantly, we can benefit from MathComp lemmas about big operators to prove the properties of the sum-product decoding thanks to our encoding (see Sect. 5.3).

Alternatively, our summary operator `\sum_(x # s , d) e x` can also be thought as $\sum_{x_1 \in \mathbb{F}_2} \cdots \sum_{x_{|s|} \in \mathbb{F}_2} e\ d[s_1 := x_1] \cdots [s_{|s|} := x_{|s|}]$ where $d[i := b]$ represents the vector d where index i is updated with b. Put formally (`enum s` below is the list $[s_1; s_2; \cdots; s_{|s|}]$):

```
Definition summary_fold (s : {set 'I_n}) d e :=
foldr (fun n0 F t ⇒ ∑_(b in 'F_2) F (t `[n0 := b])) e (enum s) d.
```

This is equivalent (`∑_(x # s, d) e x = summary_fold s d e`) but we found it easier to use `summary_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 and a channel output y. With sum-product decoding, we are concerned with evaluating $P^W_{n_0}(b|y)$ where b is the value of the n_0^{th} bit of the input codeword (see Sect. 2.2). In the following, we show that it is proportional to the following quantity:

$$P^W_{n_0}(b|y) \propto W(y_{n_0}|b) \prod_{m_0 \in F(n_0)} \alpha_{m_0,n_0}(b).$$

$\alpha_{m_0,n_0}(b)$ (formal definition below) is the contribution to the marginal aposteriori probability of the n_0^{th} bit coming from a subtree of the Tanner graph (we assume that the Tanner graph is acyclic).

We now provide the formal statement. Let `W` be a channel. Let `H` be a $m \times n$ PCM such that the corresponding Tanner graph is acyclic (hypothesis `acyclic_graph (tanner_rel H)`, where `tanner_rel` turns a PCM into the corresponding Tanner graph). Let `y` be the channel output to decode. We assume that it is receivable (hypothesis `Hy` below, see Sect. 2.2). Finally, let `d` be the vector used in the summary operator. Then the aposteriori probability $P^W_{n_0}(b|y)$ can be evaluated by a closed formula:

```
Lemma estimation_correctness (d : 'rV_n) n0 :
  let b := d /_ n0 in let P := `U C_not_empty in
  P '_ n0 `^^ W , Hy (b | y) =
  Kmpp Hy * Kpp W H y * W b (y /_ n0) * ∏_(m0 in `F n0) α m0 n0 d.
```

`Kmpp` and `Kpp` are normalization constants (see [3]). `P` is a uniform distribution. The distribution `` `U C_not_empty `` of codewords has the following probability mass function: $cw \mapsto 1/|C|$ if $cw \in C$ and 0 otherwise. α is the marginal aposteriori probability of the n_0^{th} bit of the input codeword in the modified Tanner graph that includes only function nodes from the subgraph rooted at edge m_0–n_0 and in which the received bit `y /_ n0` has been erased. The formal definition relies on the summary operator:

```
Definition α m0 n0 d := ∑_(x # `V(m0, n0) :\ n0 , d)
  W ^ _ (y # `V(m0, n0) :\ n0 | x # `V(m0, n0) :\ n0) *
    ∏_(m1 in `F(m0, n0)) INR (δ (`V m1) x).
```

`δ s x` is an indicator function that performs checksum checks:

```
Definition δ n (s : {set 'I_n}) (x : 'rV['F_2]_n) :=
  (\big[+%R/Zp0]_(n0 in s) x /_ n0) = 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 `recursive_computation` for a concrete example. The second one is the main motivation for using the summary operator. We need to make big sums commute with big prods in equalities like:

```
∏_(m0 in ‘F n0) ∑_(x # ‘V(m0, n0) :\ n0 , d) ... =
  ∑_(x # setT :\ n0 , d) ∏_(m0 in ‘F n0) ...
```

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.

Recursive Computation of α's The property above provides a way to evaluate $P^W_{n_0}(b|y)$ but not an efficient algorithm because the computation of $\alpha_{m_0,n_0}(b)$ is about the whole subgraph rooted at the edge m_0–n_0. The second property that we formalize introduces β probabilities such that α's (resp. β's) can be computed from neighboring β's (resp. α's). This is illustrated by Fig. 3 whose meaning will be made clearer in Sect. 6. We define β using α as follows:

```
Definition β n0 m0 (d : 'rV_n) :=
  W (d /_ n0) (y /_ n0) * ∏_(m1 in ‘F n0 :\ m0) α m1 n0 d.
```

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

```
Lemma recursive_computation m0 n0 d : n0 ∈ ‘V m0 →
  α m0 n0 d = ∑_(x # ‘V m0 :\ n0 , d)
    INR (δ (‘V m0) x) * ∏_(n1 in ‘V m0 :\ n0) β 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 commutations of big sums and big prods using the summary operator. Let us perform the first proof step for illustration. It consists in turning the inner product of α messages `∏_(m1 in ‘F(m0, n0) :\ m0) INR (δ (‘V m1) x)` into:

```
∏_(n1 in ‘V m0 :\ n0) ∏_(m1 in ‘F n1 :\ m0)
  ∏_(m2 in ‘F(m1, n1)) INR (δ (‘V m2) x)
```

This is a consequence of the fact that `‘F(m0, n0) :\ m0` can be partitioned (when `H` is acyclic) into smaller `‘F(m1, n1)` 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 : {set {set 'I_m}} :=
  (fun n1 ⇒ ⋃_(m1 in ‘F n1 :\ m0) ‘F(m1, n1)) @: ((‘V m0) :\ n0).
```

Once `Fgraph_part_Fgraph m0 n0` has been shown to cover `‘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, each representing a bit of the decoded code-

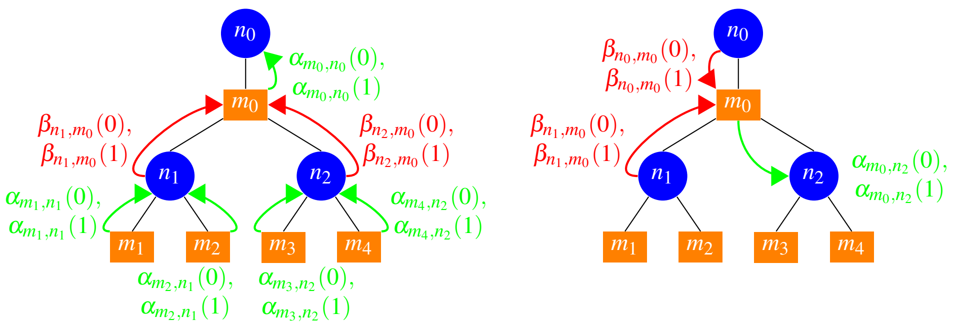
n_0
m_0
n_1
n_2
m_1
m_2
m_3
m_4
$\alpha_{m_0,n_0}(0)$, $\alpha_{m_0,n_0}(1)$
$\beta_{n_1,m_0}(0)$, $\beta_{n_1,m_0}(1)$
$\beta_{n_2,m_0}(0)$, $\beta_{n_2,m_0}(1)$
$\alpha_{m_1,n_1}(0)$, $\alpha_{m_1,n_1}(1)$
$\alpha_{m_4,n_2}(0)$, $\alpha_{m_4,n_2}(1)$
$\alpha_{m_2,n_1}(0)$, $\alpha_{m_2,n_1}(1)$
$\alpha_{m_3,n_2}(0)$, $\alpha_{m_3,n_2}(1)$
$\beta_{n_0,m_0}(0)$, $\beta_{n_0,m_0}(1)$
$\beta_{n_1,m_0}(0)$, $\beta_{n_1,m_0}(1)$
$\alpha_{m_0,n_2}(0)$, $\alpha_{m_0,n_2}(1)$

Fig. 3. Illustrations for `sumprod_up` and `sumprod_down`. Left: `sumprod_up` computes the `up` links from the leaves to the root. Right: `sumprod_down` computes the `down` link of edge m_0–n_2 using the β's of edges m_0–n_i ($i \neq 2$).

word, its marginal aposteriori probability. One chooses to decode the n_0^{th} bit either as 0 if $P^W_{n_0}(0|y) \geq P^W_{n_0}(1|y)$ or as 1 otherwise, so as to perform MPM decoding.

The algorithm we implement is known in the literature as the forward/backward algorithm and has many applications [10]. It uses the tree view of an acyclic Tanner graph to structure recursive computations. In a first phase it computes α's and β's (see Sect. 5.3) from the leaves towards the root of the tree, and then computes α's and β's in the opposite direction (starting from the root that time). Fig. 3 illustrates this.

Concretely, we provide Coq functions to build the tree, compute α's and β's, 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 variable 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 probabilities for this bit to be 0 or 1, i.e., the pair $(W(y_{n_0}|0), W(y_{n_0}|1))$. The `up` and `down` fields are to be filled with the values of α and β (according to the kind), going to the parent node for `up`, and coming from it for `down`. Here again we will use pairs of the 0 and 1 cases. Note that the values of α's and β's need not be normalized.

Computation of α and β The function α_β takes as input the tag of the source node, and the α's and β's from neighboring nodes, excluding the destination, and computes either α or β, according to the tag. Thanks to this function, the remainder of the algorithm keeps a perfect symmetry between variable and function nodes.

```
Definition α_op (out inp : R2) :=
  let (o,o') := out in let (i,i') := inp in
  (o*i + o'*i', o*i' + o'*i).
Definition β_op (out inp : R2) :=
  let (o,o') := out in let (i,i') := inp in (o*i, o'*i').
Definition α_β k (t : tag k) : seq R2 → R2 :=
  match t with
  | Func ⇒ foldr α_op (1,0)
  | Var v ⇒ foldl β_op v
  end.
```

The definition for β 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 α's. For α, starting from the `recursive_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 `α_op`. This optimization is described in [10, Sect. 5-E]. We will of course need to prove that these definitions compute the same α's and β's as in Sect. 5.3.

Propagation of α and β `sumprod_up` and `sumprod_down` compute respectively the contents of the `up` and `down` fields.

```
Fixpoint sumprod_up {k} (n : tn_tree k unit unit)
  : tn_tree k R2 unit :=
  let children' := map sumprod_up (children n) in
  let up' := α_β (node_tag n) (map up children') in
  Node (node_id n) (node_tag n) children' up' tt.
Fixpoint seqs_but1 (a b : seq R2) :=
  if b is h::t then (a++t)::seqs_but1 (rcons a h) t else [::].
Fixpoint sumprod_down {k} (n : tn_tree k R2 unit)
  (from_above : option R2) : tn_tree k R2 R2 :=
  let (arg0, down') :=
    if from_above is Some p then ([::p],p) else ([::],(1,1)) in
  let args := seqs_but1 arg0 (map up (children n)) in
  let funs := map
      (fun n' l ⇒ sumprod_down n' (Some (α_β (node_tag n) l)))
      (children n) in
  let children' := apply_seq funs args in
  Node (node_id n) (node_tag n) children' (up n) down'.
Definition sumprod {k} n := sumprod_down (@sumprod_up k n) None.
```

The `from_above` argument is `None` for the root of the tree, or the β coming from the parent node otherwise. `apply_seq` applies a list of functions to a list of arguments. This is a workaround to allow defining `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 `tn_tree`, using the function `build_tree`, and fed to the above `sumprod` algorithm:

```
Variables (W : `CH1('F_2, B)) (y : 'rV[B]_n).
Let rW n0 := (W 0 (y /_ n0), W 1 (y /_ n0)).
Let computed_tree := sumprod (build_tree H rW (k := kv) ord0).
```

Extraction of estimations We finally recover normalized estimations from the tree:

```
Definition normalize (p : R2) :=
  let (p0, p1) := p in (p0 / (p0 + p1), p1 / (p0 + p1)).
Fixpoint estimation {k} (n : tn_tree k R2 R2) :=
  let l := flatten (map estimation (children n)) in
  if node_tag n is Var _ then
    (node_id n, normalize (β_op (up n) (down n))) :: l
  else l (* node_tag n is Func *).
```

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

```
Let estimations := estimation computed_tree.
Definition esti_spec n0 (x : 'rV_n) :=
  (`U C_not_empty) '_ n0 `^^ W, Hy (x /_ n0 | y).
Definition estimation_spec := uniq (unzip1 estimations) ∧
  ∀ n0, (inr n0, p01 (esti_spec n0) n0) ∈ estimations.
```

where `p01 f n0` applies `f`, to a vector whose n_0^{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 `recursive_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, one would use an alternative algorithm that computes the α's and β's repeatedly, propagating them in the graph until the corrected word satisfies the parity checks, failing if the result is not reached within a fixed number of iterations [10, Sect. 5]. This works well in practice but there is no proof of correctness, even informal. In place of this iterative approach, one could also build a tree approximating the graph, by unfolding it to a finite depth, and apply our functional algorithm.

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 T. Asai, T. Saikawa, K. Sakaguchi, and Y. Takahashi contributed to the formalization. The formalization of modern coding theory is a collaboration with M. Hagiwara, K. Kasai, S. Kuzuoka, and R. Obi. The authors are grateful to the anonymous reviewers for their comments. This work is partially supported by a JSPS Grant-in-Aid for Scientific Research (Project Number: 25289118).

References

1. Affeldt, R., Nowak, D., Yamada, K.: Certifying Assembly with Formal Security Proofs: the Case of BBS. Sci. Comput. Program. 77(10-11):1058-1074 (2012).
2. Affeldt, R., Hagiwara, M., Sénizergues, J.: Formalization of Shannon's theorems. J. Autom. Reasoning 53(1):63–103 (2014).
3. Affeldt, R., Garrigue, J.: Formalization of Error-correcting Codes: from Hamming to Modern Coding Theory. Coq scripts. Available at https://staff.aist.go.jp/reynald.affeldt/ecc.
4. Ballarin, C., Paulson, L.C.: A Pragmatic Approach to Extending Provers by Computer Algebra—with Applications to Coding Theory. Fundam. Inform. 34(1–2):1–20 (1999).
5. Betten, A., Braun, M., Fripertinger, H., Kerber, A., Kohnert, A., Wassermann, A.: Error-Correcting Linear Codes—Classification by Isometry and Applications. Springer (2006).
6. Dénès, M., Mörtberg, A., Siles, V.: A Refinement-Based Approach to Computational Algebra in Coq. In: Proc. of the 3rd International Conference on Interactive Theorem Proving (ITP 2012). LNCS, vol. 7406, pp. 83–98. Springer (2012).
7. Etzion, T., Trachtenberg, A., Vardy, A.: Which codes have cycle-free Tanner graphs? IEEE Trans. Inf. Theory 45(6):2173–2181 (1999).
8. Gonthier, G., Mahboubi, A., Tassi, E.: A small scale reflection extension for the Coq system. Technical Report RR-6455. INRIA (2008). Version 14 (March 2014).
9. Hagiwara, M.: Coding Theory: Mathematics for Digital Communication. Nippon Hyoron Sha (2012). In Japanese.
10. Kschischang, F.R., Frey, B.J., Loeliger, H.-A.: Factor graphs and the sum-product algorithm. IEEE Trans. Inf. Theory 47(2):498–519 (2001).
11. MacWilliams, F.J., Sloane, N.J.A.: The Theory of Error-Correcting Codes. North-Holland (1977). 7th impression (1992).
12. Richardson, T., Urbanke, R.: Modern Coding Theory. Cambridge University Press (2008).

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.

file name	contents	l.o.c.
`f2.v`	Lemmas about $\mathbb{F}_2$	215
`pproba.v`	Aposteriori probability (Sect. 2.2)	151
`natbin.v`	Naturals as bit-vectors (such as `nat2bin_cV` of Sect. 4)	618
`linear_code.v`	Formalization of linear codes (Sect. 3)	648
`repcode.v`	Example: repetition codes (see Sections 3.2 and 3.3)	418
`decoding.v`	Specifications of decoders (Sect. 3.3)	309
`hamming.v`	Hamming weight and Hamming distance and lemmas (Sect. 4)	889
`hamming_code.v`	Theory of Hamming codes (Sect. 4)	1227
`subgraph_partition.v`	Subgraphs and generic partition properties (Sect. 5.1)	1590
`tanner.v`	Tanner graphs: definitions and notations (Sect. 5.1)	217
`tanner_partition.v`	Partition properties of Tanner graphs (Sect. 5.1)	1257
`summary.v`	Summary operator (Sect. 5.2)	458
`summary_tanner.v`	Properties of the summary operator (see Sect. 5.2 for example)	797
`checksum.v`	Properties of the δ function (see Sect. 5.3)	219
`ldpc.v`	Properties of sum-product decoding (Sect. 5.3)	1127
`ldpc_algo.v`	Implementation of the sum-product algorithm (Sect. 6)	321
`ldpc_algo_proof.v`	Verification of the sum-product implementation (Sect. 6)	2456
`sumprod_test.ml`	OCaml code for testing, partly extracted (Sect. 6)	143
	Total (Coq only):	12917

Table 1. Overview of the formalization [3]