Università Politecnica delle Marche
Facoltà di Ingegneria
Corso di Laurea Triennale in Ingegneria Elettronica

Analisi e implementazione di algoritmi iterativi a bassa latenza e complessità per la decodifica di codici LDPC

Analysis and implementation of low latency and complexity iterative algorithms for LDPC codes decoding

Relatore:
Dr. Massimo Battaglioni

Tesi di Laurea di:
Mauro Michele Maria Costantino

Correlatore: Chiar.mo
Prof. Franco Chiaraluce

Anno Accademico 2023/2024

A zia Eliana,
spero di aver continuato
a renderti orgogliosa

Ringraziamenti

Prima di cominciare, ritengo doveroso dedicare una pagina a chi, direttamente o indirettamente, ha contribuito al mio percorso e all'elaborazione di questo documento.

Il primo ringraziamento sentito va al mio Relatore, il Dott. Massimo Battaglioni, per aver avuto la pazienza di spiegarmi ciò che serviva e per aver sopportato tutti i miei messaggi, anche quelli di prima mattina, e al Prof. Franco Chiaraluce per avermi fatto appassionare con le sue lezioni.

Non basterebbero tutte le pagine di questo documento a ringraziare mia mamma per essere ed essere sempre stata, se possibile, più di ciò che dovrebbe essere una madre, per avermi reso la persona che sono e per non essersi mai arresa, anche quando la resa sembrava l'unica soluzione.

Allo stesso modo, è impossibile esprimere in poche righe la gratitudine che provo verso zia Eliana, che è stata la mia seconda madre e che, dopo avermi fatto scoprire l'Ingegneria Elettronica, spero continui a guardarmi realizzare i miei sogni e raggiungere i traguardi che, in fondo, sono anche suoi.

Un ringraziamento sentito va anche a Serena per essermi stata vicina nell'ultima metà di questo percorso, per aver ascoltato le mie paure, spesso infondate, e avermi sempre spronato.

Infine, ritengo fondamentale ringraziare i miei amici, quelli veri, quelli di una vita e quelli conosciuti all'Università. Questo percorso non sarebbe stato lo stesso senza di voi.

Mauro Michele Maria Costantino

Indice

Introduzione

I codici Low-Density Parity-Check (LDPC) rappresentano ormai uno standard nel mondo delle telecomunicazioni grazie alle loro ottime capacità di correzione degli errori, a fronte di una complessità di decodifica limitata. Attualmente tali codici vengono usati per l'Ethernet 10GBASE-T, sono parte integrante dello standard Wi-Fi IEEE 802.11 e ricoprono un ruolo fondamentale nel nuovo standard Wi-Fi 6 IEEE 802.11ax. Inoltre, come riportato in [1] e [2], questa famiglia di codici viene usata anche nell'ambito delle comunicazioni satellitari e per codificare le informazioni dei canali dati previsti nello standard 5G ([3]).

Contrariamente a quanto si possa pensare da queste premesse, tuttavia, i codici LDPC non costituiscono esattamente una scoperta recente. La loro paternità, infatti, è da attribuire a Robert Gallager, che li scoprì durante il suo dottorato di ricerca nel 1960 e che nel 1962 pubblicò in [4]. A causa della loro presunta complessità nel processo di decodifica, però, i codici in questione non ricevettero particolari attenzioni fino al 1995, anno di pubblicazione di [5], documento nel quale vennero mostrate le loro ottime prestazioni. Inoltre, nel 1999 venne presentato per la prima volta in [6] il concetto di codice Spatially-Coupled LDPC (SC-LDPC), o codice LDPC convoluzionale. Questa classe di codici convoluzionali fornisce diversi vantaggi rispetto alla controparte classica ma, di contro, rende ancora più oneroso il processo di decodifica dal punto di vista computazionale.

L'oggetto di questa tesi, pertanto, è proprio l'analisi dei vari algoritmi a complessità ridotta per la decodifica dei codici sopra citati e lo studio dei decodificatori che li implementano.

Nel presente lavoro verranno innanzitutto introdotti, nel primo Capitolo, i concetti teorici alla base dei codici per la correzione d'errore, a blocco e convoluzionali, e i codici LDPC nello specifico. Nel secondo Capitolo, poi, si descriveranno dal punto di vista teorico gli algoritmi di decodifica iterativi usati nell'ambito dei codici LDPC e, ponendo particolare attenzione al Sum-Product Algorithm (SPA), verrà presentata la struttura standard del decodificatore a finestra scorrevole per codici SC-LDPC. Nel terzo Capitolo, il più corposo, verranno dapprima presentati i codici usati per l'elaborazione dei risultati, seguiti poi da un'analisi relativa al guadagno in termini di complessità computazionale dovuto all'implementazione del decodificatore di cui sopra. Successivamente, verranno presentate le modifiche proposte a tale decodificatore, introducendo il concetto di reset dei messaggi e riportandone gli effetti sulle prestazioni. Infine, verrà trattato il principale problema del decodificatore a finestra scorrevole, ovvero la propagazione degli errori. Tale fenomeno verrà prima descritto

dal punto di vista matematico e poi mostrato in maniera quantitativa sui codici presentati all'inizio del capitolo.

Tutti i dati e i risultati mostrati nel corso dell'elaborato sono stati ottenuti sfruttando le funzioni MATLAB riportate nell'Appendice A.

Capitolo 1.

Codici per la correzione d'errore

In questo capitolo vengono inizialmente illustrate le caratteristiche principali di codici a blocco e codici convoluzionali e vengono introdotte nozioni che poi saranno specializzate al caso di codici LDPC.

1.1. Codici a blocco

Consideriamo, innanzitutto, una sorgente di informazione che generi una sequenza continua di simboli binari (ovvero formata unicamente da simboli $u \in \{0, 1\}$). Nella codifica *a blocchi*, la sequenza informativa è, come suggerisce il nome, divisa in blocchi di lunghezza fissata n, ognuno dei quali trasporta k simboli informativi, comunemente detti bit.

Pertanto, fissato k, esisteranno 2^k sequenze distinte $\mathbf{u} = (u_0, u_1, \ldots, u_{k-1})$ ognuna delle quali verrà poi codificata in una sequenza $\mathbf{c} = (c_0, c_1, \ldots, c_{n-1})$, detta *parola di codice*, secondo determinate regole stabilite dal codificatore di canale. L'insieme di tali 2^k parole di codice è proprio ciò che forma un *codice a blocco* (n, k). Va sottolineato che gli $n - k$ simboli aggiunti ad ogni messaggio di input dal codificatore di canale sono simboli di *ridondanza* e, in quanto tali, non trasportano nuova informazione (motivo per cui si parla di "simboli" e non "bit"). Essi tuttavia sono necessari per fornire al codice capacità di *rivelazione* e *correzione* degli errori di trasmissione. Tali proprietà sono strettamente collegate alla *distanza minima*, ovvero al numero minimo di simboli che distingue due parole all'interno di uno stesso codice.

Un parametro importante e significativo per la descrizione di un codice a blocco, e di un codice in generale, è il *rate*, interpretabile come il numero medio di bit (intesi come unità di misura fondamentale dell'informazione) trasportati da ogni simbolo di una parola di codice. Tale parametro, nel caso dei codici a blocco, è dato quindi da

$$R = \frac{k}{n} \tag{1.1}$$

I codici a blocco possono essere divisi in due categorie, *lineari* e *non lineari*, anche se questi sono poco utilizzati in pratica. Pertanto, da [7] si riporta la seguente definizione di codici a blocco lineari:

Definizione 1. Un codice a blocco binario di lunghezza n formato da 2^k parole di codice è detto *codice a blocco* (n, k) *lineare* se e solo se le sue 2^k parole di codice formano un sottospazio vettoriale V di dimensione k di tutte le n-uple su GF(2)[1].

Poiché, quindi, un codice lineare a blocco (n, k) (indicato da ora in poi generalmente con $\mathcal{C}$), definisce un sottospazio vettoriale di dimensione k, esisteranno k parole di codice linearmente indipendenti $\mathbf{g}_0, \mathbf{g}_1, \ldots, \mathbf{g}_{k-1}$ a partire dalle quali è possibile ottenere tramite combinazione lineare tutte le possibili parole di codice $\mathbf{c} \in \mathcal{C}$ ovvero, in altri termini, esisterà una *base* di $\mathcal{C}$ formata da $\mathbf{g}_0, \mathbf{g}_1, \ldots, \mathbf{g}_{k-1}$. Indicando il messaggio da codificare con $\mathbf{u} = (u_0, u_1, \ldots, u_{k-1})$ è possibile ricavare la corrispondente parola di codice $\mathbf{c} = (c_0, c_1, \ldots, c_{n-1})$ sfruttando la base di cui sopra e prendendo i k bit di $\mathbf{u}$ come coefficienti della combinazione lineare, vale cioè la relazione:

$$\mathbf{c} = u_0\mathbf{g}_0 + u_1\mathbf{g}_1 + \cdots + u_{k-1}\mathbf{g}_{k-1} \tag{1.2}$$

Disponendo in una matrice la base di $\mathcal{C}$, si ottiene

$$\mathbf{G} = \begin{bmatrix} \mathbf{g}_0 \\ \mathbf{g}_1 \\ \vdots \\ \mathbf{g}_{k-1} \end{bmatrix} = \begin{bmatrix} g_{0,0} & g_{0,1} & \cdots & g_{0,n-1} \\ g_{1,0} & g_{1,1} & \cdots & g_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ g_{k-1,0} & g_{k-1,1} & \cdots & g_{k-1,n-1} \end{bmatrix} \tag{1.3}$$

a partire da cui è evidente che l'equazione (1.2) può essere riscritta come

$$\mathbf{c} = \mathbf{u} \cdot \mathbf{G}. \tag{1.4}$$

In virtù di tale relazione, $\mathbf{G}_{[k\times n]}$ è detta *matrice generatrice* del codice lineare a blocco (n, k) $\mathcal{C}$.

Dalla Definizione 1 si ha che $\mathcal{C}$ è un sottospazio vettoriale V delle n-uple su GF(2), ed è quindi possibile definire il suo *nullo* $\mathcal{C}_d$ come sottospazio vettoriale di V di dimensione $(n - k)$ dato da

$$\mathcal{C}_d = \{\mathbf{w} \in V : \langle \mathbf{w}, \mathbf{c} \rangle = 0 \ \forall \mathbf{c} \in \mathcal{C}\}, \tag{1.5}$$

che può essere visto a sua volta come un codice lineare a blocco $(n, n - k)$ detto *codice duale* di $\mathcal{C}$ e, in quanto tale, può essere descritto da una base di $n - k$ vettori

[1] *Campo di Galois* definito da 2 elementi all'interno del quale valgono le regole di $\mathbb{Z}_2$, ovvero l'insieme dei numeri interi modulo 2. In altri termini, $\mathbb{Z}_2 = \{0, 1\}$ e le operazioni di somma e prodotto sono definite in modulo 2

linearmente indipendenti $\mathbf{h}_0, \mathbf{h}_1, \ldots, \mathbf{h}_{n-k-1}$ che formano la matrice

$$\mathbf{H} = \begin{bmatrix} \mathbf{h}_0 \\ \mathbf{h}_1 \\ \vdots \\ \mathbf{h}_{n-k-1} \end{bmatrix} = \begin{bmatrix} h_{0,0} & h_{0,1} & \cdots & h_{0,n-1} \\ h_{1,0} & h_{1,1} & \cdots & h_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ h_{n-k-1,0} & h_{n-k-1,1} & \cdots & h_{n-k-1,n-1}. \end{bmatrix} \tag{1.6}$$

Tale matrice, oltre ad essere la generatrice del codice duale $\mathcal{C}_d$ del codice $\mathcal{C}$, verifica la relazione $\mathbf{G} \cdot \mathbf{H}^{\mathrm{T}} = \mathbf{O}$ con $\mathbf{O}$ matrice di dimensioni $k \times (n-k)$ di soli zero e $^{\mathrm{T}}$ che indica l'operazione di trasposizione. Dalle considerazioni fatte, dunque, è possibile affermare che $\mathbf{c} \in V$ è una parola di codice di $\mathcal{C}$ se e solo se $\mathbf{c} \cdot \mathbf{H}^{\mathrm{T}} = \mathbf{0}$, che implica

$$\mathcal{C} = \left\{ \mathbf{c} \in V : \mathbf{c} \cdot \mathbf{H}^{\mathrm{T}} = \mathbf{0} \right\} \tag{1.7}$$

ovvero che $\mathcal{C}$ è univocamente determinato dalla sua *matrice di parità* $\mathbf{H}_{[n-k \times n]}$. Un codice lineare a blocco dunque è univocamente determinato da due matrici, una matrice generatrice (generalmente usata per la *codifica*, anche se i codici LDPC trattati nella presente tesi rappresentano un'eccezione) e una matrice di parità (generalmente usata per la *decodifica*).

1.2. Codici convoluzionali

Come indicato in [8], è possibile generare un *codice convoluzionale* facendo transitare il messaggio informativo $\mathbf{u}$ attraverso un *registro a scorrimento* composto da K stadi (a loro volta formati da k simboli) e n sommatori, come illustrato in Figura 1.1. Per ogni gruppo di k bit in ingresso si hanno dunque n simboli in uscita e pertanto si ha un rate di $R = \frac{k}{n}$, assolutamente analogo a quello dei codici a blocco (equazione (1.1))

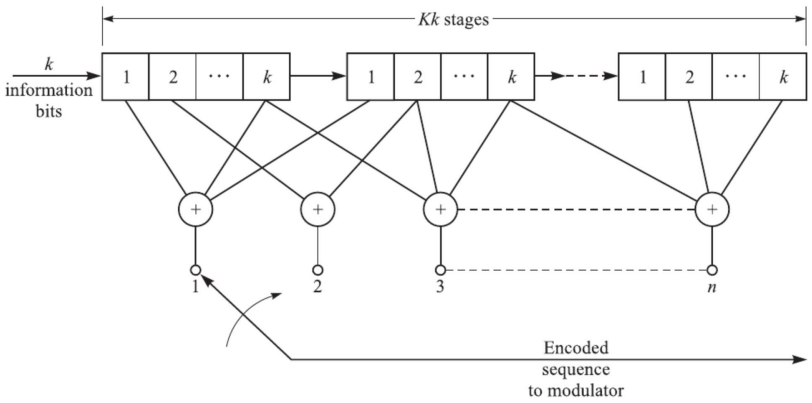


Figura 1.1.: Struttura del registro a scorrimento per codici convoluzionali [8]

Come si nota, il numero totale di stadi del registro a scorrimento è dato da $L = Kk$, col parametro K che prende il nome di *constraint length* del codice convoluzionale. Chiaramente, per costruzione stessa del registro, gli n simboli in uscita non dipendono solo dagli ultimi k bit in ingresso al codificatore, ma anche dal contenuto dei primi $(K-1)k$ stadi. Pertanto, tale registro a scorrimento è una macchina a stati finiti con $2^{(K-1)k}$ stati possibili.

I codici convoluzionali, oltre che tramite la matrice generatrice, possono essere rappresentati equivalentemente con un insieme di n vettori, uno per ciascun sommatore presente nel registro. Ognuno di questi vettori è formato da Kk simboli binari e contiene informazioni sulla connessione tra registro a scorrimento e relativo sommatore. Infatti, un 1 nella posizione i-esima del vettore n-esimo indica che esiste una connessione tra l'i-esimo stadio del registro a scorrimento e l'n-esimo sommatore, mentre uno 0 indica che tale stadio e tale sommatore non sono connessi. Ad esempio, quindi, il registro a scorrimento riportato in Figura 1.2 può essere descritto

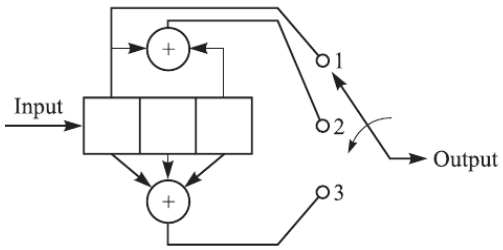


Figura 1.2.: Esempio di registro a scorrimento con $K = 3, k = 1, n = 3$ [8]

dall'insieme di vettori

$$\begin{aligned} \mathbf{g}_1 &= [100] \\ \mathbf{g}_2 &= [101] \\ \mathbf{g}_3 &= [111] \end{aligned} \tag{1.8}$$

che corrispondono alle n risposte impulsive del sistema a fronte delle k sequenze in ingresso. Quindi, indicando con $\mathbf{c}^{(i)}$ l'i-esima uscita del registro a scorrimento e con $\mathbf{u}$ la sequenza in ingresso, per il caso in esempio valgono le equazioni

$$\begin{aligned} \mathbf{c}^{(1)} &= \mathbf{u} * \mathbf{g}_1 \\ \mathbf{c}^{(2)} &= \mathbf{u} * \mathbf{g}_2 \\ \mathbf{c}^{(3)} &= \mathbf{u} * \mathbf{g}_3 \end{aligned} \tag{1.9}$$

dove $*$ rappresenta l'operazione di *convoluzione*. A partire da tali relazioni è quindi possibile ottenere la corrispondente parola di codice $\mathbf{c}$ concatenando $\mathbf{c}^{(1)}, \mathbf{c}^{(2)}$ e $\mathbf{c}^{(3)}$,

ovvero

$$\mathbf{c} = \left(c_1^{(1)}, c_2^{(1)}, c_3^{(1)}, c_1^{(2)}, c_2^{(2)}, c_3^{(2)}, c_1^{(3)}, c_2^{(3)}, c_3^{(3)}\right). \tag{1.10}$$

L'operazione di convoluzione nel tempo è equivalente alla moltiplicazione nel dominio trasformato dunque, definita la *trasformata* D[2] di $\mathbf{u}$ come

$$u(D) = \sum_{i=0}^{\infty} u_i D^i \tag{1.11}$$

e le funzioni di trasferimento delle tre risposte impulsive $\mathbf{g}_1, \mathbf{g}_2$ e $\mathbf{g}_3$ come

$$\begin{aligned} g_1(D) &= 1 \\ g_2(D) &= 1 + D^2 \\ g_3(D) &= 1 + D + D^2, \end{aligned} \tag{1.12}$$

le equazioni (1.9) possono essere riscritte nella forma

$$\begin{aligned} c^{(1)}(D) &= u(D)g_1(D) \\ c^{(2)}(D) &= u(D)g_2(D) \\ c^{(3)}(D) &= u(D)g_3(D). \end{aligned} \tag{1.13}$$

Infine, questi risultati possono essere espressi in maniera compatta definendo

$$\mathbf{G}(D) = \begin{bmatrix} g_1(D) & g_2(D) & g_3(D) \end{bmatrix} \tag{1.14}$$

a partire da cui si ottiene

$$\mathbf{c}(D) = \mathbf{u}(D) \cdot \mathbf{G}(D) \tag{1.15}$$

con $\mathbf{G}(D)$ matrice di dimensioni $k \times n$ i cui elementi sono polinomi in D di grado al massimo pari a $K - 1$ che prende il nome di *matrice generatrice nel dominio trasformato* del codice convoluzionale.

Infine, è bene sottolineare che in luogo della distanza minima introdotta nella Sezione 1.1, per quantificare le capacità correttive dei codici convoluzionali ci si riferisce alla loro *free distance*. Per definire tale parametro è innanzitutto necessario definire il concetto di *peso* di una parola binaria, dato dal numero di 1 presenti nella sequenza. Pertanto, la free distance è data dal minimo peso di una parola all'interno di un codice convoluzionale diversa dalla parola composta da soli zeri.

[2] trasformata in cui D rappresenta un ritardo unitario (*Delay*, appunto) introdotto da un elemento di memoria del registro a scorrimento. Sostituendo $D = z^{-1}$, tale trasformata è comparabile alla trasformata z ([8])

1.3. Codici LDPC

Famiglia di codici lineari presentata da Robert Gallager in [4] e principalmente ignorata per 35 anni a causa della presunta complessità del processo di decodifica. Una delle poche eccezioni, che va tuttavia menzionata, arriva nel 1981 con la pubblicazione di [9], in cui Tanner fornì una rappresentazione grafica di tali codici definita, appunto, *grafo di Tanner*. La riscoperta dei codici LDPC, poi, inizia nel 1995 con la pubblicazione di [5], articolo in cui si evidenziavano i vantaggi in termini di complessità di decodifica dovuti alla struttura della matrice di parità di tali codici.

1.3.1. Codici a blocco LDPC

In generale, i codici LDPC sono codici lineari a blocco definiti come il nullo di una matrice di parità $\mathbf{H}$ *sparsa*, cioè tale da presentare un ridotto numero di 1 in rapporto al numero di 0. Una struttura di questo tipo per la matrice di parità permette di limitare la complessità degli algoritmi di decodifica, che è proporzionale al numero di 1 presenti nella matrice, e quindi di usare operativamente codici con valori di n molto elevato che consentono di avvicinarsi al limite di Shannon[3].

La sparsità di $\mathbf{H}$ inoltre, ne permette una rappresentazione agevole sfruttando i grafi di Tanner [9], di cui si riporta di seguito un esempio.

Data la matrice di parità di un codice a blocco

$$\mathbf{H} = \begin{array}{c} \begin{matrix} v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 \end{matrix} \\ \begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix} \end{array} \begin{matrix} \\ c_1 \\ c_2 \\ c_3 \end{matrix} \tag{1.16}$$

è possibile identificare due insiemi di nodi:

- **Nodi variabile** (*variable nodes*), v_i in figura, in numerosità pari alle colonne di $\mathbf{H}$ e quindi al numero di simboli della parola di codice.
- **Nodi controllo** (*check nodes*), c_j in figura, in numerosità pari alle righe di $\mathbf{H}$ e quindi al numero di simboli di ridondanza introdotti dal codice.

Tramite questo raggruppamento è possibile realizzare il seguente grafo bipartito in cui, come si nota, vi è una connessione tra il nodo controllo c_j e il nodo variabile v_i se e solo se $\mathrm{H}(j, i) = 1$.

In funzione della distribuzione dei pesi delle rige e delle colonne di $\mathbf{H}$, è poi possibile distinguere codici LDPC *regolari* e *irregolari*. Nei codici LDPC regolari ogni

[3] In [10] è stato dimostrato che la capacità di un canale Additive White Gaussian Noise (AWGN) è data da $C = B \log_2 \left(1 + \frac{S}{N}\right)$ [bit/sec] con B banda in [Hz] e $\frac{S}{N}$ rapporto segnale-rumore. Tale relazione, di fatto, pone un limite superiore al Signal-to-Noise Ratio (SNR) dato dalla capacità stessa del canale di trasmissione

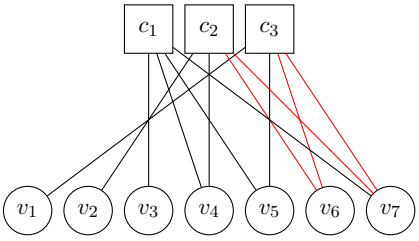


Figura 1.3.: Grafo di Tanner del codice con matrice di parità 1.16

colonna di $\mathbf{H}$ ha peso w_c e ogni riga di $\mathbf{H}$ ha peso w_r mentre, in quelli irregolari, i valori di w_c e w_r variano in base alla colonna o alla riga considerata. Va menzionato che esistono anche codici regolari solo nelle righe o nelle colonne, ovvero, rispettivamente, codici con valori di w_r fisso e w_c variabile e viceversa. In generale, i codici LDPC regolari non soffrono di *error floor*, ovvero un appiattimento della curva di Bit Error Rate (BER) ad alti valori di SNR, ma la loro regione di *waterfall*, ovvero l'intervallo di SNR in cui la curva di BER scende più rapidamente, è tipicamente lontana dal limite di Shannon, rendendo tali codici inutilizzabili per molte applicazioni con vincoli particolarmente stringenti in termini di potenza. I codici LDPC irregolari, invece, si avvicinano maggiormente al limite di Shannon nella regione di waterfall ma soffrono di errror floor, rendendo tali codici inutilizzabili per applicazioni in cui sono richiesti valori di BER molto bassi.

Per evitare problemi di convergenza degli algoritmi di decodifica illustrati nel capitolo 2, che si basano sullo scambio continuo di messaggi tra nodi controllo e nodi variabile, nei codici LDPC è bene evitare di avere cicli di lunghezza 4 nel grafo di Tanner, ovvero è bene evitare che due righe (o due colonne) abbiano più di una posizione in comune che contiene un elemento diverso da zero. Pertanto, definendo come *girth* la lunghezza del ciclo più corto presente all'interno del grafo di Tanner di un dato codice, tale vincolo, detto *row-column constraint*, impone che il girth sia sempre maggiore di 4.

Come riportato in [11], è possibile costruire codici LDPC (J, K)-regolari, ovvero tali per cui ad ogni nodo variabile sono connessi J nodi di controllo e ad ogni nodo di controllo sono connessi K nodi variabile, sfruttando i *protografi*. In particolare, un protografo con rate $R = 1 - n_c/n_v$ è un grafo bipartito (V, C, E) che connette un insieme di n_v nodi variabile $V = \{v_1, v_2, \ldots, v_{n_v}\}$ ad un insieme di n_c nodi di controllo $C = \{c_1, c_2, \ldots, c_{n_c}\}$ tramite un insieme di archi E. Tale protografo può essere rappresentato tramite la sua *matrice base* $\mathbf{B}_{[n_c \times n_v]}$, dove l'elemento $\mathrm{B}(j, i)$ rappresenta il numero di archi che connettono il nodo variabile v_i al nodo di controllo c_j. In generale, un protografo può avere più archi che connettono un nodo variabile ad un nodo di controllo, ovvero $\mathbf{B}$ può avere elementi maggiori di 1. A partire da questa struttura è possibile ricavare la matrice di parità $\mathbf{H}$ di un codice LDPC con $n = Mn_v$ nodi variabile, con M che prende il nome di *fattore di lifting*. Nello specifico, la matrice $\mathbf{H}_{[Mn_c \times Mn_v]}$ di un codice è ottenuta sostituendo ogni elemento

di $\mathbf{B}$ diverso da 0 con una somma di $\mathrm{B}(j,i)$ matrici di permutazione di dimensioni $M \times M$, ed ogni elemento di $\mathbf{B}$ uguale a 0 con una matrice $M \times M$ nulla. Come esempio, si consideri il protografo illustrato in Figura 1.4 e descritto dalla matrice di base

$$\mathbf{B} = \begin{vmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{vmatrix}. \tag{1.17}$$

In virtù di quanto descritto, la matrice di parità corrispondente ad un M-lifting di $\mathbf{B}$ è quindi data da

$$\mathbf{H} = \begin{vmatrix} \Pi_{1,1} & \Pi_{1,2} & \Pi_{1,3} \\ \Pi_{2,1} & \Pi_{2,2} & \Pi_{2,3} \end{vmatrix} \tag{1.18}$$

con $\Pi_{i,j}$ matrice di permutazione di dimensioni $M \times M$. Va infine sottolineato che, poiché un codice a blocco LDPC è definito come il nullo di una matrice di parità $\mathbf{H}$, un protografo definisce un *ensemble* di codici LDPC, formato da tutte le matrici di parità $\mathbf{H}$ ottenibili dal lifting della stessa matrice di base $\mathbf{B}$.

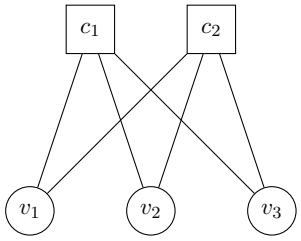


Figura 1.4.: Protografo di un insieme di codici LDPC $(2,3)$-regolari

1.3.2. Codici convoluzionali LDPC

I codici LDPC convoluzionali, o SC-LDPC, introdotti per la prima volta in [6], permettono di unire le principali caratteristiche dei codici a blocco LDPC regolari e irregolari. Infatti, da [12] si riporta che i codici SC-LDPC:

- Permettono l'uso di decodificatori iterativi il cui minimo valore di SNR per il quale si ottengono risultati affidabili, detto *threshold*, è prossimo al limite di Shannon portando ottime prestazioni nella regione di waterfall (caratteristica dei codici irregolari).
- Godono di una crescita lineare della distanza minima con la lunghezza del codice, grazie alla quale è possibile eliminare l'error floor (caratteristica dei codici regolari).

Ciò che rende interessanti questi codici nel panorama moderno è che, come mostrato in [13], i codici SC-LDPC godono del fenomeno noto come *threshold saturation*, in virtù del quale il threshold di un decodificatore che implementa un algoritmo basato su Belief Propagation (BP) tende al threshold di un decodificatore Maximum A Posteriori (MAP). Ciò implica quindi che il threshold di un algoritmo BP basato

sullo scambio di messaggi, che è "localmente ottimo", tende alla threshold di un algoritmo MAP che è, invece, "globalmente ottimo" ma che è altresì eccessivamente complesso.

I codici LDPC convoluzionali sono caratterizzati da una matrice di parità semi-infinita ottenuta replicando diagonalmente più volte una matrice detta *formatrice di sindrome* (trasposta), e indicata da qui in poi con $\mathbf{H}_S^{\mathrm{T}}$. In particolare, $\mathbf{H}_S$ è formata da una sequenza di matrici di parità di codici LDPC a blocco, che risultano essere quindi *interconnesse* tra loro. Pertanto, indicando con m_s la massima distanza che intercorre tra una coppia di blocchi connessi, detta *memoria* del codice convoluzionale, la matrice di parità di un generico codice SC-LDPC sarà nella forma:

$$\mathbf{H}_{[1,L]} = \begin{bmatrix} \mathbf{H}_0(1) & & & & \\ \vdots & \mathbf{H}_0(2) & & & \\ \mathbf{H}_{m_s}(1) & \vdots & \ddots & & \\ & \mathbf{H}_{m_s}(2) & & \mathbf{H}_0(L) & \\ & & \ddots & \vdots & \\ & & & \mathbf{H}_{m_s}(L) & \end{bmatrix} \tag{1.19}$$

in cui la notazione $[1, L]$ sta ad indicare che il codice SC-LDPC è stato *terminato* all'istante temporale $t = L$, ovvero che la matrice formatrice di sindrome $\mathbf{H}_S^{\mathrm{T}}(t) = |\mathbf{H}_0(t) \ \ldots \ \mathbf{H}_{m_s}(t)|^{\mathrm{T}}$ è stata replicata diagonalmente L volte.

Come si nota, gli elementi della formatrice di sindrome (e $\mathbf{H}_S$ stessa) sono generalmente funzioni del tempo. Tuttavia in questo lavoro ci si concentrerà unicamente sui codici SC-LDPC *tempo invarianti*, una classe di codici tale per cui $\mathbf{H}_S(t_i) = \mathbf{H}_S(t_j)\ \forall i, j \in [1, L]$ ovvero tale per cui ogni replica della formatrice di sindrome è identica alle altre.

Per le successive considerazioni si indicheranno con a il numero di colonne di $\mathbf{H}_S^{\mathrm{T}}$, e con c l'entità dello scorrimento verticale tra una sua replica e l'altra. Fissate tali notazioni è possibile definire il *rate asintotico* del codice, dato da

$$R_\infty = 1 - \frac{c}{a} \tag{1.20}$$

che, come intuibile dal nome, è il rate che il codice avrebbe considerando un numero infinito di repliche della matrice formatrice di sindrome, che differisce dal rate effettivo del codice dato da

$$R = 1 - \frac{c}{a}\left(\frac{L + m_s}{L}\right) = R_\infty - \frac{c}{a}\left(\frac{m_s}{L}\right) \tag{1.21}$$

in cui il primo termine coincide col rate asintotico e il secondo termine $\Delta R = \frac{c}{a}(\frac{m_s}{L})$ quantifica esplicitamente la *rate loss* dovuta alla terminazione del codice.

Le definizioni di cui sopra permettono di fornire un'indicazione del numero di repliche minime da considerare affinché il codice terminato abbia prestazioni com-

parabili a quelle che avrebbe in assenza di terminazione; infatti, $\mathbf{H}_S^{\mathrm{T}}$ dovrebbe essere replicato un numero di volte tale per cui $R \approx R_\infty$. In maniera analoga a quanto fatto per i codici LDPC a blocco, da [11] si descrive il concetto di *protografi convoluzionali*. Come intuibile dal nome, un protografo convoluzionale è ottenibile connettendo in cascata sequenze di protografi disgiunti. Tale connessione, detta *spatially coupling*, introduce ovviamente memoria all'interno del codice, ed è realizzata tramite operazioni di *edge spreading*. Per comprendere il significato di questa operazione, si considerino repliche di un protografo con b_v nodi variabile e b_c nodi di controllo come una sequenza infinita di grafi disgiunti. Dato un nodo variabile v_i con $i = 1, 2, \ldots, b_v$ connesso ad un certo nodo di controllo c_j con $j = 1, 2, \ldots, b_c$, tramite $\mathrm{B}(j, i)$ archi in ogni protografo, è possibile realizzare lo *spreading* degli archi originati dal nodo variabile v_i ad un istante t arbitrario connettendo tali archi a $w + 1$ nodi di controllo c_j agli istanti $t, t + 1, \ldots, t + w$, dove $w > 0$ rappresenta la *lunghezza di coupling* del grafo, o, come intuibile, la memoria del codice. Pertanto è possibile ottenere un protografo convoluzionale applicando lo spreading ai nodi variabile ad ogni istante temporale. In particolare, posto $a = \gcd(J, K)$, esisteranno degli interi positivi J', K' tali per cui $J = aJ'$ e $K = aK'$, con $\gcd(J', K') = 1$. Fissate tali notazioni, quindi, la matrice di base di un insieme di codici SC-LDPC (J, K)-regolari con lunghezza di coupling $w = a - 1$ può essere definita come

$$\mathbf{B}_{[-\infty,\infty]} = \begin{bmatrix} \ddots & \ddots & & \ddots & & & \\ & \mathbf{B}_w & \mathbf{B}_{w-1} & \cdots & \mathbf{B}_0 & & \\ & & \mathbf{B}_w & \mathbf{B}_{w-1} & \cdots & \mathbf{B}_0 & \\ & & & \ddots & \ddots & & \ddots \end{bmatrix}, \tag{1.22}$$

dove le sottomatrici $\mathbf{B}_i$ con $i = 1, 2, \ldots, w$ sono matrici di dimensioni $J' \times K'$ con tutti gli elementi uguali a 1, e in cui la notazione $[-\infty, \infty]$ indica che il codice non è stato terminato. A tale proposito, terminare un protografo convoluzionale equivale ad applicare l'edge spreading ad L copie disgiunte di un certo protografo, dove sono consentite connessioni al confine destro con wb_c nodi di controllo aggiuntivi nelle sezioni $t = L, L + 1, \ldots, L + w - 1$. Di conseguenza, in tale situazione, si hanno b_v nodi variabile e b_c nodi di controllo ad ogni istante $t = 0, 1, \ldots, L - 1$ e b_c nodi di controllo aggiuntivi ad ogni istante $t = L, L + 1, \ldots, L + w - 1$. Pertanto, indicando con $n_c = (L + w)b_c$ il numero totale di nodi di controllo e con $n_v = Lb_v$ il numero totale di nodi variabile, la matrice di base $\mathbf{B}_{[0,L-1]}$ di dimensioni $n_c \times n_v$

corrispondente al protografo convoluzionale terminato all'istante L è data da

$$\mathbf{B}_{[0,L-1]} = \begin{bmatrix} \mathbf{B}_0 & & & \\ \mathbf{B}_1 & \mathbf{B}_0 & & \\ \vdots & \mathbf{B}_1 & \ddots & \\ \mathbf{B}_w & \vdots & \ddots & \mathbf{B}_0 \\ & \mathbf{B}_w & & \mathbf{B}_1 \\ & & \ddots & \vdots \\ & & & \mathbf{B}_w \end{bmatrix}. \tag{1.23}$$

Il codice così ottenuto ha un rate dato da

$$R = 1 - \frac{n_c}{n_v} = 1 - \frac{b_c}{b_v}\left(\frac{L+w}{L}\right) = R_\infty - \frac{b_c}{b_v}\left(\frac{w}{L}\right), \tag{1.24}$$

che è assolutamente analogo a quanto riportato nell'equazione (1.21).

Riprendendo il concetto di threshold saturation accennato all'inizio della presente sezione, da [13] si riporta che

$$\lim_{w\to+\infty}\lim_{L\to+\infty}\epsilon^{\mathrm{BP}}(J,K,L,w) = \lim_{w\to+\infty}\lim_{L\to+\infty}\epsilon^{\mathrm{MAP}}(J,K,L,w) = \epsilon^{\mathrm{MAP}}(J,K) \tag{1.25}$$

dove con ϵ^{BP} si intende il threshold di un decodificatore BP e con ϵ^{MAP} il threshold di un decodificatore MAP. Come si nota, dunque, nei codici SC-LDPC le capacità correttive migliorano col crescere della lunghezza di coupling.

Capitolo 2.

Algoritmi di decodifica iterativi

Nella prima sezione di questo capitolo viene trattato uno dei principali algoritmi per la decodifica di codici LDPC, il SPA, e una sua variante, il Min-Sum (MS) Algorithm. Nella seconda sezione, invece, si approfondisce una soluzione pratica per l'applicazione del SPA a codici LDPC convoluzionali, le cui dimensioni elevate delle matrici di parità richiedono approcci che limitino per quanto possibile l'onere computazionale.

2.1. Sum-Product Algorithm e varianti

Introdotto per la prima volta da Gallager in [4], il SPA è un algoritmo di decodifica per codici LDPC basato sullo scambio di *messaggi* tra nodi variabile e nodi controllo basati sull'informazione ricavata dal canale di trasmissione (assunto, in questo lavoro, sempre AWGN) a partire dai Log-Likelihood Ratios (LLRs). In generale, il LLR di una variabile casuale U è definito come

$$L(U) = \log \frac{\Pr(U = 0)}{\Pr(U = 1)} \tag{2.1}$$

dove con $\Pr(U = x)$ si intende la probabilità che la variabile casuale U assuma il valore x.

Indicando la parola trasmessa sul canale con $\mathbf{x}_{[1\times n]}$ e la parola ricevuta con $\mathbf{y}_{[1\times n]}$, la i-esima componente del vettore dei LLR $\boldsymbol{\lambda}_{[1\times n]}$ è data da

$$\lambda_i = \log \frac{\Pr(x_i = 0 \mid y_i = y)}{\Pr(x_i = 1 \mid y_i = y)} \tag{2.2}$$

che, nel caso di trasmissione su un canale AWGN con una modulazione di tipo BPSK, si riduce a

$$\lambda_i = \frac{2y_i}{\sigma^2} \tag{2.3}$$

con σ^2 varianza del canale AWGN. In questo caso, dunque, si ha proporzionalità diretta tra LLRs e valori ricevuti. Inoltre, per com'è definita, λ_i corrisponde alla A Posteriori Probability (APP) associata al i-esimo simbolo trasmesso, ovvero alla quantità che deve essere stimata in fase di decodifica per recuperare il valore di y_i.

Come sarà più chiaro una volta esplicitati i passaggi dell'algoritmo, nodi variabile e nodi di controllo si scambiano iterativamente informazioni estrinseche al fine di stimare il valore di λ_i per $i = 1, \ldots, n$. In base a tali valori, poi, il decodificatore prende decisioni sui simboli y_i.

Per completare la notazione, definiamo l'insieme delle connessioni al nodo variabile v_i dato dai nodi controllo appartenenti a $\mathcal{N}(i) = \{j \mid \mathbf{H}(j,i) = 1\}$ e l'insieme delle connessioni al nodi controllo c_j dato dai nodi variabile appartenenti a $\mathcal{M}(j) = \{i \mid \mathbf{H}(j,i) = 1\}$. Indichiamo poi con $L_{i \to j}$ il messaggio inviato dal nodo variabile v_i e diretto al nodo controllo c_j, e con $L_{i \leftarrow j}$ il messaggio inviato dal nodo controllo c_j e diretto al nodo variabile v_i.

Fatte tali premesse è quindi possibile illustrare i passaggi del SPA.

1. *Inizializzazione dei nodi variabile*:

$$L_{i \to j} = \lambda_i \quad \forall j \in \mathcal{N}(i) \quad \forall i \in \{1, \ldots, n\} \tag{2.4}$$

2. *Aggiornamento dei nodi di controllo*:

$$L_{i \leftarrow j} = 2 \tanh^{-1} \left(\prod_{i' \in \mathcal{M}(j) \setminus \{i\}} \tanh \left(\frac{L_{i' \to j}}{2} \right) \right) \quad \forall i \in \mathcal{M}(j) \quad \forall j \in \{1, \ldots, n-k\} \tag{2.5}$$

3. *Aggiornamento dei nodi variabile*:

$$L_{i \to j} = \lambda_i + \sum_{j' \in \mathcal{N}(i) \setminus \{j\}} L_{i \leftarrow j'} \quad \forall j \in \mathcal{N}(i) \quad \forall i \in \{1, \ldots, n\} \tag{2.6}$$

4. *Calcolo LLR totale*:

$$\gamma_i = \lambda_i + \sum_{j \in \mathcal{N}(i)} L_{i \leftarrow j} \quad \forall i \in \{1, \ldots, n\} \tag{2.7}$$

5. *Decisione della parola stimata*: in base al valore associato al simbolo "0" in fase di modulazione, si ricava la parola stimata $\hat{\mathbf{x}}$ secondo i criteri di decisione

$$\hat{x_i} = \begin{cases} 1 \text{ se } \gamma_i \leq 0 \\ 0 \text{ altrimenti} \end{cases} \quad \text{per modulazioni in cui } 0 \to +1 \tag{2.8}$$

$$\hat{x_i} = \begin{cases} 1 \text{ se } \gamma_i \geq 0 \\ 0 \text{ altrimenti} \end{cases} \quad \text{per modulazioni in cui } 0 \to -1 \tag{2.9}$$

Infine, se $\mathbf{H}\hat{\mathbf{x}}^{\mathbf{T}} = \mathbf{0}$ (o si raggiunge il numero massimo di iterazioni previste) l'algoritmo termina, in caso contrario si riparte dal passo 2.

Come visto nella sezione 1.3.1, il grafo di Tanner di un codice LDPC può presentare cicli che, in termini di scambio di messaggi, implicano che prima o poi l'informazione

inviata da un determinato nodo torna al nodo stesso. Pertanto, in codici LDPC con girth basso, tali messaggi forniscono poche informazioni estrinseche e le prestazioni non sono ottimali. Infatti, indicando con g il girth di un dato codice, per struttura stessa dell'algoritmo è evidente che i messaggi iniziano a tornare ai nodi di partenza dopo $\frac{g}{2}$ iterazioni. Qualora, invece, si avessero codici caratterizzati da grafi privi di cicli si avrebbe la convergenza dell'algoritmo in un numero finito di iterazioni, ma è stato dimostrato che codici LDPC con tali caratteristiche hanno una distanza minima piuttosto ridotta e quindi pessime prestazioni in termini di BER ([8]).

Analizzando l'algoritmo appena illustrato, risulta evidente che il passo più oneroso dal punto di vista computazionale sia quello relativo all'aggiornamento dei nodi di controllo, considerato anche che l'equazione (2.5) richiede il calcolo delle funzioni trigonometriche tanh e $\tanh^{-1}$. Per applicazioni in cui la complessità computazionale costituisce un vincolo stringente è possibile ricorrere ad algoritmi a complessità ridotta che approssimino il SPA, come il MS illustrato in [14]. Innanzitutto, si fattorizza $L_{i\leftarrow j}$ in segno e ampiezza (che rappresentano, rispettivamente, il *valore* del simbolo decodificato e l'*affidabilità* del messaggio) ponendo

$$\begin{aligned} L_{i\leftarrow j} &= \alpha_{ji}\beta_{ji}, \\ \alpha_{ji} &= \operatorname{sign}(L_{i\leftarrow j}), \\ \beta_{ji} &= |L_{i\leftarrow j}| \end{aligned} \tag{2.10}$$

da cui è possibile scrivere

$$\prod_{i'\in\mathcal{M}(j)\setminus\{i\}} \tanh\left(\frac{L_{i'\to j}}{2}\right) = \prod_{i'\in\mathcal{M}(j)\setminus\{i\}} \alpha_{ji'} \cdot \prod_{i'\in\mathcal{M}(j)\setminus\{i\}} \tanh\left(\frac{\beta_{ji'}}{2}\right) \tag{2.11}$$

Quindi il passo 2 del SPA può essere posto come

$$\begin{aligned} L_{i\leftarrow j} &= \prod_{i'\in\mathcal{M}(j)\setminus\{i\}} \alpha_{ji'} \cdot 2\tanh^{-1}\left(\prod_{i'\in\mathcal{M}(j)\setminus\{i\}} \tanh\left(\frac{\beta_{ji'}}{2}\right)\right) \\ &= \prod_{i'\in\mathcal{M}(j)\setminus\{i\}} \alpha_{ji'} \cdot 2\tanh^{-1}\log^{-1}\log\left(\prod_{i'\in\mathcal{M}(j)\setminus\{i\}} \tanh\left(\frac{\beta_{ji'}}{2}\right)\right) \\ &= \prod_{i'\in\mathcal{M}(j)\setminus\{i\}} \alpha_{ji'} \cdot 2\tanh^{-1}\log^{-1}\sum_{i'\in\mathcal{M}(j)\setminus\{i\}} \log\left(\tanh\left(\frac{\beta_{ji'}}{2}\right)\right) \\ &= \prod_{i'\in\mathcal{M}(j)\setminus\{i\}} \alpha_{ji'} \cdot \phi\left(\sum_{i'\in\mathcal{M}(j)\setminus\{i\}} \phi\left(\beta_{ji'}\right)\right) \end{aligned} \tag{2.12}$$

con

$$\phi(x) = -\log\left[\tanh\left(\frac{x}{2}\right)\right] = \log\left(\frac{e^x+1}{e^x-1}\right) \tag{2.13}$$

funzione descrescente nella x e tale per cui $\phi^{-1}(x) = \phi(x)\ \forall x > 0$.

Introdotta questa formulazione alternativa per l'aggiornamento dei nodi di con-

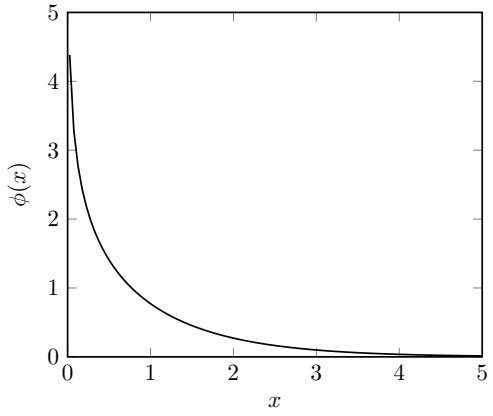


Figura 2.1.: Andamento della funzione $\phi(x)$

trollo, dall'andamento di $\phi(x)$ riportato in Figura 2.1 e dall'equazione (2.12) è possibile affermare che il termine maggiore nella somma corrisponde al β_{ji} più piccolo. Pertanto, supponendo che il termine in questione domini l'intera somma,

$$\begin{aligned}\phi\left(\sum_{i'\in\mathcal{M}(j)\setminus\{i\}}\phi\left(\beta_{ji'}\right)\right) &\approx \phi\left(\phi\left(\min_{i'}\beta_{ji'}\right)\right)\\ &= \min_{i'\in\mathcal{M}(j)\setminus\{i\}}\beta_{ji'}\end{aligned} \tag{2.14}$$

A partire dall'equazione (2.14) è quindi possibile ricavare il MS semplicemente sostituendo il passo 2 del SPA con

$$L_{i\leftarrow j} = \prod_{i'\in\mathcal{M}(j)\setminus\{i\}}\alpha_{ji'}\cdot\min_{i'\in\mathcal{M}(j)\setminus\{i\}}\beta_{ji'} \tag{2.15}$$

In [7] è stato mostrato che la penalizzazione in termini di prestazioni introdotta dal MS rispetto al SPA dipende dal codice utilizzato e dal canale di trasmissione considerato, ed è dovuta al fatto che nella prima iterazione il MS tende ad attribuire ampiezza maggiore ai messaggi, ovvero tende a sovrastimarne l'affidabilità rispetto a quanto succede col SPA. A fronte di tali considerazioni, come possibile soluzione si può introdurre un fattore di attenuazione $0 < c_{atten} < 1$ ai messaggi prima che essi vengano trasmessi ai nodi variabile, aggiornando quindi i nodi di controllo con

$$L_{i\leftarrow j} = \prod_{i'\in\mathcal{M}(j)\setminus\{i\}}\alpha_{ji'}\cdot c_{atten}\cdot\min_{i'\in\mathcal{M}(j)\setminus\{i\}}\beta_{ji'} \tag{2.16}$$

In alternativa, in [15] è stata proposta una versione modificata del MS basata sull'introduzione di un offset $c_{offset} > 0$ da sottrarre ad ogni β_{ji} dei messaggi diretti ai nodi variabile. Nell'implementazione di cui sopra, quindi, i nodi di controllo vengono

aggiornati con

$$L_{i \leftarrow j} = \prod_{i' \in \mathcal{M}(j) \setminus \{i\}} \alpha_{ji'} \cdot \max \left\{ \min_{i' \in \mathcal{M}(j) \setminus \{i\}} \beta_{ji'} - c_{offset}, 0 \right\} \tag{2.17}$$

Va sottolineato che, nelle successive analisi e simulazioni, ci si concentrerà unicamente sull'algoritmo SPA, tuttavia è lecito aspettarsi esattamente gli stessi risultati e le stesse conclusioni usando algoritmi MS e varianti. Tale scelta è motivata da questioni di implementazione degli algoritmi trattati sopra. Come riportato nella sezione A.1 dell'appendice A, il SPA è stato implementato sfruttando unicamente operazioni tra matrici, soluzione che permette di calcolare sommatorie e produttorie con solamente due operazioni. Al contrario, nel MS, la ricerca dei termini $\min_{i' \in \mathcal{M}(i) \setminus \{i\}} \beta_{ji'}$ deve necessariamente essere effettuata in maniera sequenziale sfruttando un ciclo `for` che operi sui messaggi inviati dai nodi variabile ai nodi di controllo. Pertanto, essendo il linguaggio MATLAB ottimizzato proprio per le operazioni matriciali e, al tempo stesso, più lento a svolgere operazioni basilari rispetto a linguaggi di livello più basso come C o C++, si ha che, per elevati valori di n, il tempo necessario ad eseguire il ciclo di cui sopra è superiore al tempo di esecuzione delle funzioni trigonometriche che rendono il SPA computazionalmente più oneroso del MS. Dunque, poiché i codici utilizzati in questo lavoro sono caratterizzati da valori di n elevati, l'esecuzione del SPA risulta complessivamente più rapida dell'esecuzione del MS.

2.2. SPA a finestra scorrevole per codici LDPC convoluzionali

Come risulta evidente dalla struttura del SPA, la sua complessità computazionale è strettamente collegata al numero di nodi variabile e nodi controllo presenti nella matrice di parità, ovvero dipende fortemente dalle dimensioni di $\mathbf{H}$. Pertanto, per i codici SC-LDPC l'applicazione diretta dell'algoritmo risulta essere computazionalmente e temporalmente molto onerosa. Inoltre, il SPA necessita innanzitutto che siano stati ricevuti tutti gli n simboli provenienti dal canale di trasmissione e produce il risultato, ovvero la parola stimata, solo alla fine dell'intera elaborazione. Chiaramente una soluzione di questo tipo è assolutamente inefficiente per matrici di parità teoricamente infinite, o comunque estremamente lunghe, come sono quelle dei codici SC-LDPC. Per tale ragione, in [16] è stato presentato un *decoder a finestra scorrevole* basato proprio sul SPA.

Il decodificatore in questione applica il SPA ad una porzione ristretta della matrice di parità, una *finestra* appunto, che include un numero di nodi variabile e nodi controllo proporzionale alla memoria del codice SC-LDPC. Infatti, per definizione stessa di memoria, il numero massimo di equazioni di parità che coinvolgono un determinato nodo variabile è pari a $(m_s+1)J'M$, e l'equazione di parità identificata da un determinato nodo controllo coinvolge al massimo $(m_s+1)K'M$ nodi variabile.

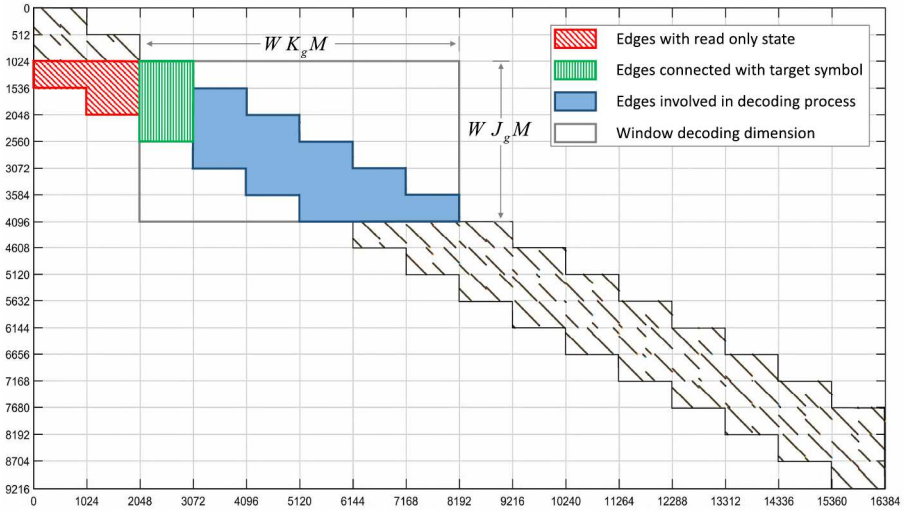
Dunque, definita una costante W che quantifichi il numero di blocchi di colonne della matrice formatrice di sindrome trasposta (indicati con $\mathbf{H}_n(t)$ nell'equazione (1.19)) considerati simultaneamente all'interno di una finestra, il decoder a finestra scorrevole applica il SPA a porzioni di $\mathbf{H}$ di dimensioni $[WJ'M \times WK'M]$. Per quanto concerne il valore di W, solitamente si assume $W = \rho(m_s+1)$ con $5 \leq \rho \leq 10$ poiché è stato verificato empiricamente che le prestazioni sono vicine a quelle ottenute con valori di W più grandi a fronte di una complessità di decodifica contenuta. Va sottolineato che ad ogni scorrimento della finestra i simboli effettivamente decodificati, indicati da ora in poi come *simboli target*, sono unicamente i primi $K'M$, e non tutti i $WK'M$. Difatti, una volta decodificati i simboli target di una certa posizione p della finestra, essa scorre di $K'M$ colonne verso destra e di $J'M$ righe verso il basso all'interno della matrice di parità. Tuttavia, per quanto detto riguardo la memoria di un codice SC-LDPC, al fine di considerare equazioni di parità complete durante il processo di decodifica, è necessario includere ad ogni posizione tutti i nodi connessi ai simboli target, anche quelli appartenenti alle m_s posizioni precedenti a quella considerata. Per tale ragione, alla finestra definita sopra è necessario aggiungere $m_sJ'M$ nodi controllo e $m_sK'M$ nodi variabile che passino i propri LLRs totali (equazione (2.7)) ai nodi compresi all'interno della finestra di decodifica.

Per terminare l'analisi, si riportano le modifiche al decoder a finestra scorrevole proposte in [17]. Tali modifiche, nello specifico, riguardano:

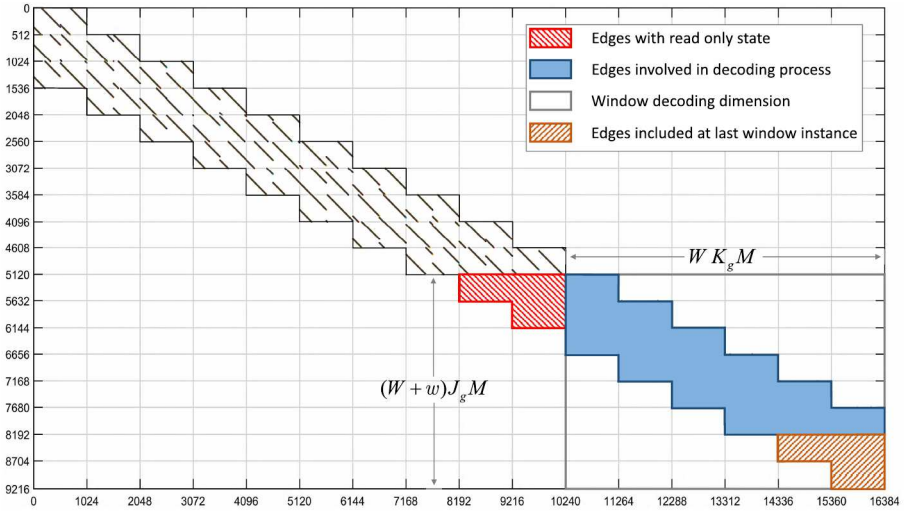
- *Terminazione anticipata della finestra scorrevole*: in generale, quando $p = L - W + 1$, la finestra scorrevole raggiunge l'estremo destro della matrice di parità. In tale situazione, nella versione convenzionale del decoder a finestra scorrevole, tale finestra viene fatta scorrere all'esterno della matrice riducendo di volta in volta il numero di nodi controllo e nodi variabile inclusi nel processo di decodifica fino all'elaborazione degli ultimi a simboli target. Al contrario, viene proposto di fermare la finestra alla posizione $p = L - W + 1$ considerando tutti i simboli inclusi in tale finestra come simboli target e estendendo la dimensione verticale di essa per includere i nodi controllo rimanenti.

- *Riuso dei messaggi*: in [18] i nodi connessi ai precedenti simboli target vengono inizializzati coi relativi LLRs totali che, nel caso di decodifica errata, causano una forte propagazione dell'errore all'interno della finestra. In [17], invece, è stato mostrato che i LLRs *estrinseci* (calcolati nell'equazione (2.6)) risultano essere una scelta migliore. Per tale ragione, ad ogni scorrimento, i nodi connessi ai precedenti simboli target vengono inizializzati con i LLRs estrinseci invece dei totali, e segue quindi che gli unici nodi ad essere inizalizzati con i LLRs provenienti dal canale sono i nodi inclusi nella posizione p e non nella posizione $p-1$.

In Figura 2.2 vi è una rappresentazione grafica del funzionamento generale del decoder (Figura 2.2a) e della sua terminazione (Figura 2.2b).



(a) Decoder a finestra scorrevole



(b) Terminazione della finestra

Figura 2.2.: Rappresentazione grafica del decoder a finestra scorrevole ([17])

In Figura 2.3, invece, sono riportate le curve di BER ottenute usando il decoder a finestra scorrevole con terminazione classica e terminazione anticipata sul codice 1 descritto in dettaglio nella Sezione 3.1 del Capitolo 3.

Come si nota, la versione con terminazione anticipata garantisce prestazioni generalmente migliori e, per tale ragione, da questo punto in poi tutti i risultati presentati faranno riferimento a questo tipo di decodificatore.

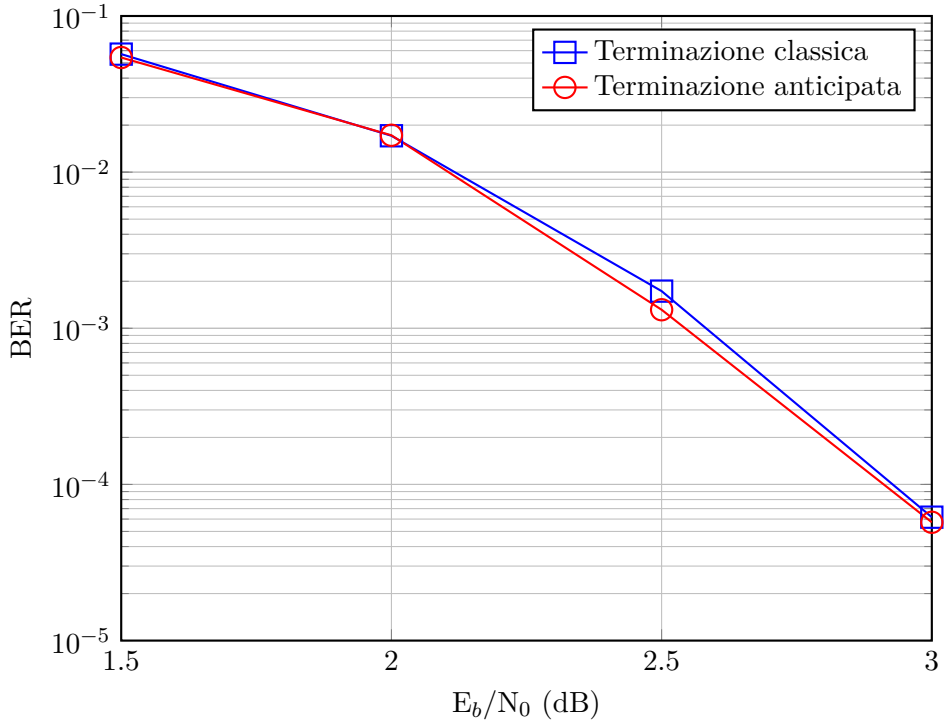


Figura 2.3.: Confronto tra decoder a finestra scorrevole con terminazione classica e terminazione anticipata, applicati al codice 1 riportato nella Sezione 3.1 con $R = 0.6567$, $L = 400$, $W = 91$, $m_s = 12$

Capitolo 3.

Implementazione del SPA a finestra scorrevole

In questo capitolo vengono inizialmente presentati i codici usati per le simulazioni. Successivamente, si analizzano le prestazioni di un decoder a finestra scorrevole, ponendo particolare attenzione al rapporto tra latenza e BER i quali, come intuibile, risulteranno essere inversamente proporzionali. Nella terza sezione, poi, si propongono delle modifiche al funzionamento del decoder basate sul reset dei messaggi mantenuti tra una posizione della finestra scorrevole e la successiva al fine di mitigare la propagazione dell'errore. Quest'ultima, infine, è proprio oggetto di studio nella quarta sezione del capitolo, nella quale, osservando l'andamento dei LLRs, si analizzano le cause d'errore del sistema di cui sopra.

3.1. Codici usati per le simulazioni

1. Codice SC-LDPC con sottomatrice formatrice di sindrome generata a partire dalla matrice polinomiale

$$H(x) = \begin{vmatrix} x^{11} & x^9 & x^8 & x^7 & x^2 & 1 & 1 & x^4 & 1 & x^3 & x & x^9 \\ x^5 & 1 & 1 & x^{10} & 1 & x & x^{10} & x^{11} & x^7 & x^9 & 1 & x^5 \\ 1 & x^3 & x^9 & 1 & x^4 & 1 & x^3 & x^{11} & x^8 & 1 & x^{10} & x^8 \\ x^{11} & x^8 & x^4 & 1 & x & x^8 & x^9 & 1 & x^2 & x^7 & x^{11} & 1 \end{vmatrix}$$

   in cui il massimo esponente rappresenta la memoria del codice, che in questo caso implica $m_s = 12$. Questa notazione rappresenta in maniera compatta una matrice formata da $m_s + 1$ blocchi di dimensioni pari a quelle di $H(x)$ in ognuno dei quali si ha $H_n(i, j) = 1$ solo se $H(x)(i, j) = x^{n+1}$. Nel caso in questione, dunque, $\mathbf{H}_S^{\mathrm{T}}$ è formata da 12 blocchi di dimensioni 4×12 e il codice ha $R_\infty = \frac{2}{3}$.

2. Codice SC-LDPC con sottomatrice formatrice di sindrome

$$\mathbf{H}_S^{\mathrm{T}} = \begin{vmatrix} \bar{H}_0 \\ \bar{H}_1 \end{vmatrix}$$

i cui elementi sono dati da

$$\bar{H}_0 = \begin{vmatrix} I^1 & I^9 & I^{13} & I^{15} & I^{16} & I^8 & I^4 & I^2 \\ I^{16} & I^8 & I^4 & I^2 & I^1 & I^9 & I^{13} & I^{25} \end{vmatrix}$$

$$\bar{H}_1 = \begin{vmatrix} I^{10} & I^5 & I^{11} & I^{14} & I^7 & I^{12} & I^6 & I^3 \\ I^7 & I^{12} & I^6 & I^3 & I^{10} & I^5 & I^{11} & I^{14} \end{vmatrix}$$

dove con I^x si intende una permutazione circolante della matrice identità di x posizioni, ovvero uno spostamento circolare di x posizioni verso l'alto o verso il basso di ognuna delle colonne di I. Ad esempio, considerata la matrice identità

$$I = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}, \tag{3.1}$$

La matrice I^3 è data da

$$I^3 = \begin{vmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{vmatrix}. \tag{3.2}$$

Il codice in questione è caratterizzato da $R_\infty = \frac{3}{4}$.

Durante l'analisi dei risultati presentati in questo lavoro, verranno indicati esplicitamente il numero di repliche L della matrice formatrice di sindrome e il rate effettivo R dei codici così ottenuti.

3.2. Prestazioni del sistema

Al fine di quantificare il guadagno in termini di latenza che si ha usando un decoder a finestra scorrevole in luogo di un decoder SPA che lavora sull'intera matrice di parità, analizziamo per prima cosa la latenza del decoder a finestra scorrevole convenzionale, ovvero quello in cui la finestra scorre di L posizioni fino a decodificare gli ultimi a simboli target. Innanzitutto, la latenza dovuta alla decodifica dei simboli target in una particolare posizione p della finestra scorrevole può essere scritta come

$$\tau^p_{\mathrm{WD}} = T^p_R(W) + T^p_D(W) \tag{3.3}$$

dove $T^p_R(W)$ è il tempo necessario a ricevere Wa simboli alla posizione p e $T^p_D(W)$ è il tempo necessario a decodificare i simboli target. Prima di procedere nell'analisi è fondamentale sottolineare che se il numero di iterazioni per l'algoritmo di decodifica dei simboli target non è fisso, ovvero se è previsto un criterio di stop (come il criterio

delle sindromi parziali riportato nella sezione 2.1), il valore di τ_{WD}^{p} è anche funzione di p.

Per semplificare la trattazione, dunque, si assume che l'algoritmo di decodifica operi per un numero di iterazioni fissato a priori e indipendente dalla posizione considerata, portando la seguente analisi a fornire un limite superiore della latenza di un decoder con un criterio di stop. Pertanto, posto T_F il tempo necessario a ricevere tutti i simboli di una trasmissione e T_D il tempo necessario alla decodifica di tutti i simboli in questione, è immediato notare che, in media, valgono le relazioni

$$T_R^p(W) = \begin{cases} \dfrac{Wa}{La} T_F = \dfrac{W}{L} T_F & \text{per } 1 \leq p \leq L - W + 1 \\ \dfrac{(L-p+1)a}{La} T_F = \left(\dfrac{L-p+1}{L} \right) T_F & \text{per } L - W + 1 \leq p \leq L \end{cases} \tag{3.4}$$

e, analogamente,

$$T_D^p(W) = \begin{cases} \dfrac{Wa}{La} T_D = \dfrac{W}{L} T_D & \text{per } 1 \leq p \leq L - W + 1 \\ \dfrac{(L-p+1)a}{La} T_D = \left(\dfrac{L-p+1}{L} \right) T_D & \text{per } L - W + 1 \leq p \leq L \end{cases} \tag{3.5}$$

Pertanto, a parità di potenza di calcolo, la latenza di decodifica di una singola finestra del decoder a finestra scorrevole è legata alla latenza del decoder SPA dalle relazioni

$$\tau_{\mathrm{WD}}^{p}(W) = \begin{cases} \dfrac{W}{L} \tau_{\mathrm{SPA}} & \text{per } 1 \leq p \leq L - W + 1 \\ \left(\dfrac{L-p+1}{L} \right) \tau_{\mathrm{SPA}} & \text{per } L - W + 1 \leq p \leq L \end{cases} \tag{3.6}$$

in virtù delle quali la latenza della finestra scorrevole è ridotta di un fattore W/L rispetto al decoder SPA fino ai simboli target alla posizione $p = L - W + 1$, oltre la quale le dimensioni della finestra scorrevole diminuiscono e il fattore di guadagno diventa $(L - p + 1)/L$.

Per quanto concerne la latenza di decodifica nell'intera trasmissione consideriamo, per semplicità, che la ricezione di un blocco di nodi variabile e la decodifica della corrispondente finestra non si verifichino contemporaneamente. Sotto tale ipotesi,

da [17] si riporta la relazione

$$\tau^F_{\mathrm{WD}} = T_F + \sum_{p=1}^{L-W+1} \frac{W}{L} T_D + \sum_{i_p=1}^{W-1} \left(\frac{W - i_p}{L} \right) T_D. \tag{3.7}$$

Considerando la versione del decoder a finestra scorrevole con terminazione anticipata dell'ultima finestra illustrata nella sezione 2.2, nella quale la finestra di decodifica non scorre oltre $p = L - W + 1$, il guadagno in termini di latenza è quantificabile dalla relazione

$$\tau^F_{\mathrm{etWD}} = T_F + \sum_{p=1}^{L-W+1} \frac{W}{L} T_D. \tag{3.8}$$

Come accennato nell'introduzione di questo capitolo, tuttavia, il guadagno in termini di latenza implica necessariamente una perdita in prestazioni. Pertanto, in Figura 3.1 si riporta l'andamento delle curve di BER al variare della dimensione della finestra scorrevole W. Per l'analisi in questione si è considerato il decoder illustrato nella sezione 2.2 applicato al codice 1 riportato nella sezione 3.1 e caratterizzato da $m_s = 12$. Come si nota dalla Figura 3.1, per $W \geq 91 = 7(m_s + 1)$ si hanno

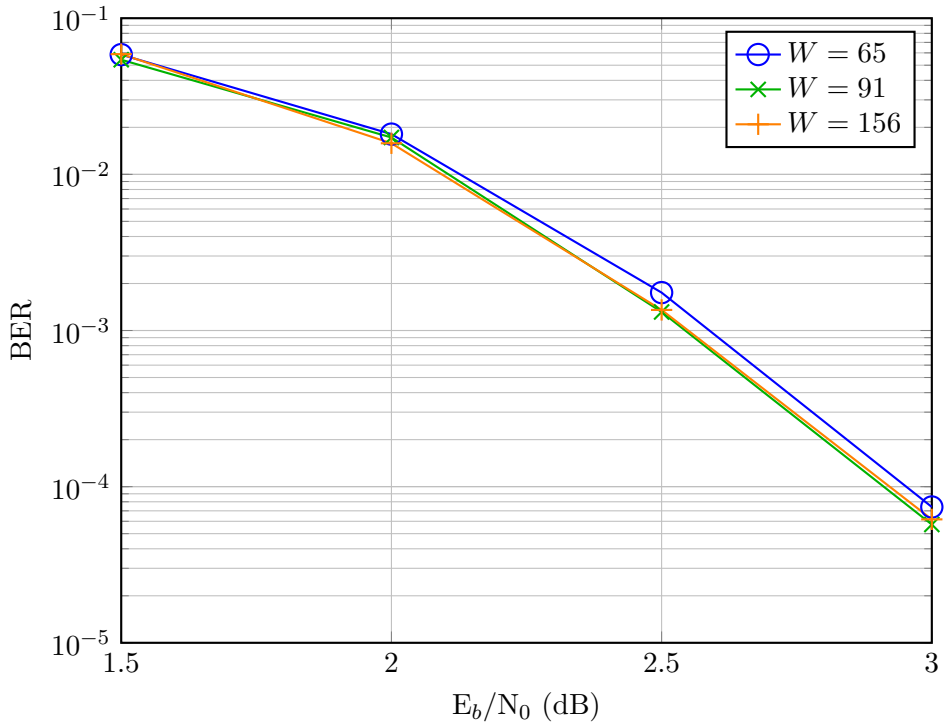


Figura 3.1.: Confronto tra decoder a finestra scorrevole al variare di W applicati al codice 1 riportato nella Sezione 3.1 con $R = 0.6567$, $L = 400$, $m_s = 12$

miglioramenti marginali in termini di prestazioni. Per questa ragione, in seguito non si considereranno dimensioni della finestra maggiori di tale valore.

Al fine di confrontare l'onere computazionale dovuto all'elaborazione di una parola di codice al variare di W, si riportano di seguito i tempi medi di esecuzione della funzione MATLAB illustrata nella sezione A.2.2 dell'appendice A per una trasmissione modulata BPSK con SNR di 3dB col codice 1 riportato nella Sezione 3.1, per il quale si è assunto $L = 2000$, ovvero $R = 0.6647$:

- $W = 65 \rightarrow 6.614$s
- $W = 91 \rightarrow 11.376$s
- $W = 156 \rightarrow 28.560$s

Come si nota, generalmente W è direttamente proporzionale alla complessità di calcolo e alle prestazioni in termini di BER e, chiaramente, inversamente proprozionale alla latenza necessaria all'elaborazione della parola trasmessa.

3.3. Effetti del reset dei messaggi

Come verrà approfondito nella sezione 3.4, i decoder a finestra scorrevole soffrono di *propagazione degli errori* dovuta alla struttura stessa del sistema. Infatti, come illustrato nella sezione 2.2, l'informazione relativa ad un determinato simbolo target viene mantenuta nei successivi m_s scorrimenti senza essere modificata, pertanto un'eventuale decodifica errata potrebbe compromettere la decodifica dei simboli successivi.

Per tentare di mitigare tale propagazione, si propone quindi di *resettare* i nodi che dovrebbero essere inizializzati con i LLRs ricavati dall'elaborazione dello scorrimento precedente a quello considerato, inizializzandoli invece ai LLRs provenienti dal canale di trasmissione. Tale scelta permetterebbe, al verificarsi di determinate condizioni, di limitare la propagazione di LLRs relativi a decisioni erronee migliorando le prestazioni in termini di tasso d'errore. Riguardo ai criteri con cui si effettua o meno un reset all'interno del processo di decodifica si propone innanzitutto:

- *Reset statico*: i nodi vengono resettati o meno in base alla posizione della finestra di decodifica. In termini matematici, viene effettuato un reset alla posizione p se $\mod(p, f) = 0$, con f fattore di reset. Tale approccio si basa sull'idea di supporre che i LLRs non siano più affidabili dopo f scorrimenti, ragione per cui si propone di resettare periodicamente i valori dei LLRs supposti erronei per evitare a priori che essi si propaghino agli scorrimenti successivi.
- *Reset dinamico*: i nodi vengono resettati quando i LLRs ereditati non sono più considerati affidabili. In particolare, per determinare l'affidabilità di tali LLRs si è scelto di controllare il numero delle iterazioni del SPA richieste per la decodifica della finestra alla posizione $p - 1$, ma chiaramente altri criteri sono possibili. A differenza dell'approccio statico, con tale soluzione si propone di resettare i valori dei LLRs solo quando essi sono effettivamente considerati

inaffidabili. Pertanto, con questo approccio il numero di reset e le posizioni in cui esso viene effettuato non sono noti prima di iniziare la decodifica.

In aggiunta ad inizializzare tutti i nodi inclusi nella finestra di decodifica p ai LLRs del canale di trasmissione, si propongono modalità di reset che riguardino solamente i simboli target nella finestra $p-1$, e tutti i blocchi in sola lettura connessi alla finestra p. Nello specifico, si propone di:

1. Resettare i simboli inclusi nella finestra di decodifica p e i simboli target decodificati nella finestra $p-1$ ai LLRs del canale, scelta che si basa sul considerare più affidabili i LLRs derivati dal canale di trasmissione rispetto a quelli derivati dalla decodifica della finestra $p-1$, invece supposti fonte di errore.

2. Resettare i simboli inclusi nella finestra di decodifica p ai LLRs del canale e i simboli target decodificati nella finestra $p-1$ a 0, scelta basata sul fatto che, in termini di LLRs, 0 rappresenta il valore di massima incertezza, ed equivale di fatto a non prendere alcuna decisione in merito al valore del simbolo considerato. Ciò implica che, con questo tipo di reset, quando i LLRs sono considerati inaffidabili l'informazione associata ai simboli target della finestra $p-1$ non viene usata per la decodifica dei simboli target della finestra p.

3. Resettare i simboli inclusi nella finestra di decodifica p ai LLRs del canale e il numero di blocchi in sola lettura ad essa connessi a 0, decodificandola come se fosse $p = 1$, scelta che rompe la dipendenza tra la finestra attuale e le precedenti m_s. Questa soluzione, a differenza delle precedenti, causa una rate loss che verrà trattata nel dettaglio in seguito.

Innanzitutto, nelle Figure 3.2 e 3.3 sono riportati confronti a parità di codice e di modalità di reset tra approccio statico e approccio dinamico.

Come si evince da tali risultati dunque, il reset dinamico risulta essere sempre migliore del reset statico, indipendentemente dal fattore di reset considerato per quest'ultimo. Nello specifico, il guadagno ottenuto dall'applicazione del reset dinamico anziché quello statico risulta più evidente all'aumentare del valore di SNR considerato. Infatti, come enfatizzato particolarmente nella Figura 3.3, la differenza in prestazioni ad alti valori di E_b/N_0 è dovuta al fatto che, in tali condizioni, i LLRs provenienti dal canale sono già molto affidabili e quindi, per natura stessa del reset dinamico, il normale processo di decodifica viene influenzato in maniera minore ovvero, nel caso specifico del reset 2, all'interno di ogni frame trasmesso sono molti meno i simboli i cui LLRs sono stati inizializzati a 0. Al contrario, utilizzando il reset statico, il numero di simboli inizializzati a 0 rimane costante a prescindere dallo scenario preso in esame e pertanto vengono resettati anche simboli con LLRs molto affidabili, compromettendo il processo di decodifica. Per tale ragione, da questo punto in poi all'interno della trattazione si farà riferimento unicamente al reset dinamico in luogo di quello statico.

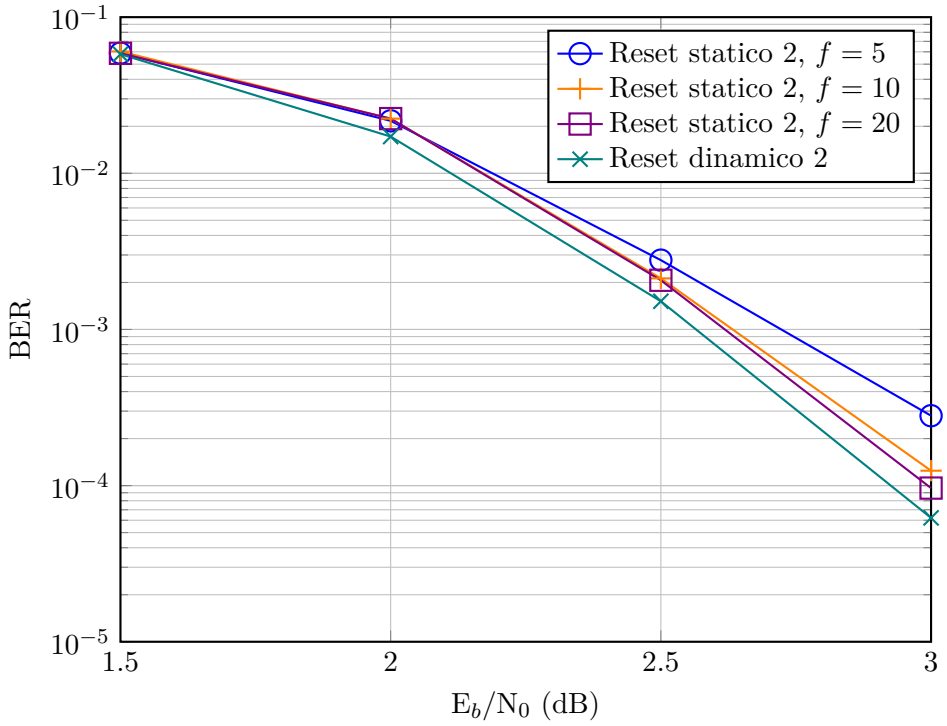


Figura 3.2.: Confronto tra reset statico e dinamico applicati al codice 1 riportato nella Sezione 3.1 con $R = 0.6567, L = 400, W = 65, m_s = 12$

Di seguito si analizzano le prestazioni delle differenti modalità di reset illustrate sopra, applicate ai codici della Sezione 3.1. Per quanto concerne il codice 1, come si nota dalla Figura 3.4, i reset 1 e 2 hanno un piccolo guadagno di BER rispetto al decoder convenzionale, senza però alterare in alcun modo il processo di codifica della parola di codice poiché non prevedono assunzioni sul valore dei simboli trasmessi o sulle equazioni di parità del codice.

Al contrario, col reset 3 si ha un visibile incremento delle performance ma, per funzionamento stesso di tale soluzione, si ha una non trascurabile rate loss (la cui analisi verrà fatta alla fine di questa sezione) dovuta al fatto che resettare in questo modo significa assumere che le porzioni di equazioni di parità relative ai blocchi in sola lettura connessi ai nodi variabile inclusi nella finestra di decodifica considerata siano verificate. Inoltre, è bene sottolineare che tutte le curve qui riportate sono state ottenute tramite simulazione Monte Carlo di trasmissioni modulate BPSK della parola di codice tutta zeri, per la quale le assunzioni di cui sopra sono sicuramente corrette. La scelta di limitarsi unicamente alla parola di codice tutta zeri è motivata dall'elevata complessità del processo di codifica. Come riportato in [19], infatti, il tradizionale processo di codifica di codici LDPC prevede innanzitutto di ricavare la matrice generatrice $\mathbf{G}$ a partire dalla matrice di parità $\mathbf{H}$ riducendo quest'ultima alla sua *forma sistematica* $\mathbf{H}' = \{\mathbf{P}, \mathbf{I}\}$ tramite il metodo di eliminazione di Gauss e, successivamente, come indicato nella Sezione 1.2 del Capitolo 1, ricavare le parole di

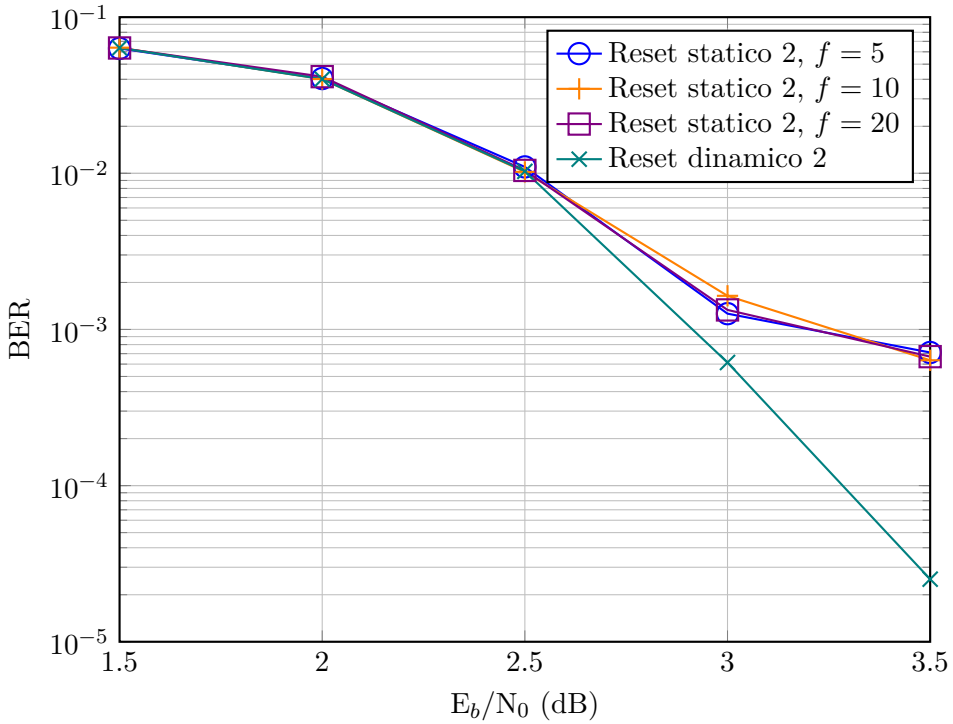


Figura 3.3.: Confronto tra reset statico e dinamico applicati al codice 2 riportato nella Sezione 3.1 con $R = 0.7450$, $L = 50$, $W = 10$, $m_s = 1$

codice tramite la relazione $\mathbf{c} = \mathbf{u} \cdot \mathbf{G}$. In particolare, la complessità computazionale dovuta alla riduzione di $\mathbf{H}_{[m \times n]}$ alla sua forma sistematica è dell'ordine di $\mathcal{O}(n^3)$ e, poiché in tale operazione solitamente viene meno la sparsità di $\mathbf{H}$, la complessità dell'effettivo processo di codifica è quella di una moltiplicazione tra matrici. In altri termini, dunque, la complessità del processo di codifica è dell'ordine di $\mathcal{O}(n^2)$, che risulta spesso proibitiva per valori elevati di n. Per completezza, è bene specificare che esistono dei metodi di codifica basati sulla matrice di parità $\mathbf{H}$. Ad esempio, in [20], è riportato un metodo di codifica che si basa sul ricavare i $n_c M$ simboli di ridondanza a partire dai $n_v M - n_c M$ simboli ricavati dalla sequenza informativa in ingresso. Nel documento di cui sopra, inoltre, è riportato un limite superiore per la distanza minima di codici SC-LDPC tempo-invarianti, dato da

$$d_{\min} \leq (n_c + 1)(n_c m_s + 1). \tag{3.9}$$

Pertanto, al fine di progettare codici con valori di $d_{\min}$ sufficentemente elevati, si può pensare di:

1. Massimizzare n_c espandendo le dimensioni delle sottomatrici. Tale soluzione, tuttavia, rischia di aumentare la complessità del processo di codifica, ragione per cui si assumono valori di m_s ridotti, solitamente 1 o 2. I codici così costruiti, quindi, vengono detti debolmente accoppiati.

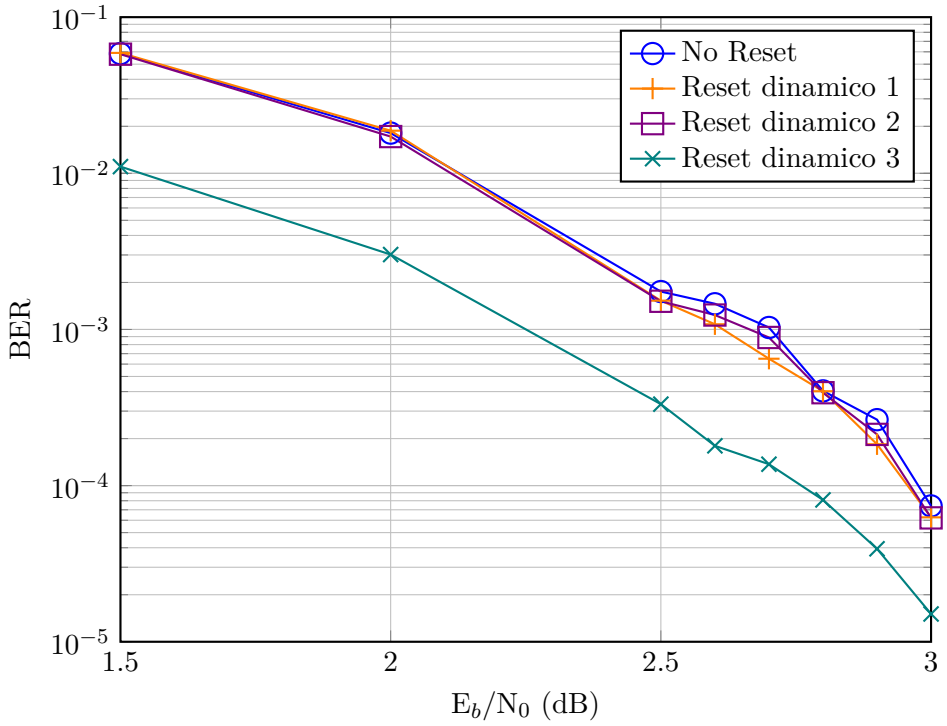


Figura 3.4.: Confronto tra diversi tipi di reset applicati al codice 1 riportato nella Sezione 3.1 con $R = 0.6567, L = 400, W = 65, m_s = 12$

2. Mantenere contenuto il valore di n_c e aumentare la memoria del codice m_s. I codici così costruiti vengono detti fortemente accoppiati.

Passando invece al codice 2, caratterizzato da una memoria molto minore rispetto al codice 1, si ottengono i risultati riportati in Figura 3.5, a partire dai quali si nota comunque un guadagno rispetto all'implementazione classica, ma senza il notevole miglioramento del reset 3 visto in Figura 3.4. Tale discrepanza di risultati è dovuta al fatto che le curve in Figura 3.4 sono riferite al codice 1, caratterizzato da una memoria $m_s = 12$ e quindi da una interdipendenza tra scorrimenti successivi molto più forte rispetto a quella del codice 2 a cui fa riferimento la Figura 3.5. Infatti, poiché il codice in questione ha memoria $m_s = 1$, si ha che l'esito della decodifica di una certa finestra non risulta essere fortemente condizionato dall'esito delle decodifiche precedenti, pertanto il contributo dovuto all'implementazione del reset dei messaggi è molto più contenuto. A fronte di tali risultati, dunque, si potrebbero progettare appositamente codici con valori di m_s elevati proprio per favorire tecniche di miglioramento della decodifica a finestra scorrevole. In quest'ottica va tuttavia sottolineato che, come riportato nell'equazione (1.21), la rate loss, a parità di L, è direttamente proporzionale alla memoria del codice terminato e, come riportato nella sezione 2.2, solitamente anche il valore di W è direttamente proporzionale ad m_s. Pertanto, progettare codici con memoria elevata implica dover gestire valori

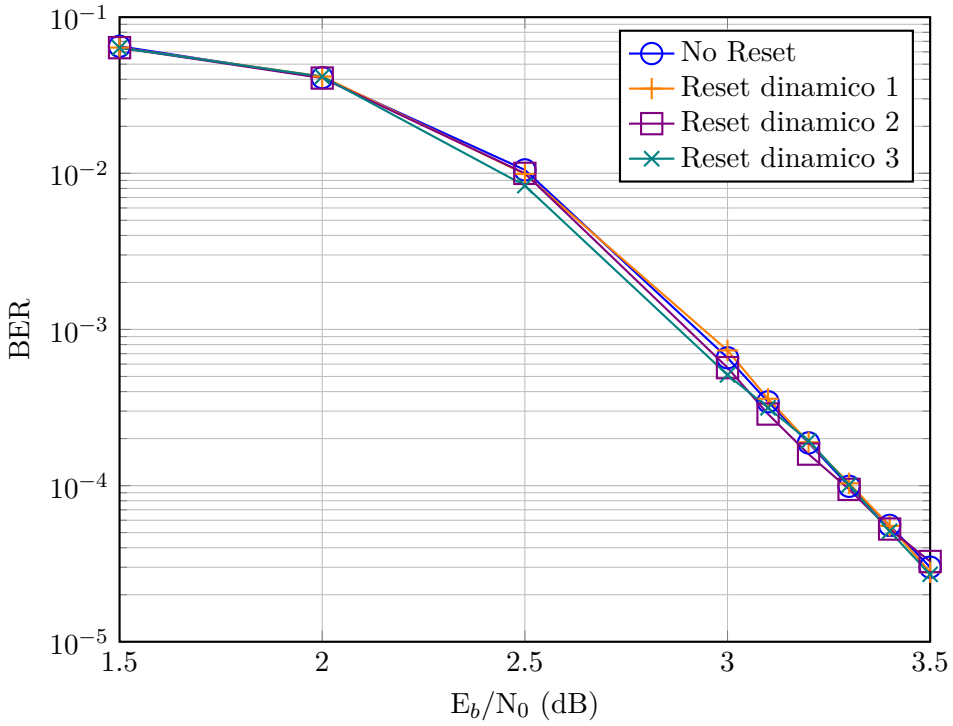


Figura 3.5.: Confronto tra diversi tipi di reset applicati al codice 2 riportato nella Sezione 3.1 con $R = 0.7450$, $L = 50$, $W = 10$, $m_s = 1$

di L maggiori (per limitare la rate loss) e finestre più grandi, ovvero, in generale, tollerare una latenza più elevata.

Per quanto concerne le modalità di reset che operano sui simboli target, ovvero le modalità 1 e 2, si è pensato anche di limitare la loro azione resettando unicamente una data percentuale di simboli target considerati meno affidabili, da intendersi come i simboli target con LLRs di valore assoluto minore. In particolare, nelle Figure 3.6 e 3.7 sono riportate le curve di BER ottenute applicando tale soluzione al codice 1, mentre nelle Figure 3.8 e 3.9 sono riportate le curve di BER ottenute applicando tale soluzione al codice 2. Tuttavia, come si nota dalle Figure citate sopra, non si ha un guadagno omogeneo e apprezzabile in termini di prestazioni.

La modalità di reset 3, invece, può essere vista come una sorta di *VNs doping*, concetto presentato e descritto nel dettaglio in [21]. In tale documento, infatti, viene proposto di fissare il valore dei LLRs inviati da alcuni nodi variabile al massimo valore noto ovvero, in virtù dell'equazione (2.8), di fissare il valore dei simboli dopati a 0, in modo da mitigare la propagazione degli errori introducendo nodi che inviino informazioni sicuramente affidabili durante il processo di decodifica. Tale soluzione, così come la modalità di reset 3 presentata in questo lavoro, garantisce un guadagno in termini di prestazioni ma causa inevitabilmente una rate loss dovuta al fatto che, fissando a 0 il valore di un certo simbolo c_j all'interno di una parola di codice $\mathbf{c} = (c_0, c_1, \ldots, c_{n-1})$, non tutte le 2^k possibili parole risulteranno valide. Quest'ultima

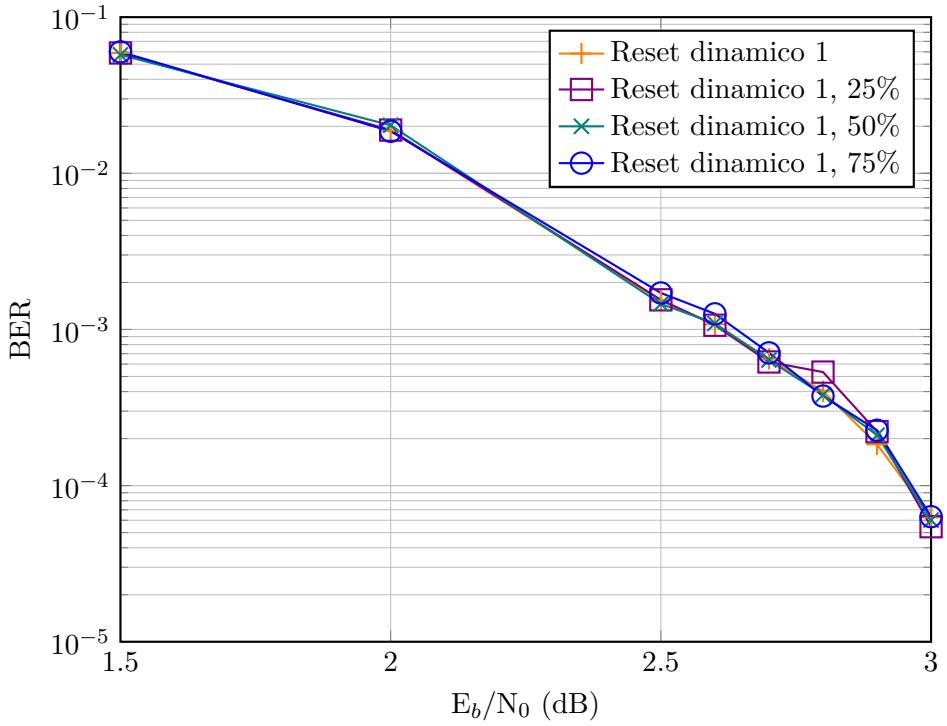


Figura 3.6.: Confronto tra diverse implementazioni del reset 1 applicate al codice 1 riportato nella Sezione 3.1 con $R = 0.6567, L = 400$, $W = 65$, $m_s = 12$

osservazione, inoltre, permette di sottolineare la principale differenza che intercorre tra doping e reset dei blocchi in sola lettura. Come indicato in [21], infatti, il doping è una soluzione che altera il processo di codifica e generazione delle parole di codice, la cui complessità dipende dai codici adoperati. Per tale ragione, inoltre, i simboli su cui effettuare il doping devono essere scelti a priori indipendentemente dall'evoluzione del processo di decodifica. Al contrario, le modalità di reset qui presentate, non necessitano di alcuna variazione nel processo di codifica e, nella versione dinamica, permettono di alterare il normale processo di decodifica solo se necessario. Inoltre, solitamente le tecniche di doping vengono applicate su codici con circolanti e memoria contenuta; al contrario, le modalità di reset qui presentate sono state applicate anche al codice 1, caratterizzato da $m_s = 12$. In [22], poi, viene presentato anche il concetto di *adaptive doping*, il cui funzionamento di base è simile a quello del reset dinamico qui esposto. Tale procedura, applicabile sia ai nodi di controllo che ai nodi variabile, consiste nell'inserire posizioni di doping all'interno del frame in base al valore medio dei LLRs di un certo numero di blocchi decodificati consecutivamente. Nello specifico, viene inviata una richiesta di doping se, dopo aver completato tutte le iterazioni necessarie a decodificare il blocco target

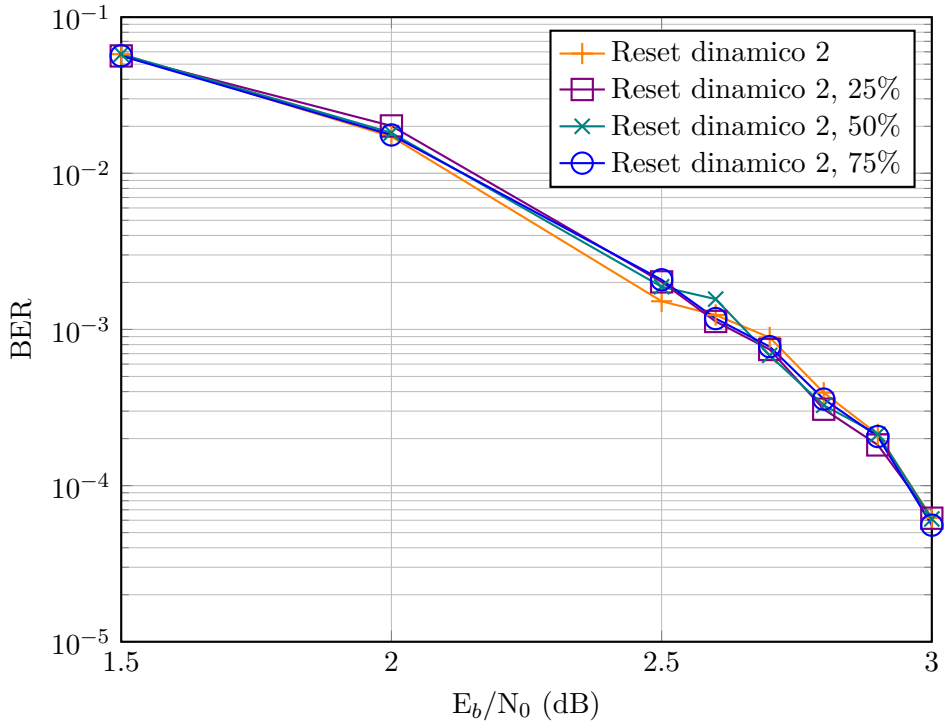


Figura 3.7.: Confronto tra diverse implementazioni del reset 2 applicate al codice 1 riportato nella Sezione 3.1 con $R = 0.6567, L = 400, W = 65, m_s = 12$

all'istante temporale t, la quantità $\mathcal{L}_t$ verifica la condizione

$$\mathcal{L}_t \triangleq \frac{1}{2M} \sum_{i=0}^{2M-1} |\mathrm{LLR}_i^t| \leq \eta \tag{3.10}$$

dove $|\mathrm{LLR}_i^t|$ è il LLR del nodo variabile v_i all'istante t, con $i = 0, 1, \ldots, 2M-1$ e η è una soglia fissata a priori. Se tale condizione è verificata, la decodifica del blocco all'istante t è considerata *fallita* e, al fallimento della decodifica di un certo numero fissato N_r di blocchi consecutivi, la richiesta di doping viene elaborata e i nuovi nodi variabile inclusi nella finestra di decodifica sono considerati dopati.

In termini di rate loss, tuttavia, può essere utile considerare ogni blocco in sola lettura eliminato dall'applicazione del reset 3 come una posizione di VNs doping. A tale proposito, da [22] si riporta l'equazione del rate effettivo di un codice SC-LDPC terminato all'istante temporale $t = L$ e caratterizzato, complessivamente, da d posizioni di VNs doping:

$$R_{\mathrm{VN}} = 1 - \left(\frac{L + m_s}{L - d}\right)(1 - R_\infty)\,. \tag{3.11}$$

A fronte di quanto premesso e dell'Equazione (3.11), dunque, per quantificare la rate loss dovuta all'applicazione del reset 3 è necessario definire il valore di d. Innanzi-

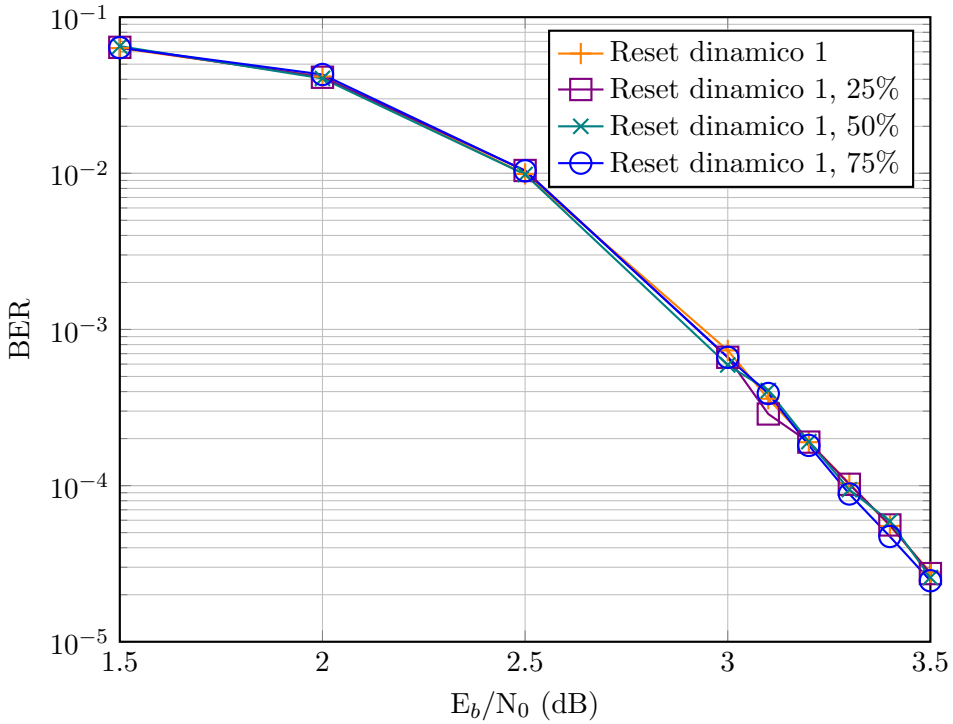


Figura 3.8.: Confronto tra diverse implementazioni del reset 1 applicate al codice 2 riportato nella Sezione 3.1 con $R = 0.7450, L = 50$, $W = 10$, $m_s = 1$

tutto va sottolineato che, applicando il reset in maniera dinamica, tale valore non è definibile a priori poiché non è definibile a priori il numero di reset N_{res} all'interno di un frame. Inoltre, ciò implica che non sono note a priori nemmeno le posizioni della finestra scorrevole p_i^*, $i = 1, 2, \ldots, N_{res}$, in cui si è effettuato un reset. Tale informazione è rilevante poiché, sebbene il numero massimo di blocchi in sola lettura connessi ad una finestra sia pari ad m_s, non vi è garanzia che tra le posizioni di reset p_i^* e p_{i+1}^* la finestra di decodifica sia nuovamente giunta a regime, ovvero non vi è garanzia che alla finestra alla posizione p_{i+1}^* siano connessi m_s blocchi in sola lettura. Pertanto, tenendo conto del fatto che, così come nel doping, i nodi su cui viene eseguito il reset rimangono tali per tutto il processo di decodifica, e in virtù di quanto descritto sopra, il rate effettivo di un codice SC-LDPC terminato all'istante temporale $t = L$ e al quale è stato applicato il reset 3 in maniera dinamica, può essere espresso come

$$R^* = 1 - \left(\frac{L + m_s}{L - \min\{p_1^*, m_s\} - \sum_{i=1}^{N_{res}-1} \min\left\{p_{i+1}^* - p_i^*, m_s\right\}} \right) (1 - R_\infty) . \quad (3.12)$$

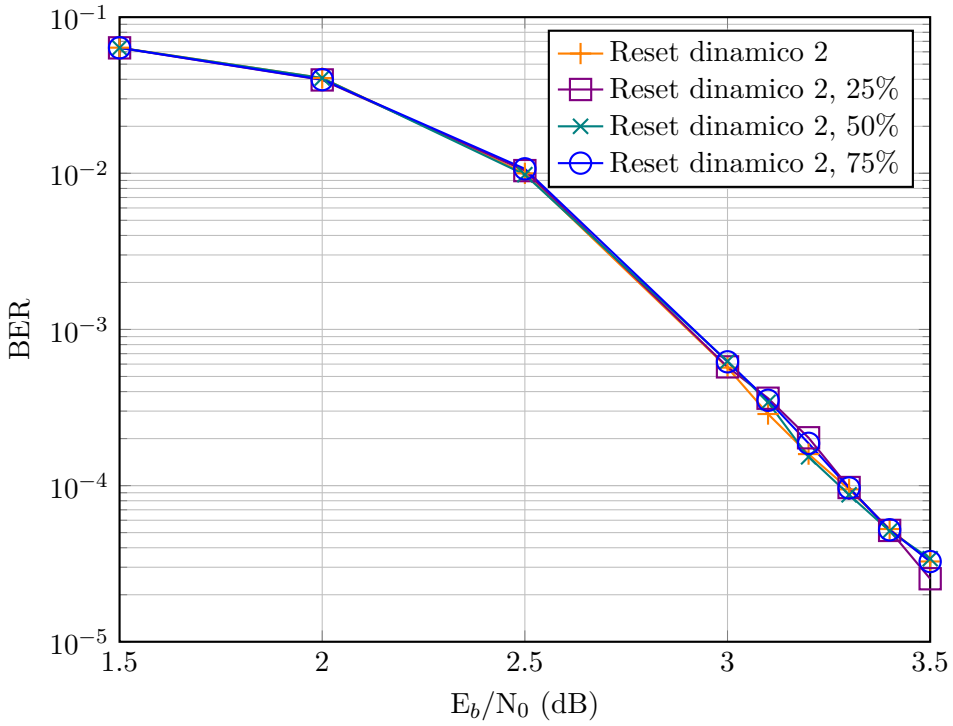


Figura 3.9.: Confronto tra diverse implementazioni del reset 2 applicate al codice 2 riportato nella Sezione 3.1 con $R = 0.7450, L = 50$, $W = 10$, $m_s = 1$

3.4. Studio della propagazione degli errori di decodifica

Come accennato più volte fino a questo punto, uno dei principali problemi del decoder a finestra scorrevole è la sua sensibilità alla propagazione degli errori di decodifica dovuta al concetto stesso di memoria di un codice convoluzionale. In generale, se per un dato scorrimento i simboli errati sono pochi e la maggior parte dei simboli decodificati correttamente ha LLRs molto affidabili, da intendersi come LLRs di segno corretto ed elevato valore assoluto, gli LLRs dei simboli errati avranno un impatto contenuto sulla decodifica dei blocchi successivi. Al contrario, se all'interno di un blocco vi sono molti simboli decodificati erroneamente con LLRs di segno sbagliato ed elevato valore assoluto, e i simboli decodificati correttamente hanno LLRs con valori assoluti bassi, tali messaggi possono influenzare negativamente il processo di decodifica dei blocchi successivi generando errori che altrimenti non si sarebbero verificati. Chiaramente, per funzionamento del decoder a finestra scorrevole, tali simboli errati possono causare una forte propagazione degli errori di decodifica che, in applicazioni per le quali L è elevato, può essere disastrosa. È infatti intuibile che in applicazioni con valori di L bassi vi sono pochi blocchi da decodificare tra una nuova trasmissione e l'altra e pertanto la propagazione di eventuali errori è strutturalmente limitata dalla lunghezza dei frame da trasmettere. Al contrario, in applicazioni con valori di L alti fino addirittura alle applicazioni per lo

streaming, in cui $L \to \infty$, il Block Error Rate (BLER) di un decodificatore a finestra scorrevole tende asintoticamente a 1 ([22]).

Per comprendere meglio la propagazione degli errori di cui sopra, da [22] si riporta un'analisi matematica del decoder a finestra scorrevole trattato come una macchina a stati finiti rappresentata graficamente in Figura 3.10. In tale diagramma si possono distinguere:

- Uno stato di errori casuali (S_0), che modella il normale funzionamento del decodificatore.
- Un certo numero di stati intermedi ($S_1, \ldots, S_{J-1}$), che modellano sequenze finite di blocchi errati.
- Uno stato di errori *burst* (S_J) che modella la possibilità di una illimitata propagazione degli errori.

Infine, viene indicato con q_i il BLER dello stato S_i con $i = 0, 1, \ldots, J$.

Prima di proseguire con l'analisi dettagliata del decoder, è bene sottolineare come sia possibile stimare i parametri del modello J, q_i a partire dai risultati di una singola simulazione seguendo quanto illustrato nell'appendice B di [22]. Innanzitutto, è necessario fissare il valore di SNR e poi simulare la trasmissione di N frame formati da L blocchi. A partire dai risultati ottenuti dalla simulazione complessiva di LN blocchi, poi, è necessario analizzare la distribuzione delle lunghezze dei burst finiti di errori, ovvero la distribuzione delle lunghezze dei burst dopo i quali il sistema ritorna allo stato S_0. Tramite tale analisi, dunque, si ricava il numero totale di burst di lunghezza fissata e finita contenuti negli N frame trasmessi. Successivamente, poi, si analizza la distribuzione delle lunghezze degli *end of frame (EOF) burst*, ovvero dei burst che includono uno o più blocchi errati alla fine di ogni frame. A partire da questi valori va deciso il numero di stati da includere nel modello, ovvero va definito il valore di J. Va sottolineato che i burst sono solitamente brevi poiché i burst più lunghi tendono ad una propagazione illimitata degli errori. Per questo motivo, al fine di limitare la complessità del modello, J va scelto grande appena da includere le lunghezze dei burst più frequenti, combinando i rari burst di lunghezza finita e superiore nei burst EOF. Fissato il valore di J è poi immediato ricavare i parametri q_i. Si indica, quindi, con λ_i il numero dei burst di lunghezza $i = 1, 2, \ldots, J$, con $\lambda_J = \lambda_{\mathrm{FL}} + \lambda_{\mathrm{EOF}}$, in cui con λ_{FL} si indica il numero dei burst di lunghezza finita maggiore o uguale a J, e con λ_{EOF} si indica il numero dei burst EOF. Inoltre, si indica con $\delta_J = \delta_{\mathrm{FL}} + \delta_{\mathrm{EOF}}$ il numero totale di blocchi errati nei burst inclusi in λ_J. Fissata la notazione, e definita come unità temporale il tempo di permanenza del sistema in un determinato stato intermedio, il numero totale di unità temporali T_i passati nello stato S_i è dato da

$$T_i = \sum_{j=i}^{J} \lambda_j, \quad i = 1, 2, \ldots, J-1, \tag{3.13}$$

mentre il numero totale di blocchi errati E_i nello stato intermedio S_i è dato da

$$E_i = \sum_{j=i+1}^{J} \lambda_j, \quad i = 1, 2, \ldots, J-1, \tag{3.14}$$

da cui segue che

$$q_i = \frac{E_i}{T_i}, \quad i = 1, 2, \ldots, J-1. \tag{3.15}$$

Per quanto concerne q_0, poi, si ha che il numero totale di unità temporali T_0 passate nello stato S_0 equivale al numero totale di blocchi decodificati correttamente, che è dato da

$$T_0 = LN - \sum_{j=1}^{J-1} j\lambda_j - \delta_J, \tag{3.16}$$

e che il numero totale di blocchi errati E_0 nello stato S_0 è dato da

$$E_0 = \sum_{j=1}^{J} \lambda_j = T_1, \tag{3.17}$$

da cui si ha quindi

$$q_0 = \frac{E_0}{T_0}. \tag{3.18}$$

Infine, per determinare q_J bisogna considerare che il numero di unità temporali T_J passati nello stato S_J è dato da δ_J meno il numero di blocchi sbagliati prima di raggiungere S_J. Ovvero, vale la relazione

$$T_J = \delta_J - (J-1)\lambda_J. \tag{3.19}$$

Tenendo conto del fatto che, una volta raggiunto, il decoder rimane nello stato S_J dopo ogni successivo errore di decodifica, e che ritorna nello stato S_0 dopo un burst di lunghezza J o superiore o dopo un burst EOF, il numero totale di blocchi errati E_J nello stato S_J è dato da

$$E_J = T_J - \lambda_J, \tag{3.20}$$

da cui si può infine ricavare

$$q_J = \frac{E_J}{T_J}. \tag{3.21}$$

Come rappresentato in Figura 3.10, il decodificatore inizia nello stato di errori casuali S_0 e passa allo stato intermedio S_1 con probabilità q_0 al verificarsi del primo blocco errato. Successivamente, o si verifica un errore nel secondo blocco e il decodificatore passa allo stato intermedio S_2 con probabilità q_1, o il blocco in questione viene decodificato correttamente e si torna allo stato S_0 con probabilità $1-q_1$, avendo quindi un singolo blocco errato. In virtù di quanto detto riguardo la propagazione degli errori, è lecito affermare che generalmente $q_1 \geq q_0$, poiché una decodifica errata di un blocco implica necessariamente che all'interno di esso vi siano dei simboli

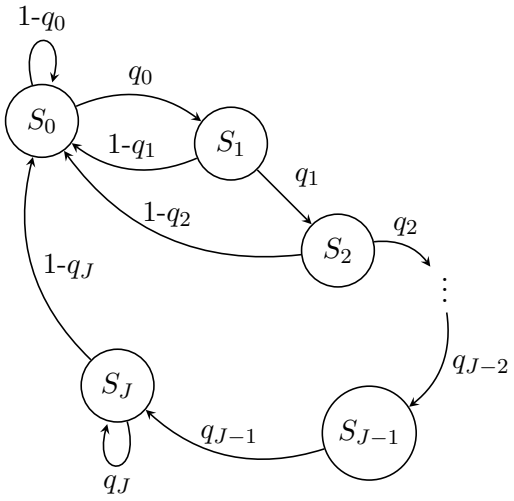


Figura 3.10.: Diagramma di stato di un generico decoder a finestra scorrevole

errati, la cui connessione con la finestra successiva potrebbe comprometterne la decodifica. In maniera analoga a quanto visto prima, a partire dallo stato intermedio S_2, il decodificatore passa allo stato intermedio S_3 con probabilità q_2 (ovvero al verificarsi di un errore di decodifica all'interno del terzo blocco), situazione in cui la finestra di decodifica contiene simboli decodificati erroneamente provenienti dai due blocchi precedenti. Ciò implica che due dei blocchi in sola lettura coinvolti nella decodifica nello stato intermedio S_2 contengono, e quindi propagano, LLRs relativi a decodifiche erronee che potrebbero compromettere il processo di decodifica del terzo blocco. Dunque, estendendo quanto detto prima, si ha $q_2 \geq q_1 \geq q_0$ poiché la configurazione del decodificatore allo stato intermedio S_2 parte da informazioni meno affidabili rispetto alla configurazione allo stato intermedio S_1. Pertanto il decodificatore ritorna allo stato S_0 con probabilità $1 - q_2$ e un burst di errori di due blocchi.

Generalizzando la trattazione, si ha che $q_{J-1} \geq q_{J-2} \geq \cdots \geq q_2 \geq q_1 \geq q_0$, e che la corretta decodifica di un blocco nello stato intermedio S_i riporta il decodficatore allo stato S_0 con un burst di errori di i blocchi, con $i = 1, 2, \ldots, J-1$. Decodificando erroneamente J blocchi consecutivi poi, il decodificatore entra nello stato di erroi burst S_J, nel quale rimane con probabilità $q_J \geq q_{J-1}$ fintantoché gli errori di decodifica continuano, e dal quale esce con probabilità $1 - q_J$ con un burst di errori di J o più blocchi. Considerato dunque che la decodifica di un blocco richiede l'inclusione all'interno della finestra scorrevole dei m_s blocchi ad esso precedenti, si ha che, se l'influenza dei blocchi con errori di decodifica è forte abbastanza, ovvero è tale per cui $q_J \to 1$, il decodificatore non è in grado di uscire dallo stato di errori burst e si ha quindi una illimitata propagazione degli errori.

I parametri del canale $\mathrm{E}_b/\mathrm{N}_0$, del decodificatore W, e della memoria del codice m_s condizionano i valori di $q_0, q_1, \ldots, q_J$. Tali probabilità, generalmente, sono fun-

zioni decrescenti di tutti e tre i parametri. Tuttavia, all'aumentare di m_s, i blocchi decodificati precedentemente hanno un'influenza più forte sui blocchi da decodificare. Per tale ragione, codici e decodificatori robusti (con elevati valori di m_s e W) hanno valori di q_0 minori e sono quindi meno inclini a raggiungere lo stato di errori burst S_J ma, una volta raggiunto tale stato, hanno elevate probabilità di rimanerci e quindi hanno elevate probabilità di incorrere in una illimitata propagazione degli errori. Tipicamente, quindi, tali configurazioni soffrono di pochi errori di tipo burst brevi e le loro performance in termini di BLER sono dominate dalla propagazione degli errori. Al contrario, configurazioni di codici e decodificatori più deboli (con valori di m_s e W relativamente bassi) hanno valori di q_0 elevati e quindi raggiungono lo stato di errori burst S_J più frequentemente, ma sono meno inclini a soffrire di propagazione illimitata degli errori. Per tale ragione, le loro performance in termini di BLER sono dominate da un numero maggiore di errori di tipo burst di lunghezza variabile.

Come fatto in [22], vengono ora ricavate le espressioni del BLER in funzione di $q_0, q_1, \ldots, q_J$ per trasmissioni prive di terminazione, ovvero tali per cui $L \to \infty$.

Caso 1. Nessuno stato intermedio ($J = 1$), $L \to \infty$.

Indicando con p_i la probabilità di essere nello stato S_i, $i = 0, 1, \ldots, J$, si ha che $p_0 = p_0(1 - q_0) + p_1(1 - q_1)$, da cui si ricava

$$p_1 = \frac{p_0 - p_0(1 - q_0)}{1 - q_1} = \frac{p_0 q_0}{1 - q_1}. \tag{3.22}$$

È quindi possibile esprimere il BLER *medio* come

$$\begin{aligned} P_{\mathrm{BL}}^{(\infty)} &= p_0 q_0 + p_1 q_1 = p_0 q_0 + p_0 q_1 \left(\frac{q_0}{1 - q_1} \right) \\ &= p_0 \left(\frac{q_0}{1 - q_1} \right). \end{aligned} \tag{3.23}$$

Poiché $p_0 + p_1 = p_0 + p_0 \left(\frac{q_0}{1-q_1} \right) = p_0 \left(\frac{1-q_1+q_0}{1-q_1} \right) = 1$, si ha che

$$p_0 = \frac{1 - q_1}{1 - q_1 + q_0} \tag{3.24}$$

e quindi che

$$P_{\mathrm{BL}}^{(\infty)} = \frac{q_0}{1 - q_1 + q_0} = \frac{r_1}{1 - q_1 + r_1} \tag{3.25}$$

con $q_0 \triangleq r_1$. Nota che quando $q_1 \to 1$, $\lim_{q_1 \to 1} P_{BL}^{(\infty)} = 1$, che corrisponde all'illimitata propagazione degli errori.

Caso 2. Uno stato intermedio ($J = 2$), $L \to \infty$.

In questo caso si ha

$$\begin{cases} p_0 = p_0(1-q_0) + p_1(1-q_1) + p_2(1-q_2) \\ p_1 = p_0 q_0 \\ p_2 = p_1 q_1 + p_2 q_2 \end{cases} \tag{3.26}$$

da cui p_2 può essere scritto anche come $p_2 = p_1 \frac{q_1}{1-q_2} = p_0 \frac{q_0 q_1}{1-q_2}$. Poiché $p_0 + p_1 + p_2 = p_0 + p_0 q_0 + p_0 \frac{q_0 q_1}{1-q_2} = 1$, si ha che

$$p_0 = \frac{1}{1 + q_0 + \dfrac{q_0 q_1}{1-q_2}} = \frac{1-q_2}{1-q_2+q_0-q_0 q_2+q_0 q_1} \tag{3.27}$$

a partire da cui è possibile esprimere il BLER medio come

$$\begin{aligned} P_{\mathrm{BL}}^{(\infty)} &= p_0 q_0 + p_1 q_1 + p_2 q_2 \\ &= \frac{q_0 - q_0 q_2 + q_0 q_1}{1-q_2+q_0-q_0 q_2+q_0 q_1} = \frac{r_2}{1-q_2+r_2} \end{aligned} \tag{3.28}$$

con $r_2 \triangleq q_0(1-q_2+q_1)$. Anche in questo caso, per $q_2 \to 1$ si ha $\lim_{q_2 \to 1} P_{BL}^{(\infty)} = 1$, ovvero si ha una propagazione degli errori asintoticamente illimitata.

Caso 3. Due o più stadi intermedi ($J \geq 3$), $L \to \infty$

Considerando inizialmente il caso $J = 3$, si ha

$$\begin{cases} p_0 = p_0(1-q_0) + p_1(1-q_1) + p_2(1-q_2) + p_3(1-q_3) \\ p_1 = p_0 q_0 \\ p_2 = p_1 q_1 \\ p_3 = p_2 q_2 + p_3 q_3 = p_1 q_1 q_2 + p_3 q_3 = p_0 q_0 q_1 q_2 + p_3 q_3 \end{cases} \tag{3.29}$$

da cui segue che $p_3 = p_0 \frac{q_0 q_1 q_2}{1-q_3}$. Poiché $p_0 + p_1 + p_2 + p_3 = 1$, si ha che

$$\begin{aligned} p_0 &= \frac{1}{1 + q_0 + q_0 q_1 + \dfrac{q_0 q_1 q_2}{1-q_3}} \\ &= \frac{1-q_3}{1-q_3+q_0-q_0 q_3+q_0 q_1-q_0 q_1 q_3+q_0 q_1 q_2}. \end{aligned} \tag{3.30}$$

Definendo $r_3 \triangleq q_0(1-q_3+q_1-q_1 q_3+q_1 q_2)$, si ha che $p_0 = \frac{1-q_3}{1-q_3 r_3}$, e quindi che il

BLER medio può essere espresso come

$$\begin{aligned} P_{\mathrm{BL}}^{(\infty)} &= p_0 q_0 + p_1 q_1 + p_2 q_2 + p_3 q_3 \\ &= p_0 q_0 + p_0 q_0 q_1 + p_0 q_0 q_1 q_2 + p_0 \frac{q_0 q_1 q_2 q_3}{1 - q_3} \\ &= p_0 \frac{r_3}{1 - q_3} = \frac{r_3}{1 - q_3 + r_3} \end{aligned} \tag{3.31}$$

tale per cui $\lim_{q_3 \to 1} P_{\mathrm{BL}}^{(\infty)} = 1$.

Per $J > 3$, posto

$$r_J \triangleq q_0 \left\{1 - q_J + q_1 \left\{1 - q_J + q_2 \left[1 - q_J + \cdots + q_{J-2}(1 - q_J + q_{J-1})\right]\right\}\right\} \tag{3.32}$$

l'espressione generale per il BLER medio è data da

$$P_{\mathrm{BL}}^{(\infty)} = \frac{r_J}{1 - q_J + r_J} \tag{3.33}$$

tale per cui $\lim_{q_J \to 1} P_{\mathrm{BL}}^{(\infty)} = 1$.

A riprova di quanto discusso, sfruttando la funzione A.3, si è analizzata in maniera quantitativa l'evoluzione dei LLRs totali al variare del numero di iterazioni di SPA per entrambi i codici riportati nella Sezione 3.1, ponendo particolare attenzione alle posizioni della finestra scorrevole p_{err} in cui vi sono stati errori di decodifica e ottenendo dunque i risultati riassunti nella Tabella 3.1.

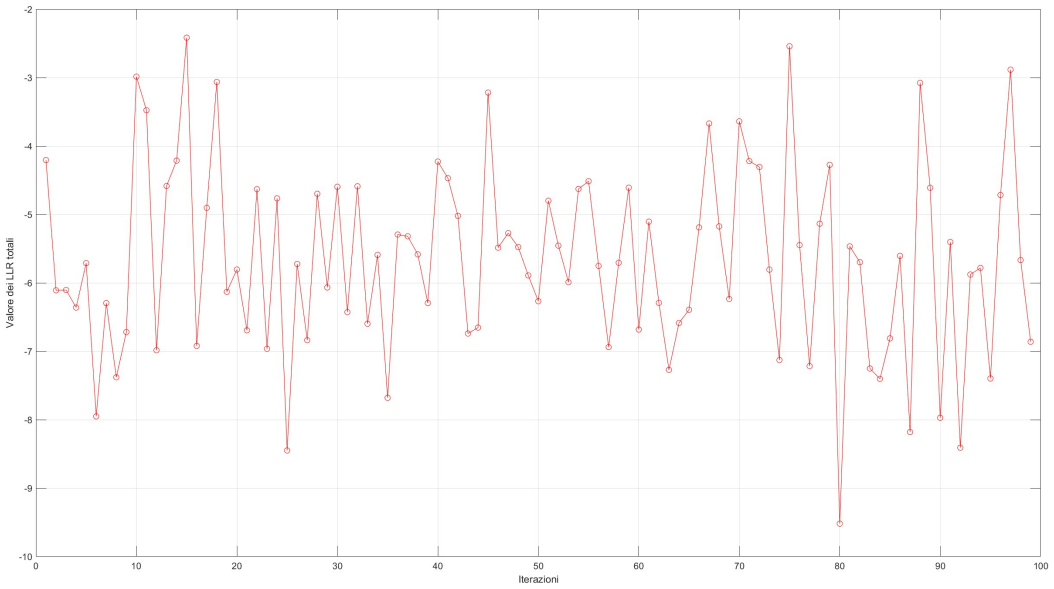
Parametri	Codice 1	Codice 2
m_s	12	1
W	65	10
R	0.7450	0.6567
p_{err}	60, 62, 63, 65, 66, 67, 68, 69	1, 2, 3, 4, 5, 6, 7, 17, 18, 19, 20, 21, 22, 23, 24, 25

Tabella 3.1.: Propagazione degli errori al variare del codice e della configurazione del decodificatore

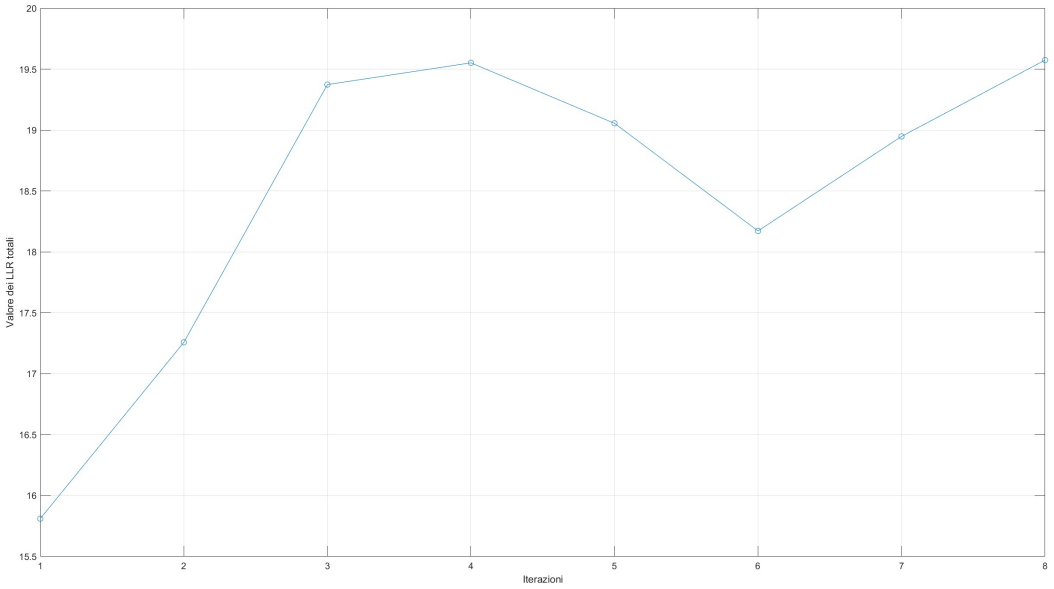
Come atteso, si è osservato che nel codice 1 gli errori di decodifica si presentano sotto forma di burst di lunghezza contenuta mentre, nel codice 2, caratterizzato da valori di m_s e W inferiori a quelli del primo, si hanno errori burst più lunghi che tuttavia non degenerano in una propagazione illimitata dell'errore.

Infatti, come si nota dalla Tabella 3.1, con un codice robusto si hanno burst di 2 ($p_{err} = 62, 63$) o 5 ($p_{err} = 65, 66, \ldots, 69$) blocchi errati mentre, con una configurazione più debole, si hanno burst di 7 ($p_{err} = 1, 2, \ldots, 7$) o addirittura 9 ($p_{err} = 17, 18, \ldots, 25$) blocchi errati, dopo i quali il decodificatore ritorna comunque allo stato S_0. Infine, nelle Figure 3.11 e 3.12 sono riportati gli andamenti di alcuni

dei LLRs di entrambi i codici e, come si nota, in presenza di simboli errati il SPA raggiunge sempre il numero massimo di iterazioni, che conferma l'efficacia del reset dinamico illustrato nella sezione 3.3, basato proprio su tale parametro.

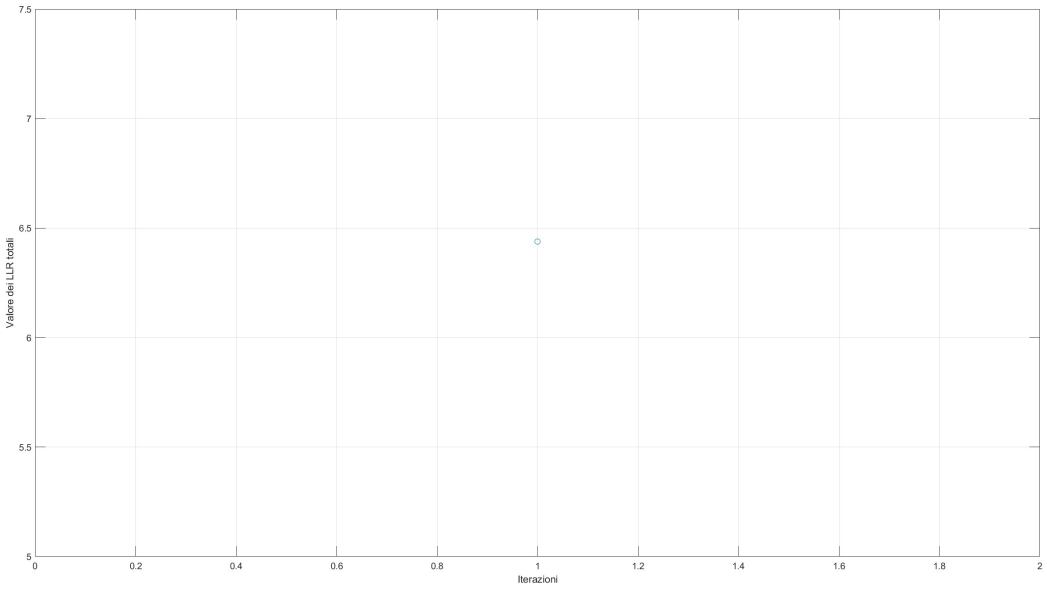


(a) LLR totale relativo al VN 12 decodificato erroneamente a $p = 69$



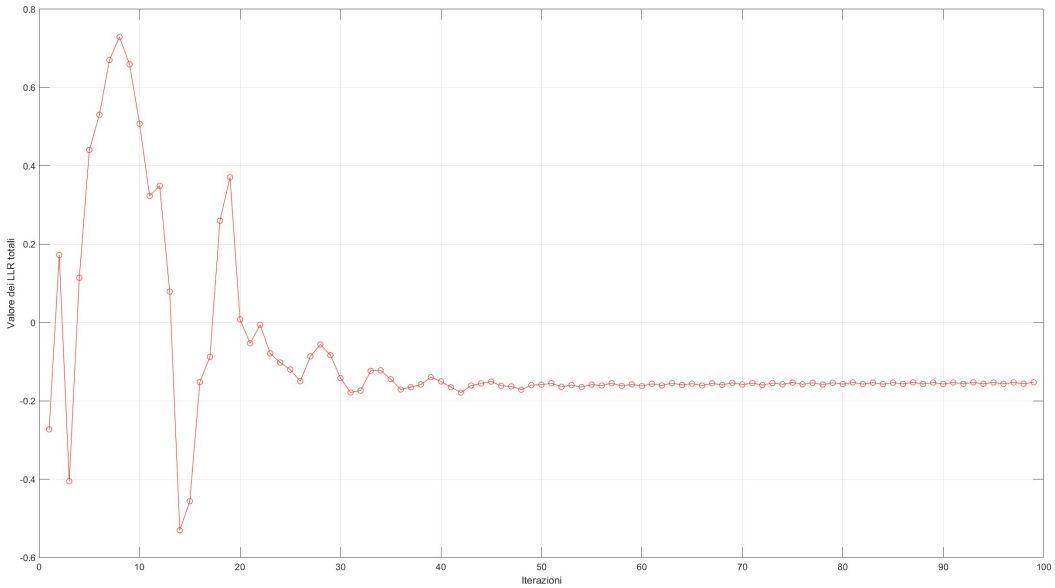
(b) LLR totale relativo al VN 12 decodificato correttamente a $p = 70$

Figura 3.11.



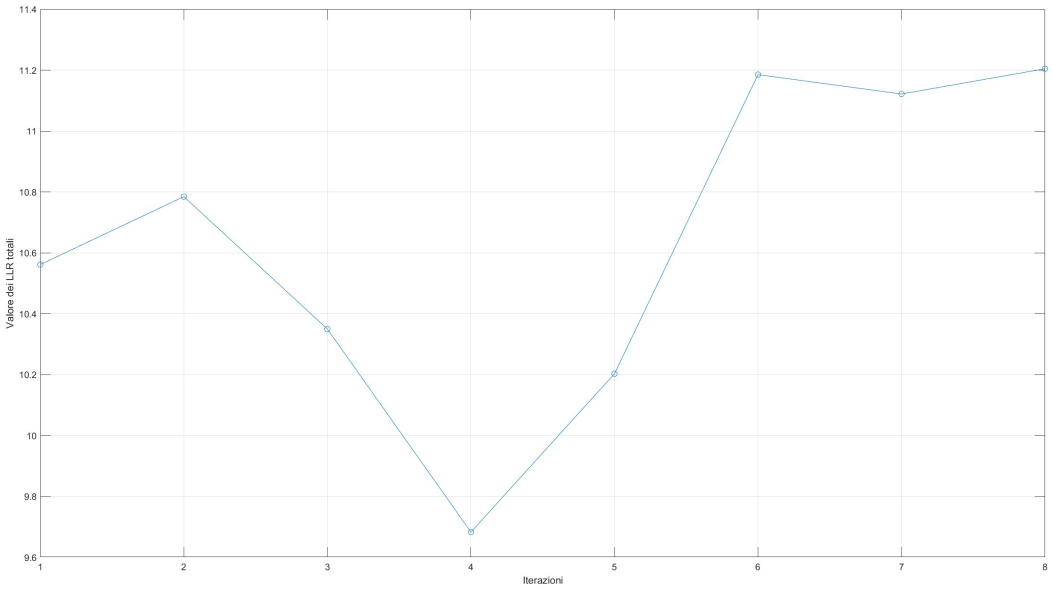
(c) LLR totale relativo al VN 12 decodificato correttamente a $p = 71$

Figura 3.11.: Andamento dei LLRs del codice 1 riportato nella Sezione 3.1

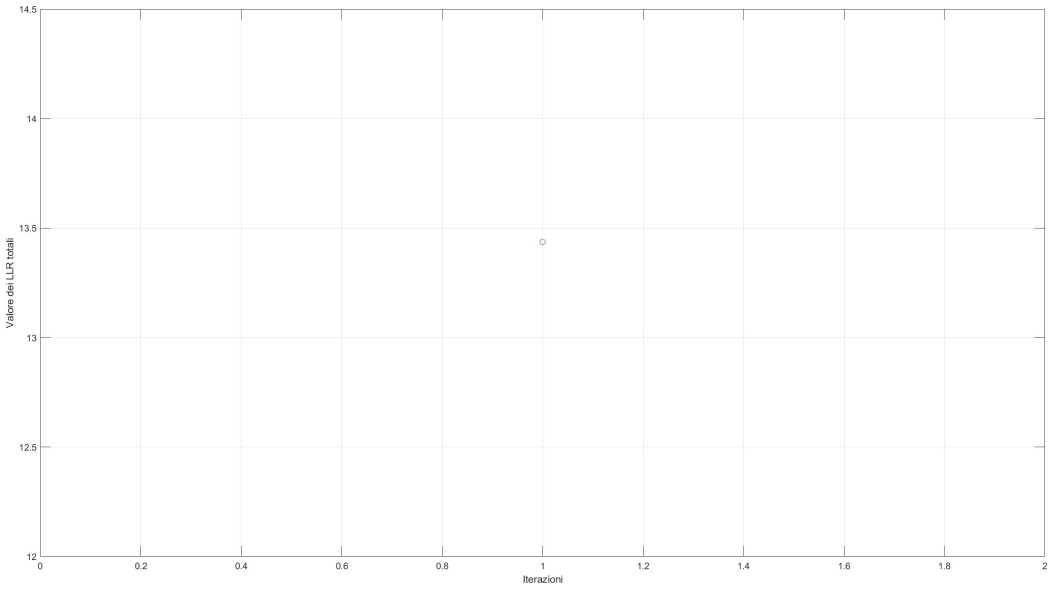


(a) LLR totale relativo al VN 12 decodificato erroneamente a $p = 25$

Figura 3.12.



(b) LLR totale relativo al VN 12 decodificato correttamente a $p = 26$



(c) LLR totale relativo al VN 12 decodificato correttamente a $p = 27$

Figura 3.12.: Andamento dei LLRs del codice 2 riportato nella Sezione 3.1

Capitolo 4.

Conclusioni

Sono state analizzate le caratteristiche dei principali algoritmi iterativi per la decodifica di codici LDPC in generale e SC-LDPC in particolare. Nello specifico, sono state proposte modifiche al decodificatore a finestra scorrevole convenzionale, che hanno portato ad un incremento delle prestazioni in termini di BER e che non richiedono alterazioni nel processo di codifica dell'informazione da trasmettere. Inoltre, il concetto di reset dinamico può essere esteso e approfondito in sviluppi futuri identificando, ad esempio, un criterio di applicazione diverso da quello relativo al numero di iterazioni di SPA o alterando in maniera diversa i LLRs in fase di reset.

A tale proposito, l'analisi quantitativa dell'evoluzione dei LLRs durante il processo di decodifica potrebbe essere un buon punto di partenza per l'elaborazione di ulteriori modifiche al decodificatore a finestra scorrevole atte a migliorarne le prestazioni pur mantenendo limitata la complessità del processo di decodifica.

Inoltre, sebbene tali modifiche e, più in generale, tale decodificatore, siano riferiti unicamente ai codici SC-LDPC, in possibili applicazioni future si potrebbero sfruttate tali algoritmi a complessità ridotta anche per la decodifica di codici LDPC a blocco caratterizzati da valori di n particolarmente elevati. Una soluzione di questo tipo permetterebbe di sfruttare i vantaggi dell'algoritmo a complessità ridotta descritto nel presente lavoro di tesi, garantendo però ottime prestazioni dovute, appunto, alla possibilità di usare codici a blocco con maggiori valori di n e, di conseguenza, migliori capacità correttive.

Come sottolineato nella Sezione 1.3.2 del Capitolo 1, in questo lavoro si è posta l'attenzione sui codici SC-LDPC tempo invarianti, pertanto applicando quanto mostrato a codici SC-LDPC *tempo varianti* si potrebbero elaborare ulteriori modifiche.

Dalle Figure 3.11 e 3.12 si notano immediatamente caratteristiche comuni ai LLRs relativi ai simboli decodificati erroneamente, come, innanzitutto, il fatto che essi presentino oscillazioni diverse da quelle dei LLRs relativi ai simboli decodificati correttamente. In generale, quindi, si potrebbero adoperare tecniche di machine learning al fine di riconoscere con maggiore precisione i LLRs errati e, di conseguenza, applicare il reset dinamico in maniera più oculata. In tal senso, come riportato, ad esempio, in [23], il machine learning è già stato incluso in algoritmi adibiti alla

decodifica di codici LDPC con l'obiettivo di ottimizzare i LLRs che vengono di volta in volta passati tra i nodi.

Infine, come già accennato nella sezione 3.3, oltre che sulla decodifica, ci si potrebbe concentrare sulla codifica efficiente dei codici trattati nel presente lavoro, prediligendo, in virtù di quanto riportato, codici con valori elevati di m_s al fine di favorire tecniche di miglioramento della decodifica a finestra scorrevole. Tale soluzione, tuttavia, richiederebbe l'impiego di valori elevati di L per limitare la rate loss dovuta alla terminazione dei codici. Per questa ragione, la progettazione di codici efficienti in tal senso, richiederebbe anche particolare attenzione a parametri quali latenza e complessià di decodifica.

Appendice A.

Implementazioni MATLAB

A.1. Algoritmi di decodifica

A.1.1. Sum-Product Algorithm

```
function [estimatedCodeword, llrTotal, current] = ...
SumProductAlgorithm(H, LLR, iterations, modulationIndicator,
    clipValue)

estimatedCodeword = zeros(1, size(H,2));
llrTotal = zeros(1, size(H,2));
rows = size(H, 1);
columns = size(H, 2);
temp = ones(rows, columns);
idx = find(H);
% inizializzazione dei nodi variabile
variableMatrix = H.*LLR;

current = 1;
while(current < iterations)
    divisor = tanh(variableMatrix*(1/2));
    temp(idx) = divisor(idx);
    % aggiornamento degli elementi di checkMatrix
    checkMatrix = 2*atanh(H.*((prod(temp, 2)./temp)));
    checkMatrix = clip(checkMatrix, -clipValue, clipValue);

    % aggiornamento dei nodi variabile
    variableMatrix = H.*(LLR+(sum(checkMatrix)-checkMatrix))
       ;
    variableMatrix = clip(variableMatrix, -clipValue,
       clipValue);

    % calcolo LLR Totali
```

```
26      llrTotal = clip(LLR + sum(checkMatrix), -clipValue,
           clipValue);
27
28      estimatedCodeword = double((sign(modulationIndicator)*
           llrTotal)<=0);
29      if (isequal(mod(estimatedCodeword*transpose(H),2), zeros
           (1, rows)))
30          return;
31      else
32          current = current+1;
33      end
34  end
```

A.1.2. Min-Sum Algorithm

```
1   function [estimatedCodeword, llrTotal] = MinSumAlgorithm(H,
       LLR, iterations, modulationIndicator)
2
3   estimatedCodeword = zeros(1, size(H,2));
4   llrTotal = zeros(1, size(H,2));
5   rows = size(H, 1);
6   columns = size(H, 2);
7
8   messages_idx = find(H);
9   left_messages = zeros(length(messages_idx),1);
10  % inizializzazione dei nodi variabile
11  variableMatrix = H.*LLR;
12  checkMatrix = H;
13  current = 1;
14  while(current <= iterations)
15      % aggiornamento dei nodi di controllo
16      right_messages = variableMatrix(messages_idx);
17      for t=1:length(right_messages)
18          alfa = sign(right_messages);
19          alfa(t) = 1;
20          beta = abs(right_messages);
21          beta(t) = inf;
22          left_messages(t) = prod(alfa)*min(beta);
23      end
24      checkMatrix(messages_idx) = left_messages;
25      % aggiornamento dei nodi variabile
```

```
26      variableMatrix = H.*(LLR+(sum(checkMatrix)-checkMatrix))
            ;
27
28      % calcolo LLR Totale
29      llrTotal = LLR + sum(checkMatrix);
30
31      estimatedCodeword = double((sign(modulationIndicator)*
            llrTotal)<=0);
32      if (isequal(mod(estimatedCodeword*transpose(H),2), zeros
            (columns,1)))
33          return;
34      else
35          current = current+1;
36      end
37  end
```

A.2. SPA a finestra scorrevole

A.2.1. Terminazione convenzionale

```
1   function [estimatedCodeword, resets, llr_evolution] =
        SpaSlidingWindow(spaIterations, Hconv, ms, a, c,
        channelLLR, numberOfBlocks, modulationIndex, clipValue,
        resetMode, resetFactor, varargin)
2
3   % gestione parametri per osservazione LLRs
4   if (nargin < 13)
5       target_window = inf;
6       target_variable_nodes = [1, 1];
7   else
8       target_window = varargin{1};
9       target_variable_nodes = varargin{2};
10      if (length(target_variable_nodes) < 2)
11          target_variable_nodes = [1, target_variable_nodes];
12      end
13  end
14
15  % gestione parametri per reset parziale dei simboli target
16  if (size(resetMode,2) == 2)
17      targetFraction = resetMode{2};
18      resetMode = resetMode{1};
19  else
```

```
20      targetFraction = 1;
21  end
22
23  estimatedCodeword = zeros(1, size(Hconv, 2));
24  % calcolo dei parametri della finestra
25  Wb = numberOfBlocks*a;
26  windowHeight = numberOfBlocks*c;
27  numberOfWindows = size(Hconv, 2)/a;
28  currentColumn = 1;
29  currentRow = 1;
30  variableMatrix = 0;
31  resets = 0;
32  isLast = false;
33  resetFlag = false;
34  readOnlyBlocks = 0;
35
36  for i=1:numberOfWindows
37      % creazione della finestra
38      bottomBoundary = min(currentRow+windowHeight-1, size(
            Hconv, 1));
39      rightBoundary = min(currentColumn+Wb-1, size(Hconv, 2));
40      if contains(resetMode, "fixed")
41          resetFlag = mod(i, resetFactor) == 0;
42      end
43      Hspa = Hconv(currentRow:bottomBoundary, currentColumn:
            rightBoundary);
44      [windowCodeword, ~, variableMatrix, ~, llr_evolution] =
            ...
45      ImprovedSumProductAlgorithm(Hspa, channelLLR(
            currentColumn:rightBoundary), ...
46      variableMatrix, readOnlyBlocks, ms, a, c, spaIterations,
            modulationIndex, isLast, clipValue, resetFlag,
            target_variable_nodes);
47      if contains(resetMode, "dynamic")
48          resetFlag = spaCurrentIteration >= resetFactor;
49      end
50      switch true
51          case contains(resetMode, "zero")
52              if (resetFlag)
53                  if (targetFraction >= 1)
54                      variableMatrix(c+1:end, readOnlyBlocks*a
                            +1:readOnlyBlocks*a+a) = 0;
```

```
55                  else
56                      % passa da matrice dei nodi variabile a
                            vettore colonna
57                      targetVector = reshape(variableMatrix(c
                            +1:end, readOnlyBlocks*a+1:
                            readOnlyBlocks*a+a), 1, []);
58                      % calcolo dei nodi da resettare
59                      targetNodes = round(targetFraction*nnz(
                            targetVector));
60                      zero_idx = find(targetVector == 0);
61                      % elimina gli 0 per non falsare la
                            ricerca dei minimi
62                      % LLRs
63                      targetVector(zero_idx) = inf;
64                      % cerca gli indici corrispondenti ai
                            LLRs col modulo
65                      % minore
66                      [~, reset_idx] = mink(targetVector,
                            targetNodes, 'ComparisonMethod','abs'
                            );
67                      temp = variableMatrix(c+1:end,
                            readOnlyBlocks*a+1:readOnlyBlocks*a+a
                            );
68                      % reset
69                      temp(reset_idx) = 0;
70                      variableMatrix(c+1:end, readOnlyBlocks*a
                            +1:readOnlyBlocks*a+a) = temp;
71                  end
72              end
73          case contains(resetMode, "channel")
74              if (resetFlag)
75                  if (targetFraction >= 1)
76                      variableMatrix(c+1:end, readOnlyBlocks*a
                            +1:readOnlyBlocks*a+a) = Hspa(c+1:end
                            , 1:a).*channelLLR(currentColumn:
                            currentColumn+a-1);
77                  else
78                      targetVector = reshape(variableMatrix(c
                            +1:end, readOnlyBlocks*a+1:
                            readOnlyBlocks*a+a), 1, []);
79                      targetNodes = round(targetFraction*nnz(
                            targetVector));
```

```
80                      zero_idx = find(targetVector == 0);
81                      targetVector(zero_idx) = inf;
82                      [~, reset_idx] = mink(targetVector,
                           targetNodes, 'ComparisonMethod', 'abs
                           ');
83                      temp = Hspa(c+1:end, 1:a).*channelLLR(
                           currentColumn:currentColumn+a-1);
84                      temp = reshape(temp, 1, []);
85                      targetVector(reset_idx) = temp(reset_idx
                           );
86                      targetVector(zero_idx) = 0;
87                      originalSize = size(variableMatrix(c+1:
                           end, readOnlyBlocks*a+1:
                           readOnlyBlocks*a+a));
88                      variableMatrix(c+1:end, readOnlyBlocks*a
                           +1:readOnlyBlocks*a+a) = reshape(
                           targetVector, originalSize);
89                  end
90              end
91          case contains(resetMode, "readOnly")
92              if (resetFlag)
93                  readOnlyBlocks = 0;
94              end
95      end
96      resets = resets + resetFlag;
97      % e selezione dei primi a elementi
98      estimatedCodeword(1, currentColumn:(currentColumn+a-1))
           = windowCodeword(1, 1:a);
99      currentRow = currentRow + c;
100     currentColumn = currentColumn + a;
101     if (readOnlyBlocks < ms)
102         readOnlyBlocks = readOnlyBlocks + 1;
103     end
104     if (i==target_window)
105         return
106     end
107 end
```

A.2.2. Terminazione anticipata

```
1 function [estimatedCodeword, resets, llr_evolution] =
     ImprovedSpaSlidingWindow(spaIterations, Hconv, ms, a, c,
```

```
    channelLLR, numberOfBlocks, modulationIndex, clipValue,
    resetMode, resetFactor, varargin)

% gestione parametri per osservazione LLRs
if (nargin < 13)
    target_window = inf;
    target_variable_nodes = [1, 1];
else
    target_window = varargin{1};
    target_variable_nodes = varargin{2};
    if (length(target_variable_nodes) < 2)
        target_variable_nodes = [1, target_variable_nodes];
    end
end

% gestione parametri per reset parziale dei simboli target
if (size(resetMode,2) == 2)
    targetFraction = resetMode{2};
    resetMode = resetMode{1};
else
    targetFraction = 1;
end

estimatedCodeword = zeros(1, size(Hconv, 2));
% calcolo dei parametri della finestra
Wb = numberOfBlocks*a;
windowHeight = numberOfBlocks*c;
numberOfWindows = floor((size(Hconv,2)-Wb)/a)+1;
currentColumn = 1;
currentRow = 1;
variableMatrix = 0;
resets = 0;
isLast = false;
resetFlag = false;
readOnlyBlocks = 0;

for i=1:numberOfWindows-1
    % creazione della finestra
    if contains(resetMode, "fixed")
        resetFlag = mod(i, resetFactor) == 0;
    end
    Hspa = Hconv(currentRow:(currentRow+windowHeight-1),
```

```
        currentColumn:(currentColumn+Wb-1));
42      [windowCodeword, ~, variableMatrix, spaCurrentIteration,
            llr_evolution] = ...
43      ImprovedSumProductAlgorithm(Hspa, channelLLR(
            currentColumn:(currentColumn+Wb-1)), ...
44      variableMatrix, readOnlyBlocks, ms, a, c, spaIterations,
            modulationIndex, isLast, clipValue, resetFlag,
            target_variable_nodes);
45      if contains(resetMode, "dynamic")
46          resetFlag = spaCurrentIteration >= resetFactor;
47      end
48      switch true
49          case contains(resetMode, "zero")
50              if (resetFlag)
51                  if (targetFraction >= 1)
52                      variableMatrix(c+1:end, readOnlyBlocks*a
                            +1:readOnlyBlocks*a+a) = 0;
53                  else
54                      % passa da matrice dei nodi variabile a
                            vettore colonna
55                      targetVector = reshape(variableMatrix(c
                            +1:end, readOnlyBlocks*a+1:
                            readOnlyBlocks*a+a), 1, []);
56                      % calcolo dei nodi da resettare
57                      targetNodes = round(targetFraction*nnz(
                            targetVector));
58                      zero_idx = find(targetVector == 0);
59                      % elimina gli 0 per non falsare la
                            ricerca dei minimi
60                      % LLRs
61                      targetVector(zero_idx) = inf;
62                      % cerca gli indici corrispondenti ai
                            LLRs col modulo
63                      % minore
64                      [~, reset_idx] = mink(targetVector,
                            targetNodes, 'ComparisonMethod','abs'
                            );
65                      temp = variableMatrix(c+1:end,
                            readOnlyBlocks*a+1:readOnlyBlocks*a+a
                            );
66                      % reset
67                      temp(reset_idx) = 0;
```

```
68                      variableMatrix(c+1:end, readOnlyBlocks*a
                            +1:readOnlyBlocks*a+a) = temp;
69                  end
70              end
71          case contains(resetMode, "channel")
72              if (resetFlag)
73                  if (targetFraction >= 1)
74                      variableMatrix(c+1:end, readOnlyBlocks*a
                            +1:readOnlyBlocks*a+a) = Hspa(c+1:end
                            , 1:a).*channelLLR(currentColumn:
                            currentColumn+a-1);
75                  else
76                      targetVector = reshape(variableMatrix(c
                            +1:end, readOnlyBlocks*a+1:
                            readOnlyBlocks*a+a), 1, []);
77                      targetNodes = round(targetFraction*nnz(
                            targetVector));
78                      zero_idx = find(targetVector == 0);
79                      targetVector(zero_idx) = inf;
80                      [~, reset_idx] = mink(targetVector,
                            targetNodes, 'ComparisonMethod', 'abs
                            ');
81                      temp = Hspa(c+1:end, 1:a).*channelLLR(
                            currentColumn:currentColumn+a-1);
82                      temp = reshape(temp, 1, []);
83                      targetVector(reset_idx) = temp(reset_idx
                            );
84                      targetVector(zero_idx) = 0;
85                      originalSize = size(variableMatrix(c+1:
                            end, readOnlyBlocks*a+1:
                            readOnlyBlocks*a+a));
86                      variableMatrix(c+1:end, readOnlyBlocks*a
                            +1:readOnlyBlocks*a+a) = reshape(
                            targetVector, originalSize);
87                  end
88              end
89          case contains(resetMode, "readOnly")
90              if (resetFlag)
91                  readOnlyBlocks = 0;
92              end
93      end
94      resets = resets + resetFlag;
```

```
95      % e selezione dei primi a elementi
96      estimatedCodeword(1, currentColumn:(currentColumn+a-1))
           = windowCodeword(1, 1:a);
97      currentRow = currentRow + c;
98      currentColumn = currentColumn + a;
99      if (readOnlyBlocks < ms)
100         readOnlyBlocks = readOnlyBlocks + 1;
101     end
102     if (i==target_window)
103         return
104     end
105 end
106
107 % terminazione della finestra scorrevole
108 if (currentColumn + Wb >= size(Hconv,2))
109     i = numberOfWindows;
110     isLast = true;
111     if contains(resetMode, "fixed")
112         resetFlag = mod(i, resetFactor) == 0;
113         resets = resets + resetFlag;
114     end
115     Hspa = Hconv(currentRow:end, currentColumn:end);
116     [windowCodeword, ~, ~, ~, llr_evolution] = ...
117     ImprovedSumProductAlgorithm(Hspa, channelLLR(
           currentColumn:end), ...
118     variableMatrix, readOnlyBlocks, ms, a, c, spaIterations,
           modulationIndex, isLast, clipValue, resetFlag,
           target_variable_nodes);
119 end
120     estimatedCodeword(1,currentColumn:end) = windowCodeword;
121 end
```

A.2.3. SPA adattato alla finestra scorrevole

```
1  function [estimatedCodeword, llrTotal, ...
2      extrinsicLLR_updated, current, llr_evolution] =
          ImprovedSumProductAlgorithm(H, channelLLR,
          extrinsicLLR, readOnlyBlocks, ms, a, c, iterations,
          modulationIndicator, isLast, clipValue, resetFlag,
          varargin)
3
4  if (nargin < 13)
```

```
5       target_variable_nodes = [1, 1];
6   else
7       target_variable_nodes = varargin{1};
8       if (length(target_variable_nodes)<2)
9           target_variable_nodes = [1, target_variable_nodes];
10      end
11  end
12  rows = size(H, 1);
13  columns = size(H, 2);
14  estimatedCodeword = zeros(1, columns);
15  llrTotal = zeros(1, columns);
16  llr_evolution = [];
17  % inizializzazione dei nodi variabile
18  variableMatrix = clip(variableMatrixInitialization(H,
        extrinsicLLR, channelLLR, readOnlyBlocks, ms, a, c,
        isLast, resetFlag), -clipValue, clipValue);
19  temp_variable = ones(size(variableMatrix));
20  idx = find(variableMatrix);
21  H_extended = sparse(zeros(size(variableMatrix)));
22  H_extended(idx) = 1;
23  current = 1;
24  while(current < iterations)
25      divisor = tanh(variableMatrix*(1/2));
26      temp_variable(idx) = divisor(idx);
27      % aggiornamento degli elementi di checkMatrix
28      checkMatrix = 2*atanh(H_extended.*((prod(temp_variable,
            2)./temp_variable)));
29      checkMatrix=clip(checkMatrix, -clipValue, clipValue);
30
31      % aggiornamento dei nodi variabile
32      temp_check = sum(checkMatrix(:, (readOnlyBlocks*a)+1:end
            ));
33      variableMatrix(:, (readOnlyBlocks*a)+1:end) = H.*(
            channelLLR+(temp_check-checkMatrix(:, (readOnlyBlocks
            *a)+1:end)));
34      variableMatrix = clip(variableMatrix, -clipValue,
            clipValue);
35      % aggiornamento dei LLR estrinseci distinguendo le fasi
            di transitorio
36      % e regime
37      if (readOnlyBlocks == ms) % regime
38          extrinsicLLR_updated = variableMatrix(:, a+1:end);
```

```
39      else % transitorio
40          extrinsicLLR_updated = variableMatrix;
41      end
42      % calcolo LLR Totale
43      llrTotal = clip(channelLLR + temp_check, -clipValue,
           clipValue);
44      llr_evolution = [llr_evolution; llrTotal(
           target_variable_nodes(1):target_variable_nodes(2))];
45      estimatedCodeword = double((sign(modulationIndicator)*
           llrTotal)<=0);
46      if (isLast)
47          partialSyndrome = estimatedCodeword*transpose(H);
48          if (isequal(mod(partialSyndrome, 2), zeros(1, rows))
               )
49              return;
50          end
51      else
52          partialSyndrome = estimatedCodeword(1:min((ms+1)*a,
               columns))*...
53              transpose(H(1:min((ms+1)*c, rows), 1:min((ms+1)*
                   a, columns)));
54          if (isequal(mod(partialSyndrome, 2), zeros(1, min((
               ms+1)*c, rows))))
55              return;
56          end
57      end
58      current = current+1;
59  end
```

A.2.4. Inizializzazione matrice dei nodi variabile

```
1   function variableMatrix = variableMatrixInitialization(Hspa,
        llrExtrinsic, channelLLR, readOnlyBlocks, ms, a, c,
       isLast, resetFlag)
2   % inizializzazione della matrice alle informazioni
       provenienti dal canale
3   variableMatrix = Hspa.*channelLLR;
4   if (resetFlag)
5       if (isLast)
6           llrExtrinsic = [llrExtrinsic; zeros(size(
               variableMatrix,1)+c-size(llrExtrinsic,1), size(
               llrExtrinsic,2))];
```

```
7           variableMatrix = [llrExtrinsic(c+1:end, (ms-
                readOnlyBlocks)*a+1:ms*a), variableMatrix];
8           return;
9       else
10          if (readOnlyBlocks == 0)
11              return
12          else
13              if (size(llrExtrinsic, 1) == size(variableMatrix
                    , 1))
14                  llrExtrinsic = [llrExtrinsic; zeros(c, size(
                        llrExtrinsic, 2))];
15              end
16              variableMatrix = [llrExtrinsic(c+1:end, 1:
                    readOnlyBlocks*a), variableMatrix];
17              return
18          end
19      end
20  else % nessun reset
21      if (readOnlyBlocks == 0)
22          return
23      else
24          if (isLast)
25              variableMatrix(1:size(llrExtrinsic,1)-c, 1:size(
                    llrExtrinsic,2)-readOnlyBlocks*a) ...
26                  = llrExtrinsic(c+1:end, (readOnlyBlocks*a)
                        +1:end);
27              llrExtrinsic = [llrExtrinsic; zeros(size(
                    variableMatrix,1)+c-size(llrExtrinsic,1),
                    size(llrExtrinsic,2))];
28              variableMatrix = [llrExtrinsic(c+1:end, (ms-
                    readOnlyBlocks)*a+1:ms*a), variableMatrix];
29              return
30          end
31          if(size(variableMatrix(1:end-c, 1:end-a), 2) == size
                (llrExtrinsic(c+1:end, (readOnlyBlocks*a)+1:end),
                 2))
32              rightBoundary = size(variableMatrix, 2)-a;
33          else
34              rightBoundary = size(variableMatrix, 2);
35          end
36          if (size(llrExtrinsic, 1) == size(variableMatrix, 1)
                )
```

```
37              variableMatrix(1:end-c, 1:rightBoundary) ...
38              = llrExtrinsic(c+1:end, (readOnlyBlocks*a)+1:end
                    );
39              llrExtrinsic = [llrExtrinsic; zeros(c, size(
                    llrExtrinsic, 2))];
40          else
41              variableMatrix(1:end, 1:rightBoundary) ...
42              = llrExtrinsic(c+1:end, (readOnlyBlocks*a)+1:end
                    );
43          end
44          variableMatrix = [llrExtrinsic(c+1:end, 1:
                readOnlyBlocks*a), variableMatrix];
45      end
46  end
```

A.3. Analisi LLR

```
1   function [llr_evolution_container, target_window,
        transmissions, channelLLR] = LLRanalysis(analysis_mode,
        pool, EbNo, rate, spaIterations, Hconv, ms, a, c,
        numberOfBlocks, modulationIndex, clipValue, resetMode,
        resetFactor)
2
3   % gestione parametri per reset parziale dei simboli target
4   if (size(resetMode,2) == 2)
5       targetFraction = resetMode{2};
6       resetMode = resetMode{1};
7   else
8       targetFraction = 1;
9   end
10
11  EbNo_lin = 10^(EbNo/10);
12  stop_condition = false;
13  var = 1/(2*rate*EbNo_lin);
14  codeword = zeros(1, size(Hconv, 2));
15  bpsk_codeword = codeword*(-2)+1;
16  % calcolo dei parametri della finestra
17  Wb = numberOfBlocks*a;
18  windowHeight = numberOfBlocks*c;
19  numberOfWindows = floor((size(Hconv,2)-Wb)/a)+1;
20  transmissions = 0;
```

```
% simula una trasmissione finche' non viene verificata la condizione di stop
while(~stop_condition)
    transmissions = transmissions + 1;
    estimatedCodeword = zeros(1, size(Hconv, 2));
    awgn_codeword = bpsk_codeword + wgn(1, size(Hconv,2), var, 'linear');
    channelLLR = (2/var)*awgn_codeword;

    target_variable_nodes = [1, a];
    currentColumn = 1;
    currentRow = 1;
    variableMatrix = 0;
    resets = 0;
    isLast = false;
    resetFlag = false;
    readOnlyBlocks = 0;

    for i=1:numberOfWindows-1
    % creazione della finestra
    if contains(resetMode, "fixed")
        resetFlag = mod(i, resetFactor) == 0;
    end
    Hspa = Hconv(currentRow:(currentRow+windowHeight-1), currentColumn:(currentColumn+Wb-1));
    [windowCodeword, ~, variableMatrix, spaCurrentIteration, llr_evolution] = ...
    ImprovedSumProductAlgorithm(Hspa, channelLLR(currentColumn:(currentColumn+Wb-1)), ...
    variableMatrix, readOnlyBlocks, ms, a, c, spaIterations, modulationIndex, isLast, clipValue, resetFlag, target_variable_nodes);
    if contains(resetMode, "dynamic")
        resetFlag = spaCurrentIteration >= resetFactor;
    end
    switch true
        case contains(resetMode, "zero")
            if (resetFlag)
                if (targetFraction >= 1)
                    variableMatrix(c+1:end, readOnlyBlocks*a+1:readOnlyBlocks*a+a) = 0;
                else
```

```
55                      targetVector = reshape(variableMatrix(c
                            +1:end, readOnlyBlocks*a+1:
                            readOnlyBlocks*a+a), 1, []);
56                      targetNodes = round(targetFraction*nnz(
                            targetVector));
57                      zero_idx = find(targetVector == 0);
58                      targetVector(zero_idx) = inf;
59                      [~, reset_idx] = mink(targetVector,
                            targetNodes, 'ComparisonMethod','abs'
                            );
60                      temp = variableMatrix(c+1:end,
                            readOnlyBlocks*a+1:readOnlyBlocks*a+a
                            );
61                      temp(reset_idx) = 0;
62                      variableMatrix(c+1:end, readOnlyBlocks*a
                            +1:readOnlyBlocks*a+a) = temp;
63                  end
64                  resets = resets + 1;
65              end
66          case contains(resetMode, "channel")
67              if (resetFlag)
68                  if (targetFraction >= 1)
69                      variableMatrix(c+1:end, readOnlyBlocks*a
                            +1:readOnlyBlocks*a+a) = Hspa(c+1:end
                            , 1:a).*channelLLR(currentColumn:
                            currentColumn+a-1);
70                  else
71                      targetVector = reshape(variableMatrix(c
                            +1:end, readOnlyBlocks*a+1:
                            readOnlyBlocks*a+a), 1, []);
72                      targetNodes = round(targetFraction*nnz(
                            targetVector));
73                      zero_idx = find(targetVector == 0);
74                      targetVector(zero_idx) = inf;
75                      [~, reset_idx] = mink(targetVector,
                            targetNodes, 'ComparisonMethod', 'abs
                            ');
76                      temp = Hspa(c+1:end, 1:a).*channelLLR(
                            currentColumn:currentColumn+a-1);
77                      temp = reshape(temp, 1, []);
78                      targetVector(reset_idx) = temp(reset_idx
                            );
```

```
79                      targetVector(zero_idx) = 0;
80                      originalSize = size(variableMatrix(c+1:
                            end, readOnlyBlocks*a+1:
                            readOnlyBlocks*a+a));
81                      variableMatrix(c+1:end, readOnlyBlocks*a
                            +1:readOnlyBlocks*a+a) = reshape(
                            targetVector, originalSize);
82                  end
83                  resets = resets + 1;
84              end
85          case contains(resetMode, "readOnly")
86              if (resetFlag)
87                  readOnlyBlocks = 0;
88                  resets = resets + 1;
89              end
90      end
91      % e selezione dei primi a elementi
92      estimatedCodeword(1, currentColumn:(currentColumn+a-1))
            = windowCodeword(1, 1:a);
93      currentRow = currentRow + c;
94      currentColumn = currentColumn + a;
95      if (readOnlyBlocks < ms)
96          readOnlyBlocks = readOnlyBlocks + 1;
97      end
98          switch analysis_mode
99              case "dynamic"
100                 if (sum(abs(estimatedCodeword))>0)
101                     stop_condition = true;
102                     target_window = i;
103                     pool_windows = pool;
104                     break;
105                 end
106             case "static"
107                 if (i==pool(1))
108                     stop_condition = true;
109                     target_window = i;
110                     pool_windows = pool(end)-pool(1);
111                     break;
112                 end
113         end
114     end
115     if (currentColumn + Wb >= size(Hconv,2))
```

```
116     isLast = true;
117     target_variable_nodes = [1, size(Hconv, 2)-currentColumn
           +1];
118     if contains(resetMode, "fixed")
119         resetFlag = mod(i, resetFactor) == 0;
120     end
121     Hspa = Hconv(currentRow:end, currentColumn:end);
122     [windowCodeword, ~, ~, ~, llr_evolution] = ...
123     ImprovedSumProductAlgorithm(Hspa, channelLLR(
           currentColumn:end), ...
124     variableMatrix, readOnlyBlocks, ms, a, c, spaIterations,
            modulationIndex, isLast, clipValue, resetFlag,
           target_variable_nodes);
125     estimatedCodeword(1, currentColumn:end) = windowCodeword
           ;
126     if contains(resetMode, "dynamic")
127         resetFlag = spaCurrentIteration >= resetFactor;
128     end
129         switch analysis_mode
130         case "dynamic"
131             if (sum(abs(estimatedCodeword))>0)
132                 stop_condition = true;
133                 target_window = numberOfWindows;
134                 pool_windows = pool;
135             end
136         case "static"
137             if (i==pool(1))
138                 stop_condition = true;
139                 target_window = numberOfWindows;
140                 pool_windows = pool(end)-pool(1);
141             end
142         end
143     end
144 end
145
146 % dopo aver trovato un errore nella decodifica analizza la
       propagazione
147 % dello stesso sui successivi pool scorrimenti
148 if (i == numberOfWindows-1)
149     llr_evolution_container = cell(1);
150     llr_evolution_container{1} = llr_evolution;
151     return
```

```
152 else
153     llr_evolution_container = cell(min(abs(i+pool_windows-
          numberOfWindows-1), pool_windows)+1, 1);
154     llr_evolution_container{1} = llr_evolution;
155     for j=i+1:min(i+pool_windows, numberOfWindows)
156         if (j == numberOfWindows)
157             isLast = true;
158             rightBoundary = size(Hconv, 2);
159             bottomBoundary = size(Hconv, 1);
160         else
161             rightBoundary = currentColumn+Wb-1;
162             bottomBoundary = currentRow+windowHeight-1;
163         end
164         if contains(resetMode, "fixed")
165             resetFlag = mod(i, resetFactor) == 0;
166         end
167         Hspa = Hconv(currentRow:bottomBoundary,
              currentColumn:rightBoundary);
168         [windowCodeword, llrProcessed, variableMatrix,
              spaCurrentIteration, llr_evolution_container{j-(i
              -1)}] = ...
169         ImprovedSumProductAlgorithm(Hspa, channelLLR(
              currentColumn:rightBoundary), ...
170         variableMatrix, readOnlyBlocks, ms, a, c,
              spaIterations, modulationIndex, isLast, clipValue
              , resetFlag, target_variable_nodes);
171         if contains(resetMode, "dynamic")
172             resetFlag = spaCurrentIteration >= resetFactor;
173         end
174     switch true
175         case contains(resetMode, "zero")
176             if (resetFlag)
177                 if (targetFraction >= 1)
178                     variableMatrix(c+1:end, readOnlyBlocks*a
                          +1:readOnlyBlocks*a+a) = 0;
179                 else
180                     targetVector = reshape(variableMatrix(c
                          +1:end, readOnlyBlocks*a+1:
                          readOnlyBlocks*a+a), 1, []);
181                     targetNodes = round(targetFraction*nnz(
                          targetVector));
182                     zero_idx = find(targetVector == 0);
```

```
183                     targetVector(zero_idx) = inf;
184                     [~, reset_idx] = mink(targetVector,
                            targetNodes, 'ComparisonMethod','abs'
                            );
185                     temp = variableMatrix(c+1:end,
                            readOnlyBlocks*a+1:readOnlyBlocks*a+a
                            );
186                     temp(reset_idx) = 0;
187                     variableMatrix(c+1:end, readOnlyBlocks*a
                            +1:readOnlyBlocks*a+a) = temp;
188                 end
189                 resets = resets + 1;
190             end
191         case contains(resetMode, "channel")
192             if (resetFlag)
193                 if (targetFraction >= 1)
194                     variableMatrix(c+1:end, readOnlyBlocks*a
                            +1:readOnlyBlocks*a+a) = Hspa(c+1:end
                            , 1:a).*channelLLR(currentColumn:
                            currentColumn+a-1);
195                 else
196                     targetVector = reshape(variableMatrix(c
                            +1:end, readOnlyBlocks*a+1:
                            readOnlyBlocks*a+a), 1, []);
197                     targetNodes = round(targetFraction*nnz(
                            targetVector));
198                     zero_idx = find(targetVector == 0);
199                     targetVector(zero_idx) = inf;
200                     [~, reset_idx] = mink(targetVector,
                            targetNodes, 'ComparisonMethod', 'abs
                            ');
201                     temp = Hspa(c+1:end, 1:a).*channelLLR(
                            currentColumn:currentColumn+a-1);
202                     temp = reshape(temp, 1, []);
203                     targetVector(reset_idx) = temp(reset_idx
                            );
204                     targetVector(zero_idx) = 0;
205                     originalSize = size(variableMatrix(c+1:
                            end, readOnlyBlocks*a+1:
                            readOnlyBlocks*a+a));
206                     variableMatrix(c+1:end, readOnlyBlocks*a
                            +1:readOnlyBlocks*a+a) = reshape(
```

```
                        targetVector, originalSize);
207                 end
208                 resets = resets + 1;
209             end
210         case contains(resetMode, "readOnly")
211             if (resetFlag)
212                 readOnlyBlocks = 0;
213                 resets = resets + 1;
214             end
215     end
216     % e selezione dei primi a elementi
217     estimatedCodeword(1, currentColumn:(currentColumn+a-1))
           = windowCodeword(1, 1:a);
218     currentRow = currentRow + c;
219     currentColumn = currentColumn + a;
220     if (readOnlyBlocks < ms)
221         readOnlyBlocks = readOnlyBlocks + 1;
222     end
223     end
224 end
```

Bibliografia

[1] S. CCSD, "Low Density Parity Check codes for use in near-earth and deep space applications," *Exp. Specification CCSDS 131.1-O-2*, 2007.

[2] O. Book, *Short Blocklength LDPC codes for TC synchronization and channel coding*, 2012.

[3] 3GPP, *3rd Generation Partnership Project; 5G; NR; Multiplexing and channel coding*, TS 38.212 V15.2.0, 2018.

[4] R. Gallager, "Low-Density Parity-Check codes," *IRE Transactions on information theory*, vol. 8, no. 1, pp. 21–28, 1962.

[5] MacKay and R. Neal, "Good codes based on very sparse matrices," in *IMA International Conference on Cryptography and Coding*, Springer, 1995, pp. 100–111.

[6] A. J. Felstrom and K. S. Zigangirov, "Time-varying periodic convolutional codes with Low-Density Parity-Check matrix," *IEEE Transactions on Information Theory*, vol. 45, no. 6, pp. 2181–2191, 1999.

[7] W. E. Ryan and S. Lin, *Channel Codes: Classical and Modern*. Cambridge University Press, 2009.

[8] J. G. Proakis and M. Salehi, *Digital Communications, 5th Edition*. McGraw-Hill Higher Education, 2008.

[9] R. Tanner, "A recursive approach to low complexity codes," *IEEE Transactions on information theory*, vol. 27, no. 5, pp. 533–547, 1981.

[10] C. E. Shannon, "A mathematical theory of communication," *The Bell system technical journal*, vol. 27, no. 3, pp. 379–423, 1948.

[11] D. G. Mitchell, M. Lentmaier, and D. J. Costello, "Spatially coupled LDPC codes constructed from protographs," *IEEE Transactions on Information Theory*, vol. 61, no. 9, pp. 4866–4889, 2015.

[12] D. J. Costello, L. Dolecek, T. E. Fuja, J. Kliewer, D. G. Mitchell, and R. Smarandache, "Spatially coupled sparse codes on graphs: Theory and practice," *IEEE Communications Magazine*, vol. 52, no. 7, pp. 168–176, 2014.

[13] S. Kudekar, T. J. Richardson, and R. L. Urbanke, "Threshold saturation via spatial coupling: Why convolutional LDPC ensembles perform so well over the BEC," *IEEE Transactions on Information Theory*, vol. 57, no. 2, pp. 803–834, 2011.

[14] N. Wiberg, “Codes and decoding on general graphs,” 1996.

[15] J. Zhao, F. Zarkeshvari, and A. H. Banihashemi, “On implementation of min-sum algorithm and its modifications for decoding Low-Density Parity-Check (LDPC) codes,” *IEEE transactions on communications*, vol. 53, no. 4, pp. 549–554, 2005.

[16] A. R. Iyengar, M. Papaleo, P. H. Siegel, J. K. Wolf, A. Vanelli-Coralli, and G. E. Corazza, “Windowed decoding of protograph-based LDPC convolutional codes over erasure channels,” *IEEE Transactions on Information Theory*, vol. 58, no. 4, pp. 2303–2320, 2011.

[17] I. Ali, J.-H. Kim, S.-H. Kim, H. Kwak, and J.-S. No, “Improving windowed decoding of SC LDPC codes by effective decoding termination, message reuse, and amplification,” *IEEE access*, vol. 6, pp. 9336–9346, 2017.

[18] N. U. Hassan, M. Schlüter, and G. P. Fettweis, “Fully parallel window decoder architecture for Spatially-Coupled LDPC codes,” in *2016 IEEE International Conference on Communications (ICC)*, IEEE, 2016, pp. 1–6.

[19] H. Qi and N. Goertz, “Low-Complexity Encoding of LDPC Codes: A New Algorithm and its Performance,” *Institute for Digital Communications, Joint Research Institute for Signal & Image Processing, School of Engineering and Electronics, The University of Edinburgh*, 2007.

[20] Z. Wu, M. Zhao, and S. Wang, “An Efficient Encoding Method for Spatially Coupled Low-Density Parity-Check Codes,” in *2019 IEEE 5th International Conference on Computer and Communications (ICCC)*, IEEE, 2019, pp. 1505–1509.

[21] M. Zhu, D. G. Mitchell, M. Lentmaier, and D. J. Costello, “Systematic doping of SC-LDPC codes,” in *2022 IEEE International Symposium on Information Theory (ISIT)*, IEEE, 2022, pp. 536–541.

[22] M. Zhu, D. G. Mitchell, M. Lentmaier, and D. J. Costello, “Error Propagation Mitigation in Sliding Window Decoding of Spatially Coupled LDPC Codes,” *IEEE Journal on Selected Areas in Information Theory*, 2023.

[23] X. Wu, M. Jiang, and C. Zhao, “Decoding optimization for 5G LDPC codes by machine learning,” *Ieee Access*, vol. 6, pp. 50 179–50 186, 2018.