POLITECNICO DI TORINO

III Facoltà di Ingegneria

Anno Accademico 2003/2004

Studio del codice di Goppa (23, 13, 5)

Davide Boltri 93671
Matteo O. Bovero 93152
Alessandro Zummo 96389

Teoria dell'Informazione e Codici

Docente: Michele Elia

Indice

1 Introduzione

Scopo del presente elaborato è lo studio del codice di Goppa (23, 13, 5). I codici di Goppa, introdotti all'inizio degli anni '70, fanno parte della famiglia dei codici a blocchi, i quali hanno la particolarità di generare le parole di codice unicamente in funzione della parola di sorgente che si vuole trasmettere. La terna (23, 13, 5) che contraddistingue il codice oggetto di analisi ne mette sinteticamente in luce in forma numerica le caratteristiche essenziali. Ogni codice a blocco infatti è caratterizzato da una terna (n,k,d) ove:

- $n = 23$ è la lunghezza della parola di codice
- $k = 13$ è la dimensione del codice
- $d = 5$ è la distanza minima del codice, ovvero il numero minimo di posizioni in cui devono differire due parole di codice per essere considerate diverse.

Il parametro d è significativo del numero di errori che il codice può correggere: se infatti 5 è la distanza minima, un codice lineare può rilevare e correggere fino a 2 errori, e può rilevare, ma non correggere, fino a 4 errori.

Al momento di effettuare la codifica, se il codificatore è sistematico, la parola di codice risultante è composta da due parti: k bit sono uguali alla parola di sorgente; i restanti $n-k$ bit sono i "bit di parità" aggiunti dal codificatore. La decodifica si serve della parola di codice e dei bit di parità per rilevare ed eventualmente correggere errori di trasmissione. La decodifica può essere fatta in due modi: tecnica della minima distanza (decodifica MD) e tecnica algebrica a coset leader unico (decodifica UCL).

Come punti di partenza per lo studio sono noti il polinomio $g(x)$ generatore del campo $GF(2^5)$ da cui si sviluppa il codice

$$g(x) = x^5 + x^4 + x^3 + x^2 + 1$$

il sottinsieme di 23 elementi del campo utilizzati per la generazione del codice

$$\Gamma = \{\gamma_0, \gamma_1, \ldots, \gamma_{22}\} = \{0, 1, x, x^2, \ldots, x^{21}\}$$

e il polinomio di Goppa $G(z)$ da cui è generato il codice:

$$G(z) = z^2 + z + 1$$

Lo studio coprirà, più in dettaglio, i seguenti punti:

- ◇ Calcolo della distribuzione dei pesi W_i
- ◇ Calcolo della distribuzione L_i dei pesi del coset leader di peso minimo
- ◇ Calcolo della distribuzione dei pesi dei coset leader relativi alla decodifica UCL
- ◇ Simulazione del codificatore sistematico determinando la matrice generatrice

- ◇ Simulazione della trasmissione, ossia del canale
- ◇ Simulazione del decodificatore algebrico UCL
- ◇ Determinazione delle espressioni polinomiali per il calcolo della probabilità di rivelare l'errore p_r, della probabilità di undetected error p_u e della probabilità di errore sulla parola p_w sul canale binario simmetrico
- ◇ Determinazione delle espressioni polinomiali per il calcolo della probabilità di errore sul bit p_b sia con la decodifica a minima distanza sia UCL sul canale binario simmetrico

Data la mole di calcoli richiesta, il lavoro svolto comprenderà sia una parte matematica per individuare le basi necessarie all'elaborazione degli algoritmi di calcolo (capitolo 2), sia una parte di programmazione per l'implentazione di tali algoritmi in un programma che consenta di fornire in output i dati richiesti (capitolo 3). Infine i risultati ottenuti saranno raccolti e analizzati al termine della relazione (capitolo 4).

2 Basi matematiche

Con la notazione $GF(q)$ si indicano i campi finiti, detti anche *Campi di Galois*, entità matematiche formate da q elementi per i quali valgono due operazioni (addizione e moltiplicazione) il cui risultato sono elementi appartenenti a loro volta al campo (proprietà di chiusura). In generale un campo $GF(q)$ esiste solo se q è un numero primo o potenza di un numero primo, nel qual caso si usa la notazione $GF(q^m)$. Per un campo $GF(q^m)$, gli elementi sono m-ple di elementi di $GF(q)$, cioè polinomi di grado $m-1$, con coefficienti in $GF(q)$.

Il campo di Galois su cui è sviluppato il codice oggetto di studio è $GF(2^5)$, pertanto i suoi elementi sono polinomi di grado 4 con coefficienti in $GF(2)$, ovvero i coefficienti possono valere 0 o 1. Per ottenere i 32 elementi del campo si procede come segue: si impone

$$x^5 + x^4 + x^3 + x^2 + 1 = 0$$

da cui si ricava, seguendo le regole per le operazioni sui campi finiti

$$x^5 = x^4 + x^3 + x^2 + 1$$

Gli elementi non nulli del campo si trovano moltiplicando successivamente ogni termine per x ed esprimendo i termini di grado maggiore o uguale a 5 in funzione dei termini precedenti. I coefficienti dei polinomi permettono di esprimere ogni elemento come una sequenza di cinque valori binari, la cui espressione viene indicata come *espansione* di $GF(2^5)$ in $GF(2)$. Quello che si ottiene è illustrato nella tabella seguente:

Elemento	Forma polinomiale	Forma binaria
0	0	00000
1	1	10000
x	x	01000
x^2	x^2	00100
x^3	x^3	00010
x^4	x^4	00001
x^5	$x^5 = x^4 + x^3 + x^2 + 1$	10111
x^6	$x^6 = x \times x^5$	11100
...	...	...
x^{30}	$x^{30} = x \times x^{29}$	01111

Una volta generati gli ementi del campo, se ne prendono i primi 23, come richiesto dall'insieme Γ dato, e li si sostituisce a z nell'epressione del polinomio di goppa $G(z)$. I polinomi $G(\gamma_i)$ che si ottengono, vengono poi invertiti con l'algoritmo di Euclide in modo da ottenere l'espressione polinomiale in $GF(32)$ dei termini $\frac{1}{G(\gamma_i)}$. Questi termini sono raccolti nella matrice di parità H di dimensione 23×2 che per i codici di Goppa ha la seguente forma:

$$H = \begin{bmatrix} \frac{1}{G(\gamma_0)} & \frac{1}{G(\gamma_1)} & \frac{1}{G(\gamma_2)} & \cdots & \frac{1}{G(\gamma_{22})} \\ \\ \frac{\gamma_0}{G(\gamma_0)} & \frac{\gamma_1}{G(\gamma_1)} & \frac{\gamma_2}{G(\gamma_2)} & \cdots & \frac{\gamma_{21}}{G(\gamma_{22})} \end{bmatrix}$$

Esprimendo i polinomi ottenuti in forma binaria (secondo le regole del campo $GF(2^5)$), la matrice H può essere portata dalla forma esponenziale in forma binaria, ottendo una matrice di parità, formata da soli 1 e 0 di dimensione 10×23. Poichè si richiede che il codice sia sistematico, la matrice H viene ridotta in forma echelon. Una matrice è detta in forma echelon ridotta se e solo se rispetta le seguenti proprietà:

- il primo elemento non nullo di una riga compare più a destra del primo elemento non nullo della riga precedente
- il primo elemento non nullo di una riga è un 1
- i termini sotto e sopra il primo elemento non nullo di una riga sono 0

Ogni matrice può essere trasformata in forma echelon ridotta con un numero finito di operazioni elementari. La matrice H permette di sapere se la parola ricevuta dopo la trasmissione è una parola di codice. Infatti, indicando con $\mathbf{s}$ la *sindrome* e con $\mathbf{r}$ la parola ricevuta dal decodificatore, se dopo l'operazione

$$\mathbf{s} = H \times \mathbf{r}$$

$\mathbf{s}$ è il vettore nullo, allora $\mathbf{r}$ è la parola di codice.

Dalla matrice H in forma echelon è possibile risalire alla matrice generatrice del codice sistematico G. La matrice G permette di effettuare l'operazione di codifica. Infatti se $\mathbf{u}$ è la parola di sorgente e $\mathbf{c}$ la parola di codice, la codifica sistematica è:

$$\mathbf{c} = G \times \mathbf{u}$$

Per mezzo del campo $GF(2^5)$ e delle matrici H e G è possibile svolgere tutti i calcoli necessari a rispondere ai quesiti esposti nell'introduzione. Nel capitolo seguente verranno illustrati gli algoritmi per il calcolo di $GF(2^5)$, di H e di G e per il loro impiego nelle routine di calcolo e simulazione.

3 Algoritmi e loro implementazione

Nella scrittura del programma per calcolatore si è tenuto in conto del fatto che alcuni risultati (generazione delle matrici, calcolo dei coset leader e dei relativi pesi) hanno bisogno di essere calcolati un'unica volta, mentre la parte di simulazione del canale sfrutta tali risultati per operare su dati di input di volta in volta diversi.

Per tale ragione si è provveduto a creare due programmi in C, `genera` e `simula`, dei quali il primo genera tutti i dati necessari alla creazione delle matrici e dei vettori che hanno bisogno di essere calcolati un'unica volta, salvandoli su file per una successiva consultazione, mentre il secondo si occupa di simulare il canale nelle fasi di codifica/decodifica. Un'interfaccia grafica in Java consente all'utente di inserire i dati di input e di far partire a scelta entrambi i programmi (se si desidera far ricalcolare tutto) oppure solo il programma `simula` se si desidera solamente simulare il canale, utilizzando quanto già calcolato in precedenza, risparmiando una notevole quantità di tempo.

La scelta del C come linguaggio di programmazione per gli eseguibili `genera` e `simula` è stata dettata dalla portabilità e dalla forte capacità di ottimizzazione dei compilatori attuali che permettono di migliorare le prestazioni, mentre la scelta di Java per la costruzione dell'interfaccia è stata dettata dalla possibilità di esportare lo stesso codice in ambienti diversi senza bisogno di ulteriori modifiche. E' stato quindi possibile creare un programma eseguibile sia in ambiente Windows sia in ambiente Linux, scelti come piattaforme di riferimento durante le fasi di sviluppo.

Le operazioni compiute dal programma `genera` sono nell'ordine le seguenti:

1. Calcolo degli elementi del campo di Galois $GF(2^5)$: come spiegato nell'introduzione matematica, tutti gli elementi del campo con esponente superiore a 4 sono creati in modo iterativo sommando al precedente il polinomio $1+x^2+x^3+x^4$. Interpretando i coefficienti di ogni polinomio come una stringa di bit si ottiene una matrice di 32 righe e cinque colonne in cui gli elementi di $GF(32)$ sono memorizzati come estensione del campo $GF(2)$;

2. Calcolo della matrice H: presi i primi 23 elementi del campo (quelli indicati nell'insieme Γ), ognuno di essi viene sostituito al posto di z nel polinomio di Goppa $G(z)$. Ogni polinomio $G(\gamma_i)$ viene poi invertito per mezzo dell'algoritmo di Euclide. I polinomi risultato dell'inversione sono sostituiti dal rispettivo elemento del campo di Galois in modo da costruire una prima forma della matrice H, che si è chiamata H esponenziale, di 2 righe e 23 colonne. Ogni elemento della matrice H esponenziale, che è un termine del tipo $x^i \in GF(32)$, viene infine sostituito dalla

rispettiva forma binaria per ottenere la matrice di parità H vera e propria, formata da 10 righe e 23 colonne di 1 e 0;

3. Riduzione di H in forma echelon: la matrice H ottenuta al passo precedente viene convertita in forma echelon per mezzo di operazioni elementari sulle righe e sulle colonne;

4. Generazione della matrice G in forma sistematica: la matrice generatrice G è una matrice di 23 righe e 10 colonne, ricavata direttamente dalla matrice H in forma echelon ottenuta al passo precedente. Questa matrice G è quella che verrà utilizzata per la simulazione del codificatore sistematico;

5. Calcolo della distribuzione dei pesi w_i: si generano tutte le 2^{13} possibili parole di codice, le si codifica sfruttando la matrice G e se ne calcola il peso, dato dal numero di 1 contenuti nella stringa di codice. Poichè parte delle informazioni calcolate in questa fase possono essere riutilizzate successivamente nel calcolo dei coset leader, si è provveduto a salvare in una struttura dati a parte le 2^{13} parole di sorgente con la relativa parola di codice e peso;

6. Calcolo dei coset leader: si tratta della parte che computazionalmente porta via più tempo perchè richiede la generazione delle 2^{23} possibili stringhe di bit che possono essere ricevute dal decodificatore. Per cercare di ottimizzare il tempo di calcolo, si generano prima i 2^{10} possibili bit di parità, poi si generano le 2^{13} possibili parole di codice (sfruttando le informazioni salvate a parte al passo precedente) e le si uniscono a formare le 2^{23} possibili stringhe di bit. Per ogni stringa di bit ottenuta si calcola il peso (numero di 1 contenuti). In questo modo è possibile calcolare per ogni parola di sorgente il coset leader. Il calcolo del coset leader è differente a seconda che si voglia utilizzare la decodifica MD oppure la decodifica UCL: nel primo caso è coset leader la parola di codice di peso minimo, nel secondo caso è coset leader la parola di codice di peso minimo oppure, se questa ha peso maggiore di 2 (numero massimo di errori corregibili dal codice) è coset leader quella che presenta solo zeri nella parte di informazione.

Al termine dell'elaborazione il programma `genera` salva nella directory `dati` tutte le informazioni calcolate: in particolare in `elementi_GF.txt` sono salvati gli elementi del campo $GF(32)$; in `H.txt` è salvata la matrice di parità H; in `echelon.txt` è salvata la matrice H in forma echelon; in `G.txt` è salvata la matrice generatrice per il codice sistematico; in `pesi.txt` è salvata la distribuzione dei pesi w_i; in `coset_md.txt` e `coset_ucl.txt` sono salvati i coset leader rispettivamente per la decodifica MD e UCL; infine in `pesi_md.txt` e `pesi_ucl.txt` sono salvate le distribuzioni dei pesi L_i dei coset leader rispettivamente per la decodifica MD e UCL.

Il programma `simula` simula il funzionamento di un sistema costituito da una sorgente di bit che codifica sistematicamente le parole di codice con il codice di Goppa (23, 13, 5), da un canale che può introdurre errori sui dati trasmessi e da un ricevitore che

decodifica secondo la tecnica UCL la stringa di bit ricevuta dal canale. Il flusso del programma comprende i seguenti passi:

1. Caricamento dei dati necessari alla simulazione: sono caricati in memoria tutti i file generati dal programma `genera` per poterli utilizzare nel corso della simulazione;

2. Simulazione del codificatore sistematico: viene operata la codifica sistematica con il seguente prodotto di matrici:

   $$\mathbf{c} = G \times \mathbf{u}$$

   dove **c** è la parola di codice e **u** è la parola di sorgente

3. Simulazione del canale: alla parola di codice ottenuta dal codificatore sistematico viene sommata la parola di errore **e** introdotta dal canale così da generare la parola **r** ricevuta dal decodificatore:

   $$\mathbf{r} = \mathbf{c} + \mathbf{e}$$

4. Simulazione del decodificatore: la decodifica viene effettuata con il metodo della sindrome. Per prima cosa si calcola la sindrome con l'operazione di prodotto matriciale:

   $$\mathbf{s} = H \times \mathbf{r}$$

   Se la sindrome **s** è nulla significa che la trasmissione non ha incontrato errori e la parola è stata trasmessa e decodificata in modo corretto. In caso contrario la trasmissione ha subito degli errori, e per poter decodificare la parola ricevuta è necessario cercare il coset leader l_r che permetta di stimare il possibile codice trasmesso $\hat{c}$

   $$\hat{c} = l_r + \mathbf{r}$$

   Il coset leader adatto è quello che soddisfa l'equazione

   $$H \times l_r = \mathbf{s}$$

5. Calcolo delle probabilità: Se con p si intende la probabilità di transizione del canale, servendosi delle varie distribuzioni dei pesi calcolate dal programma `genera` è possibile calcolare le probabilità di rivelare l'errore p_r, di undected error p_u, di errore sulla parola p_w e di errore sul bit p_b implementando le seguenti formule:

   $$p_r = 1 - \sum_{h=0}^{n} A_h p^h (1-p)^h$$

   $$p_u = \sum_{h=1}^{n} A_h p^h (1-p)^{n-h}$$

   $$p_w = 1 - \sum_{i=0}^{n} L_i p^i (1-p)^{n-i}$$

$$p_b = \frac{1}{k} \sum_{h=0}^{n} B_h p^h (1-p)^{n-h}$$

dove A_h è il numero di parole di codice di peso h, che nel caso di p_r e p_u è la distribuzione dei pesi delle parole di codice e nel caso di p_w è la distribuzione dei pesi dei coset leader nella decodifica MD o UCL.

4 Conclusioni

Per testare il programma occorre mandarlo in esecuzione da riga di comando in questo modo:

```
> java GoppaInterface
```

L'interfaccia grafica che compare è quella illustrata nella figura 1. Per inserire i dati di

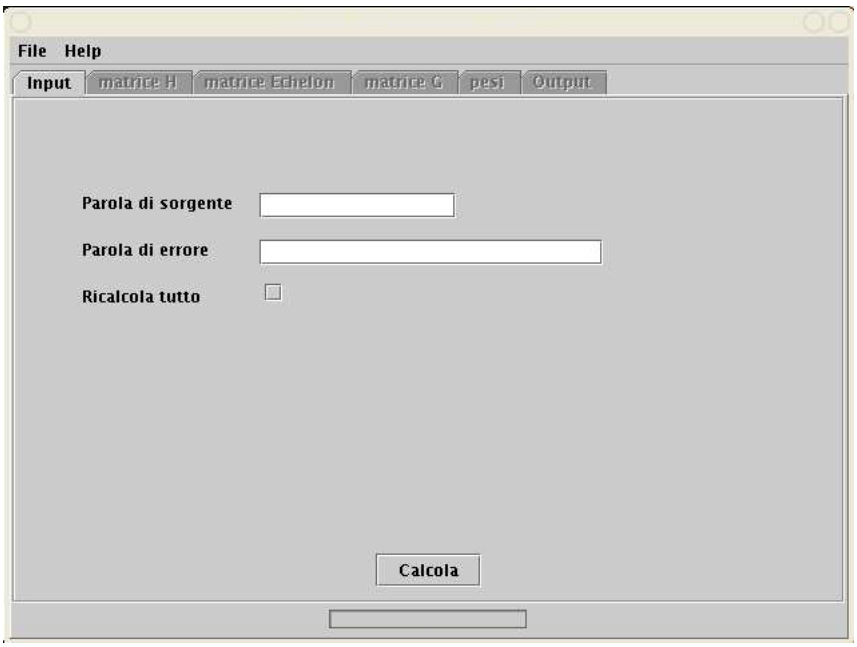


Figura 1: Finestra principale del programma

input nei form si può scegliere se aprire un file di testo già salvato su disco oppure se inserire manualmente i dati. Qualora si scelga quest'ultima opzione, nel campo ***Parola di sorgente*** si deve inserire una parola di 13 bit e nel campo ***Parola di errore*** una parola di errore di 23 bit. L'opzione ***Ricalcola tutto*** permette di scegliere se far ricalcolare tutti i dati oppure se procedere immediatamente con la simulazione utilizzando i dati precalcolati. Premendo il pulsante ***Calcola*** la barra di progressione si attiva ed il programma inizia l'elaborazione che, nel caso in cui si sia scelto di far ricalcolare tutto, dura mediamente mezzo minuto (tempi calcolati su processore a 2.5GHz).

Quando la barra si disattiva il programma ha terminato i calcoli e i tab nella parte alta della finestra diventano attivi e navigabili. Nel tab ***matrice H*** è riportata la matrice di

parità; nel tab ***matrice echelon*** è riportata la matrice H in forma echelon; nel tab ***matrice G*** è riportata la matrice generatrice del codice sistematico; nel tab ***pesi*** sono riportati i pesi delle parole di codice e dei coset leader MD e UCL; infine nel tab ***Output*** sono mostrate in ordine: la parola di codice; la parola ricevuta dal decodificatore (parola di codice + errore); la parola decodificata e il coset leader utilizzato nella decodifica UCL.

Nella directory `dati` sono memorizzati dal programma tutto quanto calcolato, e che viene qui riproposto per una più comoda consultazione:

- Matrice di parità H:

$$\begin{pmatrix}
1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\
0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\
0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0
\end{pmatrix}$$

- Matrice H in forma echelon:

$$\begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1
\end{pmatrix}$$

- Matrice generatrice G:

$$\begin{pmatrix}
1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\
1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\
0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\
1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\
1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}$$

- Distribuzione dei pesi

L_i codifica MD	L_i codifica UCL	B_i codifica MD	B_i codifica UCL
1	1	0	0
23	23	0	0
253	253	0	0
689	98	3287	3435
58	165	29035	22053
0	182	123638	101073
0	172	409520	357484
0	86	1098513	997748
0	37	2411353	2257890
0	6	4392585	4207551
0	1	6670373	6512908
0	0	8490981	8424733
0	0	9091795	9153008
0	0	8201597	8362949
0	0	6227345	6413016

L_i codifica MD	L_i codifica UCL	B_i codifica MD	B_i codifica UCL
0	0	3956097	4113211
0	0	2088851	2189862
0	0	906499	956133
0	0	316383	336860
0	0	87080	93180
0	0	18070	19619
0	0	2689	2950
0	0	251	276
0	0	10	13

- Polinomio con la distribuzione dei pesi:

$$1+37w^5+118w^6+233w^7+464w^8+817w^9+1110w^{10}+1285w^{11}+1342w^{12}+ \\ +1151w^{13}+786w^{14}+459w^{15}+239w^{16}+107w^{17}+34w^{18}+7w^{19}+2w^{20}$$

- Polinomio della distribuzione dei pesi dei coset leader MD

$$1+23L+253L^2+689L^3+58L^4$$

- Polinomio della distribuzione dei pesi dei coset leader UCL

$$1+23L+253L^2+98L^3+165L^4+182L^5+172L^6+86L^7+37L^8+6L^9+L^{10}$$

Le espressioni delle probabilità richieste sono già state riportate nel capitolo 3, ma vengono qui ripetute per maggiore chiarezza accompagnate dalle relative definizioni.

Si definisce **probabilità di errore sulla parola** p_w *la probabilità media che la parola all'uscita dal decodificatore sia diversa dalla parola trasmessa*

$$p_w = 1 - \sum_{i=0}^{n} L_i p^i (1-p)^{n-i}$$

Si definisce **probabilità di errore sul bit** p_b *la probabiltà media che un simbolo di informazione sia errato dopo la decodifica*

$$p_b = \frac{1}{k} \sum_{h=0}^{n} B_h p^h (1-p)^{n-h}$$

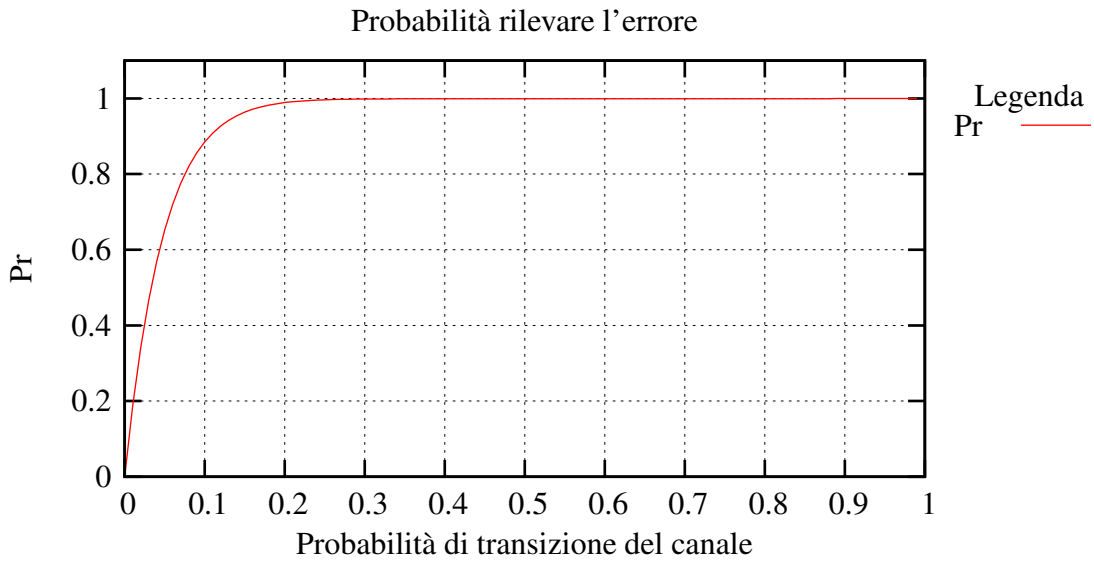
Si definisce **probabilità di rivelare l'errore** p_r *la probabilità media di riconoscere gli errori verificatisi durante la trasmissione*

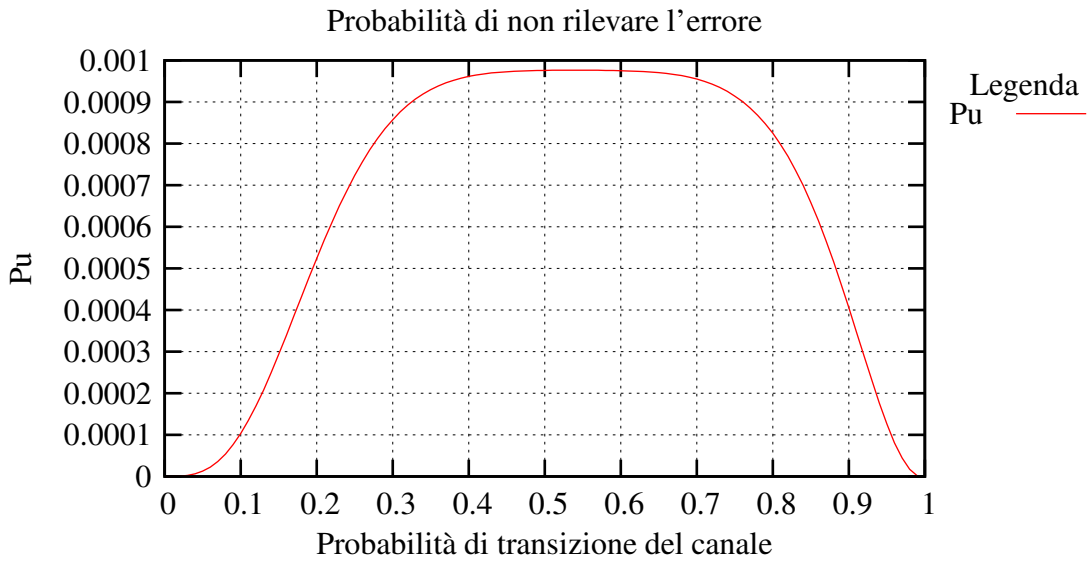
$$p_r = 1 - \sum_{h=0}^{n} A_h p^h (1-p)^h$$

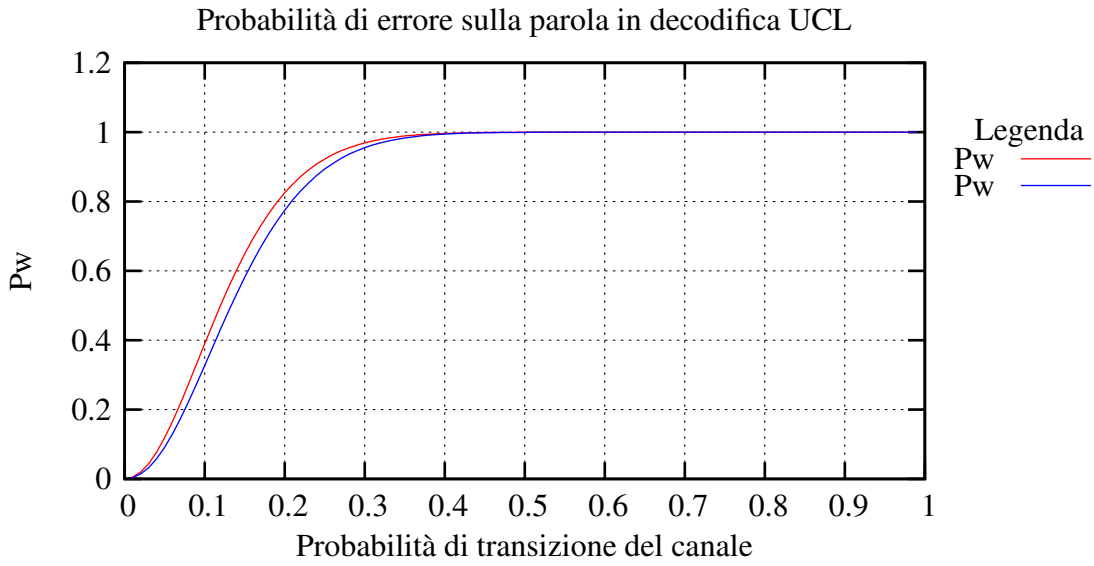
Si definisce **probabilità di undetected error** p_u *la probabilità media che durante la trasmissione si verifichino degli errori e che questi non siano rivelati*

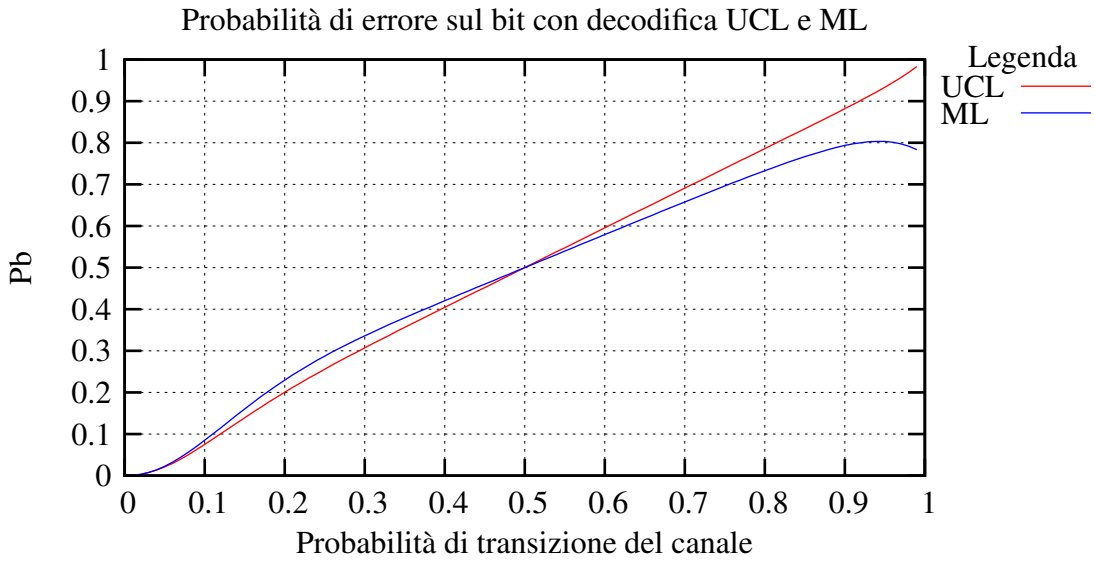
$$p_u = \sum_{h=0}^{n} A_h p^h (1-p)^h$$

Tutte le probabilità sono funzione della probabilità p di transizione del canale. Di seguito sono mostrati i grafici disegnati da GNUplot che permettono di visualizzare il comportamento delle probabilità al crescere di p. Le curve sono state disegnate per punti e i dati usati per il tracciamento sono memorizzati nei file della cartella `plot`.






Probabilità di errore sulla parola in decodifica UCL
Pw
Probabilità di transizione del canale
Legenda
Pw
Pw
0
0.2
0.4
0.6
0.8
1
1.2
0.1
0.3
0.5
0.7
0.9


Probabilità di errore sul bit con decodifica UCL e ML
Pb
Probabilità di transizione del canale
Legenda
UCL
ML
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1

5 Codice sorgente

Nelle pagine che seguono è riportato il codice sorgente in C dei programmi `genera` e `simula`. Una nota sui commenti: ogni funzione è descritta da un commento che illustra cosa fa, i parametri richiesti e l'eventuale risultato restituito. I parametri sono preceduti da delle etichette: l'etichetta `[in]` significa che il parametro è utilizzato come input dalla funzione; l'etichetta `[ou]` indica che il parametro è utilizzato come output dalla funzione, per esempio per memorizzare vettori o matrici elaborati dalla procedura; l'etichetta `[io]` indica che il parametro è utilizzato sia come input che come output, ovvero la funzione utilizza il valore contenuto nel parametro e ne scrive il risultato all'interno del parametro stesso, che quindi viene modificato al termine dell'elaborazione.

Listing 1: File `goppa.h`

```
1  #ifndef __GOPPA_H
2  #define __GOPPA_H
3
4  /*potenza a cui è elevato 2 nell'espressione GF(2 elevato m)*/
5  #define M 5
6  /*numero di elementi del campo di galois*/
7  #define Q 32
8  /*grado del polinomio di goppa*/
9  #define T 2
10 /*valori della terna (n, k, d) per la definizione del codice di
11  * Goppa*/
12 #define N 23
13 #define K 13
14 #define D 5
15 /*Assegna un valore ad infinito*/
16 #define INFTY 32768
17 /*Numero massimo di parole di sorgente disponibili su 13 bit*/
18 #define N_SORGENTI 8192
19 /*Numero massimo di parole di parita'*/
20 #define N_PARITA 1024
21
22 /*codici di errore*/
23 #define ERROR_FOPEN 1
24 #define ERROR_MATRIX 2
25 #define ERROR_DATA 3
26
27 typedef struct p_codice_st{
28     int sorgente[K];
29     int codice[N];
30     int peso_src;
31     int peso_cod;
32 } P_CODICE;
33 #endif
```

Listing 2: File genera_main.c

```
1  #include <string.h>
2  #include "goppa.h"
3  #include "utils.h"
4  #include "genera.h"
5
6  int main(int argc, char *argv[])
7  {
8      /*elementi del campo GF(32)*/
9      int GF32[Q][M];
10     /*rappresentazione del polinomio generatore del campo*/
11     int pol_gen[M+1] = {1, 0, 1, 1, 1, 1};
12     /*polinomio di Goppa*/
13     int pol_goppa[T+1] = {1, 1, 1};
14     /*matrice H in forma esponenziale*/
15     int H_exp[T][N];
16     /*matrice di parita' H*/
17     int H[T*M][N];
18     /*matrice H ridotta in forma echelon*/
19     int echelon[T*M][N];
20     /*vettore Gamma*/
21     int gamma[N];
22     /*matrice generatrice G*/
23     int G[N][K];
24     /*contiene tutte le possibili parole di sorgente e relativi
25      * codice e peso*/
26     P_CODICE codici[N_SORGENTI];
27     /*contiene la distribuzione dei pesi delle parole di codice*/
28     int dist_pesi[N+1];
29     /*coset leader per la decodifica ML*/
30     int coset_md[N_PARITA][N];
31     /*coset leader per la decodifica UCL*/
32     int coset_ucl[N_PARITA][N];
33     /*distribuzione dei pesi per i coset in decodifica ML*/
34     int pesi_md[N+1];
35     /*distribuzione dei pesi per i coset in decodifica UCL*/
36     int pesi_ucl[N+1];
37     /*distribuzione dei pesi per il calcolo della probabilita' di
38      * errore sul bit in decodifica ML e UCL*/
39     int B_md[N+1];
40     int B_ucl[N+1];
41
42     memset(GF32, 0, (Q*M)*sizeof(int));
43     crea_elementi_GF(GF32, pol_gen);
44     stampa_elementi_GF(GF32);
45     genera_gamma(gamma);
46     genera_H(H, H_exp, GF32, pol_goppa, gamma, pol_gen);
47     stampa_H(H);
48     riduzione_echelon(echelon, H);
49     stampa_echelon(echelon);
50     genera_G(G, echelon);
51     stampa_G(G);
52     genera_codici(codici, G);
```

```
53      calcola_distribuzione_pesi(codici, dist_pesi);
54      stampa_distribuzione_pesi(dist_pesi);
55      calcola_pesi_coset(codici, pesi_md, pesi_ucl, coset_md,
56              coset_ucl);
57      stampa_pesi_coset(pesi_md, pesi_ucl);
58      stampa_coset_leader(coset_md, coset_ucl);
59      calcola_B(codici, coset_md, coset_ucl, B_ucl, B_md);
60      stampa_B(B_ucl, B_md);
61      stampa_prb_errore_sul_bit_ucl(B_ucl);
62      stampa_prb_errore_sul_bit_md(B_md);
63      stampa_prb_errore_riconosciuto(dist_pesi);
64      stampa_prb_errore_non_riconosciuto(dist_pesi);
65      stampa_prb_errore_sulla_parola_ucl(pesi_ucl);
66      stampa_prb_errore_sulla_parola_md(pesi_md);
67
68      return 0;
69  }
```

Listing 3: File `genera.h`

```
1   #ifndef __GENERA_H
2   #define __GENERA_H
3
4   #include <stdio.h>
5   #include <stdlib.h>
6   #include <string.h>
7   #include <math.h>
8   #include "goppa.h"
9   #include "utils.h"
10
11  void crea_elementi_GF(int elementi[][], int pol_gen[]);
12  void genera_H(int H[][], int H_exp[][], int elementi[][],
13          int pol_goppa[], int gamma[], int pol_generatore[]);
14  void sostituisci(int pol_goppa[], int gamma_i, int elementi[][],
15          int out[]);
16  void genera_gamma(int vet[]);
17  void euclide (int pol1[],int pol2[], int inverso[]);
18  void riduzione_echelon(int H[][], int H_exp[][], int echelon[][]);
19  void genera_G(int G[][], int echelon[][]);
20  void genera_codici(P_CODICE codici[], int G[][]);
21  void calcola_distribuzione_pesi(P_CODICE codici[], int dist_pesi[]);
22  void calcola_pesi_coset(P_CODICE codici[], int pesi_md[],
23          int pesi_ucl[], int coset_md[][], int coset_ucl[][]);
24  void calcola_B(P_CODICE codici[], int coset_md[][],
25          int coset_ucl[][], int B_ucl[], int B_md[]);
26  double calcola_probabilita(int pesi[], double prb, int start);
27
28  #endif
```

Listing 4: File `genera.c`

```c
#include "genera.h"

/*crea gli elementi del campo di Galois GF(32) come estensione di
 * GF(2)
 * [ou] elementi: matrice che contiene gli elementi di GF
 * [in] pol_gen: vettore rappresentazione del polinomio generatore*/
void crea_elementi_GF(int elementi[][M], int pol_gen[M])
{
    int i, j;

    /*inizializza i primi m elementi*/
    for(i=1, j=0; i<=M; i++){
        elementi[i][j] = 1;
        j++;
    }
    /*inizializza l'elemento m pari al polinomio generatore*/
    for(j=0; j<M; j++)
        elementi[M+1][j] = pol_gen[j];
    /*inizializza gli elementi successivi*/
    for(i=M+2; i<Q; i++){
        /*se l'esponente precedente è di ordine m...*/
        if(elementi[i-1][M-1] == 1){
            shift_destra(elementi[i-1], elementi[i]);
            somma_bit_a_bit(elementi[i], pol_gen, M);
        }
        else{
            shift_destra(elementi[i-1], elementi[i]);
        }
    }
}

/*Genera il vettore Gamma che contiene i primi 23 elementi del campo
 * GF utilizzati per la costruzione del codice di Goppa
 * [ou] vet: vettore gamma che viene creato*/
void genera_gamma(int vet[])
{
    int i;

    for(i=0; i<N; i++)
        vet[i] = i;
}

/*Genera la matrice di àparit H
 * [ou] H: matrice di parita' in forma binaria
 * [ou] H_exp: matrice di parita' in forma esponenziale
 * [in] elementi: elementi del campo GF
 * [in] pol_goppa: polinomio di goppa
 * [in] gamma: sottinsieme Gamma
 * [in] pol_gen: polinomio generatore del campo GF*/
void genera_H(int H[][N], int H_exp[][N], int elementi[][M],
        int pol_goppa[], int gamma[], int pol_generatore[])
{
```

```
53      int i, j, idx;
54      int sostituto[M+1];
55      int inverso[M+1];
56
57      for(i=0; i<T; i++){
58          for(j=0; j<N; j++){
59              if(i==0){
60                  sostituisci(pol_goppa, gamma[j], elementi,sostituto);
61                  euclide(pol_generatore, sostituto, inverso);
62                  idx = trova_in_GF(inverso, elementi);
63                  if(idx > 0)
64                      H_exp[i][j] = idx-1;
65                  if(idx == 0)
66                      H_exp[i][j] = INFTY;
67                  if(idx < 0)
68                      H_exp[i][j] = -1;
69              }
70              if(i>0)
71                  H_exp[i][j] = (H_exp[0][j]+j*i-1)%(Q-1);
72          }
73      }
74      for(i=0; i<T; i++){
75          for(j=0; j<N; j++){
76              if(H_exp[i][j] == INFTY)
77                  for(idx=0; idx<M; idx++)
78                      H[i*M+idx][j] = elementi[0][idx];
79              else{
80                  for(idx=0; idx<M; idx++)
81                      H[i*M+idx][j] = elementi[H_exp[i][j]+1][idx];
82              }
83          }
84      }
85  }
86
87  /*sostiuisce un termine gamma_i all'interno del polinomio di Goppa e
88   * ne calcola il risultato
89   * [in] pol_goppa: polinomio di goppa
90   * [in] gamma_i: elemento gamma_i
91   * [in] elementi: elementi del campo GF
92   * [ou] out: risultato della sostituzione*/
93  void sostituisci(int pol_goppa[], int gamma_i, int elementi[][M],
94          int out[])
95  {
96      int vet[(M-1)*T+1];
97      int aux[(M-1)*T+1];
98      int out_aux[(M-1)*T+1];
99      int vet_len;
100     int i, j;
101
102     vet_len = (M-1)*T+1;
103     memset(vet, 0, vet_len*sizeof(int));
104     memset(out, 0, vet_len*sizeof(int));
105     memset(aux, 0, vet_len*sizeof(int));
```

```
106     memset(out_aux, 0, vet_len*sizeof(int));
107     for(i=0; i<M; i++){
108         vet[i] = elementi[gamma_i][i];
109         aux[i] = elementi[gamma_i][i];
110     }
111     if(pol_goppa[0] == 1)
112         out_aux[0] = 1;
113     if(pol_goppa[1] == 1)
114         somma_bit_a_bit(out_aux, vet, vet_len);
115     for(i=2; i<(T+1); i++){
116         if(pol_goppa[i] == 1){
117             for(j=1; j<i; j++)
118                 moltiplica_poli(vet, aux, vet_len);
119             somma_bit_a_bit(out_aux, vet, vet_len);
120         }
121     }
122     for(i=M; i<vet_len; i++){
123         if(out_aux[i]==1){
124             memset(vet, 0, vet_len*sizeof(int));
125             for(j=0; j<M; j++)
126                 vet[j] = elementi[i+1][j];
127             somma_bit_a_bit(out_aux, vet, vet_len);
128             out_aux[i] = 0;
129         }
130     }
131     for(i=0; i<(M+1); i++)
132         out[i] = out_aux[i];
133     return;
134 }
135
136 /*implementa l'algoritmo di euclide per l'inversione di un elemento
137  * del campo GF
138  * [in] pol1: polinomio dividendo
139  * [in] pol2: polinomio divisore
140  * [ou] inverso: polinomio invertito*/
141 void euclide (int pol1[], int pol2[], int inverso[])
142 {
143     int remainder_prev1[M+1];
144     int remainder_prev2[M+1];
145     int aux_prev1[M+1];
146     int aux_prev2[M+1];
147     int grado=-1;
148     int remainder[M+1];
149     int quotient[M+1];
150     int aux[M+1];
151     int i;
152
153     memset(aux, 0, (M+1)*sizeof(int));
154     /*grado pol1  > grado pol2*/
155     memcpy(remainder_prev1, pol1, (M+1)*sizeof(int));
156     memcpy(remainder_prev2, pol2, (M+1)*sizeof(int));
157     memset(aux_prev1, 0, (M+1)*sizeof(int));
158     memset(aux_prev2, 0, (M+1)*sizeof(int));
```

```
159     aux_prev1[0]=1;
160
161     for(i=0; i<(M+1); i++)
162             if(remainder_prev2[i] ==1){
163                 grado= i;
164     }
165     while(grado>0)
166     {
167         grado=-2;
168         memset(remainder, 0, (M+1)*sizeof(int));
169         memset(quotient, 0, (M+1)*sizeof(int));
170         divisione_polinomi(remainder_prev1, remainder_prev2,
171                 quotient, remainder);
172         memcpy(aux, quotient, (M+1)*sizeof(int));
173         moltiplica_poli(aux, aux_prev1, M+1);
174         somma_bit_a_bit(aux, aux_prev2, M+1);
175         memcpy(remainder_prev1, remainder_prev2, (M+1)*sizeof(int));
176         memcpy(remainder_prev2, remainder, (M+1)*sizeof(int));
177         memcpy(aux_prev2, aux_prev1, (M+1)*sizeof(int));
178         memcpy(aux_prev1, aux, (M+1)*sizeof(int));
179         for(i=1; i<M+1; i++)
180             if(remainder[i] ==1){
181                 grado= i;
182                 break;
183             }
184     }
185     /*se pol2 è il polinomio nullo*/
186     if(grado == -1)
187         memset(inverso, 0, (M+1)*sizeof(int));
188     /*se pol2 è il polinomio 1*/
189     else if(grado == 0){
190         memset(inverso, 0, (M+1)*sizeof(int));
191         inverso[0] = 1;
192     }
193     else
194         memcpy(inverso, aux, (M+1)*sizeof(int));
195     return;
196 }
197
198 /*Riduce la matrice di parita' in forma echelon
199  * [in] H: matrice di parita' in forma binaria
200  * [ou] echelon: matrice di parita' in forma echelon*/
201 void riduzione_echelon(int echelon[][N], int H[][N])
202 {
203     int i, j, k, m, n;
204     int colonna_pivot;
205     int H_copia[T*M][N];
206     int tmp[N];
207     int passo;
208     int quanti = 0;
209
210     /*inizializzazioni*/
211     memset(echelon, 0, T*M*N*sizeof(int));
```

```
212     memcpy(H_copia, H, T*M*N*sizeof(int));
213     passo=0;
214
215     while(passo < T*M){
216         colonna_pivot = -1;
217         /*cerco la colonna pivot*/
218         for(j=passo; j<N && colonna_pivot<0; j++)
219             for(i=passo; i<T*M && colonna_pivot<0; i++)
220                 if(H_copia[i][j]!=0)
221                     colonna_pivot = j;
222
223         /*se l'elemento piu' in alto della colonna pivot non e'
224          * diverso da zero, effettuo scambi di righe fino a quando
225          * questa condizione non e' verificata*/
226         i = passo+1;
227         while(H_copia[passo][colonna_pivot] == 0 && i<T*M){
228                 for(k=0; k<N; k++)
229                     tmp[k] = H_copia[passo][k];
230                 for(k=0; k<N; k++)
231                     H_copia[passo][k] = H_copia[i][k];
232                 for(k=0; k<N; k++)
233                     H_copia[i][k] = tmp[k];
234                 i++;
235         }
236         /*sommo bit a bit le righe al di sotto del pivot che iniziano
237          * con 1 ìcos che tutti gli elementi sotto al pivot diventino
238          * nulli*/
239         for(i=passo+1; i<T*M; i++)
240             if(H_copia[i][passo] == 1)
241                 for(j=passo; j<N; j++)
242                     H_copia[i][j] =
243                         (H_copia[i][j])exor(H_copia[passo][j]);
244         passo++;
245     }
246
247     /*A partire dall'ultimo pivot, azzero gli elementi che gli sono
248      * sopra nella colonna con operazioni di somma bit a bit tra le
249      * righe*/
250     for(i=(T*M)-1; i>=0; i--){
251         for(j=0; j<N && H_copia[i][j] == 0; j++)
252             ;
253         for(k=i-1; k>=0; k--)
254             if(H_copia[k][j] == 1){
255                 for(m=0; m<N; m++)
256                     H_copia[k][m] =
257                         (H_copia[k][m]) exor (H_copia[i][m]);
258             }
259     }
260
261     /*Se una colonna ha un solo 1, in tmp ne viene salvata la
262      * posizione, altrimenti viene scritto -1*/
263     memset(tmp, -2, N*sizeof(int));
264     for(j=0; j<N; j++){
```

```
265             quanti=0;
266             for(i=0; i<(T*M); i++){
267                 if(H_copia[i][j] == 1){
268                     if(quanti > 0){
269                         tmp[j] = -1;
270                         break;
271                     }
272                     quanti++;
273                     tmp[j] = i;
274                 }
275             }
276         }
277 
278         /*contrassegna con -3 le colonne che sono uguali ad altre per
279          * escluderle da calcoli successivi*/
280         for(i=0; i<N; i++)
281             if(tmp[i] >= 0)
282                 for(j=i+1; j<N; j++)
283                     if(tmp[i] == tmp[j])
284                         tmp[j] = -3;
285 
286         /*scambia gli indici delle colonne in tmp in modo da memorizzare
287          * l'ordine in cui vanno riordinate le colonne della matrice*/
288         for(i=0; i<(T*M); i++)
289             if(tmp[i]!=i){
290                 for(j=i+1; j<N; j++)
291                     if(tmp[j]==i){
292                         k = tmp[i];
293                         tmp[i] = tmp[j];
294                         tmp[j] = k;
295                         k = tmp[j];
296                         for(m=0; m<(T*M); m++){
297                             n = H_copia[m][j];
298                             H_copia[m][j] = H_copia[m][i];
299                             H_copia[m][i] = n;
300                         }
301                         break;
302                     }
303             }
304 
305         memcpy(echelon, H_copia, T*M*N*sizeof(int));
306 }
307 
308 /*genera la matrice generatrice a partire dalla matrice echelon
309  * [ou] G: matrice che viene generata
310  * [in] echelon: matrice echelon da cui partire per la generazione
311  * della matrice*/
312 void genera_G(int G[][K], int echelon[][N])
313 {
314         int i, j;
315 
316         for(i=0; i<T*M; i++)
317             for(j=0; j<K; j++)
```

```
318                 G[i][j] = echelon[i][j+T*M];
319         for(i=0; i<K; i++)
320             for(j=0; j<N; j++){
321                 if(i==j)
322                     G[i+T*M][j] = 1;
323                 else
324                     G[i+T*M][j] = 0;
325             }
326 }
327
328 /*Per tutte le possibili parole di sorgente su 13 bit genera una
329  * struttura dati contenente la parola di sorgente, la parola di
330  * sorgente codificata e il peso
331  * [ou] codici: vettore contenente le informazioni per ogni parola di
332  *       sorgente
333  * [in] G: matrice generatrice del codice*/
334 void genera_codici(P_CODICE codici[N_SORGENTI], int G[][K])
335 {
336     int num, tmp;
337     int i, j;
338
339     for(num=0; num<N_SORGENTI; num++){
340         memset(codici[num].sorgente, 0, K*sizeof(int));
341         memset(codici[num].codice, 0, N*sizeof(int));
342         codici[num].peso_src = 0;
343         codici[num].peso_cod = 0;
344         tmp = num;
345         /*converte num in binario per generare una parola di
346          * sorgente*/
347         for(i=0; i<K && tmp > 0; i++){
348             codici[num].sorgente[i] = tmp%2;
349             tmp = (int) tmp/2;
350         }
351         /*calcola il peso della sorgente contandone il numero di 1*/
352         tmp=0;
353         for(i=0; i<K; i++)
354             if(codici[num].sorgente[i] == 1)
355                 tmp++;
356         codici[num].peso_src = tmp;
357         /*codifica la parola di sorgente*/
358         for(i=0; i<N; i++)
359             for(j=0; j<K; j++)
360                 codici[num].codice[i] =
361                     (codici[num].codice[i] exor
362                      (G[i][j]*codici[num].sorgente[j]));
363         /*calcola il peso del codice contandone il numero di 1*/
364         tmp = 0;
365         for(i=0; i<N; i++)
366             if(codici[num].codice[i] == 1)
367                 tmp++;
368         codici[num].peso_cod = tmp;
369     }
370 }
```

```
371
372 /*Calcola la distribuzione dei pesi per ogni parola di codice
373  * [in] codici: vettore contenente le parole di sorgente e relativi
374  * codice e peso
375  * [ou] dist_pesi: vettore delle distribuzione pesi delle parole di
376  * codice*/
377 void calcola_distribuzione_pesi(P_CODICE codici[N_SORGENTI],
378         int dist_pesi[N+1])
379 {
380     int i;
381
382     memset(dist_pesi, 0, (N+1)*sizeof(int));
383     for(i=0; i<N_SORGENTI; i++)
384         dist_pesi[codici[i].peso_cod] += 1;
385 }
386
387 /*Calcola i coset leader e la distribuzione dei pesi dei coset sia
388  * nella codifica ML sia nella codifica UCL
389  * [in] codici: vettore contenente le parole di sorgente e relativo
390  * codice e peso
391  * [ou] pesi_md: vettore con la distribuzione dei pesi del coset
392  * leader per la decodifica ML
393  * [ou] pesi_ucl: vettore con la distribuzione dei pesi del coset
394  * leader per la decodifica UCL
395  * [ou] coset_md: vettore con i coset leader per la decodifica ML
396  * [ou] coset_ucl: vettore con i coset leader per decodifica UCL*/
397 void calcola_pesi_coset(P_CODICE codici[N_SORGENTI],
398         int pesi_md[N+1], int pesi_ucl[N+1], int coset_md[][N],
399         int coset_ucl[][N])
400 {
401     int i, j, k;
402     int codice[N];
403     int coset[N];
404     int pc, pcst;
405     int peso_minimo_md, peso_minimo_ucl;
406
407     memset(pesi_md, 0, (N+1)*sizeof(int));
408     memset(pesi_ucl, 0, (N+1)*sizeof(int));
409
410     pc = pcst = 0;
411     for(i=0; i<N_PARITA; i++){
412         /*crea la parola di codice con i 13 bit di informazione a 0*/
413         memcpy(codice, codici[i].sorgente, K*sizeof(int));
414         for(j=K; j<N; j++)
415             codice[j] = 0;
416         pc = codici[i].peso_src;
417         peso_minimo_md = peso_minimo_ucl = INFTY;
418         /*genera le possibili parole di sorgente e le somma ai bit di
419          * parita*/
420         for(j=0; j<N_SORGENTI; j++){
421             memcpy(coset, codici[j].codice, N*sizeof(int));
422             somma_bit_a_bit(coset, codice, N);
423             pcst = 0;
```

```
424             for(k=0; k<N; k++)
425                 if(coset[k] == 1)
426                     pcst++;
427             if(pcst <= T){
428                 peso_minimo_md = pcst;
429                 peso_minimo_ucl = pcst;
430                 j = N_SORGENTI;
431                 memcpy(coset_md[i], coset, N*sizeof(int));
432                 memcpy(coset_ucl[i], coset, N*sizeof(int));
433             }
434             else if(pcst < peso_minimo_md){
435                 peso_minimo_md = pcst;
436                 peso_minimo_ucl = pc;
437                 memcpy(coset_md[i], coset, N*sizeof(int));
438                 memcpy(coset_ucl[i], codice, N*sizeof(int));
439             }
440         }
441         pesi_md[peso_minimo_md]++;
442         pesi_ucl[peso_minimo_ucl]++;
443     }
444 }
445
446 /*Calcola il vettore dei pesi B da utilizzare per il calcolo della
447  * probabilita' di errore sul bit nella decodifica UCL e ML
448  * [in] codici: vettore contenente le parole di sorgente e relativi
449  *      codice e peso
450  * [in] coset_md: vettore contenente i coset leader per la decodifica
451  *      ML
452  * [in] coset_ucl: vettore contenente i coset leader per la
453  *      decodifica UCL
454  * [ou] B_ucl: vettore con la distribuzione dei pesi per il calcolo
455  *      dell'errore sul bit nel caso di decodifica UCL
456  * [ou] B_md: vettore con la distribuzione dei pesi per il calcolo
457  *      dell'errore sul bit nel caso di decodifica ML*/
458 void calcola_B(P_CODICE codici[N_SORGENTI], int coset_md[][N],
459         int coset_ucl[][N], int B_ucl[N+1], int B_md[N+1])
460 {
461     int i, j, k;
462     int incremento;
463     int codice1[N], codice2[N];
464     int peso1, peso2;
465
466     memset(B_ucl, 0, (N+1)*sizeof(int));
467     memset(B_md, 0, (N+1)*sizeof(int));
468
469     for(i=0; i<N_PARITA; i++){
470         for(j=0; j<N_SORGENTI; j++){
471             incremento = codici[j].peso_src;
472             memcpy(codice1, codici[j].codice, N*sizeof(int));
473             memcpy(codice2, codici[j].codice, N*sizeof(int));
474             somma_bit_a_bit(codice1, coset_md[i], N);
475             somma_bit_a_bit(codice2, coset_ucl[i], N);
476             peso1 = peso2 = 0;
```

```
477             for(k=0; k<N; k++){
478                 if(codice1[k] == 1)
479                     peso1++;
480                 if(codice2[k] == 1)
481                     peso2++;
482             }
483             B_md[peso1] += incremento;
484             B_ucl[peso2] += incremento;
485         }
486     }
487 }
488
489 double calcola_probabilita(int pesi[N+1], double prb, int start)
490 {
491     int i;
492     double risultato = 0;
493
494     for(i=start; i<(N+1); i++)
495         risultato += ((double)pesi[i])*(pow((double)prb,
496                     (double)i))*(pow((double)(1-prb),
497                                     (double)(N-i)));
498     return risultato;
499 }
```

Listing 5: File `utils.h`

```
1  #ifndef __UTILS_H
2  #define __UTILS_H
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <string.h>
7  #include <math.h>
8  #include "goppa.h"
9  #include "genera.h"
10
11 void shift_destra(int in[], int out[]);
12 void somma_bit_a_bit(int vet1[], int vet2[], int dim);
13 void divisione_polinomi(int dividendo[], int divisore[],
14         int quoziente[], int resto[]);
15 void moltiplica_poli(int poli1[], int poli2[], int dim);
16 void stampa_vettore(int vet[], int dim, char *);
17 int trova_in_GF(int vet[], int elementi[][]);
18 void stampa_elementi_GF(int elementi[][]);
19 void stampa_H(int H[][]);
20 void stampa_echelon(int echelon[][]);
21 void stampa_G(int G[][]);
22 void carica_dati(int H[][], int G[][], int coset[][]);
23 void carica_input(int sorgente[], int errore[]);
24 void stampa_codice(int codice[]);
25 void stampa_distribuzione_pesi(int dist_pesi[]);
26 void stampa_pesi_coset(int pesi_md[], int pesi_ucl[]);
27 void stampa_coset_leader(int coset_md[][], int coset_ucl[][]);
```

```
28 void stampa_decodifica(int parola1[], int parola2[], int coset[],
29         int errori);
30 void stampa_B(int B_ucl[], int B_md[]);
31 void stampa_prb_errore_sul_bit_ucl(int B_ucl[]);
32 void stampa_prb_errore_sul_bit_md(int B_md[]);
33 void stampa_prb_errore_riconosciuto(int pesi[]);
34 void stampa_prb_errore_non_riconosciuto(int pesi[]);
35 void stampa_prb_errore_sulla_parola_ucl(int pesi_ucl[]);
36 void stampa_prb_errore_sulla_parola_md(int pesi_md[]);
37
38 #endif
```

Listing 6: File `utils.c`

```
1  #include "utils.h"
2
3  /* Effettua lo shift a destra del contenuto di un vettore
4   * [in] in: vettore il cui contenuto va shiftato
5   * [ou] out: vettore risultato dello shift */
6  void shift_destra(int in[], int out[])
7  {
8      int i;
9
10     for(i=M; i>0; i--)
11         out[i] = in[i-1];
12     out[0] = 0;
13 }
14
15 /* Somma bit a bit due polinomi rappresentati tramite vettore
16  * [io] vet1: primo addendo, modificato dal risultato
17  * [in] vet2: secondo addendo
18  * [in] dim: lunghezza dei due vettori */
19 void somma_bit_a_bit(int vet1[], int vet2[], int dim)
20 {
21     int i;
22
23     for(i=0; i<dim; i++)
24         vet1[i] = vet1[i] exor vet2[i];
25 }
26
27 /* Calcola in algebra modulo 2 la divisione di due polinomi
28  * [in] dividendo: polinomio da dividere
29  * [in] divisore: polinomio divisore
30  * [ou] quoziente: polinomio risultato della divisione
31  * [ou] resto: eventuale resto della divisione */
32 void divisione_polinomi(int dividendo[], int divisore[],
33         int quoziente[], int resto[])
34 {
35     int i;
36     int grado_dividendo = 0;
37     int grado_divisore = 0;
38     int termine_corrente;
39     int tmp1[M+1];
```

```
40      int tmp2[M+1];
41
42      memset(tmp1, 0, (M+1)*sizeof(int));
43      memset(tmp2, 0, (M+1)*sizeof(int));
44      for(i=0; i<(M+1); i++)
45          if(dividendo[i] == 1 && i>grado_dividendo)
46              grado_dividendo = i;
47      for(i=0; i<(M+1); i++)
48          if(divisore[i] == 1 && i>grado_divisore)
49              grado_divisore = i;
50      if(grado_dividendo >= grado_divisore){
51          termine_corrente = grado_dividendo - grado_divisore;
52          quoziente[termine_corrente] = 1;
53          memcpy(tmp1, dividendo, (M+1)*sizeof(int));
54          for(i=0; i<(M+1-termine_corrente); i++)
55              tmp2[i+termine_corrente] = divisore[i];
56          somma_bit_a_bit(tmp1, tmp2, M+1);
57          memcpy(dividendo, tmp1, (M+1)*sizeof(int));
58          divisione_polinomi(dividendo, divisore, quoziente, resto);
59      }
60      if(grado_dividendo < grado_divisore)
61      {
62          memcpy(resto, dividendo, (M+1)*sizeof(int));
63      }
64      return;
65  }
66
67  /*Esegue la moltiplicazione tra due polinomi in modulo 2
68   * [io] poli1: primo polinomio moltiplicatore, sovrascritto dal
69   * risultato
70   * [in] poli2: secondo polinomio moltiplicatore
71   * [in] dim: dimensione dei polinomi moltiplicatori*/
72  void moltiplica_poli(int poli1[], int poli2[], int dim)
73  {
74      int i, j;
75      int *out;
76
77      out = (int *)malloc(dim*sizeof(int));
78      memset(out, 0, dim*sizeof(int));
79      for(i=0; i<dim; i++)
80          for(j=0; j<dim; j++)
81              if(poli1[i]!=0 && poli2[j]!=0 && i+j < dim)
82                  out[i+j] = out[i+j] exor 1;
83      memcpy(poli1, out, dim*sizeof(int));
84  }
85
86  /*Stampa il contenuto del vettore a schermo. Funzione utilizzata
87   * nel debug
88   * [in] vet: vettore da stampare
89   * [in] dim: dimensione del vettore da stampare
90   * [in] nome del file su cui si vuole stampare*/
91  void stampa_vettore(int vet[], int dim, char *file)
92  {
```

```
93      int i;
94      FILE *fp;
95
96      if(!strcmp("stdout", file))
97          fp = stdout;
98      else if((fp = fopen(file, "a"))==NULL)
99          exit(ERROR_FOPEN);
100     for(i=0; i<dim; i++)
101         fprintf(fp, "%d ", vet[i]);
102     fprintf(fp, "\n");
103     if(strcmp("stdout", file))
104         fclose(fp);
105 }
106
107 /*Cerca se una sequenza binaria ha una corrispondenza all'interno
108  * degli elementi di GF(32)
109  * [in] vet: vettore che va ricercato
110  * [in] elementi: elementi del campo in cui fare la ricerca
111  * restituisce l'indice all'interno di elementi a cui si trova
112  * l'elemento, -1 se non lo trova*/
113 int trova_in_GF(int vet[], int elementi[][M])
114 {
115     int i;
116
117     i = -1;
118     for(i=0; i<Q; i++){
119         if(!memcmp(vet, elementi[i], M*sizeof(int)))
120             return i;
121     }
122     return i;
123 }
124
125 /*Stampa su file gli elementi del campo GF32
126  * [in] elementi: matrice contenente gli elementi del campo*/
127 void stampa_elementi_GF(int elementi[][M])
128 {
129     int i, j;
130     FILE *fp;
131
132     fp = fopen("dati/elementi_GF.txt", "w");
133     if(fp == NULL)
134         exit(ERROR_FOPEN);
135     for(i=0; i<Q; i++){
136         for(j=0; j<M; j++)
137             fprintf(fp, "%d ", elementi[i][j]);
138         fprintf(fp, "\n");
139     }
140     fclose(fp);
141 }
142
143 /*Stampa su file la matrice di parita'
144  * [in] H: matrice di parita' da stampare*/
145 void stampa_H(int H[][N])
```

```
146 {
147     int i, j;
148     FILE *fp;
149 
150     fp = fopen("dati/H.txt", "w");
151     if(fp == NULL)
152         exit(ERROR_FOPEN);
153     for(i=0; i<T*M; i++){
154         for(j=0; j<N; j++)
155             fprintf(fp, "%d ", H[i][j]);
156         fprintf(fp, "\n");
157     }
158     fclose(fp);
159 }
160 
161 /*Stampa su file la matrice H in forma echelon
162  * [in] echelon: matrice da stampare*/
163 void stampa_echelon(int echelon[][N])
164 {
165     int i, j;
166     FILE *fp;
167 
168     fp = fopen("dati/echelon.txt", "w");
169     if(fp == NULL)
170         exit(ERROR_FOPEN);
171     for(i=0; i<T*M; i++){
172         for(j=0; j<N; j++)
173             fprintf(fp, "%d ", echelon[i][j]);
174         fprintf(fp, "\n");
175     }
176     fclose(fp);
177 }
178 
179 /*Stampa su file la matrice generatrice G
180  * [in] G: matrice da stampare*/
181 void stampa_G(int G[][K])
182 {
183     int i, j;
184     FILE *fp;
185 
186     fp = fopen("dati/G.txt", "w");
187     if(fp == NULL)
188         exit(ERROR_FOPEN);
189     for(i=0; i<N; i++){
190         for(j=0; j<K; j++)
191             fprintf(fp, "%d ", G[i][j]);
192         fprintf(fp, "\n");
193     }
194 }
195 
196 /*Carica da file le informazioni necessarie
197  * [ou] H: matrice di parita' in forma echelon
198  * [ou] G: matrice generatrice*/
```

```
199 void carica_dati(int H[][N], int G[][K], int coset_ucl[][N])
200 {
201     int i, j;
202     int dato;
203     FILE *fp;
204     char buffer[100];
205 
206     fp = fopen("dati/echelon.txt", "r");
207     if(fp == NULL)
208         exit(ERROR_FOPEN);
209     for(i=0; i<T*M; i++)
210         for(j=0; j<N; j++){
211             fscanf(fp, "%d", &dato);
212             if(dato!=0 && dato!=1)
213                 exit(ERROR_DATA);
214             H[i][j] = dato;
215         }
216     fclose(fp);
217     fp = fopen("dati/G.txt", "r");
218     if(fp == NULL)
219         exit(ERROR_FOPEN);
220     for(i=0; i<N; i++)
221         for(j=0; j<K; j++){
222             fscanf(fp, "%d", &dato);
223             if(dato!=0 && dato!=1)
224                 exit(ERROR_DATA);
225             G[i][j] = dato;
226         }
227     fclose(fp);
228     fp = fopen("dati/coset_ucl.txt", "r");
229     if(fp == NULL)
230         exit(ERROR_FOPEN);
231     i = 0;
232     while(fgets(buffer, 100, fp)!=NULL){
233         for(j=0; j<N; j++){
234             dato = buffer[j] - '0';
235             if(dato!=0 && dato!=1){
236                 exit(ERROR_DATA);
237             }
238             coset_ucl[i][j] = dato;
239         }
240         i++;
241     }
242     fclose(fp);
243 }
244 
245 /*Carica da file l'input del programma di simulazione del canale
246  * [ou] sorgente: parola di sorgente
247  * [ou] prob: probabilita' di errore
248  * [ou] errore: parola di errore*/
249 void carica_input(int sorgente[K], int errore[N])
250 {
251     int i;
```

```
252     int dato;
253     FILE *fp;
254 
255     fp = fopen("input.txt", "r");
256     if(fp==NULL)
257         exit(ERROR_FOPEN);
258     for(i=0; i<K; i++){
259         fscanf(fp, "%d", &dato);
260         if(dato!=0 && dato!=1)
261             exit(ERROR_DATA);
262         sorgente[i] = dato;
263     }
264     for(i=0; i<N; i++){
265         fscanf(fp, "%d", &dato);
266         if(dato!=0 && dato!=1)
267             exit(ERROR_DATA);
268         errore[i] = dato;
269     }
270     fclose(fp);
271 }
272 
273 /*Stampa su file il risultato della codifica
274  * [in] codice: parola di codice da stampare*/
275 void stampa_codice(int codice[N])
276 {
277     int i;
278     FILE *fp;
279 
280     fp = fopen("output/codice.txt", "w");
281     if(fp == NULL)
282         exit(ERROR_FOPEN);
283     for(i=0; i<N; i++)
284         fprintf(fp, "%d ", codice[i]);
285     fclose(fp);
286 }
287 
288 /*Stampa su file la distribuzione dei pesi delle parole di codice
289  * [in] vettore della distribuzione dei pesi da stampare*/
290 void stampa_distribuzione_pesi(int dist_pesi[N+1])
291 {
292     int i;
293     FILE *fp;
294 
295     fp = fopen("dati/pesi.txt", "w");
296     if(fp==NULL)
297         exit(ERROR_FOPEN);
298     for(i=0; i<N+1; i++)
299         fprintf(fp, "%d\n", dist_pesi[i]);
300     fclose(fp);
301 }
302 
303 /*Stampa su file la distribuzione dei pesi dei coset leader in
304  * codifica MD e UCL
```

```
305  * [in] pesi_md: distribuzione dei pesi dei coset leader MD
306  * [in] pesi_ucl: distribuzione dei pesi dei coset leader UCL*/
307 void stampa_pesi_coset(int pesi_md[N+1], int pesi_ucl[N+1])
308 {
309     int i;
310     FILE *fp;
311
312     fp = fopen("dati/pesi_md.txt", "w");
313     if(fp == NULL)
314         exit(ERROR_FOPEN);
315     for(i=0; i<N+1; i++)
316         fprintf(fp, "%d\n", pesi_md[i]);
317     fclose(fp);
318     fp = fopen("dati/pesi_ucl.txt", "w");
319     if(fp==NULL)
320         exit(ERROR_FOPEN);
321     for(i=0; i<N+1; i++)
322         fprintf(fp, "%d\n", pesi_ucl[i]);
323     fclose(fp);
324 }
325
326 /*Stampa su file i coset leader per la decodifica ML e UCL
327  * [in] coset_md: coset leader per la decodifica ML
328  * [in] coset_ucl: coset leader per la decoficia UCL*/
329 void stampa_coset_leader(int coset_md[][N], int coset_ucl[][N])
330 {
331     int i, j;
332     FILE *fp1, *fp2;
333
334     fp1 = fopen("dati/coset_md.txt", "w");
335     fp2 = fopen("dati/coset_ucl.txt", "w");
336     if(fp1==NULL || fp2==NULL)
337         exit(ERROR_FOPEN);
338     for(i=0; i<N_PARITA; i++){
339         for(j=0; j<N; j++){
340             fprintf(fp1, "%d", coset_md[i][j]);
341             fprintf(fp2, "%d", coset_ucl[i][j]);
342         }
343         fprintf(fp1, "\n");
344         fprintf(fp2, "\n");
345     }
346     fclose(fp1);
347     fclose(fp2);
348 }
349
350 /*Stampa su file il risultato della decodifica
351  * [in] parola: parola ottenuta dal decodificatore UCL
352  * [in] coset: coset utilizzato per la decodifica
353  * [in] errori: numero di errori rilevati*/
354 void stampa_decodifica(int parola1[N], int parola2[K], int coset[N],
355         int errori)
356 {
357     int i;
```

```
358     FILE *fp;
359
360     fp = fopen("output/decodifica_ucl.txt", "w");
361     if(fp == NULL)
362         exit(ERROR_FOPEN);
363     for(i=0; i<N; i++)
364         fprintf(fp, "%d", parola1[i]);
365     fprintf(fp, "\n");
366     for(i=0; i<K; i++)
367         fprintf(fp,"%d", parola2[i]);
368     fprintf(fp,"\n");
369     for(i=0; i<N; i++)
370             fprintf(fp,"%d", coset[i]);
371     fprintf(fp,"\n%d", errori);
372     fclose(fp);
373 }
374
375 /*Stampa su file la distribuzione dei pesi per il calcolo della
376  * probabilita' di errore sul bit nella decodifica MD e UCL
377  * [in] B_ucl: vettore con la distribuzione dei pesi per la
378  * decodifica UCL
379  * [in] B_md: vettore con la distribuzione dei pesi per la decodifica
380  * MD*/
381 void stampa_B(int B_ucl[N+1], int B_md[N+1])
382 {
383     int i;
384     FILE *fp;
385
386     fp = fopen("dati/B_ucl.txt", "w");
387     if(fp == NULL)
388         exit(ERROR_FOPEN);
389     for(i=0; i<N+1; i++)
390         fprintf(fp, "%d\n", B_ucl[i]);
391     fclose(fp);
392
393     fp = fopen("dati/B_md.txt", "w");
394     if(fp == NULL)
395         exit(ERROR_FOPEN);
396     for(i=0; i<N+1; i++)
397         fprintf(fp, "%d\n", B_md[i]);
398     fclose(fp);
399 }
400
401 /*Stampa su file i valori per plottare il grafico che rappresenta
402  * l'andamento della probabilita' di errore sul bit in decodifica UCL
403  * [in] B_ucl: vettore con la distribuzione dei pesi*/
404 void stampa_prb_errore_sul_bit_ucl(int B_ucl[N+1])
405 {
406     int i;
407     double p, P_b;
408     FILE *fp;
409
410     fp = fopen("plot/pb_ucl.dat", "w");
```

```
411     if(fp==NULL)
412         exit(ERROR_FOPEN);
413     for(i=0; i<100; i++){
414         p = (double) i/100;
415         P_b = calcola_probabilita(B_ucl, p, 0);
416         P_b = P_b / K;
417         fprintf(fp, "%2f, %2f\n", p, P_b);
418     }
419     fclose(fp);
420 }
421
422 /*Stampa su file i valori per plottare il grafico che rappresenta
423  * l'andamento della probabilita' di errore sul bit in decodifica MD
424  * [in] B_md: vettore con la distribuzione dei pesi*/
425 void stampa_prb_errore_sul_bit_md(int B_md[N+1])
426 {
427     int i;
428     double p, P_b;
429     FILE *fp;
430
431     fp = fopen("plot/pb_md.dat", "w");
432     if(fp==NULL)
433         exit(ERROR_FOPEN);
434     for(i=0; i<100; i++){
435         p = (double) i/100;
436         P_b = calcola_probabilita(B_md, p, 0);
437         P_b = P_b / K;
438         fprintf(fp, "%2f, %2f\n", p, P_b);
439     }
440     fclose(fp);
441 }
442
443 /*Stampa su file i valori per plottare il grafico che rappresenta
444  * l'andamento della probabilita' di rilevare l'errore
445  * [in] pesi: vettore con la distribuzione dei pesi*/
446 void stampa_prb_errore_riconosciuto(int pesi[N+1])
447 {
448     int i;
449     double p, P_r;
450     FILE *fp;
451
452     fp = fopen("plot/pr.dat", "w");
453     if(fp==NULL)
454         exit(ERROR_FOPEN);
455     for(i=0; i<100; i++){
456         p = (double) i/100;
457         P_r = calcola_probabilita(pesi, p, 0);
458         P_r = 1-P_r;
459         fprintf(fp, "%2f, %2f\n", p, P_r);
460     }
461     fclose(fp);
462 }
463
```

```
464 /*Stampa su file i valori per plottare il grafico che rappresenta
465  * l'andamento della probabilita' di non rilevare l'errore
466  * [in] pesi: vettore con la distribuzione dei pesi*/
467 void stampa_prb_errore_non_riconosciuto(int pesi[N+1])
468 {
469     int i;
470     double p, P_u;
471     FILE *fp;
472 
473     fp = fopen("plot/pu.dat", "w");
474     if(fp==NULL)
475         exit(ERROR_FOPEN);
476     for(i=0; i<100; i++){
477         p = (double) i/100;
478         P_u = calcola_probabilita(pesi, p, 1);
479         fprintf(fp, "%2f, %2f\n", p, P_u);
480     }
481     fclose(fp);
482 }
483 
484 /*Stampa su file i valori per plottare il grafico che rappresenta
485  * l'andamento della probabilita' di errore sulla parola in
486  * decodifica UCL
487  * [in] pesi_ucl: vettore con la distribuzione dei pesi*/
488 void stampa_prb_errore_sulla_parola_ucl(int pesi_ucl[N+1])
489 {
490     int i;
491     double p, P_w;
492     FILE *fp;
493 
494     fp = fopen("plot/pw_ucl.dat", "w");
495     if(fp==NULL)
496         exit(ERROR_FOPEN);
497     for(i=0; i<100; i++){
498         p = (double) i/100;
499         P_w = calcola_probabilita(pesi_ucl, p, 0);
500         P_w = 1 - P_w;
501         fprintf(fp, "%2f, %2f\n", p, P_w);
502     }
503     fclose(fp);
504 }
505 
506 /*Stampa su file i valori per plottare il grafico che rappresenta
507  * l'andamento della probabilita' di errore sulla parola in
508  * decodifica MD
509  * [in] pesi_ucl: vettore con la distribuzione dei pesi*/
510 void stampa_prb_errore_sulla_parola_md(int pesi_md[N+1])
511 {
512     int i;
513     double p, P_w;
514     FILE *fp;
515 
516     fp = fopen("plot/pw_md.dat", "w");
```

```
517     if(fp==NULL)
518         exit(ERROR_FOPEN);
519     for(i=0; i<100; i++){
520         p = (double) i/100;
521         P_w = calcola_probabilita(pesi_md, p, 0);
522         P_w = 1 - P_w;
523         fprintf(fp, "%2f, %2f\n", p, P_w);
524     }
525     fclose(fp);
526 }
```

Listing 7: File `simulatore_main.c`

```
1  #include <stdio.h>
2  #include "simulatore.h"
3
4  int main(void)
5  {
6      /*matrice di parita' in forma echelon*/
7      int echelon[T*M][N];
8      /*matrice generatrice del codice*/
9      int G[N][K];
10     /*parola di sorgente*/
11     int sorgente[K];
12     /*parola di errore*/
13     int errore[N];
14     /*parola di codice*/
15     int codice[N];
16     /*parola ricevuta dal decodificatore*/
17     int ricevuto[N];
18     /*coset leader per la decodifica UCL*/
19     int coset_ucl[N_PARITA][N];
20     /*numero di errori rilevati nella decodifica*/
21     int errori;
22     /*parola decodificata dal decodificatore UCL*/
23     int decodificato[K];
24     /*indice del coset utilizzato per la decodifica*/
25     int coset_idx;
26     /*distribuzione dei pesi per i coset in decodifica ML*/
27
28     carica_dati(echelon, G, coset_ucl);
29     carica_input(sorgente, errore);
30     codifica(sorgente, G, codice);
31     stampa_codice(codice);
32     applica_errore(ricevuto, codice, errore);
33     errori = decodificatore_ucl(G, echelon, ricevuto, coset_ucl,
34             decodificato, &coset_idx);
35     stampa_decodifica(ricevuto, decodificato, coset_ucl[coset_idx],
36             errori);
37
38     return 0;
39 }
```

Listing 8: File `simulatore.h`

```
#ifndef __SIMULATORE_H
#define __SIMULATORE_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "goppa.h"
#include "utils.h"
#include "genera.h"

void codifica(int sorgente[], int G[][], int codice[]);
void applica_errore(int ricevuto[], int codice[], int errore[]);
int decodificatore_ucl(int G[][], int echelon[][], int ricevuto[],
        int coset_ucl[][], int decodificato[], int *coset_idx);

#endif
```

Listing 9: File `simulatore.c`

```
#include "simulatore.h"

/* Effettua la codifica di una parola di codice
 * [in] sorgente: parola di sorgente da codificare
 * [in] G: matrice generatrice del codice
 * [ou] codice: parola di codice generata */
void codifica(int sorgente[K], int G[][K], int codice[N])
{
    int i, j;

    memset(codice, 0, N*sizeof(int));
    for(i=0; i<N; i++)
        for(j=0; j<K; j++)
            codice[i] = (codice[i] exor (G[i][j]*sorgente[j]));
}

/* Aggiunge alla parola di codice trasmessa dal codificatore l'errore
 * [ou] ricevuto: parola ottenuta al ricevitore = codice + errore
 * [in] codice: parola di codice elaborata dal codificatore
 * [in] errore introdotto dal canale */
void applica_errore(int ricevuto[N], int codice[N], int errore[N])
{
    int i;

    for(i=0; i<N; i++)
        ricevuto[i] = codice[i] exor errore[i];
}

/* Simula il decodificatore UCL
 * [in] G: matrice generatrice del codice
 * [in] echelon: matrice di parita' in forma echelon
 * [in] ricevuto: parola ricevuta dal decodificatore = codice+errore
```

```
33  * [in] coset_ucl: coset leader per la decodifica UCL
34  * [ou] decodificato: parola decodificata dal decodificatore
35  * [ou] coset_idx: indice del coset utilizzato per la decodifica
36  * restituisce il numero di errori rilevati durante la decodifica*/
37 int decodificatore_ucl(int G[][K], int echelon[][N], int ricevuto[N],
38         int coset_ucl[][N], int decodificato[K], int *coset_idx)
39 {
40
41     int i, j;
42     int cst, ps, peso;
43     int decodifica[N];
44     int sindrome[T*M];
45     int possibile_sindrome[T*M];
46
47     memset(sindrome,0,T*M*sizeof(int));
48     ps = 0;
49
50     /*calcola la sindrome*/
51     for(i=0; i<T*M; i++)
52         for(j=0;j<N;j++)
53             sindrome[i] = (sindrome[i]
54                     exor (echelon[i][j]*ricevuto[j]));
55     /*conta il numero di 1 nella sindrome*/
56     for(i=0;i<T*M;i++)
57         if (sindrome[i]==1)
58             ps++;
59
60     if(ps==0){
61         for(i=0;i<K;i++)
62             decodificato[i] = ricevuto[i+(N-K)];
63         *coset_idx = 0;
64         peso=0;
65     }
66     else {
67         *coset_idx = -1;
68         /*per tutti i coset leader*/
69         for(cst=0; cst<1024 && *coset_idx==-1; cst++){
70             /*azzera la possibile sindrome*/
71             memset(possibile_sindrome, 0, T*M*sizeof(int));
72             /*cerca coset leader che genera la sindrome*/
73             for(i=0;i<T*M;i++)
74                 for(j=0;j<N;j++)
75                     possibile_sindrome[i] = (possibile_sindrome[i]
76                             exor (echelon[i][j]*coset_ucl[cst][j]));
77             ps=0;
78             /*confronta il coset leader trovato*/
79             for(j=0; j<T*M; j++)
80                 if(possibile_sindrome[j] == sindrome[j])
81                     ps++;
82             if(ps == T*M)
83                 *coset_idx = cst;
84         }
85         peso=0;
```

```
86              for(j=0; j<N; j++)
87                  if(coset_ucl[*coset_idx][j] == 1)
88                      peso++;
89              memcpy(decodifica, coset_ucl[*coset_idx], N*sizeof(int));
90              somma_bit_a_bit(decodifica, ricevuto, N);
91              for(i=0; i<K; i++)
92                  decodificato[i] = decodifica[i+(N-K)];
93          }
94
95          return peso;
96      }
```

Riferimenti bibliografici

[1] Michele Elia: *Teoria dell'Informazione e Codici*, dispense per il corso di Teoria dell'Informazione e Codici tenuto presso il Politecnico di Torino, a.a. 2003-2004

[2] F.J. MacWilliams, N.J.A Sloane: *The Theory of Error-Correcting Code*, North-Holland editore

[3] Sandro Bellini: *Teoria dell'Informazione e Codici*, dispense per il corso di Teoria dell'Informazione e Codici tenuto presso il Politcnico di Milano, reperibile all'indirizzo:
`http://www.elet.polimi.it/upload/bellini/tinfcodNO/tinfcod2.pdf`

[4] Articolo della Wikipedia, l'enciclopedia free disponibile on line, reperibile all'indirizzo:
`http://en.wikipedia.org/wiki/Extended_Euclidean_algorithm`