Teoria dell'Informazione e Crittografia

UniShare

Davide Cozzi
@dlcgold

Indice

Capitolo 1

Introduzione

Questi appunti sono presi a lezione. Per quanto sia stata fatta una revisione è altamente probabile (praticamente certo) che possano contenere errori, sia di stampa che di vero e proprio contenuto. Per eventuali proposte di correzione effettuare una pull request. Link: `https://github.com/dlcgold/Appunti`.

Capitolo 2

Introduzione agli argomenti del corso

Si ha una sorgente che emette messaggi e li vuole mandare tramite un canale di comunicazione, che contiene del rumore che agisce sui messaggi e li rovina. Il destinatario per capire che il messaggio è rovinato ha varie tecniche. Una prima cosa che potrebbe fare è chiedere di rimandare il messaggio ma sarebbe meglio correggere *in loco* e si hanno algoritmi per farlo (tra cui lo **schema di Hamming** usando il modello del **rumore bianco**).
Un'altra tematica è la codifica stessa del sorgente, comprimendo flussi di dati, senza perdere informazioni.
Si parlerà anche dei canali di comunicazione, delle capacità e dei **teoremi di Shannon**.
Si vedranno poi le basi della crittografia, i crittosistemi storici, vari standard etc...

Capitolo 3

Teoria dell'informazione

Si hanno due sottoparti principali:

- **teoria dei codici**, per individuare e correggere gli errori. Si studia il canale di trasmissione cercando di contrastare il rumore che c'è nel canale
- **teoria dell'informazione**, in cui il focus è la sorgente delle informazioni

Vediamo il classico schemino della teoria dell'informazione:

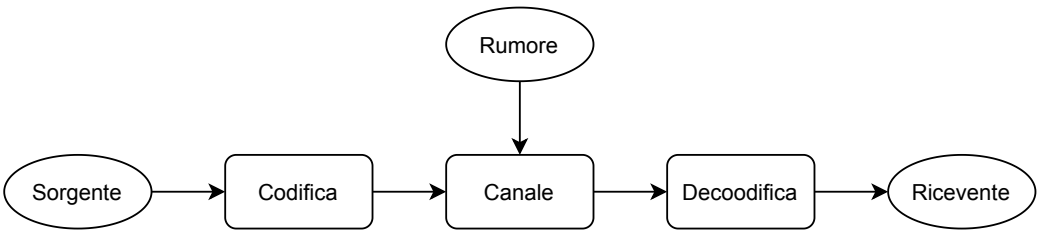


Figura 3.1: Schema di un sistema generale di comunicazione tipico della teoria dell'informazione

Avendo:

- la **sorgente** che produce segnali, dei simboli, che potrebbero essere continui (come la corrente), anche se noi li assumeremo come simboli di un alfabeto finito, avendo quindi una **sorgente discreta**
- i simboli, per poter essere spediti all'interno di un canale, vanno codificati, avendo una parte di **codifica**

- una volta codificati i simboli vanno nel **canale di trasmissione**, dove si ha del **rumore**. Tale *rumore* prende un simbolo di quelli inseriti e lo cambia
- dal canale esce o il simbolo che è entrato o il simbolo modificato dal rumore e, tipicamente, non è immediatamente utilizzabile ma deve passare per una fase di **decodifica**
- il simbolo decodificato arriva al **destinatario**

Si hanno alcune assunzioni sulla sorgente:

- è **discreta**, i simboli emessi appartengono ad un alfabeto finito. Normalmente tali simboli sono $S = \{s_1, s_2, \ldots, s_q\}$
- i simboli vengono emessi uno alla volta ad ogni **colpo di clock**. Non si ha mai che in un colpo di clock non escano simboli o che ne escano più di uno solo
- la sorgente è **senza memoria (*memoryless*)**, avendo che i simboli già usciti non influenzano per nulla il simbolo che sta per uscire. Ogni simbolo che esce non tiene conto del passato, *è come se fosse il primo*
- è **probabilistica e randomizzata**. Si ha quindi che i simboli $S = \{s_1, s_2, \ldots, s_q\}$ escono con le probabilità $(p_1, p_2, \ldots, p_q)$. Deve valere che, ovviamente, che $p_i \in [0, 1], \forall i = 1, \ldots, q$. Si ha inoltre che le varie probabilità, nel loro insieme, devono formare una distribuzione di probabilità, avendo che:

  $$\sum_{i=1}^{q} p_i = 1$$

  Potrei avere simboli con probabilità nulla di comparire ma nella pratica non è qualcosa di sensato. La sorgente la costruisco o da zero (e a quel punto un simbolo con probabilità nulla non lo metterei) o ho una sorgente che devo studiare (e qui potrei avere simboli con probabilità bassissime se non nulle, in tal caso bisognerebbe rivalutare l'assunzione dei simboli di quella sorgente). Possiamo quindi meglio dire che $p_i \in (0, 1], \forall i = 1, \ldots, q$.
  D'altro canto vedo se posso avere probabilità pari a 1 per un simbolo ma in tal caso avrei solo quello e non sarebbe interessante. Si ha quindi che:

  $$p_i \in (0, 1), \forall i = 1, \ldots, q$$

$$\sum_{i=1}^{q} p_i = 1$$

Le sorgenti che emettono i simboli secondo uno schema prefissato, deterministico, sono poco interessanti, avendo un comportamento banale

Il concetto di *spedire in un canale* può anche essere generalizzato in altre "idee", come il disco su cui salvo dei dati e il tempo per cui li salvo.
La parte di *codifica e decodifica* può essere approfondita. Nello schema in figura 3.1 ci si è infatti concentrati sul canale, avendo che la codifica serve a fare in modo che il simbolo trasmesso vada bene per essere trasmesso nel canale. La **codifica** è a sua volta suddivisa in due parti:

1. **codifica di sorgente**, che ha come obiettivo rappresentare nel modo più efficiente e compatto i simboli emessi dalla *sorgente*. Si vuole quindi comprimere la sequenza di simboli (messaggi) emessi dalla sorgente, per impegnare meno banda possibile quando andremo a spedire. Si deve considerare che ogni bit in un file compresso è essenziale per permettere di poter recuperare il contenuto compresso

2. **codifica di canale**, che ha quasi uno scopo opposto rispetto alla *codifica di sorgente*, infatti ha come obiettivo quello di contrastare il rumore e per farlo aggiunge ridondanza al messaggio (da qui il discorso sull'obiettivo opposto)

Si cerca quindi di comprimere il più possibile nella prima fase, quella di *codifica di sorgente* e di ridondare il meno possibile nella seconda, quella di *codifica di segnale*.
Per capire meglio quanto detto diamo alcune formalità.

Definizione 1. *Una **codifica** è una funzione cod che prende i simboli della sorgente $S = \{s_1, s_2, \ldots, s_q\}$ e ad ogni simbolo s_i gli assegna una stringa formata coi caratteri di un certo alfabeto Γ, l'**alfabeto della codifica**. Le stringhe di Γ^* sono tutte quelle costruite sull'alfabeto Γ di lunghezza arbitraria e finita, compresa la stringa vuota ε, che posso quindi formare coi simboli di Γ. Quindi ad ogni $s_i \in S$ assegno un $\gamma_i \in Gamma^*$, avendo che:*

$$cod : S \to \Gamma^*$$

generalmente si ha che:

$$|\Gamma| < |\Sigma|$$

e quindi i simboli di Σ *sono mappati da cod in sequenze di simboli di* Γ*, a meno che non si ritenga accettabile il fatto che due o più simboli di* Σ *vengano mappati nellostesso simbolo di* Γ*.*
Più avanti nel corso vedremo casi in cui $|\Gamma| > |\Sigma|$

Si hanno quindi i simboli $S = \{s_1, s_2, \ldots, s_q\}$ che escono con probabilità $(p_1, p_2, \ldots, p_q)$ e che vengono codificati con le stringhe $\gamma_1, \gamma_2, \ldots, \gamma_q$. Le varie $gamma_i$ sono dette **codeword**. Chiamando $l_i = |\gamma_i|$ la lunghezza di tali stringhe si ha che tali stringhe hanno associati i vari $l_1, l_2, \ldots, l_q$.
L'obiettivo quindi della *codifica di sorgente* è quello di minimizzare la lunghezza media L delle stringhe, avendo quindi una media pesata (pesata sulle probabilità):

$$L = \sum_{i=1}^{q} p_i \cdot l_i$$

Tenendo conto delle probabilità, per minimizzare L, si deve, avendo a che fare con termini che sono tutti > 0 (avendo supposto che non si hanno probabilità nulla e avendo che una codeword pari alla stringa vuota ha poco senso), fare in modo che i termini siano tutti il più piccolo possibile. Dato che le probabilità sono date mentre la codifica la sto costruendo, calcolando le codeword e di conseguenza le loro lunghezze, devo fare in modo che se la probabilità è grande la lunghezza deve essere piccola. Se invece la probabilità è piccola posso permettermi una lunghezza più grande. Parto quindi dai simboli con probabilità più grande e inizio a usare codeword più piccole possibili, usando via via quelle più lunghe.
Un'idea simile è usata nel *codice Morse* dove le lettere meno comuni hanno le sequenze più lunghe di punti, linee e spazi (avendo una codifica ternaria). La lettera più comune, la "e", ha infatti solo con un punto, la codifica più breve mentre le meno comuni hanno sequenze multiple di punti, linee e spazi che le separano (e gli spazi contano nella lunghezza di queste codeword).
Noi non sappiamo in anticipo che messaggi verranno prodotti dalla sorgente e quindi le codeword vanno studiate passo a passo, valutando i simboli più probabili per associare le codeword più brevi e i meno probabili per le codeword più lunghe.
Analizziamo meglio i codici, le codeword. Possono essere:

1. *a lunghezza fissa*, ovvero si ha che $l_1 = l_2 = \cdots = L_q$
2. *a lunghezza variabile*, avendo che ogni codeword può avere lunghezza diversa

Ne segue quindi che il discorso di minimizzare L ha senso solo in presenza di *codeword a lunghezza variabile* (potendo decidere per ogni simbolo che

codeword associare), avendo la **codifica a lunghezza variabile**.
Con *codeword a lunghezza fissa* avrei tutte le l_i uguali e quindi avrei, avendo $l_i = l, \forall i$:

$$L = \sum_{i=1}^{q} p_i \cdot l_i \sum_{i=1}^{q} p_i \cdot l = l \cdot \sum_{i=1}^{q} p_i = l \cdot 1 = l$$

avendo, come facilmente intuibile, che la lunghezza media è la lunghezza fissa stessa. Si hanno codifiche a lunghezza fissa, come banalmente numeri a 64bit etc... in tal caso si parla di **codici a blocchi**.
Usando codifiche a lunghezza fissa si hanno anche esempi interessanti come quello del *codice pesato*, detto **codice pesato 01247**. Il nome deriva dal fatto che si possono codificare le cifre da 0 a 9 (da 1 a 9 con poi lo 0 dopo il 9) sotto forma di stringhe di 5 bit usando i pesi 0,1,2,4,7 associati a ciascun bit. Vediamo la tabella con la codifica di questo codice:

	0	1	2	4	7
1	1	1	0	0	0
2	1	0	1	0	0
3	0	1	1	0	0
4	1	0	0	1	0
5	0	1	0	1	0
6	0	0	1	1	0
7	1	0	0	0	1
8	0	1	0	0	1
9	0	0	1	0	1
0	0	0	0	1	1

Si nota che ogni codeword ha sempre 3 bit pari a 0 e due bit pari a 1, avendo un **codice 2-su-5**. Banalmente i pesi si associano ai numeri 0,1,2,4,7 in modo tale che essi, sommati, formino il numero voluto (ad esempio per 1 avrò i pesi su 0 e 1, per 9 su 2 e 7 etc...). L'unico caso è il caso dello 0, che non può essere ottenuto come somma di due pesi (spesso si hanno nei codici casi speciali da gestire a parte). Per lo 0 viene quindi presa una codeword non usata per altri numeri e quindi l'unica scelta possibile è avere i pesi su 4 e 7 (visto che farebbe 11).
Su un totale ci 5 bit, avendo due bit a 1 e tre bit a 0, posso avere un numero di codeword pari a:

$$n = \binom{5}{2} = 10$$

avendo che i 5 bit sono associati ai 5 elementi dove 1 segnala che "sto usando quel peso", prendendo quindi i sottoinsiemi di due elementi a partire da un

insieme di cinque elementi, ovvero "in quanti modi posso formare sottoinsiemi che contengono due elementi a partire da un insieme di cinque elementi" o detto altrimenti "quanti sono i modi in cui posso disporre due uni all'interno di una stringa di cardinalità cinque".
In un linguaggio di programmazione privo di una struttura dati dedicata posso simulare un insieme di questo tipo tramite un vettore di bit (con 1 se l'elemento associato all'indice c'è).
Il codice a barre è detto ***codice 39*** *ed è un* ***codice 3-su-9***.
In merito alla **decodifica** si ha che anch'essa sarà di due tipi:

1. **decodifica di canale**, vedendo e c'è stato un errore di trasmissione ed eventualmente correggendolo in automatico se il codice mi consente di farlo

2. **ulteriore decodifica** che non è esattamente una *codifica di sorgente* ma quanto una *trasformazione*, dove le *codeword* vengono trasformate nel formato leggibile dal **ricevente**

3.1 Codici per individuare errori

Ci concentriamo ora sulla *codifica di canale* ignorando per ora la *codifica di sorgente*, avendo come obiettivo l'aggiunta di ridondanza a simboli, che si suppongono già codificati con codeword, in modo tale che in queste codeword, spedite nel canale dove eventualmente si possono avere modifiche causate dal rumore, vengano eventualmente riconosciuti (ed eventualmente corretti) errori in fase di *decodifica*.
Parlando di codici per individuare errori solitamente, nei disorsi, si ha che $|\Gamma| < |\Sigma|$ Qualora il ricevente con la sua decodifica si accorga che è successo qualcosa ma non si è in grado di correggere quel qualcosa si hanno i cosiddetti **codici per individuare gli errori (*error detection codes*)**. Nel caso in cui il ricevente con la sua decodifica si accorga dell'errore ci si chiede anche se può correggerlo autonomamente senza chiedere che la sorgente spedisca nuovamente il messaggio. Non sempre questa cosa si può fare ma quanto accade si parla di **codici a correzione d'errore (*error correction codes*)**.

3.1.1 Controllo di parità semplice

Vediamo come capire se un messaggio ricevuto è valido.
Si supponga di spedire un pacchetto di n bit (ma potrebbe essere qualsiasi altra cosa ma per praticità prendiamo un bit) nel canale e che da esso esca un certo pacchetto sempre di n bit (per il rumore potrebbe non essere lo stesso).

Definizione 2. *Definiamo il* ***controllo di parità****.*
Avendo una sequenza di n *bits in cui si ha* $n-1$ *bits, dette* ***cifre di messaggio*** *di* ***messaggio vero****, che chiameremo* ***msg*** *e un bit che è la* ***cifra di controllo****, che chiameremo* ***check****. Le cifre di messaggio si indicano con* $\circ$ *mentre la cifra di controllo con* $\times$ *e quindi il messaggio è del tipo:*

$\circ$	$\circ$	$\circ$	$\circ$	$\cdots$	$\circ$	$\times$

avendo $n-1$ $\circ$ *e un solo* $\times$ *(che potrebbe anche non essere in fondo, basta avere coscienza della posizione nel pacchetto, concordando la cosa tra mittente e ricevente).*
Si ha che:

- *chi spedisce ha le cifre di messaggio e deve calcolare la cifra di controllo*
- *chi riceve controlla che la cifra di controllo sia coerente con le cifre di messaggio*

*Nell'****error detection code*** *il ricevente è solo in grado di capire che la sequenza non è valida ma per farlo bisogna assumere di avere limitazioni nella sequenza di* n *bit che è entrata nel canale. Questa limitazione è che gli* n *bit entranti nel canale siano una* ***codeword valida****, avendo che, preso un sottoinsieme* M *di tutto l'insieme di* n *bit, ovvero* $M \subseteq \{0,1\}^n$*,* M *è un insieme di codeword valide. Quindi solo un messaggio appartenente a* M *può entrare nel canale. Fatta questa premessa, quando esce un messaggio, ho che, a causa del rumore, questo messaggio viene rovinato, non avendo più un messaggio valido (cosa che viene capita dal ricevente). Purtroppo può succedere che il rumore trasformi un messaggio valido in un altro messaggio valido ma non considereremo questa opzione per ora.*
Nel ***controllo di parità semplice*** *il pacchetto di* n *bit, come visto è formato da* $n-1$ *bit di messaggio e un bit di controllo, la* ***check digit****. Si procede quindi, ricordando che siamo in un caso binario, a contare il numero di* 1 *nei primi* $n-1$ *bit e se questo è dispari setto il bit di check a* 1 *e si nota che così il numero di* 1 *nel pacchetto intero di* n *bit diventa pari, avendo la cosiddetta* ***parità pari*** *(ovvero ogni sequenza valida ha un numero pari di* 1*). Controllando il numero di* 1 *il ricevente capisce se il rumore ha modificato il messaggio anche se non può capire cosa è successo (avendo quindi che il* controllo di parità semplice *è solo un* error detecting code *e non un* error correcting code*). Non potendo fare nulla, in caso di errore identificato, il ricevente può solo chiedere al mittente di inviare nuovamente il messaggio. Qualora il rumore modificasse il messaggio in modo tale che si abbia comunque*

un numero apri di 1 *si rientrerebbe nella casistica sopra descritta in cui il rumore forma ancora un messaggio valido. Questa cosa può succedere se, nel caso binario, il rumore modifica un numero pari di cifre e quindi il* controllo di parità semplice *funziona solo se viene modificato un* numero dispari di cifre.

Vediamo quindi meglio come calcolare la **check digit**.
Si rinominiamo gli n bit come:

$$x_1 x_2 \cdots x_{n-1}\, y$$

quindi con y *check digit*.
Si ha che, con $\oplus$ *xor*:

$$y = x_1 \oplus x_2 \oplus \cdots \oplus x_{n-1}$$

Ricordando che:

a	b	$a \wedge b$	$a \oplus b$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

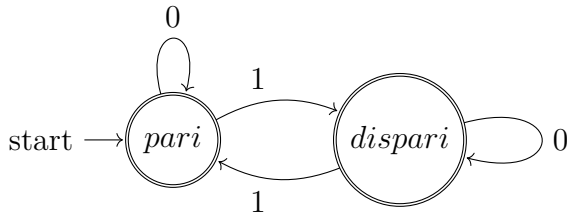
quindi vale 1 sse i due bit in input sono diversi ma questo non ci aiuta su $n-1$ input. Altrimenti si ha che vale 1 sse il numero di 1 in input è dispari e questo ci aiuta su $n-1$ input infatti la generalizzazione dello *xor* a più di due input è detta **funzione di parità**. Un altro punto di vista per considerare lo *xor* è quello della **somma a modulo 2** usando la notazione:

$$y = \bigoplus_{i=1}^{n-1} x_i = \sum_{i=1}^{n-1} x_i \bmod 2$$

avendo che faccio prima la somma e poi il modulo 2 mi dice 0 se è pari e 1 se è dispari.
Si ha inoltre una relazione interessante tra le formule scritte usando solo $\oplus$ e $\wedge$ nella cosiddetta **forma algebrica normale (*ANF*)**. Queste formule booleane possono essere trasformate in formule aritmetiche con *modulo 2*, quindi in $\mathbb{Z}_2$ dicendo che lo *xor* equivale alla *formula modulo due* e l'*and* al *prodotto modulo due* (la cosa vale in entrambi i versi).
La funzione di parità è così usata che in tutti i microprocessori, fin dagli anni settanta, si ha un *flag di parità* tra i flag della CPU, che viene settato come

appena visto a seconda dei bit caricati su un registro particolare (a volte detto *accumulatore*). Tale calcolo è facilmente mappabile in un circuito, avendo che lo *xor* gode della proprietà associativa (avendo un circuito che fa un albero di porte *xor*).

Posso anche simulare lo *xor* con un **automa a stati finiti**, con due stati "pari" e "dispari", con il "pari" stato iniziale (diciamo che input vuoto è pari):



Questo metodo ha senso se il canale è pochissimo rumoroso, avendo pochissima probabilità di avere la modifica di un bit e ancora meno di due (due modifiche si ricorda che non verrebbero rilevate essendo pari), così poca da poter ipotizzare che non avvengano mai due errori (e se mai dovesse succedere bisognerà valutare l'impatto del problema e le conseguenze). Stiamo assumendo quindi che la **probabilità d'errore** può essere **trascurabile** infatti canali di buona qualità dovrebbero sbagliare non più di un bit su un milione, per canali più affidabili anche uno su un miliardo. Possiamo quindi trascurare che possano accadere due errori e dire che il **controllo di parità** va bene.
Parliamo ora meglio di **ridondanza**, definendola formalmente.

Definizione 3. *La* ***ridondanza*** *R è definita come:*

$$R = \frac{\textit{il numero totale di simboli/cifre spediti}}{\textit{numero di simboli/cifre che sono effettivamente parte del messaggio}}$$

Nel caso del controllo di parità *i simboli che vogliamo spedire sono n bit a fronte di $n-1$ bit di vero messaggio. Si ha quindi:*

$$R = \frac{n}{n-1} = \frac{(n-1)+1}{n-1} = 1 + \frac{1}{n-1}$$

Mettendo in evidenza che la ridondanza è sempre $R \geq 1$, visto che a numeratore abbiamo almeno una cifra in più (quella della check digit*) e che quindi è sicuramente maggiore del denominatore. In realtà per avere $R = 1$ dovrei avere numeratore e denominatore uguali che non ha molto senso parlando di ridondanza, quindi nei casi* interessanti *si ha che $R > 1$. Guardando la formula la cosa è confermata da $1 + \frac{1}{n-1}$ con $\frac{1}{n-1}$ che viene detto* ***eccesso di ridondanza****.*

Definizione 4. *Possiamo **generalizzare** la definizione di **ridondanza**, indicando $tot = msg + check$:*

$$R = \frac{msg + check}{msg} = \frac{tot}{msg}$$

avendo:

- *msg numero di simboli/cifre di messaggio*
- *$check$ numero di cifre di controllo*

Ma allora (avendo $check < msg$ per avere qualcosa di sensato):

$$R = \frac{msg + check}{msg} = 1 + \frac{check}{msg}$$

*che è la forma "generale" della ridondanza. Si ha che $\frac{check}{msg}$ è **eccesso di ridondanza**.*

Si può dire di non avere necessità di "proteggere" di più il bit di parità in quanto, per la macchina, conta come tutti gli altri. Tutti vanno "protetti" nello stesso modo.
*Come ho la **parità pari** potrei avere la **parità dispari**, dove i messaggi validi hanno un numero dispari di 1. I vari ragionamenti sono analoghi, essendo tutto uguale dal punto di vista matematico, avendo un isomorfismo tra le due tecniche. La scelta tra i due dipende dai casi è dalla celta di cosa rappresentiamo con 0 e 1 (pensiamo con 0 che rappresenta assenza di segnale, in questo caso meglio usare la parità dispari, mentre se 0 e 1 rappresentassero diverse quantità di Volt andrebbe bene la parità pari).*
Nel corso si userà comunque solo la *parità pari*.

3.1.2 Il rumore bianco

Introduciamo ora un primo *modello di rumore*, il **modello del rumore bianco**.

Definizione 5. *Un **modello di rumore** è un modello matematica che descrive cosa succede nel canale quando il rumore rovina i bit.*

Definizione 6. *Il **modello del rumore bianco** consiste nell'avere il messaggio con i bit $x_1x_2 \cdots x_n$ (con magari x_n come controllo di parità ma dato che "i bit non sono colorati" la cosa non ci interessa davvero) e avere una certa probabilità p. Si hanno due condizioni:*

1. *si ha che* $p \in (0,1)$ *che è la probabilità che avvenga un errore in ogni posizione* $i \in [1,n]$ *del messaggio. Si ha quindi che la probabilità* p *è uguale in tutte le posizioni*

2. *le posizioni sono tutte indipendenti, ovvero il fatto che magari si ha un errore nella posizione* i *non influisce sulle altre. Avendo quindi l'eventi casuale* E_i *con:*

   $$E_i = \text{è avvenuto un errore in } i$$

   allora:

   $$E_i \text{ ed } E_j \text{ sono indipendenti, } \forall i \neq j$$

Le due proprietà sopra elencate rendono molto semplice il modello.
Questo però non è molto realistico, basti pensare al rumore dovuto ad uno sbalzo di corrente, dove da un bit in poi e per diversi bit si avranno alte probabilità d'errore. Quando l'errore influisce su una certa porzione di bit si dice che si ha un ***burst di errori*** *(che non può essere gestito con le tecniche per il rumore bianco, anche se si riesce con qualche workaround).*
Si è visto che $p \in (0,1)$ *infatti:*

- *se si avesse* $p = 0$ *si avrebbe che ogni bit arriverebbe sempre corretto, ma questo può avvenire solo in un mondo utopico e non in quello reale/fisico. Non esiste un canale reale non affetto da errori, quindi si ha* $p \neq 0$

- *se si avesse* $p = 1$ *si avrebbe che ogni bit del messaggio arriverebbe errato ma questa non sarebbe una brutta situazione, anzi sarebbe ottima infatti mi basterebbe avere una porta logica not della linea di trasmissione per riottenere il messaggio corretto, ottenendo un canale* $p = 0$ *d'errore. Anche questo però è irrealistico quindi* $p \neq 1$

Supponiamo ora che $p > \frac{1}{2}$ *quindi ho più probabilità che un bit arrivi sbagliato che giusto. Anche in questo caso una porta logica not alla fine della linea di trasmissione per ottenere un canale con* $1 - p$ *come probabilità d'errore. Quindi anche questo non ha molto senso quindi si considera che:*

$$p \in (0, \frac{1}{2})$$

Manca solo da valutare $p = \frac{1}{2}$.
Con $p = \frac{1}{2}$ *si ha che il bit di output è completamente causale e indipendente*

da cosa sia stato spedito. È come se il canale generasse n bit casuali con probabilità uniforme ($\frac{1}{2}$), avendo un cosiddetto ***canale completamente rumoroso****. Dal punto di vista pratico sarebbe interessante un tale canale, per altri punti di vista (come quello della crittografia), avendo infatti un* ***generatore di bit completamente casuali****. Purtroppo questo non si può fare quindi si assume $p \neq \frac{1}{2}$.*

Cerchiano di capire quale sia la probabilità che avvengano k errori con $0 \leq k \leq n$ (quindi da nessun errore a tutti gli n bit errati), che indichiamo con:

$$p[\text{ k errori }]$$

Valutiamo i vari casi:

- partiamo con 1 errore, quindi $k = 1$, avendo $p[\text{ 1 errori }]$.
  Questo significa che per il messaggio di n bit si immagina un vettore di bit associato con 0 e 1 come "bandierine" che indicano se è avvenuto un errore o no in una certa posizione. Quindi se in un certa posizione ho 0 diciamo che significa che non ho un errore di trasmissione mentre se ho 1 ho un errore. Il messaggio di n bit diventa quindi una sorta di *maschera* che con gli 1 mi dice dove è avvenuto l'errore. Se suppongo che ne è avvenuto uno solo avrò un solo 1 e bisogna calcolare la probabilità che questo avvenga.
  Supponga che l'errore sia al primissimo bit, quindi in posizione $i = 1$, avendo quindi, per il discorso delle "bandierine" che $msg = 10000\ldots0$ e quindi si ha, avendo che la probabilità che avvenga la trasmissione avvenga correttamente è $1 - p$ (cosa che avviene $n - 1$ volte), mentre p che avvenga sbagliata (cosa che avviene una sola volta):

  $$p[\text{ 1 errori }] = p^1 \cdot (1-p)^{n-1} = p \cdot (1-p)^{n-1}$$

  Posso fare $\cdot$ in quanto si è supposta l'*indipendenza* (non avendo intersezioni tra gli eventi).
  Ma questo non sta considerando tutto ma solo la prima posizione. Completando il calcolo avendo di volta in volta in somma la probabilità di un errore nella posizione i ho che:

  $$p[\text{ 1 errori }] = p^1 \cdot (1-p)^{n-1} + (1-p)^1 \cdot p^1 \cdot (1-p)^{n-2} + \cdots$$

  ma questo conto si può semplificare, avendo sempre gli stessi termini che si ripetono:

  $$p[\text{ 1 errori }] = n \cdot p^1 \cdot (1-p)^{n-1} = n \cdot p \cdot (1-p)^{n-1}$$

Infatti so che $p \cdot (1-p)^{n-1}$ è la probabilità di avere un errore in una certa posizione fissata. Mi chiedo dove posso mettere questa posizione in tutti i modi possibili nel pacchetto di n bit e ho che, avendo un solo errore, ho n modi per posizionarlo, ciascuno con probabilità $p \cdot (1-p)^{n-1}$

- passiamo a due errori, avendo $p[$ 2 errori $]$.
  Ho un ragionamento analogo. Parto supponendo di avere i due errori nelle prime due posizioni del messaggio/pacchetto, avendo quindi 1 nelle prime due posizioni della maschera. Abbiamo comunque già visto che poi il ragionamento si generalizza per qualsiasi posizione, in questo caso coppie (anche non consecutive) di posizioni. Si ha che, ipotizzando che le prime due siano errate:

  $$p[\text{ 2 errori }] = p^1 \cdot p^1 \cdot (1-p)^{n-2} = p^2 \cdot (1-p)^{n-2}$$

  Ma anche qui dobbiamo vedere la probabilità per qualsiasi coppia, facendo variare le due posizioni d'errore in tutti i modi possibili ma questo è come prendere un qualsiasi sottoinsieme di due elementi a partire da un insieme di n elementi ma questo altro non è che il calcolo che si fa tramite il coefficiente binomiale, avendo quindi:

  $$p[\text{ 2 errori }] = \binom{n}{2} \cdot p^2 \cdot (1-p)^{n-2}$$

- analogamente a quanto fatto per due errori potrei fare con tre, quattro, etc...

- possiamo generalizzare con k errori, avendo $p[$ k errori $]$.
  Si hanno quindi k uni da disporre in tutti i modi possibili nel vettore di n bit. Si ha quindi:

  $$p[\text{ k errori }] = \binom{n}{k} \cdot p^k \cdot (1-p)^{n-k}$$

  E quindi posso valutare la cosa nei due casi estremi:

  - $k = 0$, avendo 0 errori. Ho un solo modo per mettere zero 1 nella maschera di bit (da nessuna parte) e infatti (avendo poi tutti gli n bit la stessa probabilità di uscire corretti):

    $$p[\text{ 0 errori }] = \binom{n}{0} \cdot p^0 \cdot (1-p)^{n-0} = 1 \cdot 1 \cdot (1-p)^n = (1-p)^n$$

– $k = n$, avendo n errori[1]. Ho un solo modo per mettere tutti 1 nella maschera di bit (ovunque) e infatti (avendo poi tutti gli n bit la stessa probabilità di uscire errati):

$$p[\text{ n errori }] = \binom{n}{n} \cdot p^n \cdot (1-p)^{n-n} = 1 \cdot p^n \cdot 1 = p^n$$

Si nota che i due casi estremi sono "speculari".

In questo elenco puntato si è quindi ragionato sulle celle della maschera di bit associata al pacchetto e non del pacchetto in se, anche se spesso risulti ambiguo.
Consideriamo ora nuovamente $p[\text{ 1 errore }]$, si ha che, dalla generalizzazione è:

$$p[\text{ 1 errore }] = \binom{n}{1} \cdot p^1 \cdot (1-p)^{n-1} = n \cdot p \cdot (1-p)^{n-1}$$

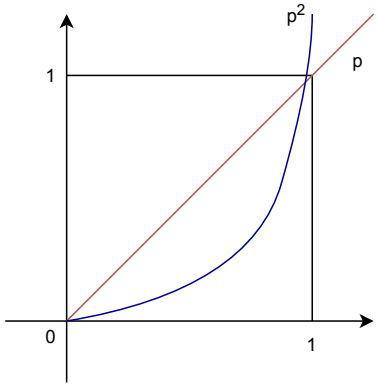
Introduciamo un'approssimazione interessante dell'analisi matematica che vale per $\alpha \in \mathbb{R}$ e $|X| < 1$, ovvero $-1 < x < 1$:

$$(1+x)^\alpha \simeq 1 + \alpha \cdot x$$

Ovvero $1 + \alpha \cdot x$ sono i primi due termini dello sviluppo in serie di $(1+x)^\alpha$.
Tratto quindi la formula per un errore in base a questa approssimazione:

$$p[\text{ 1 errore } = n \cdot p \cdot (1-p)^{n-1} \simeq n \cdot p \cdot [1 - p \cdot (n-1)] = n \cdot p - n^2 \cdot p^2 + n \cdot p^2$$

Ma so che $p \in (0,1)$ e quindi $p^2 < p$, infatti (*grafico approssimativo*):



[1]Su dispense del prof grafico con $n = 8$, $p = 0.1$ e k che varia tra 0 e 8

ma quindi, sempre approssimando (avendo quindi già due approssimazioni):

$$p[\text{ 1 errore } = n \cdot p \cdot (1-p)^{n-1} \simeq n \cdot p - n^2 \cdot p^2 + n \cdot p^2 \simeq n \cdot p$$

Quindi:

$$p[\text{ 1 errore } \simeq n \cdot p$$

Analogamente ragiono per due errori:

$$p[\text{ 2 errori }] = \binom{n}{2} \cdot p^2 \cdot (1-p)^{n-2} \simeq \binom{n}{2} \cdot p^2 \cdot [1 - p \cdot (n-2)] = \binom{n}{2} \cdot (p^2 - n \cdot p^3 + 2p^3)$$

Ma anche qui si ha che $p \in (0,1)$ e quindi $p^3 < p$, e quindi si ha:

$$p[\text{ 2 errori }] \simeq \binom{n}{2} \cdot p^2 = \frac{n \cdot (n-1)}{2} \cdot p^2$$

In generale, per k errori, con gli stessi passaggi:

$$p[\text{ k errori }] \simeq \binom{n}{k} \cdot p^k$$

I conti diventano molto più semplici.
Si è detto che se la probabilità di due errori è piccoli si può decidere di trascurarla (usando poi solo il *controllo di parità semplice*). Da queste approssimazioni vediamo che la probabilità di un errori è $\simeq n \cdot p$ mentre per due $\simeq \frac{n \cdot (n-1)}{2} \cdot p^2$. Se ipotizziamo $p \sim 10^{-6}$, quindi uno su un milione, si ha che $p^2 = 10^{-12}$ quindi assolutamente trascurabile. Si segnala comunque che questa non è una pratica standard. Normalmente si hanno, ad esempio, **low density parity codes (LDPC)** dove si sparano a caso vari controlli di qualità nell'ottica di mantenere le proprietà di correzione degli errori usando meno controlli possibile, avendo, tornando alla ridondanza $R = 1 + \frac{check}{msg}$, che si vuole usare il minor numero di cifre di controllo per abbassare l'*eccesso di ridondanza*, abbassando la ridondanza stessa, avvicinandosi quindi a 1^+ (ci si avvicina da destra ovviamente). Riducendo l'*eccesso di ridondanza* si ha che ogni cifra di controllo *copre/protegge* il maggior numero ci simboli/cifre del messaggio. *A parità di simboli inviati si vuole quindi ridurre il numero di cifre di controllo.*
Approfondiamo e usiamo quindi il *modello del rumore bianco* per vedere qual è a probabilità che il ricevente non riesca a capire che c'è stato un errore utilizzando il *controllo di parità semplice.*
Ricordiamo che il messaggio è della forma:

$$x_1 x_2 \cdots x_{n-1} y$$

con *y controllo di parità semplice* calcolato come:

$$y = \bigoplus_{i=1}^{n-1} x_i$$

Un numero dispari di errore mi segnala che ci sono stati problemi, usando la *parità pari* (i pacchetti inseriti nel canale hanno un numero apri di 1).
Vogliamo quindi la probabilità che il controllo di parità fallisca, ovvero:

$$p[\text{ controllo } \bigoplus \text{ fallisce }]$$

ma questo è uguale alla probabilità che avvenga un numero pari di errori (**zero escluso**, ovviamente):

$$p[\text{ controllo } \bigoplus \text{ fallisce }] = p[\text{ numero pari di errori }] = p[\text{ 2 errori }] + p[\text{ 4 errori }] + \cdots$$

Diciamo che, per comodità:

$$1 = (1-p) + p = [(1-p) + p]^n$$

Ma so che $(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^{n-k} b^k$, quindi:

$$1 = [(1-p) + p]^n = \sum_{k=0}^{n} \binom{n}{k} (1-p)^{n-k} p^k$$

ma questa è $\binom{n}{k}(1-p)^{n-k}p^k = p[\text{ k errori }]$ (infatti la somma di tutte le probabilità è appunto 1), quindi:

$$1 = [(1-p) + p]^n = \sum_{k=0}^{n} \binom{n}{k} (1-p)^{n-k} p^k = \sum_{k=0}^{n} p[\text{ k errori }]$$

D'altro canto posso anche dire che, sempre applicando l'espansione di $(a+b)^n$, scomponendo però $(-p)^k$ in $(-1)^k \cdot p^k$ (dove $(-1)^k$ vale 1 per k pari e -1 per k dispari). Si ha quindi:

$$(1 - 2 \cdot p)^n = [(1-p) - p]^n = \sum_{k=0}^{n} (-1)^k \cdot \binom{n}{k} \cdot p^k \cdot (1-p)^{n-k}$$

Ma quindi ho:

$$(1 - 2 \cdot p)^n = \begin{cases} \binom{n}{k} \cdot p^k \cdot (1-p)^{n-k} & \text{sse } k \text{ è pari} \\ -\binom{n}{k} \cdot p^k \cdot (1-p)^{n-k} & \text{sse } k \text{ è dispari} \end{cases}$$

Quindi se k è dispari le espansioni di 1 e $(1-2\cdot p)^n$ sono uguali ma di segno opposto mentre se k è pari sono uguali con lo stesso segno. Ma quindi questa somma delle due espansioni mi lascia col doppio dei soli termini con k pari che ci aiuta volendo calcolare proprio le probabilità con un numero di errori pari. Si ha quindi, dividendo già per due avendo il discorso del doppio:

$$\frac{1+(1-2\cdot p)^n}{2} = \sum_{t=0}^{\lfloor \frac{n}{2} \rfloor} \binom{n}{2\cdot t} \cdot p^{2t} \cdot (1-p)^{n-2t}$$

La somma va quindi da 0 alla parte intera di $\frac{n}{2}$. Nel coefficiente binomiale ho $2\cdot r$ che è una quantità sicuramente pari. In generale è come se avessi $k = 2\cdot t$ ciclando solo sui k pari. Ho quindi ottenuto:

$$p[\text{ 0 errori }] + p[\text{ 2 errori }] + p[\text{ 4 errori }] + \cdots$$

Non vogliamo però $t = 0$ quindi:

$$p[\text{ controllo } \bigoplus \text{ fallisce }] = p[\text{ numero pari di errori }] = \frac{1+(1-2\cdot p)^n}{2} - p[\text{ 0 errori }]$$

$$= \frac{1+(1-2\cdot p)^n}{2} - \binom{n}{0} \cdot p^0 \cdot (1-p)^{n-0} = \frac{1+(1-2\cdot p)^n}{2} - (1-p)^n$$

E quindi:

$$p[\text{ controllo } \bigoplus \text{ fallisce }] = p[\text{ numero pari di errori }] = \frac{1+(1-2\cdot p)^n}{2} - (1-p)^n$$

D'altro canto potrei anche calcolare $p[$ numero dispari di errori $]$:

$$p[\text{ numero dispari di errori }] = 1 - p[\text{ numero pari di errori }]$$

(*o anche modificando la sommatoria per ciclare sui k dispari*).
Facendo qualche conto[2] si ottiene che:

$$p[\text{ numero dispari di errori }] = 1 - (p[\text{ numero pari di errori }] + p[\text{ 0 errori }])$$

ovvero:

$$p[\text{ numero dispari di errori }] = \frac{1-(1-2\cdot p)^n}{2}$$

In generale il numero dispari di errori è meno interessante.

[2] i calcoli per il numero dispari di errore sono materiale extra sulle dispense del docente

3.1.3 Gestione dei burst

Come abbiamo introdotto con il solo *controllo di parità* e con il *rumore bianco* non si possono gestire i **burst di errori**. Vediamo quindi un modo semplice per gestirli.
Si supponga di voler spedire dei messaggi formati da lettere, ad esempio:

c	i	a	o

Posiamo di rappresentare ogni lettera tramite l'**ASCII standard** a 7 bit:

char	bit
c	1000011
i	1001001
a	1000001
o	1001111

Si supponga di avere dei burst di errori di lunghezza L e per semplicità assumo L di lunghezza pari alle singole word, quindi $L = 7$. Per gestire il burst spedisco prima i 7 bit della prima lettera poi quelli della seconda etc... Infine spedisco un intero pacchetto di bit di controllo di 7 bit dove ogni bit viene calcolato controllando quella posizione di bit in tutti i pacchetti precedenti, sempre tramite lo *xor*. Nel caso d'esempio si ha quindi, con x per indicare il **check** (se nella colonna sopra ho un numero apri di 1 metto 0 altrimenti 1):

c	1000011
i	1001001
a	1000001
o	1001111
x	0000100

Suppongo un burst che rovini dalla posizione 2 alla 4 incluse (avendo che quindi molto probabilmente non ci torneranno i conti facendo il check su $x[2, 4] = 000$).
Ovviamente anche qui un numero pari di errori inganna il sistema avendo comunque un *controllo di parità semplice* e anche in caso d'errore il ricevente non sa comunque dove sia avvenuto e quindi fa **detection** ma non può fare **correction**.

3.1.4 Codici pesati

Abbiamo già parlato del **codice 01247** vediamo ora un codice pesato più interessante e utilizzato.

Definizione 7. *Definiamo questo* ***codice pesato*** *come un codice per cui si hanno alcune cifre di messaggio* $msg = m_1 m_2 \cdots m_n c$ *alle quali associamo dei pesi che dipendono dalla posizione in cui si trovano le varie cifre. In particolare si ha peso:*

- 1 *per la* ***check digit*** c
- 2 *per* m_n
- *si prosegue sempre aumentando di 1 per le altre cifre*
- n *per* m_2
- $n+1$ *per* m_1

Questo si fa perché la cifra di controllo è calcolata per far ottenere:

$$m_1 \cdot (n+1) + m_2 \cdot n + \cdots + m_n \cdot 1 + c \cdot 1 = 0$$

ma ovviamente questo non sembra possibile e infatti i conti sono fatti in **modulo numero primo**, avendo per esempio, se scegliamo come numero prima 37:

$$m_1 \cdot (n+1) + m_2 \cdot n + \cdots + m_n \cdot 1 + c \cdot 1 \equiv 0 \bmod 37$$

La scelta di 37 non è causale, infatti volendo:

- rappresentare le 21 lettere dell'alfabeto inglese
- rappresentare dieci cifre da 0 a 9
- un simbolo per lo spazio

e quindi siamo a 32 simboli e ci serve un numero primo ≥ 31 e quindi va bene 37.
Vogliamo un numero prima perché se vogliamo fare i conti con le congruenze è più semplice farle in *modulo numero primo*.
Lavoriamo quindi nella classe dei resti:

$$[0]_{37}, [1]_{37}, \ldots, [36]_{37}$$

e in questo modo se facciamo le varie operazioni è tutto uguale al solito fino a 36 (cosa che non succede per le classi dei resti in modulo non numero primo). La classe dei resti in modulo numero primo è un **campo** mentre se non fosse primo si avrebbe un **anello**. In un campo se $x \cdot y = 0$ o che $x = 0$ oppure $y = 0$ (cosa che non succede negli anelli). Inoltre in un campo ho che se $x \cdot y = z$ allora $x = z \cdot y^{-1}$ (in un anello non per tutti gli y esiste un y^{-1} mentre in un campo sì).
Facendo dipendere il calcolo del peso della **check digit** da tutti gli altri pesi perché, così facendo, sopratutto nelle comunicazioni di tipo **seriale** (dove si spedisce una cifra alla volta), ci si accorge subito se una cifra è andata persa oppure se si è aggiunta cifra o se due cifre si sono scambiate (cosa comunque difficile in un sistema di comunicazione elettronico ma è utile in altre situazioni, sopratutto di conti "a mano").
Si supponga di avere delle cifre b e a, la prima con peso $k+1$ e la seconda con peso k, avendo una scrittura del tipo $cifra(peso)$:

$$b(k+1) + a(k)$$

Ipotizziamo di scambiare a e b (ora a pesa $k+1$ e b pesa k), avendo:

$$a(k+1) + b(k)$$

Ma facendo la differenza si nota che non è nulla:

$$[b(k+1) + a(k)] - [a(k+1) + b(k)] \neq 0$$

infatti ho:

$$b \cdot k + b + a \cdot k - a \cdot k - a - b \cdot k = b - a$$

ma $b - a = 0$ sse $b = a$ e quindi l'unico caso in cui non ci si accorge dello scambio è avere lo scambio di due cifre uguali che non fa cambiare il risultato. Questa idea viene usata anche nei codici a barre. Vediamo quindi un algoritmo per calcolare la cifra di controllo:

Algorithm 1 Algoritmo di calcolo dei pesi per codice pesato

```
function CheckCalc
    sum ← 0
    ssum ← 0
    while not EOF do
        read sym
        sum ← sum + sym (mod 37)
        ssum ← ssum + sum (mod 37)
    temp ← ssum + sum (mod 37)
    c ← 37 − temp (mod 37)
    return c
```

Dove:

- *sum* tiene conto della somma numerica della nostro calcolo, accumulando i vari termini
- *ssum* che è una *somma delle somme* e tiene conto implicitamente dei vari persi che crescono spostandoci da destra a sinistra come visto sopra. Si accumulano i termini e i loro pesi

I mod37 *nel ciclo sono in realtà superflui ma conviene farli per non far diventare i numeri troppo grossi. Le ultime due operazioni servono a risolvere alcune problematiche che non vediamo qui.*
Vediamo una più chiara simulazione.

Esempio 1. *Avendo, simulando per un messaggio wxyzc, con c check digit:*

msg	sum	ssum
w	w	w
x	$w+x$	$2\cdot w+x$
y	$w+x+y$	$3\cdot w+2\cdot x+y$
z	$w+x+y+z$	$4\cdot w+3\cdot x+2\cdot y+z$
c	$w+x+y+z+c$	$5\cdot w+4\cdot x+3\cdot y+2\cdot z+c$

Arrivato alla fine voglio calcolare c in modo che:

$$5\cdot w+4\cdot x+3\cdot y+2\cdot z+c\equiv 0 \bmod 37$$

Il mittente ha un messaggio e ci calcola la **check digit**. Chi riceve fa lo stesso calcolo e alla fine controlla la **check digit**. Un altro modo per il ricevente è quello di fare solo l'ultimo calcolo se farli tutti step by step.
Introduciamo un particolare tipo di codice, quello **ISBN (*International Standard BookNumber*)** dei libri, che sono legati, per il codice a barre, allo standard europeo **EAN13** (negli USA si usa lo standard **UPC**). In questo codice si hanno 10 cifre (con al più il carattere X) che un identificano in modo univoco ad un libro. Nel dettaglio:

- la prima cifra rappresenta lo stato in cui è stampato il libro. Questo ha problemi non avendo solo 9 stati che producono libri
- le successive 2 cifre sono le prime due per l'editore, anche questo è un problema in quanto alcuni stati hanno più di 100 case editrice
- le successive 6 sono il numero del libro

- l'ultima cifra è il checksum, la check digit

In realtà ho trattini dopo la prima cifra, dopo la quinta e dopo la nona ma non contano nulla ai fini del calcolo ma sono aiutati solo per facilitare la leggibilità dello stesso.
A causa dei problemi sopra descritti i codici ISBN vengono assegnati ormai con libertà dalle case editrici, usando il primo codice libero.
ISBN è un codice pesato dove i conti sono fatti in mod 11 (il più piccolo numero primo più grande di 10). Potrei avere come risultato 10 avendo mod 11 ma in quel caso uso X come checksum, come ultima cifra.

Esempio 2. *Prendiamo l'ISBN:*

$$0\text{-}1315\text{-}2447\text{-}x$$

e vogliamo verificare che sia effettivamente X. Si ha (con $\equiv$ indico i conti mod 11*):*

msg	sum	ssum
0	0	0
1	1	1
3	4	5
5	10	$20 \equiv 9$
2	$12 \equiv 1$	$21 \equiv 10$
4	5	$15 \equiv 4$
4	9	$13 \equiv 2$
7	$16 \equiv 5$	7
$X \equiv 10$	$15 \equiv 4$	$11 \equiv 0$

Avendo che effettivamente la somma mod 11 *fa 0 e quindi è verificato. Avrei inoltre, per l'algoritmo, controllando la check digit:*

$$temp = 7 + 5 = 12 \equiv 1$$

$$c = 11 - 1 = 10 \equiv X$$

3.2 Codici per correggere errori

In questo caso il ricevente non solo deve capire dove è l'errore ma deve anche correggerlo.

3.2.1 Codici rettangolari

Il modo più semplice per correggere errori è usare i **codici a correzione rettangolari**.
In questo caso il codice viene organizzato in modo logico in forma di rettangolo, ovviamente solo in modo logico/astratto. Indichiamo con $\circ$ i bit di messaggio e con x le check digit. In questa prima soluzione penso ai codici come se fossero a forma di rettangolo, con $m-1$ righe e $n-1$ colonne, con extra una colonna di check digit e una riga extra di check digit. Contando la riga di check digit e la colonna di check digit arriviamo a m righe e n colonne.

$$\begin{array}{cccccc} \circ & \circ & \circ & \dots & \circ & \mathrm{x} \\ \circ & \circ & \circ & \dots & \circ & \mathrm{x} \\ \circ & \circ & \circ & \dots & \circ & \mathrm{x} \\ \vdots & \vdots & \vdots & \ddots & \circ & \mathrm{x} \\ \circ & \circ & \circ & \dots & \circ & \mathrm{x} \\ \mathrm{x} & \mathrm{x} & \mathrm{x} & \mathrm{x} & \mathrm{x} & (\mathrm{x}) \end{array}$$

L'ultima check digit in fondo a destra è superflua, anche se a volte è lo xor di tutte le cifre di controllo, richiedendo che abbia numero pari di 1.
I codici rettangolari riescono a correggere un errore, uno e uno solo ma ovunque avvenga.
La check digit della prima riga fa la parità della prima riga, analogamente la seconda lo fa per la seconda riga etc...
Faccio un discorso analogo sulle colonne (la colonna 1 ha la check digit come prima cifra della riga di check digit etc...).
La check digit mi dice sempre se ho cifre pari tra cifre di messaggio e check digit.
Tramite la check digit a fine riga capisco che in una riga si ha un errore, e posso controllare lo stesso per la colonna. Identifico quindi il punto preciso in cui è avvenuto l'errore e semplicemente lo inverto, essendo binario.
Identificare l'errore singolo è dato dal fatto che solo una riga e solo una colonna avranno la rispettiva check digit "rotta" (nella colonna di check digit identifico la riga dell'errore mentre nella riga di check digit identifico la colonna dell'errore) e quindi posso riconoscere il preciso elemento che è errato.
Da questo discorso si capisce che la check digit in fondo a destra è in realtà inutile.
Ovviamente questo è garantito funzionare solo per un errore in quanto due errori potrebbero avere in comune riga o colonna e non saprei più capire dove siano gli errori. Potrebbe funzionare per più di un errore ma non sempre causa ambiguità e quindi questo metodo è **garantito per uno e un solo**

errore, ovunque si trovi.
Studiamo quindi la ridondanza, che ricordiamo essere in generale:

$$R = \frac{msg + check}{msg} = 1 + \frac{check}{msg}$$

Per il codice rettangolare si ha quindi:

$$R_{\square} = \frac{m \cdot n}{(n-1) \cdot (m-1)} = 1 + \frac{1}{m-1} + \frac{1}{n-1} + \frac{1}{(n-1) \cdot (m-1)}$$

Con quindi $\frac{1}{m-1} + \frac{1}{n-1} + \frac{1}{(n-1)\cdot(m-1)}$ che è il nostro *eccesso di ridondanza* e si ha che è ≥ 0, quindi in generale:

$$R_{\square} \geq 1$$

Ricordando che "i bit non sono colorati" potrei avere errori anche nelle check digit. Supponendo però che si ha al massimo un errore e supponendo di non avere, in quanto inutile, la check digit in basso a destra, si ha che al più trovo "rotta" o una riga (se ho errore un errore nella colonna di check digit) o su una colonna (se ho errore un errore nella riga di check digit), capendo così che ho "rotta" una check digit, sapendo anche quale.

Esempio 3. *Se devo spedire 24 bit posso rappresentarli in vari modi, secondo vari rettangoli (contando anche la check digit in basso a destra):*

- *una riga per 24 colonne (più la riga di controllo), con quindi 26 check digit*
- *2 righe per 12 colonne (più la riga di controllo), con quindi 15 check digit*
- *3 righe per 8 colonne (più la riga di controllo), con quindi 12 check digit*
- *4 righe per 6 colonne (più la riga di controllo), con quindi 11 check digit*
- *le simmetriche di quelle dette sopra*

Man mano che il numero di righe tende a quello di colonne (posto che, come nell'esempio precedente, non sempre può diventare uguale) si ha che le cifre di controllo diminuiscono. Tornando quindi alla formula della ridondanza vediamo che al diminuire delle check digit diminuisce l'eccesso di ridondanza $\frac{check}{msg}$ e quindi la ridondanza tende ad avvicinarsi a 1. Quindi per

scegliere n e m si cerca di fare si che siano il più uguali possibili, avendo meno cifre di controllo. Formalizzando quanto appena detto posso vedere la ridondanza la posso vedere come una funzione:

$$f(n, m)$$

avendo che $\frac{1}{m-1} + \frac{1}{n-1} + \frac{1}{(n-1)\cdot(m-1)}$ è quanto fatto dalla funzione. Si può quindi cercare il punto minimo di questa funzione trovando che è in $m = n$. Si può anche ipotizzare di poter aggiungere una cifra di messaggio per poter avere una rappresentazione con $n = m$, ma ovviamente dipende da caso a caso. Questo bit aggiuntivo a seconda del caso sarebbe 0 o 1. Si nota che aumentare il messaggio aumenta msg riducendo, a parità di check digit, $\frac{check}{msg}$, aiutando ad arrivare ad una ridondanza prossima a 1. **Non sempre ho modo di aggiungere tale bit**.
Il caso migliore è quindi con $n = m$ e si ha quindi, avendo un quadrato, il calcolo più preciso per la ridondanza:

$$R_{\square} = \frac{n^2}{(n-1)^2} = 1 + \frac{2}{(n-1)} + \frac{1}{(n-1)^2}$$

che è la ridondanza migliore che si può ottenere con i codici rettangolari.

3.2.2 Codici triangolari

Vediamo quindi i **codici a correzione triangolari**, dove la rappresentazione è appunto triangolare (a occhio una sorta di matrice triangolare).
In questo caso si ha un triangolo con un cateto di lunghezza n, ad esempio (con $n = 4$):

o o o o x
o o o x
o o x
o x
x

Ho quindi n check digit (*parte in dubbio*). Si ha infatti che le ci cifre di messaggio sono (per la **formula di Gauss**):

$$msg = \sum_{i=1}^{n-1} i = \frac{(n-1)\cdot n}{2}$$

(in altre parole il numero di diagonali che ho nella matrice meno uno).
Ho inoltre che il numero di cifre totali è:

$$tot = msg + check = \frac{(n-1)\cdot n}{2} + n = \frac{n(n+1)}{2}$$

che altro non è che la sommatoria fatta per calcolare msg ma con un'iterazione in più:

$$tot = msg + check = \sum_{i=1}^{n} i$$

Le check digit vengono calcolate dal mittente in base alle cifre sulla riga e sulla colonna della check digit. Quindi, ad esempio, per calcolare la check digit in grassetto di:

o o o o x
o o o x
o o **x**
o x
x

La calcola tramite le cifre di messaggio in verde:

o o o o x
o o o x
o o **x**
o x
x

Si nota che il bit di parità della prima riga è calcolato solo tramite la prima riga mentre il bit di parità dell'ultima riga viene calcolato solo tramite la prima colonna.
Il ricevente, per capire dove è il bit errato, verifica in quali check digit è coinvolto, che sono due check digit, e incrociando scopro quale sia il simbolo errato. Vedendo in pratica si supponga errato il bit in rosso:

o o o o x
o o o x
o o x
o x
x

Esso è coinvolto nelle seguenti cifre di parità in grassetto, calcolate, oltre che grazie alla cifra in rosso anche da quelle in verde:

o o o o x
o o o **x**
o o **x**
o x
x

Mentre le altre cifre di parità saranno corrette. Il destinatario può quindi scoprire quale sia il bit che è stato modificato per poterlo poi correggere.
Passiamo quindi alla ridondanza del codice triangolare. Si ha che:

$$R_{\triangle} = \frac{tot}{msg} = \frac{\frac{n(n+1)}{2}}{\frac{(n-1)n}{2}} = \frac{n+1}{n-1} = \frac{(n-1+2)}{n-1} = 1 + \frac{2}{n-1}$$

Confronto questa ridondanza con quella dei migliori codici rettangolari, ovvero quella dei codici quadrati, dove vedo che si ha un termine positivo in più nei codici quadrati, ovvero $\frac{1}{(n-1)^2}$, che essendo positivo può solo aumentare la ridondanza. Si ha quindi che:

$$R_{\triangle} < R_{\square}$$

e quindi i codici triangolari sono migliori di quelli rettangolari, anche dei migliori tra quelli rettangolari (quelli quadrati), e hanno una sola check digit per riga e una sola per colonna.
L'unico limite per avere un codice triangolare è quello di avere un messaggio di n cifre che permette l'espressione $\frac{(n-1)\cdot n}{2}$, ovvero che sia rappresentabile da un triangolo (per fare tornare i conti posso comunque aggiungere uno 0, per esempio), avendo n lunghezza del cateto.

3.2.3 Codici cubici

Per cercare di abbassare ancora meno la ridondanza cerchiamo la disposizione geometrica più conveniente. Si è provato quindi ad aumentare le dimensioni dello spazio in cui ci troviamo, pensando ad un cubo, ipotizzando di avere una cifra di controllo (posta in basso a destra per il piano) che controlla tutto un piano (contando che ho piani per ognuna delle tre dimensioni). Si ha quindi un bit di parità per ogni piano (che è ciò che rappresenta le cifre di messaggio) con cui posso tagliare il cubo. Nel complesso si ha quindi un intero spigolo del cubo che rappresenta le check digit che sono rappresentati da i piani in

	Quadrato		Triangolare		Cubico	
	messaggio	controllo	messaggio	controllo	messaggio	controllo
n	$(n-1)^2$	$2n-1$	$\frac{n(n-1)}{2}$	n	n^3-3n+2	$3n-2$
2	1	3	1	2	4	4
3	4	5	3	3	20	7
4	9	7	6	4	54	10
5	16	9	10	5	112	13
6	25	11	15	6	200	16
7	36	13	21	7	324	19
8	49	15	28	8	490	22
9	64	17	36	9	704	25
10	81	19	45	10	972	28

Tabella 3.1: Esempio di tabella di confronto tra codici quadrati, triangolari e cubici

una certa direzione. In totale posso sezionare in tre direzioni avendo quindi 3 spigoli di check digit che "proteggono" tutte le cifre di messaggio contenute nel cubo (che ha lato $n-1$ più la check digit).

In maniera approssimata, su circa n^3 posizioni si hanno $3(n-1)+1 = 3n-2$ check digit. Si ha quindi che la ridondanza è:

$$R_{\square} = \frac{(n-1)^3 + 3n - 2}{(n-1)^3} = 1 + \frac{3n-2}{(n-1)^3} \simeq 1 + \frac{3}{n^2}$$

Posso pensare quindi pensare di aumentare ancora le dimensioni, considerando cubi a quattro dimensioni (e non mi serve disegnarlo visto che devo solo "pensarlo"), avendo:

- $(n-1)^4$ cifre di messaggio
- $4(n-1)+1 = 4n-3$ check digit

ottenendo la seguente ridondanza:

$$R_{\square 4\text{-}dim} = 1 + \frac{4n-3}{(n-1)^4} \simeq 1 + \frac{4}{n^3}$$

(*dove praticamente si è ignorato il* -3 *nella formula della check digit*).
Posso pensare di andare quindi in k dimensioni, avendo:

- $(n-1)^k$ cifre di messaggio
- $k(n-1)+1 = kn-k+1$ check digit

ottenendo la seguente ridondanza, con k che è fissato e quindi costante:

$$R_{k\text{-}dim} = 1 + \frac{kn - k + 1}{(n-1)^k} \simeq 1 + \frac{k}{n^{k-1}}$$

Possiamo così, facendo crescere k e l'eccesso di ridondanza, ottenere una figura sempre migliore.
Ma anche se la ridondanza è sempre migliore vedo che a denominatore dell'eccesso di ridondanza $\frac{k}{n^{k-1}}$ trovo un polinomio elevato alla dimensione meno uno. Vedremo che nei **codici di Hamming**, per correggere un errore, il *gap* è esponenziale al variare delle dimensioni. Inoltre il *codice di Hamming* è **ottimale**.

3.2.4 Codici di Hamming

Hamming si pone di trovare un codice per correggere un errore nel modello del rumore bianco, avendo che sia però un **codice ottimale**, *ovvero che a parità di cifre inviate deve usare il minimo numero di check digit.* Ho sempre un pacchetto di n bit che sono di due tipi:

1. k bit di cifre di messaggio msg
2. m bit di check digit $check$

Per calcolare le check digit usa delle equazioni di controlli di parità, con una cifra di messaggio che nel complesso serve a calcolare più di una check digit (come nei codici legati alle figure numeriche). Si avranno quindi m equazioni di parità del tipo, avendo $c_1, \ldots, c_n$ cifre di parità e $x_i, \cdots x_n$ cifre di messaggio:

$$c_i = x_j \oplus x_k \oplus x_w \cdots$$

Tutte queste equazioni devono essere **linearmente indipendenti**, in modo che ogni check digit abbia informazioni differenti. Supponiamo infatti:

$$c_1 = x_1 \oplus x_2 \oplus x_5 \oplus x_7$$

$$c_2 = x_5 \oplus x_7 \oplus x_8 \oplus x_9$$

Ma in realtà c_i è una della x_j della formula infatti:

$$c_1 \oplus x_1 \oplus x_2 \oplus x_5 \oplus x_7 = 0$$

e quindi:

$$x_1 \oplus x_2 \oplus x_5 \oplus x_7 = c_1$$

Scriviamo quindi:

$$c_1 \to x_1 \oplus x_2 \oplus x_5 \oplus x_7 = 0$$

$$c_2 \to x_5 \oplus x_7 \oplus x_8 \oplus x_9 = 0$$

In generale le equazioni mi danno 0 se il numero di 1 è pari.
In ogni formula si assume di avere una sola cifra di messaggio.
Faccio poi lo $\oplus$ tra c_1 e c_2 bit a bit, ottenendo (avendo tipo x_5 da entrambe le parti si semplifica):

$$c \to x_1 \oplus x_2 \oplus x_8 \oplus x_9 = 0$$

Si supponga ora di avere una $c_3 = c$ ma avrei uno spreco di fatica in quanto le informazioni delle prime due sarebbero uguali alla terza, avendo infatti *dipendenza lineare*.
Ricordiamo che:

$$a_1\vec{v_1} + a_2\vec{v_2}, a_1, a_2 \in \mathbb{R}, \vec{v_1}, \vec{v_2} \in \mathbb{R}^n$$

con $\mathbb{R}^n$ che è uno **spazio vettoriale**.
Fare **combinazioni lineari** in questo caso prevede la somma come il $\oplus$.
Rappresento le equazioni di parità dell'esempio sopra come un vettore di $\{0,1\}^n$:

$$(1,1,0,0,1,0,1,0,0)$$

$$(0,0,0,0,1,0,1,1,1)$$

e facendo lo *xor* (che è la somma mod, 2 bit a bit) ottengo:

$$(1,1,0,0,0,0,0,1,1)$$

Si ha ora il prodotto scalare in $\{0,1\}^n$:

$$0 \cdot \vec{v} = \vec{0}$$

$$1 \cdot \vec{v} = \vec{v}$$

Quindi nel nostro caso:

$$a_1\vec{v_1} + a_2\vec{v_2}, a_1, a_2 \in \{0,1\}, \vec{v_1}, \vec{v_2} \in \{0,1\}^n$$

Quindi 0 mi dice di non considerare il vettore e uno di considerarlo, indicando il sottoinsieme di elementi da sommare. Si ha che $\{0,1\}^n$ rispetto alla somma come definita e al prodotto scalare come definito è uno **spazio vettoriale**.
Quindi, tornando a Hamming, possiamo capire, grazie a questi ragionamenti, l'indipendenza lineare tra le equazioni di parità. Con m bit di controllo si cerca di capire quante condizioni d'errore si riesce a rappresentare. Con m

cifre di controllo si hanno 2^m possibili combinazioni/configurazioni. Ogni configurazione mi deve identificare un errore diverso per poter fare anche correzione. Mi serve però anche una configurazione che indichi il *non avere errori*. Si ha quindi che si hanno:

- $2^m - 1$ condizioni di errore
- una condizione di non errore

Gli errori possono avvenire ovunque, anche nelle check digit, ma con al massimo un errore (o nella prima posizione o nella seconda etc...). SI hanno quindi n possibili condizioni d'errore. Si ha quindi che:

$$2^m \geq n + 1$$

ovvero le configurazioni che posso fare con m cifre di controllo deve indicarmi le n possibili condizioni d'errore più una per l'assenza di errore. Nel caso dei codici di Hamming "propriamente detti" si avrebbe:

$$2^m = n + 1$$

usando tutte le combinazioni in quanto il $>$ implicherebbe che sto "sprecando" qualche configurazione delle cifre di controllo. **Non sempre posso usare "$=$".**
Posso quindi capire quante siano le cifre di controllo.

Esempio 4. *Si supponga di voler spedire $k = 4$ cifre di messaggio. Ci chiediamo di calcolare m:*

$$2^m \geq n + 1$$

ma quindi, avendo $n = m + k$:

$$2^m \geq m + k + 1$$

Mi serve quindi l'm più piccolo possibile che risolva la disequazione con $k = 5$:

$$2^m \geq m + 5$$

Non si ha però una soluzione analitica ma 2^m cresce più velocemente di $m+5$ ($m = 0$ non ha senso ma lo si mette, si hanno comunque tecniche anche per non partire da 1):

m	2^m	$m+5$
0	1	5
1	2	6
2	4	7
3	8	8

quindi $m = 3$ è il più piccolo valore che risolve, risolvendo per di più con $=$ e non $\geq$. Per 4 bit di messaggio mi servono 3 cifre di controllo. Ho quindi $n = 7$.

In generale per correggere un errore devo sapere in che posizione è, sapendo l'indice da 1 a n indicante tale posizione nel pacchetto. Il metodo che lo calcola può restituire 0, indicante che non si ha errore.
Faccio una tabella con posizione/binario, ad esempio per l'esempio precedente

pos	bin
1	001
2	010
3	011
4	100
5	101
6	110
7	111
	$c_3c_2c_1$

Ho che i bit hanno m cifre, e ho $c_3c_2c_1$ (si parte da destra) come le cifre di parità ottenute dalle tre colonne di binari.
Ipotizzo un errore in posizione 5. Ho che, mettendo le x_i dove si ha 1:

$$c_1 \to x_1 \oplus x_3 \oplus x_5 \oplus x_7 = 0$$

ma ho problemi con x_5, avendo l'equazione pari a 1.

$$c_2 \to x_2 \oplus x_3 \oplus x_6 \oplus x_7 = 0$$

dove non ho problemi con x_5.

$$c_3 \to x_4 \oplus x_5 \oplus x_6 \oplus x_7 = 0$$

ma ho problemi con x_5, avendo l'equazione pari a 1.
Si ha quindi che:

$$c_1 \to x_1 \oplus x_3 \oplus x_5 \oplus x_7 = 1$$

$$c_2 \to x_2 \oplus x_3 \oplus x_6 \oplus x_7 = 0$$

$$c_3 \to x_4 \oplus x_5 \oplus x_6 \oplus x_7 = 1$$

Che quindi mi mostra dopo l'uguale la posizione in cui è avvenuto l'errore. I bit letti dal basso verso l'alto sono 101 e questo è detto **sindrome**, che

quindi nei codici di Hamming è la posizione dell'errore in codice binario.
Se la *sindrome* è 0 non ho errori (è come se avessi 000 nella prima riga della tabella).
Il ricevente è sistemato ma manca il mittente che deve calcolare gli m bit di controllo e spedire. Usando la tecnica di usare nelle equazioni le x_i relative agli 1 mi genera equazioni linearmente indipendenti.
Manca capire dove mettere la check digit. Le cifre di controllo vengono messe nelle posizioni in cui si ha un 1 solo, ovvero 001, 010 e 100, in modo che c_1 è solo nella prima equazione, c_2 nella seconda e c_3 nella terza, non dovendo risolvere in realtà il sistema lineare. Si quindi, nell'esempio sopra:

$$c_1 \oplus x_3 \oplus x_5 \oplus x_7 = 0$$

$$c_2 \oplus x_3 \oplus x_6 \oplus x_7 = 0$$

$$c_3 \oplus x_5 \oplus x_6 \oplus x_7 = 0$$

ottenendo quindi il pacchetto.

$$pacchetto = c_1 c_2 m_1 c_3 m_2 m_3 m_4$$

e quindi:

$$c_1 \oplus m_1 \oplus m_2 \oplus m_4 = 0$$

$$c_2 \oplus m_1 \oplus m_3 \oplus m_4 = 0$$

$$c_3 \oplus m_2 \oplus m_3 \oplus m_4 = 0$$

ma isolando le c_i, sommando da entrambe le parti dell'equazione:

$$c_1 = m_1 \oplus m_2 \oplus m_4$$

$$c_2 = m_1 \oplus m_3 \oplus m_4$$

$$c_3 = m_2 \oplus m_3 \oplus m_4$$

Si noti che le posizioni delle cifre di controllo c_i sono potenze di 2 $2^0, 2^1, 2^2, \ldots$ e all'aumentare delle cifre della grandezza si ha crescita logaritmica delle check digit, che quindi controllano un numero esponenziale di cifre di messaggio.

Esempio 5. *Scelgo un messaggio di 4 bit:*

$$msg = 1011$$

quindi:

$$pacchetto = c_1 c_2 1 c_3 011$$

So che, per le formule sopra (ma lo posso vedere anche col discorso di avere periodicità nella tabella di bit, pensando a come li scrivi quando fai le tabelle di verità):

$$c_1 = 0$$

$$c_2 = 1$$

$$c_3 = 0$$

quindi:

$$pacchetto = 0110011$$

che viene inviato ma al destinatario arriva:

$$pacchetto = 0110111$$

quindi con il bit in $i = 5$ errato. Si rifanno i conti e si ottiene, calcolando la sindrome:

$$s_1 = 1$$

verificata vedendo i bit dispari.

$$s_2 = 0$$

verificata vedendo a coppie alternate ma saltando il primo bit (che sarebbe la coppia di 0 se avessi 000 in cima alla tabella, torna quindi comodo partire dal fondo della tabella per fare questi ragionamenti).

$$s_3 = 1$$

verificata vedendo gli ultimi 4 bit.
Avendo sindrome $s = s_3 s_2 s_1 = 101$ che indica proprio la posizione 5.
Vediamo invece se arriva:

$$pacchetto = 0010011$$

quindi con il bit in $i = 2$ errato (e sarebbe una check digit). Si ha che:

$$s_1 = 0$$

$$s_2 = 1$$

$$s_3 = 0$$

Avendo sindrome $s = s_3 s_2 s_1 = 010$ che indica proprio la posizione 2.
Supponiamo ora che arrivi senza errori:

$$pacchetto = 0110011$$

Si ha:

$$s_1 = 0$$

$$s_2 = 0$$

$$s_3 = 0$$

I pacchetti di 7 bit che producono sindrome nulla sono quindi $2^4 = 16$*, avendo che posso fare* 2^4 *modi diversi di avere i 4 bit di messaggio ma ciascuno ha una sola tripla, 000, di check digit che mi dice che il pacchetto è valido.* Generalizzeremo poi questa cosa, ovvero che si hanno:

- n bit di pacchetto
- $\log_2 n$ check digit
- $n - \log_2 n$ cifre di messaggio
- $2^{n-\log_2 n}$ combinazioni possibili delle cifre di messaggio per avere un pacchetto senza errori

Avere due errori rende impossibile capire dove siano, rendendo impossibile la correzione.

Esempio 6. *Riprendendo l'esempio sopra si ipotizzi arrivi:*

$$pacchetto = 0010111$$

con errori in 2 e 5.
Si ha per la sindrome:

$$s_1 = 1$$

$$s_2 = 1$$

$$s_3 = 1$$

Si avrebbe posizione in 111 *ovvero 7 (e non è un caso sia 5+2 ma la cosa non ci aiuterebbe a capire dove sia l'errore), a riprova che non si riesce ad identificare due errori. Non solo, si identifica un posto corretto come errato e lo corregge, introducendo un ulteriore errore.*

La probabilità di avere due errori è comunque a $\frac{1}{2^{4096}}$ e quindi è 0, "è la stessa probabilità che un tornado assembli i pezzi sparsi di un aereo al punto di farlo funzionare".
I codici di Hamming con $=$ e non $\geq$ sono quelli tali per cui:

$$n = 2^m - 1$$

Sul file di esercizi nella pagina del corso esempio con $m = 4$ a *pagina 12, potrebbe esserci un errore di conto sulla ridondanza.* Guardare anche per spunti teorici.
Studiamo anche la ridondanza. Ricordando $2^m \geq n+1$ e nei codici "migliori" $2^m = n+1$ possiamo dire $2^m \simeq n$ e quindi $m \simeq \log_2 n$ (e con questo m posso riscrivere $k = n - \log_2 n$). Ne segue che:

$$R = 1 + \frac{check}{msg} = 1 + \frac{m}{k} = 1 + \frac{\log_2 n}{n - \log_2 n} \simeq 1 + \frac{\log_2 n}{n}$$

(*facendo un'approssimazione molto "spinta"*).
Sul file di esercizi nella pagina del corso esempio con $m = 2$ a pagina 13, occhio che non è $\frac{2}{3}$ ma $\frac{2}{1}$ nella ridondanza. Guardare anche per spunti teorici (manca la parte di esercizio in se calcolando la sindrome).
Vediamo quindi l'interpretazione geometrica del codice di Hamming, ricordando che n bit entrano nel canale ed escono n bit $\in \{0,1\}^n$. Il messaggio M è quindi $M \subseteq \{0,1\}$. Cerchiamo di capire come assegnare che un messaggio sia valido o meno.
Posso vedere $\{0,1\}^n$ come uno **spazio vettoriale** dove i vettori sono $\vec{x} = (x_1, x_2, \ldots, x_n), \forall x_i \in \{0,1\}$ ovvero $x_i \in \mathbb{Z}_2$. Due vettori in questo spazio, per definizione di spazio vettoriale, possono essere sommati. Preso anche $\vec{x} = (y_1, y_2, \ldots, y_n), \forall y_i \in \{0,1\}$ la somma altro non è che lo *xor* componente per componente $\vec{x} + \vec{y} = (x_1 \oplus y_1, x_2 \oplus y_2, \ldots)$.
Preso uno scalare $a \in \mathbb{Z}_2$ ho che $a\vec{x} = (ax_1, ax_2 \ldots)$ con il prodotto scalare che è praticamente l'*and*. Si ha che:

$$a\vec{x} = \begin{cases} \vec{0} & \text{se } a = 0 \\ \vec{x} & \text{se } a = 1 \end{cases}$$

Vediamo i messaggi come **vertici di un n-cubo** ma non tutti i messaggi sono validi. Ci serve un modo per capire quali sono validi per rendere difficile che un errore mi porti da un messaggio valido ad un altro.

Definizione 8. *Definiamo **distanza di Hamming** come il numero di posizioni in cui due vettori $\vec{x}$, $\vec{y}$ differiscono, cioè per cui $x_1 \neq y_i$. Questa è una vera e propria distanza in $\{0,1\}^n$. Tale distanza si indica con:*

$$d(\vec{x}, \vec{y})$$

con:

$$d : \{0,1\}^n \times \{0,1\}^n \to \mathbb{R}$$

Si hanno tre proprietà:

- $d(\vec{x}, \vec{y})$ *sse* $\vec{x} = \vec{y}$
- $d(\vec{x}, \vec{y}) = d(\vec{y}, \vec{x})$
- $d(\vec{x}, \vec{z}) \leq (\vec{x}, \vec{y}) + d(\vec{y}, \vec{z})$, ***diseguaglianza triangolare***

e quindi la ***distanza di Hamming*** *è una* ***metrica*** *per lo spazio* $\{0,1\}^n$. *Spesso* $d(\vec{x}, \vec{y})$ *si indica con* $d_h(\vec{x}, \vec{y})$.

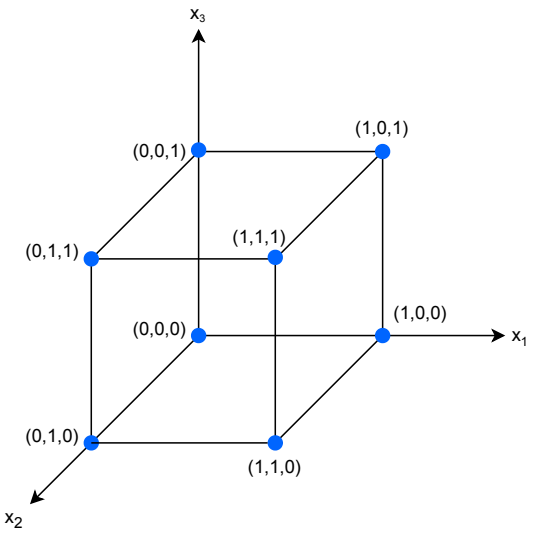
Definizione 9. *Definiamo* ***peso di Hamming*** *come il numero di bit uguali a 1 e lo si indica con:*

$$w_j(\vec{x})$$

Avendo che:

$$w_j(\vec{x}) = \sum_{i=1}^{n} x_1 = d_h(\vec{x}, \vec{0})$$

Esempio 7. *Prendiamo* $n = 3$.
Disegnano supponiamo di avere qualcosa del tipo:



Prendo i vertici (x_1, x_2, x_3)*:*

- $(0, 0, 0)$
- $(1, 0, 0)$
- $(0, 1, 0)$
- $(0, 0, 1)$

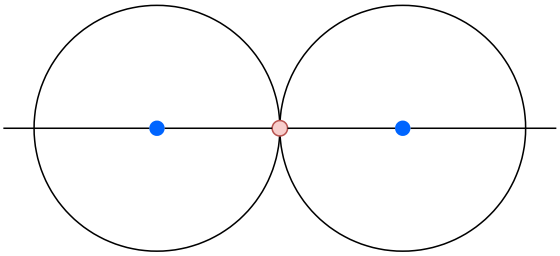
- $(1, 1, 0)$
- $(0, 1, 1)$
- $(1, 0, 1)$
- $(1, 1, 1)$

Questi sono i nostri messaggi.
***Oltre** $n = 3$ **farei lo stesso ma non saprei disegnarlo, avrei** n **lati che partono dall'origine appoggiati agli** n **assi**.*
Suppongo di spedire 010 *e che si abbia un errore che porti a* 110 *e sul cubo si nota che ci ha fatto spostare lungo un lato.*
Suppongo che da 010 *si arrivi a* 100*, con due errori, si nota che ho due lati visitati sul cubo.*
*Scegliamo quindi come messaggi validi punti che sono abbastanza distanti tra loro. Sicuramente quelli che si ottengono con un solo cambiamento non sono validi, parto quindi dal messaggio valido e impongo che tutti quelli che sono a **distanza 1** (parlando di **distanza di Hamming**) sul cubo, ovvero tutti quelli che raggiungo con un solo lato, non siano validi.*
Essendo in** $\{0, 1\}^n$ **in realtà esistono solo i vertici del cubo, i lati servono solo per capire meglio.

Generalizziamo meglio.
Se ho la distanza di Hamming e varie caratteristiche del codice ho che:

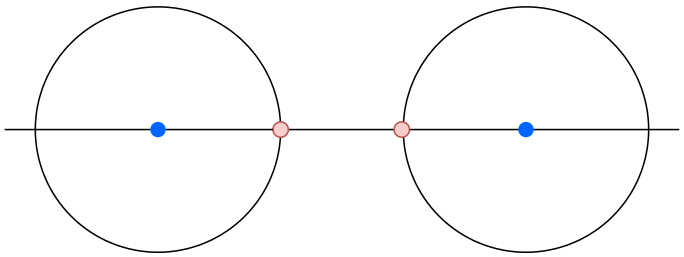
- se ho due codeword valide che hanno distanza 1 allora il codice ha capacità di **0 detection e 0 correction** in quanto potrei passare con un solo errore ad un messaggio valido
- se ho due codeword valide che hanno distanza 2 (e non esistono due codeword valide con distanza minore di 2) allora il codice ha capacità di **1 detection e 0 correction** in potrei avere più codeword valide a distanza di 1 da quella errata e quindi non saprei come fare correzione, non sapendo dove si trovi il bit errato. In ogni caso un errore viene sempre riconosciuto, anche se non può essere corretto. Se ho due errori potrei finire nuovamente in una codeword valida ma si sta assumendo che avendo scelto questo codice si sappia che la probabilità che avvengano due errori sia pressoché nulla. Il *controllo di parità semplice* è di questo tipologia.

Graficamente avrei una cosa del tipo, avendo in blu le codeword valide e in rosso quelle non valide:



Notando come non sarei in grado di capire a che codeword valida ricondurmi.

- se ho due codeword valide che hanno distanza 3 (e non esistono due codeword valide con distanza minore di 3) allora il codice ha capacità di **2 detection e 1 correction**, riuscendo a correggere un errore. Se avessi due errori riconoscerei come codeword valida una a distanza 1 da quella errata che però non è quella giusta, introducendo un nuovo errore. I *codici di Hamming* sono di questo tipo. Graficamente avrei una cosa del tipo, avendo in blu le codeword valide e in rosso quelle non valide:



Notando che con un solo errore sarei in grado di capire a che codeword ricondurmi

- se ho due codeword valide che hanno distanza 4 (e non esistono due codeword valide con distanza minore di 4) allora il codice ha capacità di **3 detection e 1 correction**, riuscendo a correggere un errore. Se avessi due errori riconoscerei avrei più codeword valide alla stessa distanza. Se avessi tre errori varrebbe quanto detto nel caso di distanza 2

- se ho due codeword valide che hanno distanza 5 (e non esistono due codeword valide con distanza minore di 5) allora il codice ha capacità di **4 detection e 2 correction**, per gli stessi discorsi fatti avendo distanza 3

Si nota quindi che al crescere della distanza di Hamming minima tra due codeword cresce la capacità di detection e correction.
Possiamo ipotizzare che la codeword valida venga avvolta da una sfera di raggio pari a r avendo che r sia tale per cui le sfere di due codeword valide non si sovrappongano. Possiamo notare che in base alle osservazioni fatte sopra r è il valore della capacità di detection.
Una sfera centrata nel punto $\vec{x}$ di raggio r è l'insieme dei punti $\vec{y} \in \{0,1\}^n$ tale che la distanza da $\vec{x}$ è minore di r, ovvero:

$$S(\vec{x}, r) = \{\vec{y} \in \{0,1\}^n | \, d_h(\vec{x}, \vec{y}) \leq r\}$$

Si noti che in $\{0,1\}^n$ somma e sottrazione bit a bit portano allo stesso risultato.
Possiamo dire che:

$$output = input + errore$$

dove *errore* è un vettore rappresentante l'errore (avendo 1 dove ho l'errore).
Quindi:

$$errore = output - input$$

Esempio 8. *Tornando all'esempio sopra ipotizzo entri* 011 *ed esca* 111*, avendo quindi* $d_h = 1$.
Sommo i due vettori:

$$011 \oplus 111 = 100$$

Avendo che 100 *è l'errore.*

Esempio 9. *Si vuole costruire un codice che consenta di correggere un errore, per poi confrontarlo con il codice di Hamming.*
Ci servono quindi codeword valide ad almeno distanza 3, prendendo messaggi validi e circondarli di sfere di raggio 1 tali che non si abbiano sovrapposizioni con altre sfere.
Si ha $\{0,1\}^n$ *come spazio vettoriale dove si hanno* 2^n *elementi/vettori.*
In generale se prendo il volume dello spazio e lo divido per il volume di una sfera ho il numero di sfere che posso metterci:

$$\frac{V_{spazio}}{V_{sfera}} \geq numero\ massimo\ di\ sfere$$

Pongo k uguale al numero di bit dei messaggi validi (non pari a n ma prendo quindi una porzione di messaggio rappresentante il messaggio valido, ai quali poi aggiungerei le m check digit).
Rivedo quindi la formula, sapendo che il volume dello spazio è il numero di vettori di quello spazio, che le sfere abbiano raggio 1*, avendo che hanno volume* $n+1$ *(il vettore rappresentante originale più un vettore per ogni possibile vettore ottenuto da quello originale con un solo errore, avendo quindi* $\binom{n}{1}$ *più il messaggio valido, ovvero* $n+1$*):*

$$\frac{2^n}{n+1} \geq 2^k$$

sapendo che $n = k + m$ *si ha che:*

$$\frac{2^{k+m}}{n+1} \geq 2^k$$

$$\frac{2^k 2^m}{n+1} \geq 2^k$$

$$2^m \geq n+1$$

Arrivando allo stesso risultato di Hamming.

Se volessi sfere di raggio maggiore di 1, prendendo per esempio con codeword distanti almeno 5 e raggio pari a 2. Avendo raggio pari a 2 avrei il centro della sfera, tutti i punti distanti 1, ovvero potendo cambiare due bit n, e tutti i punti distanti due (avendo due cambiamenti di bit), ovvero:

$$V_{sfera} = 1 + n + \binom{n}{2} = 1 + n + \frac{n(n-1)}{2}$$

$$\frac{2^n}{1 + n + \frac{n(n-1)}{2}} \geq 2^k$$

sapendo che $n = k + m$ si ha che:

$$\frac{2^{k+m}}{1 + n + \frac{n(n-1)}{2}} \geq 2^k$$

$$\frac{2^k 2^m}{1 + n + \frac{n(n-1)}{2}} \geq 2^k$$

$$2^m \geq 1 + n + \frac{n(n-1)}{2}$$

Avendo una generica sfera di raggio r avrei:

$$V_{sfera} = 1 + n + \binom{n}{2} + \binom{n}{3} + \cdots + \binom{n}{r} = \sum_{k=0}^{n} \binom{n}{k}$$

Nella realtà, con messaggi molto grandi, si usano i cosiddetti ***low density parity codes****, che consiste, per proteggere il pacchetto di n bit, nel trovare il minor numero di equazioni di parità atte a proteggere il messaggio (meno equazioni implica meno circuiti e quindi meno costo). Si procede per euristiche per capire quante e quali.*

3.3 Codifica di Sorgente

Finora si è parlato di **codifica di canale**, passiamo quindi ad approfondire la **codifica di sorgente**, cercando anche in questo caso il codice ottimale (con la lunghezza media minima).
Avendo magari una distribuzione non uniforme per la distribuzione dei caratteri non ha senso usare una **codice a blocchi** ma si usa un **codice a lunghezza variabile** (*riguardare quanto detto nella prima sezione di questi appunti*).
Ci si propone quindi di minimizzare, parlando di codici a lunghezza variabile:

$$L = \sum_{i=1}^{1} p_i l_i$$

Bisogna prima fare un discorso sull'univocità dei codici.

Esempio 10. *si supponga di avere le seguenti codeword binarie per 4 simboli* s_i*:*

- $s_1 \to 0$
- $s_2 \to 01$
- $s_3 \to 11$
- $s_4 \to 00$

Suppongo che il ricevente riceva 0011*. Cerco di capire che sequenza ha ricevuto. Deve calcolare una:*

$$cod^{-1} : \Gamma^* \to S$$

Ma ci sono delle ambiguità. Potrebbe essere;

- $s_4 s_3$
- $s_1 s_1 s_3$

Non è quindi in grado di trovare una sequenza di simboli univoca.
Ne segue che il codice ***non è univocamente decodificabile****.*

Si richiede quindi che la funzione *cod*, quindi il nostro codice, **deve essere univocamente decodificabile**.
Un algoritmo che riconosce che un codice è univocamente decodificabile non è banale.

Esempio 11. *Si supponga di avere le seguenti codeword binarie per 4 simboli* s_i*:*

- $s_1 \to 0$
- $s_2 \to 01$
- $s_3 \to 011$
- $s_4 \to 111$

Si dimostra che questo codice è ***univocamente decodificabile****, esistendo un'unica possibile divisione in codeword di ogni stringa che riceve il ricevente. Suppongo che il ricevente riceva:*

$$0111111111$$

Ma anche solo all'inizio potrei avere sia s_1 *che* s_2 *che* s_3 *ma una volta che è finito posso partire dal fondo raggruppando gli* 1 *tre a tre e mi accorgo che ho un solo modo:*

$$0 \;\; 111 \;\; 111 \;\; 111$$

$$s_1 s_4 s_4 s_4$$

Bisogna quindi aspettare la fine della trasmissione*.*

Lato elettronico per fare questa operazione uso un **contatore elettronico**.
D'altro canto aspettare la fine non è "comodo" e si vorrebbe analizzare i simboli man mano che arrivano.
Si definiscono quindi due categorie di codici a lunghezza variabile:

- **non univocamente decodificabili**
- **univocamente decodificabili**

e suddividiamo gli univocamente decodificabili in:

- **istantaneo**, dove appena arriva una codeword il ricevente sa che è arrivata completamente e che è unica
- **non istantaneo**

Esempio 12. *Trasformiamo l'ultimo esempio in un codice istantaneo. Si hanno:*

- $s_1 \to 0$
- $s_2 \to 01$
- $s_3 \to 011$
- $s_4 \to 111$

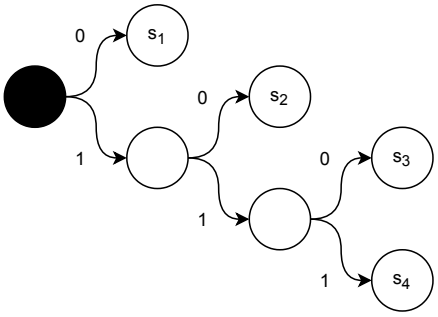
ma non andrebbero bene (per il problema dell'inizio del messaggio). D'altro canto:

- $s_1 \to 0$
- $s_2 \to 10$
- $s_3 \to 110$
- $s_4 \to 111$

ottenuto con la reverse delle stringhe, è un codice istantaneo con codeword di lunghezza pari a quelle precedenti.
Posso facilmente vedere che qualsiasi cosa arrivi posso capire che codeword è, appena l'intera codeword viene letta. Non devo aspettare l'intero messaggio ma al più due simboli.

Nell'esempio abbiamo usato gli 0 come *end of codeword*, infatti questo codice è detto **comma code**, avendo che il decodificatore vede che le codeword, tranne l'ultima, terminano per 0. Quindi o arriva uno zero terminando la codeword o arrivano 3 uni che terminano il messaggio.
Potrei inoltre vedere la decodifica come un albero di decodifica, in quanto ogni volta che arriva un simbolo posso escludere certe codeword.

Vediamo l'esempio relativo all'esempio precedente:

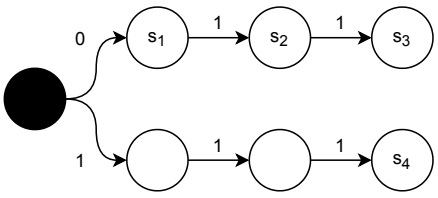


Avendo che ogni cammino dalla radice ad una voglia è una codeword con la foglia etichettata con il simbolo relativo alla codeword. Si ha quindi un sottoalbero per ogni simbolo di Γ (quindi dalla radice escono $|\Gamma|$ archi) e da ogni nodo interno esce un numero di archi minore o uguale al numero di simboli. Il ricevente sfrutta l'albero di decodifica.
Posso anche usare l'albero per costruire il codice istantaneo.
Inoltre dato che un codice istantaneo è associato ad un albero di codifica non succede mai che una codeword sia prefissa di un'altra (e lo si vede non prendo avere due foglie associate a due codeword una prefissa dell'altra).

Esempio 13. *Vediamo l'albero di decodifica di un codice non istantaneo. Si hanno:*

- $s_1 \to 0$
- $s_2 \to 01$
- $s_3 \to 011$
- $s_4 \to 111$

Si ha:



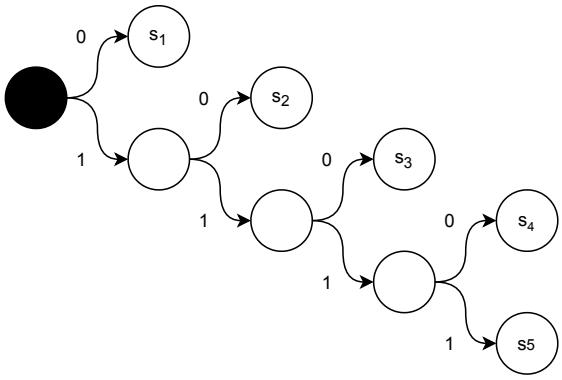
Avendo foglie che sono prefisse di altre.

Se devo costruire un codice istantaneo posso quindi usare un **comma code** o costruire un albero di decodifica che sia il più possibile simile ad un albero completo.

Esempio 14. *Si supponga di avere le seguenti codeword binarie per 5 simboli s_i:*

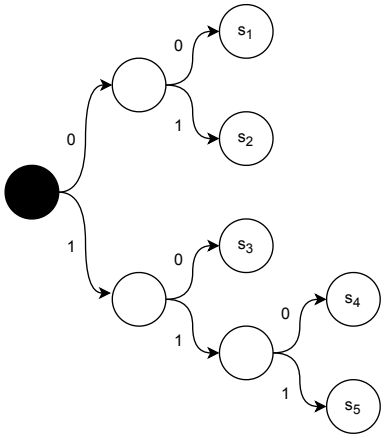
- $s_1 \to 0$
- $s_2 \to 10$
- $s_3 \to 110$
- $s_4 \to 1110$
- $s_4 \to 11110$

Si ha quindi:



(e si nota come si possa generalizzare l'albero per un comma code).

Esempio 15. *Avendo l'albero:*



Riconosco il codice:

- $s_1 \to 00$
- $s_2 \to 01$
- $s_3 \to 10$
- $s_4 \to 110$
- $s_5 \to 111$

Esempio 16. *Avendo il codice 1:*

- $s_1 \to 0$
- $s_2 \to 10$
- $s_3 \to 110$
- $s_4 \to 1110$
- $s_5 \to 11110$

e il codice 2:

- $s_1 \to 00$
- $s_2 \to 01$
- $s_3 \to 10$
- $s_4 \to 110$
- $s_5 \to 111$

assegno ad entrambi le probabilità:

- $p_1 \to 0.9$
- $p_2 \to 0.025$
- $p_3 \to 0.025$
- $p_4 \to 0.025$
- $p_5 \to 0.025$

Ho che, per il primo codice:

$$L_1 = 0.9 \cdot 1 + 0.025 \cdot 2 + 0.025 \cdot 3 + 0.025 \cdot 4 + 0.025 \cdot 4 = 1.225$$

Avendo che mediamente per rappresentare un simbolo della sorgente mi servono 1.225 *bit.*
Passo al secondo codice:

$$L_2 = 0.9 \cdot 2 + 0.025 \cdot 2 + 0.025 \cdot 2 + 0.025 \cdot 3 + 0.025 \cdot 3 = 2.05$$

Avendo che mediamente per rappresentare un simbolo della sorgente mi servono 2.05 *bit.*
Ne segue, avendo $L_1 < L_2$, per questa sorgente con queste probabilità è meglio il primo codice, il comma code.
Se avessi avuto probabilità uniforme ($p_i = 0.2$) avrei avuto:

$$L_1 = 2.8$$

$$L_2 = 2.4$$

Essendo meglio il secondo codice.

Definizione 10. *Si definisce* ***codice r-ario*** *un codice con r simboli con cui si ottengono le codeword.*
Un codice che parte da $\Gamma = \{0, 1\}$ è un codice binario.

Ad un certo punto Kraft scopre un teorema che contiene una disuguaglianza che fornisce una condizione necessaria e sufficiente affinché esista un codice istantaneo con certe lunghezze $l_1, l_2, \ldots, l_n$ date.

Teorema 1 (disuguaglianza di Kraft)**.** *Si ha che* ***esiste*** *un* ***codice istantaneo r-ario*** *per una sorgente S di q simboli, con codeword di lunghezza $l_1, l_2, \ldots, l_q$, sse:*

$$\sum_{i=1}^{q} \frac{1}{r^{l_i}} \leq 1$$

In onore di Kraft si ha che:

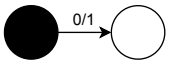
$$K = \sum_{i=1}^{q} \frac{1}{r^{l_i}} \text{ e quindi } K \leq 1$$

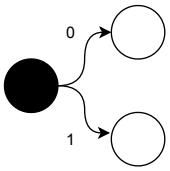
Non si sta vedendo come sono fatte le codeword ma solo se può esistere un certo codice istantaneo. Sapendo che esiste poi bisogna costruire le codeword ma il teorema non dice nulla in merito.
La condizione necessaria e sufficiente è quindi una caratterizzazione dell'esistenza del codice istantaneo.

Dimostrazione. Dobbiamo dimostrare i due versi del sse:

1. per ogni codice istantaneo r-ario di q simboli con lunghezze $l_1, l_2, \ldots, l_q$ si ha che $K \leq 1$
2. se $K \leq 1$ allora esiste un codice istantaneo con lunghezze $l_1, l_2, \ldots, l_q$

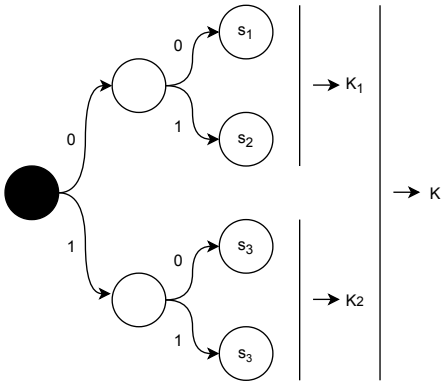
Dimostriamo la **prima parte**.
Prendo il codice e lo rappresento sull'albero di decodifica (dove ogni nodo ha al più r archi uscenti). Ricorsivamente un albero o è vuoto o è un nodo che punta ad altri nodi.
Parto con $r = 2$, avendo codeword binarie su $\{0, 1\}$ e di conseguenza alberi binari di decodifica. Procedo quindi con una dimostrazione per induzione sulla profondità dell'albero:

- **base**: profondità pari a 1. Si hanno due casi:
  1. la radice o punta ad una foglia. In questo caso:
  $$K = \frac{1}{2^1} = \frac{1}{2} \leq 1$$
  

  2. punta a due foglie (una per ogni simbolo). In questo caso:
  $$K = \frac{1}{2^1} + \frac{1}{2^1} = 1 \leq 1$$
  

- **passo induttivi**: assunto che il teorema è vero per ogni albero di profondità $n-1$ dimostriamo che il teorema vale anche per gli alberi di profondità n. In realtà così non ci va bene in quanto i sottoalberi dovrebbero avere entrambi profondità $n-1$ (cosa non vera come visto negli esempi precedenti, ad esempio "l'*albero a pettine*" del comma code). Cambiamo quindi dicendo che assunto

che il teorema è vero per ogni albero di profondità $< n$ dimostriamo che il teorema vale anche per gli alberi di profondità n.
Considero i due sottoalberi come alberi a se stanti:



Dico che $K_1 \leq 1$ per il primo sottoalbero e $K_2 \leq 1$ per il secondo sottoalbero.
Calcolo ora K di tutto l'albero ma devo capire come si rapporta a K_1 e K_2. Si ha che se attacco il primo sottoalbero alla nuova radice per costruire l'albero globale ho che attacco ad ogni codeword un simbolo, che quindi si allungano di 1:

$$\sum_{i=1}^{q} \frac{1}{r^{l_i+1}} \leq 1$$

ma altro non è che:

$$\sum_{i=1}^{q} \frac{1}{r} \cdot \frac{1}{r^{l_i}} \leq 1$$

ma allora:

$$\frac{1}{r} \cdot \sum_{i=1}^{q} \frac{1}{r^{l_i}} \leq 1 = \frac{1}{r} \cdot K_1$$

Faccio lo stesso per K_2 e quindi, avendo $r = 2$:

$$K = \frac{1}{2} \cdot K_1 + \frac{1}{2} \cdot K_2$$

ma sto sommando due quantità $\leq \frac{1}{2}$ (avendo $K_1 \leq 1$ e $K_2 \leq 1$) e quindi:

$$K \leq 1$$

Qualora si avesse $r > 2$ si ha:

- **base**, si hanno alberi r-ari, con $\Gamma = \{0, 1, 2, \ldots, r-1\}$, di probabilità 1. Tutti questi alberi hanno s foglie e $s - r$ non foglie. Quindi, essendo tutti gli l_i uguali a 1 essendo a profondità uno:

$$K = \sum \frac{1}{r^{l_1}} = \sum \frac{1}{r} = \frac{s}{r} \leq 1$$

  $\frac{s}{r} \leq 1$ in quanto $s < r$

- **passo induttivo** si ha un albero r-ario a profondità n con un certo numero di sottoalberi s e un certo numero di sottoalberi vuoti, di cardinalità $s - r$, a profondità $n - 1$. Per tutti i $K_1, K_2, \ldots K_s$ ho che $K_i \leq 1$ e quindi ottengo K aggiungendo un simbolo alle codeword dei sottoalberi non vuoto (quelli vuoti contribuiscono con 0 alla sommatoria):

$$K = \frac{1}{r}K_1 + \frac{1}{r}K_2 + \cdots + \frac{1}{r}K_s$$

  Ma per lo stesso ragionamento del caso binario, essendo ogni $K_1 < leq1$ e quindi ogni componente della sommatoria è $\leq \frac{1}{r}$:

$$K = \frac{1}{r}K_1 + \frac{1}{r}K_2 + \cdots + \frac{1}{r}K_s \leq \frac{s}{r} \leq 1$$

Vediamo la **seconda parte** della dimostrazione.
Se $K \leq 1$ dobbiamo dimostrare che esiste un codice istantaneo con lunghezze $L_1, \ldots, l_q$.
Sappiamo che:

$$K = \sum_{i=1}^{q} \frac{1}{r^{l_i}} \leq 1$$

Vediamo un esempio per capire come vedere diversamente la diseguaglianza:

Esempio 17. *se avessi $r = 2$, con quindi codeword binarie, e lunghezze 2,2,3,3,4 (con quindi $q = 5$). Si ha che:*

$$K = \frac{1}{2^2} + \frac{1}{2^2} + \frac{1}{2^3} + \frac{1}{2^3} + \frac{1}{2^4}$$

Ma quindi:

$$K = \frac{2}{2^2} + \frac{2}{2^3} + \frac{1}{2^4}$$

Avendo ora 2,2,1 come numeratori.

Indico con t_j il numero di codeword di lunghezza j. Si hanno quindi, per l'esempio precedente:

- $t_1 = 0$
- $t_2 = 2$
- $t_3 = 2$
- $t_4 = 1$

Riscrivo quindi K:

$$K = \sum_{i=1}^{q} \frac{1}{r^{l_i}} = \sum_{j=1}^{l} \frac{t_j}{r^j}$$

con $l = \max\{l_1, \ldots, l_q\}$ (per l'esempio sopra $l = 4$).
Si ha quindi che:

$$\sum_{j=1}^{l} \frac{t_j}{r^j} \leq 1$$

Ma posso dire che, per arrivare ad isolare i t_j senza alterare la diseguaglianza:

$$r^l \cdot \sum_{j=1}^{l} \frac{t_j}{r^j} \leq 1 \cdot r^l$$

ma quindi ho che:

$$r^l \cdot \sum_{j=1}^{l} t_j \cdot r^{l-j} \leq 1 \cdot r^l$$

Si ha che:

$$\sum_{j=1}^{l} t_j \cdot r^{l-j} = t_1 \cdot r^{l-1} + \cdots + t_{l-1} \cdot r + t_l \leq r^l$$

Ma quindi:

$$t_l \leq r^l - t_1 \cdot r^{l-1} - \cdots - t_{l-1} \cdot r$$

ma so anche che:

$$0 \leq t_l$$

e quindi:

$$0 \leq t_l \leq r^l - t_1 \cdot r^{l-1} - \cdots - t_{l-1} \cdot r$$

e quindi:

$$0 \leq r^l - t_1 \cdot r^{l-1} - \cdots - t_{l-1} \cdot r$$

porto quindi t_{l-1} a sinistra e divido per r, ottenendo:

$$t_{l-1} \leq r^{l-1} - t_1 \cdot r^{l-1} - \cdots t_{l-2} \cdot r$$

lo faccio per tutti i t_j partendo da t_{l-2} e arrivando a:

$$t_2 \leq r^2 - t_1 \cdot r$$

sapendo per di più che:

$$0 \leq t_2 \leq r^2 - t_1 \cdot r$$

e poi a:

$$0 \leq t_1 \leq r$$

Risalgo quindi partendo dalle codeword più corte. Cerco t_1 codeword di lunghezza 1, avendo a disposizione l'alfabeto Γ r-ario. Per farlo prendo i primi t_1 simboli e dico che sono le stringhe di lunghezza 1. Posso farlo perché $t_1 \leq r$ e mi avanzano $r - t_1$ simboli dell'alfabeto.
Passo alle codeword di lunghezza 2, me ne servono t_2. Queste codeword sono due simboli concatenati in modo che il primo simbolo non si stato utilizzato per t_1 (altrimenti creerei prefissi), quindi prendo un simbolo dai $r - t_1$ simboli dell'alfabeto. Il secondo simbolo posso mettere un qualsiasi simbolo di Γ, avendo r possibili scelte. In totale ho quindi un numero di combinazioni pari a:

$$(r - t_1) \cdot r = r^2 - t_1 \cdot r$$

e ho abbastanza combinazioni avendo che $t_2 \leq r^2 - t_1 \cdot r$ per i conti precedenti.
Le codeword le scelgo come voglio, ci sono tantissimi modi possibili.
Mi avanzano quindi $r^2 - t_1 \cdot r - t_2$.
Creo le codeword di lunghezza 3 dove i primi due simboli non devono essere una combinazione usata per quelle di lunghezza 2, facendo poi gli stessi discorsi fatti sopra.
Avanzo così per tutte le lunghezze di codeword avendo che posso costruire almeno un codice istantaneo, dimostrando la validità del teorema. □

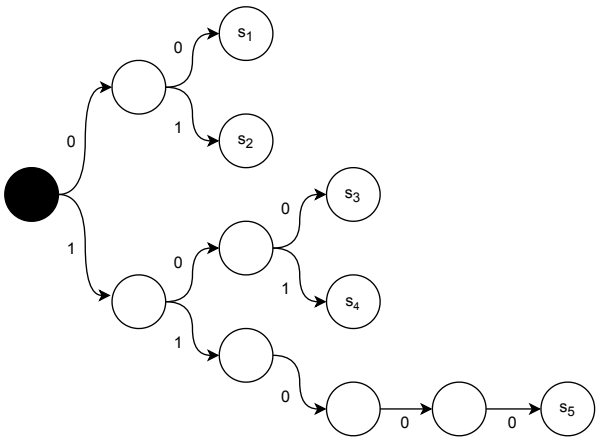
Esempio 18. *Riprendiamo l'esempio di sopra.*
Si ha che:

$$K = \frac{1}{2^2} + \frac{1}{2^2} + \frac{1}{2^3} + \frac{1}{2^3} + \frac{1}{2^4}$$

Ma quindi:

$$K = \frac{2}{2^2} + \frac{2}{2^3} + \frac{1}{2^4} = \frac{3}{4} + \frac{1}{16} \leq 1$$

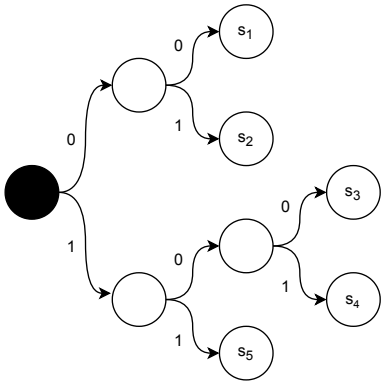
Se volessi le codeword costruisco l'albero di decodifica:



Avendo:

- $s_1 \to 00$
- $s_2 \to 01$
- $s_3 \to 100$
- $s_4 \to 101$
- $s_5 \to 1100$

Ma vedo che ho nodi interni inutili (tipo i due nodi che portano a s_5). Butto via quindi i nodi con un solo ramo in uscita:



ottenendo:

- $s_1 \to 00$

- $s_2 \to 01$
- $s_3 \to 100$
- $s_4 \to 101$
- $s_5 \to 11$

e quindi:

$$K = \frac{3}{2^2} + \frac{2}{2^3} = \frac{3}{4} + \frac{1}{4} = 1 \leq 1$$

Quindi un nodo interno che solo un figlio mi fa allungare le lunghezze delle codeword facendo sommare valori più piccoli. Se ho solo nodi interni con due figli si può dimostrare per induzione che $K = 1$ (nel caso base ho $K = 1$ e in quello induttivo vedo che ho $K_i = 1$ per ogni sottoalbero, avendo poi che $K = \frac{1}{2}K_1 + \frac{1}{2}K_2 = \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot 1 = 1$).