



UNIVERSITÀ DI PISA

UNIVERSITÀ DEGLI STUDI DI PISA



SCUOLA SUPERIORE SANT'ANNA

UNIVERSITÀ DEGLI STUDI DI PISA
Facoltà di Ingegneria

Corso di Laurea in

INGEGNERIA INFORMATICA

**Studio di un metodo efficiente
nella compressione dati**

Tesi di Laurea di

Luca Foschini

Relatori:

Prof. Roberto Grossi

Prof.ssa Nicoletta De Francesco

Indice

1	Introduzione al lavoro	2
1.1	Scopo del lavoro	2
2	Compressore a <i>compressed suffix array</i>	4
2.1	Trie, suffix tree, suffix array	4
2.2	Compressione dati	6
2.3	Teoria dell'informazione	6
2.4	Compressed Suffix Arrays	7
2.5	Burrows Wheeler transform	8
2.6	BWT e CSA	11
2.7	Wavelet tree	12
2.8	Indicizzare e comprimere	14
2.9	Semplificare bzip2	14
3	CSA: Ipotesi sull'entropia degli RLE generati	19
3.1	Analisi dell'autocorrelazione e dell'autocovarianza	20
4	Analisi delle prestazioni di vari codici prefissi	23
4.1	Codificare gli interi	23
4.2	Tentativi di codifica con codici prefissi	25
4.3	Comparazione con il codice di Huffman	26
5	Alternative al γ coding	30
5.1	Codice aritmetico	30
5.2	Implementazione	33
6	Tentativi di compressione con aritmetico	34
7	Analisi della distribuzione statistica degli RLE	36

<i>INDICE</i>	2
7.1 Metodo dei minimi quadrati	37
7.2 Analisi dei dati per determinare i parametri	39
7.3 Range Encoder	40
8 Dettagli implementativi	42
8.1 Decodificatore	45
8.2 Risultati	46
9 Versione adattiva	49
9.1 Maximum Likelihood Estimation	49
9.2 Codificatore adattivo con algoritmo ML	50
9.3 Dettagli implementativi	53
9.4 Decodificatore adattivo	55
10 Estensioni: codificatore RLE integrato	56
10.1 Risultati comparazione con γ coding	57
10.2 Comparazione con aritmetico standard	58
A Codice MATLAB	60
B Codificatore a range coder.	62

Capitolo 1

Introduzione al lavoro

I *suffix array* e i *suffix tree* sono strutture dati utilizzate nella maggior parte degli algoritmi che operano su stringhe come ad esempio *pattern matching*, *web searching*, *information retrieval* e sono molto sfruttati anche nel campo della biologia computazionale. In particolare i *suffix array* e i *suffix tree* sono alla base di una gestione efficiente delle operazioni fondamentali che si possono compiere su stringhe come ad esempio la ricerca di una parola o di una sottostringa. Con la necessità, emersa negli ultimi anni, di gestire stringhe di dimensioni sempre maggiori, sulle quali compiere operazioni di ricerca e indicizzazione (si pensi alla sequenza DNA di un batterio o, ad esempio, al *crawling* delle pagine *web* di un dominio come *mit.edu*) le strutture classiche dei *suffix array* e *suffix tree* si sono dimostrate essere inefficienti in termini di spazio in quanto occupano una dimensione varie volte superiore alla stessa stringa che devono gestire.

Proprio per questo motivo, negli ultimi anni la ricerca si è spostata nella direzione di tentare di diminuire lo spazio occupato da queste strutture dati preservando comunque la loro funzione, ovvero quello che si è cercato di fare è stato *comprimere* queste strutture. In questo senso quindi, il già maturo campo della compressione dati è venuto a incontrarsi con il relativamente giovane campo degli algoritmi di gestione e indicizzazione stringhe e sono così nati i concetti di *suffix array* compressi e gli indici FM [7].

1.1 Scopo del lavoro

Il seguente lavoro si propone di essere *in primis* un'analisi sia teorica che sperimentale dell'impatto che diversi tipi di codifiche possono avere nella compressione di *suffix array* come descritto da Grossi Gupta e Vitter negli articoli [12] e [13] e vuole essere sia un approfondimento che la continuazione di alcuni aspetti trattati in quei lavori.

Negli articoli sopra citati viene proposto un semplice metodo di costruzione di un *full-index* di un testo. Questo *full-index*, oltre a permettere operazioni di ricerca sul testo senza decomprimerlo completamente, risulta anche essere di dimensioni paragonabili a quelle che avrebbe il testo compresso con alcuni metodi commerciali noti (come ad esempio `bzip2`). In altre parole, esso equivale in tutto e per tutto a una versione compressa del testo, sulla quale però si possono compiere le operazioni di ricerca prima descritte come se il testo fosse in formato non compresso.

In particolare, come verrà spiegato in dettaglio di seguito, il metodo proposto in [12] e [13] consiste nell'applicare una semplice trasformata al testo e poi codificare con γ *coding* il risultato di questa trasformata. La scelta del γ *coding* è però stata guidata da un mero confronto empirico di performance con altri codici, non è chiaro come γ *coding*, di per sé così semplice, possa ottenere prestazioni così elevate in termini di rapporto di compressione. Nel seguente lavoro si cercherà di dare una giustificazione sperimentale e teorica alle alte performance riscontrate nell'utilizzo del γ *coding* e verrà proposto un metodo alternativo di codifica basato sull'utilizzo dell'algoritmo *range-coder*, una variante del codice aritmetico. Si vedrà come il suddetto metodo, utilizzando un modello statistico molto sofisticato costruito *ad hoc*, può raggiungere risultati migliori del γ *coding* e ottenere performance di compressioni maggiori anche dei migliori compressor commerciali (`gzip`, `bzip2`) mantenendo comunque una complessità asintotica temporale e spaziale minore di quella che avrebbe un codificatore aritmetico adattivo, che dimostra avere anche minori performance in termini di rapporto di compressione raggiunto.

Capitolo 2

Compressore a *compressed suffix array*

Secondo quanto detto nell'introduzione, *suffix array* e *suffix tree* rappresentano il cuore di molti algoritmi di ricerca e indicizzazione di stringhe, e sono la base del *compressed suffix array* descritto in [12] [13]. Passiamo quindi ad analizzare in dettaglio queste strutture dati.

2.1 Trie, suffix tree, suffix array

Il *suffix tree* [32] e il *suffix array*, come si è detto, sono strutture dati molto utili per risolvere problemi di ricerca su stringhe in modo elegante ed efficiente.

Nella sua realizzazione più semplice, un *suffix tree* di una stringa S è semplicemente un *trie* [8],[9],[1] delle n stringhe che costituiscono i suffissi di S formata da n caratteri.

Un *trie* (vedi Figura 2.1) è una struttura ad albero dove ogni nodo rappresenta un carattere e la radice rappresenta la stringa nulla, quindi ogni cammino dalla radice a un altro nodo rappresenta una stringa, descritta dai caratteri che etichettano i nodi attraversati, presi nell'ordine di attraversamento. Ogni insieme finito di parole definisce una *trie* e due parole con lo stesso prefisso portano a una diramazione del *trie* che le contiene in corrispondenza del primo carattere per cui le due parole differiscono.

È molto semplice vedere se una stringa q appartiene a un certo insieme di stringhe se le stringhe di quest'insieme sono organizzate in un *trie*. Partendo dal primo carattere di q si attraversa il *trie* lungo il ramo definito dal prossimo carattere di q . Se il ramo non esiste, q non si trova nel *trie*, altrimenti troviamo q compiendo solo $|q|$ ¹ confronti, indipendentemente dal numero totale di caratteri (nodi) che compaiono nel *trie*.

Un *suffix tree* di una stringa S può essere definito come un *trie* costruito su tutti i suffissi propri di S . I *suffix tree* danno la possibilità di testare velocemente se una data stringa q

¹Con la notazione $|x|$ si indica la lunghezza (ovvero il numero di caratteri) della stringa x

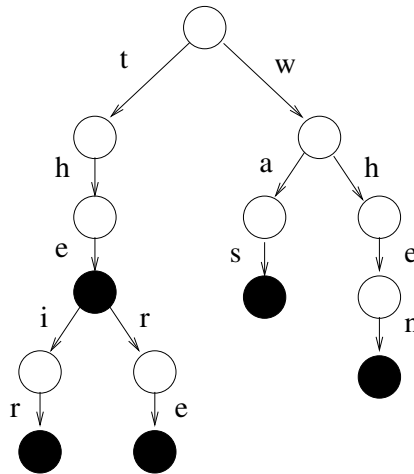


Figura 2.1: Un trie costruito sulle stringhe *the*, *their*, *then*, *was*, *when*. I nodi anneriti indicano la fine di una stringa.

è sottostringa di S poichè ogni sottostringa di S è prefisso di qualche suffisso. Il tempo di ricerca è di nuovo lineare nel numero di caratteri che costituiscono q .

Sfruttando alcuni sofisticati algoritmi un *suffix tree* per una stringa S di n caratteri può essere costruito in tempo $O(n)$ e spazio $O(n)$ [34],[20],[33].

I *suffix array* possono compiere le stesse operazioni compiute dai *suffix tree* utilizzando in media quattro volte memoria in meno rispetto ai *suffix tree*. Un *suffix array* è in più anche facile da implementare, in quanto è semplicemente un vettore che contiene i puntatori a tutti gli n suffissi di una stringa S ordinati secondo qualche ordine (di solito alfabetico). In questo modo, una ricerca binaria può verificare se una stringa q è sottostringa di S con $O(\log(n))$ comparazioni.

Un *suffix array* occupa meno spazio del corrispondente *suffix tree* perchè in esso vengono memorizzati i puntatori alla stringa originale invece che delle copie effettive delle sottostringhe.

I *suffix array* raggiungono migliori performance di ricerca rispetto ai *suffix tree* perchè in generale occupano meno memoria e grazie alla loro semplicità possono essere implementati in modo molto ottimizzato. Anche i *suffix array* possono essere costruiti in tempo $O(n)$ dove n è il numero di caratteri della stringa sulla quale vengono costruiti.

Un'estensiva trattazione dei possibili utilizzi e applicazioni di *suffix tree* e *suffix array* può essere trovata nei lavori di Gusfield [14] oppure Crochemore e Rytter [4].

2.2 Compressione dati

Una volta definite le strutture dati su cui si basano i *compressed suffix array* è necessario introdurre alcuni concetti propri della teoria della compressione dati, concetti cruciali per capire in quale modo possa essere ridotto lo spazio occupato dai *suffix tree* e *suffix array*.

2.3 Teoria dell'informazione

Per comprendere come sia possibile comprimere un messaggio (nel nostro caso un *suffix array*) è necessario capire come sia possibile diminuire lo spazio (le risorse) occupate da quel messaggio lasciando immutato il suo contenuto informativo. Risulta quindi fondamentale riuscire in qualche modo a stimare il contenuto informativo di un messaggio, e per fare ciò utilizziamo il concetto di entropia.

L'entropia di una sorgente ² è una quantità che dipende solo dalla natura statistica della sorgente stessa e ne misura il suo contenuto informativo.

Intuitivamente, più un messaggio è inaspettato (leggi "improbabile") più il suo contenuto informativo (l'informazione che ci dà) è alto, viceversa, una messaggio con probabilità di apparire pari a 1 è banale e non aggiunge informazione alla nostra conoscenza.

A seconda di come la probabilità di un simbolo (il nostro messaggio) viene messa in correlazione con quella relativa ai simboli precedenti (che formano quello che si chiama il *contesto* del simbolo) si definiscono vari tipi di entropia. Detto α l'alfabeto considerato, formato da m simboli e detta p_i la probabilità di occorrere dell' i -esimo simbolo si definiscono vari tipi di entropia.

1. Entropia di ordine zero. I simboli sono statisticamente indipendenti.

$$H_0 = - \sum_{i=1}^m p_i \log p_i$$

Considerando questa definizione di entropia, la lingua inglese avrebbe in media H_0 pari a 4.07 bit/symbol. Effettivamente, codificando un testo in lingua inglese con un codificatore di entropia di ordine 0 (ad esempio codice di Huffman) si ottengono risultati simili. (Il che significa che la dimensione del testo viene ridotta di quasi il 50%, dato che la codifica ASCII impiega 8 bit per simbolo).

²Possiamo intuitivamente intendere una sorgente come una sequenza molto lunga di simboli, eventualmente infinita

È da notare che una compressione di questo tipo è ottenuta solo considerando la frequenza dei singoli caratteri (simboli) e non le varie interdipendenze tra i caratteri, che sono molte e molto marcate. Ad esempio, supponiamo che nella lingua inglese la lettera “H” abbia una frequenza del 5%, il che significa che ogni simbolo che si viene a codificare avrà 5 possibilità su 100 di essere una “H”. Se però si conosce l’informazione che è appena stato codificato “T”, allora la probabilità che il carattere che sarà codificato di seguito sia “H” aumenta di molto.

Per sfruttare l’informazione relativa alle relazioni di interdipendenza tra i simboli sono state introdotte le definizioni di entropia di ordine superiore.

2. Entropia di ordine uno. Definiamo $p_{j|i}$ la probabilità condizionale che il carattere presente sia il j -esimo dell’alfabeto noto che il precedente era il carattere i -esimo, avremo che:

$$H_1 = - \sum_{i=1}^m p_i \sum_{j=1}^m p_{j|i} \log p_{j|i}$$

3. Generalizzando, possiamo introdurre il concetto di entropia di ordine h . Detta $B(h)$ una generica sequenza di h simboli, definiamo entropia di ordine h

$$H_h = - \lim_{h \rightarrow \infty} \frac{1}{h} \sum p_{B(h)} \log p_{B(h)}$$

Dove la somma è su tutti le possibili m^n sequenze che costituiscono $B(n)$.

L’entropia di ordine h è solo un concetto astratto, non direttamente misurabile, solo intuibile o esperibile, per questo è anche chiamata entropia empirica. Shannon per esempio ha stimato che l’entropia di ordine h del testo inglese contenente solo le 27 lettere si aggira sui 2.3 bit per simbolo. L’entropia di ordine h ci dà anche un limite superiore alla comprimibilità di un messaggio. Non è infatti possibile codificare un messaggio costituito da n simboli utilizzando meno di nH_h bit.

2.4 Compressed Suffix Arrays

Ora che sono stati introdotti i concetti fondamentali di *suffix array* e i concetti di base della compressione dati, possiamo analizzare in dettaglio i *compressed suffix array* (da ora in poi *CSA*) che sfruttano entrambi i concetti.

Abbiamo visto nella sezione precedente che i *suffix tree* e i *suffix array* rappresentano un ottimo modo di compiere operazioni di ricerca in un testo ma hanno lo svantaggio di occupare molto spazio, in particolare, un *suffix tree* richiede $4n \log n$ bit di spazio (16 volte la dimensione del testo iniziale) mentre i *suffix array* necessitano di mantenere in memoria il testo originale.

Un *CSA* contiene la stessa informazione contenuta in un *suffix array* standard ma occupa molto meno spazio rispetto ad esso. L'idea chiave che viene sfruttata dai *CSA* sta nel concetto di *self-index*. Se l'indice del testo contiene tutta l'informazione per compiere ricerche sul testo che indicizza e per recuperare porzioni di esso senza dovere accedere al testo, non vi è più alcun bisogno di tenere il testo stesso in memoria, il che si traduce in un enorme risparmio di spazio.

I *self-index* possono quindi sostituire completamente il testo come avviene nella compressione. Ma, rispetto alla versione compressa di un testo che potremmo ottenere con qualche compressore commerciale, i *self-index* mantengono comunque la possibilità di compiere sul testo le operazioni di ricerca e recupero prima descritte.

Grossi Gupta e Vitter [13] hanno recentemente proposto un *CSA* che sfrutta l'entropia di ordine h di un testo per rappresentare un *CSA* in $nH_h + O(n \log \log n / \log_{|\Sigma|} n)$ ³ bit. Il *CSA* proposto occupa quindi il minimo spazio possibile (a meno di infinitesimi di ordine inferiore) in quanto raggiunge asintoticamente l'entropia empirica del testo, limite inferiore come noto non superabile, con costante moltiplicativa uguale a 1.

Il *self-index* descritto è così potente che è in grado di contenere il testo implicitamente codificato in esso. Una ricerca sul testo richiederebbe di decomprimere una parte trascurabile di testo, inoltre, nelle implementazioni pratiche questo genere di *CSA* occupa circa dal 25% al 40% della dimensione originale del testo. Se il testo è altamente comprimibile in modo che $H_h = o(1)$ e la dimensione dell'alfabeto è piccola è dimostrato [13] che un'operazione di ricerca può essere compiuta in $o(m)$ operazioni e decomprimendo solo $o(n)$ bit. Una trattazione completa del *CSA* può essere trovata in [12],[26] e [13]. Di seguito diamo una breve descrizione equivalente del *CSA* in termini di trasformata di Burrows Wheeler (*BWT*).

2.5 Burrows Wheeler transform

La trasformata di *Burrows e Wheeler* [3] (da ora in avanti *BWT*) sin da quando fu introdotta nel 1994, è stata utilizzata in innumerevoli applicazioni, molte delle quali riguardanti

³ n è la lunghezza del testo in bit e $|\Sigma|$ è la dimensione dell'alfabeto

proprio la compressione dati. Attualmente viene utilizzata come primo stadio di trasformazione della stringa da comprimere in molti programmi di compressione commerciali, primo tra tutti `bzip2`.

La BWT opera su un blocco di bytes di una certa lunghezza, per questo viene anche chiamata *block sorting*.

Data una stringa S di n caratteri questa viene trasformata in un'altra stringa L (L è una permutazione di S) dello stesso numero n di simboli che soddisfi le seguenti ipotesi:

- Ogni regione di L tende a soddisfare il principio di località, ovvero, se un simbolo s si trova in L , allora è probabile che altri simboli uguali ad s si trovino nelle sue vicinanze.
- È possibile ricostruire la stringa S avendo a disposizione solo L .

Il metodo di *Burrows Wheeler* per comprimere una stringa S funziona come segue:

1. Viene aggiunto alla fine della stringa S un carattere, detto marcatore, che non compare all'interno della stringa stessa che ha lo scopo di segnalare la fine della stringa stessa.
2. Viene creata la stringa L permutando in un certo modo i caratteri di S . (Il metodo con il quale questa permutazione viene compiuta verrà spiegato successivamente.)
3. L'encoder comprime L e restituisce in *output* il risultato della compressione. Di norma questa compressione avviene prima utilizzando l'algoritmo *Move to Front* (da ora in avanti *MTF*) che verrà trattato in seguito, seguito dalla codifica *Run Length Encoding*⁴ (da ora in avanti *RLE*) e dal codice di Huffman.
4. Il decodificatore legge lo *stream* di *input* e ricostruisce S applicando gli stessi metodi descritti al punto 3 in ordine inverso.
5. La stringa S viene ricostruita a partire da L tramite una sorta di BWT inversa.

⁴Run Length Encoding è un tipo di codifica che trasforma una lista di interi in una lista di coppie di interi in cui il primo elemento di ogni coppia indica quale elemento compare nella lista e il secondo indica quante volte è ripetuto di seguito. Ad esempio la lista $\{1,1,1,3,3,4,4,4,1,1\}$ viene codificata in RLE come $\{\{1,3\},\{3,2\},\{4,3\},\{1,2\}\}$. L'RLE di una lista che contiene solo 2 simboli si riduce a una lista di interi che indicano le occorrenze di simboli uguali in fila (non vi è bisogno di specificare quale sia il simbolo una volta fissato qual è il primo)

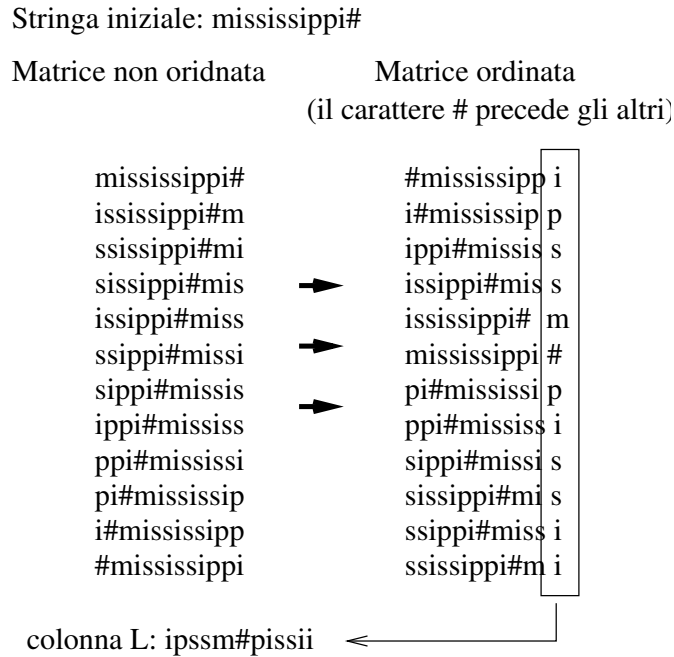


Figura 2.2: Esempio di BWT applicata alla stringa mississippi#, la cui BWT è ipssm#pissii.

L'idea centrale dell'algoritmo è costituita dal metodo con il quale viene costruita la stringa L a partire da S , che richiama molto da vicino alcuni concetti già appresi nelle sezioni precedenti relativa ai *suffix array*

Data una stringa S di n caratteri a cui è già stato aggiunto il carattere di determinazione alla fine, L viene costruita come segue:

1. Viene creata una matrice $n \times n$ di caratteri in cui ogni riga i è costituita dalla stringa S *shiftata* ciclicamente (ruotata) verso sinistra di $i - 1$ caratteri. La prima riga contiene quindi la stringa stessa, la seconda riga la stringa ruotata a destra di un carattere e così via.
2. La matrice viene ordinata in ordine lessicografico per righe.
3. La stringa L è costituita dall'ultima colonna della matrice ordinata.

Un esempio di come operi l'algoritmo BWT è riportato in Figura 2.2

È da notare che nelle implementazioni pratiche di questo algoritmo non è necessario che venga costruita esplicitamente la matrice prima descritta.

È facile da dimostrare perchè L mostra la proprietà di località precedentemente descritta.

Se per esempio le parole *nail*, *bail*, *fail*, *hail*, *jail*, *tail*, *rail*, *mail* comparissero da qualche parte in S , dopo l'ordinamento, tutte le permutazioni che iniziano con *il* comparirebbero insieme e quindi la colonna L corrispondente avrebbe un alto numero di a uno si seguito all'altra.

Possiamo quindi caratterizzare il metodo di *Burrows Wheeler* dicendo che raggruppa insieme simboli simili basandosi sui loro contesti.

La ricostruzione della stringa S a partire da L avviene come segue:

1. Viene calcolata la stringa F ordinando lessicograficamente L . Da notare che la stringa F costuirebbe la prima colonna della matrice ordinata precedentemente utilizzata nella fase di codifica.
2. Viene costruito l'array T in modo che $F[T[i]] = L[i]$ per $i = 0, 1, \dots, n$
3. Considerando nuovamente la matrice ordinata descritta nella fase di codifica si nota che in ogni riga i il simbolo $L[i]$ precede il simbolo $F[i]$ nella stringa originale S (il termine "precede" deve essere considerato in senso ciclico) Detta I la posizione del carattere marcatore di fine stringa in L , sulla riga I , $L[i]$ precede ciclicamente $F[i]$ ma $F[i]$ è il primo simbolo di S e $L[i]$ è l'ultimo. La ricostruzione comincia quindi con $L[i]$ e ricostruisce S da destra a sinistra.
4. $L[i]$ precede $F[i]$ in S per $i = 0, 1, \dots, n - 1$ Quindi, $L[T[i]]$ precede $F[T[i]]$, ma $F[T[i]] = L[i]$ da ciò si conclude quindi che $L[T[i]]$ precede $L[i]$ in S .
5. la ricostruzione quindi comincia da $L[I]$ (l'ultimo simbolo di S) e procede con $L[T[I]]$ fino ad arrivare al primo simbolo di S .

Tutto questo può essere riassunto dalla ricorrenza:

$I \leftarrow$ posizione del marcatore di fine stringa in L .

$S[n - 1 - i] \leftarrow L[T^i[I]]$, per $i = 0, 1, \dots, n - 1$,

dove $T^0[j] = j$ e $T^{i+1}[j] = T[T^i[j]]$.

2.6 BWT e CSA

Il contenuto del **CSA** descritto nelle sezioni precedenti può essere ora compreso utilizzando la nozione di **BWT** appena appresa. Ma un **CSA** è qualcosa di più della semplice **BWT**, come

detto prima, infatti, esso deve offrire la possibilità di compiere le operazioni di ricerca e estrazione prima discusse.

Un modo semplice per organizzare l'informazione contenuta nella BWT in modo che sia possibile effettuare operazioni di ricerca di sottostringhe e recupero di porzioni di testo è quella di creare delle liste separate per ogni contesto (per ogni simbolo dell'alfabeto) e organizzare queste liste in un *wavelet tree* come vedremo in dettaglio nella successiva sezione.

2.7 Wavelet tree

Il metodo più immediato per rappresentare i simboli sarebbe quello di tenere $|\Sigma|$ *bitvector* bv ove bv_i contiene uno 1 in posizione j se e solo se il j -esimo simbolo del testo coincide con l' i -esimo simbolo dell'alfabeto.

Segue un esempio:

```
stringa: mississippi#
Trasformata BWT: ipssm#pissii
5 bitvector (5 simboli)
      ipssm#pissii
bv per i: 100000010011
bv per m: 000010000000
bv per p: 010000100000
bv per s: 001100001100
bv per #: 000001000000
```

Concatenando i dizionari *bitvector* e conoscendo il numero di simboli, sarebbe quindi possibile creare uno *stream* di bit che rappresenta il testo e che possa essere facilmente decodificabile. A quanto detto sopra va aggiunto un limitato numero di bit deputati a contenere l'informazione necessaria ad accedere in modo selettivo a porzioni di dati distinti, in modo da potere effettivamente compiere le operazioni di ricerca e recupero delle informazioni immagazzinate.

Come si può chiaramente vedere, però, uno *stream* costruito giustappoendo semplicemente i *bitvector* sopra descritti (e aggiungendo l'informazione trascurabile necessaria ad accedere ai dati) sarebbe molto ridondante.

Grossi, Gupta, and Vitter [12] introducono il *wavelet tree* proprio per ridurre la ridondanza derivata dal mantenere dizionari separati per ogni simbolo che appare nel testo. Per fare

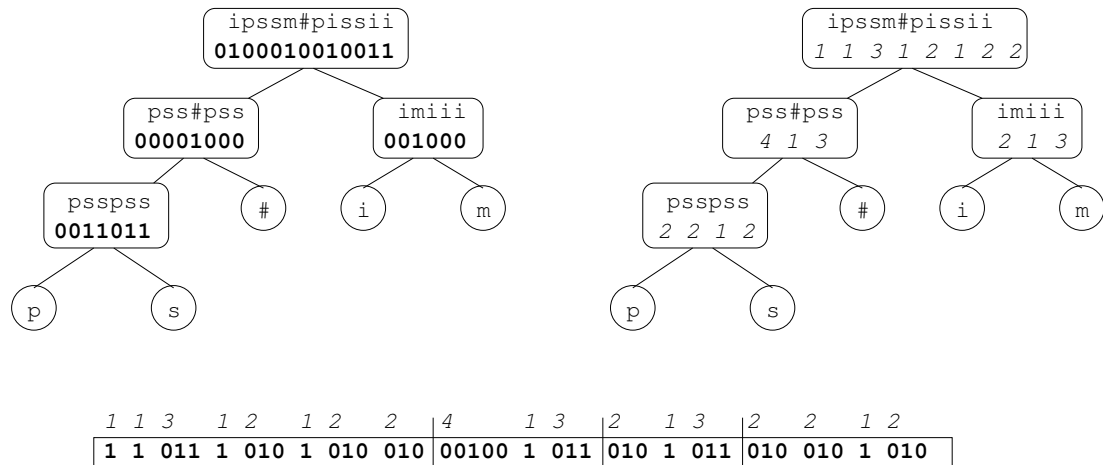


Figura 2.3: Wavelet tree costruito sulla stringa `mississippi#`, la cui trasformata è `ipssm#pissii`.

ciò, ogni successivo dizionario codifica solamente quelle posizioni che non sono già state prese in conto dai dizionari precedenti.

Codificando i dizionari in questo modo si riesce a raggiungere l'entropia di ordine h del testo come dimostrato in in Lemma 4.1 di [12]. Consideriamo il *wavelet tree* di esempio in Figura 2.3, costruito sulla stringa di testo `mississippi#`.

Supponiamo ogni ramo sinistro come associato con uno **0** e ogni ramo destro associato con un **1**. Ogni nodo interno u è un dizionario $\text{dict}[u]$ con gli elementi contenuti nel suo sottoalbero sinistro immagazzinati come **0** e quelli contenuti nel suo albero destro contenuti come **1**. Per esempio, consideriamo il nodo interno più a sinistra, le cui foglie sono `p` e `s`, i dizionari (trascurando i *leading 0*, di cui parleremo) indicano che un solo `p` appare nella stringa BWT, seguita da due `s` e così via.

Il secondo albero indica la codifica RLE dei dizionari e il *bitvector* più inferiore indica il vero *bitvector* immagazzinato su disco creato visitando per livelli ⁵ il *wavelet tree* e codificando con γ *coding* gli RLE precedentemente descritti. Il *leading 0* in ogni nodo è necessario in quanto crea un'associazione univoca tra i valori RLE e il *bitvector* ⁶

È stato dimostrato teoricamente [12] e sperimentalmente [13] che lo spazio occupato dal *wavelet tree* non dipende dalla forma dell'albero. Nell'esempio precedente abbiamo considerato il *wavelet tree* come un albero binario bilanciato semicompleto, scandito poi in

⁵Una visita per livelli di un albero viene effettuata visitando prima la radice, poi i nodi di primo livello da sinistra verso destra, poi quelli di secondo livello nello stesso ordine e così via.

⁶Per decodificare il *wavelet tree* è necessario sapere se il primo run sia di **0** o di **1** il *leading 0* aggiunge in testa ad ogni dizionario un run di zero di lunghezza 1 in modo da sciogliere l'ambiguità

modalità *heap*⁷, ma qualsiasi altra forma (non troppo vicina alla forma degenerare di una lista, però) avrebbe portato ad avere uno spazio occupato su disco simile. Gli esperimenti successivi, infatti, verteranno sui dati derivati da un *wavelet tree* avente la forma dell'albero di Huffman (v. sez. 2.9) costruito per il testo. Questa forma particolare, come detto, non cambia sostanzialmente le dimensioni dell'*output* finale anche risulta essere più efficiente in termini di tempo di elaborazione per i motivi esposti in [12].

2.8 Indicizzare e comprimere

Riassumendo quanto detto, aggiungendo una quantità di informazione trascurabile alla codifica del testo descritta sopra si ottiene quindi un *full-index* del testo che però ha dimensioni confrontabili con quelle di un testo compresso con un compressore commerciale. In particolare è dimostrato in [12] che questo tipo di compressione raggiunge l'entropia di ordine h del testo e, come detto precedentemente, conserva le proprietà di *indexing* prima esposte.

2.9 Semplificare bzip2

Riprendendo quanto detto nella sezione 2.5, la BWT è già stata utilizzata come base per un noto algoritmo di compressione, il *bzip2* [40]. In particolare, di seguito analizziamo come viene processato l'*output* della trasformata *bzip2*

bzip2

1. MTF

è una trasformata che non comprime il testo ma può essere utile per ridurre la ridondanza in molti casi, in particolare quando un simbolo che è stato visto precedentemente ha alta probabilità di comparire di nuovo.

Sia a_1, a_2, \dots, a_n un insieme di n simboli su un alfabeto Σ ; MTF, invece di restituire in *output* l'indice i del simbolo a_i che ne costituirebbe un identificatore univoco, restituisce un codice che indica la posizione del simbolo in una lista contenente tutti i simboli. Se la lista fosse fissata e immutabile, l'*output* dell'MTF costituirebbe solo una permutazione dell'*output* di un codificatore che restituisce solo l'indice dei simboli.

⁷ Quindi, la radice occupa la posizione 1, e il nodo in posizione i ha il suo padre in posizione $\lfloor i/2 \rfloor$ (se $i > 1$) e i suoi figli (se esistono) in posizione $2i$ e $2i + 1$, rispettivamente.

La lista di MTF , però, viene aggiornato dopo l'*output* di ogni simbolo, ogni volta che un simbolo viene processato, infatti, l'encoder compie le seguenti operazioni:

- Restituisce in *output* il codice del simbolo nella lista
- Aggiorna la posizione del simbolo nella lista, riportandolo in prima posizione (Da qui il nome *Move to Front*) e ricompatta l'array.

Il decodificatore MTF può ricostruire la sequenza di simboli se, ad ogni simbolo, compie le operazioni dell'encoder nello stesso ordine, e se le liste di encoder e decoder sono inizializzate allo stesso modo all'inizio della codifica.

Se la stringa sulla quale viene applicato l'algoritmo MTF ha la proprietà di località prima discussa allora i valori restituiti in *output* da MTF saranno in genere molto bassi. La dimostrazione di questo concetto è semplice, infatti, se in una stringa caratteri simili compaiono raggruppati, allora, una volta che questi verranno processati con MTF , verranno identificati da numeri d'ordine molto bassi all'interno della lista in quanto sono comparsi da poco tempo.

L'esempio sottostante chiarisce le idee. Supponiamo di avere la stringa “*dddbccdd-baa*”, MTF supponendo di iniziare il processo con una lista iniziale del tipo “a,b,c,d” procederà in questo modo:

Codifica:

Simbolo	Codice MTF	Lista MTF
d	3	a b c d
d	0	d a b c
d	0	d a b c
b	2	d a b c
c	3	b d a c
d	2	c b d a
d	0	d c b a
b	2	d c b a
a	3	b d c a
a	0	a b d c

Decodifica:

Simbolo	Codice MTF	Lista MTF
3	d	a b c d
0	d	d a b c
0	d	d a b c
2	b	d a b c
3	c	b d a c
2	d	c b d a
0	d	d c b a
2	b	d c b a
3	a	b d c a
0	a	a b d c

Se la stringa “*dddbcddbaa*” venisse codificata associando semplicemente ad ogni carattere il proprio numero d’ordine in una lista del tipo “a,b,c,d” allora la codifica sarebbe *3331233100*, mentre con MTF la codifica diventa *3002320230* che può essere codificata in modo più ottimale da un codificatore di entropia di ordine $\mathbf{0}$. I run di caratteri uguali vengono codificati in sequenze di $\mathbf{0}$, ad esempio, la stringa “*aaaaaabbbbbbaaaaaa*” verrebbe trasformata in “*000000100000100000*” che verrebbe codificata molto efficientemente da un codificatore di entropia di ordine zero che assegnerebbe meno bit a $\mathbf{0}$ poiché appare con probabilità molto maggiore rispetto a $\mathbf{1}$ in confronto con le probabilità relative con cui comparivano **a** e **b** nella stringa iniziale.

2. Huffman adattivo

Si definisce codice di Huffman [15] di una sorgente T il codice costruito come segue:

- Per ogni simbolo a_i viene calcolata la probabilità con cui il simbolo occorre nel testo in $p[i]$. Viene anche creato un vettore $cw[i]$ che contiene la *codeword* relativa ad ogni simbolo i . Inizialmente ogni *codeword* è inizializzata alla stringa nulla ϵ ed è associata a un valore distinto di $p[i]$, in particolare all’inizio $p[i]$ è associata a $cw[i]$ per ogni i .
- Si considerano i due valori $p[i]$ e $p[j]$ più piccoli di p e si aggiunge in testa alle *codeword* associate a $p[i]$ uno $\mathbf{0}$ e a quelle associate a $p[j]$ un $\mathbf{1}$.
- Si cancellano dal vettore p le posizioni i e j e si aggiunge ad esso un valore pari alla somma di $p[i]$ e $p[j]$, ricordando che a questo valore sono ora associate sia tutte le *codeword* che erano prima associate a $p[i]$ sia quelle prima associate a $p[j]$.

- (d) Si iterano i passi **2b** e **2c** fino a che p non contiene un solo elemento, che avrà valore 1 in quanto rappresenta la somma delle probabilità di occorrenza di ogni simbolo nel testo.

Esistono vari metodi per costruire efficientemente i codici di Huffman di un dato testo. In particolare, una gestione oculata della struttura dati che supporta l'array delle probabilità p (di solito si usa uno *heap*) permette di selezionare ogni volta il minimo dell'array con complessità logaritmica nel numero di simboli piuttosto che lineare (quale sarebbe la complessità di una scansione completa di p alla ricerca del minimo)

È da notare che una volta costruito il codice di Huffman questo deve essere anche scritto sullo *stream* di *output* in modo che il decodificatore possa ricostruire l'associazione tra simboli e *codeword* e quindi compiere la decodifica dello *stream*.

Il codice di Huffman è ottimo tra i codici istantanei⁸ come dimostrato in [15], ovvero, non è possibile codificare un testo **assegnando una *codeword* fissata** ad ogni simbolo in uno spazio minore di quello che si ottiene utilizzando i codici di Huffman.

Proprio in quest'ultima affermazione sta la potenza e il limite dei codici di Huffman.

In quanto codice prefisso, la codifica/decodifica del codice di Huffman può essere ottenuta molto velocemente (ad ogni simbolo è assegnato una sequenza di bit nota a priori). Il fatto che però ad ogni simbolo venga assegnata una sequenza di lunghezza intera di bit può dimostrarsi una pratica molto inefficiente.

Supponiamo infatti di avere una sorgente che emetta 2 soli simboli: **a** con $p = 0.99$ e **b** con $p = 0.01$. In questo caso l'algoritmo di Huffman assegnerebbe una *codeword* lunga un bit ad entrambi i simboli, il che è evidentemente inefficiente, in quanto significherebbe che entrambi i simboli avrebbero la stessa occupazione di spazio anche se **1** occorre molto meno spesso di **0**.

Un modo per superare questa limitazione è quella di assegnare un numero non intero di bit ai vari simboli. Questo è l'approccio utilizzato, ad esempio, dai codificatori aritmetici, che tratteremo in seguito.

Un altro grosso problema dei codici di Huffman è che non sono facili da rendere adattivi. In altre parole, l'associazione tra simboli e *codeword* viene fatta una volta per tutte e in base alla probabilità calcolate su tutto il testo che deve essere codificato,

⁸Un codice si dice istantaneo quando è possibile scomporre ogni messaggio codificato con quel codice nelle *codeword* costituenti sequenzialmente, ovvero senza dovere compiere operazioni di *lookahead*

ma all'interno del testo le distribuzioni di probabilità potrebbero cambiare molto e in questo caso il codice di Huffman si troverebbe a codificare il testo utilizzando un modello statistico non appropriato e quindi non efficiente.

Per evitare quindi che l'algoritmo di Huffman codifichi un testo utilizzando un modello probabilistico non più rispondente alla reale distribuzione dei simboli è necessario che dopo un numero predeterminato di simboli letti (conosciuto sia dal codificatore che dal decodificatore) l'algoritmo ricalcoli le *codeword* da assegnare ai simboli in base alle probabilità misurate sui simboli letti. Quando il codice di Huffman utilizza questo approccio di ricalcolo continuo del modello statistico viene definito *adattivo*. Un approccio adattivo di questo tipo può aumentare sensibilmente il dispendio di risorse di tempo e memoria impiegate nella codifica/decodifica.

Abbiamo quindi visto che il *post-processing* compiuto da `bzip2` sull'*output* della BWT è molto complesso e molto dispendioso in termini di risorse e di tempo.

È stato viceversa fatto notare che l'organizzazione dell'*output* della BWT in *wavelet tree* e la codifica dei *bitvector* generati tramite RLE costituisce un'alternativa semplice e veloce e di pari performance. Il punto cruciale rimane quello di determinare quale possa essere una codifica veloce ed efficiente per gli RLE generati dalla codifica del *wavelet tree*.

Nel lavoro [13], come già accennato prima, è stato proposto come codice da utilizzare il semplicissimo γ *coding* che comunque, a dispetto della sua semplicità, è risultato essere molto efficace in termini di rapporto di compressione raggiunto.

Scopo di questo lavoro di tesi è quello di continuare l'analisi comparativa tra i vari codici prefissi già incominciata nei lavori [12] e [13] ed estendere il paragone anche a codici di altro genere, al fine di migliorare le prestazioni raggiunte da γ *coding*.

Verrà dimostrato empiricamente utilizzando mezzi statistici e di simulazione numerica che il γ *coding* è quasi ottimale per la distribuzione degli interi nelle sequenze RLE presi in esame e che per raggiungere un eventuale miglioramento sarebbe necessario complicare notevolmente il codificatore e quindi aumentare il dispendio di risorse di tempo e spazio.

Utilizzando concetti presi in prestito dalla teoria dei segnali e dall'analisi delle serie storiche, di solito non considerati nella teoria della compressione dati *tout-court* e riutilizzando *software* libero basato su lavori di ricerca precedenti verrà poi proposto un codificatore alternativo a γ *coding* che ne migliora sensibilmente le prestazioni arrivando a superare talvolta quelle raggiunte da `bzip2`, a discapito però di una complicazione ingente del codificatore stesso.

Capitolo 3

CSA: Ipotesi sull'entropia degli RLE generati

Nel tentativo di dare una giustificazione teorica alla buona performance ottenuta da γ coding nel codificare gli RLE prodotti dalla scansione del *wavelet tree* utilizzato per riorganizzare la BWT di un testo si è proceduto *in primis* a un'analisi statistica dell'autocovarianza dei valori RLE per sequenze ottenute codificando alcuni file del *Canterbury Corpus* [44] e del *Large Canterbury Corpus*.

La speranza era quella di trovare il sussistere di una certa indipendenza tra i valori RLE il che implicherebbe che tutta l'entropia della sequenza sia di ordine zero e quindi "catturabile" da un semplice codice istantaneo (ad esempi γ coding).

Intuitivamente l'indipendenza dovrebbe sussistere in quanto, anche se i simboli appartenenti al testo iniziale fossero tutt'altro che indipendenti (come in realtà accade), l'applicazione della trasformata BWT (che trasforma una stringa con forti interdipendenze tra i caratteri in una stringa con forte proprietà di località) e la successiva organizzazione della trasformata in *wavelet tree* (che trasforma i run della BWT in singoli valori interi) dovrebbe portare a una sequenza RLE di valori non inter-dipendenti. In altre parole, quello che si spera di verificare in pratica (in teoria la dimostrazione è reperibile in [12]) è che la catena di trasformate BWT + *wavelet tree* trasformi l'entropia di ordine h del testo codificato in entropia di ordine zero.

Un passo successivo sarà poi quello di dimostrare che quest'entropia di ordine zero possa essere quasi raggiunta codificando l'*output* della catena di trasformate con γ coding.

3.1 Analisi dell'autocorrelazione e dell'autocovarianza

Per valutare l'interdipendenza tra i valori RLE facciamo uso del concetto di correlazione tra simboli preso in prestito dalla teoria dei segnali. Per un segnale reale $x(t)$ a energia finita si definisce *funzione di autocorrelazione* [17]:

$$R_x(\tau) \triangleq \int_{-\infty}^{\infty} x(t)x(t-\tau)dt$$

Trasportando lo stesso concetto nel dominio del tempo discreto e passando a una sequenza limitata x di lunghezza N definiamo la *sequenza di autocorrelazione*[41] come:

$$R_x(\tau) \triangleq \frac{1}{N-\tau} \sum_{i=1}^{N-\tau} x(i)x(i-\tau) \quad N, i, \tau \in \mathbf{N},$$

La funzione (sequenza) di autocorrelazione fornisce informazioni utili sulla rapidità di variazione del segnale nel tempo, è sempre pari, raggiunge il suo massimo nell'origine, e decresce tanto più velocemente quanto più il segnale è scorrelato.

Per avere informazioni quantitative sulla correlazione tra i simboli passiamo ad analizzare la sequenza di autocovarianza (cioè la sequenza di autocorrelazione del segnale diminuito del suo valore medio) definita come:

$$C_x(\tau) \triangleq \frac{1}{\sigma_x \cdot (N-\tau)} \sum_{i=1}^{N-\tau} (x(i) - \nu_x)(x(i-\tau) - \nu_x); \quad N, i, \tau \in \mathbf{N}, \nu \in \mathbf{R}$$

dove abbiamo sottratto il valore medio al segnale e normalizzato dividendo per la varianza del segnale stesso. L' i -esimo elemento della sequenza di autocovarianza normalizzata così definita è il *coefficiente di autocorrelazione* [25] tra il segnale x e una sua replica ritardata di τ .

Anche l'autocovarianza normalizzata raggiunge il suo massimo nell'origine (il segnale è massimamente correlato con se stesso) ma in più è sempre minore od uguale a 1 e tende a zero per τ tendente a n .

Dal un'analisi qualitativa dei grafici della sequenza di autocovarianza normalizzata calcolata sui primi $5 \cdot 10^5$ simboli di alcuni file del *Canterbury Corpus* e del *Large Canterbury Corpus*[44], vediamo chiaramente che i simboli prodotti dal codificatore RLE sono effettivamente molto scorrelati in quanto la sequenza desce molto velocemente nell'intorno dello zero (non è molto dissimile da una delta di Kronecker [18]).

Notiamo però anche che la suddetta sequenza non è monotona decrescente come ci si potrebbe aspettare (ovvero i simboli sono tanto più scorrelati quanto più sono lontani)

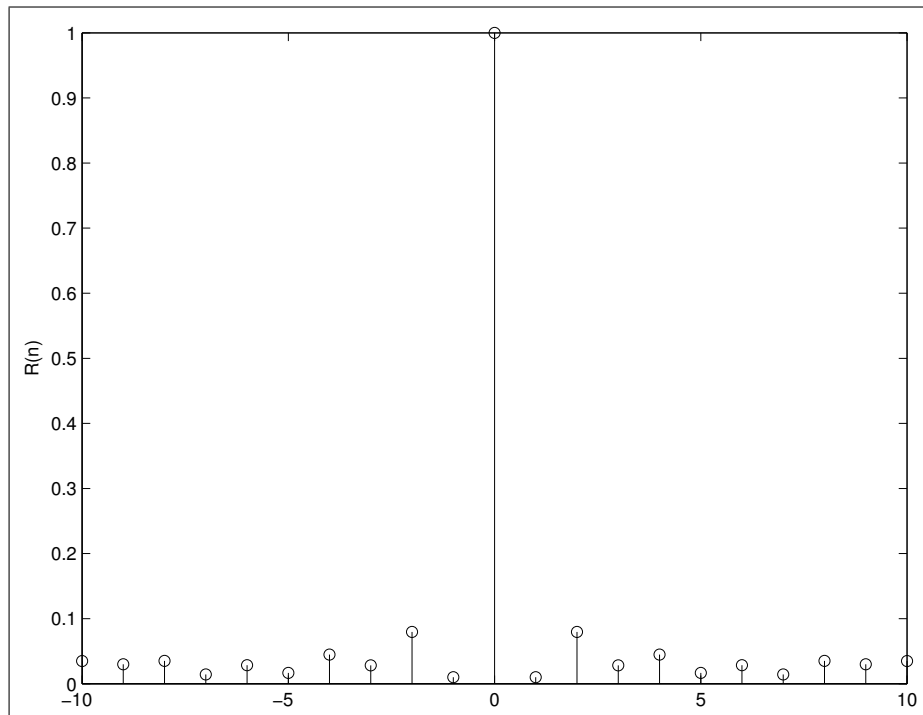


Figura 3.1: Sequenza di autocovarianza normalizzata per bible.txt.

ma, ad esempio $R(0) \gg R(1) < R(2)$. Questo significa effettivamente che la correlazione esistente tra simboli che nel testo compaiono in posizioni adiacenti è pressochè nulla, il che potrebbe fare sperare che sussista indipendenza statistica tra simboli consecutivi ¹ e che quindi anche H_1 sia nulla, probabilmente perché trasformata completamente in entropia di ordine zero dalla trasformata BWT applicata, viceversa, non è allo stesso modo nulla l'entropia di ordine 2 H_2 .

Nonostante la completa indipendenza con alta probabilità non sussista per coppie di valori a distanza 2 (in quanto, come detto, esiste una minima correlazione), di seguito consideremo quest'indipendenza come sussistente per coppie di valori RLE presi a qualsiasi distanza. Questa ipotesi di lavoro come vedremo non è eccessivamente restrittiva ed è necessaria per semplificare (anzi rendere possibili) alcuni calcoli.

Per completezza informativa è necessario riportare che sono stati fatti vari tentativi per misurare direttamente la probabilità che i valori RLE fossero indipendenti l'uno dall'altro

¹da notare che l'incorrelazione implica l'indipendenza solo per segnali con pdf normale, e questo non è certo il nostro caso, mentre l'implicazione inversa è sempre verificata. In ogni caso, la mancanza di correlazione viene spesso utilizzata per giustificare il sussistere di una *probabile* indipendenza statistica.

a diverse distanze temporali (per diversi τ).

Sono stati in particolare utilizzate la statistica *chi-quadro* [10] e alcune statistiche d'ordine come la ρ di Spearman e la τ di Kendall [6] utilizzate per valutare l'indipendenza temporale dei dati nelle serie storiche.

Purtroppo tutte le statistiche utilizzate hanno in comune come requisito quello di essere applicabili su sequenze abbastanza corte (dell'ordine della decina o al massimo centinaia di valori) quindi i risultati non sono affidabili nel nostro caso nel quale vengono trattate sequenze lunghe da alcune decine ad alcune centinaia di migliaia di valori RLE .

Sarebbe interessante, come sviluppo successivo di questo lavoro, compiere una dettagliata analisi di indipendenza dei valori RLE basandosi su tecniche statistiche adatte a sequenze di dati di tale lunghezza.

Capitolo 4

Analisi delle prestazioni di vari codici prefissi

Prima di passare ad un'analisi statistica dettagliata in termini di distribuzione di probabilità dei valori RLE nelle sequenze considerate, si è proceduto a vari test comparativi di codifica delle sequenze utilizzando diversi codici prefissi, estendendo e ampliando il confronto già portato avanti in [12].

4.1 Codificare gli interi

Come esposto più volte precedentemente, il nostro compito si riduce quindi a quello di riuscire a codificare in modo efficiente una sequenza di interi (i valori delle sequenze RLE).

Esistono vari modi per codificare sequenze di interi [35], uno di questi è utilizzare una codifica a byte variabile, in cui si utilizzano i 7 bit più significativi di un byte per codificare una parte dell'intero e l'ultimo bit si pone a **1** se la codifica continua nel byte successivo, altrimenti si mette a **0**, ad esempio, 135 viene rappresentato su due byte come 0000001100001110 che viene letto come 00000010000111 se si rimuovono i due bit meno significativi da entrambi i byte e si concatenano i restanti bit. Tipicamente questo tipo di codifica viene utilizzata nelle liste invertite e negli indici invertiti [36]. Un altro possibile (e spesso più efficiente) metodo di codifica consiste nell'utilizzare codifiche a bit variabili. In questo caso la struttura dell'insieme di dati deve essere conosciuta a priori in modo che possa essere applicato lo schema più adatto ed efficiente.

Esistono due possibili approcci alla codifica a bit variabili: codifiche parametriche e non parametriche. Le codifiche non parametriche rappresentano gli interi utilizzando un codi-

ce fissato mentre le codifiche parametriche rappresentano gli interi relativamente ad una costante calcolata o immagazzinata per la decompressione.

I codici di Elias sono un metodo di codifica non parametrico utilizzato ad esempio in grossi indici di database e in applicazioni specialistiche. I codici di Elias permettono una codifica non ambigua degli interi e non richiedono alcun tipo di separatore tra gli interi adiacenti sistemati in un array.

Esistono due distinti codici di Elias il γ coding ed il δ coding .

Nel γ coding un intero positivo x è rappresentato da $1 + \lfloor \log_2 x \rfloor$ in unario (ovvero $\lfloor \log_2 x \rfloor$ bit **0** seguiti da un bit a **1**) seguito dalla rappresentazione binaria di x meno il suo bit più significativo. Ad esempio, 9 viene rappresentato da 00001001 dato che $1 + \lfloor \log_2 9 \rfloor = 4$ cioè 0001 un unario e la rappresentazione di 9 in binario è 001 senza la cifra più significativa. In questo modo **1** viene rappresentato da **1** , ovvero, rappresentato da un solo bit. Il γ coding risulta quindi essere molto efficiente per i piccoli interi.

I codici δ di Elias sono meno efficienti per interi piccoli. In particolare, un δ coding è costituito dal γ coding di $1 + \lfloor \log_2 x \rfloor$ seguito dalla rappresentazione binaria di x .

Tra i codici parametrici, uno dei più importanti è il codice di Golomb che dipende da un parametro k che va spesso calcolato e immagazzinato.

Un intero rappresentato con il codice di Golomb viene codificato come la rappresentazione unaria del quoziente $q = \lfloor \nu - 1 \rfloor / k + 1$ seguita dalla rappresentazione del resto $r = \nu - q \cdot k - 1$. Usando questo approccio, la rappresentazione in binario di r può richiedere $\lfloor \log k \rfloor$ oppure $\lfloor \log k \rfloor$ bit.

Secondo quanto stato detto, ogni codice prefisso ha efficienza maggiore o minore a seconda del tipo di interi che deve codificare, l'efficienza quindi dipende dalla *distribuzione statistica* degli interi che devono essere codificati.

Per capire questa connessione dobbiamo richiamare il fatto che l'entropia di ordine zero di una sorgente può essere interpretata come il limite inferiore alla lunghezza media delle *codeword* che un codice che codifichi senza perdita di informazione la sorgente può raggiungere. In altre parole, non è possibile codificare la sorgente senza perdita di informazione utilizzando meno bit rispetto a l'entropia di ordine 0 moltiplicata per il numero di simboli della sorgente e assegnando ad ogni simbolo una *codeword* costituita da un numero intero di bit.

Detta $p(x)$ la generica densità di probabilità degli interi x_1, x_2, \dots, x_n si può quindi scrivere che la minima lunghezza media in bit con cui una sorgente può essere codificata è uguale all'entropia di ordine 0 della sorgente: [27]

$$\min E[L(x)] = \min \sum L(x) \cdot p(x) = \max - \sum p(x) \cdot \log_2 p(x)$$

dove la massimizzazione e la minimizzazione riguardano la densità di probabilità $p(x)$, da ciò discende che:

$$L(x) = -\log_2 p(x) \Rightarrow p(x) \propto 2^{-L(x)}$$

che significa che avendo un codice che assegna ad ogni intero x una codeword di lunghezza $L(x)$ bit il codice è ottimale¹ se e solo se gli interi da codificare hanno densità di probabilità proporzionale a $2^{-L(x)}$. Ad esempio nel γ coding la lunghezza della codeword per l'intero x è $L(x) = 1 + 2 \cdot \lfloor \log_2 x \rfloor \approx 2 \cdot \log_2 x$ ottimale per interi che hanno una distribuzione di probabilità uguale a $p(x) = 2^{-L(x)} \propto \frac{1}{x^2}$. Il δ coding è ottimale invece per interi distribuiti come $\frac{1}{x \log_2 x}$.

4.2 Tentativi di codifica con codici prefissi

In alcuni tentativi compiuti in [13] si è cercato di comprimere le sequenze RLE utilizzando come codici γ , δ , *Maniscalco* (un codice studiato appositamente per comprimere gli RLE dopo la BWT [24]), *Golomb* e *Bernoulli* (con parametro b uguale alla mediana dei dati) e *MixBernoulli* (che usa un solo bit per codificare il valore 1 e un bit in più del codice *Bernoulli* standard per codificare gli altri valori). [35]

Si è successivamente tentato di estendere il confronto anche ad altri codici prefissi, in particolare: lo ζ code, un codice parametrico; e un γ coding modificato che esegue preventivamente un *run length encoding* degli 1 consecutivi e codifica in γ coding questo RLE dopo avere emesso un simbolo di *escape* (gli 1, come vedremo, costituiscono dal 45% al 55% della totalità degli altri valori).

I risultati sono riportati nella tabella seguente per i file del *Canterbury Corpus* e del *Large Canterbury Corpus*. Accanto ad ogni file sono riportati i valori in bps (*bit per symbol*) ottenuti dai vari codici su quel file. Non sono riportati i risultati per ζ code poiché i risultati migliori si ottengono con parametro uguale ad 1 e in questo caso lo ζ coding viene a coincidere con il γ coding.

¹nel senso che raggiunge l'entropia di ordine zero assegnando ad ogni simbolo una *codeword* fissata

file	bps γ coding	bps δ coding	bps γ esc
E.coli	2.1780	2.5238	2.4763
alice29.txt	2.3527	2.5816	2.5934
asyoulik.txt	2.6304	2.9104	2.9129
bible.txt	1.6109	1.7677	1.7839
cp.html	2.6949	2.9554	2.9310
fields.c	2.4387	2.6145	2.5894
grammar.lsp	2.8121	3.0636	2.9948
kennedy.xls	1.4269	1.6051	1.4718
lctet10.txt	2.0933	2.2902	2.3047
plravn12.txt	2.4686	2.7469	2.7521
ptt5	.7731	.8600	.8617
sum	2.9500	3.2324	3.1803
world192.txt	1.4699	1.5890	1.6095
xargs.1	3.3820	3.7303	3.6564

Come si può vedere dai dati precedenti, dove per “ γ esc” si intende il γ coding modificato precedentemente descritto, il γ coding coding si rivela essere quello che ottiene la migliore performance, come già sottolineato in [13].

Per eseguire le varie codifiche con diversi codici si è rivelato molto utile utilizzare la libreria MG4J [42] che comprende un’implementazione LGPL della maggior parte degli algoritmi di codifica e indicizzazione che sono descritti nel libro [36].

Utilizzando questo strumento è stato semplice provare velocemente diversi tipi di codifica e misurarne le performance grazie alla struttura modulare dei vari codificatori implementati come *wrapper* degli *stream java* di *input* e *output*.

4.3 Comparazione con il codice di Huffman

Dall’analisi compiuta nella sottosezione precedente emerge che il γ coding si è rivelato essere imbattibile rispetto ad altri codici prefissi.

A questo punto si è deciso di confrontare le prestazioni del γ coding con quelle del codice di Huffman delle sequenze RLE di ogni file, che costituisce il codice ottimo tra i codici prefissi per quella sequenza.

Se si confrontano le lunghezze delle *codeword* di γ coding con le lunghezze delle *codeword* del codice di Huffman per le sequenze di valori RLE calcolate a partire da vari file del

*Canterbury corpus*², si nota che esse differiscono in genere al massimo di 1 le une dalle altre, e, per quanto riguarda le *codeword* che codificano gli interi più piccoli (1,2,3) che sono quelli che, come detto, costituiscono la maggioranza dei valori, le suddette lunghezze coincidono.

Questo implica quindi che le lunghezze delle sequenze RLE codificate con γ coding e Huffman devono essere necessariamente molto simili.

Di seguito riportiamo il confronto tra le prime 20 *codeword* del γ coding e di un codice di Huffman calcolato per il *bible.txt* del Canterbury Corpus.

Valore	Codice di Huffman	Lunghezza Huffman	γ coding	Lunghezza γ coding
1	0	1	1	1
2	111	3	010	3
3	100	3	011	3
4	1011	4	00100	5
5	11001	5	00101	5
6	10100	5	00110	5
7	110101	6	00111	5
8	110000	6	0001000	7
9	1101111	7	0001001	7
10	1101001	7	0001010	7
11	1100010	7	0001011	7
12	1010101	7	0001100	7
13	11011100	8	0001101	7
14	11011000	8	0001110	7
15	11000111	8	0001111	7
15	10101101	8	000010000	9
17	10101000	8	000010001	9
18	110110111	9	000010010	9
19	110110011	9	000010011	9
20	110100010	9	000010100	9

Come si nota dalla tabella, le lunghezze delle *codeword* di Huffman e γ coding coincidono per quasi tutti gli interi, quindi, considerando che la stragrande maggioranza dei valori

²si è preso ad esempio il codice calcolato per la sequenza relativa a *bible.txt* ma i codici sono molto simili per tutti i file del *Canterbury Corpus*

RLE per tutti i file è costituita da interi piccoli (come vedremo successivamente, i primi 20 naturali costituiscono più del 95% delle sequenze di valori RLE in tutti i file) è ovvio che la lunghezza della codifica delle sequenze RLE non possa variare più di tanto utilizzando Huffman piuttosto che γ coding. La precedente intuizione può essere misurata direttamente confrontando la performance effettiva dei codici di Huffman in termini di rapporto di compressione rispetto al γ coding.

Applicando sui file del *Canterbury Corpus* una codifica di tipo Huffman rispetto al γ coding l'aumento del rapporto di compressione è minimale, in media dell'ordine del 3-4%.

È inoltre da notare il fatto che in questo rapporto di compressione non viene tenuto conto dell'*over-head* determinato dalla memorizzazione del dizionario di Huffman, necessaria per potere poi decomprimere i dati.

Il valore di miglioramento prima riportato è quindi da considerarsi come un limite superiore, assolutamente non raggiungibile, considerando che nel file compresso con Huffman deve essere salvato anche l'albero di Huffman che, nel caso si utilizzi l'algoritmo di Huffman canonico, occupa su disco $2n$ byte dove n è il numero di simboli (nel nostro caso valori RLE) presenti nel file.

Nel caso delle nostre sequenze di RLE, il numero di possibili valori è davvero alto (teoricamente illimitato³) come si può vedere dalla tabella seguente:

³Teoricamente la dimensione dell'alfabeto dei valori RLE è limitata solo dal numero di caratteri che compaiono nel file che si va a codificare. Al limite, un file di n caratteri tutti uguali conterrebbe un RLE di valore pari a n . Nei file reali, come visto precedentemente, l'alfabeto contiene comunque un numero di simboli molto alto

File	Massimo valore presente nell'RLE associato
E.coli	1003065
alice29.txt	5227
asyoulik.txt	3381
bible.txt	200417
cp.html	518
fields.c	395
grammar.lsp	253
kennedy.xls	79113
lcet10.txt	8426
plravn12.txt	21472
ptt5	36316
sum	539
world192.txt	65135
xargs.1	114

di conseguenza lo spazio necessario per memorizzare l'albero di Huffman sarebbe assolutamente non trascurabile e in certi casi supererebbe le dimensioni dello stesso file codificato. In definitiva, quindi, il γ coding è di gran lunga migliore dei codici di Huffman in termini di complessità di Kolmogorov [28], poiché, anche se la dimensione del file codificato con Huffman è minore (comunque di pochissimo), rispetto a quella ottenuta codificando con γ coding, l'informazione aggiuntiva di cui Huffman necessita lo fa diventare molto più inefficiente del γ coding.

In ogni caso, questo confronto ci dà un'indicazione forte sul fatto che non sia facilmente costruibile un codice prefisso che superi le performance di γ coding, e in ogni caso, anche se si ottenesse, il miglioramento ottenibile sarebbe minimo (al massimo quello ottenuto dal codice di Huffman, che si attesta sul 3-4%)

Capitolo 5

Alternative al γ coding

Come detto nelle sezioni precedenti, il γ coding è quasi ottimale per codificare le sequenze RLE perché è molto simile al codice di Huffman.

Non è quindi possibile trovare un codice prefisso che abbia performance molto superiori a γ coding, poiché i codici di Huffman stessi, che sono ottimali per definizione, non lo migliorano se non di poco.

Esistono però vari metodi per ottenere un'efficienza superiore dei codici prefissi, due sono i più diffusi.

Il primo consiste nel codificare gruppi di valori come se fossero un unico valore *blocking*. In questo modo il bit perso per arrotondamento viene diviso tra tutti i valori che compongono il gruppo che viene codificato, il che si traduce in una diminuzione del numero totale di bit persi per arrotondamento.

In questo caso, però il numero di simboli costituenti l'alfabeto aumenta esponenzialmente rispetto al numero di valori che vengono codificati in uno stesso blocco, e dato che l'alfabeto dei valori della sequenza RLE ha già dimensioni molto elevate, questo approccio non è perseguibile.

Un altro possibile approccio migliorativo consiste nell'utilizzare codici che assegnano ai simboli un numero di bit non intero. Questo è l'approccio utilizzato, ad esempio, dai codici aritmetici

5.1 Codice aritmetico

Il codificatore aritmetico [2] è uno dei codificatori di entropia più utilizzati in quanto riesce di solito a raggiungere performance di compressione migliori rispetto al codice di Huffman. L'idea su cui si basa un codificatore aritmetico è quella per cui tutto lo *stream* di *input*

possa essere codificato in un unico reale *floating point* tra 0 e 1. Per capire meglio come opera un codificatore aritmetico, consideriamo una linea di probabilità tra 0 e 1 e assegnamo ad ogni simbolo un intervallo su questa linea basata sulla sua probabilità di comparire nello *stream*, più alta la probabilità di occorrenza del simbolo, più grande l'intervallo che viene assegnato ad esso. Gli intervalli sono tutti disgiunti e la loro unione deve dare l'intervallo completo tra 0 e 1.

Una volta definito la divisione in intervalli della linea di probabilità, si comincia a codificare i simboli. L'idea è che ogni simbolo raffina la posizione in cui il *floating point* che rappresenta lo *stream* di *input* cadrà.

Supponiamo di avere:

Simbolo	Probabilità	Intervallo
a	2	[0.0 , 0.5)
b	1	[0.5 , 0.75)
c	1	[0.7.5 , 1.0)

L'algoritmo per il codice aritmetico opera come di seguito

1. **Low** = 0
2. **High** = 1
3. Per ogni simbolo dello *stream* di *input*:
 - (a) **Range** = **High** - **Low**
 - (b) **High** = **Low** + **Range** · **high_range** (valore alto del simbolo che sta per essere codificato)
 - (c) **Low** = **Low** + **Range** · **low_range** (valore basso del simbolo che sta per essere codificato)

Dove:

Range tiene traccia di dove dovrebbe essere il nuovo intervallo.

High e **Low**, specificano il numero di output.

Vediamo un esempio per la stringa “baca” utilizzando gli intervalli di probabilità definiti in tabella 5.1:

Simbolo	Intervallo Range	Valore basso Low	Valore alto High
		0	1
b	1	0.5	0.75
a	0.25	0.5	0.625
c	0.125	0.59375	0.625
a	0.03125	0.59375	0.609375

Il numero restituito in *output* che rappresenta lo *stream* di *input* sarà 0.59375. Come si vede dall’esempio, il codificatore parte avendo un intervallo pari a l’intervallo completo tra 0 e 1 e poi lo restringe selezionando ogni volta il sotto-intervallo che rappresenta il simbolo che viene processato.

Il metodo di decodifica consiste nel vedere prima dove il numero dato in *output* si trova ed estrarre l’intervallo relativo di conseguenza, per ogni simbolo.

1. **Low** = 0
2. **High** = 1
3. Per ogni simbolo dello *stream* di input:
 - (a) **Range** = **high_range** del simbolo - **low_range** del simbolo
 - (b) **Number** = **Number** - **low_range** del simbolo
 - (c) **Number** = **Number** / **Range**

Di seguito l’esempio di come avviene la decodifica.

Simbolo	Intervallo Range	<i>Floating Point</i> Number
b	0.25	0.59375
a	0.5	0.375
c	0.25	0.75
a	0.5	0

5.2 Implementazione

Come si può dedurre dall'esempio, è necessario che il numero reale che rappresenta lo *stream* di *input* venga passato al decompressore senza arrotondamenti. L'aritmetica dei calcolatori impone però che la massima precisione raggiungibile sia di 80 bit, il che non permette di lavorare con l'intero numero. Il numero reale rappresentante lo *stream* compresso verrà quindi creato incrementalmente. Questo approccio è possibile considerando che, se le prime cifre dell'intervallo alto e quelle dell'intervallo basso coincidono, queste non verranno più modificate dalla successiva raffinazione apportata dagli altri simboli, quindi possono essere emesse in *output*. Una volta emesse in *output* queste cifre il processo di raffinazione del numero reale può essere portato avanti solo sulle cifre che differiscono tra l'estremo superiore e quello inferiore dell'intervallo.

Rimane da trattare il metodo con cui vengono assegnati gli intervalli di probabilità ai vari simboli.

Si è detto che la dimensione dell'intervallo assegnato ai simboli deve essere proporzionale alla probabilità di occorrenza dei simboli nel testo che si sta codificando. È quindi possibile scandire il testo una volta, creare il modello statistico, codificare il testo con quel modello e salvare il modello stesso nel file di *output* (il modello sarà poi necessario al decodificatore per ricostruire la divisione in intervalli e quindi il testo). Un approccio più efficiente e più utilizzato consiste nel fare partire sia il codificatore che il decodificatore con lo stesso modello statistico (ad esempio, si suppone che tutti i simboli siano equiprobabili e si assegnano a tutti i simboli intervalli di dimensione costante) che viene modificato in dipendenza dei simboli letti.

Ad esempio, prendiamo un considerazione un codificatore che abbia un modello statistico iniziale che considera tutti i simboli equiprobabili. Se il codificatore con questo modello si trovasse poi a comprimere un testo nel quale il carattere "a" occorre molto più spesso rispetto a tutti gli altri caratteri allora potrebbe aumentare la dimensione dell'intervallo che rappresenta il simbolo "a" in modo da codificarlo in un numero minore di bit (chiaramente, maggiore è la dimensione dell'intervallo, minore il numero di bit che servono per codificarlo). Questo tipo di approccio viene definito "adattivo" in quanto il codificatore "si adatta" all'*input* che sta processando. Il decodificatore può ricostruire il testo codificato compiendo le stesse operazioni di aggiornamento dell'intervallo che ha compiuto il codificatore.

Capitolo 6

Tentativi di compressione con aritmetico

Al fine di migliorare le prestazioni di compressione del γ coding si è tentato di comprimere le sequenza RLE utilizzando un codificatore aritmetico, in particolare il codificatore aritmetico fornito con il *tool* MG4J [42]¹, i risultati sono riportati nella tabella sottostante.

File	bps γ coding	bps aritmetico	% peggioramento
E.coli	2.1780	2.7797	21.64%
alice29.txt	2.3527	2.4964	5.75%
asyoulik.txt	2.6304	2.7324	3.73%
bible.txt	1.6109	1.8190	11.44%
cp.html	2.6949	2.7170	0.81%
fields.c	2.4387	2.4645	1.04%
grammar.lsp	2.8121	2.9282	3.96%
kennedy.xls	1.4269	1.6834	15.23%
lcet10.txt	2.0933	2.1727	3.65%
plrabn12.txt	2.4686	2.6591	7.16%
ptt5	.7731	.9983	22.55%
sum	2.9500	2.9184	-1.08%
world192.txt	1.4699	1.5815	7.05%
xargs.1	3.3820	3.3763	-0.16%

¹Questo codificatore si ispira a quello proposto da John Carpinelli, Radford M. Neal, Wayne Salamonsen e Lang Stuiver, che è a sua volta basato sull'*Arithmetic Coding Revisited* di Alistair Moffat [21] descritto anche in [36]

La cattiva performance rispetto al γ coding che mostra la tabella 6 dipende probabilmente dal fatto che i valori da codificare appartengono a un alfabeto teoricamente illimitato e il codificatore aritmetico di Moffat, che per determinare la lunghezza dell'intervallo da assegnare ad ogni simbolo mantiene una tabella con il conteggio cumulativo delle frequenze, funziona bene per alfabeti di piccole dimensioni.

Il motivo è semplice e dipende dal fatto che l'aritmetico sopra citato si basa su un modello statistico adattivo, tramite il quale decide ad ogni passo la dimensione dell'intervallo in cui codificare il simbolo, che viene modificato in base ai simboli già compressi. Per assegnare un intervallo a un simbolo in modo efficiente (ovvero predicendo con buona approssimazione quale sarà la frequenza del simbolo nel testo) è necessario conoscere con la stessa approssimazione la frequenza anche degli altri simboli. Nelle sequenze di RLE prese in esame, però, alcuni valori (i simboli che aritmetico deve comprimere) possono comparire molto poco di frequente, alcuni di questi anche solo una volta all'interno del file, è quindi impossibile predire efficientemente quale sarà la frequenza di tutti i simboli (e anche solo quanti saranno i simboli in totale) al fine di calcolare efficientemente gli intervalli.

Il problema consiste quindi nel fatto che un codificatore aritmetico adattivo, senza nessun particolare accorgimento e senza nessuna informazione sul possibile modello statistico dei simboli, impiega troppo "tempo" per apprendere quale sia la statistica delle sequenze di RLE, o meglio, non riesce ad apprenderla prima della fine del file stesso.

È quindi necessario dare al codificatore aritmetico qualche informazione di più sulla statistica dei simboli che deve andare a codificare.

Capitolo 7

Analisi della distribuzione statistica degli RLE

Come già detto precedentemente, è necessario conoscere la distribuzione probabilistica dei simboli con precisione per potere ottenere buone prestazioni nella compressione utilizzando un codificatore del tipo aritmetico basato su modello statistico di ordine zero.

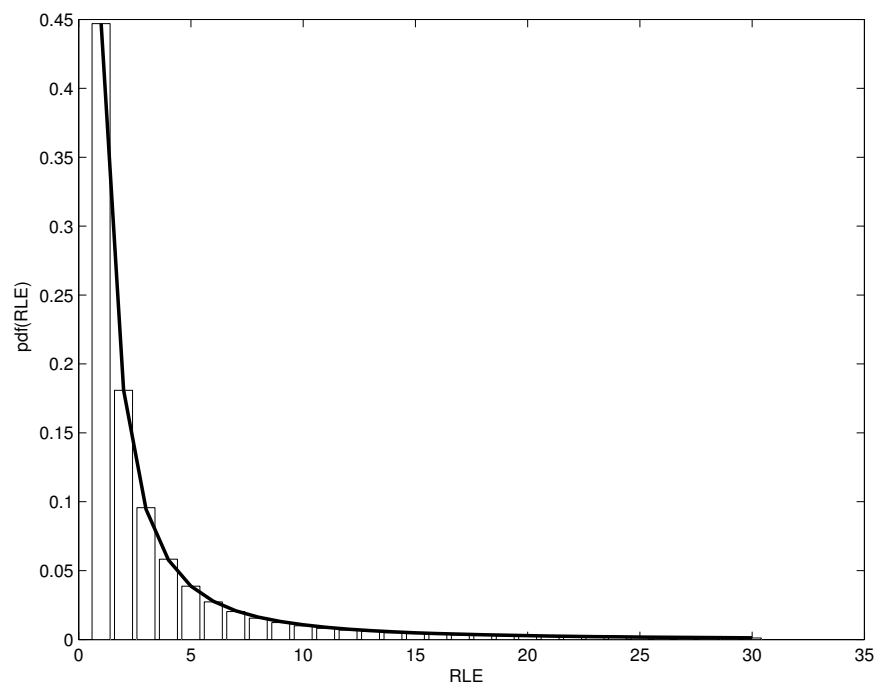


Figura 7.1: pdf di bible.txt e curva di approssimazione.

Dopo uno studio qualitativo dei grafici della densità di probabilità (da ora in avanti pdf) per vari file (v. Figura 7.1) si è cercato di trovare una funzione che approssimasse bene i dati sperimentali.

7.1 Metodo dei minimi quadrati

Per fare ciò si è utilizzato l'*utility* `LSQCurveFit` di MATLAB che esegue il *fitting* di una funzione parametrica su un insieme di dati scegliendo i parametri in modo che venga minimizzata la somma dei quadrati degli scarti tra i dati e la funzione, ovvero, detta S la seguente somma^[37]:

$$S(\theta) = \sum_{i=1}^k [f(\theta, x_i) - y_i]^2$$

Il *tool* `LSQCurveFit` trova iterativamente il valore di θ che minimizza S , e restituisce, oltre ai suddetti θ e S , anche un vettore contenente gli scarti ¹.

Sono vari gli indici che si possono utilizzare per valutare la bontà dell'approssimazione, uno di questi, il più semplice, è proprio il valore di S restituito da `LSQCurveFit`. Chiaramente, un'approssimazione ottimale avrebbe $S = 0$. S non costituisce però di per sé un'indicazione assoluta sulla bontà dell'approssimazione in quanto dipende dal valore assoluto dei dati da approssimare. Un migliore indicatore è costituito dalla media del quadrato degli scarti relativi:

$$S_\nu(\theta) = \frac{1}{k} \sum_{i=1}^k \left[\frac{f(\theta, x_i) - y_i}{y_i} \right]^2$$

Un altro indice di bontà può essere identificato nella distribuzione degli scarti. Infatti l'approssimazione è da considerarsi tanto migliore quanto più gli scarti sono distribuiti intorno allo zero con una distribuzione che tende alla distribuzione normale, in quanto significa che i dati differiscono da essa solo per un errore sperimentale (normalmente distribuito).

Nel nostro caso, in realtà, la distribuzione normale degli scarti nell'intorno dello zero non è necessariamente da considerarsi un buon indicatore di *fitting*, dato che le sequenze RLE non sono dati sperimentali a tutti gli effetti e non sono quindi necessariamente soggetti ad errore sperimentale in termini di rumore gaussiano aggiunto. Considerato questo fatto, decidiamo quindi di tenere conto solo di S_ν come indicatore della bontà del *fitting* ottenuto.

¹Si definisce scarto la differenza tra il dato sperimentale osservato per un certo valore di x e la funzione calcolata in quell' x .

Nonostante siano stati fatti molti tentativi con vari tipi di funzioni, tra cui esponenziali, polinomiali e miste (in particolare, polinomi moltiplicati per esponenziali) non si è riuscito a trovare una funzione che avesse un basso S_v e scarti distribuiti uniformemente.

Si sono ottenuti risultati migliori analizzando la distribuzione cumulativa di probabilità dei simboli (da ora in avanti, `cdf`). Utilizzando infatti lo stesso *toolbox* di MATLAB prima citato si sono ottenuti buoni risultati approssimando la `cdf` con la funzione.

$$cdf(x) = e^{-\frac{a}{x}} \quad x \in \mathbf{N} \quad x > 0 \quad (7.1)$$

La funzione (7.1) dimostra di riuscire ad approssimare molto bene i dati sperimentali in quanto ha un S_v molto piccolo (da uno a due ordini di grandezza minore rispetto altre funzioni testate);

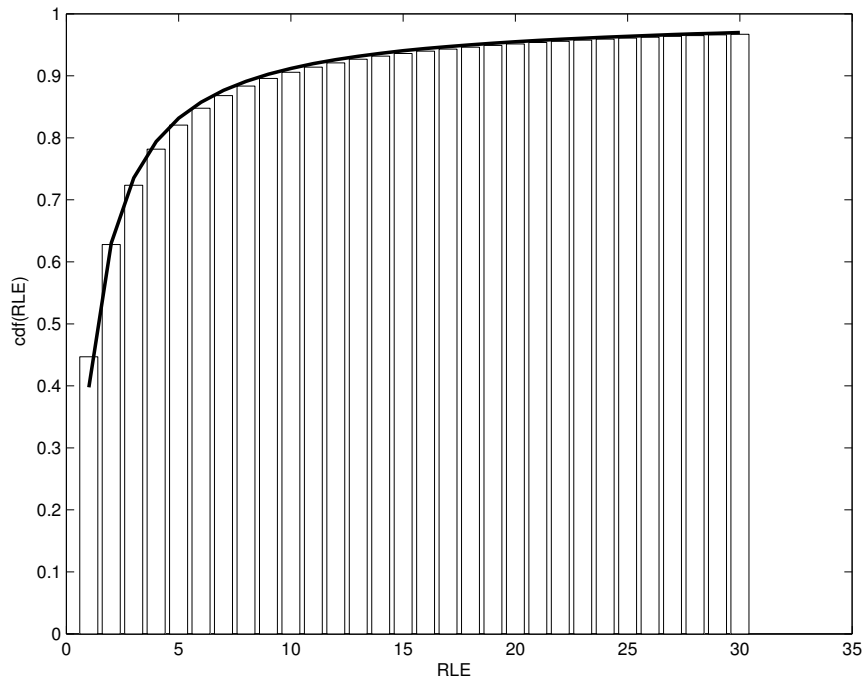


Figura 7.2: `cdf` per bible.txt e curva di approssimazione.

Considerando la `cdf` come una funzione continua su \mathbf{R} , derivandola rispetto ad x e normalizzando si è ottenuto:

$$pdf(x) = \left(\frac{ae^{-\frac{a}{x}}}{x^2} \right) / \left(\sum_{i=1}^{\infty} \frac{ae^{-\frac{a}{i}}}{i^2} \right) \quad i, x \in \mathbf{N}, x > 0 \quad (7.2)$$

dove $\sum_{i=1}^{\infty} \frac{ae^{-\frac{a}{i}}}{i^2}$ indica la sommatoria rispetto a x di tutti i simboli, ovvero è il fattore di normalizzazione. Il *fitting* della funzione (7.2) sulla pdf dei valori RLE è anch'esso molto buono e questo indica implicitamente che l'approssimazione della cdf a funzione continua su \mathbf{R} non porta errori di modellizzazione². Se si considera la pdf trovata, si nota che $\text{pdf}(x) \sim \frac{1}{x^2}$ con x che tende a infinito, ovvero:

$$\lim_{x \rightarrow \infty} e^{-\frac{a}{x}} = 1 \Rightarrow \lim_{x \rightarrow \infty} \left(\frac{ae^{-\frac{a}{x}}}{x^2} \right) / \left(\sum_{i=1}^{\infty} \frac{ae^{-\frac{a}{i}}}{i^2} \right) \propto \frac{1}{x^2}$$

Dato che, come si è visto nella sezione 4.1 γ coding è ottimale per distribuzioni proporzionali a $\frac{1}{x^2}$, ora possiamo capire il motivo per cui γ coding ottiene risultati così buoni sulle sequenze RLE. In ogni caso, come ribadito precedentemente, questo risultato indica solamente una misura della buona performance dei codici prefissi. Codici in grado di assegnare un numero frazionario di bit potrebbero ottenere miglioramenti significativi.

7.2 Analisi dei dati per determinare i parametri

Analizzando il *fitting* della funzione 7.1 su vari file di dati si nota che l'approssimazione è ugualmente molto buona per ciascuno di essi (basso S_v per tutti), mentre il parametro a cambia (anche se non sensibilmente) da file a file (tabella 7.2).

²in quanto la pdf, ricavata dalla cdf per derivazione simbolica, operazione di per sé non corretta in quanto applicabile solo se la funzione fosse continua, approssima effettivamente la pdf empirica dei valori RLE.

File	a	Scarto relativo quadratico medio
alice29.txt	0.9527	4.8e-05
asyoulik.txt	0.8352	1.45e-05
bible.txt	0.9035	2.63e-05
cp.html	1.1001	5.88e-05
kennedy.xls	0.5271	2.13e-05
lcet10.txt	0.9560	3.93e-05
plravn12.txt	0.7663	1.55e-05
ptt5	0.7619	1.08e-05
sum	1.1605	7.65e-05
world192.txt	1.0949	1.49e-04
fields.c	1.6792	5.33e-04
grammar.lsp	1.3401	1.60e-04
xargs.1	1.0997	1.15e-04
E.coli	0.5000	2.29e-04

Il *fitting* dei dati con la funzione (7.1) riesce ad ottenere in media su tutti i file uno scarto relativo quadratico medio di due ordini di grandezza inferiore rispetto ad altre approssimazioni testate.

Da questo si deduce immediatamente che per ottenere una compressione ottimale sarebbe auspicabile che il modello statistico parametrico modificasse il proprio parametro durante la compressione per adattarsi alle caratteristiche specifiche di ogni file, che si riflettono sul parametro a (compressione con modello statistico adattivo). Tratteremo successivamente la possibilità di rendere adattivo il codificatore.

7.3 Range Encoder

Una volta chiarito quale sia il modello statistico dei valori nelle sequenze RLE, ovvero il modo in cui calcolare la dimensione degli intervalli da assegnare ad ogni valore RLE, si è passato all'implementazione del programma.

L'implementazione del modello statistico è stata fatta *ex-novo* in quanto modello studiato *ad hoc* per sequenze RLE prese in esame e quindi presumibilmente non reperibile in altro modo.

Per l'implementazione del codificatore vero e proprio (codificatore di entropia) ci si è invece basati sull'implementazione del *range coder* di Michael Schindler [30] reperibile a: [43]

Il *range encoder* [19] è molto simile a un codificatore aritmetico e si differenzia da esso in quanto non tratta l'output come un numero binario ma come un numero in base 256. Questa differenza porta a un degrado delle prestazioni del tutto trascurabile (i file compressi con *range coder* hanno dimensioni in media lo 0,01% maggiori rispetto ad aritmetico) ma in compenso fa sì che il tempo impiegato nella codifica venga mediamente dimezzato rispetto ad aritmetico. Inoltre *range coder* è coperto da licenza GPL [39], a differenza del codice aritmetico che è coperto da brevetto.

Capitolo 8

Dettagli implementativi

Da ora in avanti ci si riferirà direttamente all'implementazione di M. Schindler precedente descritta. Consideriamo prima la parte di compressione, successivamente tratteremo come eseguire la decompressione.

Per codificare un simbolo x con *range encoder* è necessario:

1. Calcolare la frequenza dei simboli minori di x detta lt_f . (Si suppone che tra i simboli sia definito una relazione di ordine stretto, tipo ordinamento alfabetico).
2. Calcolare la frequenza dei simboli uguali a x , chiamiamo questo valore sy_f .
3. Chiamare la funzione

```
void encode_freq( rangecoder *rc, freq sy_f, freq lt_f, freq tot_f );
```

dove rc è il puntatore all'oggetto *range coder* utilizzato e tot_f è la frequenza di tutti i simboli.

Da notare che sy_f , lt_f , tot_f , sono numeri interi, e, per evitare rinormalizzazioni (le divisioni tra interi sono operazioni molto complesse) di solito coincidono con il numero di simboli effettivamente letto.

Range encoder mette a disposizione anche un'altra funzione per codificare un simbolo, ovvero:

```
void encode_shift( rangecoder *rc, freq sy_f, freq lt_f, freq shift );
```

che permette di codificare un simbolo x che compare con frequenza sy_f mentre tutti i simboli minori di x compaiono con frequenza lt_f solo che, al posto di utilizzare tot_f

come lunghezza totale dell'intervallo su cui normalizzare i valori di sy_f e lt_f viene utilizzato il valore di 2^{shift} cioè, in notazione \mathbb{C} , $1 \ll shift$, velocizzando notevolmente il calcolo.

Nel nostro caso si è deciso di utilizzare questa seconda funzione e di fissare la lunghezza totale dell'intervallo al valore di 2^{19} (quindi $shift = 19$) che è anche il valore consigliato dall'implementatore per evitare imprecisioni dovute a effetti di arrotondamento.

Per prima cosa è quindi necessario calcolare sy_f e lt_f . Utilizzando il modello statistico descritto nella sezione 7 i simboli x appartenenti alla sequenza RLE sono tutti positivi (non nulli).

In questo caso il valore:

$$\begin{aligned} lt_f &= 2^{19} \cdot e^{\frac{-a}{x-1}} \quad \text{se } x > 1, \quad 0 \quad \text{altrimenti} \\ sy_f &= 2^{19} \cdot e^{\frac{-a}{x}} - lt_f \end{aligned} \quad (8.1)$$

A questo punto sorge un problema: quale valore fissare per a ?

Sarebbe necessario calcolare a a partire dai simboli già codificati. In prima approssimazione fissiamo $a = 0.88$ che si è dimostrato essere un valore che raggiunge alte performance di compressione su quasi tutti i file testati. Nella sezione successiva verrà introdotto un modo che permette di inferire il valore del parametro a a partire dai valori RLE già compressi.

Un altro problema fondamentale discende dal fatto che i valori presenti nella sequenza di RLE possono essere interi di qualsiasi dimensione, mentre utilizzando la formula (8.1) per calcolare le frequenze ci si deve inevitabilmente confrontare con la precisione dell'aritmetica intera. Man mano x aumenta il valore di sy_f diminuisce (il che è coerente con il nostro modello statistico) e per valori di x che superano un certo limite si ha inevitabilmente che sy_f viene a essere nulla.

Chiaramente non ha senso pensare di codificare un simbolo in un intervallo di lunghezza nulla, la lunghezza più piccola possibile di intervallo in modo che questo sia non nullo è 1 dato che stiamo lavorando in aritmetica intera.

In realtà siamo costretti ad aggiungere 1 a questo limite inferiore per evitare errori di arrotondamento e ottenere un migliore funzionamento di *range encoder*, quindi, formalizzando quanto detto, è necessario che:

$$2^{19} \cdot (e^{\frac{-a}{x}} - e^{\frac{-a}{x-1}}) \geq 2 \quad (8.2)$$

Che, anche considerando il parametro a variabile, come sarà fatto nella prossima sezione ma tenendo conto comunque che $0.5 < a < 1.8$ come si può desumere dai dati in tabella 7.2 risulta che $x < 362$, per ciò fissiamo il valore del massimo simbolo codificabile a 350.

A questo punto si evidenzia quindi la necessità di definire come codificare i valori RLE maggiori di 350. Esistono varie possibili alternative, per quanto ci riguarda, detto x un valore maggiore di 350, i passi eseguiti dall'algoritmo di codifica che si è scelto di utilizzare sono i seguenti:

1. x viene rappresentato su tre cifre in base 350, ovvero viene espresso come $a \cdot 350^2 + b \cdot 350 + c$, con $a, b, c < 350$
2. vengono codificati tanti caratteri di *escape* quante sono le cifre di x espresso in base 350 meno una, in altre parole se $350 \leq x \leq 350^2$ viene emesso un carattere di *escape*, se $x > 350^2$ ne vengono emessi due. Se x fosse minore di 350 non verrebbero ovviamente codificati caratteri di *escape* e verrebbe codificato direttamente il simbolo x rappresentabile con un'unica cifra in base 350.

Nel nostro caso un simbolo di *escape* è un simbolo che ha:

$$\begin{aligned} lt_f &= 2^{19} \cdot e^{\frac{-a}{350}} \\ sy_f &= 2^{19} - lt_f \end{aligned} \tag{8.3}$$

ovvero è un simbolo al quale è associata la porzione di intervallo che non è coperta dagli altri simboli. Da notare che il calcolo di lt_f e sy_f non viene eseguito ogni volta ma i valori vengono calcolati una sola volta all'inizio della codifica in quanto coinvolgono solo quantità costanti.

3. Partendo dalla cifra più significativa di x espresso in base 350 non nulla si codifica la cifra calcolando le frequenze come descritto nella formula (8.1).

Le successive cifre di x vengono codificate, partendo dalla più significativa fino alla meno significativa calcolando le frequenze associate in modo uniforme ovvero detta x_i la cifra da codificare:

$$\begin{aligned} lt_f &= \frac{2^{19}}{350} \cdot x_i \\ sy_f &= \frac{2^{19}}{350} \end{aligned} \tag{8.4}$$

Il motivo per cui le cifre successive non vengono codificate utilizzando la formula (8.1) è semplice da comprendere se si pensa che non ha nessun significato pensare che le cifre meno significative dei simboli maggiori di 350 seguano qualche tipo particolare di distribuzione, esse probabilmente saranno uniformemente distribuite tra 0 e 349.

È da notare il fatto che in questo modo non possono essere codificati valori maggiori di $350^3 - 1 = 42,874,999$ ma questo non costituisce un problema poichè un valore del genere nella sequenza RLE potrebbe occorrere solo se la BWT del file originario contenesse 350^3 caratteri identici consecutivi, il che a sua volta potrebbe accadere in situazioni patologiche come ad esempio un file di 50 Megabyte costituito dalla ripetizione dello stesso carattere.

In ogni caso, è comunque possibile iterare la scomposizione dei simboli maggiori di 350 in modo illimitato, ovvero esprimerli in un numero qualsivoglia grande di cifre in base 350. Il limite di 3 cifre è stato infatti imposto per semplificare l'implementazione del codificatore e perché più che sufficiente per testare le prestazioni dello stesso sui file del *Canterbury Corpus* ma può essere facilmente rimosso.

4. codifica tre simboli di *escape* consecutivi a indicare la fine file.

8.1 Decodificatore

Il decodificatore associato al codificatore appena descritto è completamente simmetrico ad esso. Il sussistere di tale simmetria è ovvio da comprendere se si pensa che il parametro $a = 0.88$ è conosciuto sia dal codificatore che dal decodificatore e che quindi ogni simbolo codificato dal codificatore aritmetico con un certo modello statistico viene anche decodificato dal decodificatore con lo stesso modello statistico. In particolare, il decodificatore del *range encoder* mette a disposizione una funzione, simmetrica a quella utilizzate per la codifica, ovvero:

```
freq decode_culshift( rangecoder *ac, freq shift );
```

che restituisce un intero che rappresenta la frequenza di un simbolo. Detto y il valore restituito dalla precedente funzione, il decodificatore poi non fa altro che ritrovare quale sia il simbolo x per cui valga che $lt_f(x) \leq y \leq lt_f(x) + sy_f(x)$ (dove $lt_f(x)$ e $sy_f(x)$ hanno lo stesso significato utilizzato nella sezione precedente per il simbolo x). Quindi, invertendo la formula (8.1) si ottiene che il simbolo x a essere decodificato è:

$$\begin{aligned}
 x &= 1 \quad \text{se } freq = 0 \\
 x &= \frac{a}{Shift \cdot \log(2) - \log(y)} + 1 \quad \text{altrimenti}
 \end{aligned}
 \tag{8.5}$$

I precedenti calcoli sono da considerarsi eseguiti in virgola mobile e forniscono risultati interi.

Se x viene ad essere superiore a 350 significa che è stato decodificato un carattere di *escape*, in questo caso il decodificatore si comporta come segue:

1. continua a decodificare valori finchè legge simboli di *escape*.
2. dopo il primo valore letto non di *escape*, decodifica tanti altri valori quanto i caratteri di *escape* letti meno due, utilizzando come modello statistico il modello uniforme, ovvero secondo la:

$$x = 350 \cdot \frac{y}{2^{19}} + 1$$

3. costruisce il vero valore presente nelle sequenze RLE a partire da quelli letti dopo l'ultimo carattere di *escape* considerandoli come cifre in base 350 del vero valore da decodificare. Il primo valore decodificato dopo l'*escape* rappresenta la cifra più significativa di tale rappresentazione.

In altre parole, supponiamo x_1, x_2 e x_3 i tre valori decodificati dopo due caratteri di *escape*, il vero valore che compariva nelle sequenze di RLE, da estrarre da questi ultimi, viene calcolato come:

$$x = x_1 \cdot 350^2 + x_2 \cdot 350 + x_3$$

4. Termina la codifica quando vengono incontrati tre caratteri di *escape* consecutivi.

8.2 Risultati

L'implementazione del codificatore descritto precedentemente raggiunge prestazioni abbastanza soddisfacenti in termini di tempo ed efficienza di compressione. In particolare, di seguito vengono messe a confronto le prestazioni di γ coding e del codificatore *range coder* prima descritto in termini di *bps* raggiunte su vari file del *Canterbury Corpus* e su *random.txt* dell'*Artificial Corpus*.

File	γ coding	codificatore a <i>range coder</i>	% miglioramento
E.coli	2.1780	2.1017	-3.63%
alice29.txt	2.3527	2.3247	-1.2%
asyoulik.txt	2.6304	2.5875	-1.65%
bible.txt	1.6109	1.5901	-1.3%
cp.html	2.6949	2.6465	-1.82%
fields.c	2.4387	2.4186	-0.83%
grammar.lsp	2.8121	2.7648	-1.71%
kennedy.xls	1.4269	1.3998	-1.93%
lcet10.txt	2.0933	2.0650	-1.37%
plrabn12.txt	2.4686	2.4277	-1.68%
ptt5	.7731	.7582	-1.96%
sum	2.9500	2.8792	-2.45%
world192.txt	1.4699	1.4540	-1.09%
xargs.1	3.3820	3.3404	-1.24%
random.txt	6.7949	6.5210	-4.2%

File	codificatore aritmetico	codificatore a <i>range coder</i>	% miglioramento
E.coli	2.7797	2.1017	-32.25%
alice29.txt	2.4964	2.3247	-7.38%
asyoulik.txt	2.7324	2.5875	-5.6%
bible.txt	1.8190	1.5901	-14.39%
cp.html	2.7170	2.6465	-2.66%
fields.c	2.4645	2.4186	-1.89%
grammar.lsp	2.9282	2.7648	-5.91%
kennedy.xls	1.6834	1.3998	-20.26%
lcet10.txt	2.1727	2.0650	-5.21%
plravn12.txt	2.6591	2.4277	-9.53%
ptt5	.9983	.7582	-31.66%
sum	2.9184	2.8792	-1.36%
world192.txt	1.5815	1.4540	-8.76%
xargs.1	3.3763	3.3404	-1.07%
random.txt	6.1273	6.5210	6.03%

Come si evidenzia dai dati precedenti, il codificatore proposto basato sul *range coder* migliora in media di qualche punto percentuale le prestazioni del γ *coding* in termini di rapporto di compressione. Il miglioramento è sensibile (4,2%) sul file `random.txt` che contiene caratteri generati casualmente tra un insieme di 64 e su `E.coli` che contiene il patrimonio genetico dell' *Escherichia Coli* (3,63%). Il miglioramento è molto più visibile se si confrontano le prestazioni del codificatore proposto con il codificatore aritmetico trattato nella sezione 4 (tabella (8.2)), in questo caso, infatti, il codificatore a *range coder* guadagna molto in media su tutti i file, tranne che su `random.txt` sul quale perde il 6%.

Capitolo 9

Versione adattiva

Riprendendo il discorso lasciato alla sezione 7.2 trattiamo ora la possibilità di realizzare una versione adattiva del codificatore.

Per realizzare una versione adattiva è necessario trovare un modo per inferire dai simboli già compressi, conoscendo la loro pdf parametrica rispetto ad a , il valore ottimale del parametro a stesso (ovvero quel valore per il quale sarebbe più probabile per la pdf in questione generare il set di simboli precedentemente codificati). Si è scelto di utilizzare un algoritmo ML per ricavare il valore del parametro a ottimale a partire dalla sequenza RLE .

9.1 Maximum Likelihood Estimation

Dato un processo aleatorio a tempo discreto X costituito da una sequenza di variabili aleatorie X_1, X_2, \dots, X_n , sia x_1, x_2, \dots, x_n una realizzazione di questo processo, ogni variabile aleatoria X_i si suppone avere una pdf parametrica pdf $(x|\Theta)$ con $\Theta = \theta_1, \theta_2, \dots, \theta_m$, si definisce MLE [16] *Maximum Likelihood Estimator* il vettore Θ^* che massimizza la probabilità che si verifichi la realizzazione x_1, x_2, \dots, x_n .

Un algoritmo MLE può quindi essere definito come un procedimento che, a partire da x_1, x_2, \dots, x_n e conoscendo pdf $(x|\Theta)$ trovi Θ^* . La pdf parametrica in questione può essere vista come una funzione di Θ con x fissato, chiamiamo questa funzione l_x (*likelihood function*):

$$l_x(\Theta, x_1 \dots x_n) = \text{pdf} (x_1 \dots x_n | \Theta) \quad (9.1)$$

Scopo di un algoritmo ML è quindi quello di trovare il valore Θ^* che massimizza la *likelihood function* [23] calcolata in x_1, x_2, \dots, x_n .

$$\Theta^* = \underset{\theta}{\operatorname{argmax}} l_x(\Theta, x_1 \dots x_n)$$

Se le variabili X_1, X_2, \dots, X_n sono indipendenti tra loro (X_i indipendente da $X_j \forall i, j$) allora la (9.1) può essere riscritta come:

$$l_x(\Theta, x_1 \dots x_n) = \prod_{i=1}^n l_x(\Theta, x_i)$$

e massimizzare l_x equivale a massimizzare il suo logaritmo. La *log-likelihood function* per variabili aleatorie a due a due indipendenti può scriversi come

$$L_x = \log l_x(\Theta, x_1 \dots x_n) = \sum_{i=1}^n \log l_x(\Theta, x_i)$$

che risulta più semplice da essere manipolata e computazionalmente meno impegnativa.

In molti casi la L_x è una funzione continua e differenziabile, in questo caso una condizione necessaria (non sufficiente) per la sua massimizzazione è che il suo gradiente rispetto a Θ si annulli in corrispondenza di Θ^*

$$\nabla_{\Theta} L_x = \sum_{i=1}^n \nabla_{\Theta} \log l_x(\Theta, x_i)$$

dove l'operatore ∇_{Θ} indica:

$$\nabla_{\Theta} = \begin{bmatrix} \frac{\partial}{\partial \Theta_1} \\ \frac{\partial}{\partial \Theta_2} \\ \vdots \\ \frac{\partial}{\partial \Theta_n} \end{bmatrix}$$

9.2 Codificatore adattivo con algoritmo ML

Analizziamo ora la possibilità di applicare un algoritmo ML per la stima del parametro a nel nostro caso. Abbiamo detto (equazione 7.1) che la *cdf* che approssima meglio la distribuzione cumulativa di probabilità dei simboli è:

$$cdf(x) = e^{-\frac{a}{x}}$$

Mentre la *pdf* (equazione 7.2) è:

$$pdf(x) = \left(\frac{ae^{-\frac{a}{x}}}{x^2} \right) / \left(\sum_{i=1}^{\infty} \frac{ae^{-\frac{a}{i}}}{i^2} \right)$$

quindi, assumendo che i simboli siano indipendenti (ovvero sia presente solo entropia di ordine 0), la funzione di somiglianza *likelihood function* può essere fattorizzata e, secondo quanto detto nel capitolo precedente, può essere scritta come:

$$l_x(a, x_1 \dots x_k) = \left(\prod_{i=1}^k \frac{ae^{-\frac{a}{x_i}}}{x_i^2} \right) \left(\sum_{i=1}^{\infty} \frac{ae^{-\frac{a}{i}}}{i^2} \right)^{-k}$$

Nostro compito è quello di trovare il valore di a per cui questa funzione raggiunge il suo massimo:

$$a_{ML} = \underset{\theta}{\operatorname{argmax}} l_x(\theta)$$

Dato che la funzione è definita positiva, trovare l'argomento che massimizza la funzione è equivalente a trovare l'argomento che massimizza il suo logaritmo ¹. Cerchiamo quindi di trovare per quale a la funzione (*log-likelihood function*) raggiunge il suo massimo.

$$\begin{aligned} L_x(a, x_1 \dots x_k) &= \log \left(\left(\prod_{i=1}^k \frac{ae^{-\frac{a}{x_i}}}{x_i^2} \right) \left(\sum_{i=1}^{\infty} \frac{ae^{-\frac{a}{i}}}{i^2} \right)^{-k} \right) \\ &= -k \log \left(\sum_{i=1}^{\infty} \frac{e^{-\frac{a}{i}}}{i^2} \right) - 2 \sum_{i=1}^{\infty} \log(x_i) - a \sum_{i=1}^{\infty} \frac{1}{x_i} \end{aligned}$$

Dato che la funzione è differenziabile, deriviamo rispetto ad a ed eguagliamo a zero il risultato, otteniamo.

$$\partial_a L_x(a, x_1 \dots x_k) = \frac{\delta}{\delta a} \left(-k \log \left(\sum_{i=1}^{\infty} \frac{e^{-\frac{a}{i}}}{i^2} \right) - 2 \sum_{i=1}^{\infty} \log(x_i) - a \sum_{i=1}^{\infty} \frac{1}{x_i} \right) = 0$$

che può essere riformulato come:

$$-\frac{\delta}{\delta a} \log \left(\sum_{i=1}^{\infty} \frac{e^{-\frac{a}{i}}}{i^2} \right) = \frac{1}{k} \sum_{i=1}^k \frac{1}{x_i} = H_M(x)^{-1}$$

dove $H_M(x)$ indica la media armonica dei simboli x_i .

Poniamo

¹Poiché la funzione logaritmo è monotona crescente

$$f(a) = \frac{\delta}{\delta a} \log \left(\sum_{i=1}^{\infty} \frac{e^{-\frac{a}{i}}}{i^2} \right)$$

$$f(a) = \frac{\sum_{i=1}^{\infty} \frac{e^{-\frac{a}{i}}}{i^3}}{\sum_{i=1}^{\infty} \frac{ae^{-\frac{a}{i}}}{i^2}} = H_M(x)^{-1}$$

da ciò:

$$a = f^{-1}(H_M(x)^{-1}) \quad (9.2)$$

Abbiamo quindi trovato una relazione che lega il parametro a con i simboli RLE già letti. Il problema consiste ora nel trovare quale sia l'inverso della funzione $f(\cdot)$ ovvero $f^{-1}(\cdot)$ che ci permetterebbe di ricavare il valore del parametro a a partire dalla media armonica dei simboli $H_M(x)$.

La funzione in questione non è analitica ed è molto difficile da calcolare anche in un determinato punto a . Ad esempio, con $a = 0$ la funzione risulta essere:

$$f(a)|_{a=0} = \frac{\sum_{i=1}^{\infty} \frac{1}{i^3}}{\sum_{i=1}^{\infty} \frac{1}{i^2}}$$

$$= \frac{\zeta(3)}{\zeta(2)} = \frac{6\zeta(3)}{\pi^2} = 0.7307629$$

Dove $\zeta(\cdot)$ indica la funzione ζ di Riemann [11]. A causa di questa intrattabilità matematica è stato necessario ricorrere nuovamente a metodi numerici il che ci ha permesso di trovare una funzione che approssimasse in modo soddisfacente $f^{-1}(\cdot)$ in un determinato intervallo. Si è riscontrato che un generico polinomio di secondo grado approssima in modo molto soddisfacente la funzione $f^{-1}(\cdot)$ per valori di a minori di 2, il che è più che sufficiente in quanto, come già ripetuto più volte precedentemente, l'intervallo di valori entro cui a può variare è $[0.5, 1.8]$.

In appendice [A] viene riportato il codice `MATLAB` utilizzato per determinare i coefficienti del polinomio di secondo grado approssimante la (9.2) a partire dalla equazione (7.2) approssimata numericamente. I coefficienti trovati sono:

$$c = 6.96 \quad d = -16.4912 \quad e = 10.6186 \quad (9.3)$$

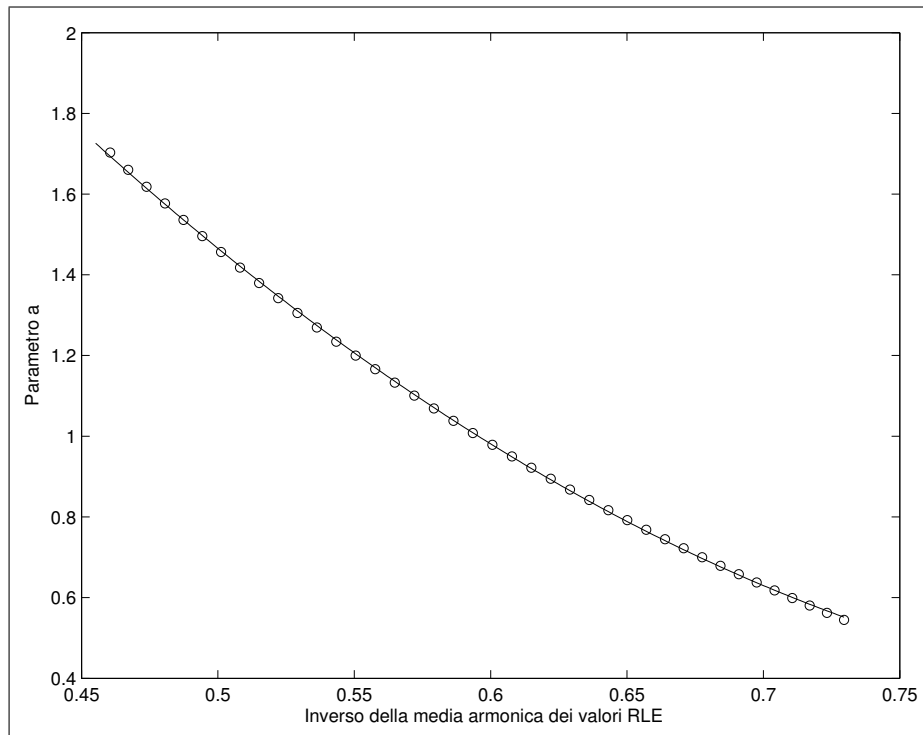


Figura 9.1: Grafico della relazione che lega a e $H_M(x)^{-1}$ e curva di secondo grado che approssima la relazione per $0.5 < a < 2$.

9.3 Dettagli implementativi

Dalla descrizione precedente possiamo passare alla seguente implementazione in pseudocodice della algoritmo adattivo:

```
sum=0;
```

Per ogni valore RLE letto, detto x tale valore:

1. viene calcolato l'inverso di x
2. l'inverso di x viene sommato a sum

che, implementato *tout-court*, significherebbe compiere due operazioni in virgola mobile per ogni intero letto.

La versione adattiva che è stata implementata compie effettivamente queste operazioni in virgola mobile poiché lo scopo dell'implementazione era solo quello di verificare l'efficienza

in termini di rapporto di compressione raggiunta dal modello adattivo e non la velocità di esecuzione.

Resta il problema di decidere dopo quanti simboli (valori RLE) eseguire il ricalcolo del parametro a .

Detto sum la somma degli inversi di x , per ricalcolare a a partire da sum è necessario:

1. calcolare l'inverso della media armonica H_M^{-1} dei simboli dividendo sum per il numero totale k di simboli letti fino al momento in cui il ricalcolo viene eseguito.
2. calcolare la (9.3):

$$c \cdot H_m^{-2} + d \cdot H_m^{-1} + e \quad (9.4)$$

Il precedente metodo richiede una divisione, due moltiplicazioni (i coefficienti c e d) e un elevamento al quadrato (per calcolare H_m^{-2}). Utilizzando la regola di Horner [31] potremmo riscrivere la (9.4) come:

$$(c \cdot H_m^{-1} + d) \cdot H_m^{-1} + e \quad (9.5)$$

in questo caso ci risparmierebbero l'elevamento al quadrato, ma i cicli macchina utilizzati per il calcolo rimarrebbero comunque molti. Risulta quindi evidente che ricalcolare a ad ogni valore RLE letto porterebbe a un grande dispendio di risorse in calcoli in virgola mobile. Oltre allo spreco di risorse che comporta, non ha nessun senso conoscere il parametro a con granularità così alta (ovvero aggiornarlo puntualmente, ad ogni dato letto) in quanto ciò non aumenta di conseguenza anche l'accuratezza con cui il codificatore apprende il modello statistico inerente alla sequenza RLE , e quindi al file, che si sta codificando.

Non è da dimenticare che un modello statistico adattivo viene costruito *nel passato* e utilizzato *nel futuro*. È quindi necessario apprendere in modo robusto il cosiddetto “passato”, ovvero i dati già letti, per potere inferire in modo abbastanza sicuro quale sarà la distribuzione dei simboli che verranno letti in futuro.

Calcolare il parametro a ad ogni iterazione non ci darebbe altro che una conoscenza accurata dei “disturbi” ai quali un modello statistico è inevitabilmente sottoposto e, cosa ancora più grave, ci farebbe utilizzare quei disturbi in modo sterile, se non dannoso, per codificare i simboli successivi con un modello statistico viziato.

D'altro canto, il parametro a deve essere ricalcolato con una certa frequenza, per permettere al codificatore di essere effettivamente adattivo, ovvero di adattare il suo modello statistico alla distribuzione dei valori RLE letti.

Da alcune prove fatte sui diversi file del *Canterbury Corpus* è emerso che si ottengono buoni risultati ricalcolando il parametro a ogni 200,000 valori RLE letti.

Un altro accorgimento da tenere in conto è quello di azzerare il contenuto di *sum* (che contiene la somma incrementale dell'inverso dei simboli letti) ogni volta che viene ricalcolato il parametro *a*. La necessità di questo accorgimento risulta presto spiegata se si intende la sequenza da codificare come costituita da gruppi di simboli consecutivi distribuiti in modo diverso (o, meglio, con parametro *a* differenti). In questo caso non avrebbe senso che sul ricalcolo del parametro *a* che verrà utilizzato per codificare i successivi 200,000 simboli influisca il comportamento statistico di simboli letti “molto tempo prima” che potevano rispondere a tutt'altra distribuzione statistica.

Quella di ricalcolare completamente le informazioni sulla statistica dei simboli letti a un certo punto del file, non tenendo più in considerazione le informazioni raccolte fino a quel punto, è una pratica diffusa, utilizzata dalla maggior parte dei compressor più noti.

Ad esempio, il noto compressore commerciale `gzip`, che implementa un algoritmo chiamato *deflation* [29] che utilizza una variante dell'algoritmo LZ77 [38] seguito da un codificatore di Huffman statico ricalcola i codici Huffman al massimo ogni 64 KB ².

Il ricalcolo del parametro *a* ogni 200,000 valori RLE fa sì che il costo computazionale ad esso associato incida in modo del tutto trascurabile sul costo computazionale totale.

L'aumento maggiore delle risorse di calcolo richieste dalla versione adattiva del codificatore viene quindi ad essere determinato solo dalle operazioni di inversione e somma, necessarie ad aggiornare l'inverso della media armonica, compiute ad ogni valore RLE letto.

È da notare comunque che le stesse operazioni potrebbero essere implementate utilizzando l'aritmetica intera apportando un grande aumento di velocità senza ledere sensibilmente la precisione dei risultati.

9.4 Decodificatore adattivo

Anche in questo caso codificatore e decodificatore rimangono perfettamente simmetrici.

Il ricalcolo del parametro *a* viene infatti eseguito nel codificatore basandosi sui valori già letti e allo stesso modo, anche il decodificatore esegue lo stesso tipo di calcolo sui dati già decodificati.

Codificatore e decodificatore si trovano quindi a decodificare gli stessi simboli utilizzando lo stesso modello statistico (in particolare lo stesso parametro *a*).

²Il ricalcolo può essere anticipato se il compressore “si accorge” che le prestazioni di compressione stanno scadendo’)

Capitolo 10

Estensioni: codificatore RLE integrato

Analizzando qualitativamente alcune sequenze RLE , come ad esempio quella generata a partire dal file *E.coli* del *Canterbury Corpus*, si nota che compaiono con una certa frequenza run di **1** di diversa lunghezza¹. Si è quindi deciso di modificare il codificatore come segue: Fissiamo c .

1. quando, tra gli ultimi c valori letti, la lunghezza media dei run di **1** supera il valore di 2, allora il codificatore entra “in modalità RLE ” e inizia a trattare diversamente gli **1** che incontra, in particolare:
2. Ad ogni **1** incontrato, il codificatore continua a leggere finchè non incontra un valore diverso da **1** .
3. il codificatore emette la codifica di **1** seguita dalla codifica del numero di **1** appena letti (la lunghezza del run di **1**) e infine, la codifica del simbolo letto diverso da **1** .
4. se la lunghezza media dei run di **1** tra gli ultimi c valori letti scende sotto al valore di 2, allora il codificatore torna in modalità normale.

Il decodificatore conosce anch'esso quando passare alla “modalità RLE ” poiché misura la lunghezza media dei run di **1** negli ultimi c simboli letti (anch'esso deve avere conoscenza del valore di c .)

La precedente modifica, sebbene fondata su una forte evidenza, non ha apportato sostanziali miglioramenti in quanto:

¹Questo fenomeno può essere facilmente spiegato considerando che la BWT raggruppa simboli simili a seconda della somiglianza del loro contesto. Se i simboli di un file hanno contesti molto corti (ovvero sono quasi indipendenti gli uni dagli altri, come potrebbe accadere in *random.txt* o *E.Coli*, la BWT del file non avrà run di simboli uguali come di solito accade e così in *output* si avranno run di **1** che indicano il succedersi nella BWT di simboli diversi

- I run di **1** appaiono in modo abbastanza sporadico ed in parti molto localizzate delle sequenze RLE
- Le lunghezze dei run di **1** vengono codificate utilizzando il modello statistico utilizzato per codificare i valori RLE . Il modello non è appropriato per le lunghezze dei run di **1** in quanto probabilmente esse hanno una diversa distribuzione statistica.

In un eventuale proseguio del seguente lavoro sarebbe auspicabile provare a codificare le lunghezza di run di **1** con un altro modello statistico.

10.1 Risultati comparazione con γ coding

I risultati in termini di *bps* ottenuti dalla versione adattiva del codificatore vengono riportati nella tabella di seguito.

File	codificatore a <i>range coder</i>	codificatore a <i>range coder</i> (versione adattiva)	% miglioramento
E.coli	2.1017	2.0758	-1.24%
alice29.txt	2.3247	2.3272	0.1%
asyoulik.txt	2.5875	2.5873	0%
bible.txt	1.5901	1.5903	0.01%
cp.html	2.6465	2.6543	0.29%
fields.c	2.4186	2.4186	0%
grammar.lsp	2.7648	2.7648	0%
kennedy.xls	1.3998	1.3968	-0.21%
lcet10.txt	2.0650	2.0684	0.16%
plrabn12.txt	2.4277	2.4269	-0.03%
ptt5	.7582	.7580	-0.02%
sum	2.8792	2.8698	-0.32%
world192.txt	1.4540	1.4550	0.06%
xargs.1	3.3404	3.3404	0%
random.txt	6.5210	6.4187	-1.59%

Come si vede dai dati precedenti, la versione adattiva non migliora in modo significativo i rapporti di compressione rispetto alla versione non adattiva sulla maggior parte dei file.

Fanno eccezione i file *random.txt* ed *E.coli* per i quali i miglioramenti raggiungono rispettivamente gli 1.59 e 1,24 punti percentuali.

La spiegazione di questo esiguo miglioramento potrebbe essere trovata nel fatto che la prima ipotesi su cui si basa il procedimento di approssimazione ML che abbiamo considerato nella sezione 9.2 è quella che suppone i valori RLE tra loro statisticamente indipendenti.

È possibile che questa indipendenza, come già detto, sussista con probabilità non molto alta nella maggior parte dei file, ma potrebbe sussistere con alta probabilità in file completamente casuali come *random.txt* o comunque molto poco deterministici, come *E.coli*. I motivi dello scarso miglioramento, al di là della mancanza di indipendenza statistica tra valori RLE, potrebbero essere ricercati anche nelle approssimazioni di calcolo accumulate prima nell'operazione di *fitting* della distribuzione con la funzione (7.2) poi con l'approssimazione della funzione con cui calcolare il parametro a con un polinomio di secondo grado.

Un altro motivo importante potrebbe risiedere nel fatto che l'algoritmo si adatta ai dati troppo lentamente e continua a comprimere nuovi valori RLE utilizzando un parametro a calcolato per i dati precedenti e ormai molto diverso da quello effettivo di quel momento. Si potrebbe tentare di ovviare a questo problema modificando il metodo di ricalcolo del parametro a che nella versione presa in esame viene ricalcolato ogni 200,000 valori, anche se alcuni tentativi di cambiare il numero di valori dopo il quale effettuare il ricalcolo di a non ha mostrato apportare significativi miglioramenti.

10.2 Comparazione con aritmetico standard

Abbiamo visto che il codificatore preso in esame che utilizza il *range coder* con il modello statistico *ad hoc* descritto nelle sezioni precedenti supera come performance il γ coding e raggiunge performance molto migliori rispetto all'aritmetico adattivo descritto in sez. 4.

Oltre alle migliorate performance in termini di rapporto di compressione, il codificatore proposto supera l'aritmetico standard anche in termini di velocità di esecuzione. La differenza sta non tanto nel codificatore (che è molto simile a quello aritmetico, anche se si per sé più veloce) quanto più al metodo di aggiornamento del modello statistico.

Nell'aritmetico adattivo standard, infatti, ad ogni simbolo letto viene aggiornata una tabella di probabilità cumulativa dei simboli, operazione che utilizzando i metodi più sofisticati oggi conosciuti, impiega asintoticamente un tempo proporzionale a [22] $\log(1 + s)$ dove s è la posizione nell'alfabeto del simbolo aggiornato e richiede uno spazio di memoria proporzionale al numero totale di simboli nell'alfabeto, mentre l'aggiornamento del modello

statistico come descritto in sez. (9.2) richiede un tempo costante che corrisponde al tempo necessario per svolgere una divisione tra interi e una somma e pochi byte di memoria. Il metodo di aggiornameto del modello statistico del codificatore descritto si dimostra quindi essere asintoticamente più efficiente di quello generico per il codificatore aritmetico standard.

Appendice A

Codice MATLAB

Di seguito è riportato il codice MATLAB utilizzato per trovare i coefficienti del polinomio di secondo grado utilizzato come approssimante della funzione (9.2).

```
pdf = inline('(xdata.^-2).*exp(-x(1)*xdata.^(-1))','x','xdata');  
cdf = inline('exp(-x(1)*xdata.^(-1))','x','xdata');
```

```
%calcola in inv l'inverso dei primi 10000 naturali  
inv=1:10000;  
inv=1./inv;
```

```
%calcola a e l'inverso della media armonica dei simboli per 1000  
%distribuzioni diverse  
for i=1:1000;  
% genera una distribuzione di probabilità con parametro i/100  
yo=pdf(i/100,1:10000);  
yo=yo/sum(yo);
```

```
% calcola l'inverso della media armonica dei simboli per la pdf creata  
cp(i)=yo(1:10000)*inv(1:10000)';
```

```
% calcola il parametro a per la dist. cumulativa corrispondente alla  
% pdf creata  
yo=cumsum(yo);  
[a(i),rn,r] = lsqcurvefit(cdf,[1],1:1000, yo(1:1000));  
end;
```

```
%trova i coefficienti per il polinomio di secondo grado che meglio
%approssima la relazione tra il parametro a e la media armonica dei simboli.
my_opt=optimset('lsqcurvefit');
my_opt.MaxIter = 5000;
my_opt.MaxFunEvals = 100000;
fun2 = inline('x(1)+x(2)*xdata+x(3)*xdata.^2','x','xdata');
[xa,rn,r] = lsqcurvefit(fun2,[1 1 1],cp(1:200),a(1:200),[],[],my_opt);
```

Appendice B

Codificatore a *range coder*.

Di seguito è riportata l'implementazione C del codificatore basato su *range coder* nella sua versione adattiva. Come si può vedere dal codice, sia l'aggiornamento adattivo che la "modalità RLE " possono essere indipendentemente attivati utilizzando rispettivamente il flag di compilazione ADAPTIVE e ENABLE_RLE.

Il seguenti listati la definizione di alcune funzioni comune al codificatore e al decodificatore

```
//valore del massimo simbolo codificabile con una sola cifra
#define MaxC 250
#define MaxC2 (MaxC*MaxC)
//numero di cifre di shift per range encoder
#define Shift 19

//numero di simboli dopo il quale viene ricalcolato il parametro a per
//il modello di simboli esponenziali
#define cRate 10000

//numero di simboli dopo il quale viene rifelezionato il modello
//statistico di compressione (tra EXP e RLE)
#define cRate1 100

//valore iniziale del parametro a del modello esponenziale
#define Aini 0.88

//valori dei coefficienti della funzione di secondo grado che
//approssima la relazione che lega tra l' inverso della media
//armonica dei simboli letti e il parametro a ottimale per la
```

```
//distribuzione esponenziale

#define c 6.96
#define d -16.4912
#define e 10.6186

//lunghezza media dei run di 1 oltre la quale si passa al modello RLE
#define Th 2.0

//tipi di modelli statistici utilizzabili per la codifica
enum Tmodel { EXP=0 , UNIFORM, RLE } model;

//restituisce il range di frequenze entro il quale si colloca il
//simbolo ch
void getltfreq(long int ch,enum Tmodel model,double a,int* ltfreq ,
              int* ltfreq1);

//aggiorna il modello statistico con i simboli letti e ricalcola i
//parametri
void update(long int ch, enum Tmodel* model,double* a);

#include <stdio.h>
#include <stdlib.h>
#ifdef unix
#include <io.h>
#include <fcntl.h>
#endif
#include <string.h>
#include <ctype.h>
#include "port.h"
#include "rangecod.h"
#include <math.h>
#include "common.h"

/*
restituisce il range di frequenze entro il quale si colloca il
simbolo
```

```

ch è il simbolo da codificare
model è il modello statistico da utilizzare per comprimere
a è il coefficiente da utilizzare nella compressione con modelli
ltfreq è il puntatore alla frequenza cumulativa dei simboli
    minori del carattere ch
ltfreq1 è il puntatore alla frequenza cumulativa dei simboli
    minori od uguali al carattere ch
*/
void getltfreq(long int ch,enum Tmodel model,double a,int* ltfreq ,
              int* ltfreq1)
{
    if(model==EXP || model==RLE)
        {
            //se il modello è esponenziale o RLE calcola le frequenze
            //secondo la:
            //PDF(x) = exp(-a/x);
            *ltfreq1=((1<<Shift)*( exp(-a/(double)(ch+1)) ));
            *ltfreq=(ch?((1<<Shift)*(exp(-a/(double)ch))):0);
        }
    else
        {
            //se il modello è uniforme calcola le frequenze /secondo la:
            //PDF(x) = 1;
            *ltfreq=(((1<<Shift)/(double)MaxC)*(double)ch);
            *ltfreq1=(((1<<Shift)/(double)MaxC)*(ch+1.0));
        }

    (*ltfreq)++;
}

/*
aggiorna il modello statistico con i simboli letti e ricalcola i
parametri
*/
void update(long int ch , enum Tmodel* model,double* a)
{
    static double sum_inv=0;
    static long int tot_symb=0;

```

```

static long int num1=0;
static long int line1=1;
static int prev=0;
double x;

//se è abilitata a compilazione la compressione con RLE calcola
//incrementalmente la lunghezza media dei run di uno
#ifdef ENABLE_RLE
if (ch)
{
if (prev) {prev=0;line1++;}
}
else
{
prev=1;
num1++;
}

//se la lunghezza media dei run di uno è superiore alla soglia Th
//passa al modello RLE, viceversa passa all'EXP
fprintf(stderr, "%f_\n", (((double)num1/((double)line1))));
if (((double)num1/((double)line1)>Th)
(*model)=RLE;
else
(*model)=EXP;
#endif

//se è abilitata a compilazione la compressione adattiva calcola
//incrementalmente la somma degli inversi dei simboli in ingresso
//dalla quale sarà poi possibile risalire al parametro a ottimale
//per la distribuzione
//PDF(x) = exp(-a/x);
#ifdef ADAPTIVE
tot_symb++;
sum_inv+=(1.0/(ch+1.0));

//ogni cRate1 simboli azzera la media dei run di 1
if (!(tot_symb%cRate1))

```

```

    {
        prev=num1=0;
        line1=1;
    }

//ogni cRate simboli ricalcola il parametro a ottimale applicando
//una specifica funzione quadratica all'inverso della media armonica
//dei simboli e azzera la somma degli inversi calcolata
//incrementalmente
if (!(tot_symb%cRate))
    {
        x=sum_inv/((double)(cRate));
        (*a)=c+d*x+e*x*x;
        sum_inv=0;
    }
#endif
}

```

Il seguente listato contiene la definizione delle funzioni relative appartenenti al codificatore.

```

/*
Compressore rangecoder con modello quasi-statico. Luca Foschini

Per rendere il compressore/decompressore adattivo, aggiungere il
flag -DADAPTIVE alle opzioni di compilazione

Per attivare l'ottimizzazione RLE, aggiungere il flag -DENABLE_RLE
alle opzioni di compilazione

Il codificatore in questione codifica simboli da 1 a MaxC utilizzando
il range encoder e basandosi su un modello statistico di distribuzione
dei simboli studiato ad-hoc del tipo  $\text{cdf}(x)=\exp(-a/x)$ 
i simboli maggiori di MaxC vengono codificati in base MaxC e
preceduti da tanti simboli di escape quanti sono le cifre
dell'intero in base MaxC che rappresenta il simbolo meno uno.
Il codificatore riesce a gestire simboli compresi tra 1 e  $\text{MaxC}^3-1$ .
*/

#include <stdio.h>

```

```

#include <stdlib.h>
#ifdef unix
#include <io.h>
#include <fcntl.h>
#endif
#include <string.h>
#include <ctype.h>
#include "port.h"
#include "rangecod.h"
#include <math.h>
#include "common.h"

/**
Codifica un simbolo secondo un determinato modello statistico
range coder è l'oggetto range coder sul quale eseguire la codifica
ch è simbolo da codificare
    model è il modello statistico da utilizzare per comprimere
a è il coefficiente da utilizzare nella compressione con modelli
    RLE o EXP
*/
inline void encode(range coder* rc, long int ch, enum Tmodel model,
                  double a)
{
    long int v[3];
    int ltfreq, ltfreq1, i, j;
    enum Tmodel m1;

    //il simbolo ch viene scritto "in basa MaxC" su tre cifre per questo
    //motivo ch deve essere minore od uguale a MaxC^3
    //V[0] contiene la cifra più significativa
    v[0]= ch/MaxC2; ch%=MaxC2;
    v[1]= ch/MaxC; ch%=MaxC;
    v[2]= ch%MaxC;

    //codifica tanti simboli di escape quante sono le cifre del numero
    //in base MaxC che deve essere codificato meno uno, quindi 1 escape
    //per 2 cifre e 2 escape per 3 cifre
    getltfreq(MaxC, model, a, &ltfreq, &ltfreq1);

```

```

j=0;while(!v[j]) j++;
if (j>2) j=2;
for(i=j;i<2;i++) encode_shift(rc,(1<<Shift)-ltfreq ,ltfreq , Shift);

//codifica le cifre in base MaxC
m1=model;
for(i=j;i<3;i++)
{
    //calcola il range nel quale deve essere codificato il simbolo
    getltfreq(v[i],model,a,&ltfreq,&ltfreq1);
    //codifica il simbolo utilizzando la codifica a shift
    //(v. rangecoder)
    encode_shift(rc ,ltfreq1-ltfreq ,ltfreq , Shift);
    //solo la cifra più significativa viene codificata il modello
    //esponenziale. Le eventuali altre due cifre vengono codificare
    //utilizzando un modello statistico uniforme.
    model=UNIFORM;
}
model=m1;
}

/*
Codifica il segnale di fine file, ovvero codifica 3 volte
consecutive il simbolo MaxC
rangecoder è l' oggetto rangecoder sul quale eseguire la codifica
one è il numero di uno da codificare nel caso si stia
utilizzando il modello RLE e sia rimasto un run di uno da codificare
a è il coefficiente da utilizzare nella compressione con modelli
RLE o EXP
*/
inline void eof(rangecoder* rc,long int one,double*a)
{
    int ltfreq ,ltfreq1 ,i;
    enum Tmodel model;

    //se il modello che si sta utilizzando per codificare è RLE e è

```

```

//rimasto ancora un run di uno da codificare, viene codificato ora.
if (one)
{
    encode(rc,0,EXP,*a);update(0,&model,a);
    encode(rc,--one,EXP,*a);update(one,&model,a);
    one=0;
}

//vengono codificate 3 caratteri di escape consecutivi
getltfreq(MaxC,EXP,*a,&ltfreq,&ltfreq1);
for(i=0;i<3;i++) encode_shift(rc,(1<<Shift)-ltfreq,ltfreq,Shift);
}

int main( int argc, char *argv[] )
{
    long int ch,one=0;
    double a=Aini;
    enum Tmodel model=EXP;
    rangecoder rc;

    //inizializza l'encoder
    start_encoding(&rc,0,0);

    while (scanf("%ld",&ch)!=EOF)
    {
#ifdef TEST
        if (!ch) continue;
#endif
        --ch;
        //se il modello che si sta utilizzando per comprimere è di tipo
        //RLE allora i run di uno vengono compressi in RLE
        if (model==RLE)
        {
            if (!(ch))
            {
                one++;
                continue;
            }
        }
    }
}

```

```

    }
    else
    {
        if(one)
        {
            //codifica uno zero per segnalare al decompressore
            //che si sta codificando la lunghezza di un run e
            //non un simbolo e aggiorna
            encode(&rc,0,model,a);update(0,&model,&a);
            //codifica la lunghezza del run e aggiorna
            encode(&rc,--one,model,a);update(one,&model,&a);
            one=0;
        }
    }
    //codifica il simbolo
    encode(&rc,ch,model,a);update(ch,&model,&a);
}

eof(&rc,one,&a);
//scrive su stderr il numero di bit scritti
fprintf(stderr,"%d_\n",done_encoding(&rc)*8);
return 0;
}

```

Il seguente listato contiene la definizione delle funzioni relative appartenenti al codificatore.

```

/*
Decompressore rangecoder con modello quasistatico.
Luca Foschini

```

Per rendere il compressore/decompressore adattivo, aggiungere il flag -DADAPTIVE alle opzioni di compilazione

Per attivare l'ottimizzazione RLE, aggiungere il flag -DENABLE_RLE alle opzioni di compilazione

Il codificatore in questione codifica simboli da 1 a MaxC utilizzando il range encoder e basandosi su un modello statistico di distribuzione

dei simboli studiato ad hoc del tipo $\text{cdf}(x)=\exp(-a/x)$
 i simboli maggiori di MaxC vengono codificati in base MaxC e
 preceduti da tanti simboli di escape quanti sono le cifre
 dell'intero in base MaxC che rappresenta il simbolo meno uno.
 Il codificatore riesce a gestire simboli compresi tra 1 e MaxC^3-1 .

```

*/

#include <stdio.h>
#include <stdlib.h>
#ifdef unix
#include <io.h>
#include <fcntl.h>
#endif
#include <string.h>
#include <ctype.h>
#include "port.h"
#include "rangecod.h"
#include <math.h>
#include "common.h"

/*
Restituisce il carattere corrispondente a una certa frequenza del
range coder
ltfreq è la frequenza cumulativa dei simboli minori del carattere
model è il modello statistico da utilizzare per comprimere
a è il coefficiente da utilizzare nella compressione con modelli
ch è il puntatore al simbolo restituito
*/
inline void getCh(int ltfreq,enum Tmodel model,double a,long int*ch)
{
  if(model==EXP || model==RLE)
    *ch = (ltfreq)?(a/(Shift*log(2.0)-log(ltfreq))):0;
  else
    *ch = (MaxC*ltfreq)/(1<<Shift);
}

int main( int argc, char *argv[] )
{

```

```

long int ch,ch1;
int ltfreq ,ltfreq1 , i ,RLEFlag=0,lCount=0;
double a=Aini;
enum Tmodel model=EXP,m1;
rangecoder rc;

//inizializza il decoder
start_decoding(&rc);

while (1)
{
    //se è stato letto il mark di fine file
    if (lCount==3) break;

    //decofica la frequenza successiva
    ltfreq = decode_culshift(&rc,Shift);
    //associa un carattere alla frequenza decodificata
    getCh(ltfreq ,model,a,&ch);

    //se è stato incontrato un escape
    if (ch>=MaxC)
    {
        getltfreq(MaxC,model,a,&ltfreq,&ltfreq1);
        ltfreq1=1<<Shift;
    }
    else
        getltfreq(ch,model,a,&ltfreq,&ltfreq1);

    //aggiornamento del decoder
    decode_update( &rc , ltfreq1-ltfreq , ltfreq , 1<<Shift );

    //se il modello che sis sta utilizzando è RLE ed è stato letto
    //uno zero
    if (model==RLE && !RLEFlag && !ch)
    {
        update(0,&model,&a);
        RLEFlag=1;
        continue;
    }
}

```

```

    }

    //se è stato letto un escape
    if (ch>=MaxC) {lCount++;continue;}

    //esegue la decodifica di un simbolo
    m1=model;
    model=UNIFORM;
    for (i=0;i<lCount;i++)
    {
        ltfreq = decode_culshift(&rc, Shift);
        getCh(ltfreq, model, a, &ch1);

        //ricostruisce il simbolo codificato in base MaxC
        ch=ch*MaxC+ch1;
        getltfreq(ch1, model, a, &ltfreq, &ltfreq1);
        decode_update( &rc, ltfreq1-ltfreq, ltfreq, 1<<Shift);
    }
    model=m1;
    lCount=0;

    if (RLEFlag)
    {
        //se è stato decodificato la lunghezza di un run di uno
        //allora scrive il run
        for (i=0;i<=ch;i++) printf("1_");
        RLEFlag=0;
    }
    //altrimenti scrive un simbolo
    else printf("%ld_", ch+1);

    //aggiorna il modello
    update(ch, &model, &a);
}
return 0;
}

```

Bibliografia

- [1] A. Aho, J. Hopcroft, and J. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading MA, 1983.
- [2] Timothy C. Bell, John G. Cleary, Ian H. Witten, *Text Compression*, Prentice Hall, Englewood NJ, 1990
- [3] M. Burrows, D.J. Wheeler, *A block-sorting lossless data compression algorithm* Digital Equipment Corporation (SRC Research Report 124), 1994
<ftp://ftp.digital.com/pub/DEC/SRC/research-reports/SRC-124.ps.zip>
- [4] M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, New York, 1994.
- [5] Peter Fenwick *A new data structure for cumulative frequency tables* , Software Practice and Experience, Vol 24, No 3, pp 327 336 Mar 1994.
- [6] Thomas S. Ferguson, Christian Genest and Marc Hallin *Kendall s tau for Serial Dependence* . The Canadian Journal of Statistics Vol. 28, 2000,
- [7] Paolo Ferragina, Giovanni Manzini *An experimental study of an opportunistic index*. Symposium on Discrete Algorithms, 2001.
- [8] E. Fredkin. *Trie memory*. Communications of the ACM, 3:490-499, 1962.
- [9] G. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures*. Addison-Wesley, Wokingham, England, second edition, 1991.
- [10] C. Gourieroux, A. Monfort *Statistics and econometric models, Vol 2*. Cambridge University press. Pagg. 111-113.
- [11] Graham,Knuth,Patashnik *Matematica Discreta*, Ulrico Hoepli Editore S.p.A. 1992. Pagg. 257-258

- [12] R. Grossi, A. Gupta, J. S. Vitter. High-order entropy-compressed text indexes. SODA 2003.
- [13] R. Grossi, A. Gupta, J. S. Vitter. When Indexing Equals Compression: Experiments with Compressing Suffix Arrays and Applications SODA 2004.
- [14] D. Gusfield. *String Algorithms*. Cambridge University Press, 1997.
- [15] David A. Huffman *A Method for the Construction of Minimum-Redundancy codes* Proceedings of the I.R.E, September 1952
- [16] Steven M. Kay *Fundamentals of Statistical Signal Processing: Estimation Theory* Prentice Hall PTR, Prentice Hall, 1993. Pag 254
- [17] Marco Luise, Giorgio M. Vitetta *Teoria dei segnali*, McGraw Hill, 1999. Pagg. 191-194.
- [18] Marco Luise, Giorgio M. Vitetta *Teoria dei segnali*, McGraw Hill, 1999. Pagg. 223-224.
- [19] G. N. N. Martin *Range encoding: an algorithm for removing redundancy from a digitised message* Proceedings of the Video and Data Recording Conference, IBM UK Scientific Center, Southampton July 24-27 1979.
- [20] E. McCreight. *A space-economical suffix tree construction algorithm*. J. ACM, 23:262-272, 1976.
- [21] Alistair Moffat, Radford M. Neal and Ian H. Witten *Arithmetic Coding Revisited* Proc. IEEE Data Compression Conference, Snowbird, Utah, March 1995).
- [22] Alistair Moffat *An Improved Data Structure for Cumulative Probability Tables* Software-Practice and Experience, 29(7):647-659, 1999.
- [23] Todd K. Moon *The Expectation-Maximization Algorithm* IEEE Signal Processing Magazine, November 1996. Pag 51, box 1.
- [24] M. Nelson. Run length encoding/RLE. DataCompression.info, <http://www.datacompression.info/RLE.shtml>.
- [25] A. Papoulis *Probabilità, variabili aleatorie e processi stocastici*, 1973 Editore Boringhieri. Pagg.232-233
- [26] Kuniyiko Sadakane *Compressed Text Databases with Efficient Query Algorithms based on the Compressed Suffix Array*. Proceedings of ISAAC'00, 2000.

- [27] C. E. Shannon *A Mathematical Theory of Communication* Reprinted with corrections from The Bell System Technical Journal, Vol. 27, pp. 379 423, 623 656, July, October, 1948.
- [28] David Salomon *Data compression, the complete reference, 2nd edition*. Springer, 2000. Pag 44.
- [29] David Salomon *Data compression, the complete reference, 2nd edition*. Springer, 2000. Pag 205.
- [30] Michael Schindler www.compressconsult.com/
- [31] R. Sedgewick *Algoritmi in C*. Addison Wesley Masson, 1993. Pag 545.
- [32] Steven S. Skiena, *The Algorithm Design Manual*. Springer-Verlag, New York 1997
- [33] E. Ukkonen. *Constructing suffix trees on-line in linear time*. In Intern. Federation of Information Processing (IFIP '92), pages 484-492, 1992.
- [34] P. Weiner. *Linear pattern-matching algorithms*. In Proc. 14th IEEE Symp. on Switching and Automata Theory, pages 1-11, 1973.
- [35] Hugh E. Williams Justin Zobel *Compressing integers for fast file access*
- [36] I.H. Witten, A. Moffat, T. C. Bell *Managing Gigabytes, 2nd edition* Morgan Kaufmann Publishers, 1999
- [37] John R. Wolberg *Prediction Analysis*, D. Van Nostrand Company, Inc, 1961. Pagg. 28-34
- [38] J. Ziv, A. Lempel (1977) *A universal algorithm for sequential data compression* IEEE Transaction on Information Theory, IT-23(3):337-343
- [39] www.gnu.org/copyleft/gpl.htm
- [40] <http://sources.redhat.com/bzip2/>
- [41] <http://ccrma-www.stanford.edu/jos/mdft/Autocorrelation.html>
- [42] Managing Gigabytes for Java mg4j.dsi.unimi.it/
- [43] www.compressconsult.com/rangecoder/
- [44] The Canterbury Corpus, <http://corpus.canterbury.ac.nz>.