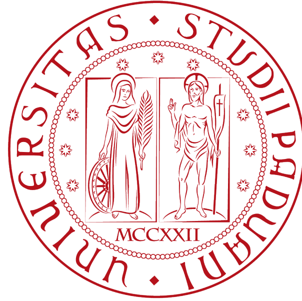

UNIVERSITA' DEGLI STUDI DI PADOVA
Dipartimento Di Ingegneria Dell'Informazione
Corso Di Laurea In Ingegneria Dell' Informazione

Confronto della comprimibilità lossless con e senza gap di database di sequenze

RELATORE: Dott.ssa Cinzia Pizzi
CORRELATORE: Dott. Fabio Cunial

LAUREANDO: Simone Brazzo

Anno Accademico 2013/2014

Sommario

Nel corso degli ultimi 20 anni il flusso e la memorizzazione di informazioni digitali è stato sempre maggiore. Questo problema ha richiesto lo sviluppo di metodi che permettessero di ridurre i costi generati da queste grandi moli di dati. Tali metodi fanno parte del ramo della teoria dell'informazione che studia la compressione dei dati. Questa disciplina ricopre un ruolo di fondamentale importanza nelle telecomunicazioni ma ha trovato un largo impiego anche nella bioinformatica per lo studio dei meccanismi biologici fondamentali. L'argomento di studio di questa tesi riguarda l'algortimo di compressione lossless LZWA. Questo algoritmo è un adattamento di LZW motivato dall'interesse per le sequenze biologiche. Esso si basa sulla presenza di sottostringhe ripetute di caratteri fissi e variabili. Nella versione lossless di LZWA tali sottostringhe vengo codificate attraverso l'uso di due puntatori. Questi puntatori rappresentano due ben definite stringhe contenute in due distinti dizionari tali che la loro fusione rappresenta la stringa di partenza. I dizionari vengono gestiti da strutture ad albero come nella versione LZW e contengono rispettivamente le informazioni sui caratteri fissi e sui caratteri variabili. L'obiettivo di questa tesi è quello di verificare i livelli a cui possono essere compressi alcuni dataset con LZWA. In particolare sono stati analizzati un dataset di dati di origine biologica e due dataset di dati riguardanti codice sorgente.

Indice

Introduzione **1**

1 Concetti di base della compressione **3**
1.1 Entropia e Ridondanza 4
1.2 Compressione Lossy e Lossless 5

2 Compressori basati su stringhe **7**
2.1 Algoritmo LZ77 7
2.1.1 Codifica 7
2.1.2 Decodifica 9
2.2 Algoritmo LZ78 10
2.2.1 Codifica 10
2.2.2 Decodifica 12
2.3 Algoritmo LZW 13
2.3.1 Codifica 13
2.3.2 Decodifica 15
2.4 Algoritmo LZMA 15
2.4.1 Codifica 15
2.4.2 Decodifica 18

3 Pattern con gap **19**
3.1 Introduzione ai pattern con gap 19
3.2 Applicazioni in ambito bioinformatico 21
3.2.1 Compressione per la memorizzazione di sequenze 21
3.2.2 Pattern Discovery 23
3.3 Compressione di immagini 24
3.4 Algoritmo LZWA 30
3.4.1 Codifica 31

3.4.2 Decodifica . . . 32
3.4.3 Aspetti strutturali . . . 33
3.4.4 Modifiche e miglioramenti . . . 34
3.4.5 Dinamica di Pattern con gap e Resolver . . . 35

4 Esperimenti **41**
4.1 Obiettivi e setting degli esperimenti . . . 41
4.2 Risultati su sequenze di RNA . . . 43
4.3 Risultati su Linux Kernel . . . 46
4.4 Risultati su file XML . . . 49

Conclusioni **55**

Bibliografia **57**

Introduzione

La disciplina della compressione dati è di fondamentale importanza nell'ambito delle telecomunicazioni ma ha trovato un largo impiego anche nel campo della bioinformatica. Sequenze come Rna e Dna ad esempio contengo informazioni che si ripetono e che possono rappresentare interazioni biologiche fondamentali. In generale i dati ripetuti all'interno di una sequenza qualsiasi trovano un stretta correlazione con i termini ridondanza ed entropia che sono alla base dei meccanismi di compressione.
I principali algoritmi di compressione dati trovano le loro fondamenta nei classici LZ77 [1] e LZ78 [2]. Tra i vari algoritmi che si sono sviluppati da questi ultimi di particolare importanza per questa tesi risulta l'algoritmo LZW [3]. Questo algoritmo si basa sull'utilizzo di un dizionario per l'analisi e la memorizzazione del stinghe nella codifica dei dati. Tali stringhe vengono codificate mediante puntatori numerici che rappresentano elementi di un dizionario. Un contributo di questa tesi è stata l'implementazione in linguaggio c++ di un algoritmo lossless basato sul funzionamento di LZW chiamato LZWA [4]. Tale algoritmo si basa sulla definizione di pattern con gap. I pattern con gap sono particolari tipi di pattern basati su sottostringhe di caratteri solidi e variabili che si possono ripetere più volte all'interno di una certa sequenza dati. La parte di codifica si basa sull'utilizzo di due dizionari costruiti su strutture dati di tipo tries. Questi dizionari rappresentato un insieme di pattern che vengono trovati e memorizzati dall'algoritmo. Nella parte di codifica viene ricercata sempre la stringa più lunga all'interno dei dizionari, questa stringa viene poi suddivisa in una stringa con gap e in una stringa di caratteri solidi. La codifica di ogni pattern è rappresentata da due puntatori ai dizionari utilizzati, che corrispondono alle rispettive stringhe con gap e solide. I pattern con gap vengono quindi ricostruiti in decodifica tramite la fusione delle stringhe selezionate da tali puntatori, ogni gap nella stringa di caratteri variabili viene sostituito in modo sequenziale carattere per carattere dalla stringa solida.

L'obiettivo principale di questa tesi è quello di analizzare il comportamento di questo algoritmo per verificare se possa essere vantaggioso, rispetto ad algoritmi classici, nella compressione di sequenze di dati rappresentanti RNA, Kernel Linux e un sorgente XML. Il confronto dei risultati viene fatto con i classici GZip [5] e 7Zip [6]. La tesi si articola nel seguente modo. Nel capitolo 1 vengono descritti i principali concetti della teoria dell'informazione su cui si basano gli algoritmi di compressione dati. Nel capitolo 2 vengono descritti i classici algoritmi basati su stringhe LZ77, LZ78, LZW, LZMA [6]. Nel capitolo 3 vengono invece introdotti i concetti di pattern con gap, alcuni ambiti applicativi, e la descrizione dettagliata dell'algoritmo LZWA. Nel capitolo 4 infine vengono illustrati e discussi i risultati di compressione in 3 diversi tipi di dataset.

1 Concetti di base della compressione

La comprensione dei meccanismi e dei codici che vengono utilizzati negli algoritmi di compressione parte dal concetto di informazione. La teoria dell'informazione venne introdotta da Shannon alla fine degli anni '40 [7]. I suoi studi portarono alla definizione di quantità di informazione associata ad un simbolo (o "self-information"). Vennero collegati i concetti di funzione logaritmica e di informazione e si dimostrò che l'informazione (in bits) contenuta in un simbolo x con probabilità $P(x)$ è:

$$i(x) = log_2(\frac{1}{P(x)}) = -log_2(P(x))$$

La teoria dell'informazione viene applicata principalmente nella trasmissione di informazioni da una sorgente a un ricevitore attraverso un canale. L'informazione è trasmessa in unità chiamate simboli (solitamente bit), e l'insieme di tutti i simboli formano l'alfabeto utilizzato. Il più importante elemento che influisce nella comunicazione attraverso un canale è il rumore. Nella comunicazione tra macchine il rumore può essere causato da diversi fattori come ad esempio imperfezione nella strumentazione hardware, campi elettromagnetici, variazioni di voltaggio, cavi ad alta resistenza. La presenza di rumore ha portato allo sviluppo di speciali codici il cui compito è quello di aumentare la probabilità che l'informazione arrivi a destinazione integra. Ci si riferisce a questi codici con il nome di codici controllo, che interessano appunto la codifica di canale. Un'altra caratteristica importante che influisce nella comunicazione è l'ammontare di volume che può essere trasferito all'interno del canale. Ogni canale di comunicazione possiede una capacità limitata, esso può trasmettere solo un limitato numero di simboli per unità di tempo. Una tecnica per aumentare il volume di dati che può essere spedito all'interno del

canale è quella di comprimere i dati prima di spedirli. I metodi di compressione dati conosciuti sono anche noti come metodi di codifica di sorgente. La caratteristica che rende possibile la compressione dati è il fatto che singoli simboli appaiono nel file dati con differenti probabilità. I dati possono essere compressi assegnando codici di lunghezza variabile a singoli simboli così che questi li vadano a sostituire. Vengono adesso illustrati i concetti fondamentali di entropia e ridondanza, necessari per lo sviluppo della compressione dati [8].

1.1 Entropia e Ridondanza

Il termine entropia definito da Shannon sta ad indicare quanta informazione porta un evento. Data un variabile discreta X che può avere n valori x_i con probabilità p_i, l'entropia $H(x)$ di X è definita come:

$$H(x) = -\sum_{i=1}^{n} p_i \cdot log_2(p_i)$$

La scelta della funzione logaritmo è stata fatta perchè soddisfa le seguenti caratteristiche:

- E' una funzione decrescente di p_i, e restituisce 0 quando la probabilità di un simbolo è 1. Questo rispecchia il fatto che un simbolo con alta probabilità porta con se poca informazione.

- Supponendo che il canale sia senza memoria ed i simboli siano indipendenti, $log(s_1 s_2) = log(s_1) + log(s_2)$, quindi scoprire che s_1 è immediatamente seguito da s_2 fornisce la stessa informazione che s_1 e s_2 vengano spediti indipendentemente l'uno dall'altro.

Con il termine ridondanza vengono identificati quell'insieme di dati che non contengono sostalziamente informazioni aggiuntive. In un messaggio o in un testo la presenza di elementi che non contengono novità informative sta ad indicare che la rimozione di tali dati non cambia il contenuto di partenza portando a compressione. Quando l'entropia è massima i dati contengono la massima informazione possibile, quindi non possono essere ulteriormente compressi. Così si definisce ridondanza una quantità che tende a zero quando l'entropia tende al suo valore massimo. Consideriamo un alfabeto Σ di simboli σ_i dove ogni simbolo appare nei dati con probabilità p_i. I dati vengono compressi rimpiazzando ogni simbolo con codici di lunghezza l_i bits. La lunghezza media di un codice è data da

$$\sum_i p_i \cdot l_i$$

e la ridondanza (il più piccolo numero di bits richiesto per rappresentare il simbolo) è:

$$R = \sum_i p_i \cdot l_i - \sum_i [-p_i \cdot log_2(p_i)]$$

La ridondanza è zero quando la lunghezza media dei codici è uguale all'entropia, questo accade ad esempio quando i codici sono i più piccoli possibili e la compressione è massima. Analogamente possiamo definire l'entropia di un singolo simbolo come $-p_i \cdot log_2(p_i)$, questo è anche il più piccolo numero di bits necessari (in media) per rappresentare il simbolo. L'entropia dei dati dipende dalle probabilità individuali p_i ed è la più grande possibile quando tutti i simboli dell'alfabeto sono equiprobabili. Data una stringa di caratteri in input è facile calcolare la probabilità di ogni suo singolo carattere. Essa può essere calcolata contando il numero di volte che il carattere si ripete nella stringa, diviso la lunghezza di tale stringa. Il principale teorema provato da Shannon afferma semplicemente che un messaggio di n simboli può essere in media compresso fino a $n \times H$ simboli. La compressione dati è sicuramente uno degli elementi che ha permesso lo sviluppo delle tecnologie multimediali negli ultimi '30 anni. Alcuni esempi dove l'impiego della compressione è stato di fondamentale importanza sono la tv digitale, la comunicazione cellulare, internet.

1.2 Compressione Lossy e Lossless

I metodi di compressione possono essere suddivisi in due grandi famiglie: i metodi lossless e lossy. I primi algoritmi permettono la ricostruzione totale dei dati in input mentre i secondi forniscono una compressione maggiore ma non permettono la ricostruzione completa dei dati. La compressione lossless viene generalmente utilizzata nelle applicazioni che non tollerano differenze tra i dati originali e quelli ricostruiti. La compressione testuale è un importante ambito applicativo della compressione lossless. In un testo infatti l'eventuale presenza di errori dovuti alla fase di ricostruzione può indurre all'incomprensione del testo, quindi è di fondamentale importanza che venga mantenuta l'integrita dei dati. Viceversa nella compressione lossy vengono perse delle informazioni, e generalmente l'intera ricostruzione dei dati non può avvenire. Accettando di perdere parte delle informazioni il tasso di compressione nel metodo lossy è generalmente più elevato rispetto a quello della compressione lossless. Molte applicazioni non richiedono comunque la ricostruzione completa dei dati in input, come ad esempio la telefonia. I principali parametri

indicanti la qualità di un algoritmo di compressione dati sono due: l'ammontare di dati che vengono compressi e la percentuale di ricostruzione dell'input. Il primo parametro viene anche chiamato tasso di compressione ed è definito come:

$$compression\ ratio = \frac{\text{dimensione file input}}{\text{dimensione file output}}$$

Supponiamo ad esempio di avere in input un immagine di dimensione pari a 65536 byte (256×256 pixels). Se l'immagine compressa risulta essere di 16384 byte diremo che le il tasso di compressione è pari a $\frac{1}{4}$, o in percentuale che abbiamo compresso il 75% dell'immagine di partenza. Un altro modo per confrontare le performance di compresse è quello di fornire il numero medio di bit richiesti per presentare un singolo campione. Questa informazione è generalmente chiamata "rate". Per esempio nel precedente caso, se assumiamo inizialmente di avere 8 bit per pixel, il numero medio di bits nella rappresentazione compressa è 2, ovvero abbiamo un rate di 2 bit per pixel. Nella compressione lossy dove parte dell'informazione viene persa è difficile stimare i precedenti parametri. Perciò viene introdotto il concetto di distorsione che indica la differenza tra i dati originali e quelli ricostruiti. Un altro termine utilizzato quando si parla di prestazioni nella compressione e decompressione viene detto fedeltà. Se diciamo che la fedeltà del file ricostruito è alta, significa che le differenze tra il file codificato e l'originale sono quasi nulle.

2 Compressori basati su stringhe

Nell'ambito della compressione dati il concetto di dizionario è di grande importanza. Esso fornisce una struttura di memorizzazione per lo sviluppo degli stadi fondamentali della compressione: la codifica e la decodifica, svolti rispettivamente dall'encoder e dal decoder. L'insieme degli algoritmi per la compressione basati sull'uso dei dizionari è dovuto al lavoro di Jacob Ziv e Abraham Lempel, che nel 1977 e 1978 pubblicarono due documenti ([1] e [2]) che di fatto ne costruirono le basi. Successivamente vennero implementati LZ77 e LZ78, frutto del loro lavoro. Da essi si sono sviluppati una serie di algoritmi che di fatto ne contengono i principi basilari nonchè le iniziali LZ. Di seguito illustreremo gli aspetti fondamentali dei "classici" algoritmi LZ77, LZ78 e discuteremo alcuni algoritmi da loro derivati come LZW e LZMA in modo tale facilitare la comprensione del prossimo capito. Nel capitolo ci serviremo spesso di una sequenza definita su un alfabeto finito Σ di simboli σ_i.

2.1 Algoritmo LZ77

L'algoritmo LZ77, anche detto LZ1, è stato introdotto in [1]. L'idea principale dell'algoritmo è quella di comprimere sostituendo un'occorrenza di un dato ripetuto con un riferimento (chiave) ad una singola copia di tale dato che si fosse precedentemente presentata in input.

2.1.1 Codifica

L'encoder mantiene una finestra scorrevole di dimensioni variabili sui dati in input. Con l'avanzare della scansione tale finestra viene shiftata verso destra in modo tale che nuovi simboli possano essere analizzati. La finestra sui dati è

divisa in due sezioni: la parte di input non ancora scandita che viene chiamata "look-ahead buffer", e la parte già scandita e codificata, che funge da dizionario e viene chiamata "search buffer". Il search buffer contiene solo i simboli codificati recentemente, data la limitata dimensione della finestra scorrevole. Nelle implementazioni di LZ77 solitamente il search buffer è delle dimensioni di qualche migliaia di byte, mentre il look-ahead buffer è qualche decina di byte. Ad esempio in [8], viene descritto l'esempio riportato in Figura 2.1. Ipotizziamo che la

← coded text... `sir␣sid␣eastman␣easily␣t` | `eases␣sea␣sick␣seals`... ← text to be read

Figura 2.1: stringa in input all'encoder LZ77 divisa in search buffer e look-ahead buffer.

sottostringa "sir_sid_eastman_easily_t" sia stata codificata, mentre la rimanente "eases_sea_sick_seals" non lo sia, procediamo come segue. L'encoder scansiona il search buffer (da destra verso sinistra) alla ricerca di un'uguaglianza con il primo simbolo nel look-ahead buffer 'e'. Viene trovata un'uguaglianza nella parola "easily" con il primo carattere, che si trova ad una distanza (offset) pari a 8 dalla fine del search buffer. L'encoder a questo punto procede a confrontare i simboli successivi ad 'e' nel search-buffer con quelli successivi al simbolo 'e' nel looh-ahead buffer alla ricerca del maggior numero di uguaglianze. I simboli che verificano tale uguaglianza sono i simboli della parola "eas", di lunghezza (length) 3. L'encoder a questo punto ripete il procedimento di confronto dei simboli nella parte di search buffer non ancora analizzata alla ricerca di una parola di lunghezza maggiore di 3. Nel nostro caso nella parola "eastman" è presente alla distanza 16 un'altra uguaglianza di lunghezza 3. L'encoder seleziona sempre l'uguaglianza più lunga, o nel caso di più uguaglianze con la stessa lunghezza seleziona quella alla distanza maggiore. Selezionare quella più distante piuttosto che quella più vicina semplifica l'encoder dato non che deve tenere traccia di quest'ultima, anche se d'altro canto segliere l'ultima permetterebbe di prendere in considerazione un offset di dimensioni ridotte. Le chiavi generate da LZ77 sono triplette formate da offset, length, e dal successivo simbolo in input. Dopo che le chiavi vengono stampate sul file compresso, la finestra scorrevole viene shiftata verso destra in corrispondenza dei simboli non ancora analizzati. Le chiavi con offset e lunghezza zero sono comuni all'inizio dello stadio di compressione, quando il search buffer è vuoto o quasi. I primi 5 passi della codifica sono portati in Figura 2.2 [8]. Naturalmente una chiave del tipo (0,0,..) fornisce una scadente compressione e può anche causare espansione. La lunghezza dell'offset è stimata in $\lceil log_2 S \rceil$ dove S è la lunghezza del buffer di ricerca. In pratica il buffer di ricerca è solitamente dell'ordine delle migliaia di byte, e l'offset è tipicamente di 10/12 bit. La lunghezza del campo length è sempre la stessa, ovvero $\lceil log_2(L-1) \rceil$, dove L è la lunghezza

	sir␣sid␣eastman␣	⇒	(0,0,“s”)
s	ir␣sid␣eastman␣e	⇒	(0,0,“i”)
si	r␣sid␣eastman␣ea	⇒	(0,0,“r”)
sir	␣sid␣eastman␣eas	⇒	(0,0,“␣”)
sir␣	sid␣eastman␣easi	⇒	(4,2,“d”)

Figura 2.2: primi 5 passi processo di codifica LZ77.

del look-ahead buffer. La lunghezza del simbolo dipende invece dalla cardinalità A dell'alfabeto utilizzato, e viene definita come $\lceil log_2 A \rceil$.

2.1.2 Decodifica

Il decoder è molto più semplice dell'encoder. LZ77 possiede un metodo di compressione asimmetrico, ovvero codifica e decodifica non agiscono in modo complementare. Il decoder deve mantenere un buffer di dimensione uguale alla finestra scorrevole dell'encoder. Quando il decoder riceve in input una chiave, ricerca nel buffer la corrispondente stringa e la scrive in output insieme al terzo campo della tripletta, shiftando al successivo elemento in input. LZ77 non fa nessuna assunzione sui dati da scandire, infatti non viene presa in considerazione la frequenza con cui questi appaiono in input. L'algoritmo, date le sue caratteristiche, confronta il look-ahead buffer con il testo scandito fino ad una certa soglia (pari alla grandezza del search buffer), mentre il testo scandito in precedenza non viene considerato. In tal modo LZ77 confronta pattern che occorrono vicini tra di loro e funziona bene con file dati che rispecchiano tale caratteristica. Riportiamo in Figura 2.3 i primi passi della decodifica LZ77. La prima chiave letta in input è (0,0,“s”), essa identifica l'inzio della decodifca (offset e lenght sono pari a 0), viene quindi stampata in output la stringa “s” e viene aggiunta al search buffer. La seconda chiave letta in input è la chiave (0,0,“i”) , allo stesso modo viene stampata in output la stringa “i” e viene aggiunta al search buffer. Lo stesso vale per le chiavi (0,0,“r”) e (0,0,“⊔”) che producono in output rispettivamente i caratteri ‘r’ e ‘u’. Appena viene letta in input la chiave (4,2,“d”), questa indica la presenza di una stringa all'interno del search buffer da calcolare. L'offset 4 permette di posizionarsi in corrispondenza del carattere ‘s’ mentre il campo length fa si che vengano copiati due caratteri in modo da formare la stringa “si”. Alla stringa “si” viene quindi concatenato il carattere ‘d’ (terzo campo della chiave) formando la stringa “sid”, viene stampato in output il risultato e la stringa viene aggiunta al search buffer. Il procedimento continua così fino alla fine del file di codici.

Chiave in input	Search buffer	Output
(0,0,"s")	empty	"s"
(0,0,"i")	...s...	"i"
(0,0,"r")	...si...	"r"
(0,0,"␣")	...sir...	"␣"
(4,2,"d")	4 | 3 | 2 | 1 — s | i | r | ␣	"sid"

Figura 2.3: primi 5 passi del processo di decodifica LZ77.

2.2 Algoritmo LZ78

Il metodo LZ78, detto anche LZ2, è stato introdotto in [2]. Tale algoritmo non impiega nessun search buffer, look-ahead buffer o finestra scorrevole, viene invece utilizzata una struttura a dizionario. Il dizionario parte inizialmente vuoto e la sua dimensione è limitata solamente dalla massima memoria disponibile. Uno dei vantaggi di LZ78 è che i dati codificati non vengono mai eliminati, nel dizionario non viene mai cancellata nessuna stringa di simboli a differenza di LZ77 in cui i pattern di ricerca considerati erano solo quelli nel search buffer. Uno degli aspetti negativi invece è che il dizionario di stringhe cresce molto rapidamente e l'occupazione di memoria con esso.

2.2.1 Codifica

L'encoder fornisce in output chiavi formate da coppie: il primo campo di ogni chiave è un puntatore al dizionario, ovvero un riferimento ad uno dei suoi nodi, il secondo è il codice di un simbolo dell'alfabeto Σ. Le chiavi non contengono l'informazione della lunghezza delle stringhe codificate (come accadeva in LZ77), poichè tale informazione è contenuta implicitamente nel dizionario. Ogni chiave corrisponde ad una stringa di simboli dell'alfabeto Σ, tale stringa viene aggiunta al dizionario dopo che viene scritta nel file compresso la chiave corrispondente. Il dizionario parte con la stringa nulla nella posizione zero. Quando un simbolo σ_i è letto in input, viene avviato il processo di ricerca all'interno del dizionario per trovare una stringa che eguagli tale simbolo. Se la ricerca fallisce, il simbolo σ_i non è presente all'interno del dizionario, e viene aggiunto nella successiva posizione

disponibile, scrivendo in output la chiave (0,σ_i). Se in una chiave il puntatore al dizionario assume valore zero, allora si indica la stringa nulla, ovvero la radice. Nel caso in cui la stringa σ_i venga trovata all'interno del dizionario, ad esempio nella posizione 37, viene letto il successivo simbolo in input σ_j, e la ricerca riparte con lo scopo di trovare una stringa nel dizionario che uguagli la concatenazione dei simboli $\sigma_i\sigma_j$. Se nessuna stringa è trovata, allora xy è aggiunta nella successiva posizione disponibile nel dizionario, e viene restituita la chiave (37,σ_j). Il valore nel primo campo della chiave indica la posizione di σ_i nel dizionario, mentre il secondo campo indica il simbolo cha ha fatto fallire la ricerca. Il processo continua così fino alla fine del file dati. Vengono mostrati in Figura 2.4 [8] i primi 15 passi della parte di codifica dell'algoritmo in cui si considera la stringa "sir_sid_eastman_easily_teases_sea_sick_seals". In generale, il simbolo corrente diviene

Dictionary		Token	Dictionary		Token
0	null				
1	"s"	(0,"s")	8	"a"	(0,"a")
2	"i"	(0,"i")	9	"st"	(1,"t")
3	"r"	(0,"r")	10	"m"	(0,"m")
4	"␣"	(0,"␣")	11	"an"	(8,"n")
5	"si"	(1,"i")	12	"␣ea"	(7,"a")
6	"d"	(0,"d")	13	"sil"	(5,"l")
7	"␣e"	(4,"e")	14	"y"	(0,"y")

Figura 2.4: primi 15 passi del processo di codifica di LZ78.

una stringa di un simbolo, l'encoder fa partire quindi il processo di ricerca della stringa all'interno del dizionario. Se la ricerca restituisce esito positivo, a tale stringa viene concatenato il successivo simbolo in input. Più numerose sono le uguaglianze nel dizionario più lunghe sono le stringhe considerate. Non appena la ricerca fallisce viene restitutito il puntatore all'ultimo simbolo dell'uguaglianza insieme al simbolo che ha fatto fallire la ricerca. La grandezza del dizionario può anche venire ricalcolata ogni volta che il programma va in esecuzione. Un grande dizionario può contenere molte stringhe, ma la migliore situazione è quella che combina un dizionario di dimensioni ridotte contenente stringe molto lunghe. Una buona struttura dati per l'implementazione di un dizionario è un albero. L'albero viene inizializzato con la stringa nulla nella radice, tutte le stringe che partono con la stringa nulla (quelle per cui il puntatore è 0) sono aggiunte all'albero come figli

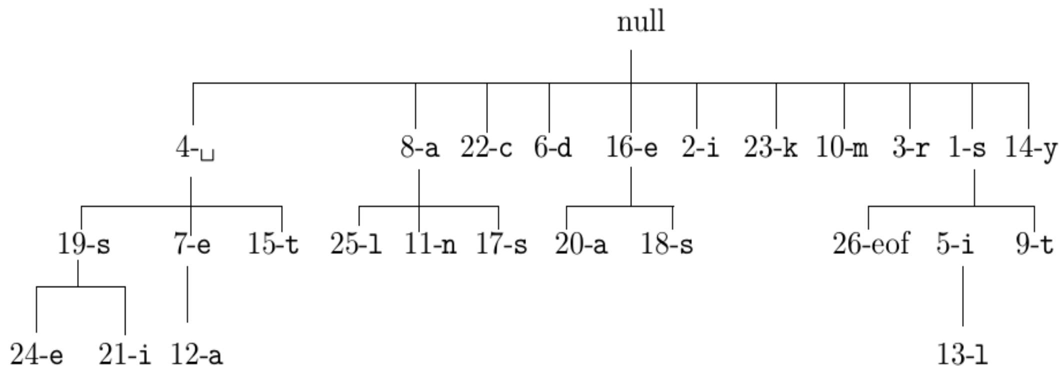

Figura 2.5: albero dei pattern costruito dall'econder LZ78.

della radice. Ogni figlio della radice diventa radice di un sottoalbero. Assumendo che l'alfabeto abbia simboli a 8 bit, ci sono 256 possibili simboli. In principio ogni nodo dell'albero può avere fino a 256 figli differenti. Il processo di inserimento di un nodo nell'albero deve quindi essere fatto in modo dinamico. Quando un nodo viene creato non possiede figli, e nessuno spazio viene riservato per essi. Ogni volta che un nodo viene aggiunto viene richiesta della memoria per esso, e dato che non viene mai cancellato questa caratteristica a volte facilita il processo di gestione della memoria. Viene riportato in Figura 2.5 [8] l'albero costruito dall'encoder sull'analisi della stringa precedentemente presa ad esempio. Con tale albero viene facilitata la ricerca e l'aggiunta di stringhe. Ad esempio, se tentiamo di cercare la stringa "sil", il primo passo dell'algoritmo è quello di scandire i figli della radice alla ricerca di un figlio etichettato con 's', se tale figlio viene trovato allora reiterando il procedimento vengono ispezionati i figli di s alla ricerca di un figlio etichettato con 'i', e così via. Un albero di questo tipo viene anche chiamato "trie".

2.2.2 Decodifica

Come nella codifica, inizialmente il dizionario contiene solo la stringa vuota. Verrà ricostruito dal processo di decodifica. Ad ogni passo una chiave (puntatore, carattere) viene letta in input. Il puntatore si riferisce sempre ad una stringa già presente nel dizionario. La stringa selezionata nel dizionario e il carattere del secondo campo della chiave vengono scritti in output. Successivamente la stringa risultante dalla loro concatenazione viene aggiunta al dizionario. Dopo la decodifica il dizionario sarà esattamente uguale a quello costruito in fase di codifica. Riportiamo in Figura 2.6 lo pseudocodice della decodifica LZ78. Consideriamo ad esempio il risultato della codifica presentato in precedenza. La prima chiave ad essere letta è (0,"s") e dato che il puntatore indica la stringa vuota, "s" viene prima aggiunta alla posizione 1 come figlio della radice e poi viene stampata in output.

La successiva chiave in input è (0,"i"), allo stesso modo della chiave precedente viene aggiunta "i" al dizionario come figlio della radice. Lo stesso avviene per le successive coppie (0,"r") e (0,"⊔"). A questo punto la coppia in input al decoder è (1,"i"), il puntatore 1 è presente nel dizionario in costruzione identificato dalla stringa "s", ad essa viene concatenata la stringa "i". La coppia così formata è "si" che, non essendo presente nel dizionario di decodifica, viene quindi aggiunta alla successiva posizione disponibile e viene stampata in output. Il processo di decodifica continua fino a che il file di codici non viene esaurito.

```
Decodifica LZ78
At the start the dictionary is empty;
W := next code word in the codestream;
C := the character following it;
output the string(W) to the codestream (this can be an empty string), and then output C;
add the string(W)+C to the dictionary;
are there more code words in the codestream? if yes, go back to step 2; if not, END.
```

Figura 2.6: pseudocodice decodifica LZ78.

2.3 Algoritmo LZW

L'algoritmo LZW è una variante di LZ78 implementata da Terry Welch nel 1984 [5]. La peculiarità di questo algoritmo è quella di gestire chiavi formate da un solo elemento, a differenza di LZ78 dove una chiave viene rappresentata da una coppia di elementi. In particolare le chiavi gestite da LZW contengono solamente un puntatore al dizionario dei pattern.

2.3.1 Codifica

Il primo passo della codifica è quello di inizializzare il dizionario con tutti i simboli dell'alfabeto Σ. Se da esempio consideriamo un alfabeto di simboli a 8 bit, l'algoritmo inizializza il dizionario con i primi 256 simboli prima che i dati vengano letti in input. Dato che il dizionario viene precedentemente inizializzato, il primo simbolo in input è sempre presente all'interno di esso. E' questa caratteristica che permette di utilizzare chiavi formate da un solo puntatore.
L'algoritmo di codifica alla generica iterazione agisce come viene ora descritto. Viene letta in input una porzione di dati che definiamo s. Sia σ il simbolo in input successivo ad s, e sia $s\sigma$ la concatenazione di s e σ. Se $s\sigma$ è presente nel dizionario, viene ridefinita s con il valore di $s\sigma$ ed il procedimento riparte considerando il

successivo simbolo in input. Se la ricerca di $s\sigma$ non è andata a buon fine stampiamo l'indice di s in output e aggiungiamo $s\sigma$ al dizionario, assegnandogli il successivo indice disponibile. Viene quindi ridefinita s con il valore di σ ed il procedimento riparte fino al termine dei dati in input. Lo pseudocodice di LZW viene illustrato in Figura 2.7 [4]. In Figura 2.8 [4] vengono riportati i primi passi della codifica LZW data in input la stringa "ababaacacbcaabcbbbac".

Initializations
Initialize dictionary trie the single characters from Σ
Initialize phrase s to first input character
Set $output \leftarrow \langle code(s) \rangle$
Body Repeat until no more input characters
    $\sigma \leftarrow$ read the next input character
    if $s\sigma$ is in the dictionary then set $s = s\sigma$ (*seek-phrase* continues)
    else (the end of a stored phrase has been reached)
        1- $output \leftarrow output \cdot \langle code(s) \rangle$
           add $s\sigma$ to dictionary
        2- $s \leftarrow \sigma$
end Body

Figura 2.7: pseudocodice codifica LZW.

$\langle 0 \rangle$	$\langle 1 \rangle$	$\langle 3 \rangle$	$\langle 0 \rangle$	$\langle 0 \rangle$	$\langle 2 \rangle$	$\langle 7 \rangle$	$\langle 1 \rangle$	$\langle 8 \rangle$	$\langle 3 \rangle$	$\langle 2 \rangle$	$\langle 1 \rangle$	$\langle 14 \rangle$	$\langle 7 \rangle$
a	b	ab	a	a	c	ac	b	ca	ab	c	b	bb	ac
ab	ba	aba	aa	ac	ca	acb	bc	caa	abc	cb	bb	bba	aca
3	4	5	6	7	8	9	10	11	12	13	14	15	16

Figura 2.8: analisi della codifica LZW data in input la stringa "ababaacacbcaabcbbbac". La prima riga a partire dall'alto rappresenta gli indici stampati in output, la seconda riga rappresenta la porzione di testo presente nel dizionario alla generica iterazione mentre la terza riga mostra i pattern aggiunti al dizionario, a cui vengono associati gli indici sottostanti.

2.3.2 Decodifica

Un'altra importante caratteristica di LZW è rappresentata dal fatto che i processi di codifica e decodifica sono perfettamente simmetrici.
Il primo passo della fase di decodifica è quello di inizializzare il dizionario con tutti i simboli dell'alfabeto Σ. Inizia quindi l'analisi dei dati precedentemente compressi. Come nella codifica, dato che il dizionario viene precedentemente inizializzato, il primo indice in input rappresenta un elemento già presente tra quelli memorizzati. Il successivo passo è quello di aggiungere al dizionario la stringa $s\sigma$, formata dalla concatenazione della stringa ricavata dall'indice precedentemente letto e dal simbolo σ, che però non è definito. Questo simbolo viene definito come il primo carattere della stringa rappresentata dal successivo indice in input.
Consideriamo nuovamente da esempio la stringa "ababaacacbcaabcbbbac". Il dizionario viene inizializzato con i simboli a,b,c rispettivamente aventi indice 0,1,2. Il primo indice letto in input dal decoder è 0. Tale indice corrisponde alla stringa "a", che viene prima cercata nel dizionario e poi stampata in output. Il successivo indice in input è 1, che corrisponde alla stringa "b". Questa stringa viene cercata nel dizionario, stampata in output e poi concatenata alla stringa stampata al passo precedente. Questa concatenazione produce la stringa "ab", che non essendo presente nel dizionario viene aggiunta utilizzando il primo indice disponibile, ovvero 3, dato che l'indice 2 è occupato dal simbolo c. La stringa "ab" viene quindi stampata in output. Il processo continua così fino alla decodifica dell'intero input.

2.4 Algoritmo LZMA

LZMA (Lempel-Ziv-Markov chain- Algorithm) [6], è una variante di LZ77. Studiato per avere un elevato tasso di compressione e una veloce fase di decodifica, il dizionario utilizzato da LZMA può avere una dimensione variabile tra 1Gb fino ad un massimo di 4Gb. Con un processore a 2 GHz si può codificare 1 Mb/s e decodificare fino a 20 Mb/s.

2.4.1 Codifica

Come nella codifica LZ77, i tre diversi tipi di dato vengono stampati in output sono lettere (tipicamente codici dell'alfabeto ASCII), lunghezze e distanze. Se durante la fase di ricerca nessuna corrispondenza viene trovata all'interno del search buffer, viene stampata una lettera appartenente all'intervallo [0,255]. Se la ricerca ha esito posivito viene stampata in output una coppia di dati (lunghezza,distanza). Data

la grande dimensione del search buffer, viene mantenuto in memoria un array contenente le ultime 4 distanze stampate in output. Se una distanza durante la fase di codifica eguaglia una di quelle all'interno dell'array, la coppia corrispondente viene stampata in output insieme ad un indice di 2 bit, che rappresenta un elemento dell'array. La localizzazione dei pattern all'interno del search buffer viene fatta attraverso l'uso di una tabella hash. L'output della tabella hash viene rappresentato dall'indice di un array chiamato anche hash-array. La grandezza dell'hash-array viene scelta come potenza di 2 che più si avvicina alla metà della grandezza del dizionario. L'indice dell'hash-array è in funzione quindi della dimensione del dizionario. Se prendiamo come esempio un dizionario di 256MB (2^8), la grandezza dell'hash-array è 2^7, e la funzione hash deve calcolare e restituire in output indici da 8 bit. Le grandi dimensioni dell'hash-array minimizzano le possibili collisioni. L'encoder LZMA può cercare all'interno della tabella hash fino a 4 byte alla volta, il numero di elementi selezionati nell'hash-array dipende dalla grandezza del dizionario. Si possono utilizzare due metodi di ricerca: hash-chain (il metodo più veloce) oppure binary-tree (il più efficiente).
Nel metodo hash-chain l'output della funzione hash è un puntantore ad una lista. Ogni lista contiene un insieme di distanze. Se ad esempio due byte xy vengono processati dalla funzione hash e viene restituito il valore 123, la lista puntata da tale indice contiene le distanze dalle coppie xy nel search buffer. Le liste possono essere di grandi dimensioni, solitamente vengono controllate solo le prime 24 distanze presenti in ogni lista. Nel nostro caso vengono controllate le prime 24 distante della lista puntata dall'indice 123 che corrispondono alle ultime 24 occorrenze di xy. La distanza scelta tra queste 24 viene codificata, emessa in output e aggiunta all'inizio della lista, in modo che l'occorenza corrispondente risulti l'ultima trovata.
Nel metodo binary-tree l'output della funzione hash viene rappresentato da un puntatore ad un hash-array di alberi binari di ricerca. Inizialmente l'hash-array è vuoto e non ci sono alberi binari in memoria. Durante la scansione dei dati, ogni byte letto andrà a definire un nodo appartenente a qualche albero binario nell'hash-array. Viene creato un albero binario di ricerca per ogni coppia di byte letti in input. Ogni volta che si accede ad un albero binario per la selezione di una sua stringa, viene modificato seguendo queste regole:

- L'albero deve sempre rimanere, dopo ogni modifica, un albero binario.
- Le stringhe cercate recentemente vengo aggiunte nelle vicinanze della radice attraverso delle operazioni di modifica dei puntatori.

Riportiamo un esempio di quest'ultimo metodo di codifica [8]. Consideriamo in input all'encoder una terna 'abm' di byte. Questa terna è presente per la prima volta in input alla posizione 11 e viene memorizzata nell'hash-array all'indice 62. La Figura 2.9 [8] mostra come un albero binario viene creato e aggiornato. I primi

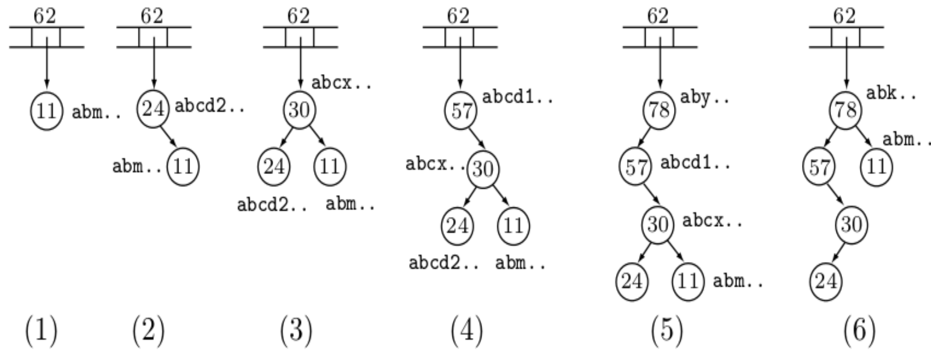

Figura 2.9: creazione e modifica di un albero nell'hash-array.

tre passi del processo sono descritti in seguito:

1. Al primo passo della codifica la funzione hash processa la coppia 'ab' restituendo l'indice 62. L'indice restituito viene quindi esaminato, risultando privo di elementi. Alla posizione 62 viene quindi costruito un albero binario definendo la radice come un nodo di indice 11 contenente la terna 'abm'. L'encoder fa in modo che la locazione dell'hash array 62 contenga un riferimento alla radice dell'albero.

2. Supponiamo ora che in input siano presenti i byte 'abcd2' alla posizione 24 del testo. I primi due byte vengono processati dalla funzione hash che restituisce nuovamente il codice 62 dell'hash-array. Nella locazione 62 viene però trovato il riferimento ad un nodo il cui indice è 11. L'encoder crea quindi un nuovo nodo con indice 24, questo va a rimpiazzare il nodo 11 diventando la nuova radice dell'albero binario. L'ordinamento lessicografico fa si che 'abcd2' sia più piccolo di 'abm', inserendo ques'ultimo come figlio destro di 24.

3. Sono ora presenti in input i byte 'abcx' alla posizione 30 del testo. La funzione hash processa i byte 'ab' fornendo sempre il codice 62. L'encoder rimpiazza la radice con il nodo 30 che rappresenta i byte 'abcx' e sposta il nodo 24, che diventa figlio sinistro di 30, dato che lessicograficamente 'abcd2' è più piccolo di 'abcx'. Il nodo 11 viceversa viene spostato nel sottoalbero destro di 30 sempre a causa dell'ordinamento lessicografico.

2.4.2 Decodifica

La decodifica LZMA, come accennato in precedenza fornisce elevate prestazioni in termini di quantità di dati decodificati al secondo. Questo avviene grazie ad alcune delle caratteristiche principali elencate in [6]. Tali caratteristiche comprendono:

1. Un dizionario di dimensioni relativamente piccole (fino a 256 MB).
2. Una velocità di decompressione stimata in 8-12 MB/s su un processore a 1000 MHz, e 500-1000 KB/s su un 100 MHz.
3. Poca memoria richiesta per la fase di decodifica (8-32 KB in aggiunta alla grandezza del dizionario).
4. Codici di piccole dimensioni generati per la decodifica (circa 8-12 KB).

Il decoder LZMA utilizza solo operazioni con interi, in particolare ci sono alcune operazioni che influiscono negativamente sulla velocità della decompressione [6]:

1. Moltiplicazioni di interi 32*16 bit.
2. Shift e operazioni aritmetiche a 32 bit.

LZMA non è in grado di allocare autonomamente la memoria necessaria per la decodifica, quindi è necessario calcolarla, allocarla è fornirla all'algoritmo. La memoria richiesta per la fase di decodifica dipende dall'interfaccia che si vuole utilizzare tra queste due:

1. Decodifica da memoria a memoria.
2. Decodifica mediante buffer.

I passi della decodifica sono i seguenti:

1. Viene letto in input il primo byte di flusso dati, che contiene caratteriste sul flusso stesso. Si procede con un controllo di correttezza di tale byte e vengono impostate delle variabili di codifica sulla base del suo valore.
2. Viene calcolato l'ammontare di memoria necessaria per l'esecuzione dell'algoritmo.
3. Viene avviata la fase di decodifica in modalità da memoria a memoria o buffer.

3 Pattern con gap

Nel seguente capitolo vengono introdotti alcuni aspetti relativi alla compressione orientata ai pattern con gap [9]. Nella sezione 3.1 viene data la definizione di pattern con gap. Nella sezione 3.2 vengono elencati alcuni ambiti della bioinformatica in cui la compressione dati ha trovato impiego. Nella sezione 3.3 viene descritta una particolare applicazione dei pattern con gap orientata alla compressione delle immagini attraverso un algoritmo offline. Infine nella sezione 3.4 viene descritto in dettaglio l'algoritmo LZWA, oggetto degli esperimenti del capitolo successivo.

3.1 Introduzione ai pattern con gap

In questa sezione si riporta la definizione di pattern con gap data in [9]. Un pattern con gap è un tipo particolare di ridondanza che emerge specialmente nella biologia molecolare e nello studio dei genomi. In semplici termini, un pattern con gap consiste di una stringa di caratteri intermittenti solidi e variabili, che appare più o meno frequentemente in una data sequenza. Dato che i pattern con gap sembrano essere un buon modello per rappresentare sequenze e sottosequenze in ambito biologico, gli algoritmi per la loro ricerca sono di grande interesse. In generale la ricerca dei pattern con gap e il loro utilizzo è reso particolarmente difficile dal fatto che il numero dei candidati pattern con gap cresce esponenzialmente con la lunghezza della sequenza studiata. Fortunatamente in molti casi è possibile una riduzione significativa del numero di pattern con gap. Ad esempio nel contesto degli schemi di sostituizione testuale, ad ogni iterazione viene naturale imporre che il pattern con gap scelto soddisfi certe condizioni massimali che prevengono una perdita di informazione. Diamo di seguito alcune definizioni utili per sviluppare l'argomento. Consideriamo una sequenza s di caratteri appartenenti all'alfabeto $\Sigma \cup \{_\}$, dove '_' indica un carattere variabile, mentre Σ è un alfabeto finito

di simboli σ_i ben definiti. Il carattere variabile '_' rende la sequenza s mutabile dato che può assumere diversi valori tra quelli di Σ. Quando s contiene almeno un carattere '_' viene detta estensibile, altrimenti viene detta rigida. Dato un pattern estensibile s, si può realizzare a partire da esso un pattern rigido, andando a sostituire ad ogni carattere variabile un carattere di Σ.

Tradizionalmente i codici utilizzati nella compressione dati vengono creati utilizzando specifiche statistiche e sintattiche. Il modello per eccellenza dell'approccio statistico viene rappresentato dalla codifica Huffman [10], dove i simboli vengono classificati in accordo con la frequenza con cui si ripetono. A tali simboli vengono associate parole di codice sempre più lunghe al diminuire della loro probabilità. Nell'approccio sintattico invece i codici vengono costruiti utilizzando pattern che esibiscono caratteristiche come la robustezza al rumore o la sincronizzazione. Il punto focale in questi sviluppi è la struttura delle parole di codice. Una parola di codice è un pattern w di lunghezza m tale che ogni altra parola deve essere alla distanza d da w, la distanza viene misurata in termini di errori di un certo tipo. Nella codifica Huffman ad esempio una volta che i caratteri vengono classificati statisticamente, si possono costruire codici che rispettano certe caratteristiche sintattiche obbedendo alla proprietà di prefisso. Analogamente una volta che un insieme di codici per la correzione degli errori viene progettato, vengono prese in considerazione le statistiche sul file in ingresso per la codifica. Solitamente l'aspetto statistico e sintattico vengono quindi considerati separatamente.
La nozione di pattern con gap che prendiamo in considerazione combina la struttura di pattern in termini di descrizione sintattica e statistica, attraverso il conteggio delle occorrenze. Gli aspetti principali della compressione mediante pattern sono i seguenti:

- Impiego di un algoritmo di ricerca di pattern con gap.

- Impostazione dei criteri per la scelta e la codifica di pattern con gap usati nella compressione.

- Utilizzo di nuovi programmi che permettono di implementare gli aspetti dei primi due punti.

La fusione dei primi due punti viene brevemente descritta in seguito. Consideriamo il processo di ricerca dei pattern con gap diviso in due stadi. Il primo stadio scopre i pattern dotati di un certo insieme di proprietà e il secondo implementa tali proprietà nella compressione. Come per i classici algoritmi di pattern solidi, l'informazione contenuta nel pattern con gap può essere ricostruita al ricevitore,

inviando separatamente informazioni relative ai gap da riempire. Il confronto del tasso di compressione raggiunto da pattern e pattern con gap dovrebbe dimostrare che i secondi forniscono un risparmio addizionale alla compressione, in termini di misure di somiglianza e classificazione di sequenze.

3.2 Applicazioni in ambito bioinformatico

Il principale scopo di utilizzo dei metodi della teoria dell'informazione nella biologia è stato, fino a poco tempo fa, quello di analisi dello "scostamento da casualità" di sequenze di nucleotidi e amminoacidi [11]. Dato il crescente numero di dati si sostiene che l'utilizzo della teoria dell'informazione potrebbe offrire dei vantaggi sostanziali nei confronti dei tradizionali metodi statistici. La teoria della compressione dati, ramo fondamentale della teoria dell'informazione e della tecnologia, ha avuto un ruolo primario nel connettere la classificazione, le statistiche e altre nozioni riguardanti sequenze complesse nel mondo della bioinformatica.
I pattern con gap rappresentano uno dei risultati della compressione dati applicata alla bioinformatica. Questa tipologia di pattern è in particolare orientata alla compressione e alla ricerca di sottosequenze biologice. Gli algoritmi di compressione che utilizzano i pattern con gap dovrebbero essere in grado di gestire delle particolari classi di ridondanze presenti in sequenze biologiche che i classici algoritmi di compressione non sono in grado di rilevare, inducendo livelli di compressione maggiori.
Vengono elencate in seguito alcune aree di rilevanza generale per la biologia computazionale, appartenenti agli ambiti della compressione dati e alla ricerca di sottosequenze statisticamente importanti, che contengono i principi su cui si basano i pattern con gap.

3.2.1 Compressione per la memorizzazione di sequenze

In [12] alcuni scenari sono stati proposti per caratterizzare il problema della compressione nella memorizzazione di sequenze genomiche:

- Modo "orrizzontale": data una sequenza biologica, questa viene compressa facendo solo uso dell'informazione contenuta nella sequenza stessa. La valutazione dei metodi di compressione è solitamente portata avanti in tale modo.

- Modo "verticale": dato un insieme di sequenze biologiche, ogni sequenza viene compressa facendo uso dell'informazione contenuta nell'intero insieme,

considerando tipicamente sottostringhe di quest'ultimo.

Il modo orizzontale è stato proposto per risolvere problemi di natura teorica e pratica. Nella teoria il suo obiettivo è quello di mettere in evidenza le proprietà statistiche e strutturali di sequenze biologiche. Nella pratica esso è utilizzato per la riduzione dei costi della trasmissione e memorizzazione delle informazioni. Il modo verticale ha essenzialmente gli stessi vantaggi ma si basa su un concetto differente. Infatti si può osservare che mentre una sequenza presa singolarmente potrebbe essere difficile da comprimere, considerare l'insieme di più sequenze potrebbe facilitare tale problema. Riportiamo in seguito alcune metodologie di compressione dati per la memorizzazione di informazioni nella bioinformatica [13].

- Metodi statistici sostitutivi

  Questi metodi combinano le principali tecniche di compressione dati: quella statistica e quella di sostituzione. In generale data una sequenza biologica, essa si divide in sottostringhe che sono facilmente comprimibili attraverso l'uno o l'altro metodo. Una particolare definizione di funzione di guadagno è utilizzata per stabilire in che quantità queste sottostringhe vengono suddivise nei due gruppi. Questo paradigma offre una grande quantità di ambiti applicativi in termini di compromesso tra compressione e velocità. Alcuni metodi che sfruttano questo paradigma sono ad esempio $GenCompress$[14] e $GeNML$[15]. Tali metodi sfruttano la naturale ridondanza delle sequenze biologiche che si presentano sotto forma di ripetizioni uguali o approssimate.

- Metodi di trasformazione

  La trasformazione di Burrows–Wheeler [16] è l'esempio più conosciuto in questa classe di metodi. Questa trasformazione si basa su un preprocessing della sequenza in analisi prima che la fase di codifica abbia inizio. Ci sono solo due algoritmi basati su tale trasformazione descritti in [17] e [18], l'ultimo dei questi ha favorito un grande passo avanti nella compressione delle sequenze biologiche, portando a nuove intuizioni sulla compressione di proteine su scala genomica. Alcune applicazioni di tale algoritmo hanno portato in evidenza la presenza di ripetute sottosequenze separate da migliaia di simboli che non erano mai state osservate prima.

- Metodi basati sulla grammatica

  In questa classe di metodi una stringa di testo x viene compressa deducendo o

utilizzando una grammatica libera dal contesto $G(x)$ che la generi. La stringa viene quindi codificata utilizzando una codifica “propria”, seguendo certe regole di produzione. Per le sequenze biologiche, esistono tre metodi appartenenti a tale classe: *DNASequitor* [19], *RNACompress* [20], *GenCompress*. In particolare *RNACompress* è stato studiato specificatamente per l'RNA, ed ha due obiettivi principali: la compressione della struttura dell'Rna, e la rappresentazione di un modello della struttura secondaria dell'Rna. Alcuni esperimenti sul primo obbiettivo mostrano un importante miglioramento nella compressione dati fino al 50% rispetto al metodo di riferimento *GenCompress*, con un guadagno di velocita in codifica e decodifica di due ordini di grandezza.

- Metodo di compressione delle tabelle

  Questo metodo, introdotto da Buchsbaum in [21], è un'applicazione di compressione di archiviazione di massa e trasmissione dei dati. Il suo obiettivo è quello di essere veloce, effettivo e online. Apostolico ha recentemente dimostrato [22] che i metodi tabellari di compressione possono essere utilizzati con successo nella classificazione, facendo crescere l'interesse di tale strumento per le indagini biologiche. E' stato ad esempio applicato con buoni risultati su un set di 1000 campioni rigurardanti la bioversità selezionati in alcuni anni da INBio [23].

3.2.2 Pattern Discovery

Il pattern discovery [24] si occupa dell'individuazione di pattern in sequenze significative. In questo contesto la compressione dati, associata al principio di Minimum Description Lenght (MDL) [25], si inserisce come utile strumento per l'individuazione di schemi biologici significativi. I pattern con gap proposti in questo capito sono un particolare tipo di pattern, che favoriscono la combinazione dei concetti di compressione dati e pattern discovery. I pattern con gap sono orientati allo studio di sequenze in cui sono presenti dati ripetuti con una certa variabilità. Questi pattern ad esempio potrebbero risultare utili nella ricerca dell'abbondanza, della scarsità, o della ripetizione di particolari sottosequenze di DNA. Meccanismi di duplicazione molecolari come ad esempio la retrotrasposizione sono responsabili della presenza di sequenze duplicate di DNA. Le strutture duplicate che questi meccanismi producono svolgono importanti funzioni sia a livello evolutivo che della regolazione. Alcuni di essi sono coinvolti anche nelle malattie umane, pertanto l'identificazione di sottosequenze ripetuti nel DNA è molto importante e rappresenta anche un importante ramo di sviluppo per gli algortimi di calcolo

combinatorio.
Nell'ambito generale del pattern discovery un grande numero di ricerche sono state dedicate alla valutazione dell'importanza statistica del verificarsi di un pattern in una lunga sequenza. Questo aspetto dà luogo principalmente a due tipi problemi. Il primo tipo di problema richiede l'identificazione di sottosequenze in una sequenza statisticamente importante, parametro che viene stabilito da una data misura. Il secondo tipo di problema riguarda l'estrazione dei pattern, di solito rappresentati da espressioni regolari ricavate da un insieme di sequenze.

3.3 Compressione di immagini

L'utilizzo delle immagini nel mondo multimediale è di fondamentale importanza, qualsiasi applicazione ad esempio utilizza una interfaccia grafica, un altro esempio di utilizzo massivo delle immagini avviene nello streaming video dove il flusso di immagini è costante dal sorgente al ricevitore. Le immagini hanno un ruolo molto importante nel mondo digitalizzato, ma hanno come aspetto negativo il fatto di essere rappresentate da file dati di grandi dimensioni. Questo è il motivo principale che ha portato allo sviluppo di tecniche per la compressione di immagini. Una fondamentale caratteristica delle immagini è che possono essere compresse eliminando alcune informazioni senza perdere l'immagine iniziale. Tale caratteristica è viene rispettata se vengono eliminate quelle componenti che non sono visibili ad occhio umano. I metodi di compressione lossy si sono quindi sviluppati nell'ambito delle immagini digitali attraverso la tecnica della rimozione dei dati irrilevanti in esse.

I principi su cui si basa la compressione mediante pattern con gap possono essere estesi alle immagini 2D. In tale adattamento, per la compressione vengono considerati blocchi di pixel solidi o influenzati da un indeterminato numero di pixel variabili. Tali blocchi vengono estratti dalla base dei motif risultanti dall'autocorrelazione dell'immagine con se stessa. I risultati suggeriscono un' elevata compressione al costo di una trascurabile perdita di informazione.

Il metodo qui riportato è descritto in [26]. Questo algoritmo è intrinsecamente offline, quindi trova una migliore applicazione nel contesto di archivi di immagini piuttosto che nel contesto delle trasmissioni. Assumiamo che l'input consista in un'immagine digitale $\mathbf{I}$, rappresentata con un array rettangolare di $\mathrm{N} = m \times n$ pixel, dove ogni pixel rappresenta un carattere di un alfabeto di pixel Σ. La nozione di suffisso per una stringa viene tradotta in questo caso in quella di campione di immagine 2D, in Figura 3.1 riportiamo un esempio da [26]. In aggiunta ai simboli

$$I_{[m,n]} = \begin{Vmatrix} i_{11} & i_{12} & \dots & i_{1n} \\ i_{21} & i_{22} & \dots & i_{2n} \\ . & & & \\ . & & & \\ . & & & \\ i_{m1} & i_{m2} & \dots & i_{mn} \end{Vmatrix} \qquad S_{ij} = \begin{Vmatrix} i_{ij} & i_{ij+1} & \dots & i_{in} \\ . & & & \\ . & & & \\ . & & & \\ i_{mj} & i_{mj+1} & \dots & i_{mn} \end{Vmatrix}$$

(a) (b)

Figura 3.1: (a) definizione di immagine come array di pixel , (b) definizione di suffisso dell'immagine

dell'alfabeto Σ, definiamo il carattere variabile '$\circ$'. Un'immagine P di dimensione $m_p \times n_p$, con $m_p < m$ e $n_p < n$, è un'occorrenza di **I** se tale immagine è alla posizione $[k, f]$ (con $k \leq m - m_p + 1$ e $f \leq n - n_p + 1$), e se P$[i, j] \preceq$ **I** $[k+i-1, f+j-1]$ è valida per $1 \leq i \leq m_p$ $1 \leq j \leq n_p$. Data un immagine $\mathbf{I} \in \Sigma$ e un intero positivo $q \leq m \times n$, un'immagine M $\in \Sigma \cup \{\circ\}$ è un q-motif di **I** con lista delle posizioni $F_M = (f_1, f_2, ...f_p)$, dove gli f_i indicano una coppia di indici in **I**, se valgono tutte le seguenti asserzioni:

- è presente almeno un carattere solido adiacente ad ogni arco di M
- p $\geq$q
- M occorre ad ogni $fi \in F_M$
- non esiste nessuna posizione $f, f \neq f_i, 1 \leq i \leq p$, tale che M occorre in **I**.

Defianiamo allora un 2-motif un particolare motif che rispetta i precedenti punti con $q = 2$. Dati due motif M_1 e M_2 di rispettive dimensione $m_1 \times n_1$, $m_2 \times n_2$, con $m_1 \preceq m_2$ e $n_1 \preceq n_2$, abbiamo che $M_1 \preceq M_2$ è verificata se $M_1[i, j] \preceq M_2[i + c, j + d]$ per $1 \leq i \leq m_1$ e $1 \leq j \leq n_1$, per qualche c e d fissati tale che $0 \leq c \leq m_2 - m_1$ e $0 \leq d \leq n_2 - n_1$. In questo caso diciamo anche che M_1 è un incluso ("sub-motif") in M_2. Il numero di motif con pixel variabili cresce esponenzialmente con la dimensione della grandezza dell'input. Questo fa si che una estrazione esaustiva dei motif sia irrealizzabile. Diamo una serie di definizioni per ridurre il numero di candidati motif da estrarre. Siano $M_1, M_2, \dots M_n$ i motif di un'immagine **I**. Un motif è massimale in composizione se non esiste nessun altro motif M_l, $l \neq i$, tale che $F_{Mi} = F_{Ml}$ e $M_i \preceq M_l$. Un motif massimale in composizione lo è anche in lunghezza se e solo se non esiste nessun motif M_j,

$j \neq i$, tale che M_i è un sub-motif di M_j e $|F_{Mi}| = |F_{Mj}|$. Un motif è massimale se lo è sia in composizione che in lunghezza. Fondamentale risulta la definizione di motif ridondante. Un motif massimale M in **I** con lista delle posizion $F_M = (f_1, f_2, ...f_p)$ è detto ridondante se esiste un sub-motif massimale M_i , $1 \leq i \leq p$, tale che $F_M = F_{M_1} \cup F_{M_2} \cup ... \cup F_{M_p}$, e $M \preceq M_i$. Data un'immagine **I** definita sull'alfabeto Σ , sia **M** l'insieme di tutti i motif massimali in **I**. Un sottoinsieme β di **M** è definita base di **M** se valgono le seguenti asserzioni:

- Per ogni M $\in \beta$, M è irridondante rispetto a $\beta - \{M\}$.
- Sia Υ (ψ) l'insieme di tutti i motif ridondanti massimali utilizzati dall'insieme dei motif impliciti nell'insieme ψ, allora $\mathbf{M} = \Upsilon(\beta)$.

La base β formata da k motif per l'immagine **I** (definita sull'alfabeto **M**) è unica per ogni valore di k. Siano σ_1 e σ_2 due caratteri dell'alfabeto $\Sigma \cup \{\circ\}$, il consenso di $\sigma_1 \oplus \sigma_2$ di σ_1 e σ_2 è definito come: $\sigma_1 \oplus \sigma_2 = \sigma_1$ se $\sigma_1 = \sigma_2$, altrimenti $\sigma_1 \oplus \sigma_2 = \circ$. Il consenso di due immagini $\mathbf{I}_1$ e $\mathbf{I}_2$ è l'immagine identificata con $C = \mathbf{I}_1 \oplus \mathbf{I}_2$ di dimensione $m \times n$, dove m = $\min\{m_1, m_2\}$ e n = $\min\{n_1, n_2\}$ tale che $C[h,k] = \mathbf{I}_1[h,k] \oplus \mathbf{I}_2[h,k]$, per $1 \leq h \leq m$ e $1 \leq k \leq n$. Considerando $\mathbf{I}_1$, $\mathbf{I}_2$ e C il loro consenso, l'eliminazione da C di tutte le righe e le colonne costituite solamente da pixel variabili crea un'immagine (eventualmente vuota) che viene chiamata "meet" di $\mathbf{I}_1$ e $\mathbf{I}_2$. L'insieme delle autocorrelazioni di **I** è dato dai meet generati da **I** attraverso i campioni $S_{ij}(1 \leq i \leq m, 1 \leq j \leq n)$. Considerando $\mathbf{I}_1$, $\mathbf{I}_2$ e C il loro consenso, l'eliminazione da C di tutte le righe e le colonne costituite solamente da pixel variabili crea un'immagine (eventualmente vuota) che viene chiamata "meet" di $\mathbf{I}_2$ e $\mathbf{I}_2$. L'insieme delle autocorrelazione di **I** è dato dai meet generati da **I** attraverso i campioni $S_{ij}(1 \leq i \leq m, 1 \leq j \leq n)$. Sia β una base (2-motif) di un immagine **I** di dimensione N appartenente all'alfabeto Σ, e sia A l'insieme delle autocorrelazioni di **I**, allora valgono le seguenti:

- $\beta \subseteq A$
- il numero di motifs in β è O(N)

Il limite sul numero di motif appena introdotto permette lo sviluppo di efficienti algoritmi per le estrazioni delle basi in tempi O(N^2) e O(N^3) da immagini con un alfabeto binario. Prendiamo ad esempio in considerazione l'esempio in Figura 3.2 [26]. L'immagine **I** considerata è in Figura 3.2.(a). L'immagine 3.2.(b) rappresenta il consenso tra **I** e il suo campione S_{13} dalla quale viene generata l'autocorrelazione A_{13} in Figura 3.2.(c). Le altre autocorrelazioni mostrate in Figura 3.3.(c)-(h) sono estratte in modo simile. Oltre a queste abbiamo $A_{41} =$

$I_{[4,3]} = \begin{Vmatrix} 0 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 1 \end{Vmatrix}$ (a) $C_{13} = \begin{Vmatrix} \circ \\ 1 \\ \circ \\ 1 \end{Vmatrix}$ (b) $A_{13} = \begin{Vmatrix} 1 \\ \circ \\ 1 \end{Vmatrix}$ (c) $A_{31} = \begin{Vmatrix} 0 & 0 & 1 \\ 1 & \circ & 1 \end{Vmatrix}$ (d) $A_{12} = \begin{Vmatrix} 0 & \circ \\ 1 & 1 \\ 0 & \circ \end{Vmatrix}$ (e)

$A_{21} = \begin{Vmatrix} \circ & 1 \\ \circ & 1 \\ 0 & 1 \end{Vmatrix}$ (f) $A_{32} = \begin{Vmatrix} 0 & \circ \\ \circ & 1 \end{Vmatrix}$ (g) $A_{22} = \begin{Vmatrix} \circ & 1 \\ 0 & \circ \end{Vmatrix}$ (h)

Figura 3.2: creazione delle autocorrelazioni dall'immagine $\mathbf{I}_{[4,3]}$.

2D Basis Extraction
Input:
- an image I

Output:
- the 2D basis $\mathcal{B} = \{M_1, M_2, \ldots, M_k\}$

begin
1: compute the set $\mathcal{A}$ of the autocorrelations of I
2: eliminate from $\mathcal{A}$ possible duplicates
3: **for each** $M_i \in \mathcal{A}$
4: compute the list of occurrences $\mathcal{F}_{M_i}$
5 $\mathcal{T} = \mathcal{A}$
6: **for each** $M_i \in \mathcal{T}$
7: **if** there exist p motifs $M_1, M_2, \ldots M_p \in \mathcal{A}$ such that $\mathcal{F}_{M_i} = \bigcup_{j=1}^{p} \mathcal{F}_{M_j}$
8: $\mathcal{T} = \mathcal{T} - \{M_i\}$
9: $\mathcal{B} = \mathcal{T}$
end

Figura 3.3: algoritmo di estrazione delle basi data in input un'immagine **I**.

$||0\ 1||$, le due autocorrelazioni A_{12} and A_{42} che consistono del solo carattere 0, e A_{23} , A_{33} che consistono solo del carattere 1. Appena l'estrazione delle basi da **I** viene conclusa, vengono calcolate le rispettive liste di occorrenze, e vengono tenuti in considerazione solo i motif irridondanti, che nel nostro esempio formano la base $\beta = \ A_{31}, A_{12}, A_{21}, A_{13}$. L'algoritmo per l'estrazione della basi viene illustrato in Figura 3.3 [26]. Il successivo passo è quello della creazione di un dizionario che memorizzi i campioni di immagine. Questi vengono ordinati secondo una stima del guadagno indotto da ognuno di essi nelle compressione. La Figura 3.4 [26]

illustra il procedimento di creazione del dizionario, dove vengono indicati con "patches" i campioni di immagine. I passi 2 e 3 vengono eseguiti in un tempo O(N), dove N ricordiamo è definita come la dimensione dell'immagine in input. Il costo totale dei passi da 1 a 3 è quindi O(N^2). Partendo dal campione $\bar{M}$ che induce il guadagno massimo, le liste delle occorrenze degli altri campioni vengono aggiornate eliminando le occorrenze che sovrappongono quelle della lista di $\bar{M}$. Il processo viene ripetuto finche tutti i campioni nella lista della base non vengono processati. I passi 4-6 possono essere svolti in un tempo O(N^3). Il dizionario D dei campioni di immagini creati è un insieme ordinato. Ogni campione contiene la propria lista delle occorrenze utilizzate nel processo di codifica. Per controllare la

DICTIONARY COMPUTATION
Input:
- an input image I
- the 2D basis $\mathcal{B}$ of I
Output:
- the dictionary of patches $\mathcal{D}$

begin
1: **for each** $M_i \in \mathcal{B}$
begin
2: compute the gain G_i of M_i
3: store M_i in $\mathcal{D}$ in a list sorted according to G_i
end
4: **for each** M_i in $\mathcal{D}$
5: **for each** M_j that follows M_i in $\mathcal{D}$
6: eliminate from $\mathcal{F}_{M_j}$ possible occurrences overlapping with some $(h.k) \in \mathcal{F}_{M_i}$
7: **for each** M_i in $\mathcal{D}$
8: **if** $\mathcal{F}_{M_i} = \emptyset$
9: $\mathcal{D} = \mathcal{D} - \{M_i\}$
end

Figura 3.4: algoritmo della creazione del dizionario di campioni dell'immagine **I**.

perdita dell'informazione nella compressione si può adottare un limite al massimo numero di pixel variabili nella generazione delle basi. Per le immagine binarie ad esempio una densità pari a 0.5% sembra portare a migliori risultati rispetto ai tradizionali metodi grazie al largo numero delle ripetizioni contenute nei dati. Le tecniche di compressione delle immagini basate sulla scelta di particolari basi 2D sembrano superare in performance i tradizionali metodi. Descriviamo in seguito alcuni risultati ottenuti in [5] applicando il metodo sopra descritto. I campioni vengono selezionati attraverso una stima di guadagno che rispetta tale definizine:

$$G_i = N - |M_i| \times |F_{M_i}| + |M_i| + \chi \times (|F_{M_i}| - 1)$$

Per esempio per il motif A_{13} della base mostrata in precedenza (Figura 3.3), che ha 3 occorrenze ed è di dimensione 3, il guadagno è dato da $G_i = 6 + 2\chi$. Con lo scopo di controllare la perdita di informazione, per le immagini binarie selezionate in Figura 3.5 [26] viene adottata un densità di pixel variabili pari al 0.5%, che sembra portare alla compressione maggiore. Per il confronto applichiamo alle stesse immagini degli schemi classici che sfruttano campioni lineari e solidi. I risultati vengono riportati in Figura 3.6 [26].

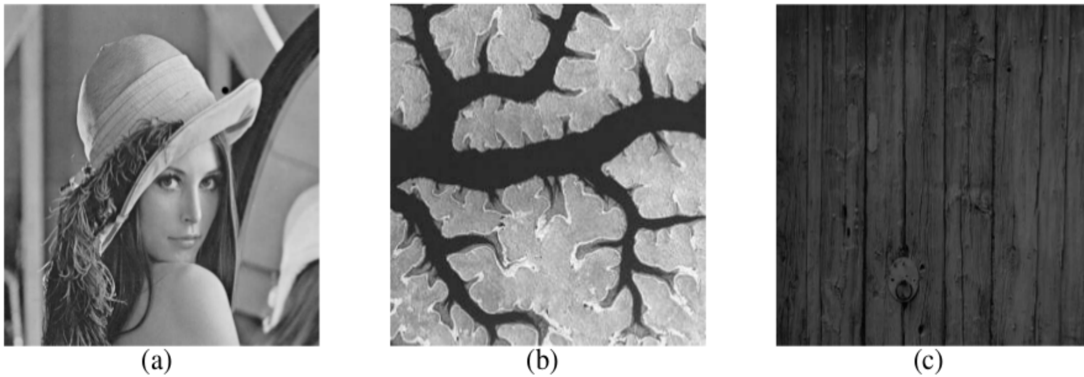

Figura 3.5: immagini (a) Lena , (b) Satellite, (c) Texture

	Solid Patches	Linear Motifs	2D Motif Basis	GZip	BZip	Jpeg 90	Jpeg 80	Jpeg 70
Lena	92.00	52.00	21.00	88.00	68.00	44.00	39.00	36.76
Satellite	92.68	60.97	36.58	56.09	46.34	39.02	31.70	29.26
Texture	80.36	42.39	29.69	43.74	33.54	38.25	35.8	34.6

Figura 3.6: risultati compressione delle immagini di Fig. 3.5 con metodi classici e con 2D Motif Basis.

Successivamente in Figura 3.7 e Figura 3.8, vengono riportati da [26] gli pseudo-codici degli algoritmi di codifica e decodifica delle immagini mediante 2D Motif.

COMPRESSION ALGORITHM
Input:
- an image I

Output:
- a multiset F of both pixels and pointers
- a set of patches $\mathcal{D}$

begin
1: **call** BASIS EXTRACTION on I and obtain $\mathcal{B} = \{M_1, M_2, \dots, M_k\}$
2: **call** DICTIONARY COMPUTATION on $\mathcal{B}$ and I and obtain $\mathcal{D} \subseteq \mathcal{B}$
3: $F \leftarrow I$
4: **for each** M_i in $\mathcal{D}$
begin
5: store in $\mathcal{D}$ the pair $< P_i, i >$ of the patch P_i corresponding to M_i and a pointer i
6: substitute in F each occurrence of M_i with the pointer i
end
end

Figura 3.7: algoritmo di compressione delle immagini.

DECOMPRESSION ALGORITHM
Input:
- a multiset F of both pixels and pointers
- a set of patches $\mathcal{D}$

Output:
- an image I

begin
1: **for every** patch P_i in $\mathcal{D}$
2: substitute the pointer i in F with P_i
3: $I \leftarrow F$
4: (Optional) interpolate I
end

Figura 3.8: algoritmo di decompressione delle immagini.

3.4 Algoritmo LZWA

L'algoritmo LZWA è introdotto in [4], esso è un adattamento di LZW orientato alla ricerca dei pattern con gap. In LZWA la parte iniziale del processo di analisi è praticamente identica a quella di LZW, eccetto che l'output tiene conto sia del riferimento alla frase sia dei caratteri che ne costituiscono le varie disambiguità. Questa tecnica è sviluppata attraverso l'uso di due dizionari, che chiameremo dizionario delle stringhe (il dizionario principale) e quello dei resolver (il dizionario ausiliario).

3.4.1 Codifica

Consideriamo come elementi del dizionario le stringe appartenenti all'alfabeto $\Sigma \cup \{_\}$, dove '_' rappresenta un carattere variabile. Inizializziamo i dizionari principale e ausiliario con i simboli di Σ. L'algoritmo procede come segue. Per trovare la succesiva frase da analizzare il primo passo è quello di posizionarsi sul generico indice i del testo. Verrà quindi ricercata la frase più lunga all'interno del dizionario principale che eguaglia quella del suffiso alla posizione i del testo. Sia s questa frase e h il suo indice nel dizionario. Utilizzeremo il puntatore di riferimento h per codificare la frase corrente. In output ogni frase analizzata verrà codificata da una coppia di puntatori che rappresenteno una frase primaria $s \in \Sigma \cup \{_\}$ del dizionario principale, e una frase secondaria $s_1 \in \Sigma$ appartenente al dizionario dei resolver. Lo pseudocodice di LZWA viene illustrato in Figura 3.9 [4]. Al momento della ricostruzione nella fase di decodifica l'informazione

Initializations
Initialize dictionary trie and resolver trie with the characters from Σ
Initialize phrase s to first input character, resolver s' to the empty word λ
Set $output \leftarrow \langle code(s), code(s') \rangle$
Body Repeat until no more input characters
    $\sigma \leftarrow$ read the next input character
    if $s\sigma$ is in the dictionary then set $s = s\sigma$ (*seek-phrase* continues)
    else, if $s_$ is in the dictionary and $s'\sigma$ is in resolvers
        then set $s = s_$, $s' = s'\sigma$ (*seek-phrase* continues)
        else (the end of a stored phrase has been reached)
            1- $output \leftarrow output \cdot \langle code(s), code(s') \rangle$
            2- if $s\sigma'$ for some $\sigma' \neq \sigma$ is in the dictionary then
                2a- if $s_$ is not already in dictionary add $s_$ to it
                2b- if $s'\sigma$ is not already in resolvers add $s'\sigma$ to it
               else add $s\sigma$ to dictionary
            3- $s \leftarrow \sigma$
end Body

Figura 3.9: pseudocodice algoritmo LZWA.

necessaria per una corretta combinazione delle due stringhe sono la posizione dei caratteri variabili presenti in s. Questa informazione in ogni caso non necessita di essere esplicitata perchè i caratteri variabili saranno riempiti con caratteri di s_1 in esatta sequenza, in modo che codifica e decodifica risultino coerenti. Prendiamo ad esempio la Figura 3.10 [4] che traccia l'analisi e la codifica della stringa "ababaacacbcaabcbbbacabbb". La prima stringa presa in considerazione

è la stringa “a”, essa viene trovata all’interno del dizionario principale nel nodo etichettato con l’indice 0, non a caso il dizionario è inizialmente popolato con tutti i caratteri dell’alfabeto Σ come figli della radice. Il dizionario viene allora espanso alla stringa “ab” a cui viene associato il codice 3 dato che i codici 1 e 2 corrispondono ai nodi “b” e “c”, e viene stampata in output la coppia $<0,\lambda>$, dove λ denota l’assenza di un resolver. La seconda frase in modo simile segue il percorso dal nodo “b”, il quale è etichettato come nodo 1. Tale nodo viene espanso al cammino “ba” etichettato con il puntatore 4, e viene stampata in output la coppia $<1,\lambda>$. La successiva stringa è “ab”, che viene trovata nell’albero al nodo 3, il quale viene espanso al cammino “aba”. A tale cammino viene assegnato il codice 5 e viene stampato in output il codice $<3,\lambda>$. Quando il suffisso aac viene letto all’interno del dizionario, viene creato un cammino attraverso un carattere variabile etichettato con il codice 6. La corrispondente coppia stampata in output è $<0,\lambda>$. La successiva frase è “ac”, questa frase raggiunge il nodo 6 utilizzando il cammino “a_” e il resolver “c”, il quale esiste nel dizionario dei resolver per inizializzazione. Il nodo 6 viene espanso in un nuovo nodo 7, seguendo il percorso “a_a” e viene emessa la coppia $<6,2>$ rappresentante il codice 6 dell’albero delle stringhe e il codice 2 dell’albero dei resolver. La successiva stringa è aba, la quale può usare il percorso verso il nodo 7 (“a_a”). Viene creato quindi un nuovo nodo etichettato con 8 che utilizza il percorso a_ab, in output viene stampato $<7,1>$. Il procedimento continua fino al termine della dei dati in input.

$\langle 0, \lambda \rangle$	$\langle 1, \lambda \rangle$	$\langle 3, \lambda \rangle$	$\langle 0, \lambda \rangle$	$\langle 6, 2 \rangle$	$\langle 7, 1 \rangle$	...
(1,0)	(10,3)	(6,2)	(12,3)	(14,2)		
a	b	ab	a	ac	aba	b...
ab	ba	aba	a-	a-a	a-ab	...
3	4	5	6	7	8	9..
				2(c)	1(b)	...

Figura 3.10: primi passi della codifica LZWA con input la stringa “ababaacabab..”.

3.4.2 Decodifica

La decodifica parte inizializzando i dizionari principale e secondario con gli elementi dell’alfabeto Σ. Ogni volta che viene letta una chiave dal file codificato vengono ricercati i puntatori all’interno dei dizionari. Se tali puntatori non sono

presenti i dizionari verranno aggiornati con nuove stringhe in corrispondenza alle successive posizioni disponibili. Consideriamo sempre la stringa “ababaacabab..” e facciamo un esempio di decodifica.

La prima chiave letta in input è $<0,\lambda>$, non ci sono resolver per questa chiave, mentre il puntatore 0 al dizionario principale indica la stringa “a”, che viene stampata in output. A questo punto viene letta in input la chiave $<1,\lambda>$, che anche questa volta non richiede nessun resolver, è possiede un puntatore alla stringa “b” del dizionario che viene stampata in output. La concatenazione della stringa fornita dalla precedente chiave con quella fornita dalla chiave attuale vanno a formare una nuova stringa “ab” nel dizionario principale che viene etichettata con la chiave 3. Quando ad esempio viene letta la chiave $<6,2>$, si verifica la presenza di un resolver, la stringa nel dizionario principale in corrispondenza a 6 è “a_” mentre il puntatore nell'albero secondario 6 corrisponde a “c”. Andremo a stampare in output la stringa “ac” fusione della due stringe in corrispondenza del carattere variabile. Il processo continua così sia per il dizionario principale sia per quello ausiliario fino alla fine del file codificato.

3.4.3 Aspetti strutturali

Ci sono O(n^3) possibili pattern con gap in un testo di n caratteri, ma la strategia di analisi proposta prende in considerazione solo un sottoinsieme lineare di questi. L'algoritmo LZWA lavora in tempo e spazio lineare nell'analisi del testo in input. La struttura del dizionario principale eccetto per la radice (ma solo in alcuni casi) è un albero binario. Ogni nodo interno possiede o un arco etichettato con un carattere di Σ, o due archi, uno etichettato con un carattere di Σ e l'altro etichettato con un carattere variabile. La dimostrazione di quest'ultima caratteristica è la seguente. Sia $\bar{x}$ il suffisso del dato che deve ancora essere codificato. La ricerca di un nuovo pattern può essere spezzata in tre stadi. Nel primo stadio viene cercato all'interno dell'albero il percorso più lungo che coincide con un prefisso di $\bar{x}$. Questa ricerca può terminare in due differenti modi, a seconda che l'ultima transizione abbia raggiunto una foglia o un nodo interno:

- Nel caso in cui venga raggiunta una foglia ξ la ricerca fallisce dato che il percorso non può continuare, a questo punto viene aggiunto un nuovo nodo, estendendo il dizionario con un nuovo arco etichettato con $\sigma \in \Sigma$.

- Nel caso in cui viene raggiunto un nodo interno υ , non possiamo che avere simultaneamente che un patterns “s_” e un resolvers “$s^1\sigma$”.

Se la ricerca termina per il fatto che “s_” non è nel dizionario allora induttivamente v possiede un figlio, e il corrispondente arco deve essere etichettato da qualche $\sigma^1 \neq \sigma$. L'algoritmo aggiunge a v un arco etichettato con “_”, il quale rende v un nodo binario, e predispone un resolver “$s^1\sigma$” per la futura decodifica. D'altro canto, se l'arresto dell'algoritmo avviene per il fatto che “$\sigma^1 \neq \sigma$” non è presente nel dizionario dei resolver, allora induttivamente v possiede già due figli, uno etichettato con $\sigma^1 \neq \sigma$ e l'altro con “_”. L'algoritmo si limita quindi ad aggiungere “s1” nell'albero dei resolver. LZWA utilizza il dizionario dei resolver indipendentemente dalla stringa che lo ha generato, questo tende a migliorare il tasso di compressione di LZW anche se d'altro canto l'algoritmo necessita di due puntatori per pattern, quindi una chiave mediamente occupa più spazio. Aver diminuito il numero di figli del dizionario a 2 rispetto a LZW porta ad un tasso di compressione maggiore del file originario.

3.4.4 Modifiche e miglioramenti

Riportiamo alcune modifiche possibili su LZWA descritte in [4]. Queste modifiche non alterano le prestazioni temporali dei tempi di codifica e decodifica. Per iniziare, la condizione che i gap di ogni nuova frase devono essere riempiti con un pattern resolver potrebbe essere eliminata nella compressione lossy: tale condizione potrebbe rallentare notevolmente il processo di crescita del vocabolario, il quale può costituire un indesiderato problema. Altre varianti sono le seguenti:

- Una codifica più efficiente dei pattern resolver può essere sviluppata. Dato che il vocabolario dei resolver cresce a volte indipendentemente da quello principale, potrebbe capitare che un resolver r^1 sia il prefisso di un altro resolver che chiamiamo r^2 già presente nell'albero dei resolver. Durante la decodifica, la lunghezza di r^1 può essere dedotta dalla struttura della frase s così che un puntatore a r^2 è sufficiente per recuperare r^1. Questo suggerisce di assegnare dinamicamente codifiche distinte soltanto alle foglie del dizionario dei resolver, il quale sfrutta il concetto di ereditarietà: ogni volta che un nodo viene espanso creando una foglia, la foglia eredita la codifica di tale nodo. Quando da una foglia si ramifica un nodo, un nuovo codice viene introdotto e assegnato a tale foglia. Da questi concetti, se ora chiamiamo $\mathrm{ext}(r^1)$ una delle più lunghe estensioni presenti di resolver r^1, allora la linea di codifica nello pseudo codice di LZWA può essere riscritta come:

$$\text{output} \leftarrow \text{output} < \text{code(s)}, \text{code}((\text{ext}(r^1)) >$$

- Limite superiore al numero di errori presenti in una frase. Modificare l'al-

goritmo in modo tale da limitare il numero k degli errori permessi in ogni frase non è compito difficile. In particolare nella versione lossy dell'algoritmo dove i gap non vengono riempiti, la riduzione del numero di frasi è generale e drastica in alcuni casi.

- Rendere ogni frase un consenso. In questa variante dell'algoritmo bisogna essere certi che ogni frase utilizzata nella codifica è una sottostringa del consenso di due suffissi del file sorgente. Per questo bisogna tenere conto durante la codifica del dizionario di pattern solidi risultanti dalla combinazione di frasi e resolver visti in precedenza. Definendo con $s \ominus s_1$ la frase che corrisponde alla combinazione della frase s e del resolver s_1, la fase di ricerca viene condizionata in tal modo:

  if ("s_" è nel dizionario principale)

  &

  (s^1 è nel dizionario dei resolver)

  &

  ($s \ominus s^1$ è nel dizionario)

- Costruire un dizionario di ricerca h-esimo con $h > 2$. Questo consiste nell'emissione di un carattere variabile non la seconda ma l'h-esima volta che un nodo di dizionario viene raggiunto. Questo porta alla costruzione di un dizionario principale di dimensioni maggiori e ad un dizionario dei resolver di dimensioni minori. Questo permette di controllare la fedeltà nella variante lossy dell'algoritmo dove i resolvers sono trascurati, e di determinare la migliore performance lossless cambiando il parametro h.

3.4.5 Dinamica di Pattern con gap e Resolver

Verranno ora illustrati gli andamenti di pattern con gap e resolver descritti in [27] al variare del numero di figli per nodo nei dizionari con le performance di classificazione in LZWA. Ci serviamo delle formule matematiche descritte in [27] per la comprensione degli andamenti. Data una stringa $x \in \Sigma^+$, sia $C(x)$ l'output di LZWA con input x, siano $D_m(x)$ e $D_f(x)$ i dizionari dei pattern e dei resolver associati durante la codifica da LZWA a x, e sia $D(x) \in \{D_m(x), D_f(x)\}$ un simbolo che indichi generalmente uno dei due dizionari di x. Definiamo $D^*(x)$ l'insieme delle stringhe di $D(x)$ che non sono il prefisso di nessun altra stringa

in $D(x)$ e $D^*(x,y)$ l'insieme delle stringhe di lunghezza massima in $D(x) \cap D(y)$. Data una stringa $s \in D(x)$ indichiamo con $l(s)$ la sua lunghezza e con $f_x(s)$ il numero di volte che viene ripetuta in $C(x)$. Per poter condurre l'analisi si sono utilizzati strumenti di misura basati sulla composizione di sottostringhe. Questi sono la metrica di Jaccard d_j tra $D(x)$ e $D(y)$ e la sua variante, d_{j^*} definite da:

$$d_j(x,y) := 1 - \frac{|D(x) \cup D(x)|}{|D(x) \cap D(x)|}$$

$$d_{j^*}(x,y) := 1 - \frac{|D^*(x) \cup D^*(x)|}{|D^*(x) \cap D^*(x)|}$$

Ci siamo serviti della metrica coseno d_c tra vettori le cui componenti sono stringhe appartententi a $D(x) \cup D(x)$ a cui sono associati i valori di lunghezze $f_x(s)$, $f_y(s)$.

$$d_c := \frac{x \times y}{\|x\| \times \|y\|}$$

Definiamo inoltre:

$$d_h(x,y) := \frac{1}{2}\left(\frac{nlog(n) + mlog(m)}{\sum\limits_{s \in D^*(x,y)} l(s)} - \frac{nlog(n)}{\sum\limits_{s \in D(x)} l(s)} - \frac{mlog(m)}{\sum\limits_{s \in D(y)} l(s)}\right)$$

dove si definiscono $n := |D(x)|$ e $m := |D(y)|$ per lo studio delle stringhe di lunghezza massima in $D(x) \cap D(y)$. Ci serviamo infine della distanza di compressione normalizzata d_{ncd} definita da:

$$d_{ncd} := \frac{max\{C(x|y), C(x|y)\}}{max\{C(x), C(y)\}} \approx \frac{min\{C(xy), C(yx)\} - min\{C(x), C(y)\}}{max\{C(x), C(y)\}}$$

Definiamo nel seguito con α il numero di figli per un generico nodo del dizionario. Per l'analisi degli andamenti di pattern e resolver prendiamo in considerazione il dataset S_1 formato considerando k stringhe per ogni elemento di differenti dataset elencati in [27].

L'algoritmo LZWA utilizza i resolver del dizionario ausiliario a prescindere dai pattern che sono utilizzati nella codifica. Questo aumenta la compressione rispetto a LZW anche se una chiave occupa più spazio nel file compresso (due puntatori rispetto ad uno). Nella Figura 3.11 [27] viene mostrato come cambiano alcune caratteristiche al variare di α per il dataset S_1. Quando α scende circa sotto

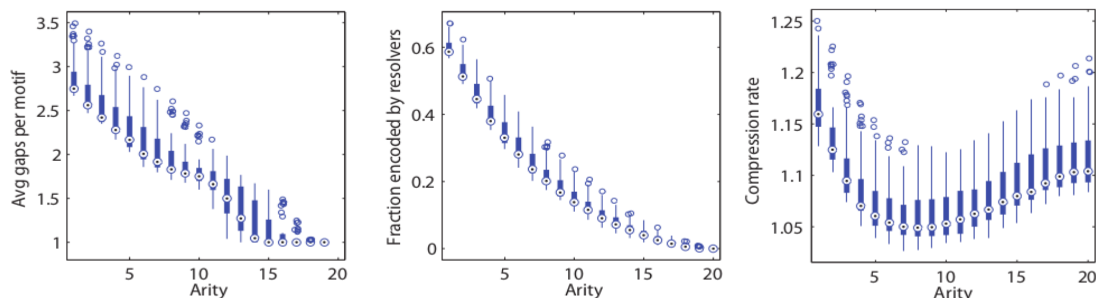

Figura 3.11: per il dataset S_1 al variare di α abbiamo il numero medio di gap per pattern, la porzione di dataset codificata dai resolver ed il livello di compressione.

il valore 8 il tasso di compressione aumenta e quello di LZW viene superato con α pari a 1, 2. I grafici mostrano che l'utilizzo indipendente dei resolver in LZWA influenza le misure di dissimilarità (Figura 3.12 [27]). Per ogni metrica $d \in \{d_j, d_{j^*}, d_c, d_h\}$ esiste un numero di figli per nodo $\bar{\alpha} < \Sigma$ tale che per tutti i valori $\alpha \geq \bar{\alpha}$ le performance delle metriche d applicate a $D_m(x)$, $D_m(y)$ sono confrontabili con quelle di LZW. Per il dataset S_1 $\bar{\alpha} \approx 7,8$, questo sta ad indicare che possiamo ignorare circa il 20% del dataset originale e ottenere ancora una rappresentazione comparabile con quella basata su LZW. Le performance degenerano con pendenza monotona quando $\alpha < \bar{\alpha}$, i resolver seguono una curva complementare. Possiamo notare che d_c e d_h non portano a grossi vantaggi in prestazioni rispetto d_j nonostante contengano informazioni aggiuntive sulla lunghezza e sulla frequenze dei pattern resolver. Aggiungendo a questo il fatto che le performance di d_{j^*} e d_j sono confrontabili porta a pensare che l'insieme dei pattern più lunghi presenti nel dizionario contengano le informazioni essenziali per la classificazione. I risutlati delle misure per D_m e D_f in termini di classificazione non permettono nussun tipo di stima sul comportamento di d_{dnc}. Ad esempio pensiamo di comprimere la stringa y utilizzando sia $D_m(x)$ che $D_f(x)$ e assumendo che $D_m(x) \approx Dm(y)$ e $D_f(x) \approx D_f(y)$. Con un valore di α abbastanza grande i pattern in $D_m(x)$ sono simili a quelli in $D_m(y)$ nonostante $D_f(x)$ e $D_f(y)$ possono essere incorrelati. Questo implica che la lunghezza media di una stringa che eguaglia il dizionario è molto grande e che che le frasi nel file compresso tendono ad estendersi lungo sottostringhe portando ad una buona compressione. D'altro canto i resolver in $D_f(y) \setminus D_f(x)$ tendono ad avere codici molto grandi perchè vengono analizzati successivamente a quelli di $D_f(y)$, e i pochi resolver in $D_f(x) \cup D_m(x)$ hanno una frequenza che non riflette quella di $D_f(y)$ iniettando rumore nella parte di resolver in output. Con un basso valore per α la porzione di output che riflette D_m è influenzata da rumore. Anche le specifiche di implementazione del compressore possono influenzare d_{dnc}: ad esempio, la lunghezza delle parole di codice nel file compresso riflettono solo debolmente la loro frequenza nel file stesso.

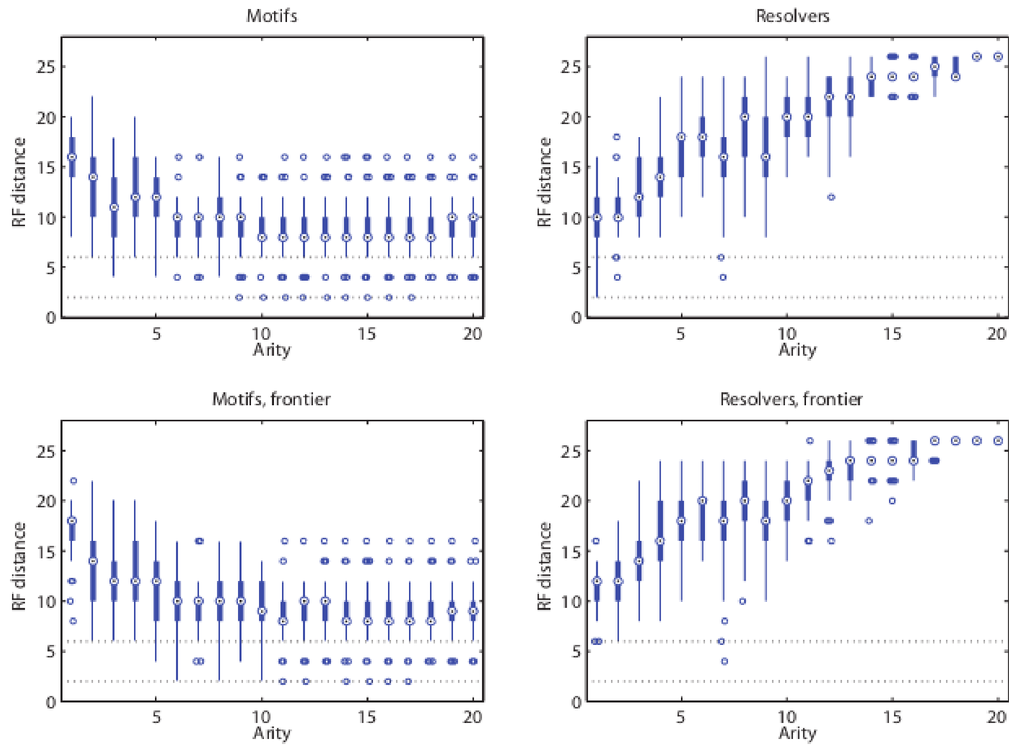

Figura 3.12: Classificazione delle performance di d_j (in alto) e d_{j^*} (in basso) su D_m (destra) and D_f (sinistra). I grafici riassumono 50 campioni e le linee tratteggiate indicano il 25 ° e il 75 ° percentile della performance raggiunta da d_{ncd} quando viene usato come compressore GZip.

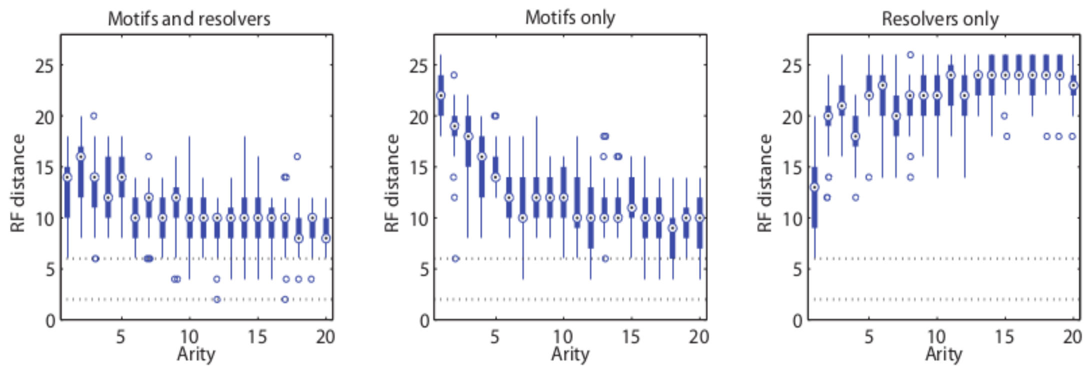

Figura 3.13: performance in termini di classificazione per d_{dnc} utilizzando la distanza di Robinson-Foulds (RF) [27] quando $D_m(y)$ viene inizializzato con $D_m(x)$ e $D_f(y)$ viene inizializzato con $D_f(x)$(sinistra); quando solo $D_m(y)$ viene inizializzato con $D_m(x)$ (in centro); quando $D_f(y)$ viene inizializzato con $D_f(x)$ (a destra). I grafici riassumono 36, 28 e 28 campioni.

Questo introduce un ulteriore componente di rumore che potrebbe ulteriormente abbassare la qualità della classificazione. I grafici in Figura 3.13 [27] mostrano che d_{dnc} è robusta ad entrambi tali sorgenti di rumore: quando sono presenti D_m o D_f le performance di classificazione hanno un trend simile a misure su un dizionario corrispondente. Quando invece sono presenti sia D_m che D_f queste performance tendono a migliorare per $\alpha < \bar{\alpha}$ e restano invece uguali per $\alpha \geq \bar{\alpha}$.

4 Esperimenti

In questo capitolo verranno illustrati alcuni esperimenti di compressione applicati su tre diversi tipi di dataset: Rfam [28], Kernel Linux [29], XML [29]. Verranno processati tutti con una versione sviluppata nel linguaggio c++ dell'algoritmo LZWA descritto nel capito precedente. I risultati ottenuti per ogni dataset saranno confrontati con algoritmi classici come GZip e 7Zip, per fornire informazioni sulla comprimibilità di tali dataset mediante pattern con gap.

4.1 Obiettivi e setting degli esperimenti

L'obiettivo degli esperimenti riportati in seguito è quello di studiare il livello di comprimibilità con gap di alcuni dataset. In particolare i nostri esperimenti utilizzano diversi tipi di codifiche dati descritte in seguito. Per la comprensione dei dettagli tecnici si introduce di seguito la notazione utilizzata: dato uno dei dataset in analisi, identifichiamo con f il file dati che lo rappresenta, $LZW_A(f)$ il risultato dell'applicazione dell'algoritmo LZWA sul file dati f, p_i un puntatore all'albero principale dei patterns, r_i un puntatore all'albero dei resolver, $\mathbf{z}(f)$ uno uno degli algoritmi GZip o 7Zip applicati a f, Σ l'alfabeto dei simboli utilizzati e σ un elemento di Σ, B($< p_i, r_i >$) la rappresentazione binaria della coppia di puntatori p_i e r_i. Gli esperimenti svolti su ogni dataset sono 4:

1. Su ogni file compresso da LZWA vengono applicati gli algoritmi $\mathbf{z}$. Vengono confrontate le dimensioni di $LZW_A(f)$ con quelle di $\mathbf{z}(LZW_A(f))$. Questo ha permesso di sfruttare le euristiche sui puntatori che nel codice di LZWA non venivano implementate, e di eliminare possibili ripetizioni di pattern, resolver, e coppie <pattern,resolver> in $LZW_A(f)$.

2. Per ciascun pattern o resolver S = $[\sigma_1,\sigma_2,\sigma_3, \dots ,\sigma_{k-1}]$ generato dall'encoder

viene assegnata la codifica S = [b(σ_1), b(σ_2), b(σ_3), ... , b(σ_{k-1})] , dove b(σ_i) identifica la rappresentazione binaria del generico simbolo σ_i di Σ in un numero di bit pari a $\lceil log_2(|\sigma| + 1)\rceil$. In questo modo il file originale f viene espanso in un file f^1 di dimensione dim $f^1 \leq 2\times$dim(f), che trasforma una certa classe di ridondanze con gap in ridondanze di sottostringhe che GZip e 7Zip sono ora in grado di rilevare. Si procede quindi a calcolare la dimensione di $\mathbf{z}(LZW_A(f))$.

3. Quando LZWA utilizza una coppia $< p_i,r_i >$, viene istituito un collegamento tra i corrispondenti nodi dell'albero principale e ausiliario. I passi per eseguire questo esperimento, suddivisi in una struttura del tipo if-elseIf-else sono i seguenti:

   - Se p_i non è mai stato stampato in output, oppure se r_i non è mai stato stampato in output, viene definita B($< p_i,r_i >$) = $'' < p_i,r_i >''$ e viene stampata in output. L'obiettivo della codifica è quello di salvare in output tutti i pattern e resolver distinti in modo che lo string compressor **z** possa rappresentare ciascuno di essi con un solo puntatore.

   - Altrimenti, se la coppia $< p_i,r_i >$ è già stata stampata in output con rappresentazione binaria B($< p_i,r_i >$), viene stampata B($< p_i,r_i >$). Questo permettere a **z** di rappresentare con un solo puntatore coppie ripetute.

   - Altrimenti, nè p_i nè r_i appaiono per la prima volta in output, ma la coppia $< p_i,r_i >$ appare per la prima volta in output. Assegnamo alla coppia $< p_i,r_i >$ la rappresentazione binaria B($< p_i,r_i >$) = B($< p_i^2,r_i^2 >$)+($'' < p_i^1,r_i^1 > ''$) e la stampiamo in output. In tale rappresentazione indichiamo con + l'operatore di concatenazione, p_i^1 e r_i^1 siano i suffissi di p_i e r_i , p_i^2 sia l'antenato più profondo di p_i che è già stato stampato in output insieme ad un antenato r_i^2 di r_i , $p_i = p_i^2 + p_i^1$, $r_i = r_i^2 + r_i^1$. L'obiettivo è quello di permettere a **z** di rappresentare con un solo puntatore stringhe (senza gap) ripetute nel file originale, prima potenzialmente scomposte in coppie <pattern,resolver> distinte.

4. Viene stampata in output la stringa $S_d = p_0p_1...p_{k-1}r_0r_1...r_{h-1}$, dove i p_i sono tutti i patterns distinti usati da LZWA nel comprimere il file f, e gli r_i sono tutti i resolvers distinti usati da LZWA nel comprimere il file f. Viene calcolata infine la dimensione di S_d e di $\mathbf{z}(S_d)$. L'obiettivo è quello di verificare che la dimensione non-compressa dei dizionari nel codice implementato (o in

altre parole, la somma delle lunghezze delle prime occorrenze di tutti i pattern e resolver nell'output di LZWA) è una parte minima dell'intero output di LZWA.

4.2 Risultati su sequenze di RNA

I risultati ottenuti sul dataset Rfam in esame (versione 11.0) sono riportati nei successivi grafici mentre ulteriori risultati congiunti vengono riportati a fine capito. L'esperimento 1 è stato eseguito con due differenti codifiche per i puntatori, ovvero a 24 e 32 bit. Tali codifiche hanno portato a generare un file di dimensioni ridotte rispetto all'originale nel primo caso (circa 10MB in meno), e un file espanso nel secondo caso. Quest'ultimo risultato deriva dal fatto che i puntatori richiesti dalla codifica non superavano in dimensione un numero a 24 bit producendo perdita di informazione. Nei seguenti esperimenti si è allora deciso di limitare il primo esperimento ad un numero di bit pari a solo 24.
In Figura 4.1 viene riportata la composizione del file compresso tramite LZWA. I dati si ripartiscono come segue: 8.9% dizionario, 36.7% coppie distinte e 54.4% coppie ripetute. Come possiamo osservare il dizionario ricopre un piccola porzione del file, questo significa che il file codificato è composto da un numero di pattern e resolver relativamente piccolo. Pattern e resolver sono inoltre presenti in quantità simili e la loro lunghezza in media è di 14 byte. I resolver sono poco più numerosi dei pattern (25983 rispetto a 30142), il loro rapporto viene riportato in Figura 4.16. Questa composizione favorisce un numero molto alto di coppie ripetute (il 54.4%). Tali coppie forniscono un tipo di ridondanza che i classici **z** dovrebbero essere in grado di rilevare. Dopo aver eseguito l'esperimento 2, il file risultante mostra una dimensione circa pari a 87MB, come previsto c'è stata un'espansione rispetto al file originale dato che sono state stampate direttamente le coppie <pattern, resolver> utilizzate. Questo dato ha permesso di risalire alla reale dimensione dell'insieme di coppie pattern resolver utilizzate nello stadio di codifica. I risultati ottenuti (Figura 4.3) nella compressione di quest'ultimo però non sono stati soddisfacenti, entrambi gli algoritmi GZip e 7Zip non hanno portato ad un file di dimensioni ridotte rispetto al file di partenza compresso direttamente con GZip e 7Zip i cui dati sono riportati in Figura 4.2. Il file Rfam risulta essere il meno comprimibile tra i dataset analizzati (di un fattore circa 6) come mostrato in Figura 4.19. Se avessimo ottenuto un file di dimensioni minori avremmo potuto confermare che un eventuale preprocessing con LZWA sarebbe servito per ottenere una compressione migliore con i classici algoritmi. L'esperimento 4 ha permesso di risalire alla dimensione del dizionario di pattern e resolver utilizzato nella codifica. Il dizionario è risultato essere di circa 8MB. Tale dizionario viene confrontato con

Composition of expanded LZWA file
Rfam version 11.0

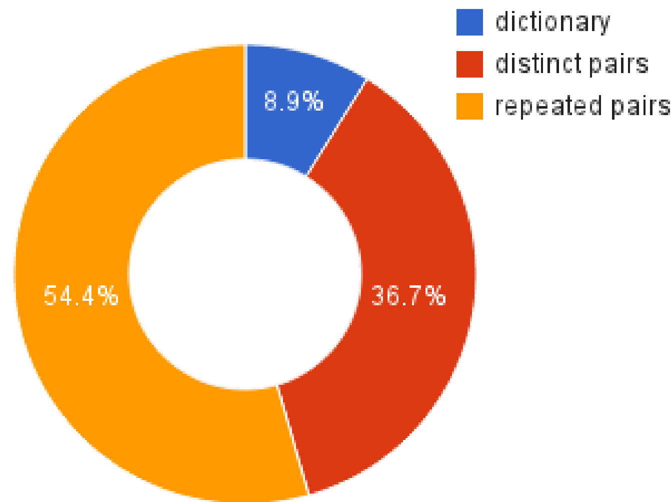

Figura 4.1: statistiche di LZWA sul dataset Rfam, in giallo abbiamo le coppie ripetute, in rosso abbiamo le coppie distinte mentre in blu il dizionario utilizzato.

Dataset		GZip		7Zip	
description	size (KB)	options	size (KB)	options	size (KB)
Rfam version 11.0, downloaded Oct 30 2013, no headers.	54087	–best	11100	-mx9	8700

Figura 4.2: risultati degli algoritmi GZip e 7Zip applicati direttamente sul dataset Rfam. In prima colonna abbiamo la descrizione del dataset in esame e la dimensione di quest'ultimo. A seguire, ripetendosi prima per GZip e poi per 7Zip abbiamo: opzioni di compressione per l'algoritmo e risultato compressione sul dataset.

	LZWA + GZip			LZWA + 7Zip		
options	size (gzip, KB)	dictionary size (KB)	size - dictionarySize (KB)	size (7zip, KB)	dictionary size (KB)	size - dictionarySize (KB)
24 bits per pointer	20600			15500		
32 bits per pointer	21900			14600		
Pointers encoded as strings, 8 bits per symbol. Experiments E2, E4.	20300	1983	18317	15700	1640	14060
Experiments E3.	30858			18918		

Figura 4.3: risultati dell'applicazione in coda degli algoritmi GZip e 7Zip sui file generati da LZWA (esperimenti 2 e 3) per il dataset Rfam. In prima colonna abbiamo le opzioni sugli esperimenti, a seguire, prima per GZip e poi per 7Zip abbiamo: dimensione del file compresso, dimensione dizionario, differenza tra la dimensione del file compresso e quella del dizionario.

	LZWA									
options	distinct patterns	distinct resolvers	avg bytes per pattern/resolver	distinct pairs	size (KB)	dictionary size (KB)	size - dictionarySize (KB)	dictionary size with naive pointers (KB)	distinct pairs size (KB)	repeated pairs size (KB)
24 bits per pointer					45100					
32 bits per pointer					60200					
Pointers encoded as strings, 8 bits per symbol. Experiments E2, E4.	252983	300142	14.3327602259887	513104	87400	7742	79658	1220.932688176	32069	47589
Experiments E3.					91488					

Figura 4.4: risultati degli esperimenti 2 e 3 condotti da LZWA sul dataset Rfam. In prima colonna è presente la descrizione delle opzioni sugli esperimenti, a seguire abbiamo: numero di pattern distinti, numero di resolver distinti, lunghezza media in byte di pattern e resolver, numero di coppie distinte utilizzate, dimensione del file in questione, dimesione del dizionario utilizzato, differenza tra dimesione file utilizzato e dimensione dizionario, dimensione del dizionario di puntatori, dimensione coppie distinte, dimensione coppie che ripetute.

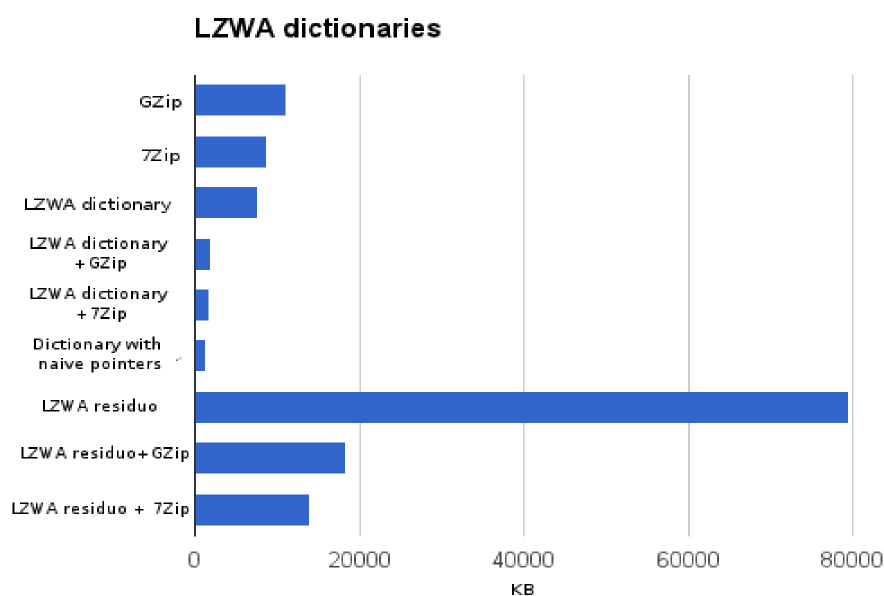

Figura 4.5: dimensioni dizionari del dataset Rfam, nella legenda a sinistra partendo dall'alto abbiamo: dimensione dizionario GZip, dizionario 7Zip, dizionario LZWA, dizionario LZWA+GZip, dizionario LZWA+GZip, dizionario dei puntatori, dizionario residuo di LZWA, dizionario residuo+GZip, dizionario residuo+7Zip

altri dizionari elencati in Figura 4.5 dove il dizionario di dimensione maggiore indica la differenza dell'insieme totale di pattern e resolver utilizzati (dimensione del file dell'esperimento 2) e del dizionario effettivamente utilizzato (risultato dell'esperimento 4). In Figura 4.21 vengono riportati gli effetti dell'utilizzo di puntatori come interi e come stringhe, i primi risultano evidentemente meno dispendiosi in termini di spazio rispetto ai secondi. Infine per quanto riguarda il terzo esperimento, sembra che il tentativo di permettere ai compressori **z** di accedere ad eventuali prefissi presenti nelle ripetizioni dei pattern e dei resolver non abbia portato a buon fine dato che la compressione non ha comportato grandi vantaggi come mostrato in Figura 4.19.

4.3 Risultati su Linux Kernel

In questa serie di esperimenti è stata presa in considerazione la concatenazione di una parte di file sorgenti del Kernel Linux della dimensione di circa 50MB. L'esperimento 1 come accennato in precedenza viene svolto utilizzando solamente una codifica a 24 bit per i puntatori. Tale codifica porta al risultato di un solo file compresso di circa 27MB per GZip e 23MB per 7Zip. La composizione del file espanso con LZWA viene riportata in Figura 4.6. Nel figura del file espanso possiamo notare che la fetta blu corrispondente al dizionario occupa il 52.4% del file originale. Il rapporto tra resolver e pattern (Figura 4.16) indica che il file linux contiene il maggior numero di resolver rispetto ai pattern di tutti i dataset studiati. Il numero di resolver è circa venti volte più grande di quello dei pattern: questo sta ad indicare che il file è stato generato utilizzando un numero di pattern piccolo in termini di confronto con i resolver che li completano. La sequenza di coppie <pattern,resolver> è comunque abbastanza comprimibile (di un fattore 7) rispetto al compressore 7Zip come riportato in Figura 4.19, un alto valore di comprimibilità rispecchia il fatto che comunque il numero di coppie ripetute nel file è elevato (il 31% del file espanso). Questo può essere un indizio sul fatto che Linux contiene molte sottostringhe ripetute di lunghezza confrontabile con una coppia, che **z** è in grado di sfruttare anche senza LZWA.
L'esperimento 2 come nel caso precedente ha portato ad un risultato indesiderato visto che il file compresso con i classici compressori **z** non ha restituito un file di dimensioni minori, neanche in questo caso il preprocessing con LZWA ha avuto un buon esito.
Per quanto rigurada l'esperimento 3 invece possiamo notare che l'utilizzo della grande ridondanza di prefissi in pattern e resolver in questo caso produce in un risultato simile alla compressione classica senza però migliorarla (Figura 4.20). In Figura 4.10 risulta evidente come il dizionario di pattern resover utilizzato da

LZWA sia di dimensioni simili al dizionario residuo che compone il file espanso.

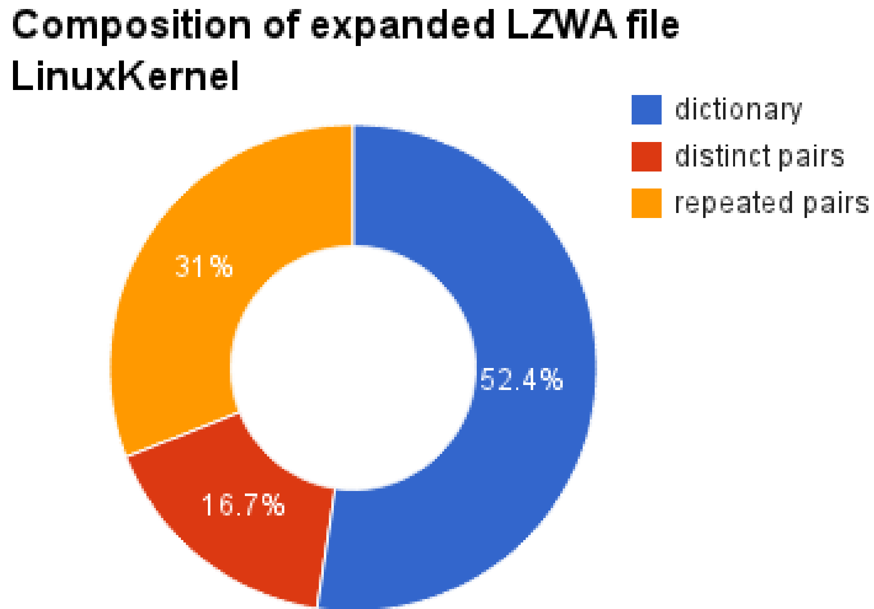

Figura 4.6: statistiche di LZWA sul dataset Linux Kernel, in giallo abbiamo le coppie ripetute, in rosso abbiamo le coppie distinte mentre in blu il dizionario utilizzato.

Dataset		gzip		7zip	
description	size (KB)	options	size (KB)	options	size (KB)
Linux Kernel	52428	–best	12211	-mx9	8403

Figura 4.7: risultati degli algoritmi GZip e 7Zip applicati direttamente sul dataset Linux Kernel. In prima colonna abbiamo la descrizione del dataset in esame e la dimensione di quest'ultimo. A seguire, ripetendosi prima per GZip e poi per 7Zip abbiamo: opzioni di compressione per l'algoritmo e risultato compressione sul dataset.

	LZWA + GZip			LZWA + 7Zip		
options	size (gzip, KB)	dictionary size (KB)	size - dictionarySize (KB)	size (7zip, KB)	dictionary size (KB)	size - dictionarySize (KB)
24 bits per pointer	20600			15500		
32 bits per pointer	21900			14600		
Pointers encoded as strings, 8 bits per symbol. Experiments E2, E4.	20300	1983	18317	15700	1640	14060
Experiments E3.	30858			18918		

Figura 4.8: risultati dell'applicazione in coda degli algoritmi GZip e 7Zip sui file generati da LZWA (esperimenti 2 e 3) per il dataset Linux Kernel. In prima colonna abbiamo le opzioni sugli esperimenti, a seguire, prima per GZip e poi per 7Zip abbiamo: dimensione del file compresso, dimensione dizionario, differenza tra la dimensione del file compresso e quella del dizionario.

	LZWA									
options	**distinct patterns**	**distinct resolvers**	**avg bytes per pattern/resolver**	**distinct pairs**	**size (KB)**	**dictionary size (KB)**	**size - dictionarySize (KB)**	**dictionary size with naive pointers (KB)**	**distinct pairs size (KB)**	**repeated pairs size (KB)**
24 bits per pointer					45100					
32 bits per pointer					60200					
Pointers encoded as strings, 8 bits per symbol. Experiments E2, E4.	252983	300142	14.3327602259887	513104	87400	7742	79658	1220.932688176	32069	47589
Experiments E3.					91488					

Figura 4.9: risultati degli esperimenti 2 e 3 condotti da LZWA sul dataset Linux Kernel. In prima colonna è presente la descrizione delle opzioni sugli esperimenti, a seguire abbiamo: numero di pattern distinti, numero di resolver distinti, lunghezza media in byte di patter e resolver, numero di coppie distinte utilizzate, dimensione del file in questione, dimesione del dizionario utilizzato, differenza tra dimesione file utilizzato e dimensione dizionario, dimensione del dizionario di puntatori, dimensione coppie distinte, dimensione coppie che ripetute.

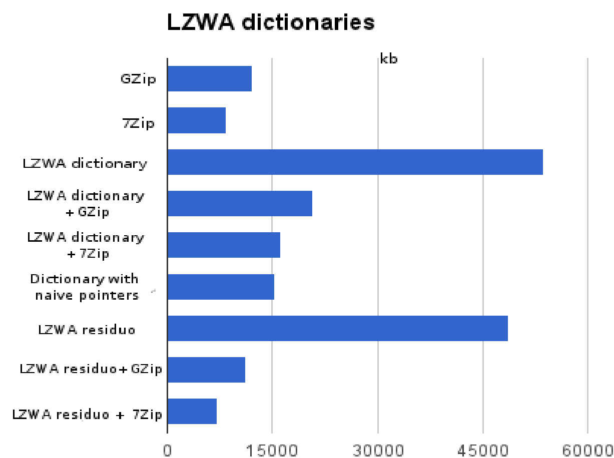

Figura 4.10: composizione dizionari file Linux Kernel, nella legenda a sinistra partendo dall'alto abbiamo: dizionario GZip, dizionario 7Zip, dizionario LZWA, dizionario LZWA+GZip, dizionario LZWA+GZip, dizionario dei puntatori, dizionario residuo di LZWA, dizionario residuo+GZip, dizionario residuo+7Zip

4.4 Risultati su file XML

Gli esperimenti successivi vengono eseguiti su un file sorgente XML di circa 50MB in cui vengono elencate informazioni bibliografiche riguardanti le principali riviste informatiche.
L'esperimento 1 a 24 bit ha portato in questo caso a generare un file espanso di circa 42MB. Nel file espanso il dizionario è come in linux molto grande, comprende infatti il 50.3% del file, mentre le coppie ripetute e quelle distinte occupano rispettivamente il 16.1% e 33.6% del file (Figura 4.11).

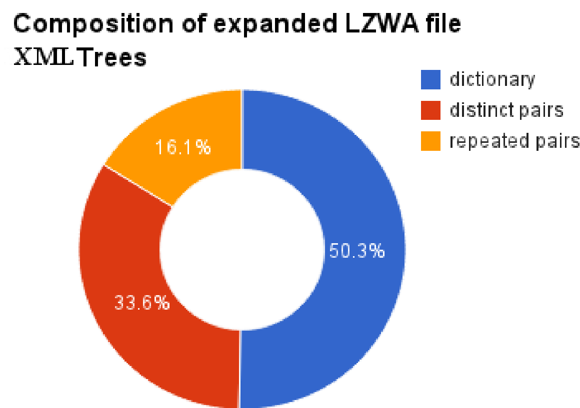

Figura 4.11: statistiche di LZWA sul dataset XML, in giallo abbiamo le coppie ripetute, in rosso abbiamo le coppie distinte mentre in blu il dizionario utilizzato.

In questo caso i resolver sono molti rispetto ai pattern (circa 6 volte tanti) come riportato in Figura 4.16. Questi valori indicano che il numero di pattern presenti nel file codificato è relativamente piccolo rispetto al numero di resolver che li completano. La Figura 4.19 mostra che il sorgente XML risulta essere il più comprimibile tra i dataset analizzati. Questo dato confrontato con la composizione del file espanso mostra che probabilmente non sono le coppie pattern resolver ripetute a generare la comprimibilà ma la presenta di sequenze di coppie ripetute. Questa può essere una caratteristica che indica la presenza nel sorgente XML di molte sottostringhe ripetute più lunghe di una generica coppia pattern-resolver. Questo può essere utilizzato dal generico compressore **z** per ottenere risultati migliori di LZWA. Il tentativo dell'esperimento 3 in questo caso è stato negativo dato che la ridondanza di prefissi non è stata utilizzata da **z** per migliorare la compressione. Similmente al caso precedente possiamo osservare come l'esperimento 4 mostra in Figura 4.15 che il dizionario utilizzato da LZWA è in dimensione simile al dizionario residuo (differenza tra il dizionario di LZWA e la dimensione del file). I valori numerici rispettivamente associati agli esperimenti 1,2,3 e 4 sono riportati

in dettaglio nella Figura 4.14, mentre i risultati dell'applicazione diretta e in coda di GZip,7Zip sono riportati rispettivamente in Figura 4.12 e 4.13.

Dataset		gzip		7zip	
description	size (KB)	options	size (KB)	options	size (KB)
XML trees	52428	–best	9034	-mx9	5957

Figura 4.12: risultati degli algoritmi GZip e 7Zip applicati direttamente sul dataset XML. In prima colonna abbiamo la descrizione del dataset in esame e la dimensione di quest'ultimo. A seguire, ripetendosi prima per GZip e poi per 7Zip abbiamo: opzioni di compressione per l'algoritmo e risultato compressione sul dataset.

	LZWA + GZip			LZWA + 7Zip		
options	size (GZip, KB)	dictionary size (KB)	size - dictionarySize (KB)	size (7Zip, KB)	dictionary size (KB)	size - dictionarySize (KB)
24 bits per pointer	15804			13488		
Pointers encoded as strings, 8 bits per symbol. Experiments E2, E4.	22616	13770	8846	14824	10512	4312
Experiments E3.	12218			6592		

Figura 4.13: risultati dell'applicazione in coda degli algoritmi GZip e 7Zip sui file generati da LZWA (esperimenti 2 e 3) per il dataset XML. In prima colonna abbiamo le opzioni sugli esperimenti, a seguire, prima per GZip e poi per 7Zip abbiamo: dimensione del file compresso, dimensione dizionario, differenza tra la dimensione del file compresso e quella del dizionario.

	LZWA									
options	distinct patterns	distinct resolvers	avg bytes per pattern/resolver	distinct pairs	size (KB)	dictionary size (KB)	size - dictionarySize (KB)	dictionary size with naive pointers (KB)	distinct pairs size (KB)	repeated pairs size (KB)
24 bits per pointer					19294					
Pointers encoded as strings, 8 bits per symbol. Experiments E2, E4.	426046	2860918	101.92	453414	84320	42404	41916	8462.94	28338.375	13577.625
Experiments E3.					37158					

Figura 4.14: risultati degli esperimenti 2 e 3 condotti da LZWA sul dataset XML. In prima colonna è presente la descrizione delle opzioni sugli esperimenti, a seguire abbiamo: numero di pattern distinti, numero di resolver distinti, lunghezza media in byte di patter e resolver, numero di coppie distinte utilizzate, dimensione del file in questione, dimesione del dizionario utilizzato, differenza tra dimesione file utilizzato e dimensione dizionario, dimensione del dizionario di puntatori, dimensione coppie distinte, dimensione coppie che ripetute.

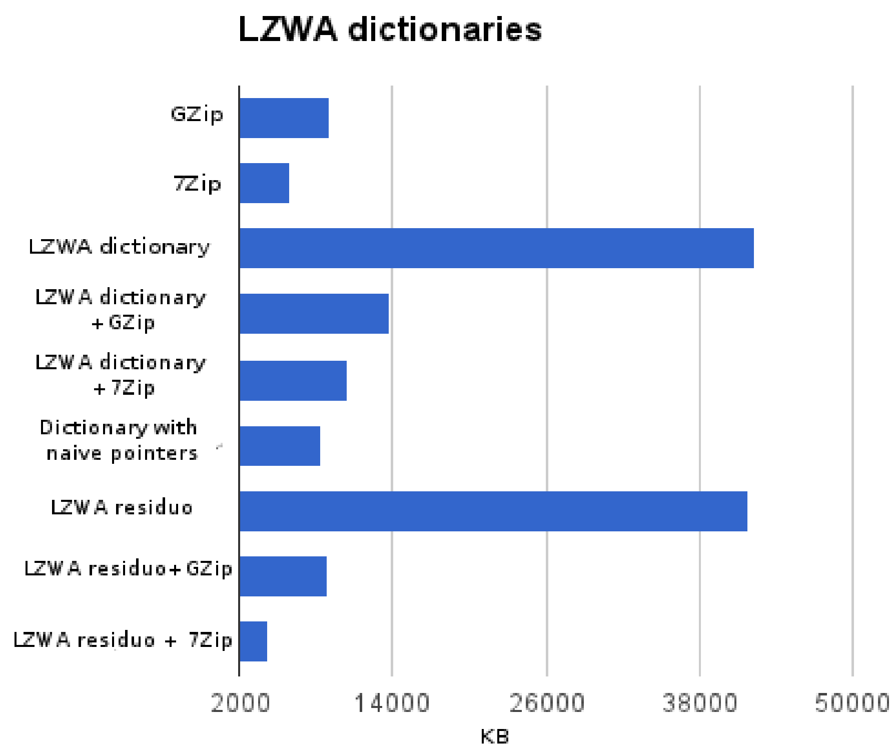

Figura 4.15: composizione dizionari file XML, nella legenda a sinistra partendo dall'alto abbiamo: dizionario GZip, dizionario 7Zip, dizionario LZWA, dizionario LZWA+GZip, dizionario LZWA+GZip, dizionario dei puntatori, dizionario residuo di LZWA, dizionario residuo+GZip, dizionario residuo+7Zip.

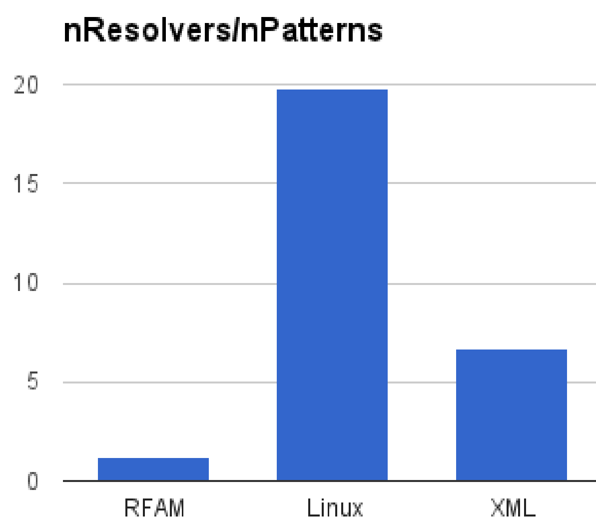

Figura 4.16: rapporto tra numero di resolver e numero di pattern per i dataset Rfam, Linux Kernel e XML.

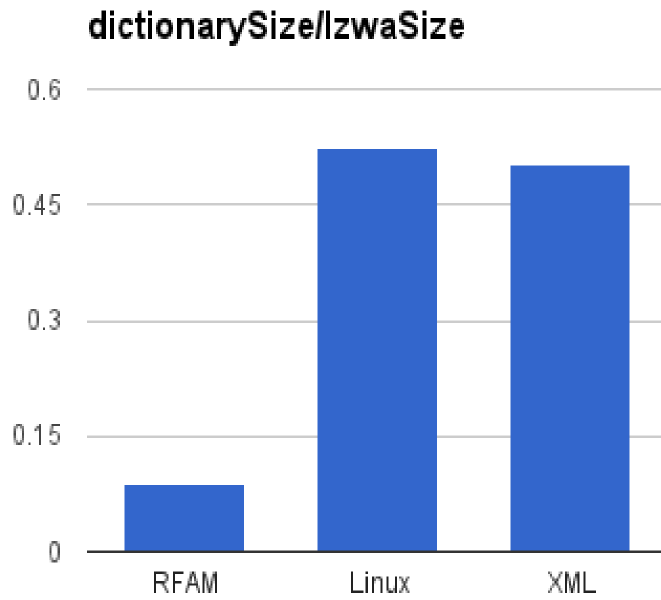

Figura 4.17: dimensione dizionario di pattern e resolver su dimensione file compresso per i dataset per Rfam, Linux Kernel e XML.

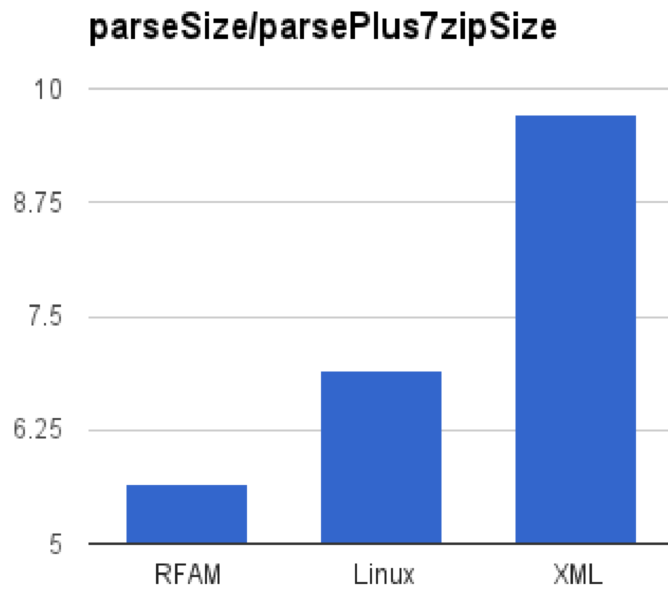

Figura 4.18: rapporto di compressione tra dimensione del file ottenuto da LZWA e quella ottenuta da LZWA + 7Zip per i dataset Rfam, Linux Kernel e XML.

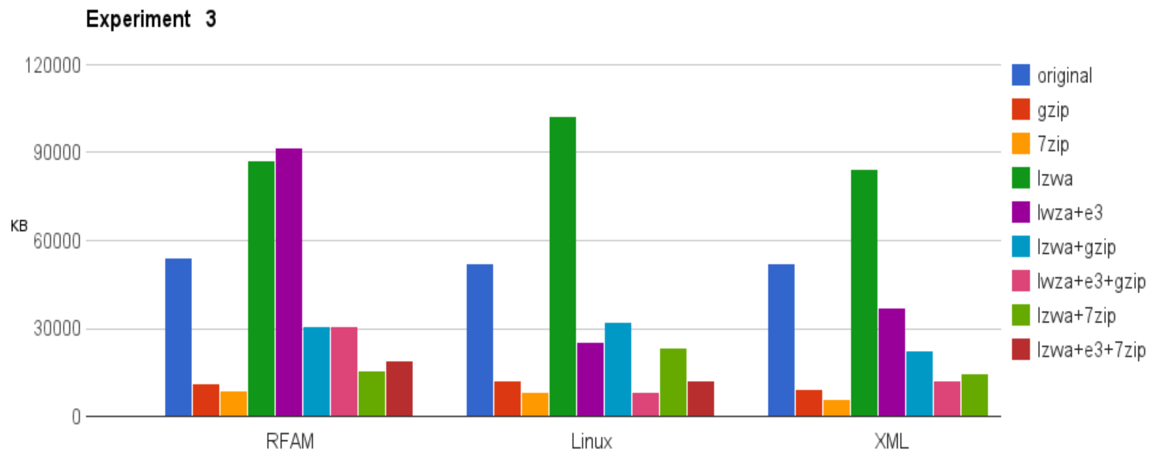

Figura 4.19: risultati dell'esperimento 3 in termini di KB, sui dataset Rfam Linux Kernel e XML.

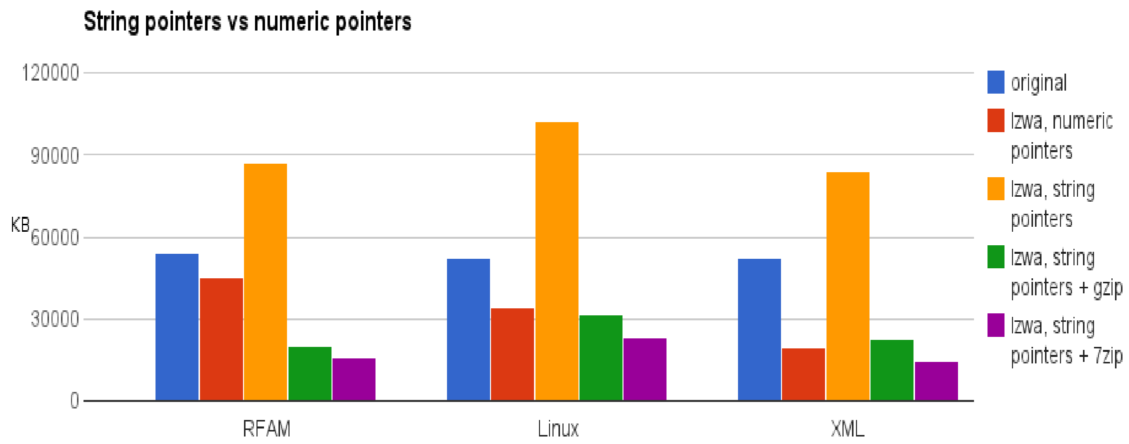

Figura 4.20: confronto in termini di KB della rappresentazione dataset mediante puntatori di stringhe e puntatori numerici per i dataset Rfam, Linux Kernel e XML.

Conclusioni

I risultati ottenuti sembrano in ogni caso portare ad una scelta orientata ai classici compressori basati su stringhe rispetto a LZWA. Risultati migliori in termini di compressione si potrebbero comunque ottenere con una implementazione più efficiente di LZWA. Si potrebbero estendere il numero di caratteri del file originale che vengono rappresentati da una coppia di puntatori. Tale parametro potrebbe essere modificato aumentando il numero di figli presenti nei dizionari di LZWA per nodo e diminuendo la periodicità con cui i gap si presentano. Questo presumibilmente porterebbe a rendere i dati più deterministici, aumentando la mole di dati compressi. I dati sembrano mostrare che il lavoro svolto vada ad approssimare la compressione generata dai metodi off-line basati sulla ricerca di pattern. Si potrebbero inoltre confrontare statistiche temporali su tali dati compressi per avere migliori riscontri. I risultati degli esperimenti inoltre suggeriscono che un interessante dataset su cui si potrebbero svolgere alcuni esperimenti e ottenere risultati migliori sono i read set. In tali insiemi, sequenze di coppie <pattern,resolver> dovrebbero avere relativamente poche coppie ripetute dato che gli errori in read distinti sono probabilmente incorrelati. Non ci sarebbero sequenze di coppie ripetute dato che i read sono ordinati in modo casuale. I classici compressori potrebbero rappresentare un read di k errori con k+1 puntatori mentre LZWA potrebbe rappresentare un read esattamente una coppia di puntatori (a regime).

Bibliografia

[1] Jacob Ziv, A. Lempel, “A Universal Algorithm for Sequential Data Compression”, IEEE Transactions on Information Theory, Vol. 33, 1977.

[2] Jacob Ziv, A. Lempel, “Compression of Individual Sequences via Variable-Rate Coding”, IEEE Transactions on Information Theory, Vol. 24, 1978.

[3] Terry A. Welch, “A Technique for High-Performance Data Compression”, Computer, Vol. 17, 1984.

[4] A. Apostolico, “Fast gapped variants for Lempel–Ziv–Welch compression”, Information and Computation, Vol. 205, 2007.

[5] http://www.gzip.org/

[6] http://www.7-zip.org/sdk.html

[7] C.E. Shannon, “Communication in the Presence of Noise”, Proceedings of the IRE, Vol. 37, 1949.

[8] David Salomon, “Data Compression: The Complete Reference”, Springer, 2007.

[9] A. Apostolico, M. Comin, L. Parida, “Bridging lossy and lossless compression by Motif Pattern Discovery”, Electronic Notes in Discrete Mathematics, Vol. 21, 2005.

[10] D. Huffman, “A Method for the Construction of Minimum Redundancy Codes”, Proceedings of the IRE, 1952.

[11] A.K. Konopka, "Information theories in molecular biology and genomics", Nat. Encyclopedia Hum. Genome, 2005.

[12] S. Grümbach, F. Tahi, "Compression of DNA sequences", In Proceedings of the IEEE Data Compression Conference, 1994.

[13] R. Giancarlo, D. Scaturro, F. Utro, "Textual data compression in computational biology: a synopsis", Bioinformatics, 2009.

[14] X. Chen, S. Kwong, M. Li, DNACompress: fast and effective DNA sequence compression", Bioinformatics, 2002.

[15] G. Korodi, I. Tabus, "An efficient normalized maximum likelihood algorithm for DNA sequence compression", ACM Transactions on Information Systems, 2005.

[16] M. Burrows, D. Wheeler, "A block-sorting lossless data compression algorithm", Technical Report 124, Digital Equipment Corporation, 1994.

[17] D. Adjeroh, F. Nan, "On compressibility of protein sequences", In Proceedings of the IEEE Data Compression Conference, 2006.

[18] D. Adjeroh, Y. Zhang, A. Mukherjee, M. Powell, Tim Bell, "DNA sequence compression using the Burrows-Wheeler transform", In Proceedings of the IEEE Computer Society Conference on Bioinformatics, 2002.

[19] N. Cherniavsky, R. Ladner, "Grammar-based compression of DNA sequences", In DIMACS Working Group on The Burrows–Wheeler Transform, 2004.

[20] Q. Liu, Yu Yang, C. Chen, J. Bu, Y. Zhang, X. Ye, "RNACompress: grammar-based compression and informational complexity measurement of RNA secondary structure", BMC Bioinformatics, 2008.

[21] A.L. Buchsbaum, D.F. Caldwell, K.W. Church, G.S. Fowler, S. Muthukrishnan, "Engineering the compression of massive tables: an experimental approach", In SODA 00: Proceedings of the Symposium on Discrete Algorithms. ACM-SIAM, 2000.

[22] A. Apostolico, F. Cunial, V. Kaul, "Table compression by record intersection". In Proceedings of the IEEE Data Compression Conference (DCC). IEEE Computer Society, 2008.

[23] http://www.inbio.ac.cr

[24] L. Parida, "Pattern Discovery in Bioinformatics Theory & Algorithms", Chapman & Hall/CRC Taylor & Francis Group, 2007.

[25] A.R. Barron, J. Rissanen, Yu Bin, "The minimum description length principle in coding and modeling", IEEE Transaction Information Theory, Vol. 44, 1998.

[26] A. Amelio, A. Apostolico, S.E. Rombo, "Image Compression by 2D Motif Basis", In Proceedings of the 2011 Data Compression Conference, 2011.

[27] A. Apostolico, F. Cunial, "Sequence Similarity by Gapped LZW", In Proceeding of Data Compression Conference, 2011.

[28] http://rfam.sanger.ac.uk

[29] http://pizzachili.dcc.uchile.cl/texts.html