UNIVERSITÀ DEGLI STUDI DI SALERNO

Progetto Compressione Dati 2019/2020

LZRR: LZ77 Parsing with Right Reference

Gruppo: RinPro

Membri: Giugliano Aniello

Cocozziello Giacomo

Professore:

Bruno Carpentieri

Sommario

1. Introduzione

La compressione dati lossless è stata ampiamente studiata all'interno dell'area informatica. Dove per "lossless" intendiamo un insieme di algoritmi di compressione dati che non porta alla perdita di alcuna informazione originale durante la fase di compressione/decompressione dei dati stessi.

Uno dei più importanti algoritmi di compressione lossless, è quello di Lempel-Ziv 77 (LZ77), che raggiunge un elevato grado di compressione. L'idea dell'algoritmo di compressione LZ77 è di comprimere una stringa data in input calcolando una sequenza di frasi copiate dalla più lunga sottostringa sulla posizione sinistra della stringa. L'analisi Bidirezionale, invece, è un'analisi per la compressione dati lossless che calcola una sequenza di frasi copiate da un'altra sottostringa (target phrase) sia nella posizione a sinistra che a destra della stringa data in input. L'analisi Bidirezionale è più generica dell'algoritmo LZ77 poiché calcola le sottostringhe sia a sinistra che a destra.

Nel paper "LZRR: LZ77 parsing with right reference", è stata presentata un'analisi bidirezionale chiamata LZRR, nella quale il numero di frasi risulta essere sempre più piccolo rispetto al numero di frasi di LZ77. LZRR è un algoritmo di tempo polinomiale che calcola frasi da una stringa partendo dalla posizione sinistra e spostandoci verso destra. La principale differenza che esiste tra LZRR e LZ77 è il modo in cui calcola la frase, mentre LZ77 sceglie sempre le sottostringhe più lunghe che occorrono precedentemente (sinistra) all'interno di una frase, LZRR usa sia le sottostringhe precedenti che quelle successive. Proprio per questo, il numero di frasi di LZRR è garantito essere minore rispetto a LZ77. Questo esperimento viene provato utilizzando delle stringhe di Benchmark (consultate sul sito http://corpus.canterbury.ac.nz/) e mostra che il numero di frasi di LZRR è approssimativamente minore di circa il 5% di LZ77. Inoltre, i due algoritmi (LZRR ed

LZ77) verranno implementati nel linguaggio di programmazione C++ all'interno della macchina virtuale Kali Linux.

2. LZ77

L'algoritmo LZ77 effettua la compressione sostituendo occorrenze ripetute di dati con riferimenti ad una copia singola di questi stessi, precedentemente trovati nel flusso di dati non compresso. Un match viene codificato da una tripla di numeri chiamata (offset, length, nc). L'offset rappresenta l'indice del carattere del match trovato, mentre la length rappresenta il numero di caratteri ripetuti ed nc rappresenta il carattere successivo nella stringa in input. Per individuare le corrispondenze, il codificatore deve tener traccia della quantità di dati recentemente utilizzati attraverso un buffer, che può essere di 2Kb, 4Kb o 32Kb.

Il buffer si divide in due parti, search buffer e lookahead buffer. Nel search buffer troviamo le stringhe che precedentemente hanno creato un match con la stringa corrente, mentre nel lookahead buffer andiamo inserire man mano i caratteri del testo. Questo buffer viene chiamato "sliding window", motivo per cui LZ77 viene spesso chiamato compressione a finestra. Il codificatore deve conservare questi dati per cercare le corrispondenze all'interno del testo dato in input mentre il decodificatore deve salvare questi dati per interpretare le corrispondenze a cui fa riferimento il codificatore.

Mostriamo lo pseudocodice dell'algoritmo LZ77:

```
while input is not empty do
    prefix := longest prefix of input that begins in window

    if prefix exists then
        i := distance to start of prefix
        l := length of prefix
        c := char following prefix in input
    else
        i := 0
        l := 0
        c := first char of input
    end if

    output (i, l, c)

    s := pop l+1 chars from front of input
    discard l+1 chars from front of window
    append s to back of window
repeat
```

Figura 1 Algoritmo Lz77

2.1 Compressione

LZ77 itera sequenzialmente all'interno della stringa di input e salva ogni nuovo match all'interno del search buffer. Il processo di compressione può essere diviso in 2 step.

1 step: trovare il più lungo match della stringa che inizia nella posizione corrente.

2 step: fornisce in output la tripla (offset, length, nc)

2.2 Esempio LZ77

Per capire a fondo come computa l'algoritmo LZ77 forniamo in input una stringa:

"abcdabcdcdab"

Partiamo dal primo carattere a sinistra della stringa in input, quindi "a", poiché è il primo carattere della stringa non ha creato match con i caratteri precedenti, e forniamo in output il fattore |a|. Spostiamo l'indice al carattere successivo e troviamo "b", poiché b non è presente precedentemente nel search buffer, inseriamo come fattore |b|. Per i caratteri successivi "c" e "d" vale lo stesso principio e diamo in output come fattori |c| , |d|. Proseguendo con l'indice all'interno della stringa di input, vediamo che la più grande sottostringa che crea un match con i caratteri presenti all'interno del buffer, è "abcd", daremo in output <0,4> dove 0 rappresenta l'offset del primo carattere della sottostringa che ha creato il match e 4 corrisponde alla lunghezza della sequenza dei caratteri del match. A questo punto andando avanti all'interno della stringa di input troviamo i caratteri "cdab". In questo caso, questi caratteri sono già presenti all'interno del search buffer e vengono codificati con <2,4>. Quindi, i fattori dati in output dall'algoritmo sono:

Fattori: <a>, <b>, <c>, <d>, <0,4> , <2,4>

Decodifica

L'algoritmo di decompressione prende in input i fattori e restituisce in output la stringa originale.

Prendiamo in input i fattori:

<a> , <b> , <c> , <d> , <0,4> , <2,4>

Partiamo dal primo fattore dato in output dal codificatore <a>, che viene decodificato con "a". Ripetiamo quest'approccio per i fattori <b> , <c> , <d>. Attualmente la stringa corrente è: "abcd"

Fattori rimanenti: <0,4> , <2,4>.

A questo punto, andiamo ad analizzare il fattore <0,4> e andiamo a sostituire questo con la sottostringa che inizia dalla posizione corrente con offset 0 e length 4.

Ora, la stringa corrente è: "abcdabcd"

Fattori rimanenti: <2,4>

Consideriamo il fattore <2,4> andando a sostituire questo fattore con la sottostringa nella stringa corrente che inizia con offset 2 e length 4.

La stringa in output è: "abcdabcdcdab"

Vedremo un'esecuzione dell'algoritmo di compressione:

Figura 2: Compressione lz77

Prende in input un file (nello2) con all'interno la seguente stringa (abcdabcdcdab) di lunghezza 13 caratteri e restituisce il file compresso nello2.lz.

Figura 3: Decompressione lz77

Nella decompressione l'algoritmo prende in input il file compresso (nello2.lz) mostra in output i fattori con i relativi puntatori.

3 LZRR

3.1 Introduzione

Un'idea chiave di LZRR è di calcolare un BP valido dal testo dato in input calcolando gradualmente il PBP valido dall'inizio di T da sinistra verso destra. Il BP (Bidirectional phrase) della stringa T è una partizione di T come sottostringa $B=f_1,f_2....f_b$ tale che ogni $f_i=T[s_i.....s_i + l -1]$ viene sia copiato da un'altra sottostringa $T[t_i...t_i + l - 1]$ (stringa di riferimento) con $s_i!=t_i$ che può sovrapporsi a $T[s_i.....s_i + l -1]$ o un carattere esplicito (singolo carattere) $f_i=T[s_i]$. La target phrase (stringa di riferimento) viene denotata come una coppia $<t_i,|f_i|>$, dove t_i rappresenta posizione di riferimento (offset) e $|f_i|$ rappresenta la lunghezza (length), e indichiamo con PBP (Partial Bidirectional Phrases) una sottosequenza di BP valida definito come BP $P = f_1,f_2...f_k$ per un prefisso di T che può essere copiato da ognuna delle sottostringhe di T, ad esempio $t_i \in \{1..n\} \setminus \{ s_i \}$ per tutti gli $i \in \{1...k\}$ per una frase target f_i, che evita di copiare se stessa. LZRR analizza la stringa utilizzando due funzioni principali LP e LF ed utilizza 4 array: Suffix Array (SA), Inverse Suffix Array (ISA), LCP array, Longest Previous Factor Array (LPF).

3.2 Funzioni LP e LF

Ogni frase viene selezionata utilizzando la funzione LP che restituisce la più lunga frase valida che segue la frase corrente LZRR.

Algorithm 1: The LP algorithm.

Input:$P = f_1, \ldots, f_p$, Output: $\mathsf{LP}(P)$;
$(k, j) \leftarrow (1, \mathsf{SA}_i[1])$;
$(k, j_{max}, \ell_{max}) \leftarrow (1, \mathsf{NIL}, 0)$;
while *true* **do**
  $j \leftarrow \mathsf{SA}_i[k++]$ // $i = |f_1 \cdots f_p| + 1$.
  if $\mathsf{lcp}(i, j) \leq \ell_{max}$ then break;
  $\ell \leftarrow \mathsf{LF}(P, j)$;
  if $\ell > \ell_{max}$ then $(j_{max}, \ell_{max}) \leftarrow (j, \ell)$;
if $\ell_{max} > 0$ then **return** $\langle j_{max}, \ell_{max} \rangle$ else **return** $T[i]$;

Figura 4: Funzione LP

Per calcolare la funzione LP si calcola la posizione di riferimento jmax cosicché LF(P,jmax) = max {LF(P,1),…. (P,n)}, e poi calcola lmax = LF(P,jmax) che risulta nella frase LZRR <jmax, lmax>. Fino a quando questo metodo ha bisogno di calcolare la funzione LF n volte, esso spende tempo Ω(n). Riduciamo il numero di computazioni di LF osservando il seguente fatto: la lunghezza della più lunga frase valida dalla posizione iniziale i e con una posizione di riferimento j non è mai più lunga di quella del LCP di T[i] e T[j]. Questo fatto suggerisce che dopo aver trovato una frase di lunghezza l', non abbiamo bisogno di computare la funzione LF per ogni posizione j, cosicché LCP di T[i] e T[j] non è più lungo di l'. per una computazione efficiente, ordiniamo le posizioni di riferimento in ordine decrescente rispetto alla lunghezza di LCP per T[i] e manteniamo queste posizioni all'interno del "Sorted Suffix Array" SA di i. Allora, evitiamo di calcolare la funzione LF per posizioni di riferimento su SA_i[kmax + 1..] per la posizione più a sinistra kmax su SA_i, cosicché la frase più lunga valida che inizia ad una posizione di riferimento nel SA_i[1..kmax] è almeno lunga quanto LCP di T[i] e T[SA_i[kmax]..]. perché esiste un jmax su SA_i[1..kmax].

Da ciò ne deriva il seguente lemma:

=> Sia l_k = max{LF(P,SA_i[1]), …… , LF(P,SA_i[k])} e kmax la posizione più a sinistra dell'array SA_i, cosicché $l_{kmax} \geq lcp(i, SA_i[kmax])$. Allora $l_{max} = l_{kmax}$ e $SA_i[..k_{max}]$ contiene j_{max}.

La funzione LP calcola ogni funzione LF dall'inizio del SA_i. Quando l'algoritmo trova k_{max} restituisce la frase corrente valida più lunga.

LF Function

L'algoritmo LF(P,j) trova la frase target valida più lunga con posizioni di riferimento j seguita dal PBP P di T estendendo gradualmente la frase target di lunghezza 1 fino a che non è più possibile trovare stringhe di riferimento copiando la frase target. Quando PBP viene calcolato, esso può includere un loop infinito di riferimenti. Questo avviene perché il PBP come frase target può essere copiata da sinistra a destra della stringa di riferimento. Questo succede quando per calcolare l'estensione P*<j,l> la posizione della frase target in P e <j, l> possono essere raggiunti reciprocamente con un numero finito di riferimenti. L'algoritmo LP evita questi casi usando una struttura dati Union Find, costruita dal PBP P. Ogni insieme disgiunto all'interno della Union Find include posizioni distinte con la stessa sorgente (character phrase) per il PBP P. La struttura dati Union Find viene inizializzata con n insiemi disgiunti e tutti contengono la posizione unica della stringa di input di lunghezza n. Usando la Union Find per il PBP P * <j,l-1> e la frase target <j,l>, la struttura dati per P * <j,l-1> può essere aggiornata dall'operazione di Union(i+l-1, j+l-1). Il loop infinito di riferimenti può essere trovato usando l'operazione di Find all'interno della Union Find. Quindi, per un PBP P valido, l'estensione della posizione iniziale i + l -1 dopo il PBP e la posizione di riferimento j + l -1, Find(i+l-1) è uguale Find(j+l-1) se e solo se un loop infinito di riferimenti esiste. Quindi, la funzione LF va a testare questa condizione ogni volta.

L'algoritmo della funzione LF è il seguente:

Algorithm 2: The LF algorithm.

Input: PBP P, union-find data structure for disjoint sets of P, reference position j;
Output: $\mathsf{LF}(P, j)$;
$\ell \leftarrow 1$;
while $T[i+\ell-1] = T[j+\ell-1]$ *and* $Find(i+\ell-1) = Find(j+\ell-1)$ **do**
  $Union(i+\ell-1, j+\ell-1)$;
  $\ell \leftarrow \ell + 1$;
return $\ell - 1$;

Figura 5: Funzione LF

3.3 Strutture Dati

Ora, andiamo ad analizzare le strutture dati che vengono utilizzate all'interno delle due funzioni LP e LF:

-SA => è una permutazione di [1..n] tale che T[SA[1]..] < …. < T[SA[n]..].

-ISA => è una permutazione di [1..n] tale che SA[ISA[i]=i] per ogni i ϵ {1,2,…,n}

-LCP => è la lunghezza del più lungo prefisso comune di T[i] e T[j].

-LPF=> LPF[i] salva la lunghezza del prefisso più lungo di T[i] che occorre precedentemente.

-Union-Find => è una struttura dati per insiemi disgiunti e supporta 3 operazioni. MakeSet, Union e Find. MakeSet(x) aggiunge l'elemento x all'interno dell'insieme e restituisce l'identificatore (id) di x e questa operazione è consentita se l'insieme non contiene x. La Union (x, y) riceve l'id di x e y, unisce i due insiemi e aggiunge un nuovo insieme XUY nell'insieme originale. Find (x) riceve l'id di x e restituisce l'id dell'insieme contenente x.

3.4 Codifica e Decodifica LZRR

In questo paragrafo mostriamo un esempio di esecuzione dell'algoritmo LZRR con relativa spiegazione.

Data in input una stringa T: "abcdabcdcdab"

L'algoritmo di compressione LZRR partendo dal carattere più a sinistra "a" scorre verso destra cercando la più lunga sottostringa che crea un match con la stringa corrente. Quindi codifica "abcd" con il fattore <4,4> in quanto nella posizione 4 è possibile trovare un'altra occorrenza di "abcd". Andiamo a scorrere l'indice fino alla posizione 4, e codifichiamo "ab" con il fattore <10,2>, in quanto nella posizione 10 troviamo l'occorrenza di "ab". Scorriamo l'indice di altre 2 posizioni, e codifichiamo il carattere "cd" con il fattore <8,2> in quanto nella posizione 8 esiste un'occorrenza di "cd". Dopo di ché troviamo in sequenza i caratteri "c,d,a,b" che non possono essere più codificati come fattori in quanto non creano nessun match con le occorrenze successive, e dunque vengono codificati con caratteri singoli. L'algoritmo fornisce in output i seguenti fattori:

<4,4> , <10,2> , <8,2> , |c| , |d| , |a| , |b|

Figura 6: Compressione LZRR

Mentre l'algoritmo di decompressione prende in input i fattori precedentemente calcolati dal compressore e la lunghezza della stringa in input e attraverso un loop va ad inserire man mano all'interno di un array di output i caratteri decompressi.

Quindi, sapendo che la lunghezza della stringa in input è 12 e poiché l'array viene indicizzato a partire da 0, inizialmente andiamo a definire le posizioni dei caratteri partendo da destra:

- "b=11"

- "a=10"

- "d=9"

- "c=8"

Adesso, prendiamo in input il fattore <8,2> e lo decomprimiamo come "cd".

E andiamo ad aggiungere i due caratteri "cd" alla stringa di output con i relativi indici:

- "d = 7"

- "c = 6"

Adesso, prendiamo il fattore <10,2> e lo decomprimiamo come "ab"

Aggiungiamo i caratteri "ab" alla stringa di output con i relativi indici:

- "b = 5"

- "a = 4"

Adesso, prendiamo in input l'ultimo fattore <4,4> e lo decomprimiamo come "abcd"

Aggiungiamo i caratteri "abcd" alla stringa di output:

- "d = 3"

- "c = 2"

- “b = 1”

- “a = 0”

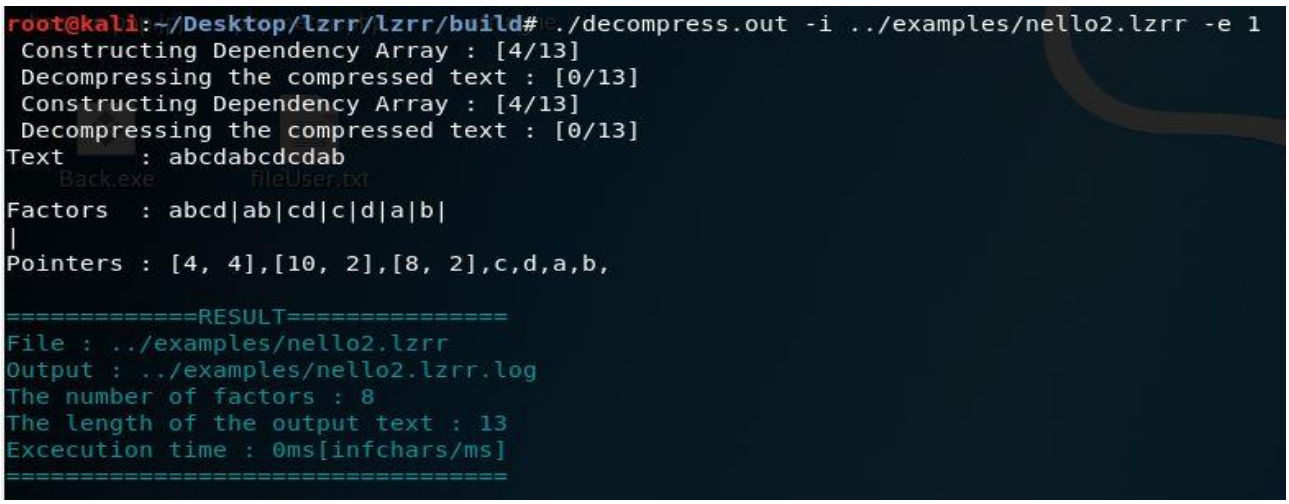


Figura 7: Decompressione LZRR

Poiché abbiamo trovato tutti i caratteri all’interno della stringa, il loop si interrompe e diamo la stringa in output: “abcdabcdcdab”

3.5 Modifica LZ77 indice Pari e Dispari

In questo paragrafo andremo a mostrare la modifica apportata al codice sia all’algoritmo LZ77 che all’algoritmo LZRR.

All’interno della classe Compress.main.cpp abbiamo aggiunto due nuovi “mode” (LZPARI ed LZDISPARI).

Dato un testo in input, andiamo a considerare soltanto i caratteri nelle posizioni pari/dispari.

```
else if (mode == "lzpar"){
string k = "";
int i=0;
for (i=0; i<text.length(); i++){
        if(i%2==0){
        k = k+text[i];
        }
}
LZ77::compress(k, writer);


}
```

Figura 8: Modifica del codice per Lz77 per indice pari

```
else if (mode == "lzdis"){
string k = "";
int i=0;
for (i=0; i<text.length(); i++){
        if(i%2==1){
        k = k+text[i];
        }
}
LZ77::compress(k, writer);


}
```

Figura 9: Modifica del codice per lz77 per indice dispari

Di seguito viene mostrata l'esecuzione dei codici precedenti:

```
root@kali:~/Desktop/lzrr/lzrr/build# ./compress.out -i ../examples/nello2 -m lzpar
Loading : ../examples/nello2
constructing LPFUD...[END]
Computing factors...[END]
=============RESULT===============
File : ../examples/nello2
Output : ../examples/nello2.lzpar
Mode : lzpar
The length of the input text : 13
The number of factors : 5
Excecution time : 0ms[infchars/ms]
==================================
```

Figura 10: Esecuzione compressione LZPAR

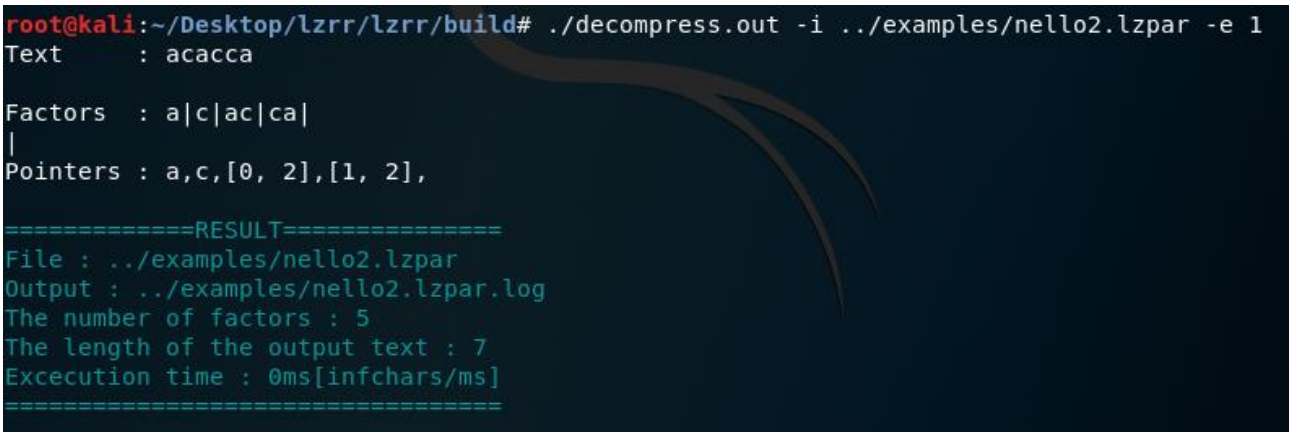


Figura 11: Esecuzione decompressione LZPAR

Figura 12: Esecuzione Compressione LZDIS

Figura 13: Esecuzione Decompressione LZDIS

3.6. Modica LZRR Pari/Dispari

In questo paragrafo andremo a mostrare la codifica apportata al codice LZRR.

Aggiungiamo nella classe Compress.main.cpp due mode (LZRRPARI e LZRRDISPARI).

```
root@kali:~/Desktop/lzrr/lzrr/build# ./compress.out -i ../examples/nello2 -m lzrrpar
Loading : ../examples/nello2
constructing Suffix Array...[END]
constructing LCP Array : [0/7]
Initializing the dependency array manager...[END]
compressing text.. : [0/7]
=============RESULT===============
File : ../examples/nello2
Output : ../examples/nello2.lzrrpar
Mode : lzrrpar
The length of the input text : 13
The number of factors : 6
Excecution time : 0ms[infchars/ms]
==================================
```

Figura 14: Compressione LZRR con indice pari

```
root@kali:~/Desktop/lzrr/lzrr/build# ./decompress.out -i ../examples/nello2.lzrrpar -e 1
 Constructing Dependency Array : [2/7]
 Decompressing the compressed text : [0/7]
 Constructing Dependency Array : [2/7]
 Decompressing the compressed text : [0/7]
Text     : acacca

Factors  : ac|a|c|c|a|
|
Pointers : [2, 2],[5, 1],[4, 1],c,a,

=============RESULT===============
File : ../examples/nello2.lzrrpar
Output : ../examples/nello2.lzrrpar.log
The number of factors : 6
The length of the output text : 7
Excecution time : 0ms[infchars/ms]
==================================
```

Figura 15: Decompressione LZRR con indice pari

Figura 16: Compressione LZRR Dispari

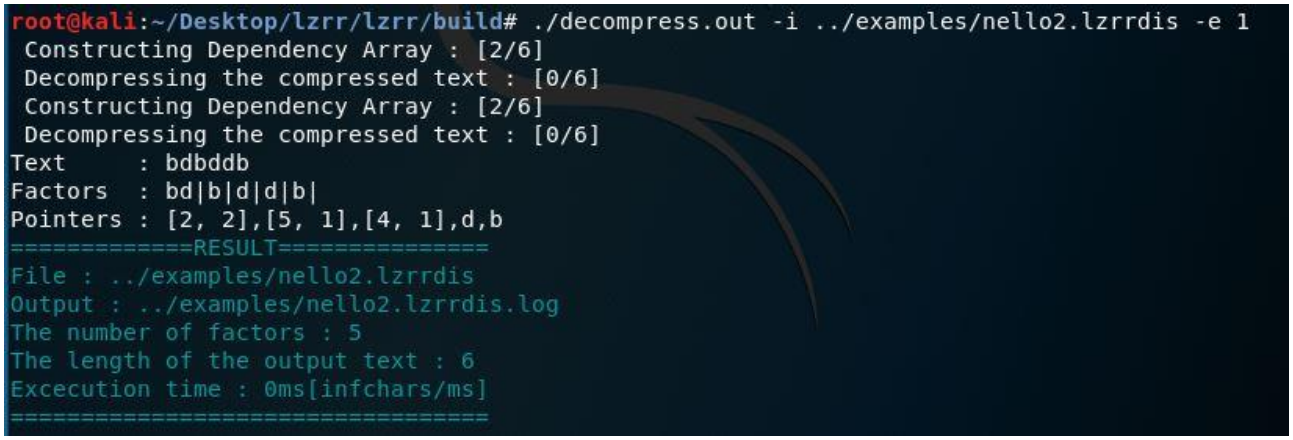


Figura 17: Decompressione LZRR Dispari

4. Analisi teorica

4.1 Tempo di esecuzione

L'algoritmo LZRR lavora in spazio lineare O(n). L'algoritmo ha bisogno di due strutture dati, la Union Find per la funzione LF e la struttura dati per calcolare la sequenza $W_p = SA_{sp}[1]$,lcp(s_p, $SA_{sp}[1]$, …… $SA_{sp}[k_p]$, lcp(i, $SA_i[k_p]$]) dove K_p è il k_{max} in LP(P_p-1) ed s_p è la posizione iniziale della p-esima frase di LZRR. Possiamo calcolare W_p(1…k) in tempo O(k) usando SA, ISA e LCP per due interi i e k. I 3 array SA, ISA e LCP, data una stringa in input possono essere costruiti in tempo O(n) e spazio O(n). In più la seconda struttura dati può essere costruita in tempo e spazio O(n) e l'algoritmo LZRR lavora in spazio O(n).

Sia G la sequenza di operazioni su un insieme disgiunto eseguite dall'algoritmo LZRR, e W la sequenza w1…wb dove b è il numero di frasi di LZRR(T). quindi, il tempo di esecuzione viene rappresentato dalla somma dei tempi di esecuzione di G e W, e dal tempo utilizzato per eseguire SA, ISA e LCP. Inoltre, W può essere calcolato in tempo O(n^2). G viene calcolato in tempo O($n^2a_{n2}n^2$). E gli array SA, ISA e LCP in tempo O(n). Come risultato finale, quindi, calcoliamo LZRR(T) in tempo O($n^2a_{n2}n^2$) e spazio O(n).

4.2 Prova del teorema

Definiamo due BP LZ'(T) e LZOR(T) e mostriamo 3 prove:

1- |LZ'(T)| = |LZ(T)|

2- |LZRR(T)| < |LZOR(T)|

3- |LZOR(T)| = |LZ'(T^R)|

Dimostriamo la prima prova: - |LZ'(T)| = |LZ(T)|.

Definiamo LZ'(T) = f_1….f_k che analizza T partendo dalla posizione destra a sinistra tale che ogni frase è la più lunga occorrenza di sottostringa

precedentemente (sinistra) trovata in T. Una idea chiave di questa prova è che LZ'(T) sceglie una sottostringa come frase LZ', poi esiste una frase LZ che inizia in posizione sulla frase LZ' ed include l'ultima posizione della frase LZ'. Questo avviene perché la frase LZ' occorre precedentemente in T e la frase LZ è la più lunga sottostringa che occorre precedentemente in T. Questo fatto suggerisce che per ogni frase LZ', |LZ(T)| < |LZ'(T)|. Al contrario, se LZ(T) sceglie una sottostringa come di LZ esiste una frase LZ' che inizia in posizione sulla frase LZ, ed include la posizione iniziale della frase LZ. Questo avviene perché la frase LZ occorre precedentemente in T e la frase LZ' è la più lunga sottostringa che occorre precedentemente in T. questo fatto ci suggerisce che per ogni frase LZ, |LZ'(T)| ≤ |LZ(T)|. Quindi, LZ'(T) = LZ(T).

Dimostriamo la seconda prova: |LZRR(T)| < |LZOR(T)|

Definiamo LZOR(T) = f_1 …. f_k analizza T da sinistra verso destra cosicché ogni frase è la più lunga sottostringa che viene osservata successivamente (destra) in T. LZOR(T) può scegliere una sottostringa in una posizione come una frase LZOR, quindi LZRR(T) può anche scegliere la sottostringa come una frase LZRR. Questo avviene perché la frase candidata con riferimento nelle posizioni a destra sono sempre frasi valide in LZRR. Poiché questo fatto avviene per ogni posizione in T, |LZRR(T) ≤ LZOR(T)|.

Dimostriamo la terza prova: |LZOR(T)| = |LZ'(T^R)|

Analizziamo la stringa usando la più lunga sottostringa dell'occorrenza successiva nella stringa il che è uguale ad analizzare la stringa inversa da destra a sinistra usando la più lunga occorrenza precedente. Quindi, |LZOR(T)| = |LZ'(T^R)|.

5. Esperimento

In questa sezione mostreremo degli esempi di esecuzione dei due algoritmi LZ77 e LZRR, e dimostreremo che LZRR lavora meglio di LZ77 in quanto comprime utilizzando meno frasi. Tuttavia, il risparmio rispetto al numero di frasi viene pagato con memoria e tempo di esecuzione maggiori.

String	String length	$\|LZ77\|$	$\|LEX\|$	$\|LZRR\|$	$\frac{\|LZRR\|}{\|LZ77\|}$
fib41	267,914,296	22	**4**	5	0.227
rs.13	216,747,218	52	**40**	51	0.981
tm29	268,435,456	56	43	**31**	0.554
dblp.xml.00001.1	104,857,600	59,385	58,537	**55,127**	0.928
dblp.xml.00001.2	104,857,600	59,556	60,220	**55,122**	0.926
dblp.xml.0001.1	104,857,600	78,167	82,879	**73,584**	0.941
dblp.xml.0001.2	104,857,600	78,158	99,467	**73,583**	0.941
sources.001.2	104,857,600	294,994	466,074	**287,411**	0.974
dna.001.1	104,857,600	308,355	307,329	**295,354**	0.958
proteins.001.1	104,857,600	355,268	364,024	**337,711**	0.951
english.001.2	104,857,600	335,815	487,586	**324,282**	0.966
einstein.de.txt	92,758,441	34,287	37,719	**31,798**	0.927
einstein.en.txt	467,626,544	89,437	96,487	**83,368**	0.932
world_leaders	46,968,181	175,670	179,503	**165,626**	0.943
influenza	154,808,555	769,286	764,634	**714,320**	0.929
kernel	257,961,616	793,915	794,058	**741,556**	0.934
cere	461,286,644	1,695,631	1,649,448	**1,597,657**	0.942
coreutils	205,281,778	1,441,384	1,439,918	**1,359,606**	0.943
Escherichia_Coli	112,689,515	2,078,512	2,014,012	**1,961,296**	0.944
para	429,265,758	2,332,657	2,238,362	**2,200,802**	0.943

Figura 18: Numero di frasi per ogni algoritmo

		Execution time [sec]			Memory consumption [MB]		
String	String length	LZ77	LEX	LZRR	LZ77	LEX	LZRR
einstein.de.txt	92,758,441	24	16	27	2,266	2,266	3,808
einstein.en.txt	467,626,544	130	85	147	11,418	11,418	19,196
world_leaders	46,968,181	8	5	16	1,148	1,148	1,939
influenza	154,808,555	42	27	51	3,781	3,781	6,351
kernel	257,961,616	71	47	88	6,299	6,299	10,602
cere	461,286,644	131	90	500	11,263	11,263	18,925
coreutils	205,281,778	56	37	68	5,013	5,013	8,453
Escherichia_Coli	112,689,515	32	22	46	2,752	2,752	4,632
para	429,265,758	125	85	203	10,481	10,481	17,609

Figura 19: Tempo di esecuzione e memoria per ogni algoritmo

5.1 Esempi su stringhe di benchmark

Di seguito mostreremo le esecuzioni di alcune stringhe di benchmark prese dal sito

http://corpus.canterbury.ac.nz/details/.

Esempio stringa alphabet.txt

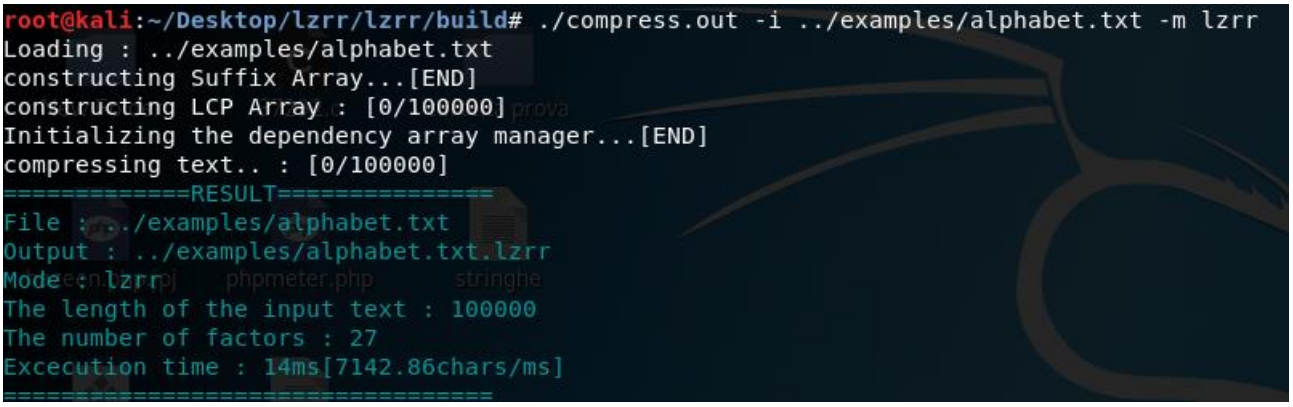
Compressione LZ77 con relativo tempo di esecuzione

L'algoritmo di compressione LZ77 prende in input il testo "alphabet.txt" con una lunghezza di 100000 caratteri, e comprime in 27 fattori in tempo 7ms.

Decompressione LZ77

```
a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z,[0, 99974]
```

Compressione LZRR con relativo tempo di esecuzione



L'algoritmo di compressione LZRR prende in input il testo "alphabet.txt" con una lunghezza di 100000 caratteri e li comprime in 27 fattori impiegando 14 ms.

Decompressione LZRR

```
[26, 99974],e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z,a,b,c,d
```

Esempio stringa di benchmark bible.txt

Compressione LZ77 con relativo tempo di esecuzione

```
root@kali:~/Desktop/lzrr/lzrr/build# ./compress.out -i ../examples/bible.txt -m lz
Loading : ../examples/bible.txt
constructing LPFUD...[END]
Computing factors...[END]
=============RESULT===============
File : ../examples/bible.txt
Output : ../examples/bible.txt.lz
Mode : lz
The length of the input text : 4047392
The number of factors : 337558
Excecution time : 762ms[5311.54chars/ms]
==================================
```

In questo esempio possiamo notare che l'algoritmo di compressione prende in input il testo "bible.txt" con una lunghezza di 4047392 caratteri e li comprime in 334558 fattori con un tempo di esecuzione di 762 ms.

Decompressione LZ77

```
root@kali:~/Desktop/lzrr/lzrr/build# ./decompress.out -i ../examples/bible.txt.lz -e 1
=============RESULT===============
File : ../examples/bible.txt.lz
Output : ../examples/bible.txt.lz.log
The number of factors : 337558
The length of the output text : 4047392
Excecution time : 65ms[62267.6chars/ms]
==================================
```

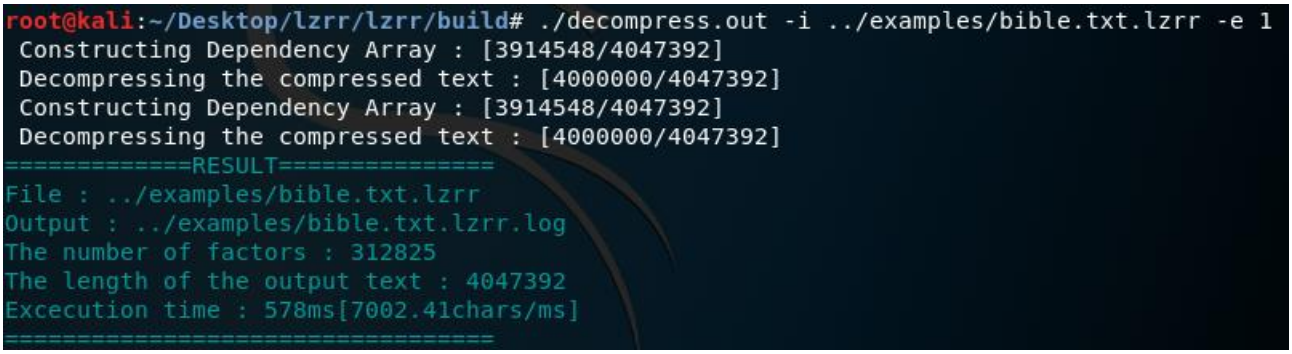
E per decomprimere l'algoritmo LZ77 impiega 65 ms.

Compressione LZRR con relativo tempo di esecuzione

```
root@kali:~/Desktop/lzrr/lzrr/build# ./compress.out -i ../examples/bible.txt -m lzrr
Loading : ../examples/bible.txt
constructing Suffix Array...[END]
constructing LCP Array : [0/4047392]
Initializing the dependency array manager...[END]
compressing text.. : [4019419/4047392]
=============RESULT===============
File : ../examples/bible.txt
Output : ../examples/bible.txt.lzrr
Mode : lzrr
The length of the input text : 4047392
The number of factors : 312825
Excecution time : 2202ms[1838.05chars/ms]
==================================
```

L’algoritmo di compressione LZRR prende in input il testo “bible.txt” avente lunghezza 4047392 caratteri, e li comprime in 312825 fattori impiegando tempo 2202 ms.

Decompressione LZRR



Mentre l’algoritmo di decompressione impiega tempo 578 ms.

Quindi, prendendo in input il testo “bible.txtx” possiamo notare che l’algoritmo di compressione LZ77 comprime in 337558 fattori mentre l’algoritmo LZRR comprime in 312825, quindi come mostrato precedentemente LZRR rispetto a LZ77 ha un grado di compressione maggiore. Ora, andremo a considerare il tempo di esecuzione dei due algoritmi. LZ77 per comprimere impiega tempo 762 ms mentre LZRR impiega tempo 2202 ms, mentre nella decompressione l’algoritmo LZ77 impiega tempo 65 ms rispetto a LZRR che impiega tempo 578 ms. In conclusione, abbiamo dimostrato in pratica che l’algoritmo LZRR comprime meglio di LZ77, pagando tuttavia un tempo di esecuzione maggiore.