UNIVERSITÀ DEGLI STUDI DI SALERNO
DIPARTIMENTO D'INFORMATICA

Compressione Dati

BBWT compression

Professore:
Carpentieri Bruno

Studente:
Baldi Antonio Matr.0522501185
Cirillo Antonio Matr.0522501257
Sabato Vincenzo Matr.0522501188
Strianese Davide Benedetto Matr.0522501458

Anno Accademico: 2022/2023

Indice

1 Introduzione alla compressione dei dati

Al momento non esiste una definizione precisa di compressione dati. Intuitivamente si può dire che è un processo che permette la codifica di un insieme di dati D in un altro insieme di dati ΔD avente una dimensione inferiore. Una caratteristica fondamentale della compressione è quella di poter risalire all'insieme D partendo da ΔD tramite un programma che permette di ricostruire lo stesso insieme di dati iniziale (compressione lossless) o una sua approssimazione accettabile (compressione lossy). Originariamente la compressione dei dati veniva utilizzata per minimizzare l'utilizzo dello spazio su disco, in quanto c'erano dischi piccoli che dovevano ospitare grandi basi di dati. Col passare del tempo questo processo ha assunto un ruolo sempre più fondamentale nella comunicazione. Infatti, senza di essa oggi non avremmo cose come la televisione in alta definizione, gli smartphone e lo streaming via internet, in quanto tutti i dati che viaggiano in rete sono dati che vengono compressi per aumentare la banda di comunicazione o per inviare, nello stesso momento, più trasmissioni sullo stesso canale. Quindi possiamo dire che la compressione dei dati viene principalmente usata per:

- **Data storage**: comprende algoritmi utili per comprimere i dati da memorizzare.
- **Data communication**: comprende algoritmi utili per inviare bit di dati su un canale di comunicazione. Questo perché il canale di comunicazione è fisicamente piccolo, di conseguenza non è possibile inviare i dati nella loro forma originale.

Come mostrato nella Figura 1, in generale il sistema di compressione dei dati è composto da 5 parti:

- **Trasmettitore**: processo che genera la sequenza di dati da comprimere e successivamente da trasmettere.
- **Compressore**: algoritmo che prende in input D e genera ΔD.
- **Canale di comunicazione**: mezzo fisico sul quale vengono inviati i dati.
- **Decompressore**: algoritmo che prende in input ΔD e genera D oppure una sua buona approssimazione.
- **Ricevitore**: processo che riceve la sequenza di dati e la elabora.

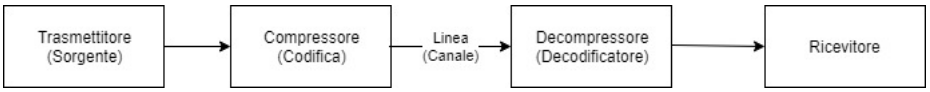


Figure 1: Sistema di compressione

In sintesi lo schema lavora nel seguente modo: il trasmettitore deve inviare dei dati, per fare ciò usa un algoritmo di compressione e di codifica in modo tale da consentire ai dati di viaggiare sul canale in maniera sicura. Il ricevitore, prima di acquisire i dati, li decomprime e li codifica attraverso un algoritmo di decompressione e di codifica. Come si può intuire questo processo è oneroso sia in termini di calcolo che in termini di tempo a causa della compressione che causa un ritardo nella trasmissione. Dallo schema precedentemente descritto è possibile fare un'ulteriore divisione delle tecniche di compressione:

- **Online Data Compression**: Il trasmettitore genera istante per istante delle porzioni della sequenza di dati iniziale da comprimere. Questo però non permette l'utilizzo di tecniche di compressione più efficienti
- **Offline Data Compression**: Il trasmettitore è a conoscenza della quantità di dati da inviare. Questo permette l'utilizzo di metodi di compressione più efficienti.

1.1 Compressione lossless e lossy

Esistono due tipi di compressione:

- **Compressione lossless**: usata quando i dati, dopo essere stati compressi, devono essere decompressi senza perdita di informazione. Questo tipo di compressione viene anche detta compressione bit preserving. Viene utilizzata su alcuni tipi di dati specifici, come ad esempio dati monodimensionali oppure quando l'acquisizione dei dati è molto costosa e quindi non si vogliono avere perdite di bit d'informazione. La compressione lossless porta a rapporti di compressione inferiori rispetto alla lossy.
- **Compressione lossy**: nella compressione lossy non c'è più un unico parametro che rappresenta di quanto è stato compresso il file ma sono presenti due valori legati tra di loro: l'ammontare della compressione (bit rate) e la fidelity, ovvero la fedeltà del file decompresso rispetto all'originale.

2 Obiettivi

L'obiettivo del progetto è quello di studiare e implementare una variante biettiva della **trasformata di Burrows-Wheeler** (BWT). In particolare, partendo dalla BWT verrà costruita una versione biettiva che si basa sulle parole di Lyndon. Inoltre, per poter migliorare la compressione, verranno aggiunti algoritmi come la **Move to Front Transform** (MTF) e la **Run-lenght encoding** (RLE) prima del processo di compressione.

Una volta implementato il pre-processing dei dati verrà eseguita la compressione tramite gli algoritmi di **Huffman** e **Arithmetic Coding** in modo tale da poter effettuare un confronto sul rapporto e sui tempi di compressione tra le due versioni della trasformata. Per la realizzazione del progetto ci siamo basati sul paper *On Bijective Variants of the Burrows-Wheeler Transform* di Manfred Kufleitner [1] e sul precedente progetto *Secure Compression and Pattern Matching Based on Burrows-Wheeler Tansform* di Ceruso Raffaele e Leo Giovanni [2].

3 Concetti teorici fondamentali

In questo paragrafo verranno presentati brevemente quelli che sono i concetti teorici che stanno alla base del progetto.

3.1 Concetti preliminari

3.1.1 Ordine lessicografico

Sia $\Sigma = a_0, ..., a_k$ un alfabeto e sia $a_0 < ... < a_k$ un ordinamento degli elementi di Σ. Siano $x, y \in \Sigma^*$, diremo che $x < y$ rispetto all'ordinamento lessicografico se x e y verificano una delle seguenti condizioni:

1. $y = xz$ con $z \in \Sigma^+$, cioè x è un prefisso di y e $x \neq y$
2. $x = zax'$, $y = zby'$, con $z, x', y' \in \Sigma^*$, $a, b \in \Sigma$ e $a < b$

Esempio Supponiamo che le lettere dalla a alla z siano ordinate come nel dizionario della lingua italiana. Allora:

- Siano $x = sole$ e $y = soleggiato \rightarrow$ per la prima condizione $x < y$
- Siano $x = df\mathbf{er}ty$ e $y = df\mathbf{il}cv \rightarrow$ per la seconda condizione $x < y$

3.1.2 Ordine infinito

Sia $w^w = www...$ la sequenza infinita ottenuta come potenza infinita di w. Definiamo l'ordine infinito come segue.

$$u \leq^w w :\leftrightarrow u^w = uuu... \leq_{lex} www... = w^w$$
$$\Leftrightarrow$$
$$ab \leq_{lex} aba$$
$$aba \leq^w ab$$

3.1.3 Coniugazione

Sia $\Sigma = \{a_0, ..., a_k\}$ un alfabeto e siano x ed y due parole definite su Σ. x ed y si dicono coniugate se esistono due parole $u, v \in \Sigma^+$ tali che:

1. $x = uv$
2. $y = vu$

Diremo che u è una rotazione ciclica di y se u e v sono coniugate.

Esempio Siano $x = abcd$ e $y = cdab$, in questo caso $u = ab$ e $v = cd$.

3.1.4 Parole di Lyndon

Una parola w è **primitiva** se $w = v^n$ implica $n = 1$. Una **parola di Lyndon** è una parola primitiva che è minore di tutte le sue coniugate rispetto all'ordine lessicografico.

Esempio La parola *banana* non è una parola di Lyndon in quanto la sua coniugata $ananab \leq_{lex} banana$. La parola *abanan* è una parola di Lyndon in quanto è più piccola di tutte le sue coniugate ($anaban, ananab, banana, nabana, nanaba$).

3.1.5 Funzione di etichettatura lambda

Sia Σ un alfabeto e sia w una stringa definita sull'alfabeto Σ. Definiamo la **funzione di etichettatura** λ_w sulla stringa w come segue:

$$\lambda_w(i) = w_i$$

Dove w_i è l'i-esimo carattere della stringa w. Per convenzione il carattere w_1 viene denotato con $first(w)$ mentre l'ultimo carattere w_n con $last(w)$.

Esempio Sia w = catamarano, allora $\lambda_w(2) = a,\ first(w) = c,\ last(w) = o$.

3.1.6 Permutazione

Sia X un insieme, diremo che una funzione biettiva $f : X \to X$ è una permutazione di X.

Esempio Sia $X = \{1, ..., n\}$ un insieme finito, ed $f : X \to X$ definita come segue: $f(i) = i + 1\ \forall\ i \in \{1, ..., n-1\}\ e\ f(n) = 1$

3.2 Huffman

Nella teoria dell'informazione, per codifica di Huffman si intende un algoritmo di codifica dei simboli usato per la compressione di dati. L'algoritmo usa la frequenza relativa di ciascun carattere per cercare di effettuare una codifica ottimale. Viene sviluppato per la prima volta nel 1952 da David A. Huffman, uno studente dottorando presso il MIT, e pubblicata su A Method for the Construction of Minimum-Redundancy Codes.
La codifica di Huffman usa un metodo specifico per scegliere la rappresentazione di ciascun simbolo, risultando in un codice senza prefissi (cioè in cui nessuna codifica binaria dei simbolo è il prefisso della codifica binaria di nessun altro simbolo) che esprime il carattere più frequente nella maniera più breve possibile. È stato dimostrato che la codifica di Huffman è il più efficiente sistema di compressione di questo tipo: nessun'altra mappatura di simboli in stringhe binarie può produrre un risultato più breve nel caso in cui le frequenze di simboli effettive corrispondono a quelle usate per creare il codice.
Per un insieme di simboli la cui cardinalità è una potenza di due e con una distribuzione probabilistica uniforme, la codifica di Huffman equivale alla semplice codifica binaria a blocchi. Le frequenze usate possono essere quelle generiche nel dominio dell'applicazione basate su medie precalcolate, o possono essere le frequenze trovate nel testo che deve essere compresso (questa variante richiede che la tabella di frequenza sia registrata insieme con il testo compresso; vi sono diverse implementazioni che adottano dei trucchi per registrare queste tabelle efficientemente).
La codifica di Huffman è ottimale quando la probabilità di ciascun simbolo in input è una potenza negativa di due. Le codifiche senza prefissi tendono a essere inefficienti sui piccoli alfabeti o quando le probabilità sono comprese tra potenze di due.

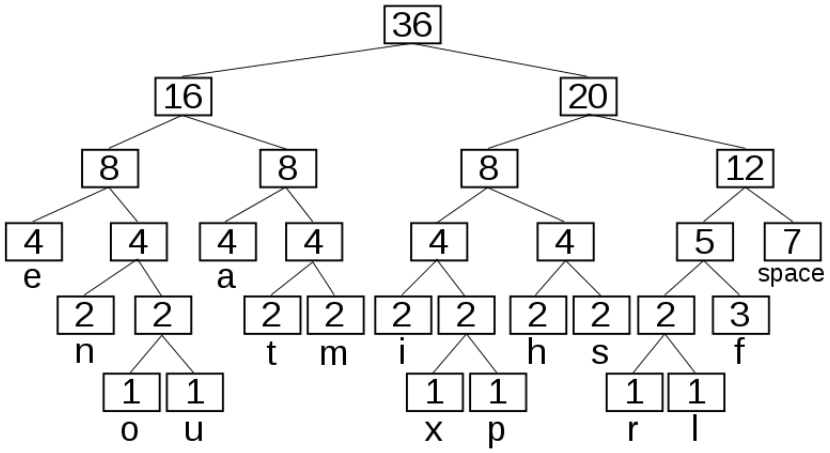


Figure 2: Codifica di Huffman della stringa "this is an example of a huffman tree"

3.3 Codifica Aritmetica

La codifica aritmetica è una tecnica di compressione senza perdita di informazione. Normalmente in informatica i dati sono rappresentati tramite un insieme fisso di bit, ad esempio i caratteri vengono spesso rappresentati con otto bit.
La codifica aritmetica, partendo dal presupposto che alcuni simboli tendono ad apparire più frequentemente di altri, assegna dei codici di lunghezza variabile ai simboli al fine di minimizzare il numero totale di bit da trasmettere. Questa strategia viene utilizzata anche da altri sistemi di codifica, come la codifica di Huffman, con la differenza che in Huffman viene associata una specifica codifica ad ogni simbolo mentre nella codifica aritmetica viene associata una singola codifica all'intero messaggio o a blocchi di questo.
Per rendere il tutto più chiaro viene riportato un semplice esempio. Supponiamo di avere una sequenza di simboli che provengono da un alfabeto di tre elementi: A, B, C. Un semplice codificatore a blocchi potrebbe trasformare ogni simbolo in una sequenza di due bit, ma sarebbe uno spreco, infatti due bit possono esprimere quattro combinazioni e quindi una combinazione non verrebbe mai usata. Possiamo pensare di rappresentare i simboli come un numero razionale compreso tra 0 e 1 in base 3 e di rappresentare ogni simbolo come una cifra del numero. Per esempio la sequenza "ABBCAB" verrebbe convertita in 0,0112013. Questa poi potrebbe essere convertita in base due e potrebbe diventare per esempio 0,0010110012 — questa sequenza usa 9 bit e al posto dei 12 bit richiesti da un codificatore ingenuo e quindi occupa il 25% in meno. Il decodificatore ovviamente dovrebbe fare i passi opposti per ottenere la sequenza di partenza.

3.4 Lempel-Ziv-Welch

Lempel-Ziv-Welch (LZW) è un algoritmo universale di compressione dei dati senza perdita creato da Abraham Lempel, Jacob Ziv e Terry Welch. È stato pubblicato da Welch nel 1984 come implementazione migliorata dell'algoritmo LZ78 pubblicato da Lempel e Ziv nel 1978. L'algoritmo è semplice da implementare e ha il potenziale per un throughput molto elevato nelle implementazioni hardware. È l'algoritmo dell'utility di compressione file Unix compress ed è utilizzato nel formato immagine GIF.
Lo scenario descritto nell'articolo di Welch del 1984 [3] codifica sequenze di dati a 8 bit come codici a lunghezza fissa di 12 bit. I codici da 0 a 255 rappresentano sequenze di 1 carattere che consistono nel corrispondente carattere a 8 bit, mentre i codici da 256 a 4095 sono creati in un dizionario per le sequenze incontrate nei dati durante la codifica. In ogni fase della compressione, i byte in ingresso vengono riuniti in una sequenza fino a quando il carattere successivo non costituisce una sequenza senza codice nel dizionario. Il codice della sequenza (senza quel carattere) viene aggiunto all'output e un nuovo codice (per la sequenza con quel carattere) viene aggiunto al dizionario.
L'idea è stata rapidamente adattata ad altre situazioni. In un'immagine basata su una tavola di colori, ad esempio, l'alfabeto naturale dei caratteri è l'insieme degli indici della tavola di colori e, negli anni '80, molte immagini avevano tavole di colori piccole (dell'ordine di 16 colori). Per un alfabeto così ridotto, i codici completi a 12 bit producevano una scarsa compressione, a meno che l'immagine non fosse di grandi dimensioni, per cui è stata introdotta l'idea di un codice a larghezza variabile: i codici iniziano in genere con un bit più largo dei simboli da codificare e, man mano che si esaurisce ogni dimensione del codice, la larghezza del codice aumenta di 1 bit, fino a un massimo prescritto (in genere 12 bit). Quando si raggiunge il valore massimo del codice, la codifica prosegue utilizzando la tabella esistente, ma non vengono generati nuovi codici da aggiungere alla tabella.
Ulteriori perfezionamenti includono un codice per indicare che la tabella dei codici deve essere cancellata e ripristinata allo stato iniziale (un "codice di cancellazione", in genere il primo valore subito dopo i valori dei singoli caratteri dell'alfabeto), e un codice per indicare la fine dei dati (un "codice di stop", in genere uno maggiore del codice di cancellazione). Il codice di cancellazione permette di reinizializzare la tabella dopo che si è riempita, consentendo alla codifica di adattarsi alle variazioni dei dati in ingresso. I codificatori intelligenti possono monitorare l'efficienza della compressione e cancellare la tabella ogni volta che la tabella esistente non corrisponde più bene all'input.
Poiché i codici vengono aggiunti in modo determinato dai dati, il decodificatore imita la costruzione della tabella quando vede i codici risultanti. È fondamentale che il codificatore e il decodificatore siano d'accordo sulla varietà di LZW utilizzata: la dimensione dell'alfabeto, la dimensione massima

della tabella (e la larghezza del codice), l'utilizzo o meno della codifica a larghezza variabile, la dimensione iniziale del codice e l'utilizzo o meno dei codici clear e stop (e i loro valori). La maggior parte dei formati che utilizzano LZW inseriscono queste informazioni nelle specifiche del formato o forniscono campi espliciti per queste informazioni in un'intestazione di compressione dei dati.

3.5 Move-To-Front Transform

Move-to-front transform (MTF) è una codifica di dati (di solito un flusso di byte) sviluppata per migliorare le prestazioni delle tecniche di codifica entropica per la compressione dei dati. L'idea principale è quella di sostituire ogni simbolo nel testo compresso con l'indice dell'ultima posizione alfabetica utilizzata. Dopo aver sostituito i successivi caratteri con una posizione nell'alfabeto, quel carattere viene spostato a sinistra nella posizione zero e l'algoritmo procede al carattere successivo.
Diamo ora una descrizione più dettagliata dell'algoritmo.
Sia "banana" una stringa nell'alfabeto "abcdefghijklmnopqrstuvwxyz" ed applichiamo la Move-to-Front (MTF). Il primo carattere della stringa "banana" è la lettera "b", che compare nella posizione 1 dell'alfabeto. L'algoritmo aggiunge il carattere "b" alla stringa di destinazione come output. Successivamente, sposta il carattere "b" in cima all'elenco e genera 'bacdefghijklmnopqrstuvwxyz'. Il carattere successivo è "a" e nell'alfabeto "bacdefghijklmnopqrstuvwxyz" appare in posizione 1, quindi aggiunge il valore "a" alla stringa di output e spostia la lettera "a" all'inizio dell'elenco per ottenere "abcdefghijklmnopqrstuvwxyz". Iterando il processo, si ottiene che la stringa "banana" viene codificata come "1,1,13,1,1,1".

Iteration	Sequence	List
banana	1	(abcdefghijklmnopqrstuvwxyz)
b**a**nana	1,1	(bacdefghijklmnopqrstuvwxyz)
ba**n**ana	1,1,13	(abcdefghijklmnopqrstuvwxyz)
ban**a**na	1,1,13,1	(nabcdefghijklmopqrstuvwxyz)
bana**n**a	1,1,13,1,1	(anbcdefghijklmopqrstuvwxyz)
banan**a**	1,1,13,1,1,1	(nabcdefghijklmopqrstuvwxyz)

Da notare che la trasformazione è reversibile. Infatti è sufficiente mantenere lo stesso elenco e decodificare sostituendo ogni indice del flusso codificato con la lettera che si trova a quell'indice nell'elenco.

3.6 Run Length Encoding

Run-length encoding (RLE) è una forma di compressione dei dati lossless in cui una serie di dati (sequenze in cui lo stesso valore di dati ricorre in

molti elementi di dati consecutivi) "run" vengono memorizzati come un singolo valore e conteggio di dati, anziché come la serie originale. L'obiettivo di tale algoritmo è quello di memorizzare queste stringhe andando a sostituire le run di valori uguali con il valore seguito da un contatore che indica quante volte quel carattere deve essere ripetuto, questo implica la reversibilità dell'algoritmo.
Consideriamo la seguente stringa:
`WWWWWWWWWWWWBWWWWWWWWWWWWWWBBBWWWWWWWWWWWWWWWWWWWWWWWWBWWWWWWWWWWWWWW`
Possiamo notare che sono presenti delle sequenze ripetute di lettere, applicando la RLE si ottiene la stringa:
`12W1B12W3B24W1B14W`
La stringa può essere interpretata come una sequenza di 12 W, 1 B, 12 W, 3 B e così via. Da notare che la sequenza originale è costituita da 67 caratteri, mentre quella ottenuta applicando la RLE è di 18. Questo comporta un enorme vantaggio perché ci permette di ridurre lo spazio di archiviazione necessario.

3.7 Trasformata di Burrows-Wheeler

La trasformata di Burrows-Wheeler (BWT) detta anche *block-sorting compression*, ha lo scopo di formattare una stringa in modo tale da migliorare l'efficienza degli algoritmi di compressione al costo di qualche calcolo computazionale in più (la trasformata ha complessità lineare).
La trasformata prende in input una stringa e dà in output una coppia composta da una stringa (stringa riordinata) e un indice.
L'algoritmo è reversibile e questo ci permette di riottenere la stringa iniziale a partire dalla coppia data in output dalla trasformata.
Prima di andare ad analizzare nel dettaglio la trasformata diamo alcune definizioni di base:

- **Alfabeto**: Insieme finito di elementi, detti **caratteri**, di cardinalità almeno uno. Un alfabeto viene denotato come $\Sigma = \{s_1, s_2, ..., s_k\}$.
- **Stringa**: Una sequenza di caratteri di un alfabeto Σ.

Definizione formale La trasformata di Burrows-Wheeler (BWT) mappa parole w di lunghezza n in una coppia (L, I) dove L è una parola di lunghezza n e I è un indice in $\{1, ..., n\}$. La parola L è solitamente indicata come la trasformata di Burrows-Wheeler di w.

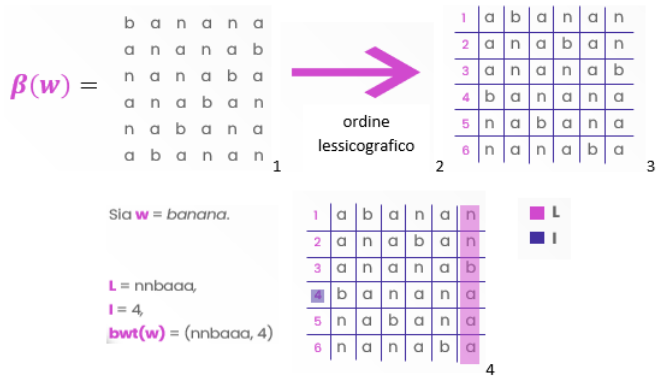


Figure 3: Procedimento BWT

Procedimento Per poter ottenere la $bwt(w)$ occorre innanzitutto costruire la $bwt - matrix$ (vedi figura 3) nel seguente modo:.

1. Si costruiscono tutte le coniugate della stringa w. L'insieme di tutte le coniugate di w definisce la matrice $\beta(w)$.

2. Si ordinano le coniugate in ordine lessicografico.

3. L'insieme di tutte le coniugate ordinate formerà l'insieme delle righe della $bwt - matrix$.

4. L sarà l'ultima colonna della $bwt - matrix$ e I l'indice della riga che contiene la parola w.

Reversibilità della BWT La **BWT** è invertibile, ovvero, dato *bwt(L(w), I)*, dove $L(w)$ è la stringa della parola w e I è l'indice di w in *bwt-matrix*(w) si può ricostruire la parola w. L'idea principale è di effettuare un ordinamento lessicografico della stringa L creando una nuova stringa F di w. Successivamente viene applicata la permutazione standard su F. Quindi, sia f_L la permutazione standard indotta su L, e sia $f_L^t(I)$ la t-esima applicazione della permutazione standard su I, allora

$$w = \lambda_L(f_L^1(I) * ... * \lambda_L(f_L^{|L|}(I))$$

Infine, trovate tutte le permutazioni, si calcolano i valori di $\lambda_L(I)$ per i valori ottenuti dalle permutazioni, avendo cosi come risultato la parola originale w.
Diamo ora una descrizione più dettagliata dell'algoritmo fornendo un esempio.
Consideriamo la stringa *banana* come input. Applicando la BWT si ottiene $bwt(w) = (nnbaaa, 4)$. Successivamente, viene effettuato un ordinamento

lessicografico su *nnbaaa*, così da ottenere la stringa *aaabnn*. Effettuando la permutazione standard si ha

$$f_L = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 4 & 5 & 6 & 3 & 1 & 2 \end{pmatrix}$$

Infine viene applicata la funzione $\lambda_L(I)$ per i valori ottenuti così da avere

$$w = \lambda_L(3)\lambda_L(6)\lambda_L(2)\lambda_L(5)\lambda_L(1)\lambda_L(4) = banana$$

3.8 Transformata di Burrows-Wheeler biettiva

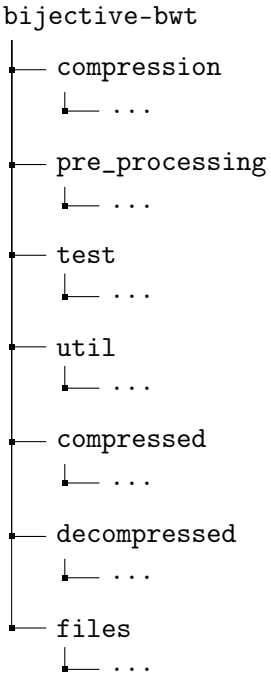
Di seguito viene presentata la variante biettiva della trasformata di Burrows Wheeler riportata nello studio [1]. Per dimostrare la correttezza della BWT biettiva, dobbiamo dimostrare che si tratta di una biezione tra l'insieme delle parole di lunghezza n e l'insieme delle parole di lunghezza n. Sia w una parola di lunghezza n e sia $BWTS(w) = \{u_1, ..., u_n\}$ il risultato della BWT biettiva di w. Dobbiamo dimostrare che esiste un'unica parola w' di lunghezza n tale che $BWTS(w') = \{u_1, ..., u_n\}$.
Per fare ciò, notiamo innanzitutto che la fattorizzazione di Lyndon di w' è unica. Questo segue dal fatto che la fattorizzazione Lyndon di una parola è unica e che la fattorizzazione Lyndon di w' è determinata dal multiset $\{u_1, .., u_n\}$.
Supponiamo ora che esista una parola w' di lunghezza n tale che $BWTS(w') = u_1, ..., u_n$. Allora, per definizione, la fattorizzazione di Lyndon di $w' = \{v_s, ..., v_1\}$ con $v_s \geq ... \geq v1$ e dove il multiset $\{u_1, ..., u_n\} = \bigcup_{i=1}^{s}[v_i]$. Occorre ora dimostrare che w' è l'unica parola di lunghezza n tale che la sua fattorizzazione di Lyndon è $\{v_s, ..., v_1\}$ e dove il multiset $\{u_1, ..., u_n\} = \bigcup_{i=1}^{s}[v_i]$.
Per fare ciò, notiamo che la fattorizzazione di Lyndon di w' è determinata dal multiset $\{u_1, ..., u_n\}$. Ciò deriva dal fatto che la fattorizzazione di Lyndon di una parola è determinata dal suo insieme di sottoparole.
Supponiamo ora che esista una parola w" di lunghezza n tale che la sua fattorizzazione di Lyndon sia $\{vs, ..., v1\}$ e che il multiset $\{u_1, ..., u_n\} = \bigcup_{i=1}^{s}[vi]$.
Dobbiamo dimostrare che $w'' = w'$. Per farlo, notiamo che l'insieme $\{u_1, ..., u_n\}$ è determinato dalla fattorizzazione di Lyndon di w". Ciò deriva dal fatto che l'insieme delle sottoparole di una parola è determinato dalla sua fattorizzazione di Lyndon. Pertanto, poiché il multiset $\{u_1, ..., u_n\}$ è determinato dalla fattorizzazione di Lyndon sia di w' che di w'', ne segue che $w' = w''$.
Questo dimostra che il BWT biettivo è una biezione tra l'insieme di parole di lunghezza n e l'insieme di parole di lunghezza n. Pertanto, il BWT biettivo è corretto.

4 Implementazione

In questo capitolo discuteremo l'organizzazione e l'implementazione della soluzione trovata per adempire agli obiettivi del progetto.

4.1 Struttura del progetto

Nei paragrafi successivi verrà fornita una breve descrizione delle directory e per ognuna di esse una spiegazione di ogni script e di ogni modulo implementato con il linguaggio di riferimento *Python*. Di seguito è raffigurata la struttura del progetto.



4.2 Directory

In questo paragrafo analizzeremo l'organizzazione del progetto e di ciascuna directory in esso, e le scelte che hanno portato a tale strutturazione.

4.2.1 Compression

Nella directory `compression` sono presenti file contenenti le implementazioni degli algoritmi di compressione. Di seguito è raffigurata la struttura della directory `compression`.

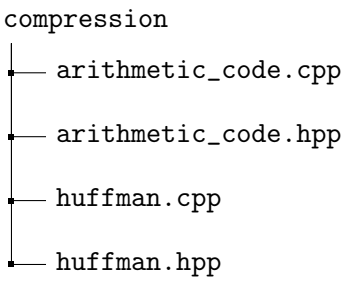


Figure 4: Struttura directory compression

Huffman

La sezione riguardante Huffman comprende due file, huffman.hpp e huffman.cpp. Il primo espone le strutture dati utilizzate e le firme dei metodi da invocare, il secondo le implementazioni dei metodi.
Nel file huffman.hpp è presente la struct `Node` definita nel seguente modo:

```
1  struct Node {
2      char data;
3      unsigned freq;
4      string code;
5      Node *left, *right;
6
7      Node() {
8          left = right = NULL;
9      }
10 };
```

Viene definita anche la classe `huffman` come di seguito:

```
1  class huffman {
2      private:
3          vector <Node*> arr;
4
5          fstream inFile, outFile;
6
7          string inFileName, outFileName;
8
9          Node *root;
10
11         class Compare {
12             public:
13                 bool operator() (Node* l, Node* r)
14                 {
15                     return l->freq > r->freq;
```

```
16                  }
17          };
18
19          priority_queue <Node*, vector<Node*>, Compare> minHeap;
20
21          #Private Methods...
22
23      public:
24          //Constructor
25          huffman(string inFileName, string outFileName)
26          {
27              this->inFileName = inFileName;
28              this->outFileName = outFileName;
29              createArr();
30          }
31
32          #Public Methods...
33  }
```

I metodi privati, utili per effettuare la codifica di Huffman, sono i seguenti:

- `void creareArr()`: inizializza un vettore di nodi dell'albero che rappresentano i valori dei caratteri ASCII e inizializza le loro frequenze a 0.

```
1  void huffman::createArr() {
2      for (int i = 0; i < 256; i++) {
3          arr.push_back(new Node());
4          arr[i]->data = i;
5          arr[i]->freq = 0;
6      }
7  }
```

- `void traverse(Node*, string)`: percorre l'albero costruito al fine di generare la codifica di Huffman per ogni carattere.

```
1  void huffman::traverse(Node* r, string str) {
2      if (r->left == NULL && r->right == NULL) {
3          r->code = str;
4          return;
5       }
6
7       traverse(r->left, str + '0');
8      traverse(r->right, str + '1');
9  }
```

- `int binToDec(string)`: converte una stringa binaria al suo valore decimale.

```
int huffman::binToDec(string inStr) {
    int res = 0;
    for (auto c : inStr) {
         res = res * 2 + c - '0';
     }
     return res;
}
```

- `string decToBin(int)`: converte un numero decimale al suo valore binario.

```
string huffman::decToBin(int inNum) {
    string temp = "", res = "";
    while (inNum > 0) {
        temp += (inNum % 2 + '0');
        inNum /= 2;
    }
     res.append(8 - temp.length(), '0');
     for (int i = temp.length() - 1; i >= 0; i--) {
     res += temp[i];
     }
     return res;
}
```

- `void buildTree(char, string&)`: costruisce l'albero di Huffman mentre viene effettuata la decodifica del file.

```
void huffman::buildTree(char a_code, string& path) {
    Node* curr = root;
    for (long unsigned int i = 0; i < path.length(); i++) {
        if (path[i] == '0') {
            if (curr->left == NULL) {
                curr->left = new Node();
            }
            curr = curr->left;
        }
        else if (path[i] == '1') {
            if (curr->right == NULL) {
                curr->right = new Node();
            }
            curr = curr->right;
        }
    }
```

```
17      curr->data = a_code;
18  }
```

- `void createMinHeap()`: legge il file da comprimere e costruisce l'albero di Huffman, ovvero un MinHeap.

```
1   void huffman::createMinHeap() {
2       inFile.open(inFileName, ios::binary | ios::in);
3       //Incremento della frequenza dei caratteri
4       //che appaiono nel file di input
5       while (true) {
6           int id;
7           id = inFile.get();
8           if (inFile.eof())
9                   break;
10          arr[id]->freq++;
11      }
12      inFile.close();
13      //Inserimento dei nodi che compaiono nel file
14      //nella coda di priorità (Min Heap)
15      for (int i = 0; i < 256; i++) {
16          if (arr[i]->freq > 0) {
17              minHeap.push(arr[i]);
18          }
19      }
20  }
```

- `void createMinHeapData(string data)`: aggiorna le frequenze dei carattere e crea un nodo basato su quest'ultime.

```
1   void huffman::createMinHeapData(string data) {
2       //Aggiorna le frequenze dei caratteri
3       //che appaiono nel file in input
4       for (unsigned char c: data) {
5           arr[c]->freq++;
6       }
7       //Inserimento dei nodi che compaiono nel file
8       //nella coda di priorità (Min Heap)
9       for (int i = 0; i < 256; i++) {
10          if (arr[i]->freq > 0) {
11              minHeap.push(arr[i]);
12          }
13      }
14  }
```

- `void createTree()`: costruisce l'albero di Huffman.

```
void huffman::createTree() {
    //Creazione dell'albero di Huffman con il
    //Min Heap creato precedentemente
    Node *left, *right;
    priority_queue <Node*, vector<Node*>, Compare> tempPQ(minHeap);
    while (tempPQ.size() != 1)
    {
        left = tempPQ.top();
        tempPQ.pop();

        right = tempPQ.top();
        tempPQ.pop();

        root = new Node();
        root->freq = left->freq + right->freq;

        root->left = left;
        root->right = right;
        tempPQ.push(root);
    }
}
```

- `void createCodes()`: genera i codici di Huffman.

```
void huffman::createCodes() {
    //Attraverso l'albero di Huffman
    //e assegno codici specifici a ciascun carattere
    traverse(root, "");
}
```

- `void saveEncodedFile()`: effettua il salvataggio della codifica nel file di output.

```
void huffman::saveEncodedFile() {
    //Salvo il file compresso (.huf)
    inFile.open(inFileName, ios::in);
    outFile.open(outFileName, ios::out | ios::binary);
    string in = "";
    string s = "";

    //Salvo i meta data (huffman tree)
    char _f, _s;
    if (minHeap.size() > 255) {
        _f = (char)255;
        _s = (char)(minHeap.size() - 255);
```

```
13      } else {
14          _f = (char)0;
15          _s = (char)minHeap.size();
16      }
17      in += _f;
18      in += _s;
19      priority_queue <Node*, vector<Node*>, Compare> tempPQ(minHeap);
20      while (!tempPQ.empty()) {
21          Node* curr = tempPQ.top();
22          in += curr->data;
23          //Salvo i 16 valori decimali che
24          //rappresentano la coda del carattere di curr->data
25          s.assign(127 - curr->code.length(), '0');
26          s += '1';
27          s += curr->code;
28          //Salvo i valori decimali di ogni codice binario a 8 bit
29          in += (char)binToDec(s.substr(0, 8));
30          for (int i = 0; i < 15; i++) {
31              s = s.substr(8);
32              in += (char)binToDec(s.substr(0, 8));
33          }
34          tempPQ.pop();
35      }
36      s.clear();
37      //Salvo i codici di ogni carattere che compaiono nel file di input
38      while (true) {
39          int id;
40          id = inFile.get();
41          if (inFile.eof())
42                  break;
43          s += arr[id]->code;
44          //Salvo i valori decimali di ogni codice binario a 8 bit
45          while (s.length() > 8) {
46              in += (char)binToDec(s.substr(0, 8));
47              s = s.substr(8);
48          }
49      }
50
51      //Se i bit restanti sono meno di 8, si aggiungono gli 0
52      int count = 8 - s.length();
53      if (s.length() < 8) {
54          s.append(count, '0');
55      }
56      in += (char)binToDec(s);
```

```
57      //Aggiungi il conteggio degli 0 inseriti
58      in += (char)count;
59
60      //Scrivi la stringa nel file di output
61      outFile.write(in.c_str(), in.size());
62      inFile.close();
63      outFile.close();
64  }
```

- `void saveEncodedFileData(string data)`: effettua il salvataggio della codifica.

```
1   void huffman::saveEncodedFileData(string data) {
2       //Salvo il file compresso (.huf)
3       outFile.open(outFileName, ios::out | ios::binary);
4       string in = "";
5       string s = "";
6
7       //Salvo i meta data (huffman tree)
8       char _f, _s;
9       if (minHeap.size() > 255) {
10          _f = (char)255;
11          _s = (char)(minHeap.size() - 255);
12      } else {
13          _f = (char)0;
14          _s = (char)minHeap.size();
15      }
16      in += _f;
17      in += _s;
18      priority_queue <Node*, vector<Node*>, Compare> tempPQ(minHeap);
19      while (!tempPQ.empty()) {
20          Node* curr = tempPQ.top();
21          in += curr->data;
22          //Salvo i 16 valori decimali che
23          //rappresentano il codice di curr->data
24          s.assign(127 - curr->code.length(), '0');
25          s += '1';
26          s += curr->code;
27          //Salvo i valori decimali di ogni codice binario a 8 bit
28          in += (char)binToDec(s.substr(0, 8));
29          for (int i = 0; i < 15; i++) {
30              s = s.substr(8);
31              in += (char)binToDec(s.substr(0, 8));
32          }
```

```
            tempPQ.pop();
        }
        s.clear();

        //Salvo i codici di ogni carattere che appaiono nel file di input
        for (unsigned char c: data) {
            s += arr[c]->code;
            //Salvo i valori decimali di ogni codice binario a 8 bit
            while (s.length() > 8) {
                in += (char)binToDec(s.substr(0, 8));
                s = s.substr(8);
            }
        }

        //Se i bit restanti sono meno di 8, si aggiungono gli 0
        int count = 8 - s.length();
        if (s.length() < 8) {
            s.append(count, '0');
        }
        in += (char)binToDec(s);
        //Aggiungi il conteggio degli 0 inseriti
        in += (char)count;

        //Scrivi la stringa nel file di output
        outFile.write(in.c_str(), in.size());
        outFile.close();
    }
```

- `void saveDecodedFile()`: effettua il salvataggio della decodifica nel file di output.

```
void huffman::saveDecodedFile() {
    inFile.open(inFileName, ios::in | ios::binary);
    outFile.open(outFileName, ios::out);
    unsigned char f;
    inFile.read(reinterpret_cast<char*>(&f), 1);
    unsigned char s;
    inFile.read(reinterpret_cast<char*>(&s), 1);
    int size = f + s;
    //Lettura del conteggio alla fine del file che è
    //il numero di bit aggiunti per rendere il valore finale di 8 bit
    inFile.seekg(-1, ios::end);
    char count0;
    inFile.read(&count0, 1);
```

```
14      //Ignoro i meta data (huffman tree) (1 + 17 * size)
15      //e leggo il file restante
16      inFile.seekg(2 + 17 * size, ios::beg);
17
18      vector<unsigned char> text;
19      unsigned char textseg;
20      inFile.read(reinterpret_cast<char*>(&textseg), 1);
21      while (!inFile.eof()) {
22          text.push_back(textseg);
23          inFile.read(reinterpret_cast<char*>(&textseg), 1);
24      }
25
26      Node *curr = root;
27      string path;
28      for (long unsigned int i = 0; i < text.size() - 1; i++) {
29          //Converto il numero decimale nel suo
30          //codice binario a 8 bit equivalente
31          path = decToBin(text[i]);
32          if (i == text.size() - 2) {
33              path = path.substr(0, 8 - count0);
34          }
35          //Attraverso l'albero di Huffman e aggiungo i risultati nel file
36          for (long unsigned int j = 0; j < path.size(); j++) {
37              if (path[j] == '0') {
38                  curr = curr->left;
39              }
40              else {
41                  curr = curr->right;
42              }
43
44              if (curr->left == NULL && curr->right == NULL) {
45                  outFile.put(curr->data);
46                  curr = root;
47              }
48          }
49      }
50      inFile.close();
51      outFile.close();
52  }
```

- `string saveDecodedFileData()`: effettua il salvataggio della decodifica.

```
1  string huffman::saveDecodedFileData() {
```

```
    inFile.open(inFileName, ios::in | ios::binary);
    string output;
    unsigned char f;
    inFile.read(reinterpret_cast<char*>(&f), 1);
    unsigned char s;
    inFile.read(reinterpret_cast<char*>(&s), 1);
    int size = f + s;
    inFile.seekg(-1, ios::end);
    char count0;
    inFile.read(&count0, 1);
    //Ignoro i meta data (huffman tree) (1 + 17 * size)
    //e leggo il file restante
    inFile.seekg(2 + 17 * size, ios::beg);

    vector<unsigned char> text;
    unsigned char textseg;
    inFile.read(reinterpret_cast<char*>(&textseg), 1);
    while (!inFile.eof()) {
        text.push_back(textseg);
        inFile.read(reinterpret_cast<char*>(&textseg), 1);
    }
    Node *curr = root;
    string path;
    for (long unsigned int i = 0; i < text.size() - 1; i++) {
        //Converto il numero decimale nel suo
        //codice binario a 8 bit equivalente
        path = decToBin(text[i]);
        if (i == text.size() - 2) {
            path = path.substr(0, 8 - count0);
        }
        //Attraverso l'albero di Huffman e aggiungo i risultati nel file
        for (long unsigned int j = 0; j < path.size(); j++) {
            if (path[j] == '0') {
                curr = curr->left;
            }
            else {
                curr = curr->right;
            }
            if (curr->left == NULL && curr->right == NULL) {
                output += curr->data;
                curr = root;
            }
        }
    }
```

```
46      inFile.close();
47      return output;
48  }
```

- `void getTree()`: legge il file per ricostruire l'albero di Huffman.

```
1   void huffman::getTree() {
2       inFile.open(inFileName, ios::in | ios::binary);
3       //Leggo la taglia di MinHeap
4       unsigned char f;
5       inFile.read(reinterpret_cast<char*>(&f), 1);
6       unsigned char s;
7       inFile.read(reinterpret_cast<char*>(&s), 1);
8       int size = f + s;
9       root = new Node();
10      for(int i = 0; i < size; i++) {
11          char aCode;
12          unsigned char hCodeC[16];
13          inFile.read(&aCode, 1);
14          inFile.read(reinterpret_cast<char*>(hCodeC), 16);
15          //Converto i caratteri decimali nel loro
16          //equivalente binario per ottenere il codice
17          string hCodeStr = "";
18          for (int i = 0; i < 16; i++) {
19              hCodeStr += decToBin(hCodeC[i]);
20          }
21          //Rimozione del padding precedentemente ignorato (127 - curr->code.l
22          //Carattere 0 e il successivo 1
23          int j = 0;
24          while (hCodeStr[j] == '0') {
25              j++;
26          }
27          hCodeStr = hCodeStr.substr(j+1);
28          //Aggiungo un nodo con i dati aCode
29          //e la string hCodeStr all'albero di Huffman
30          buildTree(aCode, hCodeStr);
31      }
32      inFile.close();
33  }
```

I metodi pubblici, esposti per effettuare la compressione e la decompressione di un file sono i seguenti:

- `void compress()`: effettua la compressione di un file e lo scrive nel file di output.

```
void huffman::compress() {
    createMinHeap();
    createTree();
    createCodes();
    saveEncodedFile();
}
```

- `void compressData(string data)`: effettua la compressione della stringa data in input e lo scrive nel file di output

```
void huffman::compressData(string data) {
    createMinHeapData(data);
    createTree();
    createCodes();
    saveEncodedFileData(data);
}
```

- `void decompress()`: effettua la decompressione di un file e lo scrive nel file di output.

```
void huffman::decompress() {
    getTree();
    saveDecodedFile();
}
```

- `string decompressData()`: effettua la decompressione e restituisce il risultato ottenuto

```
string huffman::decompressData() {
    getTree();
    return saveDecodedFileData();
}
```

Arithmetic coding

La sezione riguardante Arithmetic coding comprende due file, arithmetic_code.hpp e arithmetic_code.cpp. Il primo espone le strutture dati utilizzate e le firme dei metodi da invocare, il secondo le implementazioni dei metodi.
Il file arithmetic_code.hpp comprende tre classi: `Compress`, `Encode`, `Decode` definite nel seguente modo:

- `Compress`: classe utilizzata per aggiornare la tabella delle frequenze dei caratteri.

```
class Compress{
    public:
        unsigned char index_to_char [NO_OF_SYMBOLS];
```

```
4           int char_to_index [NO_OF_CHARS];
5           int cum_freq [NO_OF_SYMBOLS + 1];
6           int freq [NO_OF_SYMBOLS + 1];
7
8           Compress(void);
9           ~Compress(void);
10          void update_tables(int sym_index);
11  };
```

- `Encode`: classe utilizzata per effettuare l'encoding dei caratteri e la scrittura della codifica all'interno di un file.

```
1   class Encode : public Compress{
2       int low, high;
3       int opposite_bits;
4       int buffer;
5       int bits_in_buf;
6
7       ifstream in;
8       ofstream out;
9       public:
10          Encode(void);
11          ~Encode(void);
12
13          void write_bit( int bit);
14          void output_bits(int bit);
15          void end_encoding(void);
16          void encode_symbol(int symbol);
17          void encode(const char *infile, const char *outfile);
18          void encodeData(std::string str_in, const char *outfile);
19  };
```

- `Decode`: classe utilizzata per effettuare il decoding dei caratteri e la scrittura della decodifica all'interno di un file.

```
1   class Decode : public Compress{
2       int low, high;
3       int value;
4
5       int buffer;
6       int bits_in_buf;
7       bool end_decoding;
8
9       ifstream in;
10      ofstream out;
```

```
11      public:
12          Decode(void);
13          ~Decode(void);
14
15          void load_first_value(void);
16          void decode(const char *infile, const char *outfile);
17          std::string decodeData(const char *infile);
18          int decode_symbol(void);
19          int get_bit(void);
20  };
```

Di seguito le implementazioni dei metodi esposti:

- `Encode::Encode(void)`:

```
1       Encode::Encode(void){
2       buffer = 0;
3       bits_in_buf = 0;
4
5       low = 0;
6       high = MAX_VALUE;
7       opposite_bits = 0;
8   }
```

- `void Encode::encode(const char *infile, const char *outfile)`:

```
1   void Encode::encode(const char *infile, const char *outfile){
2       in.open(infile, ios::binary | ios::in);
3       out.open(outfile, ios::binary | ios::out);
4       if (!in || !out){
5           cout<<"Error: Can`t open file"<<endl;
6           return;
7       }
8       while (true){
9           int ch;
10          int symbol;
11          ch = in.get();
12          if (in.eof()){
13              break;
14          }
15          symbol = char_to_index[ch];
16          encode_symbol(symbol);
17          update_tables(symbol);
18      }
19      encode_symbol(EOF_SYMBOL);
20      end_encoding();
```

```
    in.close();
    out.close();
}
```

- void Encode::encodeData(std::string str_in, const char *outfile):

```
void Encode::encodeData(std::string str_in, const char *outfile){
    out.open(outfile, ios::binary | ios::out);
    if (!out) {
        cout<< "Error: Can`t open file" <<endl;
        return;
    }
    int i = 0;
    while (true) {
        int ch;
        int symbol;
        ch = (unsigned char) str_in[i++];
        if (str_in[i - 1] == EOF || str_in[i - 1] ==0) {
            break;
        }
        symbol = char_to_index[ch];
        encode_symbol(symbol);
        update_tables(symbol);
    }
    encode_symbol(EOF_SYMBOL);
    end_encoding();
    in.close();
    out.close();
}
```

- void Encode::encode_symbol(int symbol):

```
void Encode::encode_symbol(int symbol){
    int range;

    range = high - low;
    high = low + (range * cum_freq [symbol - 1]) / cum_freq [0];
    low = low + (range * cum_freq [symbol]) / cum_freq [0];
    for (;;){
        if (high < HALF)
            output_bits(0);
        else if (low >= HALF){
            output_bits(1);
            low -= HALF;
            high -= HALF;
```

```
        }
        else if (low >= FIRST_QTR && high < THIRD_QTR){
            opposite_bits++;
            low -= FIRST_QTR;
            high -= FIRST_QTR;
        }
        else
            break;
        low = 2 * low;
        high = 2 * high;
    }
}
```

- void Encode::end_encoding(void):

```
void Encode::end_encoding(void){
    opposite_bits++;
    if (low < FIRST_QTR)
        output_bits(0);
    else
        output_bits(1);

    out.put(buffer >> bits_in_buf);
}
```

- void Encode::output_bits(int bit):

```
void Encode::output_bits(int bit){
    write_bit(bit);
    while (opposite_bits > 0){
        write_bit(!bit);
        opposite_bits--;
    }
}
```

- void Encode::write_bit(int bit):

```
void Encode::write_bit(int bit){
    buffer >>= 1;
    if (bit)
        buffer |= 0x80;
    bits_in_buf++;
    if (bits_in_buf == 8){
        out.put(buffer);
        bits_in_buf = 0;
```

```
    }
}
```

- `Compress::Compress(void)`:

```
Compress::Compress(void){
    int i;
    for ( i = 0; i < NO_OF_CHARS; i++){
        char_to_index[i] = i + 1;
        index_to_char[i + 1] = i;
    }
    for ( i = 0; i <= NO_OF_SYMBOLS; i++){
        freq[i] = 1;
        cum_freq[i] = NO_OF_SYMBOLS - i;
    }
    freq[0] = 0;
}
```

- `void Compress::update_tables(int sym_index)`:

```
void Compress::update_tables(int sym_index){
    int i;
    if (cum_freq[0] == MAX_FREQ){
        int cum = 0;
        for ( i = NO_OF_SYMBOLS; i >= 0; i--){
            freq[i] = (freq[i] + 1) / 2;
            cum_freq[i] = cum;
            cum += freq[i];
        }
    }
    for ( i = sym_index; freq[i] == freq[i - 1]; i--);
    if (i < sym_index){
        int ch_i, ch_symbol;
        ch_i = index_to_char[i];
        ch_symbol = index_to_char[sym_index];
        index_to_char[i] = ch_symbol;
        index_to_char[sym_index] = ch_i;
        char_to_index[ch_i] = sym_index;
        char_to_index[ch_symbol] = i;
    }
    freq[i]++;
    while (i > 0){
        i--;
        cum_freq[i]++;
```

```
25      }
26  }
```

- Decode::Decode(void):

```
1   Decode::Decode(void){
2       buffer = 0;
3       bits_in_buf = 0;
4       end_decoding = false;
5
6       low = 0;
7       high = MAX_VALUE;
8   }
```

- void Decode::load_first_value(void):

```
1   void Decode::load_first_value(void){
2       value = 0;
3       for (int i = 1; i <= CODE_VALUE; i++)
4           value = 2 * value + get_bit();
5   }
```

- void Decode::decode(const char *infile, const char *outfile):

```
1   void Decode::decode(const char *infile, const char *outfile){
2       in.open(infile, ios::binary | ios::in);
3       out.open(outfile, ios::binary | ios::out);
4       if (!in || !out) {
5           cout<<"Error: Can`t open file"<<endl;
6           return;
7       }
8       load_first_value();
9       while (true){
10          int ch;
11          int sym_index;
12          sym_index = decode_symbol();
13          if ((sym_index == EOF_SYMBOL) || end_decoding)
14              break;
15          ch = index_to_char[sym_index];
16          out.put(ch);
17          update_tables(sym_index);
18      }
19      in.close();
20      out.close();
21  }
```

- std::string Decode::decodeData(const char *infile):

```
std::string Decode::decodeData(const char *infile) {
    std::string str_out = "";
    char app_char;
    std::string s;
    in.open(infile, ios::binary | ios::in);
    if (!in) {
        cout << "Error: Can`t open file" << endl;
        return "";
    }
    load_first_value();
    while (true) {
        int ch;
        int sym_index;
        sym_index = decode_symbol();
        if ((sym_index == EOF_SYMBOL) || end_decoding)
            break;
        ch = index_to_char[sym_index];
        app_char = (char)ch;
        str_out.push_back(app_char);
        update_tables(sym_index);
    }
    in.close();
    out.close();
    return str_out;
}
```

- int Decode::decode_symbol(void):

```
int Decode::decode_symbol(void){
    int range;
    int cum;
    int symbol_index;

    range = high - low;
    cum = ((((value - low) + 1) * cum_freq[0] - 1) / range);
    for (symbol_index = 1; cum_freq[symbol_index] > cum; symbol_index++);
    high = low + (range * cum_freq[symbol_index - 1]) / cum_freq[0];
    low = low + (range * cum_freq[symbol_index]) / cum_freq[0];
    for (;;){
        if (high < HALF){
                }
        else if (low >= HALF){
            value -= HALF;
```

```
            low -= HALF;
            high -= HALF;
        }
        else if (low >= FIRST_QTR && high < THIRD_QTR){
            value -= FIRST_QTR;
            low -= FIRST_QTR;
            high -= FIRST_QTR;
        }
        else
            break;
        low = 2 * low;
        high = 2 * high;
        value = 2 * value + get_bit();
    }
    return symbol_index;
}
```

- `int Decode::get_bit(void)`:

```
int Decode::get_bit(void){
    int t;
    if (bits_in_buf == 0){
        buffer = in.get();
        if (buffer == EOF){
            end_decoding = true;
            return -1;
        }
        bits_in_buf= 8;
    }
    t = buffer & 1;
    buffer >>= 1;
    bits_in_buf--;
    return t;
}
```

LZW

Il file `lzw.cpp` contiene due funzioni

- `lzwCompress(uncompressed)`: comprime i dati presi in input utilizzando la costruzione del dizionario.
- `lzwDecompress(compressed)`: decomprime i dati passati ricostruendo il dizionario e controllando che i valori siano presenti nel dizionario, in caso di errore restituisce il messaggio: "bad compressed".

4.2.2 Files

La cartella `files` contiene i file utilizzati per effettuare i test sugli algoritmi di compressione. Dopo l'esecuzione degli algoritmi vengono create due sottocartelle `compressed` e `decompressed` che contengono rispettivamente i file compressi e decompressi.

4.2.3 Pre-processing

La directory `pre-processing` contiene le implementazioni degli algoritmi di pre-processing sui dati. Di seguito viene riportata la struttura della directory.

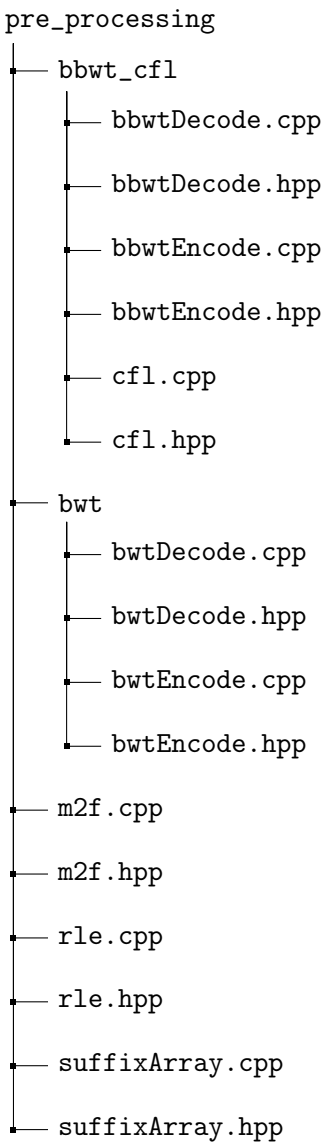


Figure 5: Struttura directory pre-processing

Trasformata di Burrows-Wheeler (BWT)
L'implementazione della BWT è suddivisa in due file, il file `bwtEncode.cpp`, contenente l'implementazione dei metodi dichiarati in `bwtEncode.hpp`, utile ad effettuare la codifica e il file `bwtDecode`, contenente l'implementazione dei metodi dichiarati in `bwtDecode.hpp`, utile ad effettuare la decodifica.
Di seguito i dettagli dell'implementazione della funzione
`string bwtEncode(const string str)`:

```
string bwtEncode(const string str) {
    string _string(str);
    _string.push_back(-1);
    vector<int> sa = suffix_array_manber_myers(_string);
    string out;
    for (int idx : sa) {
        int i = idx - 1;
        out += (i == -1) ? -1 : (unsigned char) _string[idx - 1];
    }
    return out;
}
```

Da notare come nella codifica viene fatto uso della funzione `suffix_array_manber_myers()` utile al calcolo dell'array dei suffissi di una stringa. L'algoritmo è stato proposto nel 1990 dai ricercatori Manber Udi e Myers Gene [4]. Successivamente vengono calcolate le coniugate dell'input e viene restituito il risultato. Nell'output invece dell'indice della matrice BWT è presente il carattere "-1" che indica la fine della stringa.
Di seguito l'implementazione della funzione `string bwtDecode(string str)`:

```
string bwtDecode(string str) {
    vector<map<char, int>> fm = fm_index(str);
    auto offset = createOffsets(fm.back());
    int i = str.find(-1);
    string s = recover_suffix(i, str, fm, offset);
    return s.substr(0, s.length() - 1);
}
```

Il metodo `bwtDecode()` fa uso delle seguenti funzioni di supporto:

- `vector<map<char, int>> fm_index(string bwt)`:

```
vector<map<char, int>> fm_index(string bwt) {
    vector<map<char, int>> fm;
    map<char, int> firstRow;
    for (char c : bwt) {
        firstRow[(int) (unsigned char) c] = 0;
    }
    fm.push_back(firstRow);
    for (char c : bwt) {
        map<char, int> lastRow = fm.back();
        map<char, int> currentRow;
        for (auto const &symbolCount : lastRow) {
            char symbol = symbolCount.first;
            int count = symbolCount.second;
            currentRow[(int) (unsigned char) symbol] = count + (symbol == c)
```

```
        }
        fm.push_back(currentRow);
    }
    return fm;
}
```

- map<char, int> createOffsets(const map<char, int> &lastRow):

```
map<char, int> createOffsets(const map<char, int> &lastRow) {
    map<char, int> offset;
    int n = 0;

    vector<unsigned char> keys;
    for (auto const &pair: lastRow) {
        if (pair.first != - 1)
            keys.push_back(pair.first);
    }
    sort(begin(keys), end(keys));
    keys.push_back(-1);

    for (unsigned long int i = 0; i < keys.size(); i++) {
        char symbol = keys[i];
        int count = lastRow.at(symbol);
        offset[symbol] = n;
        n += count;
    }
    return offset;
}
```

- string recover_suffix(int i,
const string &bwt, const vector<map<char, int>> &fm_index,
const map<char, int> &offset):

```
string recover_suffix(int i, const string &bwt,
const vector<map<char, int>> &fm_index, const map<char, int> &offset) {
    string suffix;
    char c = bwt[i];
    int predecessor = offset.at(c) + fm_index[i].at(c);
    suffix = c + suffix;
    while (predecessor != i) {
        c = bwt[predecessor];
        predecessor = offset.at(c) + fm_index[predecessor].at(c);
        suffix = c + suffix;
    }
```

```
12      return suffix;
13  }
```

Trasformata di Burrows-Wheeler biettiva (BBWT)
L'implementazione della BBWT è basata sull'algoritmo di Scott *et.al* [5] ed è suddivisa in tre parti: file **cfl**, file **bbwtEncode** e file **bbwtDeconde**. I file `cfl.cpp` e `cfl.hpp` sono file contenente le seguenti funzioni:

- `vector<string> cfl(string str)`:
  prende in input una stringa e restituisce la fattorizzazione di Lyndon. Di seguito viene riportato il codice della funzione:

```
1  vector<string> cfl(string str) {
2      vector<string> words;
3      while (str.length() > 0) {
4          int i = 0;
5          int j = 1;
6          while (j < (int) str.length()
7              && (unsigned char) str[i] <= (unsigned char) str[j]) {
8              if (str[i] == str[j]) {
9                  i++;
10             } else {
11                 i = 0;
12             }
13             j++;
14         }
15         int l = j - i;
16         while (i >= 0) {
17             words.push_back(str.substr(0, l));
18             str = str.substr(l);
19             i -= l;
20         }
21     }
22     return words;
23 }
```

I `bbwtEncode.cpp` e `bbwtEncoded.hpp` sono file contenente il metodo per effettuare la codifica tramite la BBWT. Inoltre sono presenti altri metodi di supporto utili alla codifica.
Di seguito i metodi di supporto:

- `bool less_rotation(const std::vector<std::string> &factors, Rotation i, Rotation j)`:

```
1  bool less_rotation(const std::vector<std::string> &factors,
2      Rotation i, Rotation j) {
```

```
    int i_w = i.w, i_r = i.r, i_length = factors[i_w].size();
    int j_w = j.w, j_r = j.r, j_length = factors[j_w].size();
    for (int k = 0; k < i_length * j_length; ++k) {
        if ((int) (unsigned char) factors[i_w][i_r]
            < (int) (unsigned char) factors[j_w][j_r]) {
            return true;
        } else if ((int) (unsigned char) factors[i_w][i_r]
            > (int) (unsigned char) factors[j_w][j_r]) {
            return false;
        }
        i_r++;
        j_r++;
        if (i_r == i_length)
            i_r = 0;
        if (j_r == j_length)
            j_r = 0;
        }
    return false;
}
```

- `std::vector<Rotation> merge_rotation(const std::vector<std::string> &factors, const std::vector<Rotation> &a, const std::vector<Rotation> &b):`

```
std::vector<Rotation> merge_rotation(const std::vector<std::string> &factors,
    int a_length = a.size(), b_length = b.size(),
        length = a_length + b_length;
    std::vector<Rotation> out(length);
    int i = 0, j = 0, k = 0;
    while (i < a_length && j < b_length) {
        if (less_rotation(factors, b[j], a[i])) {
            out[k] = b[j];
            j++;
        } else {
            out[k] = a[i];
            i++;
        }
        k++;
    }
    if (i < a_length) {
        for (int m = i; m < a_length; ++m)
            out[k++] = a[m];
    }
    else if (j < b_length) {
        for (int m = j; m < b_length; ++m)
```

```
            out[k++] = b[m];
        }
        return out;
    }
```

- `std::vector<Rotation> merge_rotations(`
  `const std::vector<std::string> &factors,`
  `std::vector<std::vector<Rotation>> rotations_of_all_factors):`

```
std::vector<Rotation> merge_rotations(
    const std::vector<std::string> &factors,
    std::vector<std::vector<Rotation>> rotations_of_all_factors) {
    std::vector<Rotation> merged;
    while (rotations_of_all_factors.size() > 0) {
        merged = merge_rotation(factors, merged,
            rotations_of_all_factors[0]);
        rotations_of_all_factors.erase(rotations_of_all_factors.begin());
    }
    return merged;
}
```

- `std::vector<Rotation> sort_rotations(`
  `const std::vector<std::string> &factors,`
  `std::vector<std::vector<Rotation>> &rotations_of_all_factors):`

```
std::vector<Rotation> sort_rotations(
    const std::vector<std::string> &factors,
    std::vector<std::vector<Rotation>> &rotations_of_all_factors){
    for(long unsigned int i = 0;
        i < rotations_of_all_factors.size();i++) {
        auto suffix_array = suffix_array_manber_myers(factors[i]);
        for (long unsigned int j = 0;
            j < rotations_of_all_factors[i].size();j++){
            rotations_of_all_factors[i][j].r = suffix_array[j];
        }
    }
    return merge_rotations(factors, rotations_of_all_factors);
}
```

- `std::string bbwtEncode(const std::string str)` utile per effettuare la codifica:

```
std::string bbwtEncode(const std::string str) {
    auto factors = cfl(str);
    std::string out(str.size(), ' ');

```

```
    std::vector<std::vector<Rotation>> rotations_of_factors;
    for (long unsigned int i = 0; i < factors.size();i++) {
        std::vector<Rotation> rotations_of_w(factors[i].size());
        for (long unsigned int j = 0; j < factors[i].size();j++) {
            rotations_of_w[j] = Rotation{(int) i, (int) j};
        }
        rotations_of_factors.push_back(rotations_of_w);
    }
    auto sorted_rotations_of_factors =
        sort_rotations(factors, rotations_of_factors);
    for (long unsigned int i = 0;
        i < sorted_rotations_of_factors.size();i++){
        int i_r = sorted_rotations_of_factors[i].r -1;
        if (i_r < 0) {
            i_r += factors[sorted_rotations_of_factors[i].w].size();
        }
        out[i] = factors[sorted_rotations_of_factors[i].w][i_r];
    }
    return out;
}
```

I `bbwtDecode.cpp` e `bbwtDecode.hpp` sono file contenente il metodo per effettuare la decodifica tramite BBWT. Anche qui sono stati implementati dei metodi di supporto alla decodifica.
Di seguito i metodi di supporto:

- `std::vector<int> _construct_t(const std::string &data)`:

```
std::vector<int> _construct_t(const std::string &data) {
    std::vector<int> t(data.size());
    std::vector<int> counts(65536, 0);
    for (int i = 0; i < (int) data.size(); i++) {
        counts[(unsigned char) data[i]]++;
    }
    std::vector<int> cum_counts(65536, 0);
    for (int i = 1; i < 65536; i++) {
        cum_counts[i] = cum_counts[i - 1] + counts[i - 1];
    }
    for (int i = 0; i < (int) data.size(); i++) {
        t[i] = cum_counts[(unsigned char) data[i]];
        cum_counts[(unsigned char) data[i]]++;
    }
    return t;
}
```

- `std::string bbwtDecode(const std::string &data)` utile alla decodifica:

```
std::string bbwtDecode(const std::string &data) {
    std::vector<int> t = _construct_t(data);
    std::string out(data.size(), '\0');
    int i = data.size() - 1;
    for (int j = 0; j < (int) data.size(); j++) {
        if (t[j] == -1) {
            continue;
        }
        int k = j;
        while (t[k] != -1) {
            out[i] = (unsigned char) data[k];
            i--;
            int k_temp = t[k];
            t[k] = -1;
            k = k_temp;
        }
    }
    return out;
}
```

Move-to-front transform (MTF)

I file `m2f.hpp` e `m2f.cpp` contengono l'implementazione della move-to-front trasform. In particolare nel primo file viene dichiarata la classe `MTF` e vengono esposti i metodi utili all'applicazione della trasformazione.
Di seguito il codice:

```
class MTF {
    unsigned char symbolTable[257];
    public:inline string encode(string str);
    inline string decode(string str);
    private:inline void moveToFront(int i);
    inline void fillSymbolTable();
};
```

Mentre nel secondo file sono presenti le implementazioni dei metodi:

- `string MTF::encode(string str)`

```
string MTF::encode(string str) {
    fillSymbolTable();
    vector<int> output;
    for(string::iterator it = str.begin(); it != str.end(); it++ ) {
        for(int i = 0; i < 257; i++) {
```

```
            if((unsigned char) *it == symbolTable[i]) {
                output.push_back(i);
                moveToFront((int) (unsigned char) i);
                break;
            }
        }
    }

    string r;
    for(vector<int>::iterator it = output.begin(); it != output.end(); it++)
        ostringstream ss;
        ss << *it;
        r += ss.str() + " ";
    }
    return r;
}
```

- string MTF::decode(string str)

```
string MTF::decode(string str) {
    fillSymbolTable();
    istringstream iss(str);
    vector<int> output;
    copy(
        istream_iterator<int>(iss),
        istream_iterator<int>(),
        back_inserter<vector<int> >(output)
    );
    string r;

    for(vector<int>::iterator it = output.begin(); it != output.end(); it++)
        r.append(1, symbolTable[*it]);
        moveToFront(*it);
    }
    return r;
}
```

- void MTF::moveToFront(int i)

```
void MTF::moveToFront(int i) {
    char t = symbolTable[i];
    for(int z = i - 1; z >= 0; z--) {
        symbolTable[z + 1] = symbolTable[z];
    }
```

```
6       symbolTable[0] = t;
7   }
```

- `void MTF::fillSymbolTable()`

```
1   void MTF::fillSymbolTable() {
2       for(int x = 0; x < 256; x++)
3           symbolTable[x] = x;
4       symbolTable[256] = -1;
5   }
```

Run-length encoding (RLE)

I file `rle.hpp` e `mrle.cpp` contengono l'implementazione della run-length encoding. In particolare nel primo file vengono esposti i metodi utili all'applicazione della trasformazione.
Di seguito il codice:

```
1   std::string RunLengthEncoding(const std::string &s);
2   std::string RunLengthDecoding(const std::string &s);
3   std::string rleEncode(const std::string &s);
4   std::string rleDecode(const std::string &s);
```

Mentre nel secondo file sono presenti le implementazioni dei metodi:

- `vector<string> explode(const string& str, const char& ch)`

```
1   vector<string> explode(const string& str, const char& ch) {
2       string next;
3       vector<string> result;
4       // For each character in the string
5       for (string::const_iterator it = str.begin(); it != str.end(); it++) {
6           // If we've hit the terminal character
7           if (*it == ch) {
8               // If we have some characters accumulated
9               if (!next.empty()) {
10                  // Add them to the result vector
11                  result.push_back(next);
12                  next.clear();
13              }
14          } else {
15              // Accumulate the next character into the sequence
16              next += *it;
17          }
18      }
19      if (!next.empty())
20          result.push_back(next);
```

```
    return result;
}
```

- std::string RunLengthEncoding(const std::string &s)

```
std::string RunLengthEncoding(const std::string &s) {
    std::string encoded_string;
    for (int i = 0; i < s.length(); i++) {
        int count = 1;
        while (i + 1 < s.length() && s[i] == s[i + 1]) {
            count++;
            i++;
        }
        encoded_string += std::to_string(count) + '-' + s[i];
    }
    return encoded_string;
}
```

- std::string RunLengthDecoding(const std::string &s)

```
std::string RunLengthDecoding(const std::string &s) {
    std::string decoded_string;
    for (int i = 0; i < s.length(); i++) {
        int count = 0;
        while (isdigit(s[i])) {
            count = count * 10 + (s[i] - '0');
            i++;
        }
        i++;  // skip the separator '-'
        for (int j = 0; j < count; j++) {
            decoded_string += s[i];
        }
    }
    return decoded_string;
}
```

- std::string rleEncode(const std::string &s)

```
std::string rleEncode(const std::string &s) {
    std::string encoded_string;
    vector<string> numbers = explode(s, ' ');
    for (int i = 0; i < numbers.size(); i++) {
        int count = 1;
        while (i + 1 < numbers.size() && numbers[i] == numbers[i + 1]) {
            count++;
            i++;
```

```
            }
            encoded_string += std::to_string(count) + ' ' + numbers[i] + " ";
        }
        return encoded_string;
    }
```

- `std::string rleDecode(const std::string &s)`

```
    std::string rleDecode(const std::string &s) {
        std::string decoded_string;
        vector<string> numbers = explode(s, ' ');
        string num = numbers[numbers.size() -1];
        if (num.size() > 1 && num[0] == '0')
            numbers[numbers.size() - 1] = "0";
        for (int i = 0; i < numbers.size(); i++) {
            int count = stoi(numbers[i]);
            i++;
            for (int j = 0; j < count; j++) {
                decoded_string += numbers[i] + " ";
            }
        }
        return decoded_string;
    }
```

SuffixArray Manber Myers

I file `suffixArray.hpp` e `suffixArray.cpp` contengono l'implementazione del metodo per la costruzione dei suffissi di una stringa. In particolare nel primo file vengono esposti i metodi utili alla creazione dei suffissi
Di seguito il codice:

```
vector <int> suffix_array_manber_myers(const string &str);
```

Mentre nel secondo file sono presenti le implementazioni dei metodi:

- `vector<int> sort_bucket(const string &str, vector<int> &bucket, int order)`

```
vector<int> sort_bucket(const string &str, vector<int> &bucket, int order) {
    map<string, vector<int>> d;
    for (int i : bucket) {
        string key = str.substr(i, order);
        d[key].push_back(i);
    }
    vector<int> result;
    for (auto &kv : d) {
        if (kv.second.size() > 1) {
            vector<int> res = sort_bucket(str, kv.second, order * 2);
```

```
            result.insert(result.end(), res.begin(), res.end());
        }
        else {
            result.push_back(kv.second[0]);
        }
    }
    return result;
};
```

- `vector <int> suffix_array_manber_myers(const string &str)`

```
vector <int> suffix_array_manber_myers(const string &str) {
    vector <int> indices(str.length());
    iota(indices.begin(), indices.end(), 0);
    return sort_bucket(str, indices, 1);
};
```

4.3 Test

Per effettuare i test è stato creato il file `main.cpp` contenente un main che in base agli input ricevuti effettua i test delle varie pipeline implementate.
Di seguito le pipeline testate:

- HUFFMAN
- ARITHMETIC CODE
- BWT -> M2F -> HUFFMAN
- BWT -> M2F -> ARITHMETIC CODE
- BBWT -> M2F -> HUFFMAN
- BBWT -> M2F -> ARITHMETIC CODE
- BWT -> M2F -> RLE -> HUFFMAN
- BWT -> M2F -> RLE -> ARITHMETIC CODE
- BBWT -> M2F -> RLE -> HUFFMAN
- BBWT -> M2F -> RLE -> ARITHMETIC CODE
- RLE -> BWT -> M2F -> RLE -> HUFFMAN
- RLE -> BWT -> M2F -> RLE -> ARITHMETIC CODE
- RLE -> BBWT -> M2F -> RLE -> HUFFMAN
- RLE -> BBWT -> M2F -> RLE -> ARITHMETIC CODE

- LZW -> HUFFMAN
- BWT -> M2F -> LZW -> HUFFMAN
- BBWT -> M2F -> LZW -> HUFFMAN
- BWT -> M2F -> RLE -> LZW -> HUFFMAN
- BBWT -> M2F -> RLE -> LZW -> HUFFMAN
- RLE -> BWT -> M2F -> RLE -> LZW -> HUFFMAN
- RLE -> BBWT -> M2F -> RLE -> LZW -> HUFFMAN

4.3.1 Main

In questo paragrafo verrà illustrata la struttura del file `main.cpp`.

Costanti e metodi
In questa sezione del main vengono dichiarate le costanti e un metodo di supporto.
Le prime due costanti `COMPRESS` e `DECOMPRESS` vengono utilizzate per distinguere il tipo di operazione da fare. Successivamente vengono definite le costanti riguardanti le pipeline da eseguire e le ultime due indicano il numero di pipeline e la grandezza del chunk.

```
20  // define MODE
21  #define COMPRESS 0
22  #define DECOMPRESS 1
23
24  // define PIPELINE
25  #define HUFFMAN                             0
26  #define ARITHMETIC_CODE                     1
27  #define BWT_M2F_HUFFMAN                     2
28  #define BWT_M2F_ARITHMETIC_CODE             3
29  #define BBWT_M2F_HUFFMAN                    4
30  #define BBWT_M2F_ARITHMETIC_CODE            5
31  #define BWT_M2F_RLE_HUFFMAN                 6
32  #define BWT_M2F_RLE_ARITHMETIC_CODE         7
33  #define BBWT_M2F_RLE_HUFFMAN                8
34  #define BBWT_M2F_RLE_ARITHMETIC_CODE        9
35  #define RLE_BWT_M2F_RLE_HUFFMAN             10
36  #define RLE_BWT_M2F_RLE_ARITHMETIC_CODE     11
37  #define RLE_BBWT_M2F_RLE_HUFFMAN            12
38  #define RLE_BBWT_M2F_RLE_ARITHMETIC_CODE    13
39  #define LZW_HUFFMAN                         14
40  #define BWT_M2F_LZW_HUFFMAN                 15
41  #define BBWT_M2F_LZW_HUFFMAN                16
42  #define BWT_M2F_RLE_LZW_HUFFMAN             17
```

```
#define BBWT_M2F_RLE_LZW_HUFFMAN            18
#define RLE_BWT_M2F_RLE_LZW_HUFFMAN         19
#define RLE_BBWT_M2F_RLE_LZW_HUFFMAN        20

#define N_PIPELINES 21

#define CHUNK_SIZE 50000
```

Infine viene implementato il metodo di supporto `bool compareFiles(const std::string& p1, const std::string& p2)` utile per effettuare il confronto tra due file. Di seguito l'implementazione:

```
bool compareFiles(const std::string& p1, const std::string& p2) {
    std::ifstream f1(p1, std::ifstream::binary|std::ifstream::ate);
    std::ifstream f2(p2, std::ifstream::binary|std::ifstream::ate);

    if (f1.fail() || f2.fail()) {
        return false; // file problem
    }

    if (f1.tellg() != f2.tellg()) {
        return false; // size mismatch
    }

    // seek back to beginning and use std::equal to compare contents
    f1.seekg(0, std::ifstream::beg);
    f2.seekg(0, std::ifstream::beg);
    return std::equal(std::istreambuf_iterator<char>(f1.rdbuf()),
                      std::istreambuf_iterator<char>(),
                      std::istreambuf_iterator<char>(f2.rdbuf()));
}
```

Metodo main

Nella funzione `main` viene prima effettuato un controllo sui parametri dati in input e successivamente in base ai valori ricevuti viene scelta la pipeline da eseguire tramite uno `switch`.
Di seguito i parametri che bisogna dare in input al main:

- `argv[1]`: indica l'operazione da fare, comprimere, `-c`, o decomprimere `-d`.
- `argv[2]`: indica l'indice della pipeline che si vuole eseguire. Indice compreso tra 0 e `N_PIPELINES` (escluso) che di default è impostato a 14.
- `argv[3]`: indica il percorso del file da dare in input il quale viene aperto e letto
- `argv[4]`: indica il percorso del file sul quale scrivere l'output

Di seguito l'implementazione della prima parte del main:

```
68  int main(int argc, char* argv[]) {
69      int OPERATION;
70      int PIPELINE;
71
72      if (argc == 5) {
73          // get mode of use
74          if (strcmp(argv[1], "-c") == 0)
75              OPERATION = COMPRESS;
76          else if (strcmp(argv[1], "-d") == 0)
77              OPERATION = DECOMPRESS;
78          else {
79              cout << "Using: main <-c/-d> <pipeline> <input_of_file> <output_file>"
80              << endl;
81              return -1;
82          }
83
84          // define array of all pipelins
85          int PIPELINES[N_PIPELINES] = {
86              HUFFMAN,
87              ARITHMETIC_CODE,
88              BWT_M2F_HUFFMAN,
89              BWT_M2F_ARITHMETIC_CODE,
90              BBWT_M2F_HUFFMAN,
91              BBWT_M2F_ARITHMETIC_CODE,
92              BWT_M2F_RLE_HUFFMAN,
93              BWT_M2F_RLE_ARITHMETIC_CODE,
94              BBWT_M2F_RLE_HUFFMAN,
95              BBWT_M2F_RLE_ARITHMETIC_CODE,
96              RLE_BWT_M2F_RLE_HUFFMAN,
97              RLE_BWT_M2F_RLE_ARITHMETIC_CODE,
98              RLE_BBWT_M2F_RLE_HUFFMAN,
99              RLE_BBWT_M2F_RLE_ARITHMETIC_CODE,
100             LZW_HUFFMAN,
101             BWT_M2F_LZW_HUFFMAN,
102             BBWT_M2F_LZW_HUFFMAN,
103             BWT_M2F_RLE_LZW_HUFFMAN,
104             BBWT_M2F_RLE_LZW_HUFFMAN,
105             RLE_BWT_M2F_RLE_LZW_HUFFMAN,
106             RLE_BBWT_M2F_RLE_LZW_HUFFMAN,
107         };
108
109         int indexPipeline;
110         sscanf(argv[2], "%d", &indexPipeline);
111         if (indexPipeline < 0 || indexPipeline > N_PIPELINES) {
112             cout << "The specified pipeline does not exist" << endl;
113                     return -1;
114         } else
115             PIPELINE = PIPELINES[indexPipeline];
```

```
116
117         // open input file
118         ifstream file(argv[3]);
119         if (!file) {
120             cout << argv[3] << ": No such file or directory" << endl;
121                     return -1;
122         }
123
124         // open output file
125         ofstream out_file;
126         out_file.open(argv[4], ios::binary | ios::out);
127         if (!out_file) {
128             cout << argv[4] << ": No such file or directory" << endl;
129                     return -1;
130         }
131
132         // read all file
133         string data((istreambuf_iterator<char>(file)),
134             istreambuf_iterator<char>());
135         file.close();
136
137         // init string compressed/decompressed
138         string output;
139
```

Successivamente viene eseguito un primo switch per distinguere le operazioni di compressione e decompressione. In ognuno dei due casi viene eseguito un ulteriore switch per selezionare la pipeline da eseguire.

```
1  switch(OPERATION) {
2
3      case COMPRESS: {
4
5          switch(PIPELINE) {
6
7              #Pipeline for compression...
8
9          }
10     }
11     break;
12
13     case DECOMPRESS: {
14
15         switch(PIPELINE) {
16
17             #Pipeline for decompression...
18
```

```
19            }
20        }
21    }
```

Il primo switch viene eseguito sulla variabile `OPERATION` che indica il tipo di operazione scelto, ovvero compressione o decompressione.

- `COMPRESS`: di seguito le implementazioni delle varie pipeline usate per la compressione:
  - `HUFFMAN`: In questa pipeline non vengono effettuate operazioni di pre-processing, viene eseguita direttamente la codifica di Huffman. Di seguito l'implementazione:

```
141  case HUFFMAN: {
142      cout << "Pipeline: Huffman" << endl;
143      huffman huffman_encoder(argv[3], argv[4]);
144      huffman_encoder.compress();
145      break;
146  }
```

  - `ARITHMETIC_CODE`: In questa pipeline non vengono effettuate operazioni di pre-processing, viene eseguita direttamente la codifica aritmetica. Di seguito l'implementazione:

```
148  case ARITHMETIC_CODE: {
149      cout << "Pipeline: Arithmetic Coding" << endl;
150      Encode arithmetic_encoder;
151      arithmetic_encoder.encode(argv[3], argv[4]);
152      break;
153  }
```

  - `BWT_M2F_HUFFMAN`: In questa pipeline come operazioni di pre-processing vengono eseguite la BWT e la move-to-front per poi effettuare la codifica di Huffman. Di seguito l'implementazione:

```
155  case BWT_M2F_HUFFMAN: {
156      cout << "Pipeline: BWT -> M2F -> Huffman" << endl;
157      unsigned long size = data.size();
158      unsigned long nPerfectChunk = size / CHUNK_SIZE;
159      for(unsigned long i = 0; i < nPerfectChunk; i++) {
160          string bwt_encoded = bwtEncode(data.substr(i * CHUNK_SIZE,
161              CHUNK_SIZE));
162          output += bwt_encoded;
163      }
164      if ((nPerfectChunk * CHUNK_SIZE) < size) {
165          string bwt_encoded = bwtEncode(data.substr(nPerfectChunk *
166              CHUNK_SIZE));
167          output += bwt_encoded;
168      }
169      MTF m2f;
170      output = m2f.encode(output);
```

```
        huffman huffman_encoder(argv[3], argv[4]);
        huffman_encoder.compressData(output);
        break;
    }
```

– `BWT_M2F_ARITHMETIC_CODE`: In questa pipeline come operazioni di pre-processing vengono eseguite la BWT e la move-to-front per poi effettuare la codifica aritmetica. Di seguito l'implementazione:

```
    case BWT_M2F_ARITHMETIC_CODE: {
        cout << "Pipeline: BWT -> M2F -> Arithmetic Coding" << endl;
        unsigned long size = data.size();
        unsigned long nPerfectChunk = size / CHUNK_SIZE;
        for(unsigned long i = 0; i < nPerfectChunk; i++) {
            string bwt_encoded = bwtEncode(data.substr(i * CHUNK_SIZE,
                CHUNK_SIZE));
            output += bwt_encoded;
        }
        if ((nPerfectChunk * CHUNK_SIZE) < size) {
            string bwt_encoded = bwtEncode(data.substr(nPerfectChunk *
                CHUNK_SIZE));
            output += bwt_encoded;
        }
        MTF m2f;
        output = m2f.encode(output);
        Encode arithmetic_encoder;
        arithmetic_encoder.encodeData(output, argv[4]);
        break;
    }
```

– `BBWT_M2F_HUFFMAN`: In questa pipeline come operazioni di pre-processing vengono eseguite la BBWT e la move-to-front per poi effettuare la codifica di Huffman. Di seguito l'implementazione:

```
    case BBWT_M2F_HUFFMAN: {
        cout << "Pipeline: Bijective BWT -> M2F -> Huffman" << endl;
        unsigned long size = data.size();
        unsigned long nPerfectChunk = size / CHUNK_SIZE;
        for(unsigned long i = 0; i < nPerfectChunk; i++) {
            string bbwt_encoded = bbwtEncode(data.substr(i * CHUNK_SIZE,
                CHUNK_SIZE));
            output += bbwt_encoded;
        }
        if ((nPerfectChunk * CHUNK_SIZE) < size) {
            string bbwt_encoded = bbwtEncode(data.substr(nPerfectChunk *
                CHUNK_SIZE));
            output += bbwt_encoded;
        }
        MTF m2f;
        output = m2f.encode(output);
        huffman huffman_encoder(argv[3], argv[4]);
        huffman_encoder.compressData(output);
```

```
        break;
    }
```

– `BBWT_M2F_ARITHMETIC_CODE`: In questa pipeline come operazioni di pre-processing vengono eseguite la BBWT e la move-to-front per poi effettuare la codifica aritmetica. Di seguito l'implementazione:

```
    case BBWT_M2F_ARITHMETIC_CODE: {
        cout << "Pipeline: Bijective BWT -> M2F"
            "-> Arithmetic Coding" << endl;
        unsigned long size = data.size();
        unsigned long nPerfectChunk = size / CHUNK_SIZE;
        for(unsigned long i = 0; i < nPerfectChunk; i++) {
            string bbwt_encoded = bbwtEncode(data.substr(i * CHUNK_SIZE,
                CHUNK_SIZE));
            output += bbwt_encoded;
        }
        if ((nPerfectChunk * CHUNK_SIZE) < size) {
            string bbwt_encoded = bbwtEncode(data.substr(nPerfectChunk *
                CHUNK_SIZE));
            output += bbwt_encoded;
        }
        MTF m2f;
        output = m2f.encode(output);
        Encode arithmetic_encoder;
        arithmetic_encoder.encodeData(output, argv[4]);
        break;
    }
```

– `BWT_M2F_RLE_HUFFMAN`: In questa pipeline come operazioni di pre-processing vengono eseguite la BWT, la move-to-front e la run-length encoding per poi effettuare la codifica di Huffman. Di seguito l'implementazione:

```
    case BWT_M2F_RLE_HUFFMAN: {
        cout << "Pipeline: BWT -> M2F -> RLE -> Huffman" << endl;
        unsigned long size = data.size();
        unsigned long nPerfectChunk = size / CHUNK_SIZE;
        for(unsigned long i = 0; i < nPerfectChunk; i++) {
            string bwt_encoded = bwtEncode(data.substr(i * CHUNK_SIZE,
                CHUNK_SIZE));
            output += bwt_encoded;
        }
        if ((nPerfectChunk * CHUNK_SIZE) < size) {
            string bwt_encoded = bwtEncode(data.substr(nPerfectChunk *
                CHUNK_SIZE));
            output += bwt_encoded;
        }
        MTF m2f;
        output = m2f.encode(output);
        output = rleEncode(output);
```

```
248         huffman huffman_encoder(argv[3], argv[4]);
249         huffman_encoder.compressData(output);
250         break;
251     }
```

– `BWT_M2F_RLE_ARITHMETIC_CODE`: In questa pipeline come operazioni di pre-processing vengono eseguite la BWT, la move-to-front e la run-length encoding per poi effettuare la codifica aritmetica. Di seguito l'implementazione:

```
251     case BWT_M2F_RLE_ARITHMETIC_CODE: {
252         cout << "Pipeline: BWT -> M2F -> RLE -> Arithmetic Coding" << endl;
253         unsigned long size = data.size();
254         unsigned long nPerfectChunk = size / CHUNK_SIZE;
255         for(unsigned long i = 0; i < nPerfectChunk; i++) {
256             string bwt_encoded = bwtEncode(data.substr(i * CHUNK_SIZE,
257                 CHUNK_SIZE));
258             output += bwt_encoded;
259         }
260         if ((nPerfectChunk * CHUNK_SIZE) < size) {
261             string bwt_encoded = bwtEncode(data.substr(nPerfectChunk *
262                 CHUNK_SIZE));
263             output += bwt_encoded;
264         }
265         MTF m2f;
266         output = m2f.encode(output);
267         output = rleEncode(output);
268         Encode arithmetic_encoder;
269         arithmetic_encoder.encodeData(output, argv[4]);
270         break;
271     }
```

– `BBWT_M2F_RLE_HUFFMAN`: In questa pipeline come operazioni di pre-processing vengono eseguite la BBWT, la move-to-front e la run-length encoding per poi effettuare la codifica di Huffman. Di seguito l'implementazione:

```
271     case BBWT_M2F_RLE_HUFFMAN: {
272         cout << "Pipeline: Bijective BWT -> M2F -> RLE -> Huffman" << endl;
273         unsigned long size = data.size();
274         unsigned long nPerfectChunk = size / CHUNK_SIZE;
275         for(unsigned long i = 0; i < nPerfectChunk; i++) {
276             string bbwt_encoded = bbwtEncode(data.substr(i * CHUNK_SIZE,
277                 CHUNK_SIZE));
278             output += bbwt_encoded;
279         }
280         if ((nPerfectChunk * CHUNK_SIZE) < size) {
281             string bbwt_encoded = bbwtEncode(data.substr(nPerfectChunk *
282                 CHUNK_SIZE));
283             output += bbwt_encoded;
284         }
```

```
285         MTF m2f;
286         output = m2f.encode(output);
287         output = rleEncode(output);
288         huffman huffman_encoder(argv[3], argv[4]);
289         huffman_encoder.compressData(output);
290         break;
291     }
```

– `BBWT_M2F_RLE_ARITHMETIC_CODE`: In questa pipeline come operazioni di pre-processing vengono eseguite la BBWT, la move-to-front e la run-length encoding per poi effettuare la codifica aritmetica. Di seguito l'implementazione:

```
291     case BBWT_M2F_RLE_ARITHMETIC_CODE: {
292         cout << "Pipeline: Bijective BWT -> M2F -> RLE"
293             "-> Arithmetic Coding" << endl;
294         unsigned long size = data.size();
295         unsigned long nPerfectChunk = size / CHUNK_SIZE;
296         for(unsigned long i = 0; i < nPerfectChunk; i++) {
297             string bbwt_encoded = bbwtEncode(data.substr(i * CHUNK_SIZE,
298                 CHUNK_SIZE));
299             output += bbwt_encoded;
300         }
301         if ((nPerfectChunk * CHUNK_SIZE) < size) {
302             string bbwt_encoded = bbwtEncode(data.substr(nPerfectChunk *
303                 CHUNK_SIZE));
304             output += bbwt_encoded;
305         }
306         MTF m2f;
307         output = m2f.encode(output);
308         output = rleEncode(output);
309         Encode arithmetic_encoder;
310         arithmetic_encoder.encodeData(output, argv[4]);
311         break;
312     }
```

– `RLE_BWT_M2F_RLE_HUFFMAN`: In questa pipeline come operazioni di pre-processing vengono eseguite la run-length-encoding, la BWT, la move-to-front e di nuovo la run-length encoding per poi effettuare la codifica di Huffman. Di seguito l'implementazione:

```
311     case RLE_BWT_M2F_RLE_HUFFMAN: {
312         cout << "Pipeline: RLE -> BWT -> M2F -> RLE -> Huffman" << endl;
313         data = RunLengthEncoding(data);
314         unsigned long size = data.size();
315         unsigned long nPerfectChunk = size / CHUNK_SIZE;
316         for(unsigned long i = 0; i < nPerfectChunk; i++) {
317             string bwt_encoded = bwtEncode(data.substr(i * CHUNK_SIZE,
318                 CHUNK_SIZE));
319             output += bwt_encoded;
320         }
```

```
        if ((nPerfectChunk * CHUNK_SIZE) < size) {
            string bwt_encoded = bwtEncode(data.substr(nPerfectChunk *
                CHUNK_SIZE));
            output += bwt_encoded;
        }
        MTF m2f;
        output = m2f.encode(output);
        output = rleEncode(output);
        huffman huffman_encoder(argv[3], argv[4]);
        huffman_encoder.compressData(output);
        break;
    }
```

– `RLE_BWT_M2F_RLE_ARITHMETIC_CODE`: In questa pipeline come operazioni di pre-processing vengono eseguite la run-length-encoding, la BWT, la move-to-front e di nuovo la run-length encoding per poi effettuare la codifica aritmetica. Di seguito l'implementazione:

```
    case RLE_BWT_M2F_RLE_ARITHMETIC_CODE: {
        cout << "Pipeline: RLE -> BWT -> M2F -> RLE"
            "-> Arithmetic Coding" << endl;
        data = RunLengthEncoding(data);
        unsigned long size = data.size();
        unsigned long nPerfectChunk = size / CHUNK_SIZE;
        for(unsigned long i = 0; i < nPerfectChunk; i++) {
            string bwt_encoded = bwtEncode(data.substr(i * CHUNK_SIZE,
                CHUNK_SIZE));
            output += bwt_encoded;
        }
        if ((nPerfectChunk * CHUNK_SIZE) < size) {
            string bwt_encoded = bwtEncode(data.substr(nPerfectChunk *
                CHUNK_SIZE));
            output += bwt_encoded;
        }
        MTF m2f;
        output = m2f.encode(output);
        output = rleEncode(output);
        Encode arithmetic_encoder;
        arithmetic_encoder.encodeData(output, argv[4]);
        break;
    }
```

– `RLE_BBWT_M2F_RLE_HUFFMAN`: In questa pipeline come operazioni di pre-processing vengono eseguite la run-length-encoding, la BBWT, la move-to-front e di nuovo la run-length encoding per poi effettuare la codifica di Huffman. Di seguito l'implementazione:

```
    case RLE_BBWT_M2F_RLE_HUFFMAN: {
        cout << "Pipeline: RLE -> Bijective BWT -> M2F"
            "-> RLE -> Huffman" << endl;
        data = RunLengthEncoding(data);
```

```
357         unsigned long size = data.size();
358         unsigned long nPerfectChunk = size / CHUNK_SIZE;
359         for(unsigned long i = 0; i < nPerfectChunk; i++) {
360             string bbwt_encoded = bbwtEncode(data.substr(i * CHUNK_SIZE,
361                 CHUNK_SIZE));
362             output += bbwt_encoded;
363         }
364         if ((nPerfectChunk * CHUNK_SIZE) < size) {
365             string bbwt_encoded = bbwtEncode(data.substr(nPerfectChunk *
366                 CHUNK_SIZE));
367             output += bbwt_encoded;
368         }
369         MTF m2f;
370         output = m2f.encode(output);
371         output = rleEncode(output);
372         huffman huffman_encoder(argv[3], argv[4]);
373         huffman_encoder.compressData(output);
374         break;
375     }
```

– `RLE_BBWT_M2F_RLE_ARITHMETIC_CODE`: In questa pipeline come operazioni di pre-processing vengono eseguite la run-length-encoding, la BBWT, la move-to-front e di nuovo la run-length encoding per poi effettuare la codifica aritmetica. Di seguito l'implementazione:

```
374     case RLE_BBWT_M2F_RLE_ARITHMETIC_CODE: {
375         cout << "Pipeline: RLE -> Bijective BWT -> M2F"
376             "-> RLE -> Arithmetic Coding" << endl;
377         data = RunLengthEncoding(data);
378         unsigned long size = data.size();
379         unsigned long nPerfectChunk = size / CHUNK_SIZE;
380         for(unsigned long i = 0; i < nPerfectChunk; i++) {
381             string bbwt_encoded = bbwtEncode(data.substr(i * CHUNK_SIZE,
382                 CHUNK_SIZE));
383             output += bbwt_encoded;
384         }
385         if ((nPerfectChunk * CHUNK_SIZE) < size) {
386             string bbwt_encoded = bbwtEncode(data.substr(nPerfectChunk *
387                 CHUNK_SIZE));
388             output += bbwt_encoded;
389         }
390         MTF m2f;
391         output = m2f.encode(output);
392         output = rleEncode(output);
393         Encode arithmetic_encoder;
394         arithmetic_encoder.encodeData(output, argv[4]);
395         break;
396     }
```

- `DECOMPRESS`: di seguito le implementazioni delle varie pipeline usate per la decompressione:

- `HUFFMAN`: In questa pipeline non vengono effettuate operazioni di pre-processing, viene eseguita direttamente la decodifica di Huffman. Di seguito l'implementazione:

```
403  case HUFFMAN: {
404      cout << "Pipeline: Huffman" << endl;
405      huffman huffman_decoder(argv[3], argv[4]);
406      huffman_decoder.decompress();
407      break;
408  }
```

- `ARITHMETIC_CODE`: In questa pipeline non vengono effettuate operazioni di pre-processing, viene eseguita direttamente la decodifica aritmetica. Di seguito l'implementazione:

```
410  case ARITHMETIC_CODE: {
411      cout << "Pipeline: Arithmetic Coding" << endl;
412      Decode arithmetic_decoder;
413      arithmetic_decoder.decode(argv[3], argv[4]);
414      break;
415  }
```

- `BWT_M2F_HUFFMAN`: In questa pipeline come operazioni di pre-processing vengono eseguite la BWT e la move-to-front per poi effettuare la decodifica di Huffman. Di seguito l'implementazione:

```
417  case BWT_M2F_HUFFMAN: {
418      cout << "Pipeline: BWT -> M2F -> Huffman" << endl;
419      huffman huffman_decoder(argv[3], argv[4]);
420      data = huffman_decoder.decompressData();
421      MTF m2f;
422      string m2f_decoded = m2f.decode(data);
423      unsigned long size = m2f_decoded.size();
424      unsigned long nPerfectChunk = size / (CHUNK_SIZE + 1);
425      for(unsigned long i = 0; i < nPerfectChunk; i++) {
426          string bwt_decoded =
427          bwtDecode(m2f_decoded.substr(i * (CHUNK_SIZE + 1), (CHUNK_SIZE + 1)));
428          output += bwt_decoded;
429      }
430      if ((nPerfectChunk * (CHUNK_SIZE + 1)) < size) {
431          string bwt_decoded =
432              bwtDecode(m2f_decoded.substr(nPerfectChunk * (CHUNK_SIZE + 1)));
433          output += bwt_decoded;
434      }
435      out_file << output;
436      break;
437  }
```

- `BWT_M2F_ARITHMETIC_CODE`: In questa pipeline come operazioni di pre-processing vengono eseguite la BWT e la move-to-front per poi effettuare la decodifica aritmetica. Di seguito l'implementazione:

```
437  case BWT_M2F_ARITHMETIC_CODE: {
438      cout << "Pipeline: BWT -> M2F -> Arithmetic Coding" << endl;
```

```
439         Decode arithmetic_decoder;
440         data = arithmetic_decoder.decodeData(argv[3]);
441         MTF m2f;
442         string m2f_decoded = m2f.decode(data);
443         unsigned long size = m2f_decoded.size();
444         unsigned long nPerfectChunk = size / (CHUNK_SIZE + 1);
445         for(unsigned long i = 0; i < nPerfectChunk; i++) {
446             string bwt_decoded =
447             bwtDecode(m2f_decoded.substr(i * (CHUNK_SIZE + 1), (CHUNK_SIZE + 1)));
448             output += bwt_decoded;
449         }
450         if ((nPerfectChunk * (CHUNK_SIZE + 1)) < size) {
451             string bwt_decoded =
452                 bwtDecode(m2f_decoded.substr(nPerfectChunk * (CHUNK_SIZE + 1)));
453             output += bwt_decoded;
454         }
455         out_file << output;
456         break;
457     }
```

– `BBWT_M2F_HUFFMAN`: In questa pipeline come operazioni di pre-processing vengono eseguite la BBWT e la move-to-front per poi effettuare la decodifica di Huffman. Di seguito l'implementazione:

```
457     case BBWT_M2F_HUFFMAN: {
458         cout << "Pipeline: Bijective BWT -> M2F -> Huffman" << endl;
459         huffman huffman_decoder(argv[3], argv[4]);
460         data = huffman_decoder.decompressData();
461         MTF m2f;
462         string m2f_decoded = m2f.decode(data);
463         unsigned long size = m2f_decoded.size();
464         unsigned long nPerfectChunk = size / (CHUNK_SIZE);
465         for(unsigned long i = 0; i < nPerfectChunk; i++) {
466             string bbwt_decoded =
467             bbwtDecode(m2f_decoded.substr(i * CHUNK_SIZE, CHUNK_SIZE));
468             output += bbwt_decoded;
469         }
470         if ((nPerfectChunk * CHUNK_SIZE) < size) {
471             string bbwt_decoded =
472                 bbwtDecode(m2f_decoded.substr(nPerfectChunk * CHUNK_SIZE));
473             output += bbwt_decoded;
474         }
475         out_file << output;
476         break;
477     }
```

– `BBWT_M2F_ARITHMETIC_CODE`: In questa pipeline come operazioni di pre-processing vengono eseguite la BBWT e la move-to-front per poi effettuare la decodifica aritmetica. Di seguito l'implementazione:

```
477     case BBWT_M2F_ARITHMETIC_CODE: {
478         cout << "Pipeline: Bijective BWT -> M2F -> Arithmetic Coding" << endl;
```

```
        Decode arithmetic_decoder;
        data = arithmetic_decoder.decodeData(argv[3]);
        MTF m2f;
        string m2f_decoded = m2f.decode(data);
        unsigned long size = m2f_decoded.size();
        unsigned long nPerfectChunk = size / (CHUNK_SIZE);
        for(unsigned long i = 0; i < nPerfectChunk; i++) {
            string bbwt_decoded =
            bbwtDecode(m2f_decoded.substr(i * CHUNK_SIZE, CHUNK_SIZE));
            output += bbwt_decoded;
        }
        if ((nPerfectChunk * CHUNK_SIZE) < size) {
            string bbwt_decoded =
                bbwtDecode(m2f_decoded.substr(nPerfectChunk * CHUNK_SIZE));
            output += bbwt_decoded;
        }
        out_file << output;
        break;
    }
```

– `BWT_M2F_RLE_HUFFMAN`: In questa pipeline come operazioni di pre-processing vengono eseguite la BWT, la move-to-front e la run-length encoding per poi effettuare la decodifica di Huffman. Di seguito l'implementazione:

```
    case BWT_M2F_RLE_HUFFMAN: {
        cout << "Pipeline: BWT -> M2F -> RLE -> Huffman" << endl;
        huffman huffman_decoder(argv[3], argv[4]);
        data = huffman_decoder.decompressData();
        data = rleDecode(data);
        MTF m2f;
        string m2f_decoded = m2f.decode(data);
        unsigned long size = m2f_decoded.size();
        unsigned long nPerfectChunk = size / (CHUNK_SIZE + 1);
        for(unsigned long i = 0; i < nPerfectChunk; i++) {
            string bwt_decoded =
            bwtDecode(m2f_decoded.substr(i * (CHUNK_SIZE + 1), (CHUNK_SIZE + 1)));
            output += bwt_decoded;
        }
        if ((nPerfectChunk * (CHUNK_SIZE + 1)) < size) {
            string bwt_decoded =
                bwtDecode(m2f_decoded.substr(nPerfectChunk * (CHUNK_SIZE + 1)));
            output += bwt_decoded;
        }
        out_file << output;
        break;
    }
```

– `BWT_M2F_RLE_ARITHMETIC_CODE`: In questa pipeline come operazioni di pre-processing vengono eseguite la BWT, la move-to-front e la run-length encoding per poi effettuare la decodifica ar-

itmetica. Di seguito l'implementazione:

```
518  case BWT_M2F_RLE_ARITHMETIC_CODE: {
519      cout << "Pipeline: BWT -> M2F -> RLE -> Arithmetic Coding" << endl;
520      Decode arithmetic_decoder;
521      data = arithmetic_decoder.decodeData(argv[3]);
522      data = rleDecode(data);
523      MTF m2f;
524      string m2f_decoded = m2f.decode(data);
525      unsigned long size = m2f_decoded.size();
526      unsigned long nPerfectChunk = size / (CHUNK_SIZE + 1);
527      for(unsigned long i = 0; i < nPerfectChunk; i++) {
528          string bwt_decoded =
529          bwtDecode(m2f_decoded.substr(i * (CHUNK_SIZE + 1), (CHUNK_SIZE + 1)));
530          output += bwt_decoded;
531      }
532      if ((nPerfectChunk * (CHUNK_SIZE + 1)) < size) {
533          string bwt_decoded =
534              bwtDecode(m2f_decoded.substr(nPerfectChunk * (CHUNK_SIZE + 1)));
535          output += bwt_decoded;
536      }
537      out_file << output;
538      break;
539  }
```

– `BBWT_M2F_RLE_HUFFMAN`: In questa pipeline come operazioni di pre-processing vengono eseguite la BBWT, la move-to-front e la run-length encoding per poi effettuare la decodifica di Huffman. Di seguito l'implementazione:

```
539  case BBWT_M2F_RLE_HUFFMAN: {
540      cout << "Pipeline: Bijective BWT -> M2F -> RLE -> Huffman" << endl;
541      huffman huffman_decoder(argv[3], argv[4]);
542      data = huffman_decoder.decompressData();
543      data = rleDecode(data);
544      MTF m2f;
545      string m2f_decoded = m2f.decode(data);
546      unsigned long size = m2f_decoded.size();
547      unsigned long nPerfectChunk = size / (CHUNK_SIZE);
548      for(unsigned long i = 0; i < nPerfectChunk; i++) {
549          string bbwt_decoded =
550          bbwtDecode(m2f_decoded.substr(i * CHUNK_SIZE, CHUNK_SIZE));
551          output += bbwt_decoded;
552      }
553      if ((nPerfectChunk * CHUNK_SIZE) < size) {
554          string bbwt_decoded =
555              bbwtDecode(m2f_decoded.substr(nPerfectChunk * CHUNK_SIZE));
556          output += bbwt_decoded;
557      }
558      out_file << output;
559      break;
```

```
560   }
```

– `BBWT_M2F_RLE_ARITHMETIC_CODE`: In questa pipeline come operazioni di pre-processing vengono eseguite la BBWT, la move-to-front e la run-length encoding per poi effettuare la decodifica aritmetica. Di seguito l'implementazione:

```
560   case BBWT_M2F_RLE_ARITHMETIC_CODE: {
561       cout << "Pipeline: Bijective BWT -> M2F -> RLE -> Arithmetic Coding"
562           << endl;
563       Decode arithmetic_decoder;
564       data = arithmetic_decoder.decodeData(argv[3]);
565       data = rleDecode(data);
566       MTF m2f;
567       string m2f_decoded = m2f.decode(data);
568       unsigned long size = m2f_decoded.size();
569       unsigned long nPerfectChunk = size / (CHUNK_SIZE);
570       for(unsigned long i = 0; i < nPerfectChunk; i++) {
571           string bbwt_decoded =
572           bbwtDecode(m2f_decoded.substr(i * CHUNK_SIZE, CHUNK_SIZE));
573           output += bbwt_decoded;
574       }
575       if ((nPerfectChunk * CHUNK_SIZE) < size) {
576           string bbwt_decoded =
577               bbwtDecode(m2f_decoded.substr(nPerfectChunk * CHUNK_SIZE));
578           output += bbwt_decoded;
579       }
580       out_file << output;
581       break;
582   }
```

– `RLE_BWT_M2F_RLE_HUFFMAN`: In questa pipeline come operazioni di pre-processing vengono eseguite la run-length-encoding, la BWT, la move-to-front e di nuovo la run-length encoding per poi effettuare la decodifica di Huffman. Di seguito l'implementazione:

```
581   case RLE_BWT_M2F_RLE_HUFFMAN: {
582       cout << "Pipeline: RLE -> BWT -> M2F -> RLE -> Huffman"
583           << endl;
584       huffman huffman_decoder(argv[3], argv[4]);
585       data = huffman_decoder.decompressData();
586       data = rleDecode(data);
587       MTF m2f;
588       string m2f_decoded = m2f.decode(data);
589       unsigned long size = m2f_decoded.size();
590       unsigned long nPerfectChunk = size / (CHUNK_SIZE + 1);
591       for(unsigned long i = 0; i < nPerfectChunk; i++) {
592           string bwt_decoded =
593           bwtDecode(m2f_decoded.substr(i * (CHUNK_SIZE + 1), (CHUNK_SIZE + 1)));
594           output += bwt_decoded;
595       }
```

```
596         if ((nPerfectChunk * (CHUNK_SIZE + 1)) < size) {
597             string bwt_decoded =
598                 bwtDecode(m2f_decoded.substr(nPerfectChunk * (CHUNK_SIZE + 1)));
599             output += bwt_decoded;
600         }
601         output = RunLengthDecoding(output);
602         out_file << output;
603         break;
604     }
```

– `RLE_BWT_M2F_RLE_ARITHMETIC_CODE`: In questa pipeline come operazioni di pre-processing vengono eseguite la run-length-encoding, la BWT, la move-to-front e di nuovo la run-length encoding per poi effettuare la decodifica aritmetica. Di seguito l'implementazione:

```
603     case RLE_BWT_M2F_RLE_ARITHMETIC_CODE: {
604         cout << "Pipeline: RLE -> BWT -> M2F -> RLE -> Arithmetic Coding"
605             << endl;
606         Decode arithmetic_decoder;
607         data = arithmetic_decoder.decodeData(argv[3]);
608         data = rleDecode(data);
609         MTF m2f;
610         string m2f_decoded = m2f.decode(data);
611         unsigned long size = m2f_decoded.size();
612         unsigned long nPerfectChunk = size / (CHUNK_SIZE + 1);
613         for(unsigned long i = 0; i < nPerfectChunk; i++) {
614             string bwt_decoded =
615             bwtDecode(m2f_decoded.substr(i * (CHUNK_SIZE + 1), (CHUNK_SIZE + 1)));
616             output += bwt_decoded;
617         }
618         if ((nPerfectChunk * (CHUNK_SIZE + 1)) < size) {
619             string bwt_decoded =
620                 bwtDecode(m2f_decoded.substr(nPerfectChunk * (CHUNK_SIZE + 1)));
621             output += bwt_decoded;
622         }
623         output = RunLengthDecoding(output);
624         out_file << output;
625         break;
626     }
```

– `RLE_BBWT_M2F_RLE_HUFFMAN`: In questa pipeline come operazioni di pre-processing vengono eseguite la run-length-encoding, la BBWT, la move-to-front e di nuovo la run-length encoding per poi effettuare la decodifica di Huffman. Di seguito l'implementazione:

```
625     case RLE_BBWT_M2F_RLE_HUFFMAN: {
626         cout << "Pipeline: RLE -> Bijective BWT -> M2F -> RLE -> Huffman"
627             << endl;
628         huffman huffman_decoder(argv[3], argv[4]);
629         data = huffman_decoder.decompressData();
630         data = rleDecode(data);
```

```
631         MTF m2f;
632         string m2f_decoded = m2f.decode(data);
633         unsigned long size = m2f_decoded.size();
634         unsigned long nPerfectChunk = size / (CHUNK_SIZE);
635         for(unsigned long i = 0; i < nPerfectChunk; i++) {
636             string bbwt_decoded =
637             bbwtDecode(m2f_decoded.substr(i * CHUNK_SIZE, CHUNK_SIZE));
638             output += bbwt_decoded;
639         }
640         if ((nPerfectChunk * CHUNK_SIZE) < size) {
641             string bbwt_decoded =
642                 bbwtDecode(m2f_decoded.substr(nPerfectChunk * CHUNK_SIZE));
643             output += bbwt_decoded;
644         }
645         output = RunLengthDecoding(output);
646         out_file << output;
647         break;
648     }
```

– `RLE_BBWT_M2F_RLE_ARITHMETIC_CODE`: In questa pipeline come operazioni di pre-processing vengono eseguite la run-length-encoding, la BBWT, la move-to-front e di nuovo la run-length encoding per poi effettuare la decodifica aritmetica. Di seguito l'implementazione:

```
647     case RLE_BBWT_M2F_RLE_ARITHMETIC_CODE: {
648         cout << "Pipeline: RLE -> Bijective BWT -> M2F -> RLE"
649             " -> Arithmetic Coding" << endl;
650         Decode arithmetic_decoder;
651         data = arithmetic_decoder.decodeData(argv[3]);
652         data = rleDecode(data);
653         MTF m2f;
654         string m2f_decoded = m2f.decode(data);
655         unsigned long size = m2f_decoded.size();
656         unsigned long nPerfectChunk = size / (CHUNK_SIZE);
657         for(unsigned long i = 0; i < nPerfectChunk; i++) {
658             string bbwt_decoded =
659             bbwtDecode(m2f_decoded.substr(i * CHUNK_SIZE, CHUNK_SIZE));
660             output += bbwt_decoded;
661         }
662         if ((nPerfectChunk * CHUNK_SIZE) < size) {
663             string bbwt_decoded =
664                 bbwtDecode(m2f_decoded.substr(nPerfectChunk * CHUNK_SIZE));
665             output += bbwt_decoded;
666         }
667         output = RunLengthDecoding(output);
668         out_file << output;
669         break;
670
671     }
```

4.3.2 test.py

Nel seguente codice viene invocato il main dichiarato precedentemente e vengono eseguite le pipeline descritte per ogni file utilizzato. Infine viene utilizzato *matplotlib* per la generazione dei grafici.

```
def generate_file_name(name: str,pipeline: str) -> str:
    return f"{name}_{pipeline}"
```

In questa funzione viene generato il nome da dare ad un file compresso.

```

def compression_ratio_from_file(uncompressed_path: str,
    compressed_path: str, decimal_digits=2) -> float:
    uncompressed_size = os.path.getsize(uncompressed_path)
    compressed_size = os.path.getsize(compressed_path)

    if compressed_size == 0:
        return 0

    return round(uncompressed_size / compressed_size,
    decimal_digits)
```

Questa funzione prende in input i path di due file, uno il file di partenza e l'altro il file compresso, e misura la ratio di compressione.

```

def plot_compression_ratio(file_name: str, algorithm: str,
    results: [dict]):
     os.makedirs('images', exist_ok=True)
    _file_name = f'{file_name}_{algorithm}_ratio.svg'
    file_path: str = os.path.join('images', _file_name)

    # create data
    ratios = [r["compression_ratio"] for r in results]
    pipelines = [
        "_",
        "BWT M2F",
        "BBWT M2F",
        "BWT M2F RLE",
        "BBWT M2F RLE",
        "RLE BWT M2F RLE",
        "RLE BBWT M2F RLE"]
    x_pos = np.arange(len(pipelines))

    # create bars
    plt.bar(x_pos, ratios, width=0.5)

    # rotation of the bar names
    plt.xticks(x_pos, pipelines, rotation=90)
    # custom the subplot layout
    if algorithm == 'arithmetic_code':
        plt.subplots_adjust(bottom=0.6, top=0.9)
    else:
        plt.subplots_adjust(bottom=0.4, top=0.9)
```

```
    # enable grid
    plt.grid(True)

    plt.title(f'{file_name}: {algorithm}')
    plt.xlabel('Pipeline')
    plt.ylabel('Compression ratio')

    # print value on the top of bar
    x_locs, x_labs = plt.xticks()
    for i, v in enumerate(ratios):
        plt.text(x_locs[i] - 0.2, v + 0.05, str(v))

    # set limit on y label
    plt.ylim(0, max(ratios) + 0.3)

    # savefig
    plt.savefig(file_path)
    plt.clf()
```

Questa funzione ci restituisce le immagini generate con la ratio di compressione

```

def plot_time_different_pipeline(file_name: str, algorithm: str,
     results: [dict]):
    os.makedirs('images', exist_ok=True)
    _file_name = f'{file_name}_{algorithm}_time.svg'
    file_path: str = os.path.join('images', _file_name)

    # create data
    compression_time = [r["compression_time"] for r in results]
    decompression_time = [r["decompression_time"] for r in
    results]
    pipelines = [
        "-",
        "BWT M2F",
        "BBWT M2F",
        "BWT M2F RLE",
        "BBWT M2F RLE",
        "RLE BWT M2F RLE",
        "RLE BBWT M2F RLE"]
    x_pos = np.arange(len(pipelines))

    # create grouped bars
    width = 0.35
    fig, ax = plt.subplots()
    rects1 = ax.bar(x_pos - width / 2, compression_time, width,
    label='Compression time')
    rects2 = ax.bar(x_pos + width / 2, decompression_time, width
    , label='Decompression time')

    # rotation of the bar names
    plt.xticks(x_pos, pipelines, rotation=90)
    ax.legend()
```

```
    # enable grid
    plt.grid(True)

    plt.title(f'{file_name}: {algorithm}')
    plt.xlabel('Pipeline')
    plt.ylabel('Time in ms')

    # print value on the top of bar
    ax.bar_label(rects1, padding=3)
    ax.bar_label(rects2, padding=3)

    # set limit on y label
    ax.margins(y=0.2)

    # savefig
    fig.tight_layout()
    plt.savefig(file_path)
    plt.clf()
```

Questa funzione ci resituisce le immagini generate con i tempi di compressione.

```

COMPRESSED_DIR_PATH: str = path.join(os.getcwd(), 'compressed')
DECOMPRESSED_DIR_PATH: str = path.join(os.getcwd(), '
    decompressed')
PATH_DIR_TEST_FILES = os.path.join(os.getcwd(), "files")

PIPELINES = [
    "HUFFMAN",
    "ARITHMETIC_CODE",
    "BWT_M2F_HUFFMAN",
    "BWT_M2F_ARITHMETIC_CODE",
    "BBWT_M2F_HUFFMAN",
    "BBWT_M2F_ARITHMETIC_CODE",
    "BWT_M2F_RLE_HUFFMAN",
    "BWT_M2F_RLE_ARITHMETIC_CODE",
    "BBWT_M2F_RLE_HUFFMAN",
    "BBWT_M2F_RLE_ARITHMETIC_CODE",
    "RLE_BWT_M2F_RLE_HUFFMAN",
    "RLE_BWT_M2F_RLE_ARITHMETIC_CODE",
    "RLE_BBWT_M2F_RLE_HUFFMAN",
    "RLE_BBWT_M2F_RLE_ARITHMETIC_CODE",
    "LZW_HUFFMAN",
    "BWT_M2F_LZW_HUFFMAN",
    "BBWT_M2F_LZW_HUFFMAN",
    "BWT_M2F_RLE_LZW_HUFFMAN",
    "BBWT_M2F_RLE_LZW_HUFFMAN",
    "RLE_BWT_M2F_RLE_LZW_HUFFMAN",
    "RLE_BBWT_M2F_RLE_LZW_HUFFMAN",
]


if __name__ == "__main__":
```

```
32
33      os.system('rm -r compressed decompressed images')
34      os.makedirs(COMPRESSED_DIR_PATH, exist_ok = True)
35      os.makedirs(DECOMPRESSED_DIR_PATH, exist_ok = True)
36
37      map_results_huffman = []
38      map_results_arith = []
39      map_results_lzw_huffman = []
40
41      for file_name in os.listdir(PATH_DIR_TEST_FILES):
42
43          for pipe in range(len(PIPELINES)):
44
45              if pipe < 14:
46                  continue
47
48              abspath_file = os.path.join(PATH_DIR_TEST_FILES,
    file_name)
49              abspath_compressed_file = os.path.join(
    COMPRESSED_DIR_PATH, generate_file_name(os.path.splitext(
    file_name)[0],PIPELINES[pipe]))
50              abspath_decompressed_file = os.path.join(
    DECOMPRESSED_DIR_PATH, generate_file_name(os.path.splitext(
    file_name)[0],PIPELINES[pipe]))
51
52              # compression test
53              start_time = time.time()
54              os.system(f'./main -c {pipe} {abspath_file} {
    abspath_compressed_file}')
55              compression_time = round((time.time() - start_time)
    * 1000)
56
57              # decompression test
58              start_time = time.time()
59              os.system(f'./main -d {pipe} {
    abspath_compressed_file} {abspath_decompressed_file}')
60              decompression_time = round((time.time() - start_time
    ) * 1000)
61
62              if (pipe > 13):
63                  result_lzw_huffman = dict()
64                  result_lzw_huffman["filename"] = file_name
65                  result_lzw_huffman["pipeline"] = PIPELINES[pipe]
66                  result_lzw_huffman["compression_ratio"] =
    compression_ratio_from_file(abspath_file,
    abspath_compressed_file)
67                  result_lzw_huffman["compression_time"] =
    compression_time
68                  result_lzw_huffman["decompression_time"] =
    decompression_time
69                  map_results_lzw_huffman.append(
    result_lzw_huffman)
70
71              else:
```

```
72
73                  if (PIPELINES[pipe].find("HUFFMAN") != -1):
74                      result_huffman = dict()
75                      result_huffman["filename"] = file_name
76                      result_huffman["pipeline"] = PIPELINES[pipe]
77                      result_huffman["compression_ratio"] =
    compression_ratio_from_file(abspath_file,
    abspath_compressed_file)
78                      result_huffman["compression_time"] =
    compression_time
79                      result_huffman["decompression_time"] =
    decompression_time
80                      map_results_huffman.append(result_huffman)
81
82                  else:
83                      result_arith = dict()
84                      result_arith["pipeline"] = PIPELINES[pipe]
85                      result_arith["compression_ratio"] =
    compression_ratio_from_file(abspath_file,
    abspath_compressed_file)
86                      result_arith["compression_time"] =
    compression_time
87                      result_arith["filename"] = file_name
88                      result_arith["decompression_time"] =
    decompression_time
89                      map_results_arith.append(result_arith)
90
91          #plot_compression_ratio(file_name, "huffman",
    map_results_huffman)
92          #plot_compression_ratio(file_name, "arithmetic code",
    map_results_arith)
93          plot_compression_ratio(file_name, "lzw huffman",
    map_results_lzw_huffman)
94
95          #plot_time_different_pipeline(file_name, "huffman",
    map_results_huffman)
96          #plot_time_different_pipeline(file_name, "arithmetic
    code", map_results_arith)
97          plot_time_different_pipeline(file_name, "lzw huffman",
    map_results_lzw_huffman)
98
99          map_results_arith.clear()
100         map_results_huffman.clear()
101         map_results_lzw_huffman.clear()
```

Infine, vengono eseguite tutte le pipeline per ogni file.

5 Risultati

Gli algoritmi implementati sono stati testati usando i dataset Canterbury Corpus e Calgary Corpus.
Gli esperimenti condotti sono stati effettuati utilizzando diverse pipeline con

differenti tecniche di pre-processing. Ciò mette in evidenza, le combinazioni di pre-processing più efficienti per tecniche di compressione dati. In particolare, si evidenziano i risultati dell'algoritmo proposto della variante biettiva della BWT e i risultati ottenuti combinando questa tecnica con altre tecniche di pre-processing simili a quelle utilizzate da bzip2 nell'utilizzo della BWT. In questi esperimenti abbiamo suddiviso la compressione dei file per chunk, ovvero comprimiamo un numero di bytes definito per ogni iterazione dell'algoritmo di compressione, in modo da non allocare in memoria ram tutti i bytes del file da comprimere. Sono stati riportati dei grafici che mostrano il tempo di compressione e decompressione necessario per ogni pipeline testata. Di seguito vengono riportati i risultati del ratio di compressione sotto forma tabellare, riportando in grassetto il risultato migliore ottenuto.

5.1 Huffman

Nella tabella 1 sono stati riportati i risultati del ratio compression per ogni file e pipeline. Nella prima riga, i valori da 0 a 6 rappresentano le diverse pipeline, rispettivamente:

- *No pre-processing*
- *BWT M2F*
- *BBWT M2F*
- *BWT M2F RLE*
- *BBWT M2F RLE*
- *RLE BWT M2F RLE*
- *RLE BBWT M2T RLE*

Table 1: Ratio di compressione utilizzando Huffman

File	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*BZIP2*
bib	1.5	**1.76**	**1.76**	**1.76**	**1.76**	1.49	1.49	4.05
book1	**1.75**	1.57	1.57	1.31	1.31	1.19	1.19	3.3
book2	1.65	**1.71**	**1.71**	1.55	1.55	1.37	1.37	3.88
geo	**1.33**	1.07	1.07	0.95	0.95	0.92	0.92	1.79
news	1.52	**1.57**	**1.57**	1.35	1.35	1.21	1.21	3.17
obj1	1.05	**1.26**	**1.26**	1.17	1.17	1.15	1.15	1.99
obj2	1.79	2.1	2.1	**3.23**	**3.23**	2.97	2.96	4.37
paper1	1.52	**1.71**	**1.71**	1.57	1.57	1.38	1.38	3.21
paper2	1.67	**1.68**	**1.68**	1.5	1.5	1.34	1.34	3.28
paper3	1.62	**1.66**	**1.66**	1.45	1.45	1.28	1.28	2.93
paper4	1.44	1.54	**1.55**	1.3	1.3	1.27	1.27	2.56
paper5	1.33	**1.52**	**1.52**	1.29	1.29	1.26	1.26	2.47
paper6	1.49	**1.72**	**1.72**	1.56	1.56	1.38	1.38	3.09
pic	4.7	2.26	2.26	**5.46**	**5.46**	5.3	5.3	10.31
progc	1.44	**1.72**	**1.72**	1.63	1.63	1.43	1.43	3.15
progl	1.61	1.97	1.97	**2.16**	**2.16**	2.03	2.03	4.59
progp	1.56	2.01	2.01	**2.3**	**2.3**	1.94	1.94	4.61
trans	1.4	1.99	1.99	**2.31**	**2.31**	1.91	1.91	5.29

Di seguito sono riportati i risultati rigurdante il tempo di compressione e decompressione per ogni file.

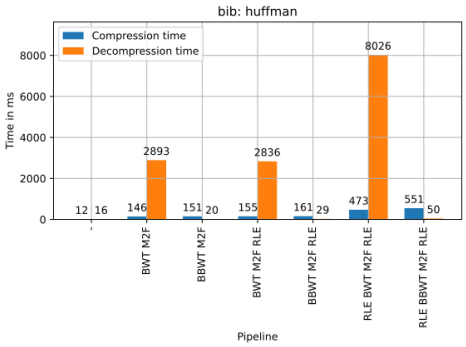


Figure 6: tempi di esecuzione

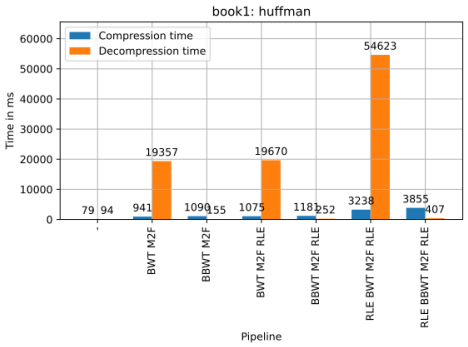


Figure 7: tempi di esecuzione

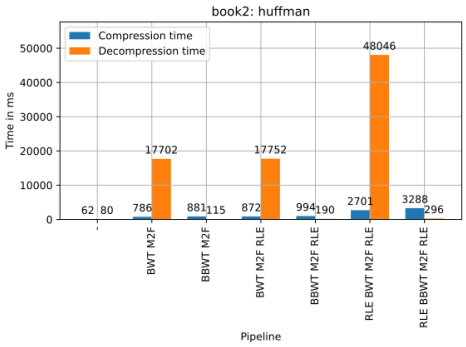


Figure 8: tempi di esecuzione

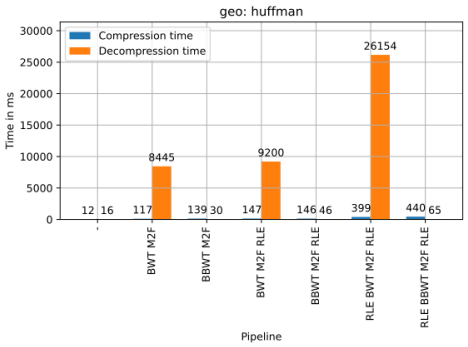


Figure 9: tempi di esecuzione

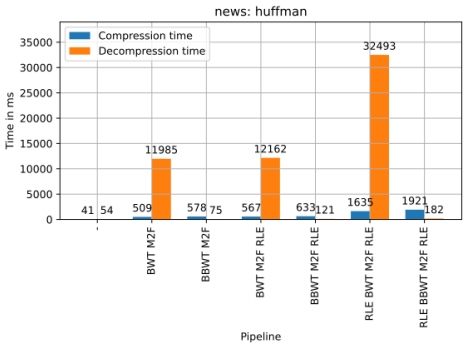


Figure 10: tempi di esecuzione

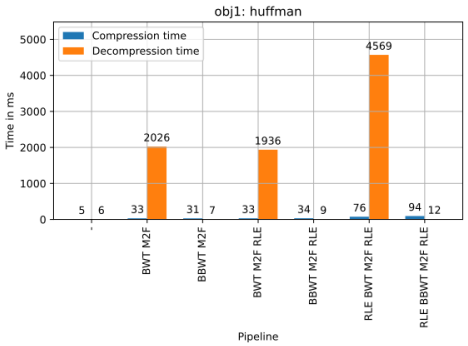


Figure 11: tempi di esecuzione

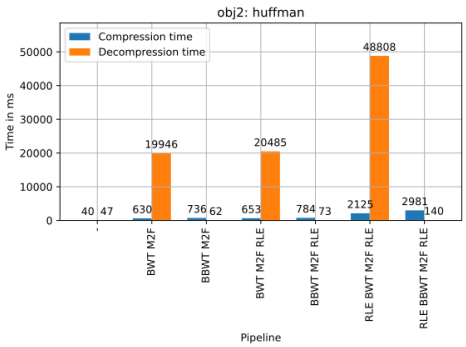


Figure 12: tempi di esecuzione

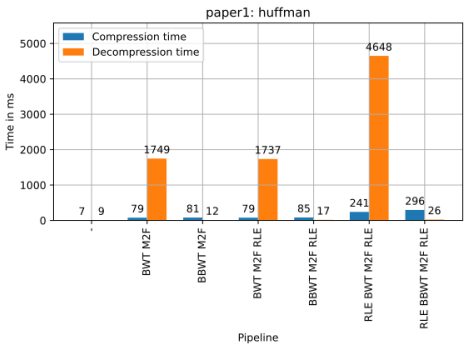


Figure 13: tempi di esecuzione

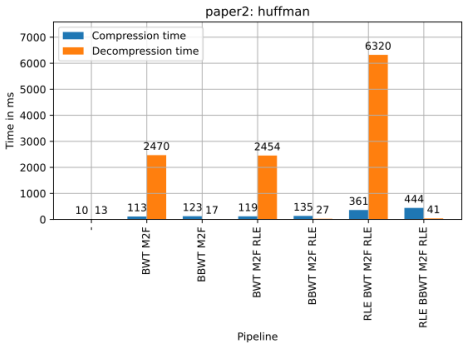


Figure 14: tempi di esecuzione

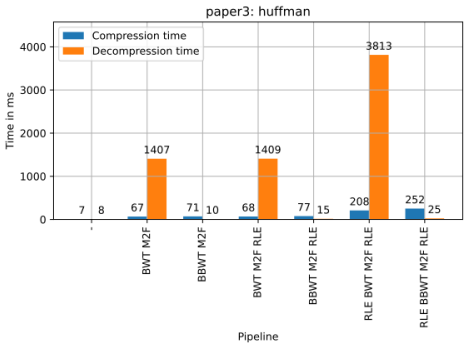


Figure 15: tempi di esecuzione

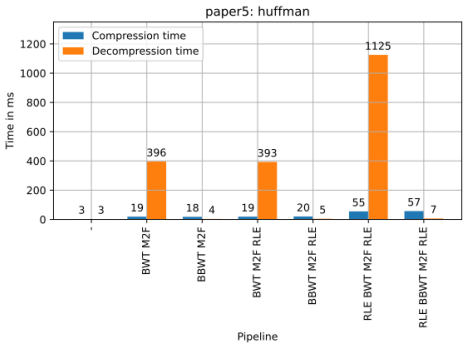


Figure 16: tempi di esecuzione

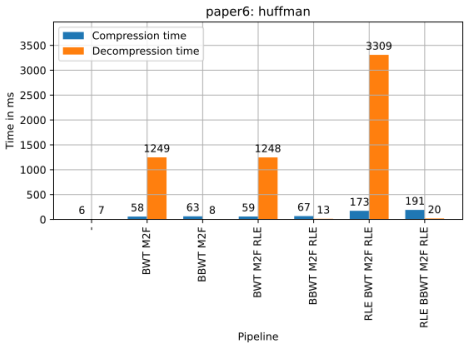


Figure 17: tempi di esecuzione

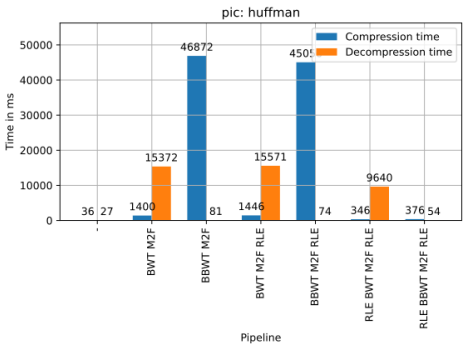


Figure 18: tempi di esecuzione

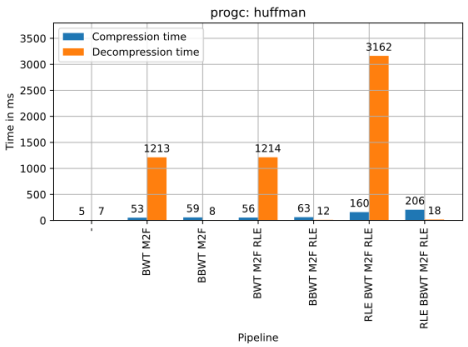


Figure 19: tempi di esecuzione

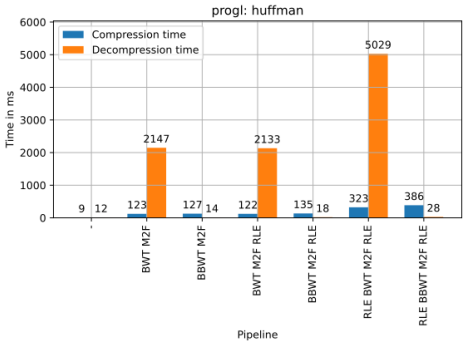


Figure 20: tempi di esecuzione

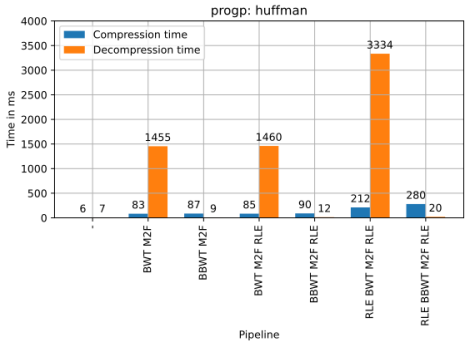


Figure 21: tempi di esecuzione

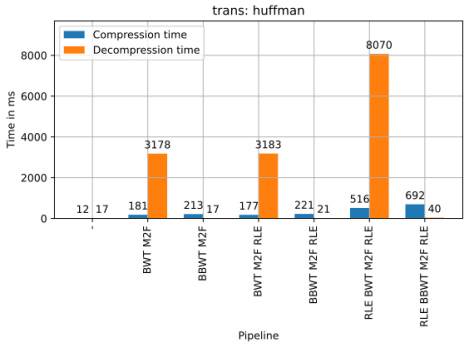


Figure 22: tempi di esecuzione

5.2 Arithmetic Coding

Come per la sezione precedente, nella tabella 2 per ogni file è stato riportato il valore relativo al ratio di compressione ottenuto con ogni tecnica di preprocessing testata. Inoltre, di seguito sono riportati i tempi di compressione e decompressione per ogni file.

Table 2: Ratio di compressione utilizzando Arithmetic Coding

File	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*BZIP2*
bib	1.53	**1.76**	**1.76**	1.75	1.75	1.47	1.47	4.05
book1	**1.76**	1.59	1.59	1.3	1.3	1.18	1.18	3.3
book2	1.67	**1.72**	**1.72**	1.54	1.54	1.36	1.36	3.88
geo	**1.41**	1.1	1.1	0.95	0.95	0.92	0.92	1.79
news	1.54	**1.57**	**1.57**	1.34	1.34	1.2	1.2	3.17
obj1	**1.34**	1.28	1.28	1.17	1.17	1.15	1.15	1.99
obj2	1.88	2.28	2.28	**3.21**	**3.21**	2.95	2.94	4.37
paper1	1.61	**1.72**	**1.72**	1.56	1.56	1.37	1.37	3.21
paper2	**1.73**	1.69	1.69	1.49	1.49	1.33	1.33	3.28
paper3	**1.7**	1.67	1.67	1.44	1.44	1.27	1.27	2.93
paper4	**1.66**	1.55	1.55	1.29	1.29	1.26	1.26	2.56
paper5	**1.58**	1.52	1.52	1.28	1.28	1.25	1.25	2.47
paper6	1.6	**1.73**	**1.73**	1.55	1.55	1.36	1.36	3.09
pic	**6.86**	2.66	2.66	5.42	5.42	5.26	5.26	10.31
progc	1.53	**1.72**	**1.72**	1.61	1.61	1.42	1.42	3.15
progl	1.68	2.01	2.01	**2.14**	**2.14**	2.02	2.02	4.59
progp	1.63	2.07	2.07	**2.28**	**2.28**	1.93	1.92	4.61
trans	1.46	2.05	2.05	**2.3**	**2.3**	1.9	1.9	5.29

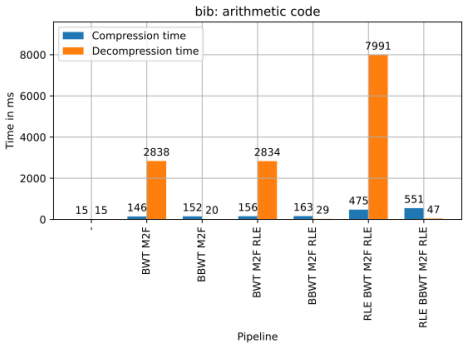


Figure 23: tempi di esecuzione

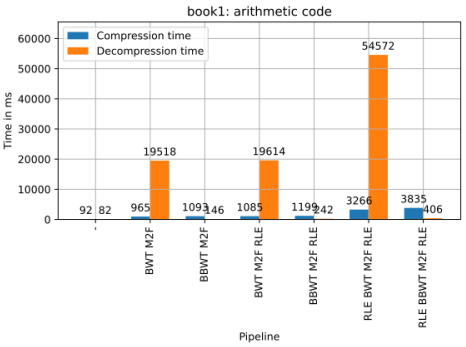


Figure 24: tempi di esecuzione

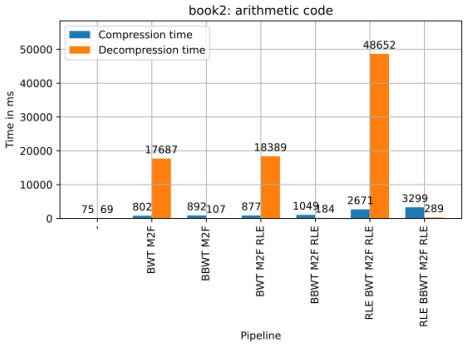


Figure 25: tempi di esecuzione

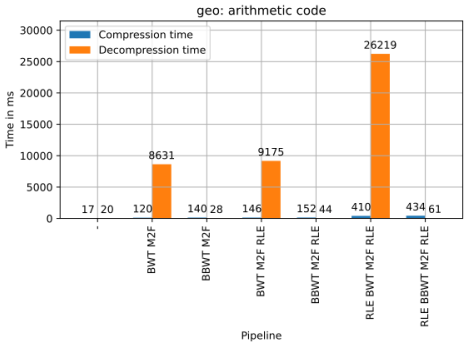


Figure 26: tempi di esecuzione

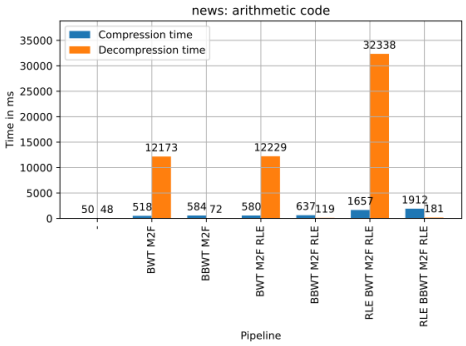


Figure 27: tempi di esecuzione

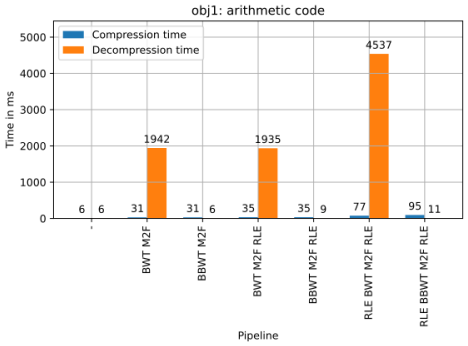


Figure 28: tempi di esecuzione

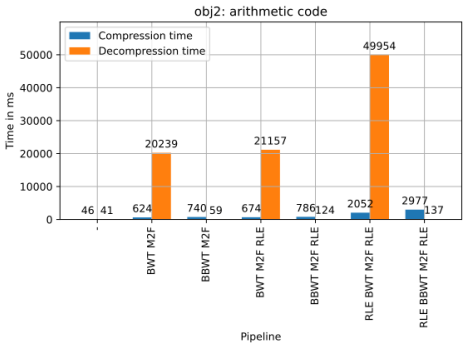


Figure 29: tempi di esecuzione

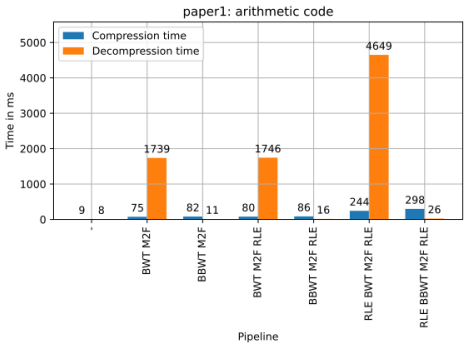


Figure 30: tempi di esecuzione

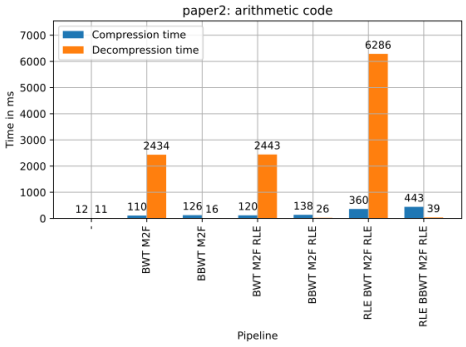


Figure 31: tempi di esecuzione

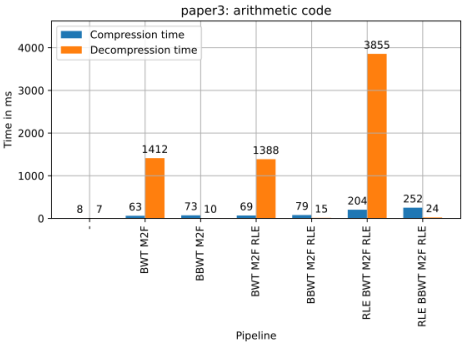


Figure 32: tempi di esecuzione

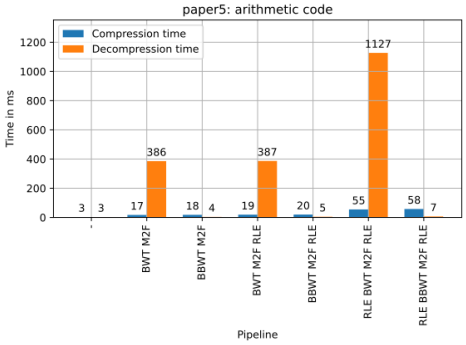


Figure 33: tempi di esecuzione

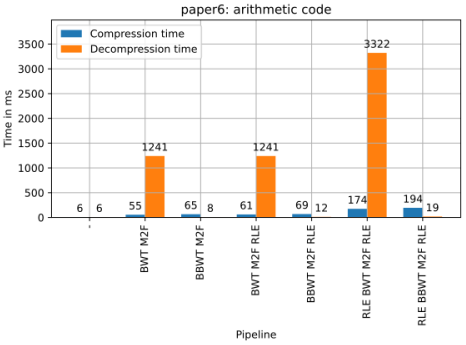


Figure 34: tempi di esecuzione

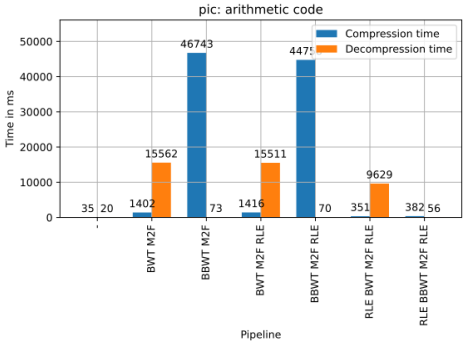


Figure 35: tempi di esecuzione

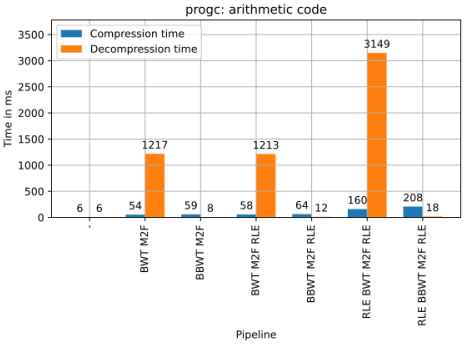


Figure 36: tempi di esecuzione

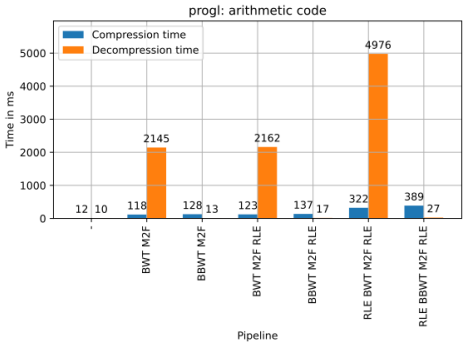


Figure 37: tempi di esecuzione

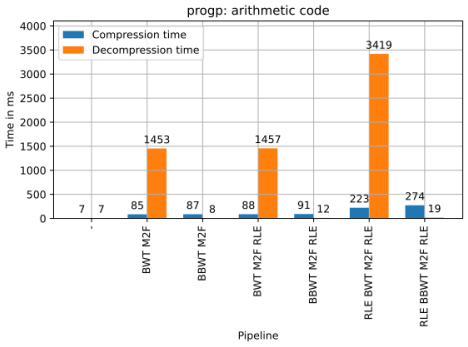


Figure 38: tempi di esecuzione

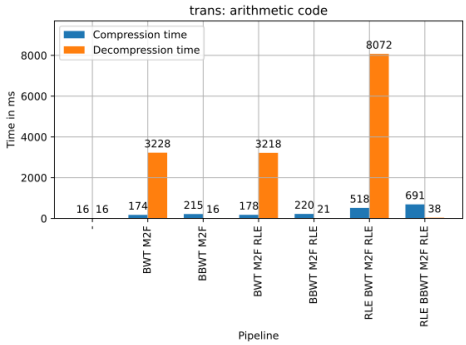


Figure 39: tempi di esecuzione

5.3 LZW - Huffman

Anche in questa sezione, nella tabella 3 è riportato il valore del ratio di compressione ottenuto con tecniche di pre-processing. Successivamente, sono stati riportati anche i tempi di compressione e decompressione per ogni file.

Ratio di compressione utilizzando LZW - Huffman

File	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*BZIP2*
bib	1.95	**2.17**	**2.17**	1.94	1.94	1.64	1.64	4.05
book1	**2.04**	1.83	1.83	1.64	1.64	1.47	1.47	3.3
book2	**2.1**	**2.1**	**2.1**	1.89	1.89	1.66	1.66	3.88
geo	**1.25**	1.09	1.09	1	1	0.96	0.96	1.79
news	**1.8**	1.77	1.77	1.59	1.59	1.42	1.42	3.17
obj1	**1.37**	1.24	1.24	1.13	1.13	1.11	1.11	1.99
obj2	1.58	**2.01**	**2.01**	1.83	1.84	1.73	1.73	4.37
paper1	1.71	**1.9**	**1.9**	1.69	1.69	1.48	1.48	3.21
paper2	1.84	**1.89**	**1.89**	1.68	1.68	1.49	1.49	3.28
paper3	1.71	**1.78**	**1.78**	1.57	1.57	1.39	1.39	2.93
paper4	**1.5**	1.46	1.46	1.28	1.28	1.25	1.25	2.56
paper5	**1.41**	**1.41**	**1.41**	1.24	1.24	1.21	1.21	2.47
paper6	1.64	**1.87**	**1.87**	1.66	1.66	1.46	1.46	3.09
pic	**6.91**	6.38	6.38	6.01	6.02	5.93	5.93	10.31
progc	1.68	**1.92**	**1.91**	1.71	1.71	1.5	1.5	3.15
progl	2.08	**2.63**	**2.63**	2.37	2.37	2.21	2.21	4.59
progp	2.05	**2.73**	**2.72**	2.47	2.47	2.1	2.09	4.61
trans	1.97	**2.77**	**2.77**	2.5	2.5	2.06	2.06	5.29

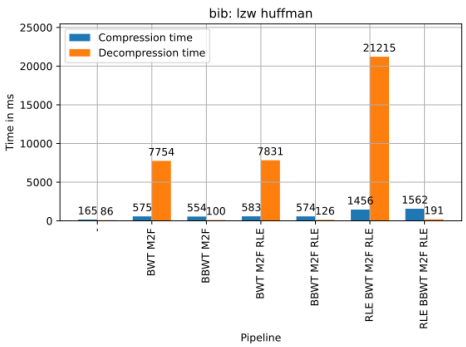


Figure 40: tempi di esecuzione

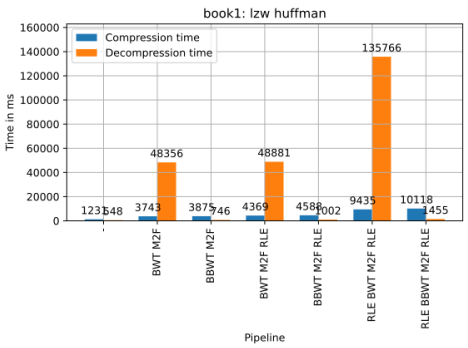


Figure 41: tempi di esecuzione

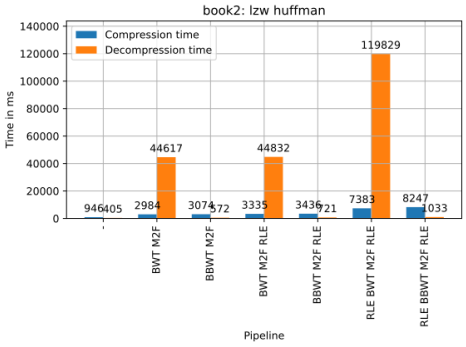


Figure 42: tempi di esecuzione

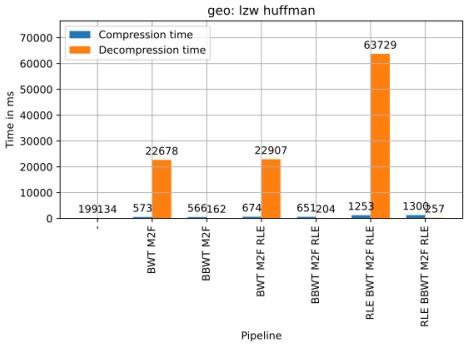


Figure 43: tempi di esecuzione

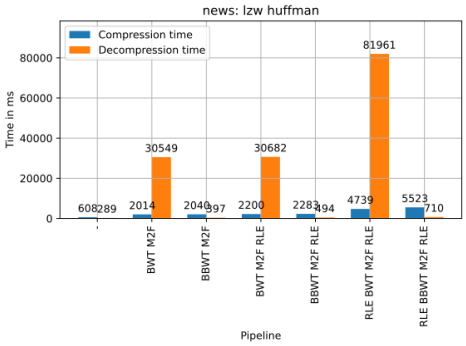


Figure 44: tempi di esecuzione

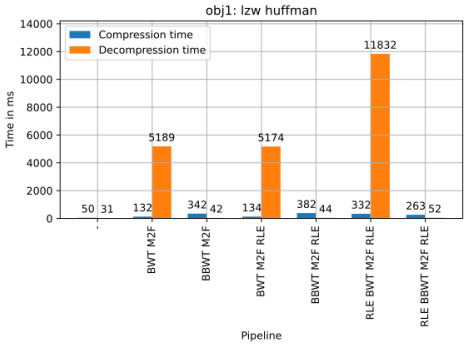


Figure 45: tempi di esecuzione

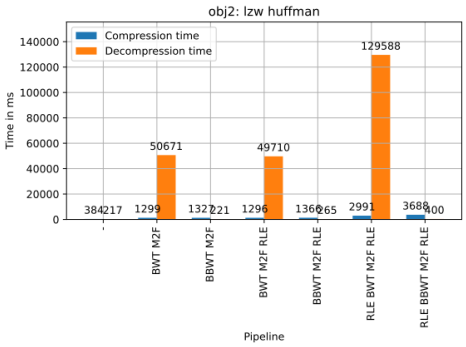


Figure 46: tempi di esecuzione

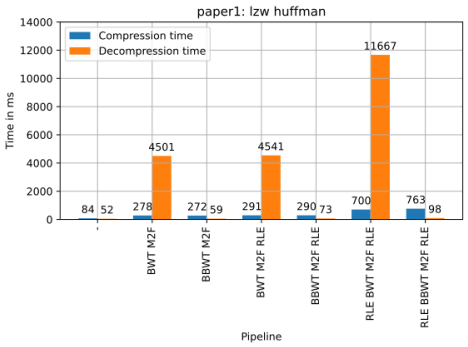


Figure 47: tempi di esecuzione

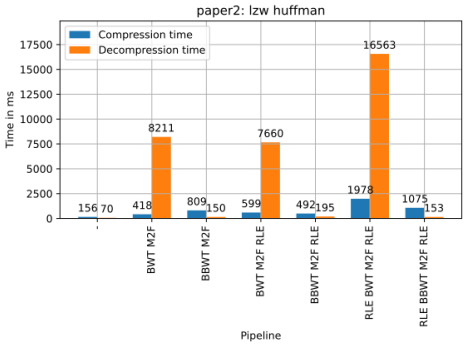


Figure 48: tempi di esecuzione

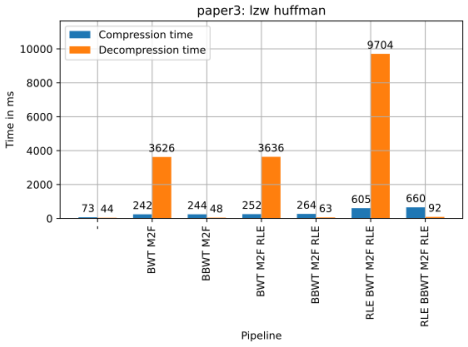


Figure 49: tempi di esecuzione

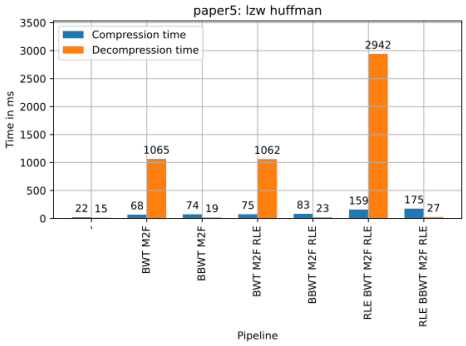


Figure 50: tempi di esecuzione

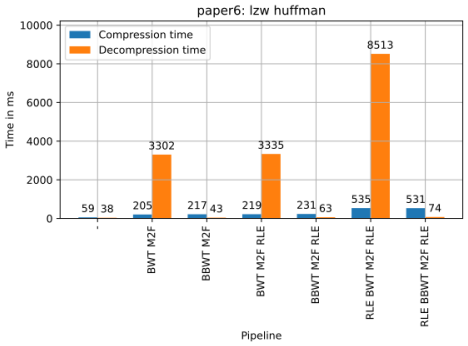


Figure 51: tempi di esecuzione

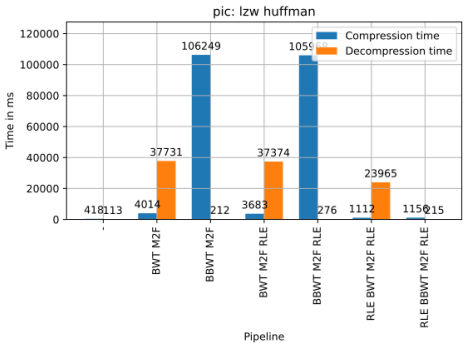


Figure 52: tempi di esecuzione

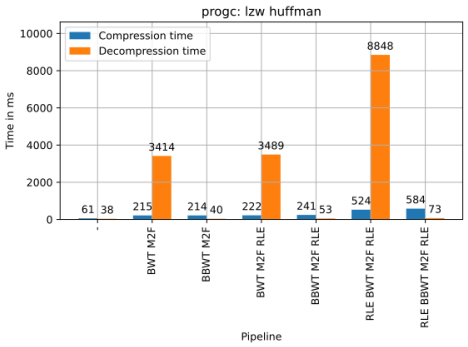


Figure 53: tempi di esecuzione

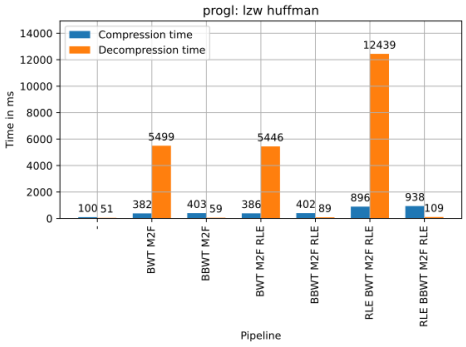


Figure 54: tempi di esecuzione

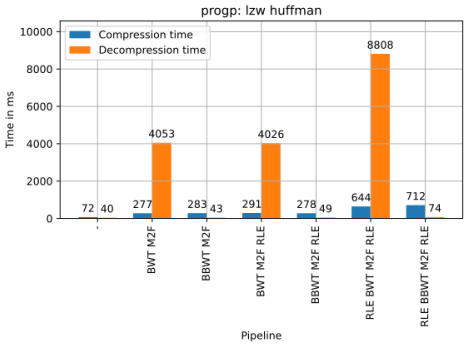


Figure 55: tempi di esecuzione

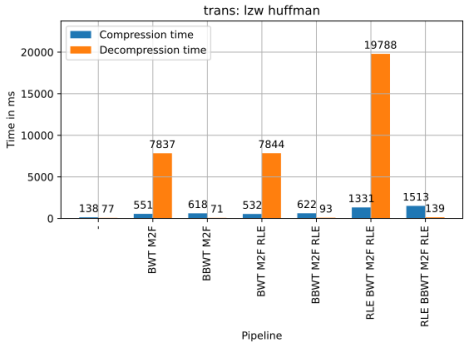


Figure 56: tempi di esecuzione

6 Conclusioni

Nella sezione precedente abbiamo descritto i risultati finali ottenuti dai diversi test effettuati. In generale possiamo concludere dicendo che la trasformata BWT combinata all'algoritmo di M2F. L'utilizzo della RLE permette di aumentare il ratio di compressione solo in alcuni casi, mentre in altri il ratio di compressione tende a diminuire. Possiamo notare come il ratio di compressione risulta avere lo stesso andamento sia nel caso in cui si utilizza Huffman, Arithmetic Code e LZW combinato ad Huffman. I valori più alti di compressione si ottengono tramite l'utilizzo di LZW combinato con la codifica di Huffman.
Per quanto riguarda la differenza tra l'utilizzo della BWT e la sua variante biettiva, possiamo notare come il ratio di compressione è quasi identico. Questo è dovuto al fatto che la BWT biettiva permette di risparmiare un byte per ogni chunk. Analizzando però i tempi necessari per la compressione e decompressione, possiamo notare come il tempo per calcolare l'inversa della BWT biettiva è notevolmente inferiore rispetto al calcolo dell'inversa della BWT.

References

[1] M. Kufleitner, "On bijective variants of the burrows-wheeler transform," *arXiv preprint arXiv:0908.0239*, 2009.

[2] L. G. Ceruso Raffaele, "Secure compression and pattern matching based on burrows-wheeler tansform," *none*, 2018.

[3] T. A. Welch, "A technique for high-performance data compression," *Computer*, vol. 17, no. 06, pp. 8–19, 1984.

[4] U. Manber and G. Myers, "Suffix arrays: a new method for on-line string searches," *siam Journal on Computing*, vol. 22, no. 5, pp. 935–948, 1993.

[5] J. Y. Gil and D. A. Scott, "A bijective string sorting transform," *arXiv preprint arXiv:1201.3077*, 2012.