UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Informatica

Corso di Laurea Magistrale in Informatica

CORSO DI COMPRESSIONE DATI

Secure Compression Based on Burrows-Wheeler Transform

STUDENTI

Lorenzo Criscuolo

Orazio Cesarano

Vincenzo Emanuele Martone

DOCENTE

Prof. **Bruno Carpentieri**

Anno Accademico 2022-2023

Sommario

Il progetto, svolto nell'ambito del corso di *Compressione Dati*, si pone come obiettivo quello di proporre un'implementazione di un metodo di compressione sicura di testo basato sulla funzione di permutazione nota in letteratura come *Burrows-Wheeler Transform*. Il paper *Secure Compression and Pattern Matching Based on Burrows-Wheeler Transform* [1] tratta tale metodo di compressione, di cui il docente del corso ha fornito un'implementazione preesistente con l'obiettivo di apportare un miglioramento. Tale implementazione propone una versione non ottimizzata dell'algoritmo trattato dal paper di cui sopra causando, in questo modo, un'espansione della dimensione dei dati che vengono forniti in input. Lo scopo dell'implementazione proposta dal presente lavoro è quello di migliorare l'implementazione di cui si dispone. Per raggiungere tale obiettivo verranno effettuate delle modifiche all'implementazione degli algoritmi che compongono la *pipeline*. Tali miglioramenti riguardano sia l'utilizzo del paradigma *multiprocessing* volto alla parallelizzazione dei passi dell'algoritmo, che un adattamento degli algoritmi alla *pipeline* proposta. Verranno, inoltre, effettuate delle revisioni all'implementazione dell'algoritmo *blocky-Move To Front* volte all'individuazione dei valori ottimali dei parametri di cui la versione sicura di tale algoritmo deve tenere conto. Infine verranno svolti dei confronti in termini prestazionali (tempo di esecuzione e rapporto di compressione) tra alcune varianti dell'algoritmo proposto che fanno uso di diversi algoritmi di compressione della famiglia *variable length Prefix Code* al fine di individuare il compressore che meglio si adatta alla strategia proposta ed implementata.

Indice

Elenco delle figure

Elenco delle tabelle

CAPITOLO 1

Introduzione

Nell'ambito del presente capitolo verrà effettuata una panoramica generale sulle problematiche trattate dal lavoro svolto, motivando la necessità della costruzione di un algoritmo di compressione sicura di testo. Verrà, successivamente, presentata un'idea ad alto livello della soluzione proposta, evidenziando le modalità mediante le quali questa vada a risolvere le problematiche descritte. Infine, verrà descritta la struttura del lavoro presentato in termini di suddivisione in capitoli.

1.1 Problematiche trattate

Una delle operazioni fondamentali nell'ambito del *processing* delle stringhe è il **pattern matching**: tale operazione consiste nell'individuazione di tutte le occorrenze di un determinato *pattern* in una data stringa input. È possibile effettuare tale tipo di operazione su dati compressi seguendo diverse strategie, la più semplice delle quali consiste nel decomprimere, in primo luogo, il file per poi successivamente effettuare le operazioni di *pattern matching* sul file decompresso. Per questioni legate alle prestazioni e all'utilizzo della memoria, vengono generalmente preferite strategie che non richiedano una decompressione dei file e che riescano a lavorare sui file compressi. Al fine di raggiungere tale obiettivo è necessario utilizzare indici di dati compressi che, se dovessero essere conservati su server di terze parti, porterebbero a gravi problematiche di privacy e confidenzialità. La soluzione concettualmente più semplice consiste nel cifrare i dati dopo che questi sono stati compressi, seguendo, in questo modo, il paradigma *encryption-after-compression*. Ulteriori approcci studiati in letteratura consistono nell'integrazione di compressione e cifratura in un unico passaggio sfruttando strutture di dati compressi utilizzate nei classici algoritmi di compressione. Diverse ricerche hanno, tuttavia, mostrato che tali approcci presentano diverse problematiche di sicurezza. In particolare il paradigma *encryption-after-compression* risulta essere insicuro anche nell'ambito di noti compressori come *WinZip 9.0* [2] [3], *WinRAR 3.42* [4] e *PKZIP 1.10 e 2.04g* [5] [6]. Infine risulta opportuno sottolineare che nessuno degli approcci discussi fino a questo momento risulta essere in grado di conservare le strutture necessarie al pattern matching.

1.2 Soluzione proposta

Le problematiche che sorgono dalla ricerca di una strategia volta alla costruzione di un algoritmo in grado di svolgere le operazioni di *pattern matching* su dati compressi, portano alla necessità di progettare un approccio *ad-hoc* per risolvere i problemi di sicurezza presenti nelle diverse soluzioni discusse. Gli autori del paper *Secure Compression and Pattern Matching Based on Burrows-Wheeler Transform* [1] hanno proposto una soluzione basata su alcuni algoritmi (ben noti in letteratura) revisionati al fine di garantire le proprietà di sicurezza che li rendano adatti ad un utilizzo reale. La *pipeline* proposta dagli autori del paper [1] è costruita come segue:

- La stringa input viene data all'algoritmo **sBWT (scrambled Burrows-Wheeler Transform)**. L'output dell'algoritmo sarà una permutazione della stringa input avente

lunghe sequenze di caratteri ripetuti vicini tra loro;

- L'output della *sBWT* viene dato in input alla **bMTF (blocky Move-To-Front**). Sfruttando il fatto che la *sBWT* restituisca lunghe sequenze di caratteri ripetuti vicini tra loro, tale algoritmo costruisce un output contenente lunghe sequenza di *0*;
- L'output della *bMTF* viene dato in input alla **RLE (Run-Length Encoding)**. Tale algoritmo fornisce una codifica compatta dell'input facendo uso di un contatore per esprimere in maniera concisa le sequenze di *0* ripetuti;
- L'ultimo passo della *pipeline* consiste nell'applicazione di un algoritmo di **Prefix Code** sull'output della *RLE*. La scelta dell'algoritmo di *Prefix Code* influisce sul rapporto di compressione dell'algoritmo complessivo;

La pipeline descritta fa uso di un layer di sicurezza implementato dagli algoritmi *sBWT* e *bMTF* che risultano essere, rispettivamente, la revisione degli algoritmi noti in letteratura come *Burrows-Wheeler Transform (BWT)* e *Move-To-Front (MTF)*. L'algoritmo proposto, inoltre, costruisce due strutture dati ausiliarie che vengono sfruttate per il *pattern matching*. Tali strutture vengono opportunamente manipolate al fine di evitare l'introduzione di vulnerabilità che vanno ad intaccare il layer di sicurezza implementato. La costruzione di tali strutture ed il protocollo utilizzato per l'implementazione del pattern matching sono ampiamente trattati in [1]. Il presente lavoro si pone l'obiettivo di effettuare una panoramica dettagliata della pipeline proposta dagli autori del paper [1] e di proporre un'implementazione in *Python* di un algoritmo di compressione **lossless** sicura di testo che sfrutti tale *pipeline* e che risulti essere quanto più efficiente possibile puntando a massimizzare il fattore di compressione. Risulta, infine, doveroso sottolineare che l'algoritmo proposto risulta essere sicuro secondo la definizione di sicurezza denominata **IND-CPA sicurezza** opportunamente definita dagli autori del paper [1] stesso, dal quale è possibile reperire ulteriori dettagli sull'algoritmo di *pattern matching* e sulle garanzie di sicurezza in termini matematici e formali.

1.3 Struttura della tesina

La trattazione del presente lavoro viene suddivisa in cinque capitoli di cui viene fornita, di seguito, una breve descrizione:

- Il capitolo 1 introduce le problematiche trattate e la soluzione proposta;

- Il capitolo 2 effettua una trattazione degli algoritmi utilizzati dalla *pipeline* proposta analizzando le motivazioni che hanno portato alla scelta di tali algoritmi;
- Il capitolo 3 analizza gli algoritmi trattati nel precedente capitolo ponendo particolare enfasi sulle scelte implementative che hanno portato all'ottenimento di determinati risultati in termini prestazionali e di fattore di compressione;
- Il capitolo 4 descrive l'attività di testing svolta nell'ambito dell'implementazione proposta, presentando i risultati ottenuti utilizzando il *Dataset* considerato;
- Il capitolo 5 conclude la trattazione con diverse considerazioni finali sul lavoro svolto e sugli eventuali sviluppi futuri;

CAPITOLO 2

Algoritmi utilizzati

Nell'ambito del presente capitolo sarà effettuata una trattazione degli algoritmi utilizzati dalla *pipeline* che si intende implementare. La trattazione in questione è volta all'analisi delle motivazioni che hanno portato alla scelta degli algoritmi stessi. Verranno, in primo luogo, descritti nel dettaglio gli algoritmi di base che sono stati scelti per costruire la *pipeline*; successivamente verranno passate a rassegna le modifiche apportate a tali algoritmi al fine di aggiungere un layer di sicurezza alla *pipeline*.

2.1 Pipeline iniziale

Gli autori di [1] hanno proposto una *pipeline* composta da diversi algoritmi, alcuni dei quali opportunamente modificati al fine di aggiungere un layer di sicurezza al processo di compressione. Nelle sezioni successive verrà fornita una descrizione delle versioni di base di tali algoritmi volta all'agevolazione della comprensione del funzionamento dell'algoritmo complessivo. L'algoritmo di compressione *lossless* di partenza prenderà in input un testo T da comprimere e sarà composto da *Burrows-Wheeler Transform, Move-To-Front, Run-Length Encoding e Variable Length Prefix Code*, restituendo un testo compresso C. Specularmente, l'algoritmo di decompressione prenderà in input il testo compresso C e sarà composto da *Inverse Variable Length Prefix Code, Inverse Run-Length Encoding, Inverse Move-To-Front e Inverse Burrows-Wheeler Transform*, restituendo il testo non compresso di partenza T.

2.1.1 Burrows-Wheeler Transform

La **Burrows-Wheeler Transform (BWT)** è un algoritmo utilizzato in diversi programmi di compressione come *bzip*, tuttavia è importante precisare che non si tratta di un algoritmo di compressione in quanto ha come unico scopo quello di permutare l'ordine dei caratteri della stringa input. Il funzionamento della *BWT* si suddivide in tre step fondamentali:

1. Data una stringa input S definita su un alfabeto Σ, si aggiunge un carattere speciale $\$ \notin \Sigma$ alla fine della stringa S ottenendo la stringa S'. Il carattere \$ sarà più piccolo in ordine lessicografico rispetto ad ogni altro carattere di Σ;

2. Si costruisce una matrice M le cui righe sono gli shift ciclici di S';

3. Si dispongono le righe di M in ordine lessicografico ottenendo la matrice M' e si restituisce l'ultima colonna di M' come risultato della *BWT*;

I vantaggi dell'utilizzo della *BWT* risiedono in due fattori principali. In primo luogo tale algoritmo tende a permutare i caratteri della stringa input ponendo quelli ripetuti vicini tra loro, grazie al fatto che le righe della matrice vengono disposte in ordine lessicografico. Ciò fa sì che la stringa restituita dalla *BWT* sia facilmente comprimibile ed è proprio questa caratteristica dell'output ad aver influenzato la scelta dei successivi algoritmi utilizzati nella *pipeline*. La caratteristica appena descritta può facilmente essere ottenuta da un semplice algoritmo che effettua un ordinamento lessicografico dei caratteri della stringa di input; la peculiarità di tale trasformata risiede nel fatto che sia reversibile, fungendo, dunque da base solida per un algoritmo di compressione *lossless*. L'algoritmo che inverte la trasformata prende in

input il risultato R della BWT (dunque l'ultima colonna della matrice). Dal momento che tale colonna contiene tutti i caratteri del testo, basterà effettuarne un ordinamento lessicografico per ottenere la prima colonna della matrice originale ordinata. R e la prima colonna, poste vicine tra loro, formano tutte le coppie di caratteri consecutivi dell'input. Si ordinano tali coppie ottenendo la prima e la seconda colonna della matrice originale ordinata. Si itera tale processo affiancando R e le coppie appena costruite e dopo aver effettuato un ordinamento lessicografico delle triple si ottengono le prime tre colonne della matrice originale. Tale processo viene iterato fino ad ottenere la matrice originale. La riga che termina con il carattere \$ conterrà l'input della trasformata.

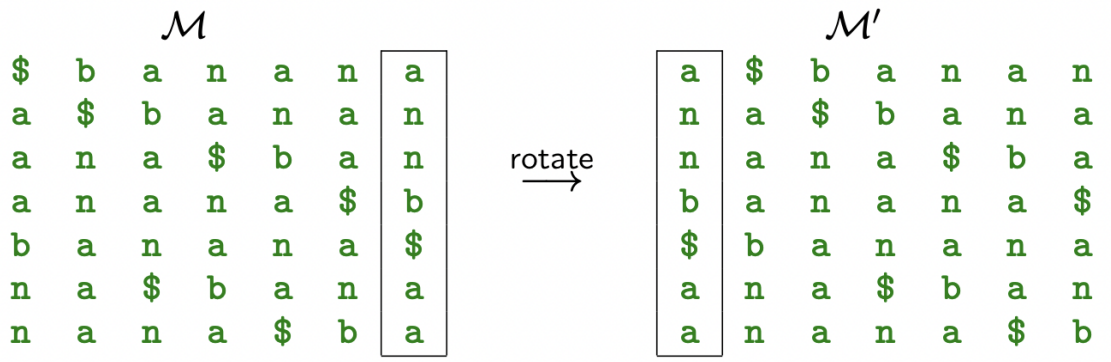

Figura 2.1: Fonte: https://www.cs.helsinki.fi/u/tpkarkka/opetus/12k/dct/lecture08.pdf

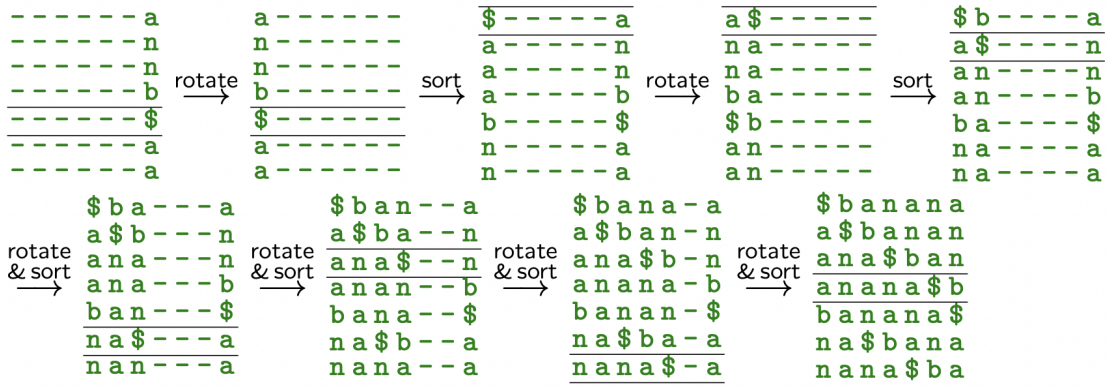

Figura 2.2: Fonte: https://www.cs.helsinki.fi/u/tpkarkka/opetus/12k/dct/lecture08.pdf

Le figure 2.1 e 2.2 chiariscono il funzionamento della trasformata e della sua inversa mediante un semplice esempio.

2.1.2 Move-to-Front

La **Move-to-Front (MTF)** è un algoritmo di codifica dei dati che consiste nel sostituire ogni simbolo della stringa input con la sua posizione in un alfabeto. Il funzionamento della *MTF* si suddivide in tre step fondamentali:

1. Data una stringa input S definita su un alfabeto Σ, viene costruita una lista L composta dai simboli di Σ disposti secondo un certo ordinamento K;

2. Per ogni simbolo s di S si codifica s con la sua posizione p in L, si aggiunge p alla stringa O da restituire in output e si sposta s in cima alla lista L;

3. L'output dell'algoritmo sarà la stringa O e l'ordinamento K;

Il vantaggio dell'utilizzo della *MTF* risiede nel fatto che nel caso di input aventi lunghe sequenze di caratteri uguali, costruirà output aventi lunghe sequenze di 0 in quanto i caratteri ripetuti avranno sempre la stessa posizione nell'alfabeto (che sarà proprio 0). La *MTF* risulta essere invertibile a patto di conoscere l'ordinamento K dei simboli dell'alfabeto Σ. Sostituendo, infatti, ogni simbolo della stringa codificata (che sarà un insieme di posizioni) con il carattere corrispondente nell'alfabeto Σ al quale viene applicato l'ordinamento O e portando tale carattere in cima all'alfabeto ad ogni iterazione, è possibile risalire alla stringa input.

input_str chars	output_arr	list
p	15	abcdefghijklmnop**p**qrstuvwxyz
a	15 1	p**a**bcdefghijklmnoqrstuvwxyz
n	15 1 14	apbcdefghijklm**n**oqrstuvwxyz
a	15 1 14 1	n**a**pbcdefghijklmoqrstuvwxyz
m	15 1 14 1 14	anpbcdefghijkl**m**oqrstuvwxyz
a	15 1 14 1 14	m**a**npbcdefghijkloqrstuvwxyz

Figura 2.3: Fonte: https://www.geeksforgeeks.org/move-front-data-transform-algorithm/

input arr chars	output str	list
15	p	abcdefghijklmnopqrstuvwxyz
1	pa	pabcdefghijklmnoqrstuvwxyz
14	pan	apbcdefghijklmnoqrstuvwxyz
1	pana	napbcdefghijklmoqrstuvwxyz
14	panam	anpbcdefghijklmoqrstuvwxyz
1	panama	manpbcdefghijkloqrstuvwxyz

Figura 2.4: Fonte: https://www.geeksforgeeks.org/inverting-move-front-transform/

Le figure 2.3 e 2.4 chiariscono il funzionamento della *MTF* e della sua inversa mediante un semplice esempio.

2.1.3 Run-length encoding

La **Run-length encoding** è il primo algoritmo di compressione che viene effettivamente utilizzato nella *pipeline*. Tale algoritmo consiste nel rappresentare sequenze di caratteri ripetuti di una stringa in maniera compatta mediante l'utilizzo di un contatore, sostituendo le occorrenze multiple del carattere ripetuto in questione con una coppia $(c, counter)$, dove c è il carattere ripetuto stesso e *counter* indica il numero di volte in cui tale carattere si ripete. Tale algoritmo è banalmente invertibile sostituendo ogni coppia $(c, counter)$ con un numero di occorrenze di c pari a *counter*. Il vantaggio dell'utilizzo della *RLE* risiede nel fatto che, lavorando sull'output delle precedenti trasformate, si troverà a comprimere stringhe aventi lunghe sequenze consecutive di 0. Tale caratteristica dell'input fa in modo che questo possa essere rappresentata in maniera concisa riducendone significativamente la lunghezza.

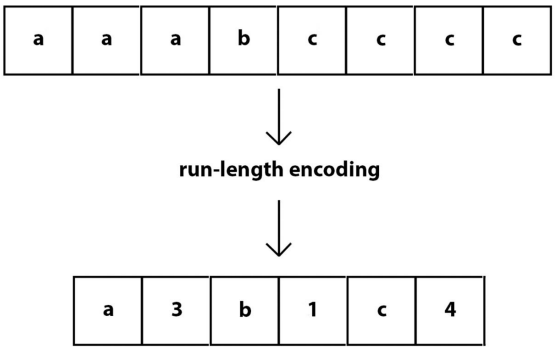

Figura 2.5: Fonte: https://api.video/what-is/run-length-encoding

La figura 2.5 chiarisce il funzionamento della *RLE* mediante un semplice esempio.

2.1.4 Variable length Prefix Code

In generale, un algoritmo di **Variable length Prefix Code** consiste nel mappare i simboli della stringa sorgente su un numero variabile di bit. Alcune delle codifiche più note in letteratura sono *Huffman coding*, *Lempel-Ziv-Welch coding* e *Arithmetic coding*. La scelta dell'algoritmo di *Variable length Prefix Code* da inserire nella *pipeline* influisce sul rapporto di compressione dell'algoritmo complessivo. Risulta opportuno sottolineare che tali algoritmi, essendo reversibili, possono essere utilizzati per la costruzione di un algoritmo di compressione *lossless*.

2.2 Pipeline sicura

La *pipeline* presentata nel paragrafo precedente è strutturata in modo da far lavorare ogni algoritmo che la compone in sinergia con gli altri. Ciascun algoritmo descritto risulta, inoltre, facilmente invertibile, ragion per cui è possibile definire il processo di compressione complessivo *lossless*. Ciò che manca a tale algoritmo è un layer di sicurezza, agevolmente implementabile negli algoritmi descritti mediante opportune modifiche che saranno trattate di seguito. Gli algoritmi interessati da tali modifiche sono la *Burrows-Wheeler Transform* di cui verrà fornita la variante sicura denominata *Scrambled Burrows-Wheeler Transform* e la *Move-to-Front* di cui verrà fornita la versione sicura denominata *Blocky Move-to-Front*. Il motivo per il quale è necessario distribuire il layer di sicurezza su due algoritmi risiede nel fatto che precedenti tentativi da parte di ricercatori di costruire un algoritmo modificando unicamente la *BWT* hanno "scoperto il fianco" ad attacchi di tipo statistico [7]. Attuando opportune modifiche alla *MTF* è possibile rendere l'algoritmo di compressione complessivo sicuro rispetto a tali attacchi. L'algoritmo di compressione *lossless* sicuro prenderà in input un testo T da comprimere e una chiave segreta K e sarà composto da *Scrambled Burrows-Wheeler Transform*, *Blocky Move-To-Front*, *Run-Length Encoding e Variable Length Prefix Code*, restituendo un testo compresso C. Specularmente, l'algoritmo di decompressione prenderà in input il testo compresso C e la chiave segreta K e sarà composto da *Inverse Variable Length Prefix Code, Inverse Run-Length Encoding*, *Inverse Blocky Move-To-Front e Inverse Scrambled Burrows-Wheeler Transform*, restituendo il testo non compresso di partenza T. La sicurezza della *pipeline* appena descritta poggia le proprie fondamenta sul concetto di **IND-CPA sicurezza**. Intuitivamente, tale nozione di sicurezza garantisce che un avversario a cui non è nota la chiave segreta K non sia in grado di stabilire se un testo compresso C sia il risultato della compressione sicura di una stringa T_0 o di una stringa T_1 (dove T_0 e T_1 sono stringhe *isomorfe*[1] scelte dall'avversario stesso che ha accesso ad un oracolo di compressione che comprime utilizzando K) sfruttando una strategia migliore del "tirare a indovinare". Gli autori di [1] forniscono, oltre che una definizione formale di *IND-CPA sicurezza*, una prova del fatto che l'algoritmo in questione sia *IND-CPA sicuro*. Nei prossimi paragrafi verranno descritte le modifiche da apportare agli algoritmi per far sì che l'algoritmo di compressione soddisfi tale nozione di sicurezza.

[1] Due stringhe s e t sono isomorfe se ogni occorrenza di ogni carattere di s può essere sostituita per ottenere t mantenendo l'ordinamento dei caratteri, e.g., *egg* e *add* sono stringhe isomorfe, mentre *foo* e *bar* non lo sono.

2.2.1 Scrambled Burrows-Wheeler Transform

Per aggiungere un layer di sicurezza alla *BWT* è possibile modificarla in modo tale che questa prenda in input una chiave segreta K, mediante la quale viene calcolato un ordinamento lessicografico segreto che servirà per disporre le righe della matrice generata dall'algoritmo. L'algoritmo risultante da tale modifica, denominato **Scrambled Burrows-Wheeler Transform** prende in input:

- una stringa S da comprimere definita su un alfabeto Σ;
- una chiave segreta K;
- una funzione di permutazione basata su chiave *Perm*;

Il funzionamento dell'algoritmo si suddivide in quattro step fondamentali:

1. Si sceglie un numero random r e si computa $Perm(r, K) \rightarrow \xi$, dove ξ denota l'ordinamento lessicografico segreto;
2. Si aggiunge un carattere speciale $\$ \notin \Sigma$ alla fine della stringa S ottenendo la stringa S'. Il carattere \$ sarà più piccolo in ordine lessicografico rispetto ad ogni altro carattere di Σ;
3. Si costruisce una matrice M le cui righe sono gli shift ciclici di S';
4. Si dispongono le righe di M nell'ordine lessicografico segreto ξ ottenendo la matrice M' e si restituisce l'ultima colonna di M' come risultato della *sBWT*;

L'unica differenza tra le due versioni dell'algoritmo risiede nel fatto che mentre nella *BWT* le righe della matrice M vengono disposte in ordine lessicografico, nella *sBWT* tale ordinamento lessicografico varia in base alla chiave K. L'utilizzo di un ordinamento lessicografico fa sì che anche le stringhe di output della *sBWT* presentino lunghe sequenze di caratteri ripetuti, dunque il vantaggio intrinseco della *BWT* viene preservato anche nella sua versione sicura. La proprietà di invertibilità della *sBWT* vale a patto di conoscere l'ordinamento lessicografico segreto utilizzato per disporre le righe della matrice M. In altri termini è necessario conoscere la chiave segreta K, il numero random r generato in fase di compressione e la funzione di permutazione *Perm*.

2.2.2 Blocky Move-to-front

La modifica da apportare alla *Move-to-front* al fine di aggiungere un layer di sicurezza non differisce molto da quella che interessa la *BWT*. Anche questa revisione dell'algoritmo, infatti, poggia la proprie fondamenta sulla costruzione di permutazioni segrete dell'alfabeto utilizzato. In particolare, la versione sicura della *MTF*, denominata **Blocky Move-to-front** prende in input:

- una stringa S da comprimere definita su un alfabeto Σ;
- un parametro intero L che indica la dimensione del blocco;
- una chiave segreta K;
- una funzione di permutazione basata su chiave *Perm*;
- un vettore di inizializzazione IV;
- una funzione hash f

Il funzionamento dell'algoritmo si suddivide in tre step fondamentali:

1. La stringa S viene suddivisa in un certo numero di blocchi, ciascuno dei quali costituito da L caratteri;
2. Per ogni blocco $block_i$ si costruisce una permutazione $Perm(K, IV \text{ or } f(block_{i-1})) \rightarrow \xi_i$ dei caratteri nell'alfabeto Σ. Durante la prima iterazione dell'algoritmo il secondo parametro di *Perm* è IV, mentre durante le iterazioni successive sarà l'hash (calcolato mediante la funzione f) del blocco precedente;
3. Per ogni blocco $block_i$ costruito si eseguono gli step classici della MTF utilizzando l'ordinamento segreto computato;

Nel momento in cui la *bMTF* si trova a lavorare sugli output della *sBWT* produce stringhe contenenti lunghe sequenze di 0. La lunghezza di tali sequenze dipende dal valore del parametro L: più grande sarà questo parametro, più lunga sarà la sequenza. Una trattazione relativa alla scelta del parametro L verrà svolta nei capitoli successivi. Dal momento che la *bMTF* lavora permutando l'alfabeto ogni L caratteri non è possibile per un avversario condurre attacchi di tipo statistico come accadeva nel caso in cui l'unico algoritmo della *pipeline* interessato dalle revisioni di sicurezza era la *BWT*. L'invertibilità della *bMTF* è garantita a patto di conoscere le permutazioni corrispondenti ad ogni blocco dell'input. In altri termini è necessario conoscere

la chiave segreta K, il vettore di inizializzazione IV, la dimensione del blocco L, la funzione di permutazione $Perm$ e la funzione hash f.

CAPITOLO 3

Implementazione

Nell'ambito del presente capitolo verranno trattate le scelte effettuate per l'implementazione degli algoritmi facenti parte della *pipeline* di compressione. Verrà posta particolare attenzione ai dettagli implementativi della *sBWT* in quanto risulta essere l'algoritmo più pesante in termini di spazio occupato e di tempo impiegato. Verrà, successivamente, fornita una descrizione delle strategie impiegate per l'implementazione della *bMTF* e della *RLE*. Il capitolo si conclude con la descrizione della scelta dell'algoritmo di *Variable Length Prefix Code* da utilizzare.

3.1 Panoramica sullo sviluppo dell'algoritmo

La scelta del linguaggio di programmazione da utilizzare per l'implementazione dell'algoritmo di compressione sicuro è ricaduta su *Python*. Tale scelta è stata guidata dalla ricerca di un linguaggio semplice e flessibile che consentisse di implementare in modo agevole e veloce gli algoritmi descritti soffermandosi sui dettagli rilevanti. Il codice sorgente dell'algoritmo sviluppato è reperibile al seguente link: https://github.com/vincenzo-emanuele/Progetto-Compressione-Dati. Come mostrato nella figura 3.1 il progetto è stato suddiviso in 4 *packages*: *sbwt, bmtf, pc e rle*, ognuno dei quali contiene il codice sorgente del corrispondente algoritmo della *pipeline*. L'implementazione delle *pipeline* di compressione e di decompressione è demandato a due file che fungono da tester dei due processi, denominati *compressione.py* e *decompressione.py*. La *pipeline* complessiva è implementata dal file *tester.py* che orchestra l'esecuzione invocando il modulo di compressione e quello di decompressione. La cartella *TestFiles* contiene i file input utilizzati per testare l'algoritmo con i relativi file output prodotti dal processo di compressione.

Figura 3.1: Struttura del progetto

3.2 Implementazione della sBWT

La prima importante questione da affrontare in fase di implementazione della *sBWT* riguarda la realizzazione del layer di sicurezza che si concretizza nell'utilizzo di una chiave segreta per computare un ordinamento lessicografico segreto da utilizzare in fase di disposizione delle righe della matrice. Risulta, inoltre, necessario affrontare la questione relativa alle prestazioni dell'algoritmo. Implementare la *sBWT* seguendo gli step dell'algoritmo, infatti, non risulta praticabile in quanto la costruzione della matrice porterebbe ad un'esplosione della complessità in termini di tempo e di spazio. Per avere un'idea sulla complessità totale,

basti pensare al fatto che la matrice da costruire contiene un numero di elementi pari al quadrato del numero di caratteri della stringa da comprimere; ciò significa che un file di appena 1MB porterebbe alla costruzione di una matrice di 1.099.511.627.776 elementi, occupando oltre 1TB di memoria. Per evitare tali situazioni è possibile applicare diverse ottimizzazioni che poggiano le proprie fondamenta sul fatto di lavorare sulla matrice senza costruirla esplicitamente, facendo uso di strutture ausiliarie costruite *ad-hoc*. I paragrafi successivi descrivono nel dettaglio l'implementazione del layer di sicurezza e le ottimizzazioni attuate.

3.2.1 Implementazione del layer di sicurezza

Nell'ambito del presente lavoro, la realizzazione del layer di sicurezza della *sBWT* è un'implementazione del lavoro svolto dagli autori di [1]. Una descrizione ad alto livello del funzionamento della *sBWT* è stata fornita nel paragrafo 2.2.1 ed è stata implementata dagli *script* del *package sbwt*. Lo scopo di questo layer di sicurezza è quello di ottenere una permutazione dell'alfabeto su cui è definita la stringa input da comprimere. Nello specifico, mediante la libreria *random* di *Python* viene generato un numero casuale r da 0 a 9999999 che viene salvato su un file denominato *rfile.txt*. A tale r viene accodata la chiave segreta K fornita in input eventualmente dall'utente dell'algoritmo. La chiave risultante $Key = r + K$ viene utilizzata come seed per il generatore di numeri casuali fornito dalla libreria *random* di *Python* mediante il quale vengono generati valori attribuiti a ciascun carattere dell'alfabeto da permutare. In fase di disposizione delle righe della matrice, l'algoritmo di ordinamento farà uso di tale associazione *simbolo dell'alfabeto-numero casuale* creando, così, una permutazione dell'alfabeto input. Tale algoritmo risulta invertibile a patto di salvare il numero casuale r e di conoscere la chiave K; infatti, l'utilizzo dello stesso seed $Key = r + K$ fa sì che i numeri casuali ottenuti dal generatore siano esattamente gli stessi, garantendo, in questo modo, l'ottenimento della permutazione dell'alfabeto utilizzata in fase di compressione.

3.2.2 Suffix Array

Al fine di evitare la costruzione esplicita della matrice della *BWT* sono stati utilizzati i **suffix array**, una struttura dati ben nota nell'ambito degli algoritmi. Formalmente, data una stringa S, il *suffix array* A di S è un array di interi contenente le posizioni iniziali dei suffissi di S in ordine lessicografico. Ad esempio, data la stringa $S = banana\$$ ($\$ è il carattere speciale di *EOF*), si indicizza la stringa come illustrato nella figura 3.2.
Successivamente si considerano tutti i suffissi: {*banana$, anana$, nana$, ana$, na$, a$, $*} e li si

i	1	2	3	4	5	6	7
$S[i]$	b	a	n	a	n	a	$

Figura 3.2: Fonte: https://en.wikipedia.org/wiki/Suffix_array

dispone in ordine lessicografico come illustrato nelle figure 3.3 e 3.4.

Suffix	i
banana$	1
anana$	2
nana$	3
ana$	4
na$	5
a$	6
$	7

Figura 3.3: Fonte: https://en.wikipedia.org/wiki/Suffix_array

Suffix	i
$	7
a$	6
ana$	4
anana$	2
banana$	1
na$	5
nana$	3

Figura 3.4: Fonte: https://en.wikipedia.org/wiki/Suffix_array

Il *suffix array* risultante sarà $S = \{7, 6, 4, 2, 1, 5, 3\}$. Chiarito il funzionamento della struttura dati in questione, risulta utile descrivere il modo in cui la *sBWT* la utilizza. La matrice M utilizzata dalla *sBWT* contiene, su ogni riga, tutti i suffissi di S ordinati in ordine lessicografico. L'algoritmo computa, in primo luogo, il *suffix array* A relativo a S. Dal momento che tale *array* contiene gli indici di tutti i suffissi di S ordinati in ordine lessicografico, vi sarà una corrispondenza biunivoca tra le righe di M e i suffissi "puntati" dagli indici di A. In particolare, la *i-esima* riga di M inizierà con il suffisso "puntato" dall'*i-esimo* elemento di A. Risulta, dunque, immediato risalire all'ultimo carattere dell'*i-esima* riga di M in quanto sarà

proprio l'*i-1-esimo* carattere di L.
L'ottimizzazione appena descritta riduce la complessità spaziale della *sBWT* da *quadratica* a *lineare*. La complessità temporale, invece, dipende dall'implementazione scelta per la costruzione dei *suffix array*; nell'implementazione proposta dal presente lavoro è stata utilizzata un'implementazione avente complessità $\mathcal{O}(n \log^2 n)$. Dal momento che la *sBWT*, una volta aver ottenuto il *suffix array*, computa il suo output in tempo $\mathcal{O}(n)$, l'algoritmo complessivo avrà complessità totale $\mathcal{O}(n \log^2 n)$. La costruzione dei *suffix array* è implementata dalla classe *suffix.py* situata nel package *sbwt*.

3.2.3 Variante a blocchi e parallelizzazione

Al fine di ridurre il tempo richiesto dalla compressione, la *sBWT* viene eseguita su blocchi dell'input di taglia fissata. La dimensione da usare per il blocco è stata scelta in maniera "empirica"; intuitivamente un blocco di dimensione troppo piccola rende meno efficace il raggruppamento dei caratteri effettuato dalla *sBWT* mentre un blocco di dimensione troppo grande rende il *processing* di ciascuno di questi troppo oneroso. Un buon compromesso si ottiene lavorando con blocchi aventi dimensione *300 kB*. Grazie al supporto di *Python* al *multiprocessing* (fornito dalla libreria *multiprocessing*), il *processing* dei blocchi avviene in parallelo, apportando un ulteriore miglioramento al tempo di esecuzione dell'algoritmo.

3.2.4 Calcolo dell'inversa

Il calcolo dell'inversa della trasformata è implementato evitando la costruzione esplicita della matrice al fine di scongiurare un'esplosione della complessità. Il paper [8] propone una strategia per il calcolo dell'inversa della *BWT* che evita la costruzione della matrice. L'algoritmo prende in input il risultato L della *sBWT*, l'ordinamento lessicografico O utilizzato in fase di compressione e restituisce la stringa non compressa S facendo uso di strutture ausiliarie denominate C e P:

- C è un dizionario costruito in modo tale che $C[ch]$ è il numero totale di istanze in L dei caratteri che precedono *ch* nell'ordinamento definito sull'alfabeto;
- P è una lista costruita in modo tale che $P[i]$ è il numero di istanze del carattere $L[i]$ nel prefisso $L[0, \ldots, i-1]$ di L;

Grazie a tali strutture, l'algoritmo riesce ad ottenere S. Le modalità mediante le quali tale stringa viene costruita si basano sul fatto che non sia necessario costruire esplicitamente una

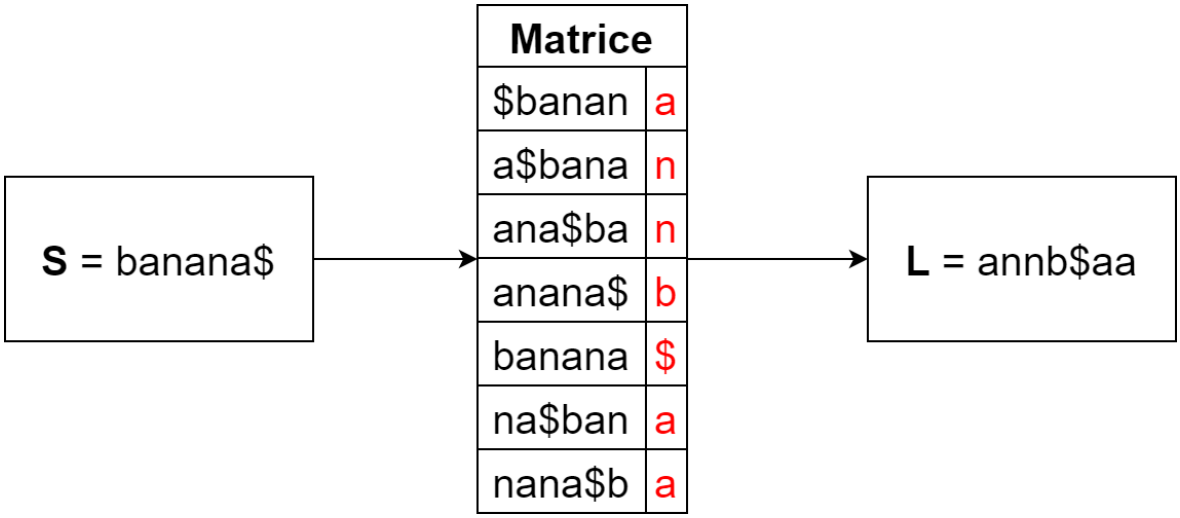

Figura 3.5: Funzionamento della BWT

matrice per sfruttarne le proprietà. Dal momento che il funzionamento dell'algoritmo non risulta essere del tutto intuitivo, la sua trattazione verrà affrontata utilizzando come supporto un esempio pratico. Come illustrato nella figura 3.5 l'output restituito dalla *BWT* sarà l'ultima colonna della matrice ordinata degli shift ciclici di S. Dal momento che il carattere di *EOF* è il più piccolo secondo l'ordinamento lessicografico, la prima riga sarà sempre lo shift ciclico che come primo carattere ha l'*EOF* seguito da S (nel caso dell'esempio la riga 0 della matrice è *$banana*). Ciò implica che l'ultimo carattere del testo da ricostruire (escludendo l'EOF) è il primo carattere di L ($L[i]$, con $i = 0$), in questo caso a. Dal momento che per motivi di praticità l'output verrà ricostruito al contrario, il prossimo carattere da individuare è il predecessore di a in S. Dato che si dispone solo della stringa L, è necessario individuare la riga che termina con il carattere ch cercato in quanto corrisponderà esattamente all'indice di L contenente ch; tale riga conterrà i caratteri della riga appena individuata *shiftati* a destra di un carattere, dunque, comincerà con a. Sfruttando l'ordinamento della matrice è possibile accedere agevolmente alle righe che cominciano per a; a tale scopo risulta utile utilizzare la struttura C in quanto $C[a]$ contiene il numero di occorrenze dei caratteri precedenti ad a nell'ordinamento lessicografico presenti in L (in questo caso $C[a] = 1$). La riga 1 di M inizierà sicuramente per a, tuttavia la matrice potrebbe contenere più righe che cominciano con a, per cui è necessario stabilire quale di queste contenga come ultimo carattere il predecessore di a cercato. Per fare ciò l'algoritmo si serve della struttura P in quanto $P[i]$ indica quante volte il carattere $L[i]$ (in questo caso a) appare all'interno del prefisso $L[0, \ldots, i-1]$. Il valore $P[i]$ fungerà da *offset* indicando, tra tutte le righe che cominciano con a, quella desiderata. A questo punto, l'algoritmo pone $i = C[L[i]] + P[i]$ e ripete questi passi complessivamente n

volte (dove n è la lunghezza di L). Ulteriori dettagli sul funzionamento e sulla correttezza di tale algoritmo sono reperibili in [8].

3.3 Implementazione della bMTF

Il layer di sicurezza implementato nella *bMTF* consiste, in primo luogo, nel suddividere l'input in blocchi di dimensione L per poi, successivamente, eseguire il classico algoritmo di *MTF* descritto nel paragrafo 2.1.2. Il vantaggio di tale operazione risiede nel fatto che cambiare l'alfabeto periodicamente rende inefficaci attacchi di tipo statistico che poggiano le loro fondamenta sul fatto che uno stesso carattere ripetuto più volte nel testo cifrato corrisponda allo stesso carattere nel testo in chiaro. In altri termini, il cifrario a sostituzione monoalfabetica implementato dalla *sBWT* viene, in questo modo, trasformato in un cifrario a sostituzione polialfabetica rendendo, in questo modo, l'algoritmo aderente alla nozione di *IND-CPA sicurezza*. L'algoritmo in questione è implementato dallo script *bmtf.py* situato nel *package bmtf*. Fa uso di un vettore di inizializzazione noto (per semplicità è *hard-coded*) e di una chiave K fornita eventualmente dall'utente dell'algoritmo, che risulta essere la stessa utilizzata per la *sBWT*. Dopo aver suddiviso l'input in blocchi, ciascuno di dimensione L, calcola la permutazione dell'alfabeto da usare per codificare quello specifico blocco facendo uso di un seed S ottenuto dalla concatenazione dell'hash del vettore di inizializzazione (se sta codificando il primo blocco) o dell'hash del blocco precedente (se sta codificando un blocco diverso dal primo) e della chiave K. Analogamente a quanto avviene per la *sBWT*, la permutazione dell'alfabeto viene calcolata utilizzando S come seed per il generatore di numeri casuali fornito dalla libreria *random* di *Python*. Un importante questione da trattare nell'ambito dell'implementazione della *bMTF* riguarda la dimensione dei blocchi in cui suddividere l'input. Intuitivamente, blocchi di dimensione troppo grande portano ad un minor numero di permutazioni dell'alfabeto, mentre blocchi di dimensione troppo piccola portano ad una minore efficacia della *bMTF* (in altri termini le sequenze di 0 saranno più brevi). Il valore da assegnare a L è stato scelto empiricamente ed un buon compromesso è stato raggiunto fissando L a 1024 byte.

3.4 Implementazione della RLE

Dal punto di vista implementativo, non sono state apportate modifiche rilevanti alla *RLE* rispetto alla descrizione illustrata nel paragrafo 2.1.3. L'algoritmo è implementato dallo *script*

rle.py situato nel *package rle*.

3.5 Scelta dell'algoritmo di PC

L'ultimo componente della *pipeline* implementata è l'algoritmo di *variable length Prefix Code*. La scelta dell'algoritmo di *PC* da utilizzare influisce sui tempi di esecuzione e sul rapporto di compressione complessivi. Gli algoritmi di *PC* impiegati nella *pipeline* non sono stati sviluppati da zero ma sono state utilizzate implementazioni in *Python* preesistenti. La scelta dell'algoritmo di *PC* da utilizzare è stata effettuata in maniera empirica in base alle prestazioni riscontrate confrontando tempi di compressione, decompressione e rapporto di compressione; una trattazione approfondita sui risultati ottenuti dal confronto di tali algoritmi è affrontata nel paragrafo 4.3. Lo script *pc.py* situato nel *package pc* implementa gli algoritmi di *PC* utilizzati consentendo, mediante l'utilizzo di un parametro input, di scegliere il compressore desiderato. Nello specifico, gli algoritmi in questione sono: *Huffman Coding*, *Arithmetic Coding* e *Lempel-Ziv-Welch Coding*.

3.6 Implementazione della pipeline

Nell'ambito dell'implementazione effettuata, gli algoritmi descritti fino a questo momento sono "orchestrati" da un modulo che funge da compressore (reperibile nel *main package src* sotto il nome *compression.py*) e da un modulo che funge da decompressore (reperibile nel *main package src* sotto il nome *decompression.py*). I due moduli vengono, a loro volta, invocati dallo script *tester.py* che si occupa di simulare un *workflow* completo di compressione e decompressione prendendo in input da riga di comando il nome del file da comprimere e la chiave segreta da usare durante la *sBWT* e la *bMTF*. Durante un'esecuzione completa dell'algoritmo di compressione vengono generati diversi file, tuttavia solo una parte di essi va conservata per poter effettuare la decompressione. In particolare i file da conservare sono i seguenti: *outputDictBWT.txt*, *outputPC.txt*, *rfile.txt* e *outputPCCodec.txt* (nel caso in cui viene utilizzato *Huffman*) o *outputDictLZW.txt* (nel caso in cui viene utilizzato *LZW*). D'altro canto, l'algoritmo di decompressione produce un unico file denominato *decompressed.txt* che risulta essere identico al file di input. Dopo l'esecuzione completa della *pipeline*, lo script *tester.py* si occupa di verificare che il file di input sia identico all'output della decompressione.

CAPITOLO 4

Testing

Nell'ambito del presente capitolo verranno descritti i risultati ottenuti dalla fase di testing dell'algoritmo implementato. In primo luogo verrà descritto il *Dataset* impiegato per il testing. Successivamente sarà presentato l'ambiente di esecuzione sul quale è stato eseguito l'algoritmo. Infine verranno presentati i tempi di esecuzione di compressione e decompressione con i relativi rapporti di compressione effettuando un confronto tra i risultati ottenuti impiegando diversi algoritmi di *variable length Prefix Code*.

4.1 Dataset

Dal momento che l'algoritmo implementato lavora solo su testo, il *Dataset* impiegato è costituito da diversi file di testo, di tipologia (racconti, codice sorgente, ...) e lunghezza variabile. I suddetti file sono reperibili al seguente link: https://github.com/vincenzo-emanuele/Progetto-Compressione-Dati/tree/main/src/TestFiles/Input.

4.2 Ambiente di esecuzione

Per svolgere la fase di testing dell'algoritmo implementato è stato utilizzato un *MacBook Pro M1 (2020)* avente le seguenti specifiche:

- S.O.: MacOS Montrey 12.2.1;
- CPU: Chip Apple M1;
- RAM: 8 GB;
- SSD : 512 GB;
- Versione di Python installata: 3.8;

4.3 Risultati

La fase di testing condotta ha come scopo quello di verificare la bontà dell'algoritmo in termini computazionali (spazio e tempo) e di appurare che la decompressione vada a buon fine senza perdita di informazione, garantendo, dunque, che l'algoritmo di compressione implementato sia *lossless*. Le tabelle 4.1 e 4.2 indicano, per ogni file del *Dataset*, la sua dimensione non compressa, la sua dimensione a seguito della compressione, il tempo impiegato per la compressione (corrisponde al tempo di esecuzione dello script *compression.py* che implementa la *pipeline* di compressione), il tempo impiegato per la decompressione (corrisponde al tempo di esecuzione dello script *decompression.py* che implementa la *pipeline* di decompressione) e il rapporto di compressione ottenuto. In particolare la prima tabella si riferisce alla *pipeline* di compressione che utilizza *Huffman* come algoritmo di *variable length Prefix Code*, mentre la seconda si riferisce alla *pipeline* che fa uso di *Lempel-Ziv-Welch*. Non è stato effettuato un report riguardante *Arithmetic Coding* in quanto l'implementazione utilizzata risulta essere estremamente inefficiente portando a tempi di esecuzione non ragionevoli (nell'ordine di ore).

Input	Size non compr.	Size compr.	Tempo compr.	Tempo decompr.	%
alice29.txt	152.062 byte	77.081 byte	5.44 s	0.41 s	49.31
asyoulik.txt	125.179 byte	68.812 byte	4.13 s	0.36 s	44.14
cp.html	24.603 byte	12.342 byte	0.15 s	0.03 s	49.83
fields.c	11.150 byte	5.457 byte	0.16 s	0.03 s	51.06
grammar.lsp	3.721 byte	2.295 byte	0.04 s	0.013 s	38.33
huffman.c	6.986 byte	3.582 byte	0.04 s	0.008 s	48.73
lcet10.txt	419.235 byte	202.267 byte	9.51 s	0.70 s	51.75
manual.ps	1.766.625 byte	316.505 byte	15.69 s	1.58 s	82.08
plrabn12.txt	481.861 byte	265.393 byte	10.07 s	0.78 s	44.92
shrek.txt	70.658 byte	40.572 byte	0.87 s	0.09 s	42.58

Tabella 4.1: Risultati ottenuti con Huffman

La *pipeline* di testing è implementata dallo script *testing.py* a cui vanno passati i parametri da linea di comando che indicano il nome del file di input e la chiave segreta da utilizzare.
Come evidenziato dagli istogrammi 4.1, 4.2 e 4.3, non sono state riscontrate differenze evidenti tra l'utilizzo di *Huffman* e quello di *LZW*. Tendenzialmente *Huffman* presenta tempi di esecuzione ed un rapporto di compressione leggermente migliori. Dopo aver eseguito le fasi di compressione e decompressione, la *pipeline* di testing si assicura che il processo sia avvenuto in maniera *lossless*. Per fare ciò confronta il file di input con l'output della decompression e: se il contenuto di questi due differisce anche solo di un bit, comunicherà a video il fallimento della decompressione. Tale verifica ha restituito esito positivo su ognuno dei file del *Dataset* utilizzato. Al fine di comprendere la bontà dell'algoritmo implementato è stata svolta un'attività di confronto con lo stato dell'arte della compressione mediante algoritmi basati sulla trasformata di *Burrows-Wheeler*. In particolare il *Dataset* utilizzato per testare l'algoritmo è stato sottoposto al compressore **bzip2** di cui è stata impiegata un'implementazione *built-in* in *Python* mediante la libreria *bz2*. Dai confronti effettuati (di cui è disponibile un *report* nella tabella 4.3) è emerso che *bzip2* presenta un fattore di compressione circa del 50% migliore rispetto a quello della *pipeline* proposta ed un tempo di esecuzione decisamente più basso sia in fase di compressione che in fase di decompressione.

Input	Size non compr.	Size compr.	Tempo compr.	Tempo decompr.	%
alice29.txt	152.062 byte	81.130 byte	5.61 s	0.26 s	46.65
asyoulik.txt	125.179 byte	73.462 byte	4.10 s	0.22 s	41.31
cp.html	24.603 byte	13.832 byte	0.40 s	0.04 s	43.78
fields.c	11.150 byte	5.788 byte	0.16 s	0.02 s	48.09
grammar.lsp	3.721 byte	2.347 byte	0.04 s	0.007 s	36.93
huffman.c	6.986 byte	3.813 byte	0.09 s	0.01 s	45.42
lcet10.txt	419.235 byte	212.831 byte	14.40 s	1.004 s	49.23
manual.ps	1.766.625 byte	330.373 byte	22.26 s	2.58 s	81.30
plrabn12.txt	481.861 byte	281.687 byte	15.04 s	1.14 s	41.54
shrek.txt	70.658 byte	43.728 byte	2.02 s	0.13 s	38.11

Tabella 4.2: Risultati ottenuti con LZW

Input	Size non compr.	Size compr.	Tempo compr.	Tempo decompr.	%
alice29.txt	152.062 byte	43.172 byte	0.016 s	0.005 s	71.61
asyoulik.txt	125.179 byte	39.569 byte	0.013 s	0.004 s	68.39
cp.html	24.603 byte	7.624 byte	0.003 s	0.0007 s	69.01
fields.c	11.150 byte	3.039 byte	0.002 s	0.0003 s	72.74
grammar.lsp	3.721 byte	1.283 byte	0.0009 s	0.0002 s	65.52
huffman.c	6.986 byte	2.110 byte	0.001 s	0.0003 s	69.80
lcet10.txt	419.235 byte	107.648 byte	0.03 s	0.009 s	74.32
manual.ps	1.766.625 byte	162.220 byte	0.37 s	0.03 s	90.82
plrabn12.txt	481.861 byte	145.545 byte	0.04 s	0.01 s	69.80
shrek.txt	70.658 byte	23.346 byte	0.006 s	0.003 s	66.96

Tabella 4.3: Risultati ottenuti con bzip2

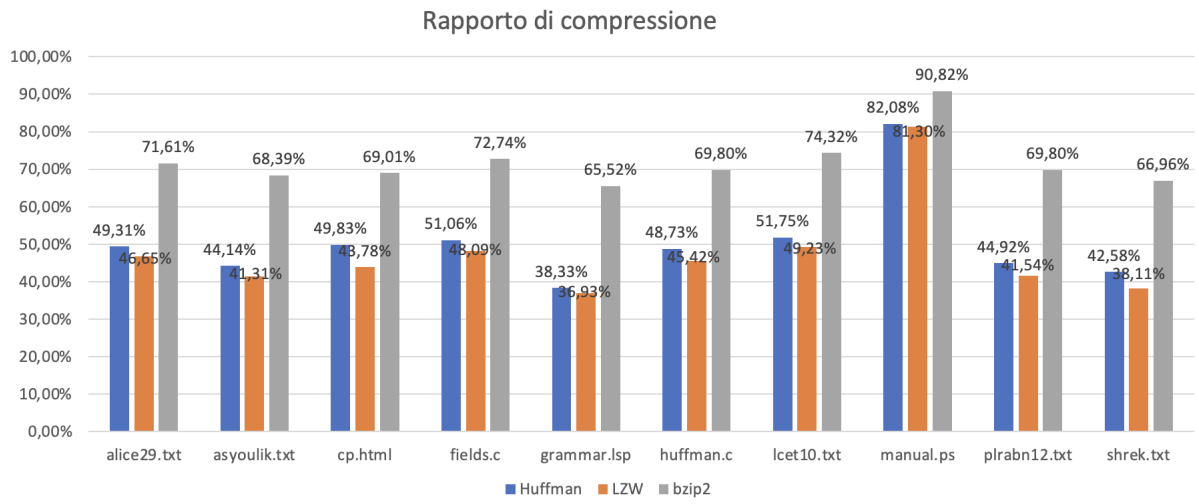

Figura 4.1: Rapporto di compressione

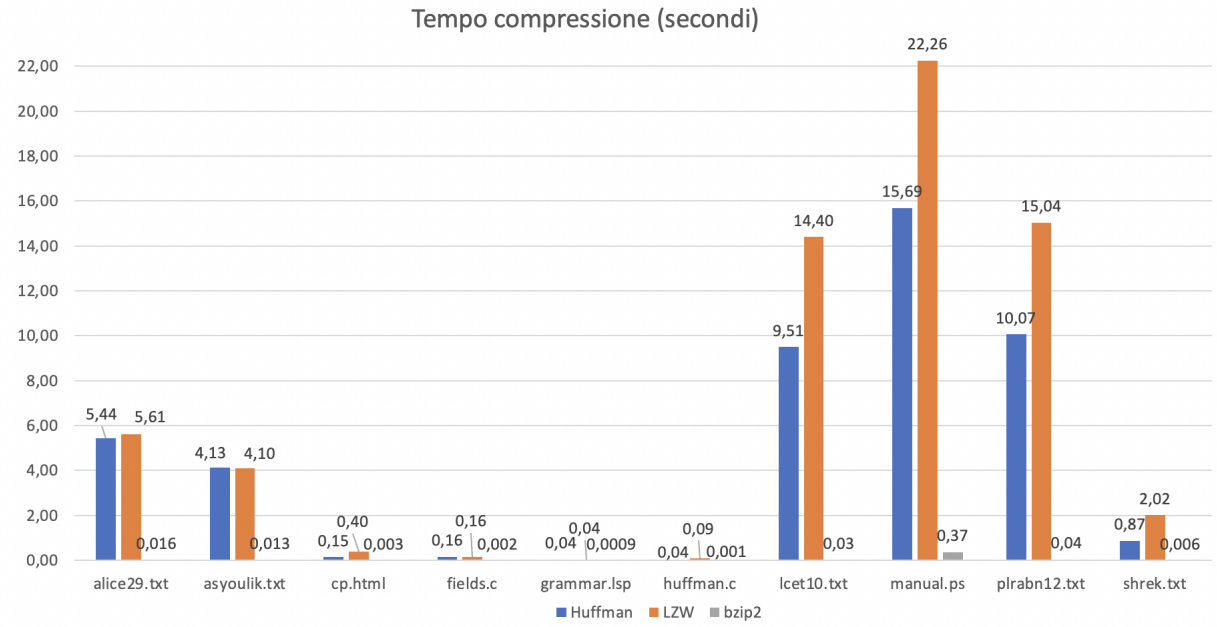

Figura 4.2: Tempo di compressione

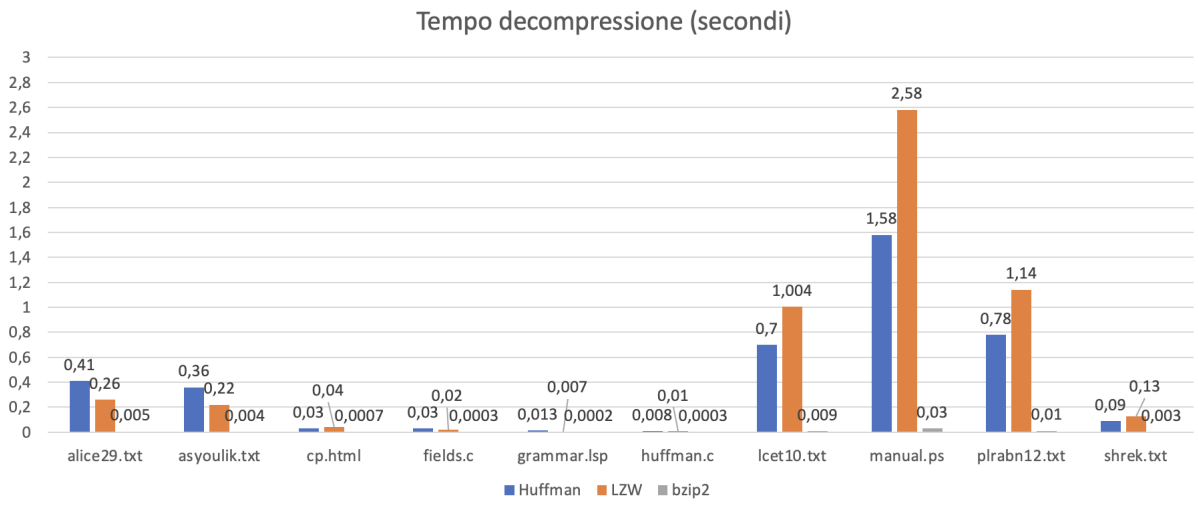

Figura 4.3: Tempo di decompressione

CAPITOLO 5

Conclusioni

Nell'ambito del presente capitolo verranno svolte considerazioni finali sul lavoro effettuato, principalmente dal punto di vista della bontà dell'implementazione proposta. Infine saranno trattati brevemente gli eventuali sviluppi futuri implementabili a partire dal lavoro svolto.

5.1 Considerazioni generali

La fase di implementazione dell'algoritmo è stata preceduta da uno studio dei diversi componenti della *pipeline* complessiva volto alla comprensione del funzionamento della stessa. Il lavoro svolto si propone come obiettivo quello di essere un'implementazione di [1] e di svolgere un confronto tra alcune varianti dello stesso. Il risultato finale risulta essere un algoritmo di compressione sicuro avente un tasso di compressione che si avvicina al 50% nel caso del *Dataset* considerato. Inoltre, grazie al layer di sicurezza distribuito su due componenti differenti, l'algoritmo implementato resiste ad attacchi di tipo statistico e risulta essere una solida base per la costruzione di un algoritmo di *pattern matching* che fa uso di indici di dati compressi salvati su *Cloud*.

5.2 Sviluppi futuri

L'algoritmo implementato presenta diversi spunti a partire dai quali è possibile costruire lavori futuri. Nello specifico, risulta possibile:

- Apportare miglioramenti all'algoritmo grazie alla parallelizzazione della *I-bMTF* in quanto l'inversione lavora su blocchi tra loro indipendenti utilizzando informazioni note all'inizio della decompressione;
- Utilizzare un'implementazione della costruzione dei *suffix array* avente complessità $\mathcal{O}(n)$ al fine di velocizzare la fase di compressione;
- Implementare un algoritmo di *pattern matching* che fa uso dell'algoritmo di compressione implementato mediante la costruzione di opportune strutture di supporto;

Bibliografia

[1] G. Zeng, M. He, L. Zhang, J. Zhang, Y. Chen, and S. M. Yiu, "Secure compression and pattern matching based on burrows-wheeler transform," in *2018 16th Annual Conference on Privacy, Security and Trust (PST)*, pp. 1–10, IEEE, 2018. (Citato alle pagine i, 2, 3, 6, 10, 16 e 29)

[2] T. Kohno, "Attacking and repairing the winzip encryption scheme," in *Proceedings of the 11th ACM conference on Computer and communications security*, pp. 72–81, 2004. (Citato a pagina 2)

[3] P. H. Phong, P. D. Dung, D. N. Tan, N. H. Duc, and N. T. Thuy, "Password recovery for encrypted zip archives using gpus," in *Proceedings of the 2010 Symposium on Information and Communication Technology*, pp. 28–33, 2010. (Citato a pagina 2)

[4] G. S.-W. Yeo and R. C.-W. Phan, "On the security of the winrar encryption feature," *International Journal of Information Security*, vol. 5, no. 2, pp. 115–123, 2006. (Citato a pagina 2)

[5] M. Stay, "Zip attacks with reduced known plaintext," in *International Workshop on Fast Software Encryption*, pp. 125–134, Springer, 2001. (Citato a pagina 2)

[6] E. Biham and P. C. Kocher, "A known plaintext attack on the pkzip stream cipher," in *International Workshop on Fast Software Encryption*, pp. 144–153, Springer, 1994. (Citato a pagina 2)

[7] M. Stanek, "Attacking scrambled burrows-wheeler transform," *Cryptology ePrint Archive*, 2012. (Citato a pagina 10)

[8] M. Burrows and D. Wheeler, "A block-sorting lossless data compression algorithm," in *Digital SRC Research Report*, Citeseer, 1994. (Citato alle pagine 18 e 20)