UNIVERSITÀ DEGLI STUDI DI SALERNO

DIPARTIMENTO DI INFORMATICA

Corso di Laurea Magistrale in Informatica

Compressione Dati 2022-2023

Bruno Carpentieri

Grammar compressor for text file applied on Dataset containing genome strings

Studenti

Giovanni Musacchio Antonio Saporito Andrea Sorrentino

Veronica Marcantuono Aldo Claudini

Table of Contents

Chapter 1

Introduzione

La compressione dati, codifica di sorgente o riduzione bit-rate è il processo di codifica di informazioni utilizzando un minor numero di bit per la rappresentazione originale. La compressione testuale applicata su stringhe di genoma negli ultimi tempi ha portato ad un accumulo rapido di dati genomici, infatti, c'è stata una crescente domanda di metodi efficienti per la compressione di essi. Sono state utilizzate delle tecniche già esistenti combinandole tra esse, l'abbiamo chiamato questo metodo BRLH per la compressione (Burrows-Wheeler-Transform, Run Lenght encoding, Lempel-Ziv-Welch, Huffman Encoding) ed RBRLH per la decompressione cioè sono stati utilizzati gli algoritmi inversi. La nostra scelta di utilizzare questa tecnica è stata perché ha una serie di vantaggi rispetto ad altre tecniche di compressione di genomi.

Alcune funzionalità come Burrows-Wheeler-Transform che va a calcolare le ripetizioni massime che avvengono nell'input, esso che comprime una serie di stringhe in modo significativo anche con collezioni altamente ripetitive. Run Lenght sfrutta molto il contesto della ripetizione, in quanto la BWT associa un numero di ripetizioni per ogni lettera, può quindi fare analisi di sequenza complesse su raccolte di genomi in uno spazio ridotto. Lempel-Ziv ci permette di avere una riduzione di spazio e le grammatiche permettono di manipolare l'insieme di letture in forma compressa, unica pecca riscontrata è che questo algoritmo rallenta un po' la compressione. Infine, Huffman che appartiene a una famiglia di algoritmi a lunghezza variabile delle parole, ciò significa che i

singoli simboli che si verifica frequentemente in una collezione le vine assegnato una frequenza breve e viceversa se meno frequente, vediamo notevole efficienza durante la fase di compressione. Mentre per la decompressione come già accennato prima vengono utilizzati le stesse tecniche utilizzando gli algoritmi inversi.

Possiamo dire che queste non sono le tecniche più comuni che esistano nella bioinformatica, infatti, i genomi vengono ricostruiti a partire da enormi insieme di stringhe di DNA brevi e sovrapposte, chiamate letture di sequenziamento. Come sappiamo l'assemblaggio dei genomi è costoso, in quanto devono essere affrontati calcoli estensivi delle sovrapposizioni tra lettura ed allineamento tra genomi. Di conseguenza queste grandi insieme di letture, sono anche la forma più comune in cui sono disponibili dati di sequenziamento.

Chapter 2

Background

L'algoritmo proposto in questo lavoro presenta l'utilizzo di alcune trasformate e algoritmi di compressione dati già esistenti. In questo capitolo, andremo ad illustrarli al fine di comprendere al meglio l'algoritmo proposto e le sue varie fasi intermedie. Le parti sostanziali illustrate in questo capitolo saranno: Burrows-Wheeler transform (BWT), Lempel-Ziv-Welch (LZW), Run-length Encoding (RLE) e infine la codifica di Huffman.

2.1 Burrows-Wheeler transform (BWT)

La Trasformata di Burrows-Wheeler (BWT) è un algoritmo che prende in input una stringa di caratteri e la riorganizza in modo tale da renderla più comprimibile. Lo fa convertendo la stringa in sequenze di caratteri simili, che possono essere codificati in modo più efficiente utilizzando tecniche come la codifica per lunghezza di corsa. Possiamo dividere l'esecuzione di questa trasformata in tre fasi:

- **Inserimento carattere** Si inserisce un carattere che non fa parte dell'alfabeto della stringa, alla fine di quest'ultima
- **Rotazioni** Si crea una matrice con le rotazioni della stringa. Per ogni riga, si inserisce una rotazione che è lo shift della riga precedente. La prima riga è la stringa originale

- **Riordinamento** Si ordinano le righe della matrici in ordine lessicografico e si dà in output l'ultima colonna della matrice riordinata

Più nel dettaglio, prendiamo una stringa di esempio: "BANANA". A questa stringa, aggiungiamo alla fine un carattere esterno al suo alfabeto, ad esempio $. Innanzitutto, creiamo una matrice di tutte le rotazioni della stringa, la prima riga della matrice è la stringa originale, la seconda riga è la stringa ruotata una volta a sinistra, la terza riga è la stringa ruotata due volte a sinistra, e così via. L'aspetto di questa matrice è visibile nella figura A:

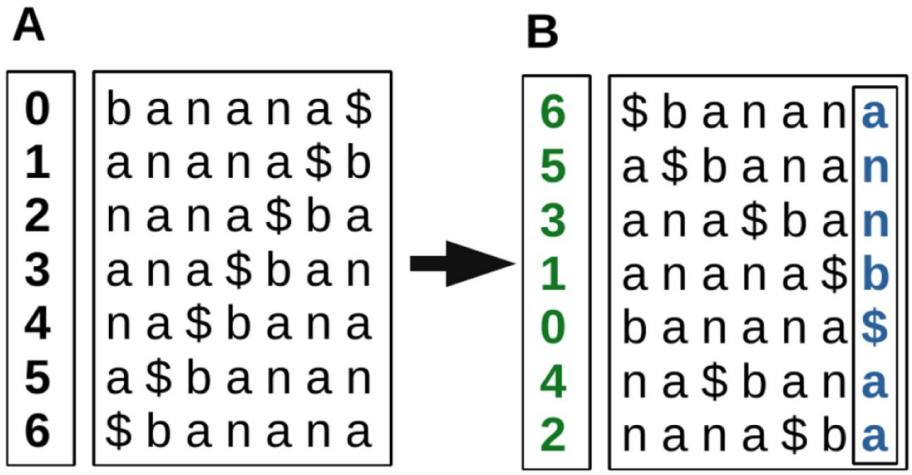


Figure 2.1: Burrows-Wheeler transform of "BANANA"

Nella figura B, invece, troviamo le righe della prima matrice ordinate in maniera lessicografica. Ciò ci permette di calcolare la trasformata prendendo l'ultima colonna di questa matrice, nel nostro caso: annb$aa.

2.2 Lempel-Ziv-Welch (LZW)

L'algoritmo di codifica Lempel-Ziv-Welch (LZW) è un algoritmo di compressione dei dati che è stato descritto per la prima volta da Jacob Ziv e Abraham Lempel nel 1978, e poi ulteriormente perfezionato da Terry Welch nel 1984. È un algoritmo senza perdita di informazione, il che significa che può comprimere i dati senza perdere alcuna informazione.

L'algoritmo funziona sfruttando le ripetizioni di sotto-stringhe all'interno della stringa originale in input. Quando si trova una ripetizione, a quella precisa sotto-stringa viene assegnato un codice e, al termine della ricerca, la stringa viene restituita con i vari codici e le sotto-stringhe non ripetute. La codifica LZW è spesso utilizzata in combinazione con altri algoritmi di compressione, come la codifica di Huffman, per ottenere risultati ancora migliori.

2.3 Run-Lenght Encoding (RLE)

La codifica per lunghezza di corsa (RLE) è una semplice tecnica di compressione dei dati che codifica i dati sostituendo le sequenze di valori dei dati consecutivi con un solo valore e un conteggio del numero di volte in cui quel valore appare. Ciò può essere utilizzato per comprimere i dati che hanno molte sequenze di valori consecutivi, poiché la rappresentazione compressa sarà molto più corta dei dati originali. Per esempio, consideriamo la seguente stringa:

AAABBBCCCCCDDEEEE

Possiamo applicare la RLE a questa stringa sostituendo le sequenze di valori consecutivi con un solo valore e un conteggio del numero di volte in cui quel valore appare. La stringa codificata risultante sarebbe:

3A3B5C2D4E

In questo esempio, la stringa codificata è molto più corta della stringa originale, poiché ha sostituito molte sequenze consecutive di valori con un solo valore e un conteggio.

2.4 Huffman Encoding

La codifica di Huffman è un algoritmo di compressione dei dati utilizzato per rappresentare simboli con codici di lunghezza variabile in modo da minimizzare la lunghezza media dei codici. L'algoritmo utilizza una tecnica di codifica basata sulla frequenza di ogni simbolo nei dati da comprimere. I simboli più frequenti hanno codici più corti, mentre quelli meno frequenti hanno codici

più lunghi. Per creare i codici di Huffman, viene prima calcolata la frequenza di ogni simbolo nei dati e quindi vengono creati dei nodi per ogni simbolo. I nodi vengono quindi combinati in base al loro peso (frequenza) fino a creare un albero di codifica. Una volta creato l'albero, i codici per ogni simbolo vengono ottenuti percorrendo l'albero dalla radice verso i nodi foglia. Ogni volta che si passa da un nodo interno a un figlio, viene aggiunto un "0" per il figlio sinistro o un "1" per il figlio destro al codice. I codici ottenuti vengono quindi utilizzati per codificare i dati originali, creando una rappresentazione compatta dei dati che può essere facilmente decompressa per ottenere i dati originali. La codifica di Huffman è spesso utilizzata nei sistemi di archiviazione dei dati per ridurre la quantità di spazio necessaria per memorizzare i file.

Chapter 3

Implementazione

Il file contiene tutte le funzioni necessarie ad effettuare la compressione e la decompressione. È stato strutturato in modo tale da eseguire le funzioni dell'algoritmo in cascata, questo per poter consentire il facile inserimento di ulteriori algoritmi di compressione/decompressione. Di conseguenza, le varie tecniche di compressione e decompressione sono state strutturate sotto forma di funzioni. Tali funzioni verranno applicate su ogni riga del file preso in considerazione. L'implementazione degli algoritmi sopra citati è basata interamente su Python, questo perché grazie alla sua natura interattiva, dinamica e portatile consente lo sviluppo di applicazioni in modo semplice, in oltre può eseguire lo stesso codice su molteplici piattaforme.

Usare il paradigma object oriented ci ha dato la possibilità di strutturare il codice in maniera modulare. Gli sviluppatori possono utilizzare facilmente Python con altri linguaggi di programmazione noti come Java, C, e C++. Per quanto riguarda l'ambiente di sviluppo abbiamo usato PyCharm, questo perché è particolarmente utile per chi sviluppa in Python dato che integra diverse funzionalità utili nel development. Ad esempio, il completamento della sintassi delle istruzioni, l'help online, l'evidenziazione degli errori di sintassi durante lo sviluppo, la console di Python in una finestra dell'editor, ecc.

Il nostro algoritmo prende un file in input, lo legge riga per riga ed applica le varie funzioni di compressione o decompressione su ognuna di essa. La scelta

è ricaduta su questa strategia perché dopo alcune prove è risultato essere il metodo più efficiente. Il risultato della compressione viene memorizzato in un file di testo chiamato “compressed.txt”, mentre per la decompressione viene generato un nuovo file chiamato “decompressed.txt”. Infine, tutti i risultati intermedi generati durante il processo di compressione o decompressione verranno memorizzati in file temporanei che saranno ripresi dall' applicazione per generare l' output di compressione o decompressione.

Per eseguire l'algoritmo bisognerà:

1. Aprire il file main.py, il quale verrà trattato in seguito nel dettaglio;
2. Eseguire il file main.py;
3. Specificare se si vuole comprimere oppure decomprimere con l'uso delle lettere **c** oppure **d**
   (a) In caso di compressione specificare il nome del file senza estensione. Tale processo produrrà dei file utili per la decompressione.
   (b) In caso di decompressione si andrà a decomprimere l'ultimo file compresso.

L'output della compressione verrà salvato in un nuovo file chiamato compressed.txt, mentre l'output della decompressione verrà salvato nel file chiamato decompressed.txt.

3.1 Requisiti

Il progetto è stato sviluppato completamente in Python, in particolare usando la versione 3.7.9. Come interprete è stato utilizzato l'interprete di sistema, inoltre per avviare il progetto, se necessario, bisognerà installare la libreria **pickle**. Il modulo pickle implementa un basilare ma potente algoritmo, per serializzare e deserializzare una struttura di oggetti in Python. "Pickling" (NdT: serializzazione) è il processo mediante il quale una gerarchia di oggetti Python viene convertita in un flusso di byte, e "Unpickling" (NdT: deserializzazione) è l'operazione inversa, nella quale un flusso di byte viene riconvertito in una gerarchia di oggetti. Per fare ciò basterà aprire il terminale e digitare:

pip install pickle

```
1  import os
2  import time
```

Listing 3.1: Import delle librerie necessarie

Il **modulo OS** del linguaggio Python ha diverse funzioni utili per far interagire il programma con il sistema operativo del computer (Windows, Linux o Mac OS). Il **modulo time** definisce struct time per mantenere i valori di data e ora con i componenti separati in modo da essere facilmente accessibili. Nel nostro caso verrà utilizzato per analizzare i tempi di compressione e decompressione dei database.

3.2 Struttura del progetto

Il progetto è strutturato in un unico package contenente il main, i dataset ed il codice per generare i Dataset. Nel corso della descrizione del codice, assumeremo che il lettore possieda già basi del linguaggio di programmazione Python. Di seguito è riportata un'immagine per dare un'idea della struttura:

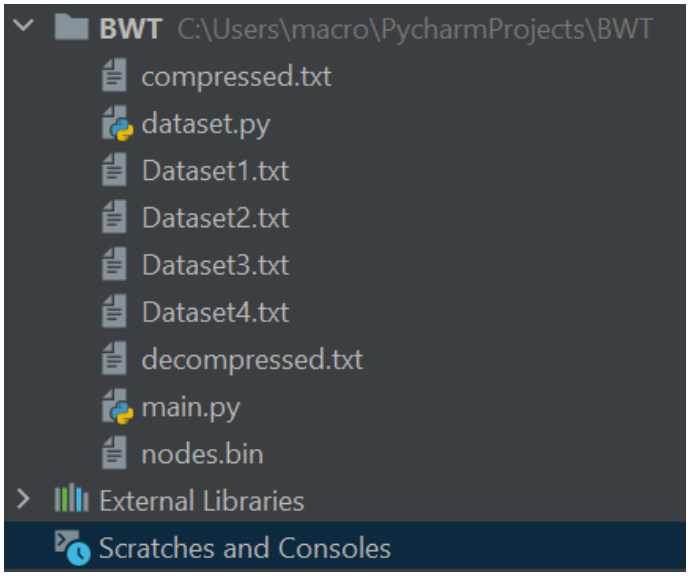


Figure 3.1: Struttura del progetto

Dopo aver parlato dei requisiti, di come funziona l'applicazione da utente e della struttura del progetto, andiamo a vedere nel dettaglio il codice. Partiamo con la descrizione del file main.py. N.B. Alcuni metodi non verranno esplicati poiché ritenuti di facile comprensione.

3.3 main.py

Il file si divide in due parti, nella prima parte viene creata una classe per la gestione di tutti e quattro gli algoritmi che usiamo. La classe che abbiamo chiamato Solution comprende 16 funzioni ed una sottoclasse "Nodes" che serve per la gestione degli alberi di Huffman.

```
# Funzione per applicare BWT
def BWT_Encode(self, s):
    # inizio BWT con parola presa da input
    table = [input_string[i:] + input_string[:i] for i in range(len(input_string))]
    # print('table = ', table)
    table = sorted(table)
    # print('sorted table = ', table)
    last_column = [row[-1:] for row in table]  # Last characters of each row

    bwt = ''.join(last_column)
    pos = bwt.find("$")

    return (bwt, pos)
```

Figure 3.2: BWT Encode

Questa funzione in input rende il contesto e una variabile di tipo stringa, genera poi una tabella che servirà per la rotazione delle stringhe, queste ultime vengono ordinate e per ogni riga della tabella viene eliminato l' ultimo carattere, infine il risultato viene salvato nella variabile bwt e viene salvata la posizione del "$" che servirà per la decompressione.

```
def iBWT(self, bwt):
    table = [""] * len(bwt)  # Make empty table

    for i in range(len(bwt)):
        table = [bwt[i] + table[i] for i in range(len(bwt))]  # Add a column of r
        # print('unsorted = ', table)
        table = sorted(table)
        # print('sorted    =', table)

    inverse_bwt = [row for row in table if row.endswith("$")][0]  # Find the correct row (ending in $)
    inverse_bwt = inverse_bwt.rstrip("$")  # Get rid of start and end markers

    return inverse_bwt
```

Figure 3.3: BWT Decode

La iBWT in input prende il contesto ed il dato da decomprimere e in output restituisce il dato decompresso. la funzione ha il compito di decomprimere la tabella in input, per prima cosa crea una nuova tabella vuota con la lunghezza uguale a quella dell'input "bwt". Dopo di che alla tabella vengono aggiunte le stringhe compresse e vengono riordinate. La parte cruciale è la ricerca della riga corretta che termina con il dollaro "&", quest'ultimo poi viene eliminato grazie alla funzione .rstrip("$").

```python
# Funzione per applicare LZW
def LZW_Encode(self, rle_output):
    """Compress a string to a list of output symbols."""

    # Build the dictionary.
    dict_size = 256
    dictionary = dict((chr(i), i) for i in range(dict_size))
    # in Python 3: dictionary = {chr(i): i for i in range(dict_size)}
    uncompressed = rle_output[0]
    w = ""
    result = []
    for c in uncompressed:
        wc = w + c
        if wc in dictionary:
            w = wc
        else:
            result.append(dictionary[w])
            # Add wc to the dictionary.
            dictionary[wc] = dict_size
            dict_size += 1
            w = c

    # Output the code for w.
    if w:
        result.append(dictionary[w])
    return result, rle_output[1]
```

Figure 3.4: LZW Encode

Nella funzione viene inizializzato un dizionario per contenere le stringhe a carattere singolo corrispondenti a tutti i possibili caratteri in input. L' algoritmo funziona eseguendo una scansione della stringa di input per sottostringhe successivamente più lunghe fin quando non e trova una che non è nel dizionario. Quando viene trovata una stringa che non è presente nel dizionario, l'indice per la stringa senza l' ultimo carattere (per esempio la sottostringa più lunga presente nel dizionario) viene recuperato dal dizionario e inviato all' output e la nuova stringa viene aggiunta al dizionario con il successivo codice disponibile. L' ultimo carattere di input viene quindi utilizzato come punto di partenza successivo per la ricerca di sottostringhe.

```
def iLZW_Decode(self, iHuffmanOutput):
    """Decompress a list of output ks to a string."""
    from io import StringIO

    # Build the dictionary.
    dict_size = 256
    dictionary = dict((i, chr(i)) for i in range(dict_size))
    # in Python 3: dictionary = {i: chr(i) for i in range(dict_size)}

    # use StringIO, otherwise this becomes O(N^2)
    # due to string concatenation in a loop
    result = StringIO()
    w = chr(iHuffmanOutput.pop(0))
    result.write(w)
    for k in iHuffmanOutput:
        if k in dictionary:
            entry = dictionary[k]
        elif k == dict_size:
            entry = w + w[0]
        else:
            raise ValueError('Bad compressed k: %s' % k)
        result.write(entry)

        # Add w+entry[0] to the dictionary.
        dictionary[dict_size] = w + entry[0]
        dict_size += 1

        w = entry
    return result.getvalue()
```

Figure 3.5: LZW Decode

L'algoritmo di decodifica funziona leggendo un valore dall'input codificato e producendo la stringa corrispondente dal dizionario. Tuttavia, il dizionario completo non è necessario, solo il dizionario iniziale che contiene stringhe a carattere singolo (e che di solito è codificato nel programma, invece di essere inviato con i dati codificati). Invece, il dizionario completo viene ricostruito durante il processo di decodifica nel modo seguente: dopo aver decodificato un valore e generato una stringa, il decodificatore lo concatena con il primo carattere della stringa decodificata successiva e aggiorna il dizionario con la nuova stringa. Il decodificatore passa quindi all'input successivo (già letto nell'iterazione precedente) e lo elabora come prima e così via fino a quando non ha esaurito il flusso di input.

```
# Funzione per applicare RLE
def RLE_Encode(self, bwt_output):
    input_list = bwt_output[0]
    # inizio RLE con passaggio da bwt
    res = ""
    tmp = input_list[0]
    count = 1
    for i in range(1, len(input_list)):
        if input_list[i] != tmp:
            res += str(count) + tmp
            tmp = input_list[i]
            count = 1
        else:
            count += 1
    return (res + str(count) + tmp), bwt_output[1]
```

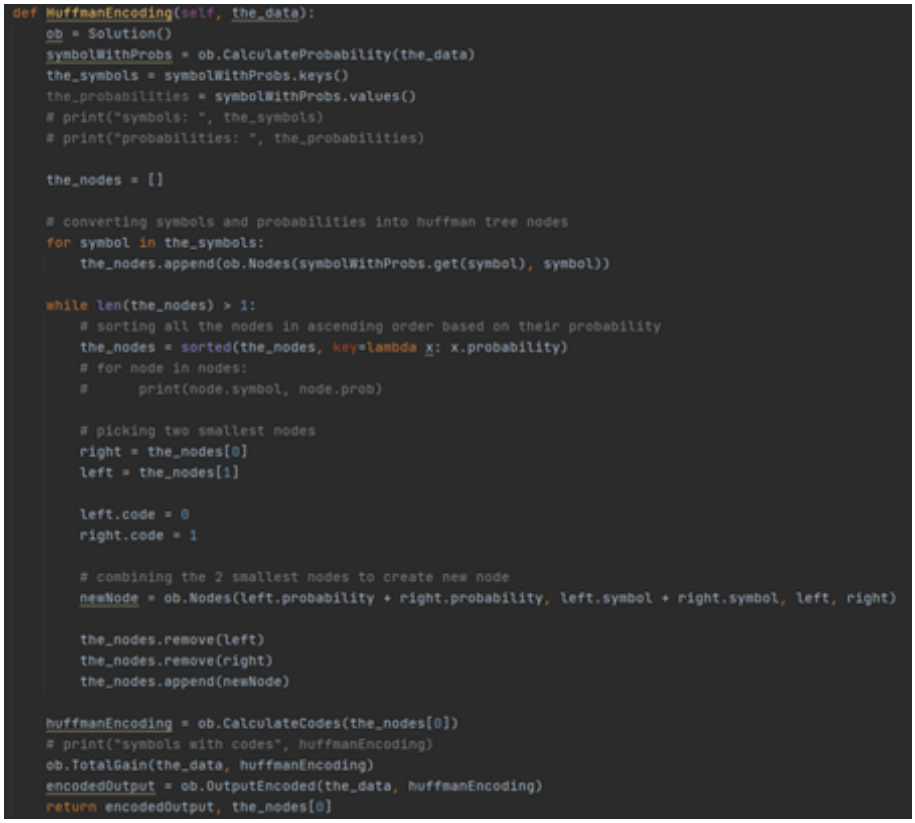
Figure 3.6: RLE Encode

Questo algoritmo prende in input il contesto e l'output della bwt che ha generato prima l' applicazione, dopo di che prende la stringa in input e in maniera iterativa aggiunge nella variabile res due valori consecutivi, count che rappresenta il numero di volte che un carattere viene ripetuto e tmp che è il carattere in questione. Dopo di che viene assegnato il nuovo carattere nella variabile tmp e posto ad 1 la variabile count, altrimenti incrementa il contatore di 1. Il ciclo termina dopo che è stata analizzata tutta la stringa in input e restituisce due valori, la concatenazione di res + il numero di cout + l'ultimo carattere di input_list, il secondo valore in output è bwt_output[1].

```
def iRLE(self, huffmanOutput):
    output = ""
    num = ""
    for i in huffmanOutput:
        if i.isalpha():
            output += i * int(num)
            num = ""
        else:
            num += i
    return output
```

Figure 3.7: RLE Decode

Questa funzione prende in input il contesto e una lista, inizializza due variabili di tipo stringa e itera per quanto è la lunghezza della lista e controlla se l'iesimo carattere è una lettera, se sì allora moltiplica la lettera per "num" e la aggiunge nell' output, altrimenti somma i a num e lo salva nella medesima variabile. Infine, genera in output la variabile di tipo stringa "output".



```
def HuffmanEncoding(self, the_data):
    ob = Solution()
    symbolWithProbs = ob.CalculateProbability(the_data)
    the_symbols = symbolWithProbs.keys()
    the_probabilities = symbolWithProbs.values()
    # print("symbols: ", the_symbols)
    # print("probabilities: ", the_probabilities)

    the_nodes = []

    # converting symbols and probabilities into huffman tree nodes
    for symbol in the_symbols:
        the_nodes.append(ob.Nodes(symbolWithProbs.get(symbol), symbol))

    while len(the_nodes) > 1:
        # sorting all the nodes in ascending order based on their probability
        the_nodes = sorted(the_nodes, key=lambda x: x.probability)
        # for node in nodes:
        #      print(node.symbol, node.prob)

        # picking two smallest nodes
        right = the_nodes[0]
        left = the_nodes[1]

        left.code = 0
        right.code = 1

        # combining the 2 smallest nodes to create new node
        newNode = ob.Nodes(left.probability + right.probability, left.symbol + right.symbol, left, right)

        the_nodes.remove(left)
        the_nodes.remove(right)
        the_nodes.append(newNode)

    huffmanEncoding = ob.CalculateCodes(the_nodes[0])
    # print("symbols with codes", huffmanEncoding)
    ob.TotalGain(the_data, huffmanEncoding)
    encodedOutput = ob.OutputEncoded(the_data, huffmanEncoding)
    return encodedOutput, the_nodes[0]
```

Figure 3.8: Huffman Encode

Una codifica Huffman può essere calcolata creando prima un albero di nodi:

1. Creare un nodo foglia per ogni simbolo e aggiungerlo alla coda di priorità.
2. Mentre c'è più di un nodo nella coda:
   (a) Rimuovere il nodo con priorità più alta (probabilità più bassa) due volte per ottenere due nodi.
   (b) Creare un nuovo nodo interno con questi due nodi come figli e con probabilità uguale alla somma delle probabilità dei due nodi.
   (c) Aggiungere il nuovo nodo nella coda
3. Il nodo rimanente è il nodo radice e l'albero è completo

NB: questa funzione utilizza anche una classe creata ad-hoc che è la classe Nodes per la gestione dei nodi dell' albero.

```python
class Nodes:
    def __init__(self, probability, symbol, left=None, right=None):
        # probability of the symbol
        self.probability = probability

        # the symbol
        self.symbol = symbol

        # the left node
        self.left = left

        # the right node
        self.right = right

        # the tree direction (0 or 1)
        self.code = ''
```

Figure 3.9: Classe che descrive un nodo

```python
def HuffmanDecoding(self, encodedData, huffmanTree):
    treeHead = huffmanTree
    decodedOutput = []
    for x in encodedData:
        if x == '1':
            huffmanTree = huffmanTree.right
        elif x == '0':
            huffmanTree = huffmanTree.left
        try:
            if huffmanTree.left.symbol == None and huffmanTree.right.symbol == None:
                pass
        except AttributeError:
            decodedOutput.append(huffmanTree.symbol)
            huffmanTree = treeHead

    # string = ''.join([str(item) for item in decodedOutput])
    return decodedOutput
```

Figure 3.10: Huffman Decode

La funzione di decodifica di Huffman prende in input il contesto, il dato codificato e l' albero di Huffman, per prima cosa inizializza treeHead e decodeOutput che servirà per il risultato, poi visita tutto l' albero fino ad arrivare alle foglie dove sono salvati i simboli. Una volta arrivati ai nodi foglia salva i dati contenuti nella lista decodeOutput e aggiorna il vecchio albero con il nuovo. Infine, ritorna la variabile decodedOutput che contiene il risultato della decodifica.
Queste 4 funzioni sono delle funzioni di supporto create appositamente per la codifica di Huffman:

1. def CalculateProbability
2. def CalculateCodes
3. def OutputEncoded
4. def TotalGain

```
def CalculateProbability(self, the_data):
    the_symbols = dict()
    for item in the_data:
        if the_symbols.get(item) == None:
            the_symbols[item] = 1
        else:
            the_symbols[item] += 1
    return the_symbols
```

Figure 3.11: Calculate Probability

Questa funzione è una funzione di supporto che viene utilizzata per calcolare le probabilità dei simboli in un dato specifico (quello passato in input). La suddetta funzione viene usata nella funzione HuffmanEncoding.

```
def CalculateCodes(self, node, value=''):
    ob = Solution()
    # a huffman code for current node
    newValue = value + str(node.code)

    if (node.left):
        ob.CalculateCodes(node.left, newValue)
    if (node.right):
        ob.CalculateCodes(node.right, newValue)

    if (not node.left and not node.right):
        ob.the_codes[node.symbol] = newValue

    return ob.the_codes
```

Figure 3.12: Calculate Codes

Questa funzione di supporto per stampare i codici dei simboli percorrendo l'albero di Huffman.

```
def OutputEncoded(self, the_data, coding):
    encodingOutput = []
    for element in the_data:
        # print(coding[element], end = '')
        encodingOutput.append(coding[element])

    the_string = ''.join([str(item) for item in encodingOutput])
    return the_string
```

Figure 3.13: Output Encoded

```
def TotalGain(self, the_data, coding):
    # total bit space to store the data before compression
    beforeCompression = len(the_data) * 8
    afterCompression = 0
    the_symbols = coding.keys()
    for symbol in the_symbols:
        the_count = the_data.count(symbol)
        # calculating how many bit is required for that symbol in total
        afterCompression += the_count * len(coding[symbol])
```

Figure 3.14: Total Gain

Una funzione di supporto per calcolare la differenza di spazio tra dati compressi e non compressi

```
def checkAlphanumeric(self, s):
    if s.isalnum():
        if any(char.isdigit() for char in s):
            return False
        else:
            return True
    else:
        return False
```

Figure 3.15: Check Alphanumeric

Funzione che prende in input una stringa, cicla sulla stringa per esaminare ogni carattere e controlla se ogni carattere è una lettera. Se trova un numero, lo rimuove e restituisce la stringa pulita.

```
def deleteSpecialCharacters(self, s):
    for char in s:
        if not char.isalpha():
            s = s.replace(str(char), "")
    return s
```

Figure 3.16: Delete Special Characters

Funzione che viene usata per eliminare i caratteri speciali in una stringa, questa vene usata nella fase di compressione per il controllo alfanumerico.

```
def saveToBin(self, the_tree):
    import pickle

    with open("nodes.bin", "wb") as f:
        pickle.dump(the_tree, f)
```

Figure 3.17: Save To Bin

Funzione di supporto che prende in input una struttura ad albero e tramite la libreria pickle che gestisce gli stream I/O crea un file "nodes.bin" in modalità lettura e scrittura e salva la struttura dati sul file.

```
def readFromBin(self):
    import pickle

    with open("nodes.bin", "rb") as f:
        return pickle.load(f)
```

Figure 3.18: Read From Bin

Funzione di supporto che viene usata per leggere dal file "nodes.bin" la struttura dati salvata. Viene invocata nella fase di compressione per salvare i nodi dell'albero di Huffman.

Nella seconda parte del codice verrà chiesto all'utente se desidera comprimere oppure decomprimere, l'utente dovrà rispondere inserendo c/d. In caso di risposta non valida, verrà rifatta all'utente la stessa domanda

- In caso di compressione, verrà chiesto all'utente di specificare quale file desidera comprimere (senza inserire l'estensione in quanto è stata preimpostata su .txt).
- In caso di decompressione verrà decompresso l'ultimo file che è stato compresso.

Per quanto riguarda la compressione, il codice è stato strutturato nel seguente modo:

1. Si elimina il carattere di fine riga dal file specificato ed eventuali righe vuote;

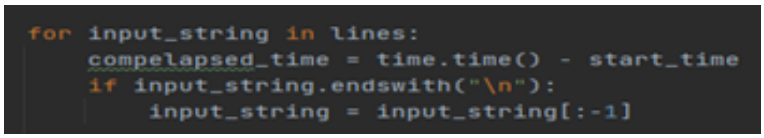


Figure 3.19: Eliminazione carattere di fine riga e righe vuote

2. Per ogni riga del file specificato si prenderà la stringa di input e si elimineranno eventuali caratteri speciali, in quanto provocheranno errore nella compressione;

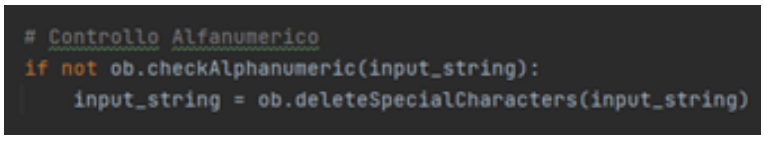


Figure 3.20: Eliminazione caratteri speciali

3. Per ogni riga del file applico in cascata le funzioni:

   (a) BWT: esegue la funzione BWT sulla stringa di input del file originale, dopodiché da tale stringa verrà rimosso il valore $ (di cui viene salvata la posizione) per poter eseguire le funzioni successive senza errori.

```
bwt_result = ob.BWT_Encode(input_string)
print("BWT: ", bwt_result[0])
no_dollar_string = bwt_result[0].replace("$", "")
tmp = list(bwt_result)
tmp[0] = no_dollar_string
bwt_result = tuple(tmp)
print("BWT No-Dollar: ", no_dollar_string)

if not no_dollar_string:
    continue
```

Figure 3.21: Applicazione BWT

   (b) RLE: prende come input la stringa di output della BWT senza il valore $ ed applica la funzione RLE;

```
# Chiamata funzione RLE
rle_result = ob.RLE_Encode(bwt_result)
print("RLE: ", rle_result)
```

Figure 3.22: Applicazione RLE

   (c) LZW: prende come input la stringa di output della RLE ed applica la funzione LZW;

```
# Chiamata funzione LZW
lzw_result = ob.LZW_Encode(rle_result)
print("LZW: ", lzw_result)
```

Figure 3.23: Applicazione LZW

(d) Huffman: prende come input la stringa di output della LZW ed applica la funzione Huffman. Tale funzione darà in output due file: un file .txt compresso, contenente su ogni riga la stringa compressa e posizione del dollaro e un file .pickle contenente le informazioni sull'albero per la codifica di Huffman;

```
# Chiamata funzione Huffman
the_data = lzw_result[0]
encoding, the_tree = ob.HuffmanEncoding(lzw_result[0])

nodi.append(the_tree)
```

Figure 3.24: Applicazione Huffman

4. Infine viene creato il file "compressed.txt" dove verrà salvato il risultato, si salvano i nodi nel file temporaneo "nodes.bin" e si stampa il tempo di compressione.

```
    with open('compressed.txt', "a+", ) as file_object:
        file_object.seek(0)
        data = file_object.read(100)
        if len(data) > 0:
            file_object.write("\n")
        file_object.write(output_string)

ob.saveToBin(nodi)

print("\n" + str(compelapsed_time) + " -> Compression elapsedTime")
break
```

Figure 3.25: Creazione file compressed.txt e salvataggio nodi in bin

Per quanto riguarda la decompressione verranno eseguiti i medesimi passaggi ma in modo inverso.

Chapter 4

Risultati

4.1 Dataset

I Dataset utilizzati per i nostri test sono stati generati manualmente attraverso un apposito codice Python che permette di generare stringhe di genoma composte da 24 caratteri, su ogni riga del file viene inserita una stringa di genoma univoca, di conseguenza, i dataset generati non presentano ridondanza di stringhe. Il numero di righe è stato scelto appositamente in modo incrementale per poter effettuare diversi test.

I nomi dei Dataset utilizzati sono riportati nella tabella seguente:

Nome Dataset	**Numero di righe**
Dataset1.txt	3000
Dataset2.txt	5000
Dataset3.txt	8000
Dataset4.txt	10000

4.2 Ambiente d'esecuzione

Di seguito sono riportate le caratteristiche principali del PC su cui sono stati effettuati i test:

- S.O.Windows 11 Pro for Workstations
- Sistema operativo a 64 bit, processore basato su x64
- RAM 32 GB
- CPU 11th Gen Intel(R) Core(TM) i7-1165G7 2.80 GHz
- SSD 500 GB

4.3 Risultati

I test sono stati effettuati con la macchina riportata in 5.2. Per ogni dataset utilizzato abbiamo testato il nostro algoritmo tramite il file main.py descritto nel capitolo di implementazione, ricavando così i tempi impiegati dal nostro algoritmo su ciascun dataset. Per ogni dataset abbiamo testato l'algoritmo per 4 volte in modo da avere una stima più accurata dei tempi. Nella seguente tabella sono stai riportati i tempi di esecuzione espressi in secondi.

Nome Dataset	**Tempo compressione**	**Tempo decompressione**
Dataset1.txt	32,80	31,82
Dataset2.txt	60,30	52,42
Dataset3.txt	71,70	91,93
Dataset4.txt	75,83	112,06

Per una maggiore chiarezza abbiamo rappresentato attraverso dei grafici i risultati ottenuti. Come è possibile vedere dai grafici sottostanti, per file con poche righe, i tempi di compressione e decompressione sono pressochè simili. All'aumentare del numero di righe contenute nel file, i tempi di decompressione diventano maggiori dei tempi di compressione.

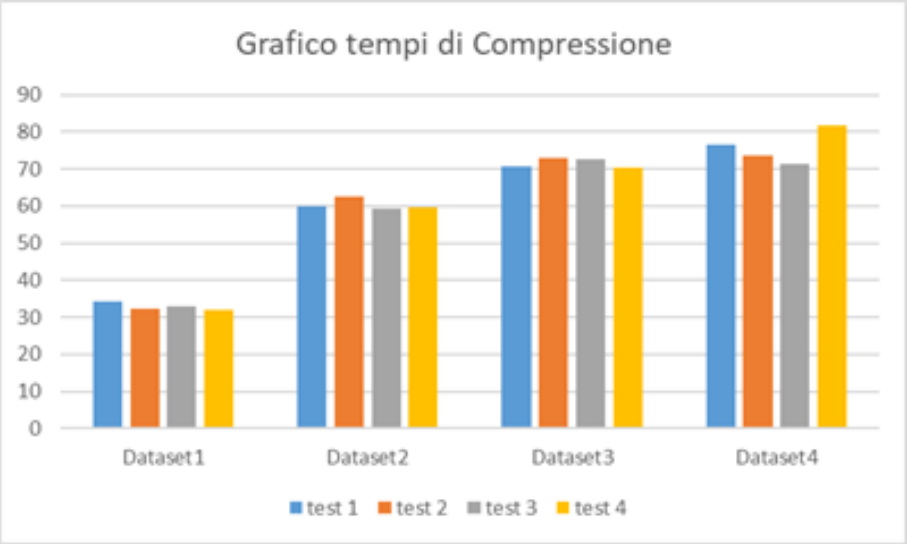


Figure 4.1: Tempi di compressione

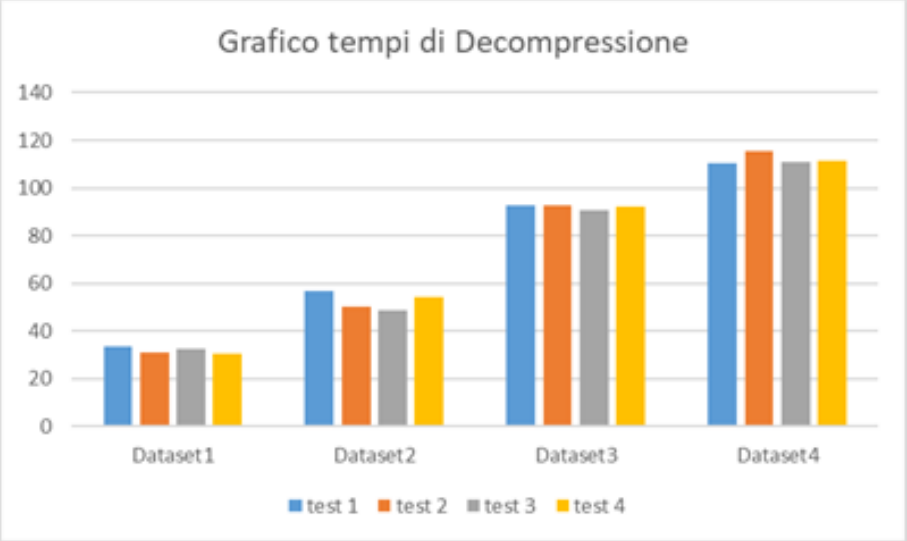


Figure 4.2: Tempi di decompressione

Chapter 5

Conclusioni

Come descritto nel capitolo 4, il lavoro finale si basa sull'utilizzo di quattro trasformate: **BWT**, **RLE**, **LZW**, **Huffman**. Infine possiamo a affermare che il nostro algoritmo non è del tutto lossless in quanto, a seguito della compressione si vanno a perdere alcune informazioni presenti nel file originale:

- Righe vuote
- Caratteri speciali
- Caratteri numerici
- Spazi tra parole

Pertanto, alcuni dei possibili sviluppi futuri sarebbero:

- ottimizzare la fase di compressione evitando la perdita di tali informazioni;
- applicare ulteriori algoritmi di compressione/decompressione al fine di aumentare il livello di sicurezza della compressione;
- migliorare la compressione del file per quanto concerne la size del file;

E' possibile visionare il progetto assieme alla sua documentazione su GitHub al link: https://github.com/antosap/ProgettoCD