OsDNA

Ferdinando D'Avino, Lino Sarto, Sergiy Shevchenko

March 2019

1 Introduzione

Lo scopo del nostro progetto è stato la progettazione e lo sviluppo di un algoritmo di compressione dati, dedicato a input monodimensionali, nello specifico file di testo che codificano tramite caratteri le sequenza di nucleotidi che compongono il DNA.

2 Cenni sul dominio applicativo

Il **DNA** o **acido desossiribonucleico** è un acido nucleico contenente tutte le informazioni genetiche di un organismo vivente. La sua famosa forma a doppia elica incrociata è "tenuta insieme" da coppie di **basi azotate** che si legano solo in specifiche combinazioni.

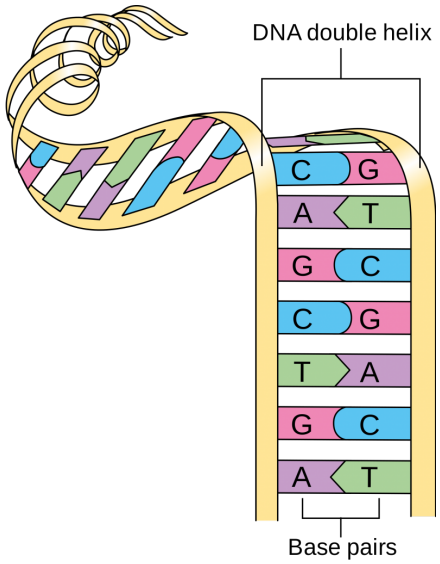

Figure 1: The DNA

Nello specifico le basi azotate che compongono il DNA sono 4:

- Adenina (A)
- Citosina (C)
- Guanina (G)
- Timina (T)

Ognuna di queste può essere legata unicamente ad una singola altra base azotata, infatti la **G**uanina può legarsi solo con la **C**itosina (e viceversa) mentre l' **A**denina solo con la **T**imina (e viceversa). Da ciò appare chiaro che per memorizzare una sequenza di DNA è necessario codificare solo una delle due sequenze di basi azotate, in quanto la seconda può essere facilmente calcolata per complementarità. Nonostante questo enorme risparmio in termini di elementi da memorizzare, file che codificano intere sequenze di DNA arrivano comunque a pesare svariati GigaByte rendendo poco agevole ogni tipo di operazione con questi ultimi, dall'invio al semplice storing.

3 Idea Generale

L'idea alla base dell'algoritmo di compressione OsDNA sfrutta la grossa taglia di questo genere di file combinata al fatto che l'alfabeto che li compone sia sempre di soli 4 caratteri (A, C, G, T). E' abbastanza intuitivo il fatto che pochi caratteri ripetuti una moltitudine di volte presenteranno molto probabilmente sequenze di caratteri uguali più o meno lunghe; sostituendo opportunamente parte di queste con un numero intero che indica quante volte quel carattere si ripete, non sarà più necessario memorizzare per esteso la sequenza. Inoltre un enorme risparmio è dato dalla codifica dei 4 caratteri possibili che utilizzerà soli 2 bit per carattere:

- A : 00
- C : 01
- G : 10
- T : 11

E' per questo motivo che il nostro algoritmo considera solo i caratteri del file corrispondenti a questo piccolo alfabeto semplicemente scartando gli altri. Ciò potrebbe essere un problema nel momento in cui si fornisce come input un file nel formato *fasta*, il suddetto formato infatti presenta altri caratteri atti a rappresentare con un singolo simbolo coppie, triple o quadruple di basi azotate. Per i nostri test, vista la natura didattica di OsDNA, i file utilizzati, quando in formato fasta, prima di essere passati come input venivano puliti dai caratteri "sporchi" tramite degli script esterni.

3.1 Trigger Size

Nonostante l'idea risulti abbastanza semplice la progettazione ha dovuto far fronte a qualche scoglio; la codifica utilizzata per le basi azotate infatti utilizza tutte le possibili combinazioni di due bit rendendo impossibile distinguere da essi qualunque altra informazione codificata nel file che potrebbe sempre essere "tradotta" come una qualche sequenza di basi. Questa situazione ha reso impossibile l'inserimento di un simbolo speciale atto a notificare nello stream la presenza di un numero intero invece che una base azotata. La soluzione adottata quindi è stata l'introduzione di un parametro battezzato **Trigger Size**, un numero che indicherà dopo quanti caratteri uguali uno di seguito all'altro sarà presente l'intero che ci informa su quanti altri caratteri uguali erano presenti nel file originale.
Fissato un numero di bit per rappresentare i valori interi, valore battezzato **Bit Size**, abbiamo la possibilità di rapprensentare un certo range di interi, ad esempio utilizzando 3 bit potremo rappresentare tutti i valori da 0 a 7. Siccome non si può mai essere certi di quale sia il valore più grande che dovremo codificare, la Bit Size si mantiene su un valore scelto tramite una metrica che illustreremo successivamente e si prevede una tecnica per la gestione della situazione di overflow. Semplicemente una volta che viene riconosciuto nello stream un valore intero, se ne leggeranno altri finchè il valore letto sarà il massimo rappresentabile con quel numero di bit.
L'utilizzo di questa metodologia però presenta anche dei contro, se da un lato infatti abbiamo la possibilità di riconoscere un intero durante la lettura di uno stream, altrimenti indistinguibile, dall'altro dobbiamo accettare un caso svantaggioso in termini di verbosità della codifica. Ogni volta in cui ci troviamo a leggere una sequenza di caratteri uguali lunga esattamente il valore **Trigger Size** dobbiamo comunque inserire un numero intero che ci dice quanti ancora ne seguono, anche se non ne sono presenti altri (indicheremo ciò con lo 0). Questa situazione non viene risolta ma il danno viene mitigato tramite un' attenta scelta della **Trigger Size** e della **Bit Size**.

3.2 Pseudocodice

```
function compressCore
    optParam ← optParamCalc
    writeHeader(optParam)
    currChar ← readFromFile(1)
    lastChar ← currChar
    lastOccLen ← 1
    while (readBuff = readFromFile(1024)) ≠ NULL do
        for i < readBuff.size() do
            currChar ← readBuff[i]
            if lastChar = currChar then
                lastOccLen + 1
            else updateBuffer(optParam, currChar)
                lastOccLen ← 1
            end if
            lastChar ← currChar
        end for
      resetBuffer
    end while
     bufferFlush(optParam, currChar)
end function
```

4 Trigger Size, Bit Size: Metrica

Nella sezione 3.1 si è anticipata l'idea di utilizzo dei parametri Trigger Size e Bit Size. In questa sezione si illustrerà nel dettaglio la scelta della metrica utilizzata. Di fatto attraverso un'analisi di "brute force" è possibile individuare i parametri Trigger Size e Bit Size *ottimi*.
A tal fine viene eseguito l'algoritmo *optParamCalc* che scorre il file di input interamente una singola volta prima di effettuare la compressione. Tale algoritmo valuta quanti bit si andrebbero a risparmiare, per qualche valore di Trigger Size e di Bit Size, relativamente a quelli necessari per la rappresentazione con la sola codifica che utilizza due bit per carattere.
Un miglioramento immediato è quello di utilizzare valori diversi di Trigger Size e Bit Size per ogni carattere del nostro alfabeto {A, C, G, T}. Quindi *optParamCalc* confronta il guadagno sulla codifica a due bit per qualche valore di Trigger Size e di Bit Size per ogni carattere dell'alfabeto. Con più precisione i valori di Trigger Size e Bit Size su cui si effettua il confronto sono tutti i valori che potrebbero "ragionevolmente" portare un vantaggio nella compressione rispetto al file di input.

4.1 Trigger Size e Bit Size: dettagli implementativi

Come si è già discusso individuare i valori ottimi di Trigger Size e Bit Size per ogni carattere migliora il *compression ratio* del nostro algoritmo. Al fine di

rendere più chiara tale idea si passerà ad illustrare qualche dettaglio implementativo partendo dalle strutture dati utilizzate. Le ripetizioni di ogni carattere sono state mantenute in un array a due dimensioni *occMatr[4][MaxValue]*. Nella prima dimensione si fa riferimento ai singoli caratteri dell'alfabeto, mentre la seconda indica il numero di occorrenze per ogni carattere. Un'altra struttura rilevante che utilizza l'algoritmo è *bitAdvantage[bitEncode][numMatch]*. L'obiettivo di quest'ultima è quello di memorizzare quale sarebbe il vantaggio in numero di bit, utilizzando $bitEncode \in \{2, ..., BitEncodeSize\}$ bit per la codifica delle ripetizioni di un carattere e $numMatch \in \{0, ..., MaxValue\}$ occorrenze consecutive di un carattere. Da quanto detto la prima dimensione indica il numero di bit utilizzati per codificare le ripetizioni di un carattere mentre la seconda dimensione indica le occorrenze consecutive di un carattere. Nella Figura 2 è rappresentata la matrice *bitAdvantage* con alcuni dei suoi valori. La prima

Bit Encode	0 Match	1 Match	2 Match	3 Match	4 Match	5 Match	6 Match	7 Match	...
2	-2	0	2	2	4	6	6	8	...
3	-3	-1	1	3	5	7	9	8	...
4	-4	-2	0	2	4	6	8	10	...
5	-5	-3	-1	1	3	5	7	9	...
6	-6	-4	-2	0	2	4	6	8	...
7	-7	-5	-3	-1	1	3	5	7	...
8	-8	-6	-4	-2	0	2	4	6	...
9	-9	-7	-5	-3	-1	1	3	5	...

Figure 2: Bit Advantage

colonna indica il numero di bit scelto per la codifica del numero di occorrenze di un carattere, mentre le colonne successive indicano quanti bit si risparmierebbero ogni volta che sono presenti un certo numero di ripetizioni consecutive di un carattere. Si noti che per far si che la decompressione funzioni correttamente è necessario che l'algoritmo di compressione specifichi la fine di una sequenza ripetuta di un carattere.
Ad esempio consideriamo la sequenza $AAACCCCCCCCA$. Supponiamo si utilizzino 3 bit per codificare il numero di occorrenze ripetute di A e 2 bit per codificare le ripetizioni di C. In questo caso la codifica risultante sarà 00 010 01 11 11 00 00 000. I primi due bit codificano il carattere A, i successivi 3 bit codificano quante ripetizioni di A sono presenti, in questo caso due. Successivamente segue il carattere C codificato con 01 e poi la sequenza 11 11 11 che ci indica il numero di ripetizioni di C codificata a blocchi di due bit. Ora è necessario che si utilizzino ulteriori 2 bit settati a 0 per indicare la fine delle ripetizioni del carattere C. Successivamente troviamo 00 che è la codifica di A seguito da 000 che indica il numero di ripetizioni di A, in questo caso nessuna. Confrontiamo quindi il risultato ottenuto con la semplice codifica a due bit, con A: 00 e C:01; il risultato ottenuto dalla codifica a due bit è 00 00 00 11 11 11 11 11 11 11 00. Tale tecnica ci porta in questo caso a risparmiare 6 bit rispetto alla codifica a due bit. Tale risultato può essere facilmente calcolato dall'algoritmo

optParamCalc utilizzando la matrice Bit Advantage. Nell'esempio l'algoritmo trova due match che seguono il carattere A. Poiché utilizziamo 3 bit per indicare le occorrenze ripetute di A abbiamo 1 bit di vantaggio rispetto alla codifica a due bit. Successivamente l'algoritmo incontra il carattere C seguito da 6 match. Utilizzando 2 bit per le ripetizioni di C abbiamo un vantaggio di 6 bit rispetto alla codifica a due bit. Infine il carattere A non ha ripetizioni, quindi avremo una perdita di 3 bit. Facendo la somma 1 + 6 + 2 - 3 otteniamo il risultato atteso.
La struttra dati *triggerSizeAdvantage[4][BitEncodeSize][MaxValue]* ha l'obiettivo di memorizzare il vantaggio di ogni codifica utilizzata al variare di *bitEncode* per qualche ripetizione di un carattere al variare di *numMatch*.

4.2 Statistiche e Grafi

In questa sezione riportiamo i risultati delle sperimentazioni, confrontando le prestazioni di osdna con altri 6 algoritmi di compressione e con osdna stesso, sfruttando però diversi tipi di *pre-processing* e *post-processing*.

Ognuno di questi test è stato eseguito su una macchina con il presente hardware: **Intel Xeon Processor (Skylake, IBRS) 2.10 GHz, 8Gb Ubuntu 18.04.2 LTS**.

Per i test abbiamo utilizzato un dataset composto di 3 file

- Lambda Virus (0,046 Mb)
- Homo_sapiens.GRCh38.dna (2930,366 Mb)
- SRR741411_2 (7613,130 Mb)
- GCF_000223135.1_CriGri_1.0_genomic Cricetulus griseus (2212 Mb)
- GCF_000005005.2_B73_RefGen_v4_genomic Zea Mays (2006 Mb)
- GCF_000001405.38_GRCh38.p12_genomic Homo Sapiens (2952 Mb)
- GCA_000404065.3_Ptaeda2.0_genomic Pinus taeda (19595 Mb)

I grafici che seguono mostrano le prestazioni medie dei diversi algoritmi sotto diversi punti di vista, nello specifico misureremo la velocità di compressione e decompressione ed il compression ratio di ognuno. Inoltre indicheremo con un grafico anche la performance generale di ogni algoritmo sfruttando una metrica che mette in relazione velocità ed il rapporto di compressione.

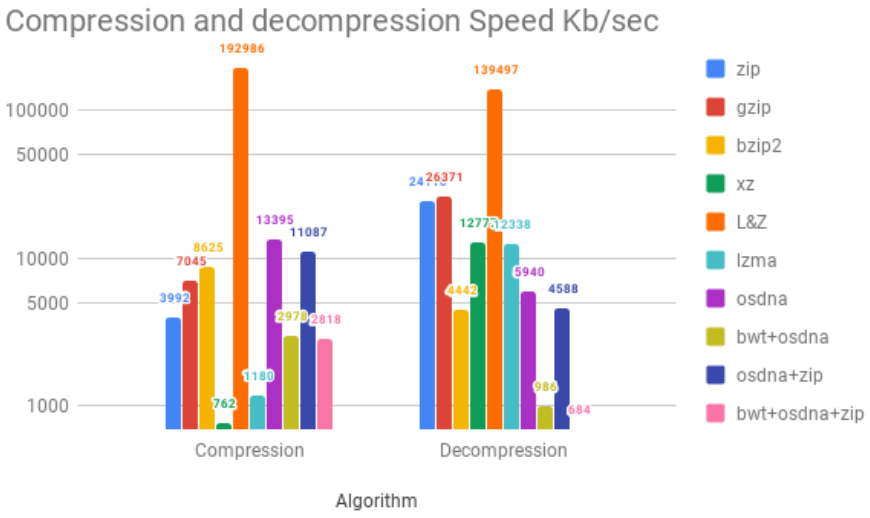

Figure 3: Compression and Decompression speed

ratio.png

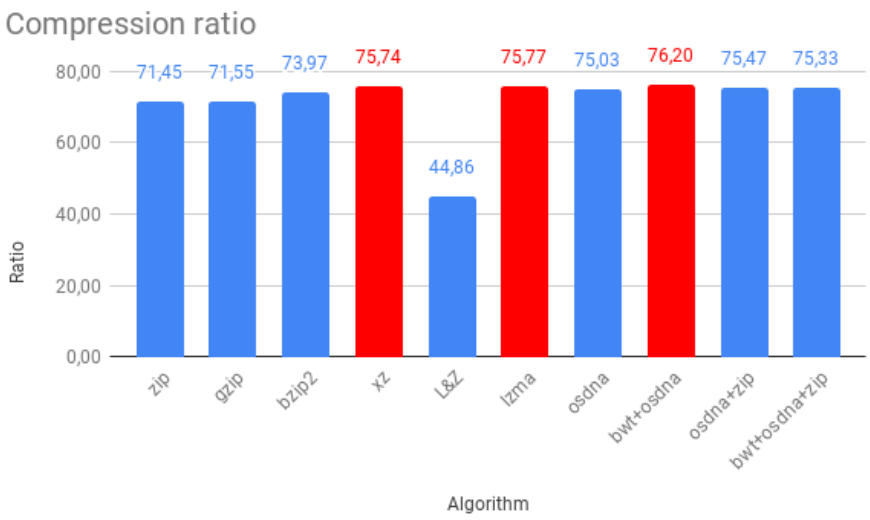

Figure 4: Compression ratio

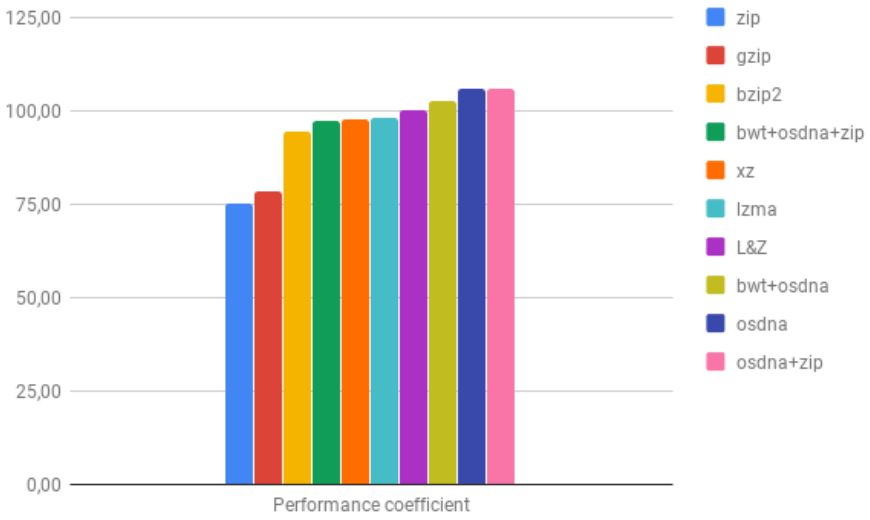

Figure 5: Performance coefficient

5 Conclusioni e sviluppi futuri

I risultati delle sperimentazioni mostrano che gli algoritmi con il miglior tasso di compressione (medio relativamente ai 3 file) sono **xz**, **lzma**, **bwt + osdna**. Nonostante l'overhead che comporta il preprocessing, **bwt + osdna** risulta essere oltre che il più performante in termini di compressione anche il più rapido (**bwt+osdna : 2,91 Mbs**). E' da notare infatti che **xz** ed **lzma** pagano gli ottimi risultati di compressione con un tempo di esecuzione che si potrebbe definire proibitivo (**xz : 0,74 Mbs**, **lzma : 1,15 Mbs**). La situazione è differente per quanto riguarda i tempi di decompressione, infatti gli algoritmi che decomprimono più rapidamente sono **L&Z**, **gzip**, **zip**; è facile notare che questi ultimi sono tutti algoritmi a dizionario, che per loro natura effettuano una decompressione molto rapida ma che d'altro canto occupano le ultime posizioni nella classifica del compression ratio.

Da queste osservazioni possiamo concludere che il postprocessing con zip non ci dà un miglioramento tale in termini di compressione per far si che valga la pena accettare l'overhead in tempo di esecuzione. Un caso differente si verifica per il preprocessing con bwt che migliora il nostro algoritmo al punto da farlo competere con gli altri algoritmi più performanti testati. Da ciò è immediato pensare ad un primo sviluppo futuro, ovvero, l'integrazione della bwt in osdna, in modo da non dover essere dipendenti da librerie esterne ed utilizzare una codifica interna per la rappresentazione della posizione dei caratteri fine stringa (\$). Un ulteriore miglioramento potrebbe essere portato dal calcolare la bwt partendo dal file codificato con due bit per carattere, in modo da ridurre la dimensione dell'input ed accellerare il preprocessing. Un'ultima feature utile la si potrebbe ottenere con l'estensione dell'header per comprendere anche i nomi dei diversi geni, che nel formato *fasta* vengono indicati all'inizio del file.

Concludendo osdna risulta essere l'algoritmo con il miglior trade-off tra compression ratio e compression speed.