Università degli Studi del Piemonte Orientale
"Amedeo Avogadro"

Dipartimento di Scienze e Innovazione Tecnologica
Corso di Laurea Magistrale in Informatica

TESI DI LAUREA

ALGORITMI SEMI-EXTERNAL PER LA COSTRUZIONE DI SUFFIX ARRAY E BURROWS-WHEELER TRANSFORM DI GRANDI COLLEZIONI DI DOCUMENTI E BIOSEQUENZE.

Candidato
Matteo Marescotti

Relatore
Prof. Giovanni Manzini

Anno accademico 2013 / 2014

1 Introduzione 4
1.1 Notazione e concetti base 6
1.2 Suffix tree 7
1.3 Suffix Array 8
1.4 Burrows-Wheeler Transform 10
2 Algoritmi di costruzione 16
2.1 BWTE 17
2.2 Ottimizzazioni in BWTE 20
3 Collezioni di stringhe 28
3.1 Algoritmi di costruzione per collezioni 29
3.2 BWTC e BWTCW 29
4 Strutture dati per rank 34
4.1 Wavelet Tree 34
4.2 BWT32 36
5 Test 38
6 Conclusioni 44
Bibliografia 45

Abstract. La Burrows-Wheeler Transform (BWT), originariamente inventata come parte di un algoritmo di compressione dei dati di tipo *lossless* (Burrows & Wheeler, 1994), è attualmente molto utilizzata come componente principale per la costruzione di indici compressi. Tuttavia lo spazio richiesto per sua costruzione è sovente un impedimento nelle applicazioni dove si devono trattare collezioni di sequenze di decine di GB. In letteratura l'unico strumento disponibile per il calcolo della BWT di collezioni di documenti è BEETL_BWT presentato in (Bauer, et al., 2011). Tale strumento è però dedicato alle sole collezioni ottenute da sequenziamento di DNA e di conseguenza lavora solo con documenti aventi tutti la stessa lunghezza e con un alfabeto di 5 caratteri. In questa tesi viene proposto un nuovo algoritmo per ottenere la BWT di grosse collezioni di documenti che non ha queste limitazioni. Il nuovo algoritmo è di tipo semi-external nel senso che non richiede una quantità di memoria proporzionale alla dimensione dell'input, utilizzando anche la memoria esterna mediante accessi sequenziali.

1 INTRODUZIONE

L'ammontare di informazioni disponibili di varia natura sta aumentando esponenzialmente grazie alla continua crescita del web e alla elevata velocità di elaborazione e trasmissione che offrono molti sistemi informatici. La maggior parte di queste informazioni consistono di testo inteso come una stringa di caratteri di qualunque lunghezza, che rappresenta non solamente linguaggio naturale, ma anche programmi, set di dati, risultati di sequenziamento biologico, sequenze biologiche e così via.

In ognuno di questi scenari si necessita di un metodo per ottenere l'informazione dal testo al fine di poterne usufruire. Una operazione è alla base di tutti questi metodi: la ricerca di sottostringhe.

La ricerca di sottostringhe (string matching) è il processo di ricerca di occorrenze di una stringa più corta, detta pattern, all'interno di un testo solitamente molto più grande. Qualunque applicazione di elaborazione dei testi sfrutta internamente un meccanismo basilare di string matching per implementare funzionalità più sofisticate come conteggio delle parole più frequenti o ricerca di sequenze similari ad un campione all'interno di un DNA. Gli sviluppi in questo campo quindi hanno un impatto positivo notevole in moltissime applicazioni.

Ci sono due modalità per affrontare la ricerca di sottostringhe: sequenziale e indicizzata. Lo string matching sequenziale non richiede alcun *preprocessing* del testo, si scorre il testo carattere per carattere alla ricerca di ciascuna occorrenza del pattern. Lo string matching indicizzato invece prevede una parte di *preprocessing* del testo in cui si costruisce una particolare struttura dati, l'indice. Tale struttura permette in maniera più veloce di trovare tutte le occorrenze di un pattern all'interno del testo senza scorrere tutto il testo.

L'indicizzazione è la scelta più conveniente quando il testo è molto grande, ha cambiamenti molto rari e le ricerche saranno frequenti. In questo caso è preferibile spendere tempo per la creazione di un indice al fine di favorire tutte le successive ricerche. Una condizione necessaria però è possedere lo spazio sufficiente alla memorizzazione dell'indice e provvederne un accesso efficiente.

Il problema dello spazio in memoria di massa non sembra molto significante infatti è possibile usufruire di enormi capacità ad un costo contenuto. Il problema reale è l'accesso efficiente alla memoria di massa, storicamente l'operazione che più lentamente è riuscita ad evolversi. Solo negli ultimi anni, grazie agli SSD (Solid State Drives) è stato possibile avere ottimi tempi di accesso casuale, ma a scapito della capacità attualmente più contenuta rispetto ai comuni Hard Disk.

Il classico e meno sofisticato indice per string matching, il suffix array, occupa da 4 a 8 volte la dimensione del testo (Manber & Myers, 1990). Questo risulta essere un problema perché seppur disponendo di una quantità sufficiente di memoria principale per contenere l'intero testo, l'indice deve risiedere nella memoria di massa.

A complicare la situazione, molti indici non sono ottimizzati per funzionare il memoria secondaria e quindi subiscono enormi rallentamenti durante l'accesso casuale. D'altra parte utilizzare l'indice solamente nei casi in cui è possibile memorizzarlo interamente in RAM è troppo limitativo e spesso non così vantaggioso.

La compressione è una tecnica per limitare la dimensione di un file e può essere lossy, ossia durante l'operazione di compressione si ha una perdita di informazione o, al contrario, *lossless*. Nel caso dei testi non ci si può permettere di alterare o perdere l'informazione quindi la scelta ricadrà sempre sulle tecniche di compressione *lossless* che offrono una tecnica per rappresentare lo stesso testo, ma utilizzando meno spazio. Data la relazione fra memoria principale e secondaria in termini di tempi di accesso e la velocità di elaborazione delle moderne CPU, l'utilizzo della compressione aiuta molto a ridurre i tempi di trasferimento totali.

1.1 NOTAZIONE E CONCETTI BASE

1.1.1 STRINGHE

Sia $S = S[1]S[2]\ ...\ S[n]$ una stringa o sequenza di caratteri di lunghezza n, $|S| = n$. Ciascun carattere è un elemento di un insieme finito totalmente ordinato Σ, detto *alfabeto*, $|\Sigma| = \sigma$. Ciascun carattere della stringa è indicizzato come $S[i]$ oppure $S_i\ \forall\ 1 \leq i \leq n$.

Si denota con $S[i,j]$ la sottostringa di S dal carattere i al carattere j:

$$S_i S_{i+1}\ ...\ S_j\ \forall\ i \leq j \leq n.$$

Un prefisso è una sottostringa di S nella forma $S_1 S_2\ ...\ S_i\ \forall\ 1 \leq i \leq n$.

Un suffisso è una sottostringa di S nella forma $S_i S_{i+1}\ ...\ S_n\ \forall\ 1 \leq i \leq n$.

L'ordine lessicografico "<" fra stringhe è definito come segue: siano a e b caratteri e X e Y stringhe. $aX < bY \iff a < b\ oppure\ a = b\ e\ X < Y$.

Le rotazioni di S sono l'insieme di stringhe:

$$\{S, \quad S_i S_{i+1} \dots S_n S_1 \dots S_{i-1}\ \forall\ 1 < i < n, \qquad S_n S_1 S_2 \dots S_{n-1}\}$$

La stringa inversa di S è una stringa $S' = S_n S_{n-1}\ ...\ S_1$.

1.1.2 INDICI

Come accennato nell'introduzione esistono diversi tipi di indici. Di seguito è esposta una panoramica sulle caratteristiche.

I full-text indexes sono indici che non contengono informazioni circa i caratteri presenti nel testo. Un problema serio di questi indici è lo spazio utilizzato, che va a sommarsi a quello del testo.

Un indice si dice succinto quando riesce a fornire funzionalità di ricerca veloci usando uno spazio proporzionale a quello del testo (ad esempio il doppio). Un concetto più forte è quello di indice compresso, che sfrutta le regolarità del testo al fine di occupare uno spazio proporzionale a quello del testo compresso.

Un indice compresso che contiene l'informazione necessaria a riprodurre qualunque porzione del testo è detto self-index. Un self-index quindi sostituisce il testo.

La possibilità di utilizzare un indice che abbia uno spazio simile a quello del testo, di rimpiazzarlo, e di fornire metodi per una ricerca veloce di sottostringhe, ha portato grossi sforzi all'attività di ricerca nel settore che è evoluta molto negli ultimi anni.

1.2 SUFFIX TREE

Il suffix tree (Weiner, 1973), di una stringa $S, |S| = n$ è definito come un albero avente le seguenti caratteristiche:

1. Possiede n foglie numerate da 1 a n
2. Ad eccezione della radice, ogni nodo interno possiede almeno due figli
3. Ogni arco è etichettato con una sottostringa di S
4. Due archi che condividono lo stesso nodo non posso avere etichette che iniziano con la stessa lettera
5. La stringa ottenuta concatenando tutte le etichette degli archi attraversati nel percorso dalla radice sino ad un nodo foglia numerato con i è
$$S[i,n] \ \forall \ 1 \leq i \leq n$$
6. Per evitare che un suffisso sia prefisso di un altro ed assicurare la presenza di esattamente n foglie, viene aggiunto al termine di S un simbolo che non compare nella stringa, solitamente indicato con "**$**"

Un suffix tree è quindi formato da n nodi foglia, $n - 1$ nodi interni e la radice. La dimensione totale quindi è $2n$ nodi.

La ricerca di pattern P all'interno del suffix tree inizia dalla radice; si cerca un arco avente etichetta un prefisso di P. Se si consuma tutta l'etichetta si procede la ricerca sul nodo raggiunto dall'arco utilizzando la parte rimanente di P. Se non si riesce a consumare tutta l'etichetta perché di lunghezza maggiore della parte rimanente di P allora le occorrenze di P in S sono il numero di nodi foglia presenti nel sottoalbero del nodo raggiunto dall'arco. Se l'etichetta non corrisponde oltre un certo carattere P non compare all'interno di S.

Si tratta di un full-text index non succinto poiché lo spazio occupato è $\Theta(n \log n)$ bits.

1.3 SUFFIX ARRAY

Il suffix array (Manber & Myers, 1990; Gonnet, et al., 1992) è utilizzato come full-text index ed è fra le strutture dati più semplici e efficaci per tale scopo. Si tratta di un indice non succinto poiché richiede spazio $\Theta(n \log n)$.

È stato introdotto nel 1990 come un'alternativa semplice e efficiente ai suffix trees, anch'essi indici non succinti e introdotti nella sezione precedente.

Il suffix array di una stringa è definito come un array di numeri interi, permutazioni degli indici di partenza dei suffissi della stringa in ordine lessicografico.

Il suffix array di $S_{1,n}$ è un Array $A[1...n]$ che contiene le permutazioni nell'intervallo $[1,n]$ tali per cui $S_{A[i],n} < S_{A[i+1],n}\ \forall\ 1 \leq i < n$ dove " $<$ " fra stringhe è l'operatore di ordine lessicografico.

Il suffix array può essere ottenuto utilizzando qualunque algoritmo di ordinamento poiché si tratta semplicemente di ordinare n suffissi. Tale operazione però diventa particolarmente onerosa se ci sono lunghe sottostringhe ripetute all'interno del testo. Questa eventualità peggiora l'operazione di confronto fra due suffissi perché accumunati da un prefisso iniziale comune.

Grazie alla presenza dell'elenco dei suffissi ordinato, la ricerca di un pattern P in S è possibile eseguirla tramite ricerca binaria, quindi in tempo $O(\log n)$. Essendo un full-text index necessita però del testo.

Esistono diversi algoritmi, a partire dall'originale in tempo $\Theta(n \log n)$ (Manber & Myers, 1990), sino alle implementazioni in tempo $\Theta(n)$ (Kim, et al., 2005; Kärkkäinen, et al., 2006; Ko & Aluru, 2005).

1.3.1 ESEMPIO

Di seguito è mostrato un esempio di creazione del suffix array di una stringa.

Per poter confrontare sempre in maniera univoca due suffissi viene aggiunto alla fine della stringa un carattere non presente in Σ: "\$", con l'unico vincolo ad essere lessicograficamente più piccolo di tutti i caratteri dell'alfabeto.

La stringa di esempio $S = "swiss \bullet miss \bullet missing\$"$, $|S| = 19$.

Indice	Suffisso
1	swiss•miss•missing$
2	wiss•miss•missing$
3	iss•miss•missing$
4	ss•miss•missing$
5	s•miss•missing$
6	•miss•missing$
7	miss•missing$
8	iss•missing$
9	ss•missing$
10	s•missing$
11	•missing$
12	missing$
13	issing$
14	ssing$
15	sing$
16	ing$
17	ng$
18	g$
19	$

Elenco ordinato da i.

Indice	Suffisso
19	$
6	•miss•missing$
11	•missing$
18	g$
16	ing$
3	iss•miss•missing$
8	iss•missing$
13	issing$
7	miss•missing$
12	missing$
17	ng$
5	s•miss•missing$
10	s•missing$
15	sing$
4	ss•miss•missing$
9	ss•missing$
14	ssing$
1	swiss•miss•missing$
2	wiss•miss•missing$

Elenco ordinato lessicograficamente da $S[i,n]$

Tabella 1 Elenchi di tuple $(\mathbf{i},\ \mathbf{S}[\mathbf{i},\mathbf{n}])\ \forall\ \mathbf{1} \leq \mathbf{i} \leq \mathbf{n}$

Il suffix array quindi è la prima colonna della seconda tabella, come da definizione introdotta nella sezione 1.3.

Essendo un array di permutazioni di n interi occupa spazio $\Theta(n\ \log n)$. Ciò lo rende un indice non succinto.

1.4 BURROWS-WHEELER TRANSFORM

Come già accennato, la Burrows-Wheeler Transform (BWT) è stata concepita come parte di un algoritmo di compressione dei dati di tipo *lossless*.

La trasformazione reversibile produce una permutazione della stringa originale in cui gli stessi caratteri tendono a raggrupparsi in blocchi; tale caratteristica permette migliori risultati nella compressione di una BWT piuttosto che del testo originale. Per questi motivi, nel 1994 Burrows e Wheeler suggerivano la BWT come passo di *preprocessing* per la compressione (Burrows & Wheeler, 1994), che ne diventerà l'applicazione più rilevante, ad esempio è alla base dell'algoritmo `bzip2`.

Al fine di rendere veloce l'operazione di ordinamento e senza aggravare le prestazioni globali di compressione, è possibile dividere una grossa stringa in diversi blocchi e calcolare in maniera indipendente la BWT di ciascuno di questi. Questo approccio inoltre aggiunge robustezza all'archivio in caso di danneggiamento di una parte del file.

Successivamente la BWT è diventata un importante self-index poiché le affinità al suffix array possono essere sfruttate per renderla una sorta di *suffix array compresso*, come presentato in (Ferragina & Manzini, 2000). Questo suo utilizzo ha trovato molte importanti applicazioni in informatica e bioinformatica.

Mentre nello scenario della compressione è possibile dividere la stringa originale in blocchi, nella creazione di un self-index questa operazione non risulta ammissibile perché, per ottenere risultati ottimali nelle ricerche di sottostringhe di qualunque lunghezza, è richiesta la BWT di tutto il testo. In questo caso, lavorando con grossi testi, sono critici sia il tempo di esecuzione che la memoria occupata durante l'operazione di creazione della BWT.

Lo spazio utilizzato dalla BWT senza l'ausilio di tecniche di compressione è $\Theta(n \log \sigma)$; questo la rende un indice succinto.

Siccome la BWT può essere ricavata in maniera semplice (efficiente in tempo e spazio) dal testo e dal suffix array, i progressi nella creazione di quest'ultimo hanno migliorato indirettamente le prestazioni della creazione della BWT.

Esistono anche algoritmi che computano la BWT direttamente senza passare prima dal suffix array, ad esempio (Bauer, et al., 2011).

1.4.1 ESEMPI

Nelle prossime sezioni sono proposti alcuni esempi: trasformata BWT, ricerca di pattern utilizzando la BWT, e l'operazione per ottenere il testo originale partendo dalla BWT: la trasformata BWT inversa. In tutti gli esempi si utilizza la stringa già vista nell'esempio nella sezione 1.3.1.

1.4.1.1 Trasformata BWT

La BWT di S è la stringa $B[1...n]$ in cui:

$$B_i = \begin{cases} S[A[i]-1] \iff A[i] > 1 \\ S[n] \iff A[i] = 1 \end{cases} \quad \forall\, 1 \leq i \leq n$$

L'indice i tale che $B_i = S[n]$ è detto EOF; nell'esempio EOF = 18.

Sono ora elencate le rotazioni ordinate della stringa S; l'ultimo carattere della rotazione che inizia nell'indice i è

$$\begin{cases} S[i-1] \;\forall\, 1 < i \leq n \\ S[n] \iff i = 1 \end{cases}$$

Da questo si deduce che la lista dei caratteri che in S sono precedenti a ciascuna delle rotazioni ordinate, coincide con la definizione di BWT. Il carattere precedente, per la definizione in sezione 1.1.1, è l'ultimo carattere della rotazione.

Di seguito è mostrata la matrice delle rotazioni ordinate lessicograficamente in cui la prima colonna è chiamata F, e l'ultima, la BWT, è chiamata L.

Indice	F	Rotazione[2,n-1]	L
19	$	swiss•miss•missin	g
6	•	miss•missing$swis	s
11	•	missing$swiss•mis	s
18	g	$swiss•miss•missi	n
16	i	ng$swiss•miss•mis	s
3	i	ss•miss•missing$s	w
8	i	ss•missing$swiss•	m
13	i	ssing$swiss•miss•	m
7	m	iss•missing$swiss	•
12	m	issing$swiss•miss	•
17	n	g$swiss•miss•miss	i
5	s	•miss•missing$swi	s
10	s	•missing$swiss•mi	s
15	s	ing$swiss•miss•mi	s
4	s	s•miss•missing$sw	i
9	s	s•missing$swiss•m	i
14	s	sing$swiss•miss•m	i
1	s	wiss•miss•missing	$
2	w	iss•miss•missing$	s

Tabella 2 Matrice delle rotazioni di ***S*** con evidenziate le colonne ***F*** ed ***L***

La BWT di $S =$ "$swiss \bullet miss \bullet missing\$$" è $B =$ "$gssnswmm \bullet\bullet isssiii\$s$".

Esiste una proprietà molto importante fra la colonna L e la colonna F: l'ordine relativo di occorrenze dello stesso carattere è mantenuto in entrambe le colonne. Questa proprietà è sfruttata per la ricerca e per ricavare il testo originale dalla BWT.

1.4.1.2 Ricerca di pattern utilizzando la BWT

Essendo la BWT un self-index è possibile eliminare il testo e ovviamente anche il suffix array in quanto entrambi contengono informazioni ridondanti; sezione 1.1.2.

Ordinando i caratteri della BWT B, che corrispondono all'ultima colonna (chiamata L) della matrice quadrata delle rotazioni, si ottiene nuovamente la prima colonna, chiamata F. Per ottenerla però non è necessario usare algoritmi di ordinamento. Si contano le

occorrenze di ciascun carattere in L e si dispongono in un array di dimensione $\sigma + 1$ chiamato Occ.

Per fare questo si utilizza una funzione:

$$\text{Map} : \Sigma \cup \{"\$"\} \rightarrow \mathbb{N}.$$

In ciascuna posizione $\text{Occ}[\text{Map}(c)]$ sono contante le occorrenze di c in L con il vincolo:

$$i < j \Leftrightarrow \text{Map}(i) < \text{Map}(j)\ \forall\ 1 \leq i \leq \sigma + 1$$

Il carattere "\$", per definizione il più piccolo lessicograficamente, è quindi sempre mappato nella posizione di Occ $i = 1$. Le occorrenze dei restanti σ caratteri saranno indicate nelle posizioni $2 \leq i \leq \sigma + 1$.

Da questo si genera l'array cumulativo delle occorrenze:

$$\text{FC} = [\ \textstyle\sum_{j=1}^{i} \text{Occ}[j]\ \ \forall\ 1 \leq i \leq \sigma + 1\ ]$$

Quindi ciascun intervallo:

$$[\ \text{FC}[i] - \text{Occ}[i] + 1,\ \text{FC}[i]\ ]\ \forall\ 1 \leq i \leq \sigma + 1$$

indica l'intervallo in cui compare il carattere $c = \text{Map}(i)$ in F.

Il carattere "\$" si troverà sempre nell'intervallo $[1,1]$, quindi sempre e solo in prima posizione. Questo è confermato dalla presenza di un'unica occorrenza lessicograficamente più piccola di tutte quelle presenti nell'alfabeto.

Come anticipato, esiste una proprietà molto importante fra F ed L: l'ordine relativo di occorrenze dello stesso carattere è mantenuto in entrambe le colonne. Ciò significa che l'i-esima occorrenza del carattere c in F coincide con l'i-esima occorrenza del carattere c in L. Questa proprietà è sfruttata per effettuare il *mapping L-F*: mappare un'occorrenza di un carattere in L alla sua stessa occorrenza in F.

Trovare l'indice in F dell'i-esima occorrenza del carattere c è molto semplice:

$$\text{FC}[\text{Map}(c)] - \text{Occ}[\text{Map}(c)] + 1 + i$$

Risulta però più difficile la stessa operazione in L.

Sia P = "mis" il pattern da cercare all'interno della stringa S. Utilizzando la *backward search* si procede alla ricerca delle occorrenze dall'ultimo carattere di P, procedendo a ritroso. Si inizia cercando l'intervallo delle occorrenze del carattere "s", in F. Si tratta di un'operazione molto semplice:

$$[\ \mathrm{FC}[\mathrm{Map}("s")] - \mathrm{Occ}[\mathrm{Map}("s")] + 1,\ \mathrm{FC}[\mathrm{Map}("s")]\ ]$$

il cui risultato è $[12, 18]$.

Nella sottostringa $L[12, 18]$ sono presenti tutti i caratteri precedenti a ciascuna delle occorrenze di "s" in F. Quindi fra questi si cercano solo le occorrenze del carattere precedente di P: "i"; sono 3: $\{L[15],\ L[16],\ L[17]\}$.

Per ottenere questo risultato è necessario introdurre l'operazione $\mathrm{rank}(c, i)$. Questa operazione conta quante sono le occorrenze di c nei primi i caratteri di L. Quindi per ottenere il risultato voluto l'operazione da eseguire è:

$$k = \mathrm{rank}("i", 18) - \mathrm{rank}("i", 12)$$

che restituisce $4 - 1$.

Il risultato ottenuto comporta che le 3 occorrenze trovate sono la 2°, 3° e 4°. Si procede quindi in maniera semplice a trovare le corrispondenti occorrenze in F, la prima è:

$$\mathrm{FC}[\mathrm{Map}("i")] - \mathrm{Occ}[\mathrm{Map}("i")] + 1 + \mathrm{rank}("i", 12)$$

le altre sono nelle posizioni a seguire sino all'ultima in:

$$\mathrm{FC}[\mathrm{Map}("i")] - \mathrm{Occ}[\mathrm{Map}("i")] + 1 + \mathrm{rank}("i", 12) + k - 1$$

Il risultato è l'intervallo di F $[6, 8]$.

Si inizia quindi ricorsivamente a cercare quante occorrenze del carattere "m" sono le precedenti a quelle del carattere "i" appena trovate, quindi comprese nell'intervallo in $L[6, 8]$:

$$k = \mathrm{rank}("m", 8) - \mathrm{rank}("m", 6)$$

Il risultato è 2, significa che sono presenti 2 occorrenze del pattern P = "mis" in S.

La fase critica in questa procedura di ricerca è l'operazione $\text{rank}(c, i)$. Una ricerca lineare anche con conteggi parziali ausiliari è comunque in tempo $\Theta(n)$.

Utilizzando una struttura *wavelet tree*, che verrà presentata nella sezione 4.1, è possibile tramite una fase di *preprocessing* in tempo $\Theta(n)$, ottenere una ricerca in tempo $\Theta(\log \sigma)$. Questo porta l'intera ricerca di un pattern P ad essere possibile in tempo $\Theta(|P| \log \sigma)$, quindi indipendente dalla lunghezza del testo n.

1.4.1.3 Trasformata BWT inversa

Essendo la BWT l'ultima colonna della matrice delle rotazioni ordinate di una stringa S, è facile intuire che coincida con la BWT di qualunque rotazione di S. Ordinando le rotazioni infatti si perde l'informazione di quale delle rotazioni corrisponda alla stringa originale.

Tramite il *mapping L-F* è possibile risalire da una qualunque occorrenza di un carattere c in L all'indice i in cui si trova in F; $L[i]$ corrisponde quindi al carattere precedente a c in S. Iniziando questa procedura da un carattere casuale in posizione i in B e iterandola n volte, si ottiene l'inversa della rotazione inizialmente presente nella matrice in riga i; il risultato è inverso perché il *mapping L-F* restituisce ogni volta il carattere precedente. Continuando ad iterare ci si troverà in un loop.

Per ottenere la stringa inversa al testo originale S è quindi necessario partire dall'ultimo carattere di S in B. Come anticipato, la posizione del carattere "$" in B è chiamata EOF. Questa posizione corrisponde all'indice della matrice delle rotazioni in cui è presente la stringa originale. Viene quindi utilizzato EOF come indice di partenza della procedura iterata n volte di *mapping L-F* che restituirà la stringa originale inversa.

2 ALGORITMI DI COSTRUZIONE

Esistono molti algoritmi per il calcolo del suffix array o della BWT con differenti complessità in tempo e spazio. La modalità di utilizzo della memoria porta a suddividere gli algoritmi in internal, semi-external e external.

Gli algoritmi internal, durante l'esecuzione, utilizzano una quantità di memoria RAM dipendente dalla lunghezza del testo in input. Tutti i dati necessari all'esecuzione sono mantenuti in RAM e non viene fatto uso della memoria esterna. Questa caratteristica li rende molto veloci ma, nel contempo, limita il loro utilizzo. Infatti spesso è infattibile il loro utilizzo perché anche i computer più moderni faticano a disporre della quantità necessaria di memoria per costruire la BWT dei documenti o biosequenze oggi disponibili.

Gli algoritmi semi-external sono –almeno in questa tesi– algoritmi che utilizzano una quantità di memoria dipendente dall'input (come gli algoritmi internal), sfruttando anche memoria di massa. In pratica cercano di sfruttare la memoria di massa per i dati che necessitano di accesso sequenziale, mentre la memoria RAM contiene solamente i dati che richiedono un accesso diretto.

Gli algoritmi external utilizzano una quantità di memoria indipendente dalla dimensione dell'input e solitamente fornita dall'utente come parametro di avvio. In ausilio, per operazioni sequenziali, viene utilizzata la memoria di massa. Questo approccio porta rallentamenti significativi, quindi gli algoritmi external vengono utilizzati solamente quando la memoria disponibile è minore di quella richiesta per l'esecuzione di un algoritmo internal. Tale eventualità, che si presenta spesso già per file relativamente piccoli, rende gli algoritmi external molto importanti perché finalmente risolvono il problema della memoria.

Di seguito alcuni esempi di algoritmi di queste tre categorie utilizzati per la creazione del suffix array e della BWT.

Dal punto di vista teorico il suffix array può essere costruito in tempo $O(n)$ ed in spazio $O(n \log n)$ (Kim, et al., 2005; Ko & Aluru, 2005; Kärkkäinen, et al., 2006). Questi tempo e spazio sono asintoticamente ottimali ma, per i motivi visti sopra, nella pratica

siamo interessati ad algoritmi che richiedono la minore quantità di memoria possibile, cioè solamente quella necessaria a contenere l'input (il testo) e l'output (il suffix array).

E' ampiamente concordato che nella pratica, l'algoritmo internal di Yuta Mori `divsufsort` è il più veloce nella computazione del suffix array, anche se la richiesta in tempo nel caso peggiore è $O(n^2)$. L'utilizzo di memoria è di $5n$ bytes per $n < 2^{31}$, $9n$ bytes per $2^{31} \leq n < 2^{63}$. Quindi per computare il suffix array di un genoma umano (~3 GB $>$ 2^{31}) sono necessari ~27GB di memoria RAM, una quantità piuttosto elevata.

Il problema della costruzione della BWT, come anticipato in sezione 0, è possibile risolverlo in tempo $O(n)$ partendo dal suffix array. Si osserva però che sia input (il testo) che output (la BWT), richiedono spazio $O(n \log \sigma)$. Come conseguenza è possibile costruirla utilizzando spazio asintoticamente inferiore a $\theta(n \log n)$ necessario nel caso della creazione partendo dal suffix array. In (Okanohara & Sadakane, 2009) è presentato un algoritmo internal per la creazione della BWT in tempo lineare che utilizza spazio $O(n \log \sigma \log \log_{\sigma} n)$.

L'algoritmo presentato in (Beller, et al., 2013) è un esempio di algoritmo semi-external per la creazione della BWT che utilizza spazio in memoria RAM pari a n bytes. Tale richiesta talvolta può essere difficile da soddisfare, ad esempio per collezioni di articoli (~50 GB) è difficile disporre della memoria necessaria.

L'algoritmo presentato in (Ferragina, et al., 2012) è un esempio di algoritmo external per il calcolo sia della BWT che del suffix array. Dato che è il punto di partenza per questa tesi sarà illustrato più dettagliatamente nella prossima sezione.

2.1 BWTE

L'algoritmo `bwte` è parte della libreria chiamata `bwtext` (Ferragina, et al., 2012) e scaricabile all'indirizzo: http://people.unipmn.it/manzini/bwtdisk/.

Si tratta di un algoritmo external per il calcolo della BWT. Come parametri di avvio sono richiesti:

- Il percorso del file contenente il testo T', $|T'| = n$
- La quantità di memoria m che si desidera utilizzare per il calcolo

L'output sarà la BWT del testo in input inverso. Per questo motivo per ottenere la BWT B di un testo T è necessario fornire come parametro il nome del file contenente il testo inverso, ossia T'. La lettura sequenziale di T' è equivalente alla lettura inversa del testo T, quindi T' sostituisce T ed il primo carattere letto in T' è l'ultimo del testo T.

Esiste una limitazione per la memoria: $m < 6 \times 2^{32}$.

Il file verrà diviso in blocchi di dimensione $b = \lfloor {}^{m}/_{6} \rfloor$. Si procederà iterativamente alla computazione della BWT e/o del suffix array di un blocco e la fusione di quest'ultimo con i precedenti finché non sono stati esauriti i blocchi.

Il primo passo di computazione consiste nel leggere al più $b - 1$ caratteri da T' e salvarli in memoria invertiti, concatenando il carattere di fine stringa:

$$C_1 = \begin{cases} T'[1,\ b-1]'\$ \ \equiv\ T[n-b+2,\ n]\$ \Leftrightarrow n \geq b \\ T''\$ \ \equiv T\$ \Leftrightarrow n < b \end{cases}.$$

Per poter offrire la possibilità di computare la BWT di un testo con alfabeto sino a 256 caratteri, il funzionamento interno prevede che il carattere di fine stringa sia un carattere dell'alfabeto. Tramite indici però rimane conosciuta la sua posizione in maniera tale da poter ricostruire T' correttamente. Il carattere di fine stringa continuerà ad essere indicato come \$ poiché si tratta di un comodo metodo di rappresentazione.

La stringa appena salvata in memoria $C_1, |C_1| = b$ occupa spazio b bytes ed è indicata come "primo blocco". Si tratta di un suffisso di T che termina con il carattere di fine stringa lessicograficamente più piccolo di tutti gli altri. Per questo motivo è possibile confrontare tutti i suffissi e quindi generare il suffix array.

Viene utilizzato l'algoritmo `divsufsort` per la computazione del suffix array, che comporta un utilizzo di $4b$ bytes e portando l'utilizzo globale di memoria a $5b$. Il risultato della computazione è il suffix array del primo blocco, chiamato A_1.

È possibile ottenere la BWT del primo blocco, chiamata B_1, utilizzando ancora b bytes di memoria e portando quindi l'utilizzo globale a $6b \cong m$.

$$B_1[i] = \begin{cases} T[A_1[i] - 1] \ \Leftrightarrow\ A_1[i] \ > \ 1 \\ \$ \Leftrightarrow\ A_1[i] \ = 1 \end{cases} \quad \forall\, 1 \leq i \leq b$$

L'indice i tale che $B_1[i] = \$$ è detto EOF e viene salvato nel file della BWT.

Se $n < b$ l'operazione è conclusa e $B \equiv B_1$.

Se $n \geq b$, invece, iniziano $h = \left\lceil \frac{n+1}{b} \right\rceil$ iterazioni sui blocchi successivi. Verranno letti sino a b caratteri per ogni blocco successivo, tranne nell'ultimo dove la fine di T' può rendere l'ultimo blocco più piccolo:

$$C_j, |C_j| = \begin{cases} b \iff 1 < j < h \bigvee (n+1)\%b = 0 \\ (n\%b) + 1 \iff \text{altrimenti} \end{cases} \quad \forall\, 1 < j \leq h$$

Ad ogni iterazione $j > 1$ sono computati il suffix array e la BWT del blocco C_j e gli array FC e Occ introdotti nella sezione 1.4.1.2 che ne emulano la colonna F.

Durante la creazione del suffix array del blocco $C_j, j > 1$ Può capitare che il confronto fra due suffissi dello stesso blocco C_j non sia possibile perché hanno in comune un prefisso di lunghezza tale da raggiungere $C_j[|C_j|]$. In questa situazione non è possibile proseguire con il confronto dei caratteri successivi perché appartenenti al blocco precedente, C_{j-1}. Per risolvere questo problema, prima di eliminare C_{j-1} si completa un array di bit chiamato gt. Questo array, in ogni posizione $1 \leq i < n$ indica $C_{j-1} < T[i,n]$, dove "<" è l'operatore di ordine lessicografico. Quindi gt$[i] = 1$ se il suffisso $T[i,n]$ che inizia nel blocco C_{j-1} è lessicograficamente maggiore dell'intero suffisso $C_{j-1} \ldots C_1\$$. Grazie a questo è possibile confrontare sempre due suffissi.

L'ordine relativo fra i caratteri della BWT parziale B_{j-1} sarà mantenuto nella nuova BWT parziale B_j perché anch'essa è la colonna L di rotazioni di suffissi ordinati lessicograficamente. L'incognita è quanti caratteri di B_{j-1} separano due caratteri consecutivi della BWT del blocco C_j perché relativi a suffissi di C_j lessicograficamente compresi tra i due. Viene quindi creato l'array gaps, un array che per ogni posizione $1 < i \leq |C_j| + 1$ indica quanti suffissi di B_{j-1} sono lessicograficamente compresi fra i suffissi $C_j[A_j[i-1],\ |C_j|]$ e $C_j[A_j[i],\ |C_j|]$.

Il valore contenuto in gaps$[1]$ indica quanti suffissi in B_{j-1} sono minori di qualunque suffisso di C_j. Il valore contenuto in gaps$[|C_j| + 1]$ indica quanti suffissi in B_{j-1} sono maggiori di qualunque suffisso di C_j.

La fase di merge quindi copia sequenzialmente i caratteri dalla vecchia BWT parziale alla nuova BWT parziale inserendo nelle posizioni opportune i caratteri della BWT del blocco C_j. Le posizioni di inserimento sono fornite dall'array gaps.

La creazione dell'array gaps è possibile perché la lettura dei testi avviene in ordine inverso, quindi si parte dal carattere di fine stringa che è il suffisso più piccolo, a cui vengono affissi i caratteri man mano letti.

2.2 OTTIMIZZAZIONI IN BWTE

In questa tesi sono state sviluppate due ottimizzazioni a `bwte`.

La prima riguarda il suffix array: è stata implementata la funzionalità di creazione del suffix array del testo in input inverso.

La seconda riguarda l'ottimizzazione dei tempi di lettura/scrittura tramite funzioni che gestiscono il lavoro in maniera parallela.

2.2.1 SUFFIX ARRAY

È stata implementata la funzionalità di creazione del suffix array del testo T inverso.

Il suffix array è creato ad ogni passo sul blocco corrente per ricavare la BWT. Se ci si trova al passo $j = 0$ viene salvata come B_1; se ci si trova al passo $j > 1$ è necessaria per la creazione dell'array gaps, e per il merge successivo.

Nel primo caso è possibile semplicemente salvare su disco il suffix array come A_1.

Per i passi successivi, il suffix array viene cancellato dopo la creazione della BWT perché l'array gaps richiede troppo spazio: non sarebbe possibile mantenere l'utilizzo di memoria entro il limite m utilizzando blocchi di dimensione $b = {}^{m}/_{6}$.

Ultimata la creazione di gaps però è possibile notare che i valori in esso contenuti sono gli stessi necessari al merge del suffix array; questo perché c'è una corrispondenza biunivoca fra la BWT ed il suffix array.

L'unico accorgimento essenziale durante l'operazione di merge al passo j è dover incrementare il valore di tutti i suffissi in A_{j-1} di $|C_j|$. Questo perché in A_j gli indici

sono relativi a suffissi il cui indice di partenza è aumentato a causa dell'inserimento del nuovo blocco.

Dopo aver effettuato il merge della BWT di C_j con B_{j-1}, è possibile computare nuovamente il suffix array di C_j *in-place*, ossia sovrascrivendo la BWT di C_j senza la richiesta di ulteriore spazio. Questo è fatto in due passi: prima è computato *in-place* l'array rank_prev, e poi sempre *in-place* è computato il suffix array. La tecnica utilizzata è simile a quella introdotta per la prima volta in (Manzini, 2004) e di seguito accennata.

L'array RankPrev di un testo $T, |T| = n$ mantiene una corrispondenza fra indici del suffix array: l'indice di un suffisso e l'indice del suffisso che è precedente in T. Questo array è definito nelle posizioni $i > 1$ perché non è possibile ottenere l'indice nel suffix array del suffisso precedente al primo: non ci sono caratteri prima.

$$\text{SA}\big[\text{RankPrev}[i]\big] + 1 = \text{SA}[i] \;\; \forall\, 1 < i \leq n$$

Una volta ottenuto l'array RankPrev si procede, sempre in-place, alla computazione del suffix array e all'operazione di merge apposita.

Per il salvataggio del suffix array è stato creato un formato apposito. Trattandosi di una permutazione di n interi nell'intervallo $1 \dots n$, si è deciso di utilizzare $b = \log_2 n$ bit per ciascuna locazione. Nei primi 64 bit del file è indicato n utilizzando l'ordinamento dei bit *little-endian*. Le stringhe binarie, ciascuna lunga b bit, corrispondenti alle locazioni del suffix array, sono concatenate e salvate su disco utilizzando l'ordinamento *little-endian*. Lo spazio occupato su disco risulta quindi essere $n \log_2 n$ bit.

È stato inoltre creato un programma che legge un file in questo formato. All'apertura del file legge i primi 64 bit in cui è rappresentato il valore n e calcola $b = \log_2 n$. Successivamente trasforma tutti i valori ottenuti dalle porzioni di b bit del resto del file in interi a 32 o 64 bit, salvando così su file una versione estesa del suffix array.

2.2.2 PARALLELISMO

L'operazione di creazione dell'array gaps legge un carattere per volta da T' ricavando quindi ogni volta un suffisso di T formato dal nuovo carattere letto affisso al suffisso precedente. Si calcola l'indice i della colonna F di C_j che rende il suffisso appena

ottenuto lessicograficamente compreso fra $C_j[A[i-1],n]$ e $C_j[A[i],n]$. Si incrementa $\mathrm{gaps}[i]$ e si procede alla lettura di un nuovo carattere.

Questa procedura sfrutta molto sia l'accesso al disco che il tempo CPU. Non è possibile parallelizzare il lavoro del calcolo degli indici di gaps da incrementare perché sono un insieme di operazioni consecutive i cui risultati dipendono dai risultati precedenti. Risulta però possibile parallelizzare la lettura di T' salvando un certo numero di caratteri in un buffer in modo tale da renderli disponibili immediatamente quando richiesti. Questo approccio, nel caso ottimale, annulla le chiamate di sistema bloccanti del thread principale che si occupa del calcolo dell'array gaps.

L'operazione di merge invece sfrutta in maniera intensiva l'accesso al disco perché viene letto T' e copiato carattere per carattere nella nuova BWT parziale inserendo dove indicato da gaps un carattere della BWT del blocco attuale. In questo caso è possibile parallelizzare sia la lettura che la scrittura su disco in modo tale da rendere disponibile immediatamente un carattere, che verrà inserito in un buffer in memoria scritto su disco in un momento successivo. Questo approccio, nel caso ottimale, annulla le chiamate di sistema bloccanti del thread principale che si occupa di gestire l'operazione di merge.

Al file di ottenere questi miglioramenti e sfruttare le moderne architetture CPU che permettono, grazie alla presenza di più core CPU, l'esecuzione realmente parallela di più thread, è stata sviluppata la libreria `io_class`.

La libreria, scritta nel linguaggio C, fornisce una astrazione asincrona alla lettura e scrittura di qualunque file. È stato fatto utilizzo dei threads definiti nello standard POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995), e dei semafori definiti nello standard POSIX.1b, Real-time extensions (IEEE Std 1003.1b-1993) (IEEE Standards Association, s.d.).

La libreria fornisce un insieme di funzioni di apertura, lettura/scrittura e chiusura di un file eventualmente compresso. Le funzioni di apertura restituiscono un puntatore ad una struttura che indica univocamente l'istanza di apertura di quel file. Sarà necessario fornire in input questo puntatore a tutte le altre funzioni perché contiene informazioni circa le strutture allocate per le operazioni sul file. Un puntatore restituito dalla funzione di apertura in lettura non sarà utilizzabile nelle funzioni di scrittura e viceversa.

Viene avviato un thread per ciascun file aperto in lettura o scrittura. La lettura e la scrittura saranno il più possibile gestite parallelamente. Il thread principale, in cui è in esecuzione in programma, fornisce i dati al thread di lettura/scrittura che si occupa di gestirli; se quest'ultimo è occupato e non può accettare o fornire altri dati la chiamata sarà bloccante ed il thread principale verrà bloccato. Utilizzando dei valori opportuni per la dimensione ed il numero di buffers, questa eventualità risulta piuttosto rara.

2.2.2.1 Funzioni per la lettura asincrona di un file

- `file *_bwtext_io_open(char*, filter*)`

Viene aperto il file il cui percorso è contenuto nella stringa.
Viene avviato il thread di lettura che, in parallelo, legge dal file e riempie un insieme di n buffers, ciascuno di dimensione b. Entrambi i valori sono automatici o definiti tramite parametri di avvio.
I puntatori di lettura corrente sono impostati al buffer $cb = 0$ alla posizione $cp = 0$.
Per la lettura è possibile utilizzare un filtro che si occupa di decomprimere il file nel caso fosse compresso e restituire i caratteri del file originale. Anche questa operazione è fatta in parallelo dal thread.
Al termine viene restituita l'istanza di apertura del file.

- `int32 _bwtext_io_getc(file*)`

Utilizzando l'istanza di apertura viene restituito il prossimo carattere prelevato dal buffer in lettura. Vengono utilizzati gli indici di lettura cb e cp per risalire al carattere corrente; se il file è finito viene restituito EOF.
Se il buffer cb non è stato ancora riempito del tutto questa chiamata sarà bloccante. Si rimarrà in attesa fino a che il thread di lettura abbia riempito tutto il buffer. Per questa operazione vengono utilizzati i semafori. Utilizzando dei valori di n e b opportuni rispetto alla velocità del disco ad alla quantità di lavoro del thread principale, questa eventualità viene resa molto rara.
Al termine vengono aggiornati i contatori di lettura.

- `bool _bwtext_io_moretocome(file*)`

Questa funzione indica se alla prossima richiesta di `_bwtext_io_getc` verrà restituito EOF oppure un nuovo carattere.

- `uint32 _bwtext_io_copy_reverse(file*, uint8*, uint32)`

Questa funzione ha come input l'istanza di apertura, una stringa da riempire ed un numero di caratteri. Vengono copiati nella stringa al più i caratteri richiesti al contrario, utilizzando iterativamente la funzione `_bwtext_io_getc`.

Viene restituito il numero di caratteri copiati effettivamente.

- `void _bwtext_io_rewind(file*)`

Questa funzione ha come input l'istanza di apertura di un file e ne azzera gli indici predisponendo tutto alla lettura del file dall'inizio.

- `void _bwtext_io_close(file*)`

Questa funzione ha come input l'istanza di apertura di un file e provvede a terminare la lettura e liberare tutte le strutture allocate.

2.2.2.2 *Funzioni per la scrittura asincrona di un file*

- `file *_bwtext_io_open_write(char*, filter*)`

Analogamente alla lettura, l'apertura di un file per la scrittura è immediata e restituisce un puntatore all'istanza di apertura del file.

Viene avviato il thread di scrittura e vengono allocati un insieme di n buffers, ciascuno di dimensione b. Il thread di scrittura è messo in attesa che almeno uno di questi n buffers sia pieno e quindi pronto alla scrittura.

Può essere utilizzato un filtro che prevede a comprimere l'output prima di scriverlo sul disco. Anche questa operazione è fatta in parallelo.

- `void _bwtext_io_puts(file*, uint8*, uint64)`

Questa funzione ha come input l'istanza di apertura di un file, una stringa e la dimensione della stringa. Viene copiata la stringa nel buffer corrente fra quelli dedicati alla scrittura su disco. Se è stato riempito un buffer ed il prossimo è ancora in scrittura questa chiamata sarà bloccante. Si rimarrà in attesa fino a che il thread di scrittura abbia scritto tutto il buffer su disco. Per questa operazione vengono utilizzati i semafori. Utilizzando dei valori di n e b opportuni rispetto alla velocità del disco ad alla quantità di lavoro del thread principale, questa eventualità viene resa molto rara.
Al termine vengono aggiornati i contatori di scrittura.

- `uint64 _bwtext_io_close_write(file*)`

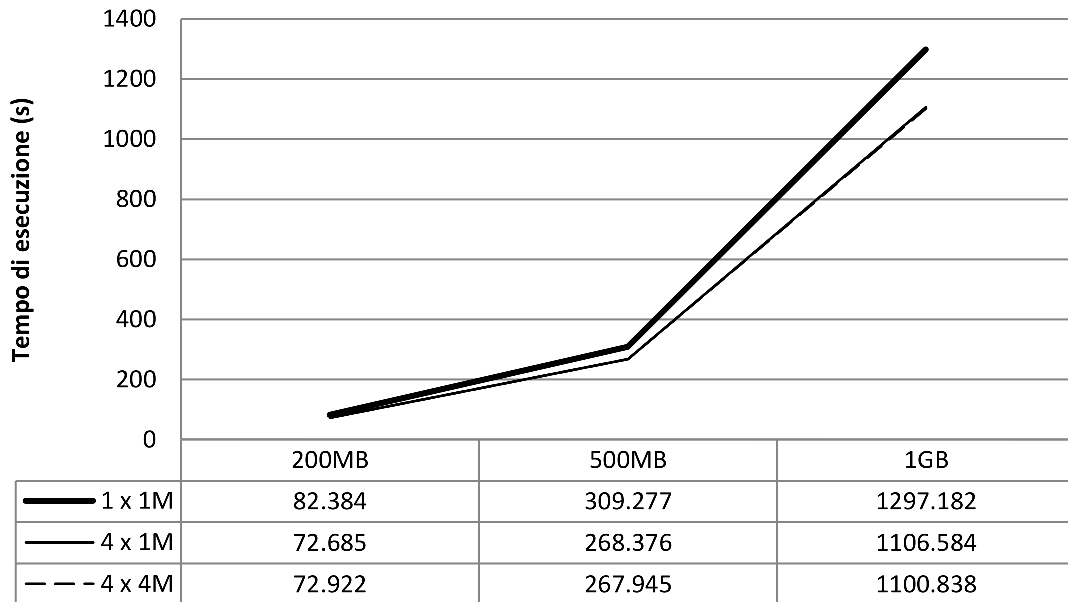
Questa funzione ha come input l'istanza di apertura di un file e provvede a terminare la scrittura e liberare tutte le strutture allocate.

2.2.3 Test

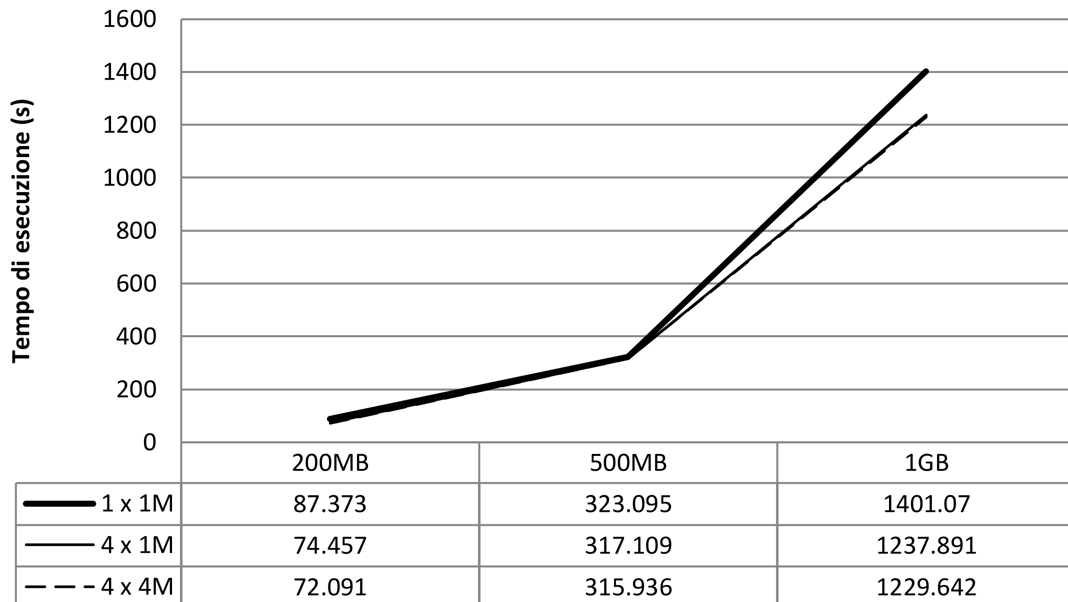
Di seguito i grafici del tempo di esecuzione in secondi di `bwte` nei vari casi, limitando la memoria utilizzata ad 1GB, quindi a blocchi di circa 170MB.

Le prove sono state effettuate su 3 files in input con alfabeto di 256 simboli e compressi tramite l'algoritmo `gzip`. La BWT in output è stata direttamente compressa con lo stesso algoritmo, mentre il suffix array no. Compressione e decompressione sono quindi a carico della libreria `io_class` e quindi gestite dal thread parallelo.

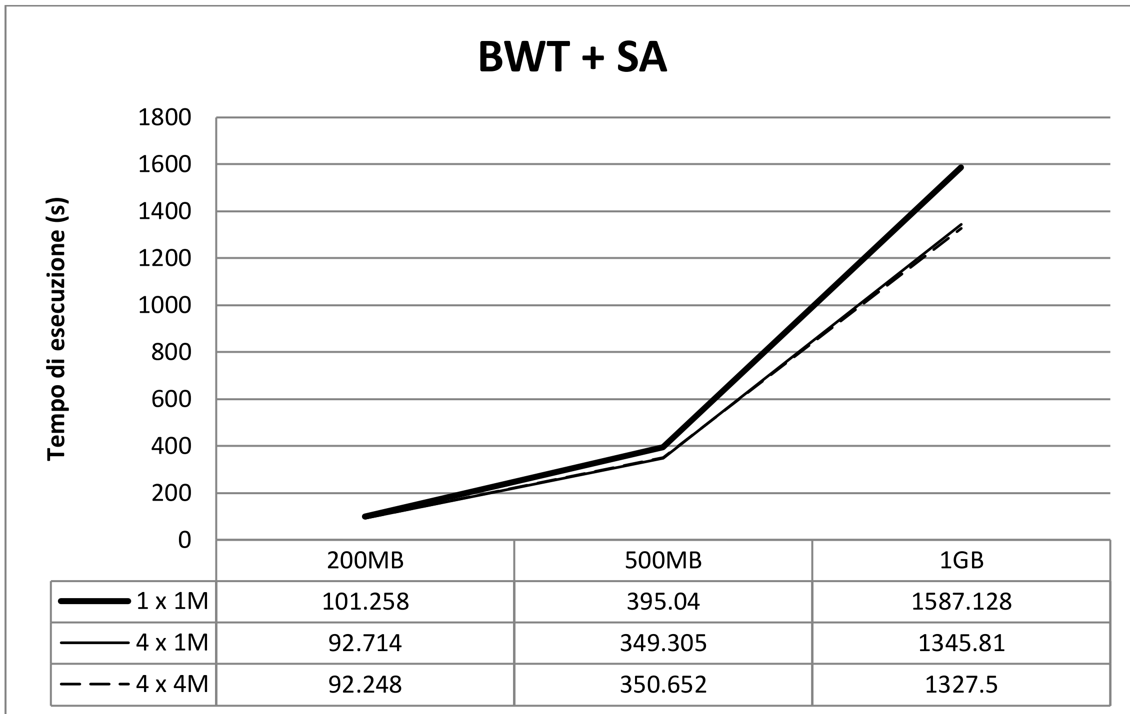
È stato scelto di usare 1 blocco da 1 MB come caso meno esoso in cui non si ha parallelismo, 4 blocchi da 1M come caso intermedio e infine 4 blocchi da 4 MB. Non è stato scelto di utilizzare il caso 1 blocco da 4 MB poiché i risultati sono analoghi, se non peggiori, al caso 1 blocco da 1MB: in entrambi i casi non si ha parallelismo.



	200MB	500MB	1GB
1 x 1M	82.384	309.277	1297.182
4 x 1M	72.685	268.376	1106.584
4 x 4M	72.922	267.945	1100.838



	200MB	500MB	1GB
1 x 1M	87.373	323.095	1401.07
4 x 1M	74.457	317.109	1237.891
4 x 4M	72.091	315.936	1229.642



	200MB	500MB	1GB
1 x 1M	101.258	395.04	1587.128
4 x 1M	92.714	349.305	1345.81
4 x 4M	92.248	350.652	1327.5

2.2.3.1 Osservazioni

Si nota che il solo utilizzo di più blocchi, e quindi del parallelismo, riduce i tempi di circa il 23% in tutti i test effettuati. Questo miglioramento delle prestazioni è dovuto al numero ridotto di chiamate di sistema bloccanti del thread principale.

L'aumento della dimensione dei singoli blocchi porta miglioramenti per file molto grandi.

3 COLLEZIONI DI STRINGHE

Una collezione di stringhe è definita come un insieme di m testi di lunghezza variabile indicata come l'insieme ordinato $\{T_1, T_2, \dots T_m\}$.

Scenari di questo tipo sono, ad esempio, il risultato del sequenziamento di un genoma oppure, la collezione formata da articoli provenienti dai portali di Wikimedia Foundation (Wikimedia Foundation, Inc., s.d.) come Wikipedia.

I recenti progressi nel campo del sequenziamento del DNA spingono a considerare sempre di più il problema della calcolo della BWT di una collezione di stringe. L'operazione di sequenziamento del genoma umano arriva a produrre più di un miliardo di sequenze, ciascuna lunga da 100 a 300 caratteri. Questo insieme di dati può essere generato in pochi giorni da una singola macchina per il sequenziamento. Affrontare in maniera opportuna la compressione e l'indicizzazione è cruciale al fine di poter lavorare in tempi più brevi su un insieme così grande.

La quantità di informazioni disponibili nei portali di Wikimedia Foundation sta crescendo esponenzialmente. La fondazione stessa mette a disposizione l'intero set di articoli di ciascun portale liberamente, per poter essere fruito anche senza connessione. La dimensione di tale archivio è di circa 50 GB; anche in questo scenario risulta quindi importante la compressione e l'indicizzazione ai fini della ricerca.

Per rappresentare una collezione è possibile utilizzare un file per ogni testo oppure un unico file contenente tutti i testi opportunamente separati. Il secondo metodo di rappresentazione è da preferire poiché grosse collezioni non sarebbero gestibili con buone prestazioni da un normale file system.

La BWT di tutta la collezione, che servirà da unico self-index per ricerche sui singoli testi di cui è composta, dovrà essere progettata in modo da mantenere l'informazione sulla separazione dei testi; questo per evitare risultati errati in fase di ricerca di un pattern. Utilizzando la BWT della banale concatenazione dei testi di cui è composta, infatti, durante una ricerca si può incorrere in falsi positivi così formati: suffisso di un testo concatenato al prefisso del testo successivo.

In alcuni casi può essere importante poter risalire agli identificativi delle stringhe che contengono il pattern cercato; un esempio è la collezione di articoli.

3.1 ALGORITMI DI COSTRUZIONE PER COLLEZIONI

L'approccio più significativo al problema della costruzione di BWT di collezioni di dati è (Bauer, et al., 2011). Nell'articolo citato sono presentate due varianti dell'algoritmo: una internal ed una external. La versione internal utilizza spazio $O(m \log m)$ bits per computare la BWT di una collezione di m stringhe. La versione external utilizza spazio costante indipendente da m.

Ci sono due grosse limitazioni all'implementazione di questo algoritmo: l'alfabeto limitato e la dimensione costante di tutte le stringhe nella collezione. Trattandosi di un algoritmo fortemente orientato alla costruzione della BWT del risultato di una operazione di sequenziamento, è limitato al solo utilizzo dell'alfabeto di sequenze di DNA composto da 5 caratteri: $\{A, C, G, T, N\}$. L'altra limitazione riguarda la lunghezza delle stringhe, che possono essere di qualunque lunghezza, a condizione che tale lunghezza sia la stessa per tutte le stringhe della collezione.

Queste limitazioni non rendono possibile l'utilizzo di questo algoritmo per la creazione della BWT di una collezione di articoli, quindi di diversa lunghezza e con un alfabeto più grande.

3.2 BWTC E BWTCW

Si presenta ora un algoritmo semi-external per la creazione della BWT di collezioni di documenti e biosequenze di lunghezza diversa ed alfabeto sino a 256 caratteri. L'algoritmo, chiamato `bwtc` è parte della libreria chiamata `bwtext` (Ferragina, et al., 2012) e scaricabile all'indirizzo: http://people.unipmn.it/manzini/bwtdisk/. La tecnica utilizzata per il funzionamento è ispirata a `bwte`, introdotto nella sezione 2.1.

La computazione della BWT di una collezione di m stringhe al più lunghe n richiede tempo $O(nm^2)$ e spazio $6n$ bytes.

Esiste una versione alternativa, chiamata `bwtcw`, equivalente nel funzionamento ma che utilizza una struttura dati di tipo *wavelet tree* per l'operazione di rank. Questa versione richiede spazio $3n$ bytes.

Di seguito è presentato nel dettaglio il funzionamento dell'algoritmo in maniera indipendente dalla funzione rank utilizzata.

L'input è formato da un file contenente un elenco di m coppie di nomi di file, una per riga. In ciascuna coppia il primo elemento indica il nome del file contenente il testo invertito, il secondo elemento indica il nome del file contenente la BWT del testo associato non invertito. Si indica con $(T'_j, B_j)\ \forall\ 1 \leq j \leq m$ la coppia formata dal contenuto dei due files presenti nell'elenco alla riga j.

La lettura sequenziale di T'_j è equivalente alla lettura inversa del testo T_j che ha costruito B_j. Quindi il primo carattere letto in T'_j è l'ultimo del testo T_j.

L'output è la BWT della collezione $\{T_1, T_2, \dots T_m\}$..

Al termine di ogni passo di computazione $1 \leq j \leq m$ è salvata su disco la BWT parziale, ossia la BWT della collezione formata dai primi j testi; al passo iniziale $j = 1$ corrisponde a B_1.

Al passo di computazione $j + 1$ viene letta B_{j+1} e salvata in memoria insieme agli array FC e Occ [1.4.1.2], ricavati durante la lettura, che ne simulano la colonna F. Inizia quindi l'operazione di lettura sequenziale del testo formato dalla concatenazione $T'_1 T'_2 \dots T'_j$ che porterà al completamento di un array di interi chiamato gaps, $|\mathrm{gaps}| = |B_{j+1}| + 1$. La somma di tutte le locazioni di gaps corrisponde alla lunghezza dei testi concatenati appena letti:

$$\sum_{i=1}^{|B_{j+1}|+1} \mathrm{gaps}[i] = |T'_1 T'_2 \dots T'_j|$$

Questo array indicherà come effettuare l'operazione di merge fra la BWT della collezione di testi $\{T_1, \dots T_j\}$ (parziale salvata su disco) e B_{j+1}. Il risultato dell'operazione di merge sarà il salvataggio su disco della BWT della collezione di testi $\{T_1, \dots T_{j+1}\}$ che, se $m = j + 1$, diventerà la BWT completa, altrimenti verrà utilizzata per l'iterazione successiva.

L'ordine relativo fra i caratteri di B_{j+1} sarà mantenuto nella nuova BWT parziale perché anch'essa è la colonna L di rotazioni di suffissi ordinati lessicograficamente.

L'incognita è quanti caratteri della vecchia BWT separano due caratteri consecutivi di B_{j+1} perché relativi a suffissi lessicograficamente compresi tra i due.

La fase di merge quindi copia sequenzialmente i caratteri dalla vecchia BWT parziale alla nuova BWT parziale inserendo nelle posizioni opportune i caratteri di B_{j+1}. Le posizioni di inserimento sono fornite dall'array gaps. Nella locazione gaps$[i]$ è indicato quanti caratteri della vecchia BWT parziale separano i caratteri $B_{j+1}[i-1]$ e $B_{j+1}[i]$, $\forall\ 1 < i \le |B_{j+1}|$.

Nella locazione gaps$[1]$ è indicato il numero di suffissi in $T_1T_2 \dots T_j$ lessicograficamente inferiori a tutti quelli di T_{j+1}. Quindi il carattere $B_{j+1}[1]$, cioè il carattere in colonna L corrispondente al suffisso lessicograficamente più piccolo di T_{j+1}, andrà inserito nella nuova BWT parziale dopo gaps$[1]$ caratteri della vecchia BWT parziale.

Successivamente $\forall\ 1 < i < |B_{j+1}|$, in gaps$[i]$ è indicato il numero di suffissi in $T_1T_2 \dots T_j$ che sono lessicograficamente maggiori di quello in posizione $i-1$ in T_{j+1} (il cui carattere in colonna L corrisponde a $B_{j+1}[i-1]$), ma inferiori a quello in posizione i in T_{j+1} (il cui carattere in colonna L corrisponde a $B_{j+1}[i]$). Quindi dovranno essere copiati gaps$[i]$ caratteri dalla vecchia BWT parziale che separano $B_{j+1}[i-1]$ e $B_{j+1}[i]$.

L'ultima locazione, gaps$[|B_{j+1}|+1]$, contiene il numero di suffissi in $T_1T_2 \dots T_j$ che sono lessicograficamente maggiori di tutti i suffissi di T_{j+1}. A questo punto sono stati inseriti correttamente tutti i $|B_{j+1}|$ caratteri da B_{j+1}e rimangono da copiare gaps$[|B_{j+1}|+1]$ dalla vecchia BWT parziale alla nuova BWT parziale per completarla.

A questo punto, come anticipato, è stata creata e salvata su disco la BWT della collezione $\{T_1, \dots T_{j+1}\}$. Se $m < j+1$ si procede all'iterazione successiva, altrimenti si termina avendo creato correttamente la BWT della collezione $\{T_1, T_2, \dots T_m\}$. Di seguito è mostrato come avviene la creazione dell'array gaps, essenziale nell'operazione di merge fra la BWT della collezione $\{T_1, T_2, \dots T_j\}$ e B_{j+1}.

Per la creazione dell'array gaps è necessario confrontare tutti i suffissi in $T_1T_2 \dots T_j$ con quelli di B_{j+1}. (Si ricorda che le coppie $(T'_j, B_j)\ \forall\ 1 \le j \le m$ sono formate da testo invertito e BWT del testo non invertito; il testo T_j non è disponibile.)

Ricapitolando, ci si trova quindi al passo j ed inizia la lettura sequenziale di $T'_1 T'_2 \dots T'_j$ per creare l'array gaps relativo alla BWT B_{j+1} salvata in memoria insieme agli array FC e Occ [1.4.1.2], ricavati durante la lettura, che ne simulano la colonna F.

Alla sola apertura di ogni $T'_k \; \forall \; 1 \leq k \leq j$ viene incrementato di una unità gaps[1]. Questo perché virtualmente è stato letto il carattere di fine stringa di T_k indicato come "$\$_k$" che, per convenzione, è lessicograficamente inferiore di tutti i caratteri di B_{j+1}.

Quindi successivamente viene letto il primo carattere, $T'_k[1]$ che, come anticipato, corrisponde all'ultimo carattere di T_k che ha costruito B_k. Questo carattere indica il primo carattere di un suffisso di T_k così formato: $T'_k[1]\$_k$. Questo suffisso sarà più piccolo di tutti quelli in B_{j+1} che iniziano con il carattere $T'_k[1]$. L'indice di partenza di questi suffissi è facile trovarlo, si tratta dell'indice in colonna F in cui inizia a comparire $T'_k[1]$:

$$\text{cur_rank} = \text{FC}[\text{Map}(T'_k[1])] - \text{Occ}[\text{Map}(T'_k[1])] + 1$$

Quindi andrà incrementata la locazione gaps[cur_rank].

Successivamente, alla lettura di un nuovo carattere $T'_k[i]$, $2 \leq i \leq |T'_k|$, ci si troverà a valutare il suffisso di T_k: $T'_k[i]T'_k[i-1] \dots \$_k$ che si nota essere il risultato della concatenazione del carattere appena letto con il suffisso precedente di cui si conosce cur_rank. Verrà quindi iterata la procedura che trasforma cur_rank dall'attuale a quello relativo al nuovo suffisso ottenuto per concatenazione con il carattere appena letto. Tale indice indicherà quale locazione di gaps incrementare.

Per effettuare questa trasformazione si calcola, come prima, l'indice di partenza dei suffissi di T_{j+1} che iniziano con $T'_k[i]$, quindi l'indice in colonna F di B_{j+1} in cui inizia a comparire $T'_k[i]$:

$$\text{cfirst} = \text{FC}[\text{Map}(T'_k[i])] - \text{Occ}[\text{Map}(T'_k[i])] + 1$$

Se cur_rank del suffisso precedente è 1, allora $\text{cur_rank} = \text{cfirst}$ e dopo aver incrementato gaps[cur_rank] sarà possibile proseguire all'iterazione successiva. Questo perché il suffisso precedente era il più piccolo fra tutti quelli di T_{j+1}, quindi il nuovo suffisso sarà il più piccolo fra quelli di T_{j+1} che iniziano per $T'_k[i]$.

Se cur_rank del suffisso precedente è maggiore di 1, allora sarà necessario calcolare $\text{rank}(T'_k[i], \text{cur_rank} - 1)$ sulla colonna L di B_{j+1}; il risultato indica in che posizione a partire da cfirst si posiziona il nuovo suffisso.

$$\text{cur_rank} = \text{cfirst} + \text{rank}(T'_k[i], \text{cur_rank} - 1)$$

Si procede ad incrementare gaps[cur_rank], e quindi alla iterazione successiva.

Al termine dell'operazione c'è stato un incremento per ogni carattere letto, quindi è verificata la condizione posta all'inizio:

$$\sum_{i=1}^{|B_{j+1}|+1} \text{gaps}[i] = |T'_1 T'_2 \dots T'_j|$$

L'operazione di rilettura dei testi si ripete m volte, e all'interazione $1 < j \leq m$ vengono riletti nuovamente i primi j testi. Questo come anticipato, rende la complessità in tempo $O(nm^2)$. In spazio invece è necessario contenere la BWT B_{j+1}, una struttura idonea all'array gaps, ed una struttura idonea ad effettuare in tempo accettabile le operazioni rank durante la creazione di gaps. Come è stato anticipato esistono due versioni di questo algoritmo: `bwtc` e `bwtcw`. Il primo utilizza un sistema di conteggi parziali accessibili direttamente, a cui va aggiunto il conteggio sequenziale solo nelle posizioni adiacenti a quella cercata, per un totale di $6n$ bytes di memoria. Il secondo utilizza una struttura dati di tipo wavelet tree per un totale di $3n$ bytes di memoria.

4 Strutture dati per rank

Come anticipato, `bwtc` e `bwtcw` differiscono solamente per la struttura dati dedicata all'operazione rank, la cui implementazione banale richiede tempo $O(n)$ con n lunghezza dell'input.

Di seguito sono presentate le due alternative, entrambe richiedono tempo indipendente alla dimensione dell'input e dipendente alla dimensione dell'alfabeto.

4.1 Wavelet Tree

Il Wavelet tree è una struttura ad albero binario completo che permette di effettuare l'operazione rank su una stringa $S \in \Sigma^*$, $|S| = n$, $|\Sigma| = \sigma$, in tempo $O(\log \sigma)$.

L'albero possiede σ nodi foglia e $\sigma - 1$ nodi interni. Ad ogni nodo foglia è assegnato un carattere di Σ; è possibile risalire al nodo corrispondente un carattere tramite la funzione

$$\text{MapLeaf} : \Sigma \rightarrow \mathbb{N}.$$

L'indice dei nodi o vertici $v_i \; \forall \; 1 \leq i \leq 2\sigma - 1$ è assegnato in maniera sequenziale partendo dalla radice v_1, e proseguendo per livelli da sinistra a destra. A ciascuno dei soli nodi interni $v_i \; \forall \; 1 \leq i \leq \sigma - 1$ è associata una etichetta $t_i \in \{0, 1\}^*$.

Il figlio sinistro di un nodo interno v_i è indicato come $s(n_i)$ e corrisponde al nodo v_{2i}. Il figlio destro di un nodo interno invece è indicato come $d(v_i)$ e corrisponde al nodo v_{2i+1}. Il padre di un nodo $v_i \; \forall \; 2 \leq i \leq 2\sigma - 1$ è indicato come $p(v_i)$ e corrisponde al nodo $v_{\lfloor \frac{i}{2} \rfloor}$.

La costruzione delle etichette dei nodi interni è realizzata scorrendo i caratteri di S. Ad ogni carattere $S_i \; \forall \; 1 \leq i \leq n$ viene calcolato $N = v_{\text{MapLeaf}(S_i)}$. Iniza ora una procedura iterativa: viene concatenato un carattere a $t_{p(N)}$, l'etichetta appartenente al nodo padre di N. Se ci si trova in un nodo con indice pari viene concatenato "1", altrimenti viene concatenato "0". Si procede effettuando l'assegnamento $N = p(N)$ ed iterando sino al raggiungimento del nodo radice. Al nodo radice quindi sarà associata un'etichetta $t_1, |t_1| = n$ poiché durante l'iterazione, ad ogni occorrenza S_i è stata raggiunta la radice e concatenato il carattere di provenienza.

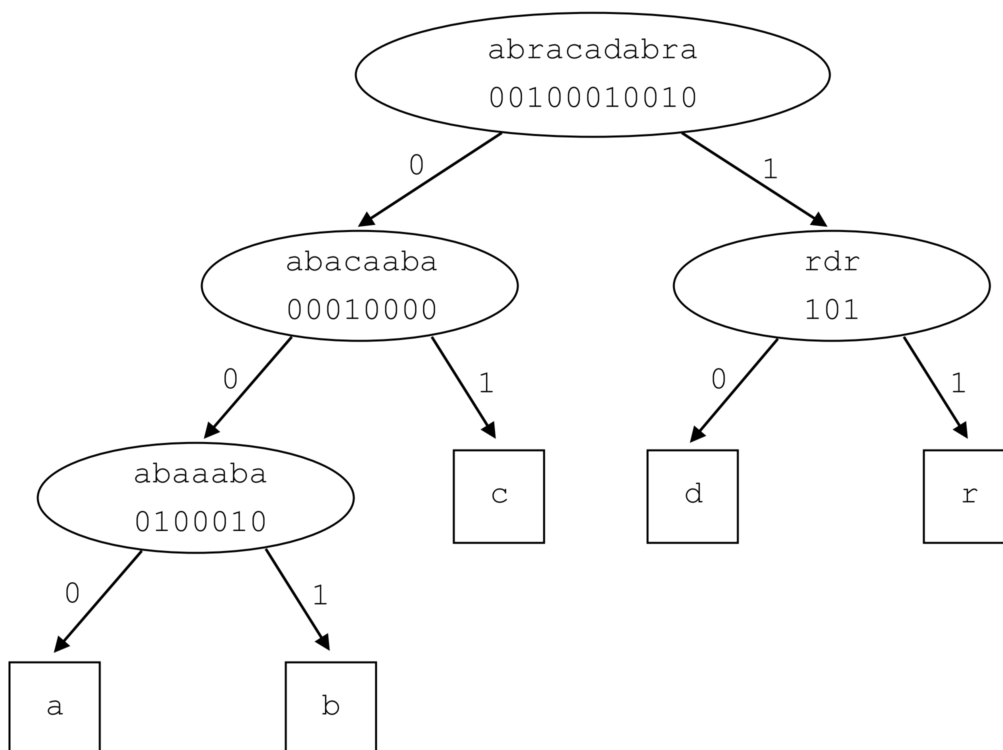


Figura 1 Wavelet tree della stringa "***abracadabra***".

Vengono ora introdotte le operazioni $\text{rank0}(v, i)$ e $\text{rank1}(v, i)$. Rispettivamente calcolano il numero di occorrenze di "1" e "0" sul nodo v sino alla posizione i.

L'operazione $\text{rank}(c, i)$ è eseguita calcolando un array di nodi P che corrisponde al percorso da v_1 a $v_{\text{MapLeaf}(c)}$; $|P| = l$. Partendo da $P[j = 0] = v_1$, si itera sino $N[l - 1]$, cioè sino ad avere $N[i]$ un nodo interno, e sino a che $i > 0$:

$$\begin{cases} i = \text{rank0}(N[j], i) \;\Leftrightarrow\; N[j+1] = s(N[j]) \\ i = \text{rank1}(N[j], i) \;\Leftrightarrow N[j+1] = d(N[j]) \end{cases}$$

Il risultato dell'ultima operazione rank0 o rank1 è il risultato finale.

Se in una iterazione $i = 0$ si termina ritornando il valore 0.

L'iterazione compie $O(\log \sigma)$ passi, che corrisponde all'altezza dell'albero. Le singole operazioni rank0 o rank1 possono essere $O(1)$ utilizzando contatori parziali delle occorrenze.

L'utilizzo in spazio è $O(n \log \sigma)$ bit. I contatori parziali sono lineari con la lunghezza di S, e ciascuna occorrenza di S occupa un bit in ogni nodo di un percorso dalla radice alla foglia associata.

4.2 BWT32

La struttura BWT32 è una struttura dati che contiene una BWT e ne facilita le operazioni rank. E' possibile realizzare questa struttura su una BWT $B \in \Sigma^*$, $|B| = n$, $|\Sigma| = \sigma < 2^8$ utilizzando due array.

Un array di n interi a 32 bit $\mathrm{BWT32}[1 \dots n]$.

Un array di dimensione $\left\lceil \frac{n}{2^{24}} \right\rceil$ contenente locazioni formate da array $[1 \dots \sigma]$ di interi a 64 bit chiamato SB.

Ciascuna locazione di BWT32 viene divisa in due parti: una di 8 bit necessari a contenere un carattere di B, e l'altra di 24 per gestire i conteggi. Si indicano $\mathrm{CC}[i]$ e $\mathrm{CB}[i]$ rispettivamente il valore destinato al carattere e quello destinato ai conteggi nella locazione $\mathrm{BWT32}[i]$.

I caratteri in questa struttura sono valori interi compresi nell'intervallo $0 \dots \sigma - 1$.

La creazione è effettuata inserendo i caratteri B_i in $\mathrm{CC}[i]$ $\forall\, 1 \le i \le n$ e creando un array SB, $|SB| = \left\lceil \frac{n}{2^{24}} \right\rceil$. Nella locazione $\mathrm{SB}[1]$ e nelle prime σ locazioni di CB è inserito il valore 0.

Lo spazio riservato ai conteggi delle successive locazioni $\mathrm{CB}[i]$ $\forall\, \sigma < i \le n$, contiene il numero di occorrenze del carattere $i \bmod \sigma$ in $B[\left\lfloor \frac{i}{2^{24}} \right\rfloor,\ i]$.

Tale valore, se $\left\lceil \frac{i}{2^{24}} \right\rceil = \left\lceil \frac{i-\sigma}{2^{24}} \right\rceil$, può essere ricavato sommando $\mathrm{CB}[i - \sigma]$ al numero di occorrenze di $i \bmod \sigma$ nella sottostringa $B[i - \sigma + 1,\ i]$.

Se invece $\left\lceil \frac{i}{2^{24}} \right\rceil \neq \left\lceil \frac{i-\sigma}{2^{24}} \right\rceil$ viene inserito nella locazione $\mathrm{SB}[\left\lceil \frac{i}{2^{24}} \right\rceil]\ [(i \bmod \sigma) + 1]$ il numero di occorrenze del carattere $i \bmod \sigma$ dall'inizio del file, che corrisponde alla somma:

$$\mathrm{SB}[\left\lceil \frac{i}{2^{24}} \right\rceil - 1]\ [(i \bmod \sigma) + 1] + \mathrm{CB}[i - \sigma]$$

Il valore contenuto in $\mathrm{CB}[i]$ sarà semplicemente il numero di occorrenze di $i \bmod \sigma$ in $B[i - \sigma + 1,\ i]$.

L'operazione $\mathrm{rank}(c, i)$ sarà ottenuta trovando il valore

$$k = \mathrm{MAX}_j\ (1 \leq j \leq i \wedge j \bmod \sigma = c)$$

Si procede poi a trovare il numero di occorrenze di c in $B[1, k]$

$$\mathrm{SB}[\left\lceil \frac{k}{2^{24}} \right\rceil]\ [c + 1] + \mathrm{CB}[k]$$

Se $k = j$ il risultato corretto è già stato ottenuto. Altrimenti se $k < j$ bisogna contare le occorrenze di c in $B[k + 1, i]$ in tempo $O(\sigma)$ e sommarle al valore ottenuto.

La procedura impiega complessivamente tempo $O(\sigma)$.

5 TEST

Come anticipato nella sezione 3.2 il costo per computare la BWT di una collezione di m stringhe al più lunghe n è in tempo $O(nm^2)$ e in spazio $6n$ (BWTC) oppure $3n$ (BWTCW). Per le collezioni ottenute da un sequenziamento di DNA, in cui il numero di stringhe ottenute è nell'ordine delle decine di milioni, il termine al quadrato nella complessità in tempo risulta eccessivo. Inoltre in questi casi non sarà possibile sfruttare pienamente la capacità delle moderne memorie interne, poiché le stringhe appartenenti alla collezione sono lunghe al più poche centinaia di caratteri. L'utilizzo della memoria interna risulta quindi limitato a qualche KB.

Per far fronte a queste limitazioni si è deciso di impostare un limite alla memoria interna disponibile M e di raggruppare le stringhe della collezione in blocchi di dimensione massima $n = {}^{M}/_{6}$ per BWTC e $n = {}^{M}/_{3}$ per BWTCW. I blocchi sono ottenuti concatenando le stringhe precedenti, più piccole, sino al raggiungimento del limite imposto. Il numero di blocchi, supponendo N la dimensione della collezione, sarà $m = \lceil {}^{N}/_{2\text{GB}} \rceil$ per BWTC e $m = \lceil {}^{N}/_{4\text{GB}} \rceil$ per BWTCW. Il termine m, che compare al quadrato nella complessità in tempo, viene così ridotto e l'utilizzo di memoria viene portato al limite imposto.

La concatenazione delle stringhe avviene utilizzando un carattere separatore che non deve comparire altrove nelle stringhe della collezione originale. Si crea così una limitazione all'alfabeto Σ delle stringhe originali: $|\Sigma| \leq 255$.

Questo approccio inoltre, durante la fase di ricerca di un pattern, rende molto costoso risalire all'indice della stringa che lo contiene nella collezione originale. Sarà possibile solo risalire all'indice del blocco che contiene, fra le altre, la stringa risultato del match. Per le collezioni ottenute da un sequenziamento di DNA questa non è una limitazione considerevole. Per le collezioni di articoli invece può esserlo.

Nei test si è impostato $M = 12\text{GB}$ e quindi le dimensioni dei blocchi sono fissate a $n = 2\text{GB}$ per BWTC e $n = 4\text{GB}$ per BWTCW.

Per ottenere le BWT dei singoli blocchi lunghi al più n, necessarie per l'esecuzione di BWTC o BWTCW, si è utilizzato l'algoritmo semi-external presentato in (Beller, et al.,

2013), qui chiamato `construct_bwt`, che utilizza spazio n. Il computer su cui sono state eseguite le prove ha un microprocessore con 8 cores, quindi sarà possibile eseguire parallelamente più istanze del programma sino al raggiungimento del limite di memoria imposto a 12GB. Quindi è possibile creare la BWT di un massimo di 6 blocchi da 2GB nel caso di BWTC e sino a 3 blocchi da 4GB nel caso di BWTCW.

Le BWT ottenute, insieme ai blocchi che le hanno originate saranno l'input a BWTC o BWTCW.

I test effettuati non solo comparano BWTC a BWTCW, ma danno una panoramica il più completa possibile sulle alternative per la creazione della BWT delle collezioni. Di seguito sono presentate le alternative trattate.

Per collezioni la cui dimensione massima è 12GB è possibile eseguire direttamente `construct_bwt` sull'intera collezione poiché l'utilizzo di memoria non eccede i limiti imposti.

Per le sole collezioni di sequenze di DNA della stessa lunghezza è possibile utilizzare l'algoritmo presentato in (Bauer, et al., 2011) qui chiamato `BEETL_BWT`.

I test comparano quindi l'esecuzione combinata di `construct_bwt` sui blocchi e BWTC/BWTCW sulla collezione dei blocchi, `construct_bwt` sull'intera collezione (solo dove possibile), e `BEETL_BWT` solo sulle collezioni che rispettano i suoi limiti. Tutti i test sono stati effettuati limitando la memoria utilizzata a 12GB.

Le collezioni per i test sono le seguenti:

- seq150: collezione di sequenze di DNA tutte lunghe 150 caratteri
- seq300: collezione di sequenze di DNA tutte lunghe 300 caratteri
- wiki: collezione di articoli Wikipedia (Wikimedia Foundation, Inc., s.d.)

Di seguito il numero di stringhe che compongono le collezioni utilizzate nei test:

Collezione	8 GB	12 GB	16 GB	20 GB
seq150	57 266 230	85 899 344	106 666 659	---
seq300	27 391 865	41 070 105	53 333 326	66 666 670
wiki	1 500 422	2 618 356	3 425 952	4 722 215

L'unica alternativa valida per generare la BWT in ognuno di questi casi è l'esecuzione combinata di `construct_bwt` sui blocchi e BWTC/BWTCW sulla collezione dei blocchi. Di seguito i grafici e i commenti.

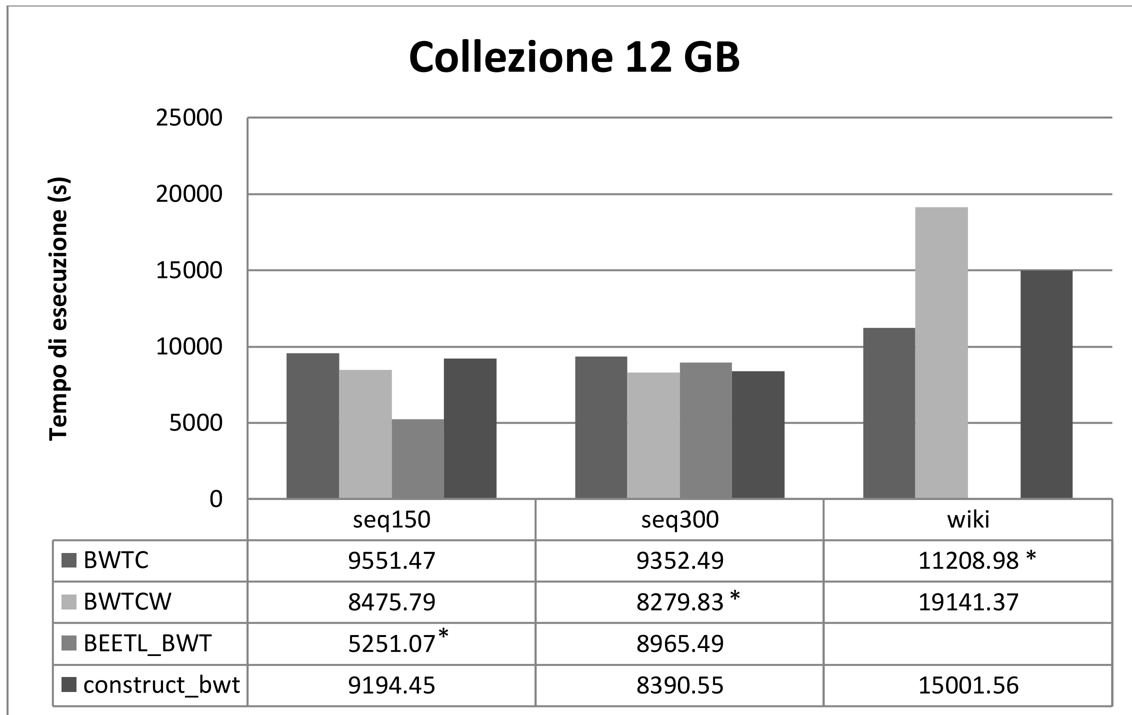


Figura 2

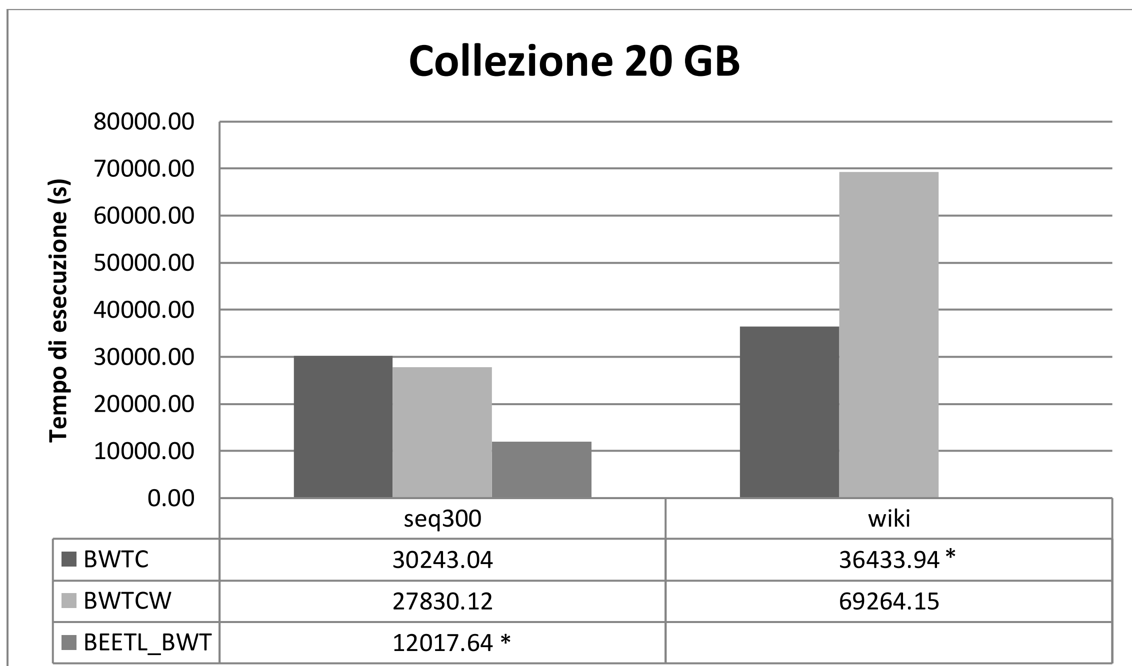
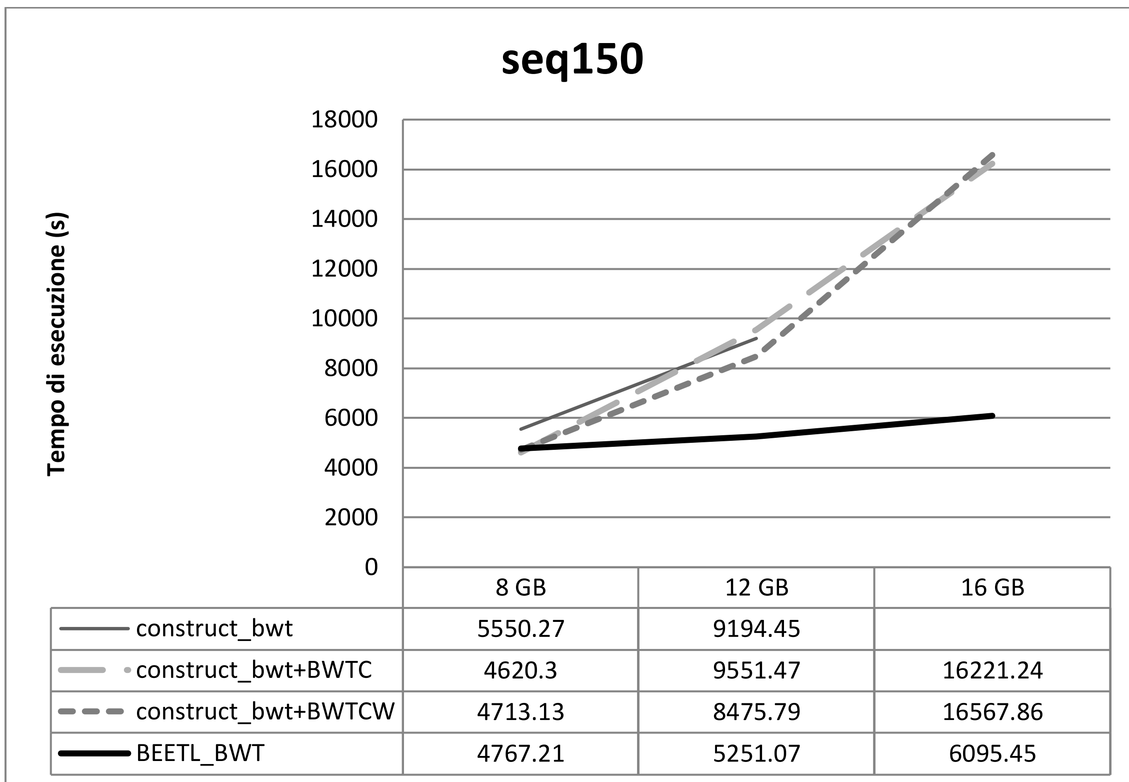
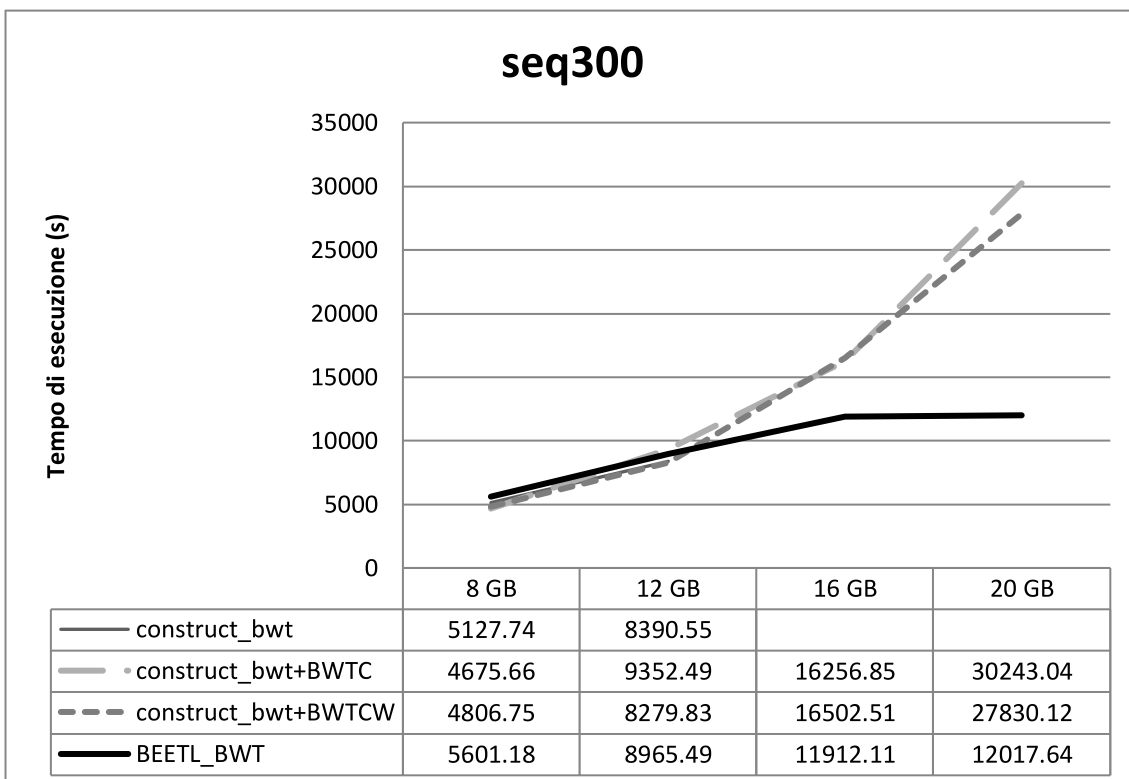


Figura 3



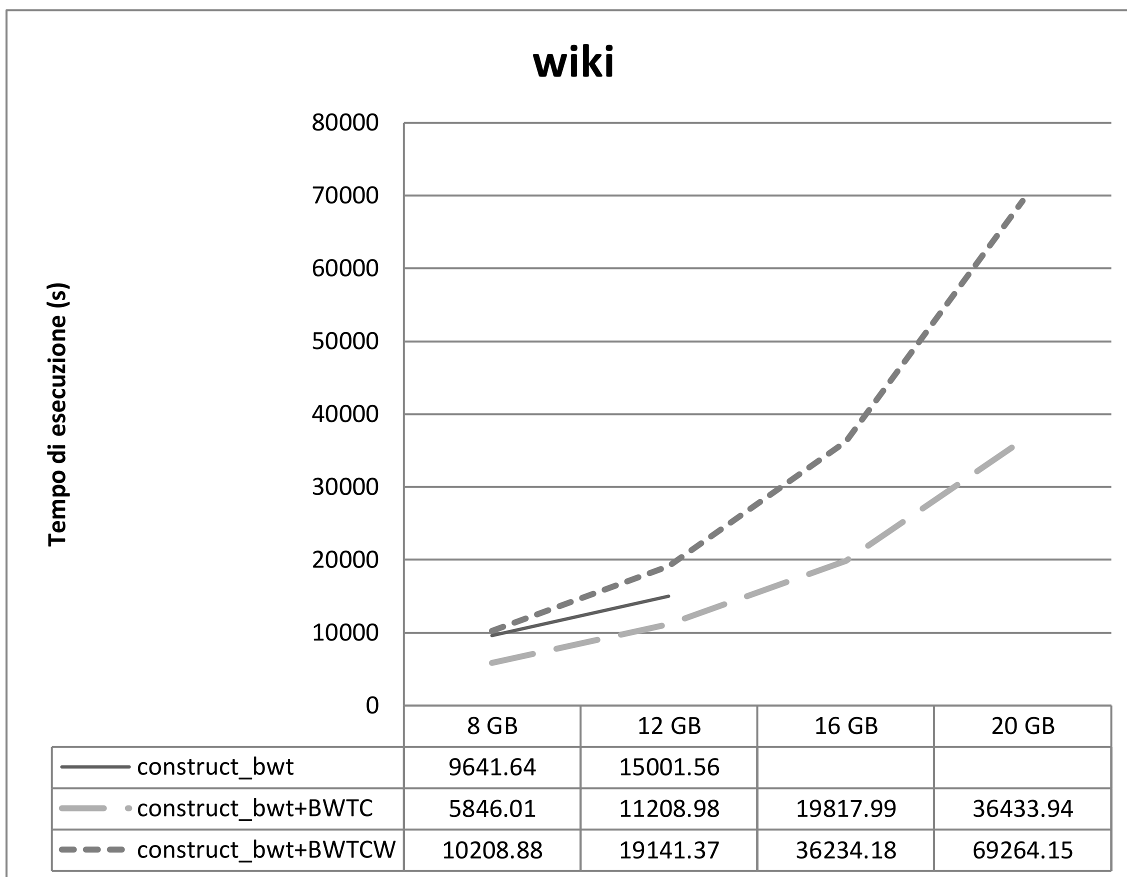
	8 GB	12 GB	16 GB
construct_bwt	5550.27	9194.45	
construct_bwt+BWTC	4620.3	9551.47	16221.24
construct_bwt+BWTCW	4713.13	8475.79	16567.86
BEETL_BWT	4767.21	5251.07	6095.45

Figura 4



	8 GB	12 GB	16 GB	20 GB
construct_bwt	5127.74	8390.55		
construct_bwt+BWTC	4675.66	9352.49	16256.85	30243.04
construct_bwt+BWTCW	4806.75	8279.83	16502.51	27830.12
BEETL_BWT	5601.18	8965.49	11912.11	12017.64

Figura 5



	8 GB	12 GB	16 GB	20 GB
construct_bwt	9641.64	15001.56		
construct_bwt+BWTC	5846.01	11208.98	19817.99	36433.94
construct_bwt+BWTCW	10208.88	19141.37	36234.18	69264.15

Figura 6

Dalla Figura 2 si nota che la soluzione proposta in questa tesi è la migliore per collezioni di sequenze di DNA lunghe e di articoli wiki. Nel caso di collezioni di DNA corte il più veloce è `BEETL_BWT`.

La Figura 3 mostra `BEETL_BWT` il più veloce su collezioni di sequenze lunghe ma si nota che il tempo cresce al crescere della lunghezza delle stringhe. Per quanto riguarda le collezioni generiche l'unica alternativa possibile è quella presentata in questa tesi, e data la dimensione dell'alfabeto risulta migliore BWTC.

I grafici successivi mostrano l'andamento, all'aumentare della dimensione della collezione in input, del tempo di esecuzione dei vari tool a disposizione.

Nella pagina seguente è mostrata la tabella di tutte le prove eseguite.

BWT di Collezione di stringhe con limite di memoria 12 GB								
Dimensione	Modalità	Blocchi	BWT	Collezione	Tempo bwt	Collezione	Tempo collez.	TOTALE
8 GB	Blocchi + BWT + BWTC/W	2 GB 4 files	construct_bwt	wiki	2301.95	bwtc	3544.06	5846.01
				seq300	1699.45		2976.21	4675.66
				seq150	1630.73		2989.57	4620.3
		4 GB 2 files		wiki	4313.27	bwtcw	5895.61	10208.88
				seq300	2706.21		2100.54	4806.75
				seq150	2805.07		1908.06	4713.13
	BWT completa	8 GB 1 file		wiki				9641.64
				seq300				5127.74
				seq150				5550.27
	BWT collezione		BEETL_BWT	seq300				5601.18
				seq150				4767.21
12 GB	Blocchi + BWT + BWTC/W	2 GB 6 files	construct_bwt	wiki	2389.09	bwtc	8819.89	11208.98
				seq300	1788.26		7564.23	9352.49
				seq150	1715.03		7836.44	9551.47
		4 GB 3 files		wiki	4317.57	bwtcw	14823.8	19141.37
				seq300	2796.3		5483.53	8279.83
				seq150	2826.02		5649.77	8475.79
	BWT completa	12 GB 1 file		wiki				15001.56
				seq300				8390.55
				seq150				9194.45
	BWT collezione		BEETL_BWT	seq300				8965.49
				seq150				5251.07
16 GB	Blocchi + BWT + BWTC/W	2 GB 8 files	construct_bwt	wiki	4821.18	bwtc	14996.81	19817.99
				seq300	3378.25		12878.6	16256.85
				seq150	3212.94		13008.3	16221.24
		4 GB 4 files		wiki	8591.47	bwtcw	27642.71	36234.18
				seq300	5618.91		10883.6	16502.51
				seq150	5763.45		10804.41	16567.86
	BWT collezione	16 GB 1 file	BEETL_BWT	seq300				11912.11
				seq150				6095.45
20 GB	Blocchi + BWT + BWTC/W	2 GB 10 files	construct_bwt	wiki	4912.40	bwtc	31521.54	36433.94
				seq 300	3418.64		26824.4	30243.04
		4 GB 5 files		wiki	9001.05	bwtcw	60263.1	69264.15
				seq 300	5793.60		22036.52	27830.12
	BWT collezione	20 GB 1 file	BEETL_BWT	seq300				12017.64

6 CONCLUSIONI

In questa tesi è stato presentato un algoritmo per la creazione della BWT di una collezione di documenti in due versioni che differiscono fra loro per la struttura dati utilizzata nell'operazione rank. Dai test svolti nella precedente sezione si nota che per collezioni di documenti aventi un alfabeto di pochi caratteri si ha una maggiore velocità nella versione BWTCW che utilizza la struttura wavelet tree, presentata nella sezione 4.1. Per collezioni di documenti aventi un alfabeto più numeroso invece si ha una maggiore velocità nella versione BWTC che utilizza la struttura BWT32, presentata nella sezione 4.2.

Nei test è stato fissato un limite di 12 GB alla memoria disponibile a ciascun programma eseguito e si è confrontato il tempo di esecuzione della soluzione proposta con:

- `construct_bwt` (Beller, et al., 2013) solo con collezioni di dimensione massima pari alla memoria disponibile
- `BEETL_BWT` (Bauer, et al., 2011) solo per collezioni di documenti che rispettano le limitazioni di alfabeto di 5 caratteri e di stessa lunghezza di ogni documento.

Dai risultati di questo confronto si nota che `BEETL_BWT` è più veloce su collezioni di sequenze corte ed il tempo di esecuzione sembra crescere molto al crescere della lunghezza delle sequenze. La soluzione proposta in questa tesi di abbinare l'esecuzione di `construct_bwt` a quella di BWTC/BWTCW, è l'unica alternativa attualmente disponibile per creare la BWT di una collezione di documenti di qualunque lunghezza e con alfabeto sino a 255 caratteri.

La versione BWTC è in grado di computare anche il suffix array della collezione di documenti, anche se nei test non è menzionato, ed è l'unico strumento in grado di farlo.

La versione BWTCW è ancora in una fase iniziale di sviluppo; per questo motivo è possibile ottimizzare ulteriormente l'esecuzione in modo da favorire migliori prestazioni.

BIBLIOGRAFIA

Bauer, M. J., Cox, A. & Rosone, G., 2011. Lightweight BWT Construction for Very Large String Collections. *Proceedings of CPM,* pp. 219-231.

Beller, T., Zwerger, M., Gog, S. & Ohlebush, E., 2013. Space-Efficient Construction of the Burrows-Wheeler Transform. *SPIRE,* pp. 5-13.

Burrows, M. & Wheeler, D., 1994. A block sorting lossless data compression algorithm. *Digital Equipment Corporation.*

Ferragina, P., Gagie, T. & Manzini, G., 2012. Lightweight data indexing and compression in external memory. *Algorithmica 63,* pp. 707-730.

Ferragina, P. & Manzini, G., 2000. Opportunistic data structures with applications. *Proc. IEEE Symposium on Foundations of Computer Science,* pp. 390-398.

Gonnet, G., Baeza-Yates, R. & Snider, T., 1992. New indices for text: PAT trees and PAT arrays. *Information retrieval: data structures and algorithms.*

IEEE Standards Association, s.d. *POSIX - Austin Joint Working Group.* [Online] Available at: http://standards.ieee.org/develop/wg/POSIX.html

Kärkkäinen, J., Sanders, P. & Burkhardt, S., 2006. Linear work suffix array construction. *Journal of the ACM,* 53(6), p. 918–936.

Kim, D., Sim, J., Park, H. & Park, K., 2005. Constructing suffix arrays in linear time. *J. Discr. Alg. 3,* pp. 126-142.

Ko, P. & Aluru, S., 2005. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms,* 3(2-4), pp. 143-156.

Manber, U. & Myers, G., 1990. Suffix arrays: a new method for on-line string searches. *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms,* pp. 319-327.

Manzini, G., 2004. Two Space Saving Tricks for Linear Time LCP Array Computation. *Dipartimento di Informatica, Università del Piemonte Orientale.*

Okanohara, D. & Sadakane, K., 2009. A linear-time Burrows-Wheeler transform using induced sorting. *SPIRE 2009. LNCS,* Volume 5721, pp. 90-101.

Weiner, P., 1973. Linear pattern matching algorithms. *14th Annual IEEE Symposium on Switching and Automata Theory,* pp. 1-11.

Wikimedia Foundation, Inc., s.d. *Wikimedia Downloads.* [Online] Available at: http://dumps.wikimedia.org/