Università degli Studi di Milano - Bicocca
Scuola di Scienze
Dipartimento di Informatica, Sistemistica e Comunicazione
Corso di Laurea Magistrale in Informatica

Algoritmi per la trasformata di Burrows-Wheeler Posizionale con compressione run-length, RLPBWT

Relatore: *Prof.ssa Raffaella Rizzi*

Correlatore:

Tesi di Laurea Magistrale di:
Davide Cozzi
Matricola 829827

Anno Accademico 2021-2022

Abstract

Indice

Capitolo 1

Introduzione

Capitolo 2

Preliminari

In questo capitolo verranno specificate tutti i concetti fondamentali atti a comprendere i metodi usati in questa tesi che le motivazioni della stessa. In primis, verranno introdotte le *motivazioni biologiche*, al fine di dare uno scopo pratico agli algoritmi e alle strutture dati introdotte, per procedere poi con un breve excursus dei fondamenti teorici presenti allo stato dell'arte.
Dal punto di vista tecnico verranno quindi introdotti i *bit vector* e gli *straight-line programs*, fondamentali sia per la *run-length encoded Burrows-Wheeler Transform*, introdotta qui insieme alla versione classica della trasformata, che per la mia variante relativa alla *Position Burrows-Wheeler Transform*, che verrà anch'essa trattata nel capitolo.

2.1 Motivazioni Biologiche

2.2 Bit vector

TUTTE LE TABELLE VANNO VERIFICATE!!!
Nell'ambito delle *strutture dati succinte*, una delle strutture dati principali, ormai sviluppatasi in molteplici varianti, è quella denominata **bit vector**.

Definizione 1. *Si definisce un* ***bit vector*** B *come un array di lunghezza* n*, popolato da elementi binari. Formalmente si ha quindi:*

$$B[i] = \{0, 1\},\ \forall i\ t.c.\ 0 \leq i < n$$

In alternativa si potrebbe avere come formalismo:

$$B[i] = \{\bot, \top\},\ \forall i\ t.c.\ 0 \leq i < n$$

Nel corso degli ultimi anni si sono sviluppate diverse varianti dei *bit vector*, finalizzate ad offrire diversi costi di complessità spaziale e diversi tempi computazionali per le principali funzioni offerte.
Il primo vantaggio di questa struttura dati, nelle varianti che si andranno poi a nominare, è quella di garantire *random access* in tempo costante pur sfruttando varie tecniche per la memorizzazione efficiente della stessa in memoria. Lo spazio necessario per l'implementazione, presente in *SDSL* [1], delle principali varianti è visualizzabile in tabella B.1. Il secondo vantaggio consiste nel fatto che i *bit vector* permettono l'implementazione efficiente di due funzioni:

1. la **funzione rank**
2. la **funzione select**

Tali funzioni, al costo di $\mathcal{O}(n)$ bit aggiuntivi, possono essere supportate in tempo costante. Questo è però un discorso prettamente teorico, infatti si vedrà come, nelle implementazioni in *SDSL*, le complessità temporali delle due funzioni non siano mai entrambe costanti.

2.2.1 Funzione rank

La prima funzione che si approfondisce è la **funzione rank**. Tale funzione permette di calcolare il *rango* di un dato elemento del bit vector B, $|B| = n$. In altri termini, data una certa posizione i del *bit vector*, la funzione restituisce il numero di 1 presenti fino a quella data posizione, escluda. Più formalmente si ha:

$$rank_B(i) = \sum_{k=0}^{k<i} B[k],\ \forall i \text{ t.c. } 0 \leq i < n$$

Come detto, da un punto di vista teorico, al costo di $\mathcal{O}(n)$ bit aggiuntivi in memoria tale funzione sarebbe supportata in tempo $\mathcal{O}(1)$. Questo però non risulta vero nelle principali implementazioni. La complessità temporale varia infatti a seconda dell'implementazione, anche in conseguenza al fatto che si ha una quantità diversa di bit aggiuntivi salvati in memoria. La tabella con le complessità temporali stimate della *funzione rank*, per le varianti di *bit vector* implementate in *SDSL*, è visualizzabile in tabella B.3.

2.2.2 Funzione select

La seconda funzione fondamentale è la **funzione select**. Tale funzione permette, dato un valore intero i, di calcolare l'indice dell'i-esimo valore pari a 1 nel *bit vector* B, tale che $|B| = n$. Più formalmente si ha che:

$$select_B(i) = \min\{j < n \mid rank_B(j+1) = 1\},\ \forall i \text{ t.c. } 0 < i \leq rank_B(n)$$

Anche in questo caso vale lo stesso discorso fatto per la *funzione rank* in merito alla complessità temporale e ai bit aggiuntivi. La tabella con le complessità temporali stimate della *funzione select*, per le varianti di *bit vector* implementate in *SDSL*, è visualizzabile in tabella B.3.
Si può quindi vedere un semplice esempio esplicativo.

Esempio 1. *Ipotizziamo di avere il seguente bit vector B, di lunghezza $n = 14$:*

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*
1	0	0	**1**	0	**1**	0	**1**	0	**1**	0	0	**1**	0

Si ha che, per esempio:

$$\texttt{rank}(6) = 3$$

$$\texttt{select}(5) = 9$$

Da un punto di vista pratico si vedrà nel corso di questa tesi come l'uso di tali strutture, nel dettaglio l'uso dei *bit vector plain* e dei *bit vector sparsi*, sia fondamentale sia nella costruzione delle *strutture run-length encoded* che nelle interrogazioni alle stesse.

2.3 Straight-Line Program

Nel contesto *bioinformatico* una delle principali problematiche è la gestione di testi molto estesi. Pensiamo, ad esempio, al caso umano. Il primo cromosoma, il più lungo tra i cromosomi umani, conta circa 247.249.719 *bps* (paia di basi), nonostante, è bene segnalare, l'uomo non sia affatto l'essere vivente con il genoma più esteso. Fatta questa breve premessa è facile comprendere l'importanza degli algoritmi e delle strutture dati atte alla compressione di testi.
Per questa tesi si è provveduto all'uso di uno di tali algoritmi di compressione, ovvero i **Straight-line programs (*SLPs*)**. Parlando in termini generici un *SLP* è una **grammatica context-free** che genera una e una sola parola [2]. Si parla, a causa di ciò, di **grammar-based compression**.

Definizione 2. *Sia dato un alfabeto finito Σ per i simboli terminali. Sia data una stringa $s = a_1, a_2, \ldots, a_n \in \Sigma^*$, lunga n e costruita sull'alfabeto Σ. Si ha quindi che $a_i \in \Sigma$, $\forall i$ t.c. $1 \leq i \leq n$, denotando con $alph(s) = \{a_1, a_2, \ldots a_n\}$ l'insieme dei simboli della stringa s.*
*Un **SLP** sull'alfabeto Σ è una grammatica context-free $\mathcal{A}$:*

$$\mathcal{A} = (\mathcal{V}, \Sigma, \mathcal{S}, \mathcal{P})$$

dove:

- $\mathcal{V}$ *è l'insieme dei simboli non terminali*
- Σ *è l'insieme dei simboli terminali*
- $\mathcal{S} \in \mathcal{V}$ *è il simbolo iniziale non terminale*
- $\mathcal{P}$ *è l'insieme delle produzioni, avendo che:*

$$\mathcal{P} \subseteq \mathcal{V} \times (\mathcal{V} \cup \Sigma)^*$$

Tale grammatica, per essere un SLP, deve soddisfare due proprietà:

1. *si ha una e una sola produzione* $(A, \alpha) \in \mathcal{P}$, $\forall A \in \mathcal{V}$ *con* $\alpha \in (\mathcal{V} \cup \Sigma)^*$ *(si noti che la produzione* (A, α) *può anche essere indicata con* $A \to \alpha$*)*
2. *la relazione* $\{(A, B) \mid (A, \alpha) \in \mathcal{P},\ B \in alph(\alpha)\}$ *è aciclica*

Si ha quindi che la grandezza dell'SLP è calcolabile come:

$$|\mathcal{A}| = \sum_{(A,\alpha) \in \mathcal{P}} |\alpha|$$

Si ha quindi che il linguaggio generato da un *SLP*, $\mathcal{A}$, consiste in una singola parola, denotata da $eval\mathcal{A}$.
A partire dall'*SLP* $\mathcal{A}$ si genera quindi un **albero di derivazione**, che nel dettaglio è un *albero radicato e ordinato*, dove la *radice* è etichettata con $\mathcal{S}$, ogni *nodo interno* è etichettato con un simbolo di $\mathcal{V} \cup \Sigma$ e ogni foglia è etichettata con un simbolo di Σ.
Si vede quindi un esempio chiarificatore [3].

Esempio 2. *Si prenda la seguente stringa:*

$$s = GATTAGATACAT\$GATTACATAGAT$$

Si potrebbe produrre il seguente SLP*:*

$$S \to ZWAY\$ZYAW$$

$$Z \to WX$$

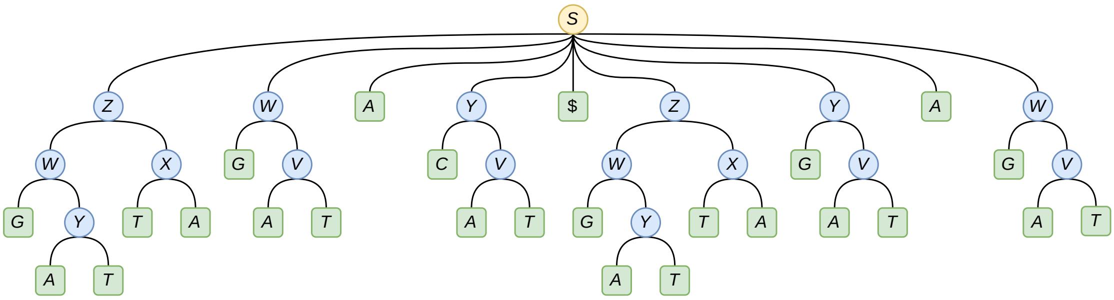
$$Y \to CV$$

$$X \to TA$$

$$W \to GV$$

$$V \to AT$$

Al quale corrisponde il seguente albero di derivazione*, dove il simbolo iniziale non terminante, ovvero la radice, è indicata con un cerchio giallo, i simboli non terminanti, ovvero i nodi interni, sono indicati dai cerchi blu mentre i simboli terminanti, ovvero le foglie, sono indicati dai quadrati verdi:*



Nel 2020, Gagie et al. [3], a cui si rimanda per approfondimenti, proposero una variante degli *SLPs* che garantisse miglioramenti prestazionali per il *random access* alla grammatica stessa. Sfruttando, ad esempio, i *bit vector sparsi* si è quindi potuto garantire *random access* su un testo T, tale che $|T| = n$, compresso tramite *SLP*, in tempo:

$$\mathcal{O}(\log n)$$

VERIFICARE BENE COSA SIA n.
L'uso di tale variante degli *SLP* è stato cruciale, come si vedrà più avanti in questa tesi, per la costruzione della versione run-length encoded sia della **Burrows-Wheeler Transform (*BWT*)** che della **Positional Burrows-Wheeler Transform (*PBWT*)**.

2.3.1 Longest Common Extension

Oltre a permettere un veloce *random access* alla testo compresso, la variante degli *SLPs* proposta da Gagie et al. permette di effettuare un'altra operazione in modo "veloce": le **Longest Common Extension (*LCE*) queries**.

Definizione 3. *Dato un testo T, tale che $|T| = n$, il risultato della* ***LCE query*** *tra due posizioni i e j, tali che $0 \leq i, j < n$, corrisponde al più lungo prefisso comune tra le sotto-stringhe che hanno come indice di partenza i e j, avendo quindi il più lungo prefisso comune tra $T[i : n-1]$ e $T[j : n-1]$.*

Sfruttando l'*SLP* del testo T è quindi possibile effettuare due *random access* al testo compresso, in i e j, per poi "risalire" l'albero al fine di computare il prefisso comune tra $T[i : n-1]$ e $T[j : n-1]$. Quindi il calcolo di una *LCE query* di lunghezza l è effettuabile in tempo:

$$\mathcal{O}\left(1 + \frac{l}{\log n}\right)$$

VERIFICARE BENE COSA SIA n.
L'implementazione della variante degli *SLPs* proposta da Gagie et al. è disponibile su *GitHub* al link: ShapedSlp.

2.4 Suffix Array

Nel 1976 Manber e Myers proposero una struttura dati per la memorizzazione di stringhe e la loro interrogazione, efficiente sia per l'aspetto temporale che spaziale, chiamata **Suffix Array (*SA*)** [4].

Definizione 4. *Dato un testo T, \$-terminato, tale che $|T| = n$, si definisce **suffix array** di T, denotato con SA_T, un array lungo n di interi, tale che $SA_T[i] = j$ sse il suffisso di ordine j, ovvero $T[SA_T[i] : n-1]$, è l'i-esimo suffisso nell'ordinamento lessicografico dei suffissi di T. In altri termini quindi il **suffix array** altro non è che una permutazione dell'intervallo di numeri interi $[0, n-1]$.*

Grazie a questa definizione si può quindi dire che, presi $i, i' \in \mathbb{N}$ tali che $0 \leq i < i' < n$ allora vale che, indicando con $<$ anche l'ordinamento lessicografico:

$$T[SA_T[i] : n-1] < T[SA_T[i'] : n-1]$$

Si vede quindi esempio chiarificatore.

Esempio 3. *Si prenda la stringa:*

$$s = \mathit{mississippi\$},\ \ |s| = 12$$

Si producono quindi i seguenti suffissi e il loro riordinamento:

Indice del suffisso	**Suffisso**
0	mississippi$
1	ississippi$
2	ssissippi$
3	sissippi$
4	issippi$
5	ssippi$
6	sippi$
7	ippi$
8	ppi$
9	pi$
10	i$
11	$

Indice del suffisso	**Suffisso**
11	$
10	i$
7	ippi$
4	issippi$
1	ississippi$
0	mississippi$
9	pi$
8	ppi$
6	sippi$
3	sissippi$
5	ssippi$
2	ssissippi$

Ottenendo quindi che:

$$SA_T = [11, 10, 7, 4, 1, 0, 9, 8, 6, 3, 5, 2]$$

L'uso del *suffix array* è spesso accompagnato dal **Longest Common Prefix**.

Definizione 5. *Si definisce il **Longest Common Prefix (**LCP**)** di un testo T, tale che $|T| = n$, denotato con LCP_T, come un array lungo $n+1$, contenente la lunghezza del prefisso comune tra ogni coppia di suffissi consecutivi nell'ordinamento lessicografico dei suffissi, quindi in SA_T. Più formalmente LCP_T è un array tale che, avendo $0 \leq i \leq n$ e indicando con $lcp(x, y)$ il più lungo prefisso comune tra le stringhe x e y:*

$$LCP_T[i] = \begin{cases} -1 & se\ i = 0 \vee i = n \\ lcp(T[SA_T[i-1]:n], T[SA_T[i]:n]) & altrimenti \end{cases}$$

Esempio 4. *Riprendendo l'esempio precedente si avrebbe quindi:*

Indice	$\mathbf{SA_T}$	$\mathbf{LCP_T}$	**Suffisso**
0	11	-1	$
1	10	0	i$
2	7	1	ippi$
3	4	1	issippi$
4	1	4	ississippi$
5	0	0	mississippi$
6	9	0	pi$
7	8	1	ppi$
8	6	0	sippi$
9	3	2	sissippi$
10	5	1	ssippi$
11	2	3	ssissippi$
12	-	-1	-

Senza entrare in ulteriori dettagli relativi all'algoritmo di pattern matching tramite *SA* e *LCP*, in quanto non centrali per il resto della trattazione, risulta comunque interessante riportare le complessità temporali. Si ha quindi che per l'algoritmo di query su *SA* senza l'uso dell'*LCP* si ha, per un testo lungo n e un pattern lungo m:

$$\mathcal{O}(m \log n)$$

Con l'uso dell'*LCP* questo si riduce a:

$$\mathcal{O}(m + \log n)$$

Per ulteriori approfondimenti si rimanda al testo di Gusfield [5].

2.5 Trasformata di Burrows-Wheeler

Introdotta nel 1994 da Burrows e Wheeler con lo scopo di comprimere testi, la **Burrows-Wheeler Transform** [6] è divenuta ormai uno standard nel campo dell'*algoritmica su stringhe* e della *bioinformatica*, grazie ai suoi molteplici vantaggi sia dal punto di vista della complessità temporale che da quello della complessità spaziale.
Nel dettaglio la BWT è una *trasformata reversibile* che permette una *compressione lossless*, quindi senza perdita d'informazione. Tale trasformazione vien costruita a partire dal riordinamento dei caratteri del testo in input, fattore che ha portato all'evidenza per cui caratteri uguali tendono ad essere posti consecutivamente all'interno della stringa prodotta dalla trasformata.

Definizione 6. *Dato un testo T \$-terminato, tale che $|T| = n$, si definisce la* ***Burrows-Wheeler Transform (BWT)*** *di T, denotata con BWT_T, come un array di caratteri lungo n dove l'elemento i-esimo è il carattere che precede l'i-esimo suffisso T nel riordinamento lessicografico. Più formalmente si ha che, con $0 \leq i < n$:*

$$BWT_T[i] = \begin{cases} T[SA_T[i] - 1] & se\ SA_T[i] \neq 1 \\ \$ & altrimenti \end{cases}$$

In termini più pratici, la BWT di un testo è calcolabile riordinando lessicograficamente tutte le possibili **rotazioni** del testo T.

Definizione 7. *Si definisce* ***rotazione i-esima****, denotata con $rot_T(i)$ di un testo T, tale che $|T| = n$, come la stringa ottenuta dalla concatenazione del suffisso i-esimo con la restante porzione del testo. Più formalmente si ha che, avendo $0 \leq i < n$:*

$$rot_T(i) = T[i : n - 1] \cdot T[0 : i - 1]$$

Data questa definizione quindi la BWT del testo T risulta essere l'ultima colonna della matrice che si ottiene riordinando tutte le *rotazioni* di T, che altro non sono che i suffissi già riordinati per il calcolo del SA a cui viene concatenata la parte restante del testo.
Un altro array spesso utilizzato insieme alla BWT è il cosiddetto **array F**, lungo $|T|$, che altro non è che l'array formato dalla prima colonna della matrice delle rotazioni. In termini ancora più semplicistici l'array F è banalmente l'array formato dal riordinamento lessicografico dei caratteri del testo T.
Per chiarezza si vede un esempio.

Esempio 5. *Si prenda la stringa:*

$$s = mississippi\$, \ \ |s| = 12$$

Si produce la seguente matrice delle rotazioni riordinate:

Indice	$\mathbf{SA_T}$	$\mathbf{F_T}$	Rotazione	$\mathbf{BWT_T}$
0	11	$	$mississippi	i
1	10	i	i$mississipp	p
2	7	i	ippi$mississ	s
3	4	i	issippi$miss	s
4	1	i	ississippi$m	m
5	0	m	mississippi$	$
6	9	p	pi$mississip	p
7	8	p	ppi$mississi	i
8	6	s	sippi$missis	s
9	3	s	sissippi$mis	s
10	5	s	ssippi$missi	i
11	2	s	ssissippi$mi	i

Avendo quindi:

$$F_T = \$iiiimppssss$$

$$BWT_T = ipssm\$pissii$$

2.5.1 Trasformata di Burrows-Wheeler run-length

2.5.2 Matching Statistics

2.5.3 R-index

2.5.4 PHONI

2.6 Trasformata di Burrows-Wheeler posizionale

2.6.1 Implementazione originale

Gli algoritmi di Durbin

Limiti spaziali

2.6.2 Varianti della PBWT

PBWT multi-allelica

PBWT con struttura LEAP

PBWT dinamica

PBWT bidirezionale

Recenti sviluppi

Capitolo 3

Metodo

3.1 Introduzione agli strumenti usati

3.1.1 SDSL

3.1.2 BigRepair

3.1.3 ShapedSlp

Ricostruzione del panel

3.2 Introduzione alle varianti della RLPBWT

3.2.1 Perché un'implementazione run-length

3.3 Mapping nella RLPBWT

3.4 RLPBWT naive

3.4.1 Algoritmo per match massimali

3.5 RLPBWT con bitvectors

3.6 Algoritmo per match massimali

3.7 RLPBWT con pannello denso

3.7.1 Algoritmo con matching statistics

3.8 RLPBWT con SLP

3.8.1 Algoritmo con matching statistics

3.9 Funzione Phi

3.9.1 Costruzione della struttura di supporto

3.9.2 Estensione dei match

Capitolo 4

Risultati

4.1 Ambiente di benchmark

4.1.1 Descrizione input

4.2 Analisi della complessità

4.2.1 Analisi temporale

4.2.2 Analisi spaziale

Capitolo 5

Conclusioni

5.1 Sviluppi futuri

5.1.1 K-mems

5.1.2 RLPBWT multi-allelica

5.1.3 RLPBWT con dati mancanti

Bibliografia e sitografia

[1] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337, 2014.

[2] Markus Lohrey. Algorithmics on slp-compressed strings: A survey. *Groups-Complexity-Cryptology*, 4(2):241–299, 2012.

[3] Travis Gagie, Giovanni Manzini, Gonzalo Navarro, Hiroshi Sakamoto, Louisa Seelbach Benkner, Yoshimasa Takabatake, et al. Practical random access to slp-compressed texts. In *International Symposium on String Processing and Information Retrieval*, pages 221–231. Springer, 2020.

[4] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948, 1993.

[5] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.

[6] Michael Burrows and David Wheeler. A block-sorting lossless data compression algorithm. 1994.

Appendice A

Pseudocodici

Appendice B

Tabelle

Tabella B.1: Stime dello spazio occupato per la memorizzazione di alcune varianti di *bit vector*. Si assume un bit vector di lunghezza n con un numero di bit posti pari a 1 (o $\top$) pari a m. K indica un valore costante.

Variante	**Spazio occupato**
Plain bitvector	$64\lceil \frac{n}{64} + 1 \rceil$
Interleaved bitvector	$\approx n \left(1 + \frac{64}{K}\right)$
H_0*-compressed bitvector*	$\approx \lceil \log \binom{n}{m} \rceil$
Sparse bitvector	$\approx m \left(2 + \log \frac{n}{m}\right)$

Tabella B.2: Complessità temporali stimate della *funzione rank* per alcune varianti di *bit vector*, con la quantità di bit aggiuntivi richiesta. Si assume un bit vector di lunghezza n, con un numero di bit posti pari a 1 (o $\top$) pari a m, e un numero k di valori prima della posizione richiesta.

Variante	**Bit aggiuntivi**	**Complessità temporale**
Plain bitvector	$0.0625 \cdot n$	$\mathcal{O}(1)$
Interleaved bitvector	128	$\mathcal{O}(1)$
H_0*-compressed bitvector*	80	$\mathcal{O}(k)$
Sparse bitvector	64	$\mathcal{O}\left(\log \frac{n}{m}\right)$

Tabella B.3: Complessità temporali stimate della *funzione select* per alcune varianti di *bit vector*, con la quantità di bit aggiuntivi richiesta. Si assume un bit vector di lunghezza n, con un numero di bit posti pari a 1 (o $\top$) pari a m.

Variante	**Bit aggiuntivi**	**Complessità temporale**
Plain bitvector	$\leq 0.2 \cdot n$	$\mathcal{O}(1)$
Interleaved bitvector	64	$\mathcal{O}(\log n)$
H_0*-compressed bitvector*	64	$\mathcal{O}(\log n)$
Sparse bitvector	64	$\mathcal{O}(1)$