

img/logo\_unimib.pdf

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

**Relatore:** *Prof. Raffaella Rizzi*

**Correlatore:** *Dr. Yuri Pirola*

**Tesi di Laurea Magistrale di:**

*Davide Cozzi*

*Matricola 829827*

**Anno Accademico 2021-2022**

*E pensare che  
mi iscrissi ad informatica  
per fare il sistemista!*

# Indice

<b>1</b>	<b>Introduzione</b>	<b>3</b>
<b>2</b>	<b>Preliminari</b>	<b>5</b>
2.1	Bitvector . . . . .	5
2.1.1	Funzione rank . . . . .	7
2.1.2	Funzione select . . . . .	7
2.2	Straight-Line Program . . . . .	8
2.2.1	Longest Common Extension . . . . .	10
2.2.2	Librerie . . . . .	11
2.3	Suffix Array . . . . .	11
2.3.1	Longest common prefix . . . . .	12
2.3.2	SA inverso . . . . .	13
2.3.3	LCP permutato . . . . .	14
2.3.4	Funzione phi . . . . .	15
2.4	Trasformata di Burrows-Wheeler . . . . .	16
2.4.1	Trasformata di Burrows-Wheeler run-length encoded . . . . .	22
2.4.2	RLBWT e r-index . . . . .	22
2.4.3	Match massimali con RLBWT . . . . .	24
2.4.4	Uso delle LCE query . . . . .	28
2.5	Trasformata di Burrows-Wheeler posizionale . . . . .	32
2.5.1	Match massimali con aplotipo esterno . . . . .	39
2.5.2	Varianti della PBWT . . . . .	47
2.5.3	Una prima proposta run-length encoded . . . . .	49
<b>3</b>	<b>Metodo</b>	<b>53</b>
3.1	Perché la compressione run-length . . . . .	53
3.2	Matching Statistics per la RLPBWT . . . . .	54
3.3	Componenti per la RLPBWT . . . . .	57
3.3.1	Componente per il mapping . . . . .	58
3.3.2	Componente per le threshold . . . . .	70
3.3.3	Componente per i prefix array samples . . . . .	73

3.3.4	Componenti per il random access e le LCE query . . . . .	74
3.3.5	Componente per la struttura Phi . . . . .	76
3.4	Strutture dati per la RLPBWT . . . . .	82
3.4.1	Calcolo degli SMEM con LCP . . . . .	83
3.4.2	Calcolo degli SMEM con matching statistics . . . . .	87
<b>4</b>	<b>Risultati sperimentali</b>	<b>98</b>
4.1	Pannelli del 1000 Genome Project . . . . .	100
4.1.1	Riproducibilità degli esperimenti . . . . .	101
4.2	Risultati della sperimentazione . . . . .	102
4.2.1	Costruzione delle strutture e calcolo degli SMEM . . . . .	103
4.2.2	Tempo di una singola query . . . . .	113
<b>5</b>	<b>Conclusioni</b>	<b>119</b>
5.1	Sviluppi futuri . . . . .	120
	<b>Riferimenti</b>	<b>122</b>

# Capitolo 1

## Introduzione

Negli ultimi anni si è assistito ad un cambio di paradigma nel campo della bioinformatica, ovvero il passaggio dallo studio della sequenza lineare di un singolo genoma a quello di un insieme di genomi, provenienti da un gran numero di individui, al fine di poter considerare anche le varianti geniche. Questo nuovo concetto è stato nominato per la prima volta, nel 2005, da Tettelin [1] con il termine di *pangenoma*. Grazie ai risultati ottenuti in pangenomica, ci sono stati miglioramenti sia nel campo della biologia che in quello della medicina personalizzata, grazie al fatto che, con il pangenoma, si migliora la precisione della rappresentazione di multipli genomi e delle loro differenze.

Il genoma umano di riferimento (GRCh38.p14), è composto da circa 3.1 miliardi di basi, con più di 88 milioni varianti tra i genomi sequenziati, secondo i risultati ottenuti nel 1000 Genome Project [2]. Considerando come la quantità dei dati di sequenziamento sia destinata ad aumentare esponenzialmente nei prossimi anni, grazie al miglioramento delle tecnologie di sequenziamento (Next Generation Sequencing e Third-Generation Sequencing), risulta necessaria la costruzione di algoritmi e strutture dati efficienti per gestire una tale informazione. In merito, uno degli approcci più usati per rappresentare il pangenoma è un pannello di aplotipi [3], ovvero, computazionalmente, una matrice di  $M$  righe, corrispondenti agli individui, e  $N$  colonne, corrispondenti ai siti con le varianti. Si specifica che, con il termine aplotipo, si intende l'insieme di alleli, ovvero di varianti, che un organismo eredita da ogni genitore.

In questo contesto trova spazio uno dei problemi fondamentali della bioinformatica, ovvero quello del pattern matching. Inizialmente tale concetto era relativo allo studio di un piccolo pattern all'interno di un testo di grandi dimensioni, ovvero il genoma di riferimento. Ora, con l'introduzione del pangenoma, tale problema si è adattato alle nuove strutture dati.

Lo scopo di questa tesi è ottimizzare il problema del pattern matching, inteso come ricerca dei *set-maximal exact match* (SMEM) tra un aplotipo esterno e un

pannello di aplotipi, in una delle strutture dati più utilizzata: la *trasformata di Burrows-Wheeler Posizionale (PBWT)* [4]. Il progetto di tesi, svolto in collaborazione con il prof. Gage (Dalhousie University) e la prof.ssa Boucher (University of Florida), tra gli autori dei principali risultati ottenuti per la **trasformata di Burrows-Wheeler run-length encoded** [5] [6] [7] [8], ha quindi permesso lo sviluppo di diverse strutture dati composte per la variante **run-length encoded** della **PBWT**, detta **RLPBWT**.

## Struttura della tesi

Nel Capitolo 2 si introdurranno i concetti di base, di ambito computazionale e bio-informatico, necessari a comprendere questa tesi. Nel Capitolo 3 verranno discussi i contributi di questa tesi, descrivendo le soluzioni algoritmiche e le metodologie utilizzate per raggiungere gli obiettivi prefissati. Nel dettaglio verranno presentate varie strutture dati che saranno le componenti necessarie alla produzione delle strutture dati per la *RLPBWT*. Nel Capitolo 4 si discuteranno i risultati ottenuti durante la sperimentazione sui dati reali della *phase 3* del **1000 Genome Project**, progetto, che ha avuto inizio nel 2008, il quale ha visto lo sforzo della comunità scientifica internazionale per catalogazione delle variazioni geniche umane. Infine, nel Capitolo 5, si trarranno le conclusioni di questo progetto di tesi discutendone anche i prospetti futuri.

**DC** L'intera introduzione va estesa

# Capitolo 2

## Preliminari

In questo capitolo verranno specificati tutti i concetti fondamentali, allo stato dell'arte, atti a comprendere i metodi usati in questa tesi. Si introdurranno i concetti di:

- bitvector
- straight-line program e longest common extension query
- suffix array e longest common prefix
- trasformata di Burrows-Wheeler e la sua variante run-length encoded
- trasformata di Burrows-Wheeler posizionale

L'unione di tutte queste strutture e di queste tecniche ha permesso la creazione della *RLPBWT*. *A livello di notazione, si specifica inoltre che, con la notazione  $T[i, j]$  si intende la sottostringa del testo/sequenza/riga/colonna  $T$ , iniziante all'indice  $i$  e terminante all'indice  $j$  incluso. Qualora si avesse  $j > i$  allora si parla di sottostringa nulla  $\varepsilon$ .*

### 2.1 Bitvector

Nonostante qualche primo risultato isolato, si identifica, con la tesi di dottorato di Jacobson del 1988 [9], l'inizio dello studio delle **strutture dati succinte**. Jacobson, con questo termine, denota quelle strutture dati che usano, con  $N$  numero dei differenti oggetti da memorizzare,  $\log N + o(\log N)$  bit. Ad esempio, assumendo un array di  $n$  bit, una struttura dati succinta utilizza  $n + o(n)$  bit, avendo infatti  $N = 2^n$  [10].

Un anno dopo, Jacobson [11] notò come una delle *strutture dati succinte* fondamentali allo sviluppo di altre strutture efficienti in memoria fossero i cosiddetti **bitvector**.

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

$$B[i] \in \{0, 1\}, \forall i \text{ t.c. } 0 \leq i < n \quad (2.1)$$

In alternativa si potrebbe avere, come formalismo:

$$B[i] \in \{\perp, \top\}, \forall i \text{ t.c. } 0 \leq i < n \quad (2.2)$$

Nel corso degli ultimi anni si sono sviluppate diverse varianti dei *bitvector*, finalizzate ad offrire diversi costi di complessità spaziale e diversi tempi computazionali per le principali funzioni offerte.

Il primo vantaggio di questa struttura dati è quella di garantire *random access* in tempo costante, pur sfruttando varie tecniche per la memorizzazione efficiente della stessa in memoria. A tal proposito, lo spazio necessario per l'implementazione, presente nella **Succinct Data Structure Library (SDSL)** [12] (una delle principali librerie, scritta in C++11, per strutture dati succinte) delle principali varianti è visualizzabile in tabella 2.1. Il secondo vantaggio consiste nel fatto che i *bitvector* permettono l'implementazione efficiente di due funzioni:

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

Un'implementazione naive delle stesse richiederebbe tempo  $\mathcal{O}(n)$ , dovendo scansionare l'intero bitvector. In realtà, tali funzioni, al costo teorico di  $o(n)$  bit aggiuntivi, possono essere supportate in tempo costante. Si noti però che, nelle implementazioni di *SDSL*, le complessità temporali delle due funzioni possono variare, sia in termini di bit aggiuntivi che di complessità temporale, a seconda della tipologia di *bitvector*.

Tabella 2.1: Stime dello spazio occupato per la memorizzazione di alcune varianti di *bitvector*. Si assume un bitvector di lunghezza  $n$  con un numero di bit posti pari a 1 (o  $\top$ ) pari a  $m$ .  $K$  indica la *block size* per l'*interleave*.

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

DC Mi è sembrato utile riportare tali tabelle



### 2.1.1 Funzione rank

La prima funzione che si approfondisce è la **funzione rank**. Tale funzione permette di calcolare il *rank* di un dato elemento del bitvector.

**Definizione 2.** Dato un bitvector  $B$ , lungo  $n$ , e data una certa posizione  $i$  del bitvector, la **funzione rank** restituisce il numero di 1 presenti fino a quella data posizione, esclusa. Più formalmente si ha che:

$$\text{rank}_B(i) = \sum_{k=0}^{k < i} B[k], \quad \forall i \text{ t.c. } 0 \leq i < n \quad (2.3)$$

Come detto, da un punto di vista teorico, al costo di  $o(n)$  bit aggiuntivi in memoria tale funzione sarebbe supportata in tempo  $\mathcal{O}(1)$ . La complessità temporale varia però 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 *bitvector* implementate in *SDSL*, è visualizzabile in tabella 2.2.

Tabella 2.2: Complessità temporali stimate della *funzione rank* per alcune varianti di *bitvector*, con la quantità di bit aggiuntivi richiesta. Si assume un bitvector di lunghezza  $n$ , con un numero di bit posti pari a 1 (o  $\top$ ) pari a  $m$ , e un numero  $k$  di *rank sample*.

Variante	Bit aggiuntivi	Complessità temporale
<i>Plain bitvector</i>	$0.0625 \cdot n$	$\mathcal{O}(1)$
<i>Interleaved bitvector</i>	128	$\mathcal{O}(1)$
<i>H<sub>0</sub>-compressed bitvector</i>	80	$\mathcal{O}(k)$
<i>Sparse bitvector</i>	64	$\mathcal{O}(\log \frac{n}{m})$

### 2.1.2 Funzione select

La seconda funzione fondamentale è la **funzione select**, la quale permettere di ottenere l'indice di un simbolo  $\sigma = 1$  nel bitvector.

**Definizione 3.** Dato un bitvector  $B$ , lungo  $n$ , e dato un valore intero  $i$ , la **funzione select** calcola l'indice dell' $i$ -esimo valore pari a 1 nel bitvector  $B$ , lungo  $n$ . Più formalmente si ha, quindi, che:

$$\text{select}_B(i) = \min\{j < n \mid \text{rank}_B(j+1) = i\}, \quad \forall i \text{ t.c. } 0 < i \leq \text{rank}_B(n) \quad (2.4)$$

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 *bitvector* implementate in *SDSL*, è visualizzabile in tabella 2.3.

Tabella 2.3: Complessità temporali stimate della *funzione select* per alcune varianti di *bitvector*, con la quantità di bit aggiuntivi richiesta. Si assume un bitvector di lunghezza  $n$ .

Variante	Bit aggiuntivi	Complessità temporale
<i>Plain bitvector</i>	$\leq 0.2 \cdot n$	$\mathcal{O}(1)$
<i>Interleaved bitvector</i>	64	$\mathcal{O}(\log n)$
<i>H<sub>0</sub>-compressed bitvector</i>	64	$\mathcal{O}(\log n)$
<i>Sparse bitvector</i>	64	$\mathcal{O}(1)$

**Esempio 1.** *Ipotizziamo di avere il seguente bitvector  $B$ , 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:*

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

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

Si vedrà, nel corso di questa tesi, come l'uso di tali strutture, nel dettaglio l'uso dei *plain bitvector* e dei *bitvector sparsi*, sia fondamentale per lo studio delle due strutture *run-length encoded*.

## 2.2 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 per la compressione di testi.

Per questa tesi si è quindi pensato all'uso dei cosiddetti **Straight-Line Programs**

(**SLP**). In termini generici, un *SLP* è una **grammatica context-free** che genera una e una sola parola [13], parlando, quindi, di **grammar-based compression**.

**Definizione 4.** Sia dato un alfabeto finito  $\Sigma$  di simboli terminali. Sia data una stringa  $s = a_1, a_2, \dots, a_n \in \Sigma^*$ , lunga  $n$  e costruita sull'alfabeto  $\Sigma$ , avendo  $a_i \in \Sigma$ ,  $\forall i$  t.c.  $1 \leq i \leq n$ . Si denota, con  $\text{alph}(s) = \{a_1, a_2, \dots, a_n\}$ , l'insieme dei simboli della stringa  $s$ .

Si definisce **SLP**, costruito sull'alfabeto  $\Sigma$ , una grammatica context-free  $\mathcal{A}$  tale che:

$$\mathcal{A} = (\mathcal{V}, \Sigma, \mathcal{S}, \mathcal{P}) \quad (2.5)$$

Dove:

- $\mathcal{V}$  è l'insieme dei simboli non terminali
- $\Sigma$  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:

$$\mathcal{P} \subseteq \mathcal{V} \times (\mathcal{V} \cup \Sigma)^* \quad (2.6)$$

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

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

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

$$|\mathcal{A}| = \sum_{(A, \alpha) \in \mathcal{P}} |\alpha| \quad (2.7)$$

Il linguaggio  $\mathcal{A}$  generato da un *SLP* consiste in una singola parola, denotata da  $\text{eval}(\mathcal{A})$ .

A partire dall'*SLP*  $\mathcal{A}$  si genera 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  $\Sigma$ .

**Esempio 2.** Si prenda, ad esempio [14], la seguente stringa:

$$s = \text{GATTAGATACAT\$GATTACATAGAT}$$

Si potrebbe produrre il seguente *SLP*:

DC verificare questo secondo punto

- $S \rightarrow ZWAY\$ZYAW$
- $X \rightarrow TA$
- $Z \rightarrow WX$
- $W \rightarrow GV$
- $Y \rightarrow CV$
- $V \rightarrow AT$

Al quale corrisponde il seguente albero di derivazione:



Si noti che 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, Gagic et al. [14] proposero un articolo, a cui si rimanda per approfondimenti, in merito a miglioramenti prestazionali per il *random access* all'*SLP*, anche tramite l'uso dei *bitvector sparsi*.

Si stima che il tempo necessario al *random access* su un testo  $T$ , compresso tramite *SLP* e lungo  $n$ , sia in tempo:

$$\mathcal{O}(\log n) \quad (2.8)$$

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)**.

### 2.2.1 Longest Common Extension

Oltre a permettere *random access* alla testo compresso, l'uso degli *SLP* permette di effettuare un'altra operazione in modo efficiente: le **Longest Common Extension (LCE) queries**.

**Definizione 5.** 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 le due sottostringhe.

Si stima che il calcolo di una *LCE query* di lunghezza  $l$  sia effettuabile, per l'*SLP* di un testo  $T$  lungo  $n$ , in tempo:

$$\mathcal{O}(\log n + l) \quad (2.9)$$

Che di fatto è proporzionale a:

$$\mathcal{O}(\log n) \quad (2.10)$$

I due concetti di *SLP* e *LCE query* verranno generalizzati, in questa tesi, all'uso su matrici permettendo una rappresentazione compatta in memoria, con *random access*, per un pannello di aplotipi.

DC capire se sono necessari altri approfondimenti

## 2.2.2 Librerie

Da un punto di vista implementativo, l'oggetto contenente l'*SLP* del pannello viene costruito ed interrogato mediante l'uso della libreria **ShapedSlp**<sup>1</sup>, implementazione dei risultati ottenuti da Gagie et al. [14]. Inoltre, tale libreria basa il suo funzionamento sull'uso di un'altra libreria, detta **BigRePair**<sup>2</sup>, che implementa i quanto studiato da Gagie et al. [15] in merito alla compressione, via uso di grammatiche, di file con frequenti ripetizioni (come possono essere, nel nostro caso, pannelli binari di aplotipi).

In termini di pipeline si procede quindi:

1. generando la *grammatica* tramite *BigRePair*, che accetta come file di input un file `txt` “raw” ma anche un file in formati più standard come i *FASTA*
2. generando l'*SLP* tramite *ShapedSlp* specificatamente a partire dai risultati di *BigRePair* (si segnala che la libreria accetta anche input prodotti tramite altri tool che non verranno qui approfonditi)

## 2.3 Suffix Array

Nel 1976, Manber e Myers [16] proposero una struttura dati per la memorizzazione di stringhe e la loro interrogazione, efficiente sia in termini di uso della memoria che di complessità temporale. Tale struttura venne denotata **Suffix Array (SA)**.

<sup>1</sup><https://github.com/itomomoti/ShapedSlp>

<sup>2</sup><https://gitlab.com/manzai/bigrepair>

**Definizione 6.** Dato un testo  $T$ ,  $\$$ -terminato (assumendo che il simbolo  $\$$  sia sempre il simbolo lessicograficamente minore nell'alfabeto di studio), tale che  $|T| = n$ , si definisce **suffix array** di  $T$ , denotato con  $SA_T$ , un di interi array lungo  $n$ , tale che  $SA_T[i] = j$  sse il suffisso di ordine  $j$ , ovvero  $T[j, n-1]$ , è l' $i$ -esimo suffisso nell'ordinamento lessicografico dei suffissi di  $T$ . Ne segue che, presi  $i, i' \in \mathbb{N}$  tali che  $0 \leq i < i' < n$  allora vale che, indicando con  $\prec$  l'ordinamento lessicografico:

$$T[SA_T[i], n-1] \prec T[SA_T[i'], n-1] \quad (2.11)$$

Il **suffix array** è quindi una permutazione dei numeri interi in  $\{0, n-1\}$ .

**Esempio 3.** Si prenda la stringa:

$$s = \text{mississippi}\$, \quad |s| = 12$$

Si producono quindi i seguenti suffissi e il loro riordinamento:

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

Ottenendo quindi che:

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

### 2.3.1 Longest common prefix

L'uso del *suffix array* è spesso accompagnato da un'altra struttura dati, detta **Longest Common Prefix**.

**Definizione 7.** Si definisce il **Longest Common Prefix (LCP)** di un testo  $T$  lungo  $n$ , 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, ovvero l'ordinamento specificato da  $SA_T$ . 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  $y$ :

$$LCP_T[i] = \begin{cases} -1 & \text{se } i = 0 \vee i = n \\ |lcp(T[SA_T[i-1], n], T[SA_T[i], n])| & \text{altrimenti} \end{cases} \quad (2.12)$$

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

Indice	SA <sub>T</sub>	LCP <sub>T</sub>	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	ssissippi\$
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  $m$ :

$$\mathcal{O}(m \log n) \quad (2.13)$$

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

$$\mathcal{O}(m + \log n) \quad (2.14)$$

Per ulteriori approfondimenti in merito agli algoritmi di pattern matching basati su *suffix array* e ai relativi *acceleratori*, si rimanda al testo di Gusfield [17].

### 2.3.2 SA inverso

Ai fini di poter comprendere future definizioni si presenta anche la permutazione inversa dei valori del *suffix array*, detta **Inverse Suffix Array (ISA)**. Grazie a tale permutazione inversa, dato un indice di suffisso, è possibile sapere in quale posizione si trovi tale suffisso nel *suffix array*.

**Definizione 8.** Dato il **suffix array**  $SA_T$ , costruito su un testo  $T$  di lunghezza  $n$ , si definisce l'**inverse suffix array**, denotato con  $ISA_T$ , come:

$$ISA_T[i] = j \iff SA_T[j] = i, \forall i \in \{0, n-1\} \quad (2.15)$$

**Esempio 5.** Riprendendo l'esempio precedente si avrebbe quindi:

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

### 2.3.3 LCP permutato

Un'altra permutazione che bisogna introdurre è il **permuted longest-common-prefix array (PLCP)** [18]. Tale permutazione permette una rappresentazione succinta in memoria dell' $LCP$  [19], permettendo di ottenere gli stessi risultati di quest'ultimo. Un'altro vantaggio è che la sua ricostruzione richiede un minor costo computazionale.

**Definizione 9.** Si definisce il **permuted longest-common-prefix array**, denotato con  $PLCP_T$ , costruito a partire da un testo  $T$  di lunghezza  $n$ , come un array tale per cui [8]:

$$PLCP_T[p] = \begin{cases} -1 & \text{se } ISA_T[p] = 0 \\ LCP_T[ISA_T[p]] & \text{altrimenti} \end{cases}, \forall p \in \{0, n-1\} \quad (2.16)$$

**DC** L'intera sottosezione potrebbe essere quasi totalmente rimossa ma almeno al definizione serve per il calcolo di tutte le occorrenze di un MEM, come in PHONI

Quindi, i valori sono in ordine di posizione, ovvero l'ordine originale dato dagli indici dei suffissi, e non lessicografico. In altri termini, si ha una permutazione dei valori di  $LCP_T$  tale per cui [18]:

$$PLCP_T[SA_T[p]] = LCP_T[p], \forall p \in \{1, n-1\} \quad (2.17)$$

**Esempio 6.** Riprendendo l'esempio precedente si avrebbe quindi:



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

Ciò che permette una rappresentazione compatta del  $PLCP$  è descritto nel seguente lemma [20].

**Lemma 1.** *Dato un testo  $T$ , tale che  $|T| = n$ , si ha che:*

$$PCLP_T[i] \geq PLCP_T[i - 1] - 1, \forall i \in \{1, n - 1\} \quad (2.18)$$

Grazie a tale lemma si può memorizzare l' $PLCP$  sparso.

**Definizione 10.** *Dato un intero  $q$ , per il quale calcolo (basato sul lemma precedente) si rimanda al paper di Kasai [20], si definisce **array  $PCLP$  sparso**, lungo  $\left\lfloor \frac{n}{q} \right\rfloor$  e denotato  $PLCP_q$ , l'array che memorizza ogni  $q$ -esimo valore del  $PLCP$ , avendo che:*

$$PLCP_q[i] = PLCP_T[iq] \quad (2.19)$$

**DC** Capire se serve variante sparsa e se serve esempio

### 2.3.4 Funzione phi

L'ultimo concetto che si introduce sono le **funzioni**  $\varphi$  e  $\varphi^{-1}$ , usate per poter identificare i valori precedenti e successivi di un dato valore in  $SA_T$ . Essere sono utili al fine di poter sia ricostruire efficientemente il  $PLCP$  di un testo  $T$  (per i dettagli si rimanda all'articolo di Kärkkäinen [18]) che di permettere, come si vedrà più avanti nella sezione 2.4, il riconoscimento di tutte le occorrenze di un **match massimale esatto** (**MEM**) in  $T$  [8].

**Definizione 11.** *Dato un testo  $T$  di lunghezza  $n$  si definiscono le funzioni, che di fatto sono permutazioni dei valori di  $SA_T$ ,  $\varphi$  e  $\varphi^{-1}$  come [8]:*

$$\varphi(p) = \begin{cases} null & \text{se } ISA_T[p] = 0 \\ SA_T[ISA_T[p] - 1] & \text{altrimenti} \end{cases}, \forall p \in \{0, n - 1\} \quad (2.20)$$

$$\varphi^{-1}(p) = \begin{cases} \text{null} & \text{se } ISA_T[p] = n - 1 \\ SA_T[ISA_T[p] + 1] & \text{altrimenti} \end{cases}, \quad \forall p \in \{0, n - 1\} \quad (2.21)$$

Si noti che si ha il valore *null* quando, rispettivamente, si studia il primo e l'ultimo valore del suffix array in quanto non hanno, sempre rispettivamente, l'antecedente e il successore. Infatti, semplicemente, tali funzioni restituiscono i due valori, se esistenti, di  $SA_T$  adiacenti ad un valore del suffix array dato.

Analogamente, sempre coi medesimi vincoli, possono essere definite come [18]:

$$\varphi[SA[p]] = SA[p - 1] \quad (2.22)$$

$$\varphi^{-1}[SA[p]] = SA[p + 1] \quad (2.23)$$

**Esempio 7.** Riprendendo l'esempio precedente si avrebbe quindi:

Indice	$SA_T$	$ISA_T$	$\varphi$	$\varphi^{-1}$	Suffisso
0	11	5	1	9	\$
1	10	4	4	0	i\$
2	7	11	5	<i>null</i>	ippi\$
3	4	9	6	5	issippi\$
4	1	3	7	1	ississippi\$
5	0	10	3	2	mississippi\$
6	9	8	8	3	pi\$
7	8	2	10	4	ppi\$
8	6	7	9	6	sippi\$
9	3	6	0	8	sissippi\$
10	5	1	11	7	ssippi\$
11	2	0	<i>null</i>	10	ssissippi\$

Infatti, ad esempio, il valore 9 in  $SA_T$  è preceduto dal valore  $\varphi(9) = 0$  ed è seguito dal valore  $\varphi^{-1}(9) = 8$ .

## 2.4 Trasformata di Burrows-Wheeler

Introdotta nel 1994 da Burrows e Wheeler con lo scopo di comprimere testi, la **Burrows-Wheeler Transform** [21] 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 dell'efficienza in memoria.

Nel dettaglio la *BWT* è una *trasformata reversibile* che permette una *compressione lossless*, quindi senza perdita d'informazione. Tale trasformazione viene costruita a partire dal riordinamento dei caratteri del testo in input, riordinando lessicograficamente le cosiddette **rotazioni** del testo. Interessante è la proprietà per cui

caratteri uguali tendono ad essere posti consecutivamente all'interno della stringa prodotta dalla trasformata. Questa proprietà è causata dalle ripetizioni di sottostringhe all'interno del testo stesso.

**Definizione 12.** 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:

$$BWT_T[i] = \begin{cases} T[SA_T[i] - 1] & \text{se } SA_T[i] \neq 1 \\ \$ & \text{altrimenti} \end{cases}, \forall i \in \{0, n-1\} \quad (2.24)$$

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

**Definizione 13.** Si definisce **rotazione  $i$ -esima** di un testo  $T$  lungo  $n$ , denotata con  $rot_T(i)$ , come la stringa ottenuta dalla concatenazione del suffisso  $i$ -esimo con la restante porzione del testo. Più formalmente si ha che, denotando con  $X \cdot Y$  la concatenazione tra la stringa  $X$  e la stringa  $Y$ :

$$rot_T(i) = T[i, n-1] \cdot T[0, i-1], \forall i \in \{0, n-1\} \quad (2.25)$$

Data questa definizione, quindi, la  $BWT$  del testo  $T$  risulta essere l'ultima colonna della matrice, detta **Burrows-Wheeler Matrix (BWM)**, 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 è l'array formato dalla prima colonna della  $BWM$ . In pratica l'array  $F$  è, banalmente, l'array formato dal riordinamento lessicografico dei caratteri del testo  $T$ .

**Esempio 8.** Si prenda la stringa:

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

Si produce la  $BWM_T$ , da cui si estraggono  $F_T$  e  $BWT_T$ :

Indice	$SA_T$	$F_T$	$BWM_T$	$BWT_T$
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

L'importanza di questa trasformata è dovuta soprattutto al fatto che sia *reversibile*, implicando quindi che a partire da  $BWT_T$  è possibile ricostruire  $T$ . Questo è possibile grazie ad una proprietà intrinseca della trasformata che viene riassunta nel concetto di **LF-mapping**.

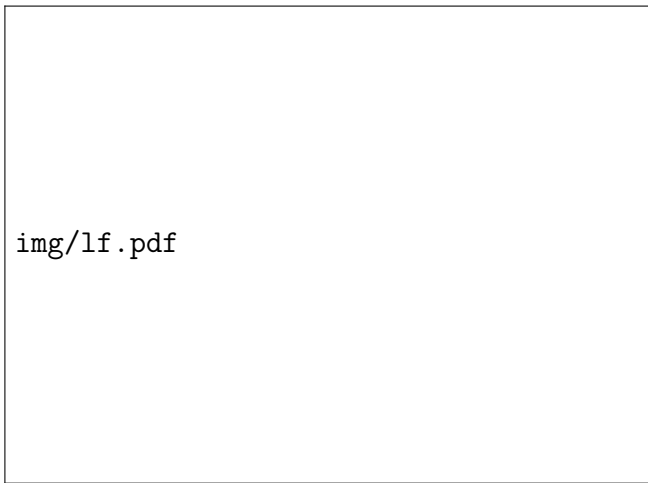
**Definizione 14.** Dato un testo  $T$ , tale che  $|T| = n$ , data la sua  $BWT_T$  e il suo array  $F_T$  si definisce **LF-mapping** come la proprietà per la quale l' $i$ -esima occorrenza di un carattere  $\sigma$  in  $BWT_T$  corrisponde all' $i$ -esima occorrenza dello stesso carattere in  $F_T$ .

Grazie a questa definizione è possibile partire dall'ultimo carattere del testo, \$, e ricostruire l'intero testo a ritroso. Si vede quindi un breve esempio.

**Esempio 9.** Si riprende l'esempio precedente, avendo:

$$BWT_T = ipssm$piissii \text{ e } F_T = $iiiiimppssss$$

E avendo i seguenti caratteri associati dall'LF-mapping:



img/lf.pdf

Si comincia dal simbolo '\$' in  $BWT_T$ , che è l'ultimo carattere di  $T$ . Si ha che esso corrisponde al primo e unico simbolo '\$' in  $F_T$ , all'indice 0. Tale simbolo, per l'ovvia proprietà delle rotazioni è preceduto dal simbolo  $BWT_T[0] = 'i'$  in  $T$ . Quindi 'i' precederà '\$' in  $T$ :

$$T = \dots i\$$$

Si sa inoltre che tale 'i' è il primo 'i' in  $BWT_T$ . Si cerca quindi il primo simbolo 'i' anche in  $F_T$ , sapendo che sono lo stesso simbolo nel testo. A questo punto il simbolo allo stesso indice di tale 'i' nella  $BWT_T$ , ovvero il simbolo 'p', sarà il simbolo che precede 'i' nel testo:

$$T = \dots pi\$$$

Proseguendo a ritroso si ricostruisce l'intero testo:

$$T = mississippi\$$$

DC Questo esempio serve davvero?

## FM-index

Tramite l'uso dell'*LF-mapping* è possibile risolvere il problema di ricerca di un pattern all'interno del testo, tramite l'algoritmo nominato **backward search**. Questa tecnica consiste nell'iterare il pattern da destra a sinistra e salvare, di volta in volta, un intervallo sul *suffix array*. Nel dettaglio, ipotizzando di essere in posizione  $i$  del pattern, tale intervallo è relativo a quei suffissi che hanno come prefisso il suffisso  $i$ -esimo del pattern. Tale intervallo viene quindi esteso usando il carattere  $P[i - 1]$  selezionando il nuovo intervallo sul *suffix array*. Tale aggiornamento è detto **backward step** e consiste nell'aggiornare l'intervallo sul suffix array a quei suffissi del testo che, estesi a sinistra col carattere  $(i - 1)$ -esimo del pattern, presentano un match con  $P[i - 1, |P| - 1]$ .

Usando la *BWT* è possibile usare due funzioni, dette  $C$  e  $Occ$ , per computare la *backward search*.

**Definizione 15.** Dato un testo  $\$$ -terminato  $T$ , lungo  $n$  e costruito su alfabeto  $\Sigma$ , si definisce la funzione  $C$ , tale che:

$$C : \Sigma \cup \$ \rightarrow \mathbb{N} \quad (2.26)$$

avendo che, dato un carattere  $\sigma \in \Sigma$ ,  $C(\sigma)$  restituisce il numero di occorrenze dei caratteri lessicograficamente più piccoli di  $\sigma$  in  $T$ .

**Definizione 16.** Dato un testo  $\$$ -terminato  $T$ , lungo  $n$  e costruito su alfabeto  $\Sigma$ , e la sua  $BWT_T$ , si definisce la funzione  $Occ$ , tale che:

$$Occ : \Sigma \cup \$ \times \{0, n\} \rightarrow \mathbb{N} \quad (2.27)$$

avendo che, dato un carattere  $\sigma \in \Sigma$  e una posizione  $i$  della  $BWT_T$ ,  $Occ(\sigma, i)$  restituisce il numero di occorrenze del carattere  $\sigma$  nei primi  $i$  elementi di  $BWT_T$ .

Questa coppia di funzioni prende il nome di **FM-index** [22], il quale è definito essere un *self index* in quanto è possibile tenere in memoria solo l'indice per ottenere i risultati medesimi della  $BWT_T$ , ricordando anche che da essa si può ricostruire il testo  $T$ .

**Esempio 10.** Si prenda la stringa:

$$T = \text{mississippi}\$, |s| = 12$$

Che produce:

$$BWT_T = \text{ipssm\$pissii}$$

Si ha, per  $C(\sigma)$ :

$\sigma$	$C(\sigma)$
\$	0
i	1
m	5
p	6
s	8

Mentre per  $Occ(\sigma, i)$  si ha:

0	0	0	0	0	0
1	0	1	0	0	0
2	0	1	0	1	0
3	0	1	0	1	1
4	0	1	0	1	2
5	0	1	1	1	2
6	1	1	1	1	2
7	1	1	1	2	2
8	1	2	1	2	2
9	1	2	1	2	3
10	1	2	1	2	4
11	1	3	1	2	4
12	1	4	1	2	4
i/ $\sigma$	\$	i	m	p	s

Dato un simbolo  $\sigma$  del pattern e il precedente intervallo  $[f, g)$  su  $SA_T$ , eseguire il **backward step**, tramite l'*FM-index*, aggiornando  $f$  e  $g$  nel seguente modo:

$$f' = C(\sigma) + Occ(\sigma, f), \quad g' = C(\sigma) + Occ(\sigma, g) \quad (2.28)$$

Ritornando il nuovo intervallo  $[f, g) \leftarrow [f', g')$  sse  $f' < g'$ . Si segnala che tali variabili sono inizializzate con  $f = 0$  e  $g = n$ .

Tale calcolo altro non è che l'*LF-mapping*. Infatti, partendo da un intervallo su  $SA_T$  (che è anche un intervallo su  $BWT_T$ ), si identifica quali suffissi sono preceduti dal simbolo del pattern voluto. Tale simbolo, se il pattern ha un'occorrenza fino al carattere in analisi, sarà presente in sottointervallo di  $[f, g)$  sulla  $BWT_T$ . Una volta identificati tali caratteri su  $BWT_T$  si usano  $C(\sigma)$  e  $Occ(\sigma, i)$ , per trovare tali caratteri su  $F_T$ , calcolando il nuovo intervallo  $[f, g)$ .

DC capire se dire meglio

**Esempio 11.** Si assuma il pattern  $P = iss$  da voler ricercare nel testo  $T = mississippi\$$ . Si ha, in termini di inizializzazione che  $f = 0$ ,  $g = 12$ ,  $\sigma = P[|P| - 1] = P[2] = s$ . Si calcolano i nuovi  $f'$  e  $g'$ :

$$f' = C(s) + Occ(s, 0) = 8 + 0 = 8$$

$$g' = C(s) + Occ(s, 12) = 8 + 4 = 12$$

Ottenendo l'intervallo  $[8, 12)$  sul suffix array.

Si prosegue leggendo il carattere  $\sigma = P[1] = s$ :

$$f' = C(s) + Occ(s, 8) = 8 + 2 = 10$$

$$g' = C(s) + Occ(s, 12) = 8 + 4 = 12$$

Limitando quindi l'intervallo a  $[10, 12)$ . Si noti che tale intervallo corrisponde ai due simboli "s" presenti in  $BWT_T[8, 11]$ , che sono esattamente i simboli in  $F_T[10, 11]$ . Un ulteriore aggiornamento, col carattere  $\sigma = P[0] = i$ , comporta:

$$f' = C(i) + Occ(i, 10) = 1 + 2 = 3$$

$$g' = C(i) + Occ(i, 12) = 1 + 4 = 5$$

Avendo l'intervallo finale su  $SA_T$  del match, ovvero:  $[3, 5)$ . Seguendo l'intero ragionamento sul suffix array si avrebbe:

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

⇓

Indice	SA <sub>T</sub>	F <sub>T</sub>	BWM <sub>T</sub>	BWT <sub>T</sub>		Indice	SA <sub>T</sub>	F <sub>T</sub>	BWM <sub>T</sub>	BWT <sub>T</sub>
0	11	\$	\$mississippi	i		0	11	\$	\$mississippi	i
1	10	i	i\$mississipp	p		1	10	i	i\$mississipp	p
2	7	i	ippi\$mississ	s		2	7	i	ippi\$mississ	s
3	4	i	issippi\$miss	s		3	4	i	issippi\$miss	s
4	1	i	ississippi\$m	m		4	1	i	ississippi\$m	m
5	0	m	mississippi\$	\$	⇒	5	0	m	mississippi\$	\$
6	9	p	pi\$mississip	p		6	9	p	pi\$mississip	p
7	8	p	ppi\$mississi	i		7	8	p	ppi\$mississi	i
8	6	s	sippi\$missis	s		8	6	s	sippi\$missis	s
9	3	s	sissippi\$mis	s		9	3	s	sissippi\$mis	s
10	5	s	ssippi\$missi	i		10	5	s	ssippi\$missi	i
11	2	s	ssissippi\$mi	i		11	2	s	ssissippi\$mi	i

Avendo quindi che le occorrenze del pattern  $P = iss$  iniziano alle posizioni  $SA_T[3] = 4$  e  $SA_T[4] = 1$  del testo.

### 2.4.1 Trasformata di Burrows-Wheeler run-length encoded

Come già introdotto, la *BWT* tende ad avere caratteri uguali in posizioni consecutive all'interno della sua sequenza. Si è quindi pensato, fin da subito, ad un modo efficiente per memorizzare in modo compresso testi mediante l'uso del **run-length encoding**. Tale tecnica consiste nel memorizzare le cosiddette **run**, ovvero sequenze massimali di caratteri uguali, mediante coppie:

(carattere, lunghezza della run)

**Esempio 12.** Vediamo un breve esempio.

Si ipotizzi di avere la seguente stringa:

$s = aaaacctgggggg$

Una sua memorizzazione run-length sarebbe del tipo:

$\{(a, 4), (c, 2), (t, 1), (g, 6)\}$

### 2.4.2 RLBWT e r-index

In questa direzione, nel 2005, Mäniken e Navarro proposero la **Run-Length encoded Burrows-Wheeler Transform (RLBWT)** [5].

**Definizione 17.** Dato un testo  $T$  si definisce la **RLBWT** di  $T$  come la rappresentazione run-length encoded della  $BWT_T$ , denotandola con  $RLBWT_T$ . Si noti che, avendo  $r$  come numero di run nella  $BWT_T$ :

$$|RLBWT_T| = r \quad (2.29)$$



L'uso di tale struttura risulta particolarmente efficiente, ad esempio, volendo creare un'unica  $BWT$  a partire dalla concatenazione di multipli genomi. Infatti, tale concatenazione conterrà, per ovvie ragioni biologiche, diverse regioni genomiche ripetute.

Una strategia per la memorizzazione in modo compatto la  $RLBWT$  è quella di memorizzare:

- una stringa  $c$ , tale che  $|c| = r$ , contenente un solo carattere per ogni run della  $BWT_T$
- un bitvector  $bv$ , lungo quanto  $BWT_T$ , tale che  $bv[i] = 1$  sse  $BWT_T[i]$  è il primo carattere, detto anche *testa*, di una run

**Esempio 13.** Si prenda ad esempio la seguente  $BWT_T$ :

$$BWT_T = acggtcccaa$$

Si hanno:

$$c = acgtca$$

$$bv = 1110110010$$

Mäniken e Navarro hanno proposto anche il seguente teorema.

**Teorema 1.** Dato un testo  $T$ , tale che  $|T| = n$ , se ne può costruire la  $RLBWT$  in uno spazio  $\mathcal{O}(r)$  tale per cui si possono conteggiare tutte le occorrenze di un pattern  $P$ , tale che  $|P| = m$ , in tempo:

$$\mathcal{O}(m \log n) \tag{2.30}$$

La struttura dati dietro questo risultato ha preso il nome di **r-index**. Tale indice consiste in:

- la  $RLBWT$
- dei *suffix array sample*

Nonostante questi ottimi risultati, per poter computare l'*r-index*, si richiedeva anche la costruzione dei *suffix array samples* in spazio  $\mathcal{O}(r)$ .

Grazie a tale indice, dato un testo  $T$ , tale che  $|T| = n$ , e dato un pattern  $P$ , tale che  $|P| = m$ , è stato possibile:

- conteggiare le occorrenze (*count query*) del pattern nel testo, in tempo  $\mathcal{O}(m \log n)$ , con spazio  $\mathcal{O}(r)$

- localizzare tali occorrenze (*locate query*) in tempo  $\mathcal{O}(s)$ , con spazio  $\mathcal{O}\left(\frac{r}{s}\right)$ , avendo  $s$  come distanza tra due *SA samples*

Si ha quindi che i *SA samples* sono di un ordine di grandezza maggiore, in termini di memoria, rispetto alla *RLBWT*.

Nel 2017, Policriti and Prezza [23] proposero un teorema fondamentale in questo ambito.

**Teorema 2** (Toehold lemma). *Dato un testo  $T$ , tale che  $|T| = n$ , e dato un pattern  $P$ , tale che  $|P| = m$ , si può computare l'intervallo sulla  $BWT_T$  contenente i  $k$  caratteri precedenti le occorrenze di  $P$  in  $T$  in spazio  $\mathcal{O}(r)$  e in tempo:*

$$\mathcal{O}(m \log \log n) \quad (2.31)$$

Questo risultato dimostra come identificare **un** *SA sample* nell'intervallo contenente il pattern  $P$ . Il limite è dato dal fatto che non si supporta la localizzazione di tutte le  $k$  occorrenze degli *SA samples* in quell'intervallo.

Nel 2020 Gagie et al [6], combinando la *RLBWT* e il *Toehold lemma*, trovarono una soluzione a questo problema, mediante la definizione della funzione  $\varphi$  (che nel dettaglio si dettaglierà più avanti). Tale funzione ha permesso di avere le *locate query* in spazio  $\mathcal{O}(r)$ . Tale risultato si riassume nel seguente teorema.

**Teorema 3.** *Dato un testo  $T$ , tale che  $|T| = n$ , si può memorizzare  $T$  in spazio  $\mathcal{O}(r)$  tale che si possano trovare tutte le  $k$  occorrenze di un pattern  $P$ , lungo  $m$ , in tempo:*

$$\mathcal{O}((m + k) \log \log n) \quad (2.32)$$

Nel dettaglio, i risultati di Gagie portarono a definire l'**r-index** tramite l'uso dei valori del *SA* all'inizio e alla fine di ogni run come *suffix array sample*. Si è quindi ottenuto che i *suffix array sample* possono essere memorizzati in spazio proporzionale al numero di run, pur permettendo in modo efficiente le *locate query*. Per i dettagli in merito alla costruzione dell'*r-index* si rimanda ai paper di Kuhnle et al. [24], di Mun et al. [25] e di Boucher et al. [26].

### 2.4.3 Match massimali con RLBWT

Dopo aver introdotto l'**r-index** bisogna brevemente come avvenga il calcolo dei cosiddetti **Maximal Exact Match (MEM)**, ovvero match esatti, tra un pattern e un testo, che non possono essere estesi in alcuna direzione.

**Definizione 18.** *Dato un testo  $T$ , con  $|T| = n$ , e un pattern  $P$ , con  $|P| = m$ , si definisce una sottostringa  $P[i, i + l - 1]$ , di lunghezza  $l$ , **MEM** di  $P$  in  $T$  se:*

- $P[i, i + l - 1]$  è una sottostringa di  $T$

- $P[i - 1, i + l - 1]$  non è una sottostringa di  $T$  (non si può estendere a sinistra)
- $P[i, i + l]$  non è una sottostringa di  $T$  (non si può estendere a destra)

L'importanza nel calcolo dei match massimali esatti si ritrova nel loro uso nei metodi di allineamento basati sul **paradigma seed-and-extend**. Tale paradigma, sfruttato in algoritmi di allineamento come **BLAST** [27], uno degli allineatori più usati al mondo, si basa sul trovare *MEM* di piccola lunghezza, i *seed* appunto, per poi continuare l'allineamento tramite algoritmi più sofisticati, spesso basati sulla *programmazione dinamica*.

Nel 2020, Bannai et al. [28] mostrarono come il calcolo dei *MEM* fosse equivalente al calcolo delle **Matching Statistics (MS)**, un concetto teorico molto usato in *bioinformatica*. Informalmente, per ogni posizione  $i$  del pattern, le matching statistics riportano la lunghezza e una posizione di inizio sul testo della più lunga sottostringa comune tra il testo e  $P[i, |P| - 1]$ .

**Definizione 19.** Dato un testo  $T$ , con  $|T| = n$ , e un pattern  $P$ , con  $|P| = m$ , si definisce **matching statistics** di  $P$  su  $T$  un array  $MS$  di coppie  $(pos, len)$ , lungo quanto il pattern, tale che:

- $T[MS[i].pos, MS[i].pos + MS[i].len - 1] = P[i, i + MS[i].len - 1]$ , quindi si ha un match tra  $P$  e  $T$  lungo  $MS[i].len$  a partire da  $MS[i].pos$  in  $T$  e da  $i$  in  $P$
- $P[i, i + MS[i].len]$  non occorre in  $T$ , quindi il match non è ulteriormente estendibile

Una volta calcolato l'array  $MS$  si ha il seguente lemma.

**Lemma 2.** Dato un testo  $T$ , un pattern  $P$  lungo  $m$  e il corrispondente array di matching statistics  $MS$ , si ha che:

$$P[i, i + l - 1], \forall 0 < i \leq m \quad (2.33)$$

è un **MEM** di lunghezza  $l$  in  $T$  sse:

$$MS[i].len = l \wedge MS[i - 1].len \leq MS[i].len \quad (2.34)$$

Inoltre, qualora si avesse  $i = 0$ , si ha che  $P[0, l - 1]$  è un **MEM** di lunghezza  $l$  in  $T$  sse:

$$MS[0].len = l \wedge MS[0].len \geq MS[1].len \quad (2.35)$$

Per costruire l'array delle matching statistics l'approccio naive è quello di sfruttare interamente l'*LCP array* ma, sempre nell'articolo di Bannai et al.[28], si è presentato una semplice concetto in grado di ottimizzare il processo, quello delle **threshold**. Questa piccola struttura dati memorizza il minimo valore dell'*LCP array* tra due run consecutive del medesimo simbolo nella *BWT*.

**Definizione 20.** Dato un testo  $T$  e date  $BWT_T[j', j]$  e  $BWT_T[k, k']$  due run consecutive dello stesso carattere in  $BWT_T$ , si definisce **threshold** la posizione:

$$j < i \leq k \text{ tale che } i \text{ è l'indice del minimo valore in } LCP[j + 1, k] \quad (2.36)$$

Rossi et al., nel 2021, sfruttarono tutte le conoscenze relative alla **RLBWT**, all'**r-index** e alle **matching statistics** per ideare **MONI: A Pangenomics Index for Finding MEMs** [7]. In questa soluzione si ha quindi la costruzione, in due *sweep*, tramite l'**algoritmo di Bannai**, dell'array delle *matching statistics*. Infatti si ha:

- un primo sweep che computa i vari  $MS[i].pos$
- un secondo sweep che, tramite random access sul testo  $T$  computa i vari  $MS[i].len$ , confrontando direttamente le due sottostringhe del testo e del pattern. Contemporaneamente a tale calcolo, l'algoritmo annota gli eventuali *MEM*

Nel dettaglio, per computare i valori  $MS[i].pos$ , si procede scorrendo il pattern  $P$ , lungo  $m$ , da destra a sinistra. Brevemente i passi dell'algoritmo sono i seguenti:

1. si inizia cercando l'ultima occorrenza, di indice  $q$ , di  $P[i]$ ,  $i = m - 1$ , in  $BWT_T$ , "virtualmente" essendo essa in memorizzata in modo compatto tramite compressione *run-length*
2. si procede tramite *LF-mapping* a partire da  $q$ , arrivando in una nuova posizione  $q$  per le medesime motivazione descritte precedentemente nel caso della *BWT*
3. si memorizza  $MS[i].pos = SA_T[q]$
4. a questo punto si hanno due alternative:
  - se  $BWT_T[q] = P[i - 1]$  si procede con il mapping come in 2.
  - se  $BWT_T[q] \neq P[i - 1]$  si deve selezionare un nuovo  $q$  tale per cui  $BWT_T[q] = P[i - 1]$ . Questo può essere o l'indice della coda della run precedente di simboli  $P[i - 1]$  o la testa della run successiva di simboli  $P[i - 1]$ . Qualora non si debba scegliere, ovvero la run attuale

non è preceduta/succeduta da una run di simboli  $P[i-1]$ , si sceglie, rispettivamente, la testa della run successiva o la coda della run precedente di simboli  $P[i-1]$ . Altrimenti si usa la threshold relativa al carattere  $P[i-1]$ , la cui posizione viene denotata  $t$ . Qualora si ha che  $q < t$  si procede scegliendo la coda della run precedente mentre, avendo  $q \geq t$ , si seleziona la testa della run successiva. La scelta basata sulla posizione della threshold è dettata dal fatto che, in tal modo, si seleziona, di volta in volta, il suffisso più lungo che presenta un match con il suffisso, esteso a sinistra con  $P[i-1]$ , del pattern. Una volta scelto il nuovo  $q$  si procede con il mapping come in 2.

DC ridire meglio e dimostrare

5. si itera fino ad esaurimento del pattern

Lo pseudocodice è visualizzabile all'algoritmo 2.1.

Questa pubblicazione è stata uno dei punti di partenza per riadattare quanto studiato sulla *RLBWT* classica al fine di ottenere risultati analoghi per la *RLPBWT*. Per ulteriori dettagli sull'implementazione, sul calcolo delle *threshold* e sui risultati sperimentali si rimanda direttamente al paper di *MONI* [7].

**Algoritmo 2.1** Algoritmo di Bannai per il calcolo dell'array delle matching statistics tra un pattern  $P$  e un testo  $T$ . Per semplicità si ignorano i casi in cui  $q$  non è definito. Si assume inoltre che  $P[m-1]$  occorre in  $T$ . Con  $LF(\cdot)$  si intende il calcolo dell'*LF-mapping*.

```

1: function COMPUTE_MS( $P, T, SA_T, BWT_T$ )
2:    $MS \leftarrow [(pos : 0, len : 0) \dots (0, 0)]$   $\triangleright |P| = m, |T| = n, |MS| = m$ 
3:    $q \leftarrow$  posizione dell'ultima occorrenza di  $P[m-1]$  in  $BWT_T$ 
4:    $pos \leftarrow SA_T[q]$ 
5:   for every  $i \in [0, m-1]$  do
6:     if  $BWT_T[q] \neq P[i]$  then
7:       if  $BWT_T[q]$  è prima della relativa threshold per  $P[i]$  then
8:          $q \leftarrow$  posizione dell'occorrenza precedente di  $P[i]$  in  $BWT_T$ 
9:       else
10:         $q \leftarrow$  posizione dell'occorrenza successiva di  $P[i]$  in  $BWT_T$ 
11:       $pos \leftarrow SA_T[q]$ 
12:       $MS[i].pos \leftarrow pos$ 
13:       $q \leftarrow LF(q), pos \leftarrow pos - 1$ 
14:   for every  $i \in [0, m-1]$  do
15:      $MS[i].len \leftarrow MS[i-1].len - 1$ 
16:     while  $P[i + MS[i].len] = T[MS[i].pos + MS[i].len]$  do
17:        $MS[i].len \leftarrow MS[i].len + 1$ 
18:   return  $MS$ 

```

DC Sistemare pseudo Bannai

### 2.4.4 Uso delle LCE query

Nel 2021, Boucher, Gaggie, Rossi et al. proposero un ulteriore miglioramento di quanto fatto in *MONI*, con **PHONI: Streamed Matching Statistics with Multi-Genome References**.

In questo progetto non solo si sostituì l'uso delle *thresholds* con l'uso delle **LCE query**, riducendo l'algoritmo ad un solo *sweep* sull'array delle matching statistics (permettendo un uso "online" dell'algoritmo), ma si esplicitò anche l'uso delle funzioni  $\varphi$  e  $\varphi^{-1}$  e dell' $PLCP_T$  per il riconoscimento di tutte le occorrenze di ogni *MEM* tra un pattern e un testo, nel modo riportato all'algoritmo 2.2 [8].

A tal fine si sfrutta infatti il seguente teorema [6].

**Teorema 4.** *Dato un testo  $T$ , tale che  $|t| = n$ , si può memorizzare  $T$  in  $\mathcal{O}(r)$ , con  $r$  numero di run, tale che, dato un indice  $p \in \{0, n-1\}$  si possano computare  $\varphi(p)$ ,  $\varphi^{-1}(p)$  e  $PLCP[p]$  in tempo:*

$$\mathcal{O}(\log \log n) \quad (2.37)$$

Si è quindi potuto migliorare e semplificare l'**algoritmo di Bannai** usato in *MONI*. Infatti, sfruttando le *LCE query*, avendo il testo  $T$  in memoria sotto forma di *SLP*, è possibile computare contemporaneamente sia i vari  $MS[i].pos$  che i vari  $MS[i].len$ . Infatti, a differenza di quanto visto in *MONI*, qualora si debba effettuare la scelta la si fa in base al risultato delle *LCE query*, che comporta anche il calcolo di  $MS[i].len$ . Sfruttando, infatti,  $MS[i+1].len$  e la lunghezza del risultati dell'*LCE query* è possibile tenere conto di eventuali overlap tra i match e computare correttamente  $MS[i].len$ . Alternativamente, qualora si possa proseguire avendo un match tra  $BWT_T[q]$  e  $P[i-1]$ , il calcolo  $MS[i].len$  avviene a partire da  $MS[i+1].len$ , incrementandolo di uno avendo aggiunto un carattere a sinistra. Inoltre, come nel caso dell'*algoritmo di Bannai*, si ha anche il computo dei *MEM* nel momento in cui si hanno a disposizione i valori  $MS[i].len$ . Il riconoscimento della run a cui appartiene un certo indice e degli indici delle teste delle run avviene tramite l'uso di *bitvector*. Infatti,  $\forall \sigma \in \Sigma$ , con  $\Sigma$  alfabeto in uso, si ha un *bitvector*  $B_\sigma$ , lungo  $n$  (ovvero quanto il testo), tale che:

$$B_\sigma[i] = \begin{cases} 1 & \text{se } BWT_T[i] = \sigma \\ 0 & \text{altrimenti} \end{cases} \quad (2.38)$$

L'algoritmo 2.3 [8] riporta il calcolo completo dell'array delle matching statistics presente in *PHONI*, la cui complessità temporale è stimata in:

$$\mathcal{O}(m \log \log n) \quad (2.39)$$

Per ulteriori approfondimenti si rimanda al paper di riferimento [8].

**DC** Spiegare molto meglio e verificare tempo

---

**Algoritmo 2.2** Algoritmo per il calcolo della lista di tutte le occorrenze di una sottostringa del pattern,  $P[i, j]$ , in un testo  $T$ , a partire dall'array delle matching statistics  $MS$ .

---

```

1: function ALL_OCC( $MS, i, j, P, T$ )
2:   if  $MS[i].len < j - i + 1$  then
3:     return
4:    $p \leftarrow MS[i].pos$ 
5:    $occ \leftarrow []$ 
6:    $push(occ, p)$ 
7:   while  $PLCP[p] \geq j - i + 1$  do
8:      $p \leftarrow \varphi(p)$ 
9:      $push(occ, p)$ 
10:   $p \leftarrow \varphi^{-1}(MS[i].pos)$ 
11:  while  $p \neq null \wedge PLCP[p] \geq j - i + 1$  do
12:     $push(occ, p)$ 
13:     $p \leftarrow \varphi^{-1}(p)$ 
14:  return  $occ$ 

```

---



---

**Algoritmo 2.3** Algoritmo per il calcolo dell'array delle matching statistics in *PHONI*. Per semplicità si ignorano i casi in cui  $q, q'$  e  $q''$  non sono definiti. Si assume inoltre che  $P[m-1]$  occorre in  $T$ . Con  $LF(\cdot)$  si intende il calcolo dell'*LF-mapping*.

---

```

1: function COMPUTE_MS( $P, T, SA_T, BWT_T$ )
2:    $MS \leftarrow [(pos : 0, len : 0) \dots (0, 0)]$   $\triangleright |P| = m, |T| = n, |MS| = m$ 
3:    $q \leftarrow select_{P[m-1]}(1)$ 
4:    $MS[m-1] \leftarrow (SA_T[q] - 1, 1), q \leftarrow LF(q)$ 
5:   for  $i = m - 2$  to  $0$  do
6:     if  $BWT_T[q] = P[i]$  then
7:        $MS[i] \leftarrow (MS[i+1].pos - 1, MS[i+1].len + 1), q \leftarrow LF(q)$ 
8:     else
9:        $c \leftarrow rank_{P[i]}(q)$ 
10:       $q' \leftarrow select_{P[i]}(c), q'' \leftarrow select_{P[i]}(c + 1)$ 
11:       $l' \leftarrow \min(MS[i+1].len, |LCE(SA_T[q'], MS[i+1].pos)|)$ 
12:       $l'' \leftarrow \min(MS[i+1].len, |LCE(SA_T[q''], MS[i+1].pos)|)$ 
13:      if  $l' \geq l''$  then
14:         $MS[i] \leftarrow (SA_T[q'] - 1, l' + 1), q \leftarrow LF(q')$ 
15:      else
16:         $MS[i] \leftarrow (SA_T[q''] - 1, l'' + 1), q \leftarrow LF(q'')$ 
17:  return  $MS$ 

```

---

**Esempio 14.** Si ripropone nuovamente l'esempio con (anche se il pattern ha un carattere in più rispetto agli esempi precedenti):

$$T = \text{mississippi\$} \text{ e } P = \text{miss}$$

Studiando il calcolo dell'array delle matching statistics sia con MONI che con PHONI. Si assume che  $|T| = n$  e  $|P| = m$ .

Si hanno, avendo in ultima colonna i 5 bitvector relativi alle threshold per ogni simbolo:

Indice	SA <sub>T</sub>	F <sub>T</sub>	BWM <sub>T</sub>	BWT <sub>T</sub>	B <sub>\$</sub>	B <sub>i</sub>	B <sub>m</sub>	B <sub>p</sub>	B <sub>s</sub>	\$imps
0	11	\$	\$mississippi	i	0	1	0	0	0	11111
1	10	i	i\$mississipp	p	0	0	0	1	0	01000
2	7	i	ippi\$mississ	s	0	0	0	0	1	00000
3	4	i	issippi\$miss	s	0	0	0	0	0	00000
4	1	i	ississippi\$m	m	0	0	1	0	0	00000
5	0	m	mississippi\$	\$	1	0	0	0	0	00011
6	9	p	pi\$mississip	p	0	0	0	1	0	00000
7	8	p	ppi\$mississi	i	0	1	0	0	0	00000
8	6	s	sippi\$missis	s	0	0	0	0	1	01000
9	3	s	sissippi\$mis	s	0	0	0	0	0	00000
10	5	s	ssippi\$missi	i	0	1	0	0	0	00000
11	2	s	ssissippi\$mi	i	0	0	0	0	0	00000

DC verificare bene threshold

Iniziamo con l'algoritmo visto in MONI.

Si ha che  $P[m-1] = 's'$ , ne segue, seguendo la stessa notazione vista sopra e cercando l'ultima occorrenza di 's' in  $BWT_T$ , che  $q = 9$ . Si procede quindi con l'LF-mapping avendo che  $LF(9) = 11$ . A questo punto si ha il valore di  $MS[m-1].pos$ :

$$MS.pos = ???SA_T[11] = ???2$$

Si ha poi che  $BWT_T[11] = 'i' \neq P[m-2] = 's'$ . Non avendo alcuna run di simboli 's' sotto l'attuale run di simboli 'i' si procede aggiornando  $q$  con l'indice della cosa della precedente run di simboli 's', avendo quindi  $q = 9$ . Si procede con l'LF-mapping avendo che  $LF(9) = 11$  e si aggiorna l'array  $MS.pos$ :

$$MS.pos = ??SA_T[11]2 = ??22$$

Si ha, a questo punto, che  $BWT_T[11] = 'i' = P[m-2] = 's'$ . Si procede con l'LF-mapping, ottenendo  $LF(11) = 4$  e aggiornando  $MS$ :

$$MS.pos = ?SA_T[4]22 = ?122$$

Infine, avendo  $BWT_T[4] = 'i' \neq P[m-3] = 'm'$  si conclude il calcolo dell'array  $MS$  con l'ultimo LF-mapping. Infatti  $LF(4) = 5$ , avendo quindi:

$$MS.pos = SA_T[5]122 = 0122$$



A questo punto, tramite random access al testo, si calcolano i vari  $MS.len$ . Partendo da sinistra, si calcola per primo  $MS[i].len$  con  $i = 0$ , cercando il più lungo prefisso comune tra  $P[i, m - 1] = miss$  e  $T[MS[0].len, m - 1 - i] = miss$ , che è, in questo caso, lungo 4. Si procede per tutti i valori di  $MS.pos$ , ottenendo:

$i$	0	1	1	3
$P$	m	i	s	s
$pos$	0	1	2	2
$len$	4	3	2	1

Avendo che  $P[0, 4 - 1] = P[0, 3]$  è un MEM di  $P$  in  $T$ .

Si passa ora al calcolo tramite PHONI.

Si inizia avendo  $q = select_{P[m-1]}(1) = select_{s'}(1) = 2$ , ovvero ponendo  $q$  pari all'indice della prima occorrenza di  $P[m - 1]$  in  $BWT_T$ . Seguendo l'algoritmo si ottiene, essendo  $SA_T[2] = 7$ , quindi:

$i$	0	1	1	3
$P$	m	i	s	s
$pos$	?	?	?	6
$len$	?	?	?	1

Si procede con l'LF-mapping, avendo  $LF(2) = 8$ . Si ha che  $BWT_T[8] = P[m - 2]$  e quindi, essendo  $SA_T[8] = 6$  si ottiene:

$i$	0	1	1	3
$P$	m	i	s	s
$pos$	?	?	5	6
$len$	?	?	2	1

Anche in questo caso, essendo  $LF(8) = 10$  ed essendo  $BWT_T[10] = P[m - 3]$ , si aggiornano senza ulteriori passaggi i valori dell'array delle matching statistics:

$i$	0	1	1	3
$P$	m	i	s	s
$pos$	?	4	5	6
$len$	?	3	2	1

Infine, avendo che  $LF(10) = 3$ , si ha  $BWT_T[3] \neq P[m - 4]$ . In questo caso si potrebbe ottimizzare il calcolo del nuovo indice, sapendo che è presente una sola

occorrenza del carattere desiderato, 'm', in  $BWT_T$ , ma, ai fini dell'esempio, si mostra il calcolo completo. Innanzitutto bisogna capire quanti caratteri  $P[m-4] = 'm'$  si hanno prima di  $q = 3$ . Si ha che  $\text{rank}_{m'}(3) = 0$ . A questo punto si selezionano, tramite  $\text{select}_{m'}$ , l'indice della testa della run precedente di caratteri 'm' (che in questo caso non esiste e gli si assegna il valore 0) e della run successiva:

$$q' = \text{select}_{m'}(3) = 0$$

$$q'' = \text{select}_{m'}(4) = 4$$

Seguendo l'algoritmo si ha che:

$$l' = \min(3, |\text{LCE}(SA_T[0], 4)|) = \min(3, |\text{LCE}(11, 4)|) = \min(3, 0) = 0$$

$$l'' = \min(3, |\text{LCE}(SA_T[4], 4)|) = \min(3, |\text{LCE}(1, 4)|) = \min(3, 4) = 3$$

Avendo  $l'' \geq l'$  si aggiorna  $MS$  di conseguenza, avendo  $SA_T[q''] = SA_T[4] = 1$ :

$i$	0	1	1	3
$P$	m	i	s	s
$pos$	0	4	5	6
$len$	4	3	2	1

Avendo che  $P[0, 4-1] = P[0, 3]$  è un MEM di  $P$  in  $T$ .

DC Esempio da  
sistemare

## 2.5 Trasformata di Burrows-Wheeler posizionale

Presentata nel 2014 da Richard Durbin [4], la **Positional Burrows-Wheeler Transform (PBWT)**, traducibile con *trasformata di Burrows-Wheeler posizionale*, è una struttura efficiente per la memorizzazione e l'interrogazione di **pannelli di aplotipi**.

La costruzione di tali pannelli avviene tramite il riconoscimento delle variazioni di un singolo nucleotide tra le sequenze genomiche di diversi individui, ovvero dei cosiddetti **Single-Nucleotide Polymorphism (SNP)**. Ogni variazione, identificata per un certo nucleotide in una posizione specifica, viene detto **allele**. La combinazione di tutte le *varianti alleliche*, ereditate, a meno di mutazioni, da ogni genitore, forma l'**aplotipo** di un certo individuo. Come visibile in figura 2.1 [29], la costruzione parte dai vari sequenziamenti (nell'immagine relativi a diversi cromosomi ma il procedimento è uguale partendo da diversi individui) da cui si identificano le varianti alleliche. Da queste ultime si costruiscono gli aplotipi da cui si estraggono i cosiddetti *tag SNPs*, ovvero le possibili alternative per una certa

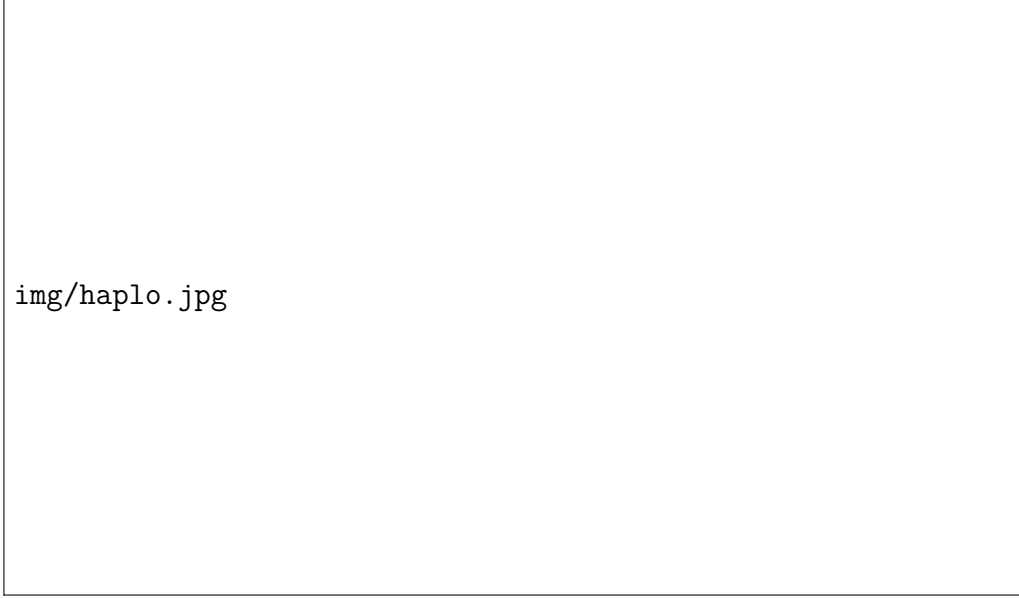


Figura 2.1: Schema di ottenimento del pannello di aplotipi.

variante allelica. Questi ultimi, tendenzialmente rappresentati per l'uomo da due caratteri vista la sua natura *diploide*, formano, l'alfabeto del pannello. L'informazione combinata di tutti gli aplotipi in un individuo è detta, invece, **genotipo**. Formalmente si considera un pannello  $X$  di  $M$  aplotipi  $x_i$ , con  $i = 0, \dots, M-1$ , su  $N$  siti, indicizzati tramite  $k = 0, \dots, N-1$ , tale per cui tutti i siti sono considerati biallelici. Da un punto di vista computazionale, quest'ultima assunzione comporta che il pannello  $X$  è costruito sull'alfabeto ordinato  $\Sigma = \{0, 1\}$ , con  $0 \prec 1$ . Si ha la sostituzione dei *tag SNPs*, per un certo sito, con tale alfabeto. Ne segue che:

$$x_i[k] = \{0, 1\} \quad (2.40)$$

Prima di proseguire con la trattazione è bene fornire la descrizione di alcuni formalismi utilizzati:

- si denota, per una qualsiasi riga  $x_i$ , con  $x_i[k_1, k_2)$  la **sottostringa** di  $x_i$  che inizia alla colonna  $k_1$  e termina alla colonna  $k_2 - 1$
- date due righe  $x_i$  e  $x_j$ , si definisce un **match** tra le due righe, iniziante alla colonna  $k_1$  e terminante alla colonna  $k_2 - 1$ , sse:

$$x_i[k_1, k_2) = x_j[k_1, k_2) \quad (2.41)$$

- un match tra due righe  $x_i$  e  $x_j$ , come definito al punto precedente, è definito **localmente massimale** sse non si ha alcuna estensione a destra o sinistra

che comporti un ulteriore match, avendo quindi che:

$$(k_1 = 0 \vee x_i[k_1 - 1] \neq x_j[k_1 - 1]) \wedge (k_2 = N \vee x_i[k_2] \neq x_j[k_2]) \quad (2.42)$$

- comparando una sequenza  $z$  ad un pannello di aplotipi  $X$  si definisce che  $s$  ha un **set-maximal exact match (SMEM)** con  $x_i$ , che inizia alla colonna  $k_1$  e termina alla colonna  $k_2 - 1$ , sse tale match è *localmente massimale* e non si ha alcun altro match di  $z$  con un altro  $x_j$  che include ed estende l'intervallo  $[k_1, k_2)$ . Si ha che  $z$  può avere uno *SMEM*, tra  $k_1$  e  $k_2 - 1$ , con più di un aplotipo del pannello

Si noti che il match tra le due sequenze nella *PBWT* è tale sse iniziano entrambi nella stessa colonna e terminano nella stessa colonna. Questo vincolo, da cui deriva il termine “posizionale” e che, di fatto, impedisce l’uso degli algoritmi tradizionali visti con la *BWT*, è dato dal fatto che una colonna rappresenta un preciso sito di una variante genica.

La costruzione di questa struttura dati si basa, ad ogni colonna  $k$ , sul riordinamento lessicografico delle sequenze di aplotipi basato sull’ordinamento inverso dei prefissi terminanti in colonna  $k - 1$ . I valori presenti in colonna  $k$ , dopo il riordinamento, altro non sono che i valori che andranno a popolare la cosiddetta **matrice PBWT**, che rappresenta la vera e propria trasformata. Si noti che avere le sequenze ordinate in base ai prefissi invertiti alla  $k$ -esima colonna permette di identificare i match con maggior facilità in quanto, ad ogni colonna, aplotipi con suffisso comune (o prefisso comune in ordine inverso) saranno in posizioni consecutive all’interno della trasformata.

La computazione di tutti i riordinamenti non presenta difficoltà dal punto di vista computazionale in quanto, conoscendo l’ordinamento in colonna  $k$ , si può derivare facilmente l’ordinamento in colonna  $k + 1$ , studiando solo i valori riordinati alla colonna precedente.

**Definizione 21.** Dato un aplotipo  $i$ , appartenente al pannello  $X$ , e un indice di colonna  $k$ , si definisce il **prefix array**  $a_k$  come una permutazione degli indici  $0, \dots, M - 1$  tale che  $a_k[i] = j$  sse  $x_j$  è l’ $i$ -esimo aplotipo di  $X$  nell’ordinamento inverso dei prefissi ottenuto alla colonna  $k$ . Quindi  $a_k[i] = m$ , con  $m < M$ , altro non è che l’indice della sequenza  $x_m$  del pannello  $X$  da cui deriva il prefisso  $i$ -esimo nell’ordine inverso in colonna  $k$ .

Data questa definizione ne segue che la *matrice PBWT* si ottiene direttamente andando a vedere, per ogni colonna, gli indici del *prefix array* e prendendo i valori del pannello  $X$  secondo l’ordine espresso da esso.

Per comodità di rappresentazione definiamo formalmente i valori della *matrice PBWT* con il seguente formalismo:

$$y_i^k[j] = x_{a_k[i]}[j] \quad (2.43)$$

avendo quindi che  $y_i^k$  denota la sequenza  $i$ -esima secondo l'ordinamento ottenuto per la colonna  $k$ . Si può quindi accedere al valore  $j$ -esimo, ovvero il valore in colonna  $j$ , di tale sequenza. Possiamo quindi meglio spiegare perché risulti semplice computare i vari *prefix array*. Infatti, si ha che l'ordinamento degli elementi per  $a_{k+1}$  si ottiene a partire dall'ordinamento in  $a_k$ . Si considerano, infatti, i valori  $y_i^k[k]$  e la precedenza del valore 0 sul valore 1 per riordinare in modo stabile tali valori.

Come anticipato, prefissi simili saranno consecutivi nei riordinamenti fino alla colonna  $k$ -esima. Quindi, risulta utile tenere traccia della posizione iniziale dei match tra prefissi vicini.

DC Un po' confusionario

**Definizione 22.** Si definisce **divergence array** l'array  $d_k$  tale che  $d_k[i]$  è l'indice colonna iniziale del match massimale a sinistra terminante in  $k$  tra l' $i$ -esimo aplotipo e il suo precedente nell'ordinamento ottenuto alla colonna  $k$ -esima. Formalmente, dato  $i > 0$ , si definisce  $d_k[i]$  come il più piccolo  $j$  tale che:

$$y_i^k[j, k) = y_{i-1}^k[j, k) \quad (2.44)$$

Ne segue che

$$y_i^k[k-1] \neq y_{i-1}^k[k-1] \implies d_k[i] = k \quad (2.45)$$

Per definizione, non avendo una riga precedente con cui effettuare il confronto:

$$d_k[0] = k \quad (2.46)$$

Si può quindi dimostrare che l'inizio di qualsiasi match massimale, terminante in colonna  $k$ , tra qualsiasi  $y_i^k$  e  $y_j^k$ , con  $i < j$ , è calcolabile facilmente avendo che è dato da:

$$\max_{i < m \leq j} d_k[m] \quad (2.47)$$

Si noti che al posto del **divergence array** si può usare anche una variante del **Longest Common Prefix (LCP) array**.

**Definizione 23.** Si definisce **Longest Common Prefix (LCP) array** l'array  $l_k$  che, anziché memorizzare l'indice d'inizio del match massimale a sinistra da due aplotipi consecutivi nell'ordinamento ottenuto alla colonna  $k$ -esima, tiene traccia della lunghezza di tale match. Formalmente si ha, quindi, che:

$$l_k[i] = k - d_k[i] \quad (2.48)$$

Fatte queste premesse possiamo quindi fornire una definizione formale di **PB-WT**.

**Definizione 24.** Dato un pannello di  $M$  aplotipi con  $N$  siti  $X = \{x_1, x_2, \dots, x_M\}$ , si definisce la **PBWT** di  $X$  come una collezione di  $N + 1$  coppie di array  $(a_k, d_k)$ , con  $0 \leq k \leq N$ , dove ogni  $a_k$  è detto **prefix array** e ogni  $d_k$  è detto **divergence array**.

L'algoritmo per la costruzione di  $a_{k+1}$  e  $d_{k+1}$  a partire da  $a_k$  e  $d_k$  è disponibile all'algoritmo 2.4. Si può quindi notare come il costo della costruzione dei due insiemi di array sia:

$$\mathcal{O}(NM) \quad (2.49)$$

Ai fini della trattazione dell'algoritmo di match con un'aplotipo esterno ricordiamo

**DC** Serve altro?  
Serve spiegare i  
dettagli dell'al-  
goritmo?

**Algoritmo 2.4** Algoritmo di Durbin per la costruzione di  $a_{k+1}$  e  $d_{k+1}$  a partire da  $a_k$  e  $d_k$ .

---

```

1: function BUILDPREFIXANDDIVERGENCEARRAYS( $k, M, a_k, d_k$ )
2:    $u \leftarrow 0, v \leftarrow 0$ 
3:    $p \leftarrow k + 1, q \leftarrow k + 1$ 
4:    $a \leftarrow [], b \leftarrow [], d \leftarrow [], e \leftarrow []$ 
5:   for every  $i \in [0, M - 1]$  do
6:     if  $d_k[i] > p$  then
7:        $p \leftarrow d_k[i]$ 
8:     if  $d_k[i] > q$  then
9:        $q \leftarrow d_k[i]$ 
10:    if  $y_i^k[k] = 0$  then
11:       $a[u] \leftarrow a_k[i], d[u] \leftarrow p$ 
12:       $u \leftarrow u + 1, p \leftarrow 0$ 
13:    else
14:       $b[v] \leftarrow a_k[i], e[v] \leftarrow q$ 
15:       $v \leftarrow v + 1, q \leftarrow 0$ 
16:   $a_{k+1} \leftarrow \text{concatenate}(a, b)$ 
17:   $d_{k+1} \leftarrow \text{concatenate}(d, e)$ 

```

---

un'ulteriore definizione.

**Definizione 25.** Si definisce  $\alpha_k$  come l'inverso della permutazione data dal **prefix array**  $a_k$ , avendo che:

$$\alpha_k[i] = j \iff a_k[j] = i$$

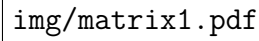
Grazie a queste prime definizioni è possibile notare alcune prime forti correlazioni, fattore chiave nello sviluppo di questa tesi, che sussistono tra *BWT* e *PBWT* (e le rispettive varianti *run-length encoded*). Nella seguente tabella si ricordano queste prime correlazioni:

BWT	PBWT
$SA_T$	$a_k$
$ISA_T$	$\alpha_k$
$LCP_T$	$d_k$ o $l_k$

**Esempio 15.** Si assuma il seguente pannello  $X$  e di voler calcolare  $y^6$ :

X	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14
00	1	0	0	1	0	0	0	0	0	0	0	1	1	0	1
01	1	0	0	1	1	0	0	1	0	0	0	0	0	1	1
02	1	0	0	1	1	0	0	1	0	0	0	1	0	0	1
03	1	0	0	1	1	0	0	1	0	0	0	1	0	0	1
04	0	1	0	1	0	1	0	0	0	0	0	1	0	0	1
05	0	1	0	1	0	1	0	0	0	0	0	1	0	0	1
06	0	1	0	1	0	1	0	0	0	0	0	1	0	0	1
07	0	1	0	1	0	1	0	0	0	0	0	0	1	0	1
08	0	1	0	0	1	0	0	0	0	1	1	1	0	0	1
09	0	1	0	1	0	0	0	0	1	0	0	0	0	1	1
10	0	1	0	1	0	0	0	0	1	0	0	0	0	1	1
11	0	1	0	0	1	0	0	0	0	0	1	1	0	0	0
12	0	1	0	0	1	0	0	0	1	0	1	1	0	0	1
13	0	1	0	0	1	0	0	0	1	0	1	1	0	0	1
14	0	1	0	0	0	0	0	0	1	0	0	0	1	0	1
15	0	1	0	0	0	0	0	0	1	0	0	0	1	0	1
16	0	1	0	1	0	0	0	0	0	0	0	1	1	0	1
17	1	1	0	0	0	1	0	0	0	0	0	1	1	0	1
18	0	1	1	0	1	0	0	0	0	0	0	1	0	0	1
19	0	1	1	0	1	0	1	0	0	0	0	0	1	0	1

Si inizia riordinando il pannello con l'ordine inverso alla quinta colonna, avendo che  $y^6$  altro non è che la sesta colonna del pannello così riordinato. Ne segue che  $a_6$  è la colonna degli indici, che è stata ottenuta con la permutazione data dall'ordinamento, e  $d_6$  la colonna iniziale in cui terminano i match tra righe consecutive nel rioridnamento le sottolineature (evidenziati nell'immagine seguente dalle sottolineature):



Avendo, nel complesso:

$$a_6 = [14, 15, 0, 9, 10, 16, 8, 11, 12, 13, 18, 19, 1, 2, 3, 17, 4, 5, 6, 7]$$

$$\alpha_6 = [2, 12, 13, 14, 16, 17, 18, 19, 6, 3, 4, 7, 8, 9, 0, 1, 5, 15, 10, 11]$$

$$d_6 = [6, 0, 4, 2, 0, 0, 5, 0, 0, 0, 3, 0, 4, 0, 0, 6, 4, 0, 0, 0]$$

$$l_6 = [0, 6, 2, 4, 6, 6, 1, 6, 6, 6, 3, 6, 2, 6, 6, 0, 2, 6, 6, 6]$$

Con il calcolo di tutti gli  $a_k$  si otterrebbe la seguente **matrice PBWT**:

X	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14
00	1	1	0	1	1	0	0	0	1	0	0	1	1	1	1
01	1	1	0	1	1	0	0	0	1	0	0	1	1	1	1
02	1	1	0	1	1	1	0	0	0	1	1	1	0	1	1
03	1	1	0	1	1	0	0	0	1	0	0	1	1	0	1
04	0	1	0	1	0	1	0	0	1	0	0	1	1	0	1
05	0	1	0	1	0	1	0	0	0	0	0	1	0	0	1
06	0	1	0	1	0	1	0	0	0	0	0	1	0	0	0
07	0	1	0	1	1	1	0	0	0	0	0	0	1	0	1
08	0	1	0	0	1	0	0	0	1	0	0	0	1	0	1
09	0	1	0	1	0	0	0	0	1	0	0	0	0	0	1
10	0	1	0	1	1	0	0	0	0	0	0	1	1	0	1
11	0	1	0	0	1	0	1	1	0	0	0	1	0	0	1
12	0	1	0	0	1	0	0	1	0	0	0	0	0	0	1
13	0	1	0	0	0	0	0	1	0	0	0	0	0	0	1
14	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
16	0	0	0	1	0	0	0	0	0	0	0	1	0	0	1
17	1	0	1	0	0	0	0	0	0	0	1	1	0	0	1
18	0	0	1	0	0	0	0	0	0	0	1	1	0	0	1
19	0	1	0	0	0	0	0	0	0	0	1	1	0	0	1



### 2.5.1 Match massimali con aplotipo esterno

Durbin, nel suo articolo, propone diversi algoritmi per l'uso effettivo della sua trasformata. Ad esempio, viene proposto un algoritmo per il calcolo di match interni ad  $X$  più lunghi di una lunghezza minima  $L$  e uno per la ricerca di tutti i *set-maximal match* interni ad  $X$  in tempo lineare.

Di interesse per questa tesi è però il cosiddetto **algoritmo 5**. Tale algoritmo si propone di trovare tutti i *set-maximal match* tra il pannello  $X$  e un aplotipo esterno  $z$ , assumendo  $|z| = N$ , sempre avendo che una colonna  $k$  del pannello e una posizione  $k$  della query rappresentano il medesimo sito.

L'idea dietro l'algoritmo è quella di usare tre indici:  $e_k$ ,  $f_k$  e  $g_k$ . Nel dettaglio  $e_k$  tiene traccia dell'inizio del più lungo match, terminante in colonna  $k$ , tra  $z$  e un qualche  $y_i^k$ . L'intervallo  $[f_k, g_k) \subseteq [0, \dots, M)$  invece identifica il sotto-intervallo di  $a_k$  contenente gli indici degli aplotipi appartenenti a tale match. Si noti come si riprenda quindi l'idea, vista con la **backward search** per la *BWT*, di studiare un intervallo  $[f_k, g_k)$  su  $SA_T$  per identificare i match tra un pattern e un testo.

**Definizione 26.** Dato un pannello  $X$ , con  $M$  aplotipi/righe e  $N$  siti/colonne, e un aplotipo query  $z$ , tale che  $|z| = N$ , si definisce un **Set-Maximal Exact Match (SMEM)**, iniziante in colonna  $e_k$  e terminante il colonna  $k$ , tra la query  $z$  e le righe del pannello indicizzate dai valori compresi nell'intervallo  $[f_k, g_k)$  in  $a_k$  sse:

$$z[e_k, k) = y_i^k[e_k, k) \wedge z[e_k - 1] \neq y_i^k[e_k - 1], \forall i \text{ t.c. } f_k \leq i < g_k \quad (2.50)$$

Si noti che  $g_k = M$  sse  $y_{M-1}^k$  appartiene alle righe per le quali si ha tale SMEM.

Bisogna quindi come aggiornare  $e_k$ ,  $f_k$  e  $g_k$  passando dalla colonna  $k$  alla colonna  $k+1$ . Si procede esattamente come visto per la **backward search**, selezionando, per calcolare  $[f_{k+1}, g_{k+1})$ , il sottointervallo di  $[f_k, g_k)$  in cui si hanno aplotipi che possono essere estesi a destra con il simbolo  $z[k+1]$ . L'idea è quella per cui, avendo  $f_{k+1} < g_{k+1}$  allora sicuramente ho ancora delle righe che presentano un match che parte da  $e_{k+1} = e_k$  e termina in  $k$  che può essere esteso in  $k+1$ . In caso contrario, avendo  $f_{k+1} = g_{k+1}$ , non si hanno match estendibili e quindi si può concludere che quelli terminanti in colonna  $k$  erano match massimali. In questo secondo caso bisogna poi aggiornare  $e_{k+1}$ , ottenendo i relativi  $f_{k+1}$  e  $g_{k+1}$ , al fine di trovare la nuova colonna da cui parte lo SMEM successivo e le righe del pannello per le quali si ha tale SMEM.

Bisogna, quindi, capire come funzionino la variante del **backward-step** visto per la *BWT*. Tale mapping, guidato dal carattere corrente dell'aplotipo query, permette di ottenere  $f_{k+1}$  e  $g_{k+1}$  a partire da  $f_k$  e  $g_k$ .

Per effettuare il mapping abbiamo bisogno di tre componenti, che, intuitivamente, svolgono la medesima funzione di  $C$  e  $Occ$  per la *BWT*:

1. l'array  $c$  tale per cui  $c[k] = j$  sse la colonna  $k$  contiene  $j$  occorrenze di 0
2. l'array  $u_k$  tale per cui, alla colonna  $k$ -esima,  $u_k[i] = j$  sse  $j$  è il numero di occorrenze di 0 prima dell'indice  $i$  nella colonna  $k$
3. l'array  $v_k$  tale per cui, alla colonna  $k$ -esima,  $v_k[i] = j$  sse  $j$  è il numero di occorrenze di 1 prima dell'indice  $i$  nella colonna  $k$

Tali valori possono essere computati e memorizzati in fase di costruzione della **PBWT**, come visibile direttamente nell'algoritmo 2.4 per quanto riguarda  $u$  e  $v$ . Per quanto riguarda  $c$  si ha che potrebbe essere banalmente calcolato anch'esso in fase di costruzione della **PBWT**, tenendo ogni volta traccia del numero di 0 incontrati nella colonna  $k$ -esima.

Sfruttando i valori di questi 3 array possiamo quindi effettuare lo step/mapping alla colonna successiva, definito, per comodità, da una funzione:

$$w_k : \{0, \dots, M\} \times \Sigma \rightarrow \{0, \dots, M\} \quad (2.51)$$

tale per cui:

$$w_k(i, \sigma) = \begin{cases} u_k[i] & \text{se } \sigma = 0 \\ v_k[i] + c[k] & \text{se } \sigma = 1 \end{cases} \quad (2.52)$$

Tale funzione è rappresentabile in pseudocodice come nell'algoritmo 2.5.

Risulta interessante notare, come confermato anche dall'algoritmo di costruzione stesso, che si ha:

$$a_{k+1} [w_k(i, y_i^k[k])] = a_k[i] \quad (2.53)$$

Avendo che tale mapping permette di “seguire” una determinata riga all'interno delle varie permutazioni dettate dai vari  $a_k$ .

---

**Algoritmo 2.5** Algoritmo per il mapping nella PBWT.

---

```

1: function  $w(k, i, s, c_k, u_k, v_k)$ 
2:   if  $s = 0$  then
3:     return  $u_k[i]$ 
4:   else
5:     return  $c_k + v_k[i]$ 
```

---

**Esempio 16.** Vediamo un piccolo esempio chiarificatore, riprendendo l'esempio 15, ricordando che:

$$a_5 = [14, 15, 17, 0, 4, 5, 6, 7, 9, 10, 16, 8, 11, 12, 13, 18, 19, 1, 2, 3]$$

$$\alpha_5 = [3, 17, 18, 19, 4, 5, 6, 7, 11, 8, 9, 12, 13, 14, 0, 1, 10, 2, 15, 16]$$

$$a_6 = [14, 15, 0, 9, 10, 16, 8, 11, 12, 13, 18, 19, 1, 2, 3, 17, 4, 5, 6, 7]$$

$$\alpha_6 = [2, 12, 13, 14, 16, 17, 18, 19, 6, 3, 4, 7, 8, 9, 0, 1, 5, 15, 10, 11]$$

Si ha, ad esempio, con  $k = 5$  e  $i = 2$ , che:

$$a_6[w_5(2, y_2^5[5])] = a_5[2]$$

Avendo:

$$w_5(2, y_2^5[5]) = w_5(2, 1) = v_5[2] + c[5] = 0 + 15 = 15$$

Si ha che:

$$a_6[15] = 17 = a_5[2]$$

Dimostrando come con tale funzione si possa, di fatto, “seguire” la riga 17, capendo da quale posizione arrivi della colonna permutata precedente.

Pensando alla permutazione inversa del **prefix array**, si ottiene un risultato interessante, permettendo di ottenere risultati per la colonna  $k + 1$  a partire dalla colonna  $k$ -esima:

$$\alpha_{k+1}[i] = w_k(\alpha_k[i], x_i[k]) \quad (2.54)$$

**Esempio 17.** Si riprendono i dati dell'esempio precedente e si vuole calcolare, sempre con  $k = 5$  e  $i = 2$ :

$$\alpha_6[2] = w_5(\alpha_5[2], x_2[5]) = w_5(18, 0) = 13$$

Come volevasi dimostrare.

L'ultima equazione ci suggerisce quindi che la funzione  $w$  consente il corretto aggiornamento di  $f_k$  e  $g_k$ . Definendo, infatti:

$$f_{k+1} = w_k(f_k, z[k]) \quad (2.55)$$

si ha che  $f_{k+1}$  sarà l'indice, in  $a_{k+1}$ , della prima sequenza  $y_j^k$ , con  $j \geq f_k$ , per la quale  $y_j^k[k] = z[k]$ . Analogamente, pensando alla prima sequenza per cui si ha un mismatch dopo l'aggiornamento dell'intervallo, si calcola:

$$g_{k+1} = w_k(g_k, z[k]) \quad (2.56)$$

Si hanno quindi, dopo il calcolo dei potenziali  $f_{k+1}$  e  $g_{k+1}$  due possibili casi:

1. si ha che  $f_{k+1} < g_{k+1}$ . In questo caso si hanno ancora match che partono da  $e_k$  e terminano in  $k$  che si estendono anche in  $k + 1$ . In altri termini si ha che un sottointervallo non nullo di  $[f_k, g_k)$  è relativo a righe che presentano  $z[k + 1]$  come simbolo in colonna  $k + 1$ . In tal caso, si prosegue con l'iterazione, avendo  $e_{k+1} = e_k$

**DC** Capire se dire altro su  $w()$

2. si ha che  $f_{k+1} = g_{k+1}$ . In questo caso non si hanno match che partono da  $e_k$  e terminano in  $k$  che sono anche estendibili in  $k+1$ . Bisogna quindi annotare i match terminanti in  $k-1$ , nell'intervallo  $[f_k, g_k)$  su  $a_k$ , e poi ricalcolare i nuovi  $e_{k+1}$ ,  $f_{k+1}$  e  $g_{k+1}$ . Il punto fondamentale per poter calcolare i nuovi indici è che, virtualmente, l'aplotipo  $z$  si trova, in colonna  $k$ , o subito prima o subito dopo il blocco di aplotipi indicizzati da  $[f_k, g_k)$  su  $a_k$ , secondo l'ordinamento dato dalla medesima colonna. Di conseguenza si può inferire che, essendo  $z$  nell'ordinamento in  $k$  o subito prima di  $f_k$  o subito dopo  $g_k$  ed avendo  $f_{k+1} = g_{k+1}$ :

$$y_{f_{k+1}-1}^{k+1} \prec z \prec y_{f_{k+1}}^{k+1} \quad (2.57)$$

Ne segue direttamente che:

$$e_{k+1} \leq d_{k+1}[f_{k+1}] \quad (2.58)$$

Avendo che il nuovo indice di partenza del match sarà almeno nella colonna indicata da  $d_{k+1}[f_{k+1}]$ , essendo esso calcolato tra  $y_{f_{k+1}-1}^{k+1}$  e  $y_{f_{k+1}}^{k+1}$ , tra le quali sequenze è virtualmente compresa la query  $z$ .

Si considera quindi, come punto di partenza:

$$e_{k+1} = d_{k+1}[f_{k+1}] - 1 \quad (2.59)$$

Studiando, di conseguenza,  $z[e_{k+1}]$ , si hanno due casi possibili, dati dal fatto che, per la nozione di *divergence array* e di ordinamento dei prefissi inversi con  $0 \prec 1$ :

$$y_{f_{k+1}-1}^{k+1}[e_{k+1}] = 0 \neq y_{f_{k+1}}^{k+1}[e_{k+1}] = 1 \quad (2.60)$$

Si ha quindi che:

- (a) se tale valore è 0 allora  $z$  ha un match migliore con  $y_{f_{k+1}-1}^{k+1}$  rispetto che con  $y_{f_{k+1}}^{k+1}$ . Si aggiorna quindi  $e_{k+1}$ , decrementandolo, fino a che si ha match tra  $z[e_{k+1} - 1]$  e  $y_{f_{k+1}-1}^{k+1}[e_{k+1} - 1]$ . Infine si decrementa  $f_{k+1}$  fino a che  $d_{k+1}[f_{k+1}] \leq e_{k+1}$ , trovando quelle righe per il quale il *divergence array* non supera il valore di  $e_{k+1}$ . Si ottengono in tal modo le sequenze, nel riordinamento in  $k+1$ , che hanno un match da  $e_{k+1}$  a  $k+1$ . Invece  $g_{k+1}$  resta fisso, avendo che  $y_{g_{k+1}}^{k+1}$  presenta un mismatch in colonna  $k+1$
- (b) se tale valore è 1 allora, per l'ordinamento,  $z$  ha un match migliore con  $y_{f_{k+1}}^{k+1}$  rispetto che con  $y_{f_{k+1}-1}^{k+1}$ . Si aggiorna quindi  $e_{k+1}$ , decrementandolo, fino a che si ha match tra  $z[e_{k+1} - 1]$  e  $y_{f_{k+1}-1}^{k+1}[e_{k+1} - 1]$ . Infine si incrementa  $g_{k+1}$  fino a che  $d_{k+1}[g_{k+1}] \leq e_{k+1}$ , per lo stesso

ragionamento del caso precedente. Si noti che si permette di ottenere  $g_{k+1} = M$  avendo che tale valore risulta escluso in  $[f_{k+1}, g_{k+1})$ . In tal modo si segnala che la riga indicizzata con  $a_{k+1}[M - 1]$ , in colonna  $k + 1$ , presenta un match. Invece  $f_{k+1}$  resta fisso, avendo che  $y_{f_{k+1}}^{k+1}$  presenta un mismatch in colonna  $k + 1$ .

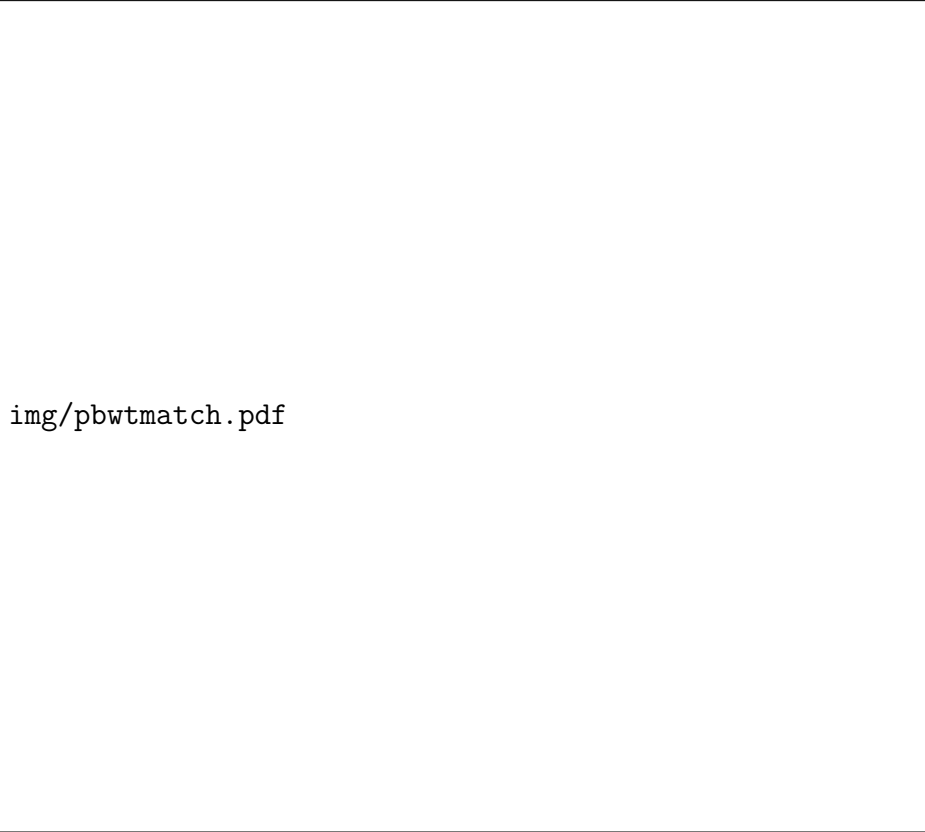
In termini di inizializzazione, per permettere il funzionamento dell'algoritmo, si hanno:

$$f_0 = g_0 = e_0 = 0$$

Quindi il primo step sarà già un caso in cui  $f_k = g_k$  qualora  $x_0[0] = y_0^0 \neq z[0]$ .

**Esempio 18.** *Mostrare un esempio completo di esecuzione richiederebbe troppo spazio quindi ci si limita a mostrare cosa succede nel caso in cui, ad un certo punto dell'esecuzione si hanno  $f_{k+1} = g_{k+1}$ .*

*Si assuma il pannello e la matrice PBWT visti all'esempio 15 con una query  $z$ . Nel complesso si identificano i seguenti match:*



img/pbwtmatch.pdf

*Si assuma di essere in colonna  $k = 6$ , avendo, dopo i calcoli fatti in colonna  $k = 5$ :*

- $f_6 = 6$
- $g_6 = 10$
- $e_6 = 0$

Avendo quindi che, a partire dalla colonne 0 fino alla colonna  $6-1=5$ , si hanno le righe nel range  $[6, 10)$  di  $a_6$  che matchano con  $z[0, 5]$ . Tali righe sono, nel dettaglio, quelle indicizzate  $\{8, 11, 12, 13\}$ .

Bisogna quindi aggiornare  $f_7$  e  $g_7$ . Si assuma che  $z[6] = 1$  e che:

$$y^6 = 000000000000100000000, \quad c[6] = 19$$

Si calcolano quindi:

$$f_7 = w_6(6, 1) = v_6[6] + c[6] = 0 + 19 = 19$$

$$g_7 = w_6(10, 1) = v_6[10] + c[6] = 0 + 19 = 19$$

Avendo quindi  $f_7 = g_7$  si procede, in primis, annotando i match terminanti in  $k = 5$ .

Seguendo l'algoritmo si ha quindi un primo aggiornamento di  $e_{k+1}$ , che viene inizializzato a, avendo in memoria  $d_7$  con random access in tempo costante:

$$e_7 = d_7[19] - 1 = 7 - 1 = 6$$

Questo viene fatto in quanto, come detto, l'aplotipo  $z$  si trova o subito prima del blocco di aplotipi  $[f_k, g_k)$ .

Essendo inoltre  $z[e_7] = z[6] = 1$  si procede aggiornando  $g_7$  e tenendo fermo  $f_7$ , avendo  $z[6] = 1$ . Si procede quindi inizializzando il nuovo  $g_7$ :

$$g_7 = f_7 + 1 = 20$$

ricordando che  $g_k$  può "superare" le dimensioni del pannello essendo escluso in  $[f_k, g_k)$ .

A questo punto si segue la linea specificata da  $f_7$  in  $a_7$  a ritroso, partendo da  $e_7 - 1$ , fino a che si hanno match con  $z$ , aggiornando così il valore di  $e_7$ .

In questo caso non si hanno altre operazioni, in quanto  $g_7 = M$  ma, qualora non lo fosse stato, si sarebbe incrementato  $g_7$  fino a che il corrispondente  $d_7[g_7]$  sarebbe stato minore o uguale di  $e_7$ , identificando tutte le nuove righe che hanno un match da con  $z[e_7, 6]$ .

DC Forse sono necessarie immagini?

L'algoritmo 5, consultabile all'algoritmo 2.6 secondo i calcoli di Durbin, ha complessità:

$$\mathcal{O}(N + c) \tag{2.61}$$

tale risultato è stimato tale in quanto si ritiene che il numero di accessi ai loop interni sia limitato dalla costante rappresentante il numero di match,  $c$ . Nonostante ciò tale complessità temporale è ancora in corso di studio in quanto si hanno in letteratura evidenze della sua non correttezza. Un esempio è il paper di Naseri [30], dove si afferma che l'intuizione per cui tale costante  $c$  limiti superiormente gli accessi ai loop innestati sia falsa. Si noti che nell'articolo non viene però precisata una nuova misura per la complessità dell'algoritmo ma solo che la stima di Durbin è empiricamente accettabile come *caso medio*:

$$\text{Avg. } \mathcal{O}(N + c) \quad (2.62)$$

In ogni caso, una soluzione naïve, impiegherebbe tempo:

$$\mathcal{O}(N^2 M) \quad (2.63)$$

Si comprende, quindi, come tale algoritmo e tale struttura siano stati rivoluzionari per lo studio di pannelli di aplotipi.

---

**Algoritmo 2.6** Algoritmo 5 di Durbin.

---

```

1: function FIND_SET_MAXIMAL_MATCHES_FROM_Z( $z$ )
2:   for  $k \leftarrow 0$  to  $N$  do
3:      $e, f, g \leftarrow \text{Update\_Z\_Matches}(k, z, e, f, g)$ 
4:
5:   function UPDATE_Z_MATCHES( $k, z, e, f, g$ )
6:      $f' \leftarrow w(k, f, z[k])$ 
7:      $g' \leftarrow w(k, g, z[k])$ 
8:     if  $f' < g'$  then ▷ se  $k$  è  $N - 1$  match da  $e_k$  a  $N - 1$ 
9:        $e' \leftarrow e_k$ 
10:    else ▷ match da  $e_k$  a  $k$ 
11:       $e' \leftarrow d_{k+1}[f'] - 1$ 
12:      if  $z[e'] = 0$  and  $f' > 0$  then
13:         $f' \leftarrow g' - 1$ 
14:        while  $z[e' - 1] = y_{f'}^{k+1}[e' - 1]$  do  $e' \leftarrow e' - 1$ 
15:        while  $d_{k+1}[f'] \leq e'$  do  $f' \leftarrow f' - 1$ 
16:      else
17:         $g' \leftarrow f' + 1$ 
18:        while  $z[e' - 1] = y_{f'}^{k+1}[e' - 1]$  do  $e' \leftarrow e' - 1$ 
19:        while  $g' < M$  and  $d_{k+1}[g'] \leq e'$  do  $g' \leftarrow g' + 1$ 
20:    return  $e', f', g'$ 

```

---

### Limiti spaziali

Bisogna affrontare la tematica della complessità in spazio di tale algoritmo. Si ipotizzi di non ricalcolare, colonna per colonna, comportando un'incremento dal punto di vista temporale, tutti gli array necessari a costruire la *PBWT* e a permettere di computare la funzione  $w(i, \sigma)$ .

Ricapitolando, per poter eseguire l'algoritmo 5, si necessita di avere in memoria, con *random access* in tempo costante:

- il **pannello**  $X$ , di dimensione  $NM$
- l'insieme dei **prefix array**  $a$ , di dimensione  $NM$
- l'insieme dei **divergence array**  $d$ , di dimensione  $NM$
- i **vettori**  $u_k$  e  $v_k$ , per ogni colonna  $k$ , complessivamente di dimensione  $2NM$
- il **vettore**  $c$ , di dimensione  $N$

Possiamo quindi dire che si ha una complessità in memoria pari a:

$$\mathcal{O}(NM) \tag{2.64}$$

Nel dettaglio, Durbin stesso ha proposto una stima quantitativa di tale memoria richiesta, ovvero<sup>3</sup>:

$$13NM \text{ bytes} \tag{2.65}$$

Per poter capire meglio la problematica conseguente a tali richieste di spazio, prendiamo, ad esempio, un pannello di medie dimensioni, con  $N = 30000$  e  $M = 100000$ . Ne segue che, secondo la stima di Durbin, si necessitano  $\sim 36.32\text{GB}$  di memoria. Inoltre, una stima sperimentale di tale richiesta di memoria può essere confermata con l'esecuzione dell'implementazione ufficiale della *PBWT*<sup>4</sup>. Infatti, monitorando con `time` il picco di memoria durante l'esecuzione, si ha che esso corrisponde a  $\sim 40.76\text{GB}$ , comprensivi anche di tutto ciò che è “a contorno” all'algoritmo stesso. I dati quindi sembrano confermare le stime di Durbin, confermando l'alto uso di memoria richiesto dall'*algoritmo 5*. Questa è stata la motivazione principale per cui si è sviluppata, in questa tesi magistrale, una versione **run-length encoded** della struttura dati che permettesse di effettuare query con un aplotipo esterno.

<sup>3</sup><https://github.com/richarddurbin/pbwt/blob/0de8d02df1b77146ded81e9e196991fdab520767/pbwtMatch.c#L252>

<sup>4</sup><https://github.com/richarddurbin/pbwt>



## 2.5.2 Varianti della PBWT

Negli anni immediatamente successivi all'articolo di Durbin, una miriade di articoli e ricerche sono state svolte per migliorare la *PBWT*, crearne varianti o utilizzarla per portare a compimento vari studi. Non essendo tali lavori direttamente correlati a questa tesi non verranno approfonditi ma, soprattutto nell'ottica dei prospetti futuri, è bene citarne i principali.

### PBWT multiallelica

La prima variante che si introduce è la **PBWT multiallelica** (*mPBWT*), proposta da Naseri et al. nel 2019 [31]. Questo lavoro estende la *PBWT* di Durbin generalizzandola ad un alfabeto arbitrario.

Dal punto di vista delle motivazioni biologiche, questa soluzione risulta fondamentale, oltre che per lo studio di specie multialleliche (soprattutto nel mondo vegetale) in quanto gli studi riportano come, nell'uomo, la presenza di siti triallelici sia sotto stimata.

Da un punto di vista prettamente algoritmico si sono quindi estesi i concetti di  $c$ ,  $u_k$  e  $v_k$  visti nella *PBWT* per ottenere un vero e proprio *FM-index* in grado di lavorare su alfabeto arbitrario  $\Sigma$ , con conseguente forte aumento dello spazio richiesto in memoria. Da un punto di vista della complessità temporale, invece, si ha che le stime asintotiche degli algoritmi devono tenere conto anche della grandezza dell'alfabeto stesso, avendo però che, essendo esso tendenzialmente di dimensioni ridotte, questo fatto non comporta, in media, particolari problematiche dal punto di vista dei tempi di calcolo. Le complessità temporali della *mPBWT* infatti sono incrementate di un fattore  $t$ , con  $t = |\Sigma|$ , e se tale valore è assunto costante ad inizio computazione, avendo che difficilmente si ha  $t \gg 2$ , la complessità temporale non subisce variazioni considerevoli.

### PBWT con struttura LEAP

Sempre nel 2019, Naseri et al.[32] proposero anche una variante della *PBWT* che permettesse il calcolo non solo dei match massimali, come per l'algoritmo 5 di Durbin, ma anche qualsiasi match di lunghezza maggiore uguale ad una lunghezza arbitraria  $L$ . Tale algoritmo fu nominato **PBWT-query**. Inoltre, nello stesso articolo, proposero una struttura dati aggiuntiva, detta **LEAP** (*Linked Equal/Alternating Position*), che, al costo della memorizzazione di otto array aggiuntivi che permettessero di effettuare dei salti nella *matrice PBWT* (salvando gli indici del precedente/prossimo valore nella colonna uguale/diverso) e di memorizzare gli indici dei valori nel *divergence array* relativi a tali indici, ottimizzava i tempi dell'algoritmo per la *PBWT-query* ottenendo l'algoritmo detto (**L-PBWT-query**). Da un punto di vista computazionale si noti che la complessità dell'algoritmo per

DC Serve dire di più sulla struttura LEAP?

la *PBWT query*, con match di lunghezza minima  $L$  è:

$$\mathcal{O}(N + c(R - L + 1)) \quad (2.66)$$

Avendo:

- $R$  lunghezza media dei match
- $c$  numero totale dei match

In merito ai tempi dell'algoritmo *L-PBWT-query* si ha invece che è, al costo di  $8NM$  interi aggiuntivi in memoria, con  $N$  e  $M$  dimensioni del pannello:

$$\mathcal{O}(N + c) \quad (2.67)$$

### PBWT dinamica

Sanaullah et al., nel 2021, proposero la **Dynamic PBWT** (*d-PBWT*) [30] col fine di superare le limitazioni imposte dalle strutture statiche usate nella *PBWT* di Durbin. Si è quindi pensato di sostituire l'uso degli array, statici, con l'uso di *linked list*, dinamiche.

Grazie alle *linked list*, si è reso possibile l'aggiornamento efficiente della *matrice PBWT* all'aggiunta di un nuovo aplotipo nel pannello o alla rimozione di uno già presente nel pannello.

Da un punto di vista computazionale, è interessante notare come le implementazioni degli algoritmi di Durbin presentino la medesima complessità asintotica. Infatti, ad esempio, la creazione della *d-PBWT* richiede tempo:

$$\mathcal{O}(NM) \quad (2.68)$$

Invece, l'aggiunta e la rimozione di un aplotipo sono entrambe in tempo:

$$Avg. \mathcal{O}(N) \quad (2.69)$$

### PBWT con wildcard

La tematica dei dati mancanti è una tematica aperta in *bioinformatica*. I sequenziatori infatti presentano un range d'errore dal 1% al 15%, si ha a volte un basso *coverage* (ovvero il numero di read che contengono la base sequenziata per un certo locus del genoma) e la fase di assemblaggio del genoma può comportare errori. Questo, in fase di produzione dei pannelli, implica che, in determinati casi, non si sappia quale sia l'allele corretto per un individuo riferendosi ad un sito.

Nel 2020, Williams e Mumey [33] proposero quindi l'uso della **PBWT con wildcard** al fine di disegnare un algoritmo in grado di trovare i match interni ad

un pannello biallelico con dati mancanti, rappresentati come *wildcard* mediante il simbolo “\*” (avendo quindi  $\Sigma = \{0, 1, *\}$ ).

In termini computazionali gli autori sono riusciti a formulare un algoritmo in grado tutti i match interni (ovvero i *blocchi*) massimali al pannello in tempo, con  $T$  numero di blocchi:

$$\mathcal{O}(NMT) \quad (2.70)$$

DC Serve dire altro?

## IMPUTE5

Per citare un uso della *PBWT* si può introdurre il concetto di **genotype imputation**, ovvero il processo con il quale si predicono genotipi non ancora osservati in un campione di individui, usando un pannello di aplotipi. Questo tipo di studio si basa sui dati prodotti dai **GWAS** (*Genome-wide association studies*), studi il cui scopo è quello di esaminare multipli genomi alla ricerca di associazioni tra varianti genetiche e malattie (o outcome specifici delle stesse), identificando varianti genomiche che sono statisticamente associate al rischio per una malattia. A tal fine, nel 2020, Rubinacci et al. proposero **IMPUTE5** [34], un metodo basato sulla *PBWT* per la *genotype imputation*, in grado di studiare pannelli di grandi dimensioni.

DC Aggiungere qualcosa?

### 2.5.3 Una prima proposta run-length encoded

A fine 2021, Gaggie et al. [35] hanno iniziato a teorizzare una variante **run-length encoded** della *PBWT*, basandosi sui risultati già ottenuti sulla *BWT* classica con la *RLBWT*.

Pensando alla costruzione della *PBWT*, con  $M$  individui e  $N$  siti, si ha che ogni colonna della *matrice PBWT* è ottenuta tramite la permutazione data dal *prefix array*. Denotiamo tale permutazione, alla colonna  $k$ , con  $\pi_k$ ,  $\forall 1 \leq k < N$ . Ipotizziamo ora di voler studiare la riga  $i$ -esima del pannello originale. Si ha che, al variare della colonna  $k$  sulla *matrice PBWT*, la posizione della riga  $i$  è ricostruibile applicando le varie permutazioni :

$$i, \pi_1(i), \pi_2(\pi_1(i)), \dots, \pi_{N-1}(\dots(\pi_2(\pi_1(i)))\dots) \quad (2.71)$$

Il punto fondamentale si ritrova nel fatto che l'autore asserisce:

*Notice  $\pi_k$  can be stored in space proportional to the number of runs in the  $k$ th column of the *PBWT*...*

Nell'articolo si propone quindi una struttura dati formata da  $N$  “tabelle” dove, la  $j$ -esima riga della  $k$  tabella contiene:

- l'indice  $p$  di inizio della  $j$ -esima run nella colonna  $k$  della *matrice PBWT*
- il valore  $\pi_k(p)$ , avendo che:

$$\pi_k(p) = \begin{cases} p - v_k[p] & \text{if } y_p^k[k] = 0 \\ c[k] + v_k[p] - 1 & \text{if } y_p^k[k] = 1 \end{cases} \quad (2.72)$$

- l'indice della run contenente il simbolo  $\pi_k(p)$  nella colonna  $k + 1$  della *matrice PBWT*
- un booleano per capire se la prima run è composta da simboli  $\sigma = 0$  o  $\sigma = 1$

Il paper presenta anche il metodo per l'estrazione della  $i$ -esima riga:

1. si cerca della prima “tabella” la riga relativa alla run, con indice di testa  $p$ , contenente l'indice  $i$ . Si noti che la prima “tabella”, relativa alla colonna  $k = 0$  non presenta permutazioni e quindi l'indice  $i$  del pannello è anche l'indice  $i$  della *matrice PBWT*
2. si calcola la permutazione per l'indice  $i$  (alla prima operazione si avrà  $k = 1$ ):

$$\pi_k(i) = \pi_k(p) + i - p \quad (2.73)$$

3. si cerca poi la riga relativa alla run contenente il simbolo  $\pi_k(p)$  nella “tabella” successiva e si scansionano le righe di tale tabella a partire da quella appena identificata fino a trovare la run che contiene  $\pi_k(i)$  (alla prima operazione si avrà  $k = 1$ ). Infine, si estrae il simbolo relativo a tale run
4. si ripete la procedura dal punto 2) per ogni colonna  $k$

Inoltre, vengono proposti ulteriori ottimizzazioni, basate sul metodo detto **fractional cascading**. Con tale rappresentazione, si riesce a ridurre il numero di run che devono essere scansionate nel passaggio 3. Questo risultato ha un tradeoff in termini di spazio. Infatti, aumentando il numero totale di righe in tutte le tabelle di un fattore al più  $(1 + \frac{1}{d})$ , è possibile garantire che si avranno al più  $d$  iterazioni, in ogni tabella, per ottenere l'estrazione del simbolo desiderato.

*Si segnala che, nel paper, non vengono specificati metodi per effettuare query a tale struttura dati, indicando solo che dovrebbe essere possibile interrogare tale struttura a tabelle.*

Per ulteriori dettagli si rimanda al paper di riferimento [35].

**Esempio 19.** *Supponendo di voler ricostruire la riga  $i = 9$ , si assuma la matrice PBWT, avendo che in rosso sono segnati i simboli appartenenti alla riga 9:*

X	01	02	03	04	05	06	07	08	09	10	11	12
00	1	1	0	0	0	1	0	0	1	1	1	1
01	1	1	0	0	0	1	0	0	1	1	1	1
02	1	1	1	0	0	0	1	1	1	0	1	1
03	1	1	0	0	0	1	0	0	1	1	0	1
04	1	0	1	0	0	1	0	0	1	1	0	1
05	1	0	1	0	0	0	0	0	1	0	0	1
06	1	0	1	0	0	0	0	0	1	0	0	0
07	1	1	1	0	0	0	0	0	0	1	0	1
08	0	1	0	0	0	1	0	0	0	1	0	1
09	1	0	0	0	0	1	0	0	0	0	0	1
10	1	1	0	0	0	0	0	0	1	1	0	1
11	0	1	0	1	1	0	0	0	1	0	0	1
12	0	1	0	0	1	0	0	0	0	0	0	1
13	0	0	0	0	1	0	0	0	0	0	0	1
14	0	0	0	0	0	0	0	0	0	0	0	1
15	0	0	0	0	0	0	0	0	0	0	0	1
16	1	0	0	0	0	0	0	0	1	0	0	1
17	0	0	0	0	0	0	0	1	1	0	0	1
18	0	0	0	0	0	0	0	1	1	0	0	1
19	0	0	0	0	0	0	0	1	1	0	0	1

Si costruiscono quindi le seguenti tabelle [35]:

img/trick.jpg

Nelle tabelle, in rosso, si hanno le varie  $\pi_k(i)$  calcolate nel processo, ottenute, se necessario, iterando a partire dalle  $\pi_k(p)$ , segnalate in azzurro.

Si hanno infatti i seguenti calcoli, ovvero i vari  $\pi_j(i) = \pi_j(p) + i - p$  relativi alle permutazioni in colonna  $k$ , per l'estrazione della riga 9:

- $\pi_1(9) = 17 + 9 - 9 = 17$
- $\pi_2(17) = 4 + 17 - 13 = 8$
- $\pi_3(8) = 4 + 8 - 8 = 3$
- $\pi_4(3) = 0 + 3 - 0 = 3$
- $\pi_5(3) = 0 + 3 - 0 = 3$
- $\pi_6(3) = 16 + 3 - 3 = 16$
- $\pi_7(16) = 2 + 16 - 3 = 15$
- $\pi_8(15) = 2 + 15 - 3 = 14$
- $\pi_9(14) = 3 + 14 - 12 = 5$
- $\pi_{10}(5) = 1 + 5 - 5 = 1$
- $\pi_{11}(1) = 17 + 1 - 0 = 18$

Sfruttando quindi il valore booleano (non rappresentato nelle tabelle ma esistente) che ci dice con che simbolo inizia una colonna e sapendo che, essendo un pannello binario si alternano le run con simboli  $\sigma = 0$  e  $\sigma = 1$ , si può ricostruire la riga 9 del pannello originale:

$$x_9 = 100001000011$$

# Capitolo 3

## Metodo

In questo capitolo verranno illustrate le metodologie usate in questa tesi, trattando, sia dal punto di vista teorico che sperimentale, tutte le soluzioni che hanno portato alla costruzione di diverse varianti della **RLPBWT**.

Nel dettaglio, si approfondiranno tutte le varianti della **RLPBWT** ottenute durante lo studio, evidenziandone pro e contro.

### 3.1 Perché la compressione run-length

Prima di proseguire con la spiegazione dettagliata delle varianti della **RLPBWT** è bene dare una prima motivazione al perché si sia ritenuto utile sviluppare una variante **run-length encoded** della **PBWT**.

Citando direttamente il paper di Durbin [4]:

*Furthermore we can also expect the  $y$  arrays to be strongly run-length compressible. This is because population genetic structure means that there is local correlation in values due to linkage disequilibrium, which means that haplotypes with similar prefixes in the sort order will tend to have the same allele values at the next position, giving rise to long runs of identical values in the  $y$  array. So the PBWT can easily be stored in smaller space than the original data.*

Quindi il risultato atteso è quello per cui aplotipi simili, che, ad ogni step, saranno consecutivi nel riordinamento, è molto probabile presentino lo stesso allele nella colonna di cui si sta in quel momento calcolando la permutazione. Ne segue che, all'interno della *matrice PBWT*, è molto probabile che si abbiano lunghe run di simboli  $\sigma = 0$  e di simboli  $\sigma = 1$ .

Si noti quindi che si ottiene quindi il medesimo risultato atteso avuto con la *BWT*, avendo che caratteri uguali è molto probabile vengano posti in posizioni consecutive all'interno della *BWT* stessa. Si hanno quindi le stesse premesse che hanno portato

alla *RLBWT*, considerando, inoltre, che, come in quel caso, non si tratta solo di memorizzare la struttura con compressione run-length ma di lavorare direttamente con la struttura dati compressa, risolvendo il problema del calcolo degli *SMEM* senza decomprimere la struttura dati. Ipotizzando come, per una certa colonna della *matrice PBWT*, il numero di run sia molto minore della lunghezza della colonna stessa, si deduce facilmente che l'uso della compressione run-length possa comportare una riduzione significativa della memoria necessaria al calcolo degli *SMEM*.

## 3.2 Matching Statistics per la RLPBWT

Prima di discutere nel dettaglio delle strutture dati per la *RLPBWT*, è bene introdurre il concetto di **matching statistics** nel caso della *PBWT* (e quindi anche della sua variante *run-length encoded*).

**Definizione 27.** Dato un pannello  $X$ , di dimensioni  $M \times N$ , con  $M$  individui e  $N$  siti, e un aplotipo esterno/pattern  $z$ , tale che  $|z| = N$ , si definisce *matching statistics* di  $z$  su  $X$  un array  $MS$  di coppie  $(row, len)$ , di lunghezza  $N$ , tale che (avendo che  $x_i$  indica l' $i$ -esima riga del pannello  $X$ ):

- $x_{MS[i].row}[i - MS[i].len + 1, i] = z[i - MS[i].len + 1, i]$ , ovvero si ha che l'aplotipo query ha un match, terminante in colonna  $i$ , con la riga  $MS[i].row$
- $z[i - MS[i].len, i]$  non è un suffisso terminante in colonna  $i$  di un qualsiasi sottoinsieme di righe di  $X$ . In altri termini il match non deve essere ulteriormente estendibile a sinistra

Inoltre, analogamente al caso della variante classica, si ha il seguente lemma.

**Lemma 3.** Dato un pannello  $X$ , di dimensioni  $M \times N$ , con  $M$  individui e  $N$  siti, un aplotipo esterno/pattern  $z$ , tale che  $|z| = N$ , e il corrispondente array di *matching statistics*  $MS$  si ha che:

$$z[i - l + 1, i]$$

è uno **SMEM** di lunghezza  $l$  in con la riga  $MS[i].row$  del pannello  $X$  sse:

$$MS[i].len = l \wedge (i = N - 1 \vee MS[i].len \geq MS[i + 1].len)$$

Si vedrà in sezione 3.3.5 come calcolare, a partire da tali *SMEM*, tutte le righe del pannello per le quali si ha lo stesso *SMEM*.

Il calcolo dell'array  $MS$  di  $z$  rispetto al pannello  $X$  si basa su due fasi:



1. la fase di **start**
2. la fase di **extend**

Si assuma di avere due indici  $i$  e  $j$ ,  $0 \leq i \leq j < N$ , tali per cui  $z[i, j]$  è un suffisso di uno tra  $x_0[0, j]$ , ...,  $x_{M-1}[0, j]$ .

La *fase di extend* estende il match di  $z[i, j]$  a  $z[i, j + 1]$  sse:

- $j < M$
- $z[i, j + 1]$  è un suffisso di uno tra  $x_0[0, j + 1]$ , ...,  $x_{M-1}[0, j + 1]$

D'altro canto la *fase di start* cerca il più piccolo indice  $i'$ , avendo  $i \leq i' \leq j$ , tale per cui  $z[i', j]$  è un suffisso di uno tra  $x_0[0, j]$ , ...,  $x_{M-1}[0, j]$ .

Si ha quindi il computo di ogni valore  $MS[i]$ ,  $\forall i \in [0, N)$ , dell'array delle *matching statistics*:

- si assume inizialmente che  $MS[0].len = 0$ , quando  $i = 0$
- si applica la *fase di start* per cercare il minimo indice  $i'$ , avendo  $i \leq i'$ , tale che  $z[i', i' + MS[i].len]$  è un suffisso di uno tra  $x_0[0, i' + MS[i].len]$ , ...,  $x_{M-1}[0, i' + MS[i].len]$ . Inoltre, per minimalità di  $i'$ , si ha che,  $\forall i < j < i'$ ,  $MS[j].len = MS[j - 1].len + 1$
- a questo punto si itera la *fase di extend* per trovare il più lungo prefisso  $z[i', k]$  che è anche un suffisso di uno tra  $x_0[0, k]$ , ...,  $x_{M-1}[0, k]$ , avendo che  $MS[i'].len = k - i' + 1$
- avendo che  $i' > i$  si può procedere induttivamente al calcolo dell'array  $MS$

In altri termini, più “pratici”, il calcolo dell'array  $MS$  avviene nel seguente modo:

- si parte da una riga arbitraria  $i$  della prima colonna
- se  $x_i[0] = z[0]$  si procede salvando  $MS[0].row = i$
- qualora si abbia  $x_i[0] \neq z[0]$  si seleziona o l'ultima riga della run precedente o la prima riga della run successiva a quella a cui appartiene la riga  $i$ . Tale riga,  $j$ , verrà salvata in  $MS$ , avendo  $MS[0].row = j$
- a questo punto si effettua il mapping verso la colonna successiva,  $k$ , e, a seconda di avere o meno un match con  $z[k]$ , si procede come nei casi visti sopra

DC PARTE  
PRESA DAL  
PAPER: RIVE-  
DERE PRO-  
FONDAMEN-  
TE, FORSE  
MANCA ANA-  
LISI COM-  
PLETO MI-  
SMATCH

Si noti che non si è parlato di come calcolare i vari  $MS[i].len$ , questo in quanto si hanno due soluzioni, che rispecchiano quanto visto con *MONI* [7] e *PHONI* [8] per la *RLBWT*:

1. si possono usare le *threshold* per capire che nuova riga selezionare in caso di mismatch. In tal caso i vari  $MS[i].len$  devono essere calcolati dopo il calcolo di  $MS[i].row$  tramite *random access* al panel
2. si possono usare le *LCE query* per capire che nuova riga selezionare in caso di mismatch e in tal caso il calcolo delle  $MS[i].len$  avviene in contemporanea

Entrambi i concetti verranno definiti e caratterizzati nel corso del capitolo.

DC magari ref  
alle section

**Esempio 20.** Si riprenda, al fine di vedere un esempio di calcolo dell'array  $MS$ , l'esempio 18, con un pannello e i match con la query  $z$ :

img/pbwtmatch.pdf

In tal caso l'array  $MS$  sarebbe, avendo scelto come riga iniziale la 19:

$k$	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14
$z$	0	1	0	0	1	0	1	0	0	0	1	1	1	0	1
row	19	19	16	15	13	13	19	19	19	19	11	11	17	17	17
len	1	2	3	4	5	6	4	5	6	7	4	5	2	3	4

Dove si possono riconoscere i vari *SMEM*, la cui colonna di fine è segnalata in verde, secondo la definizione data sopra (anche in questo caso i dettagli del calcolo verranno esplicitati successivamente).

### 3.3 Componenti per la RLPBWT

Lo sviluppo di questo progetto di tesi è stato tale per cui si sono sviluppate varie implementazioni della **RLPBWT**. Infatti, avendo a che fare con strutture dati fortemente dipendenti dal tipo di dato (in termini, ad esempio, di “sparsità” intrinseca del pannello) e dall’implementazione (soprattutto in termini di strutture dati succinte), sarebbe stato difficile limitarsi a stime teoriche che potevano non essere confermate in fase sperimentale. Una sola caratterizzazione asintotica avrebbe potuto comportare sottostime e sovrastime e si è quindi deciso di sviluppare diverse varianti della *RLPBWT*.

Al fine di una miglior trattazione di tali implementazioni, si è deciso di suddividere le stesse in *componenti*, le quali, adeguatamente “assemblate” permetteranno, in varie accezioni, il calcolo degli *SMEM*. Tali componenti, che verranno dettagliate in seguito, sono:

- le componenti per il mapping tra la colonna  $k$ -esima e la colonna  $k + 1$ , ovvero, riprendendo la notazione di Durbin, le strutture run-length encoded per gli array  $c$ ,  $u_k$  e  $v_k$ . Nel dettaglio, tale componente è implementata in due varianti:
  1. *mapping tramite intvector* (MAP-INT)
  2. *mapping tramite bitvector sparsi* (MAP-BV)
- la componente per la memorizzazione, anch’essa proporzionale al numero di run, delle threshold. Anche in questo caso si hanno due varianti, corrispondenti di fatto alle due varianti della componente del mapping:
  1. *threshold con intvector* (THR-INT)
  2. *threshold con bitvector sparsi* (THR-BV)
- la componente per la memorizzazione della permutazione ad ogni colonna della *matrice PBWT*, ovvero dei *samples di prefix array* (PERM)

- la componente in grado di garantire *random access* al pannello. Si hanno due possibilità:
  1. *random access con bitvector* (RA-BV)
  2. *random access con SLP* (RA-SLP)
- la componente per le *longest common extension query* (LCE)
- la componente per l'intero *longest common prefix array* (LCP), già descritto nella sezione 2.5
- la componente per permettere il calcolo delle *funzioni*  $\varphi$  e  $\varphi^{-1}$  (PHI)

### 3.3.1 Componente per il mapping

La prima componente, nelle sue due varianti, che si descrive è quella relativa al mapping tra una colonna e la sua successiva. Bisogna quindi memorizzare per ogni colonna  $k$ , in modo proporzionale al numero di run della stessa, le informazioni atte a ottenere i medesimi risultati ottenibili con la funzione  $w(i, \sigma)$ , secondo la notazione di Durbin [4].

#### Mapping con intvector

La prima variante che si descrive è quella denominata MAP-INT.

L'ispirazione iniziale per tale componente è stata data dall'articolo di Gagie et al [35], nonostante si abbiano, di fatto, diverse modifiche strutturali. Riprendendo quanto descritto al termine della sezione 2.5, si è quindi deciso di memorizzare gli indici delle *teste di run*, ovvero gli indici iniziali di ogni run. Ovviamente questa informazione non è sufficiente per poter sapere se una run sia composta da simboli  $\sigma = 0$  o simboli  $\sigma = 1$ . Fortunatamente, essendo lo studio limitato, come per la *PBWT*, a pannelli costruiti su alfabeto binario  $\Sigma = \{0, 1\}$ , si è potuto sfruttare il fatto che le run si alternano tra un carattere e l'altro. Basta quindi tenere in memoria un valore booleano nominato  $start_k$ , che permetta di capire se, in colonna  $k$ , la prima run sia una run di simboli  $\sigma = 0$ . Infatti le run di indice pari presentano lo stesso simbolo della prima run e quindi, dato un qualsiasi indice di run, è possibile sapere quale sia il simbolo corrispondente a tale run. L'implementazione di questo concetto è visualizzabile all'algoritmo 3.1 e richiede tempo costante.

Si memorizzano gli indici delle teste di run in un array  $p_k$ , di lunghezza pari al numero di run in colonna  $k$ . In pratica si memorizza un indice  $i$  sse:

$$y_{i-1}^k[k] \neq y_i^k[k] \quad (3.1)$$

**Algoritmo 3.1** Algoritmo per estrazione simbolo da una run in una colonna

---

```

1: function GET_SYMBOL( $s, r$ )  ▷  $s = \top$  sse la prima run ha simbolo  $\sigma = 0$ ,  $r$  indice di run
2:   if  $s$  then
3:     if  $r \bmod 2 = 0$  then return 0 else return 1
4:   else
5:     if  $r \bmod 2 = 0$  then return 1 else return 0

```

---

Il passaggio successivo è stato quello di capire se le informazioni necessarie al mapping fossero tutte necessarie. In altri termini se, data la colonna  $k$  nella matrice  $PBWT$ , fossero necessari  $c[k]$ ,  $u_k$  e  $v_k$ . In merito al valore  $c[k]$ , per quanto calcolabile, ipotizzando di avere solo  $p_k$ , in tempo  $\mathcal{O}(r)$ , dove  $r$  è il numero di run della colonna  $k$ -esima, si è deciso che si potesse calcolarlo in fase di costruzione delle  $RLPBWT$  e memorizzarlo esattamente come per la  $PBWT$ . In merito invece ai vettori  $u_k$  e  $v_k$  si è cercato un modo per ottenerne una rappresentazione che implicasse avere un solo valore per ogni run della colonna. In altri termini si è cercato di capire se fosse possibile tenere in memoria  $r$  valori che permettessero di effettuare comunque il mapping, a partire da un indice arbitrario  $i \in \{0, \dots, M - 1\}$ . Anche in questo caso l'alternanza data dal caso binario ha permesso di trovare una semplice soluzione. I valori di  $u_k$  e  $v_k$  crescono infatti in modo alternato. Infatti, a seconda del simbolo  $\sigma$  rappresentato in una data run, si ha che solo i valori dell'array relativo a tale simbolo, nel range di indici di quella run, verranno incrementati, ad ogni passo, di una unità. Facendo un semplice esempio, se siamo in una run di 0 e iteriamo virtualmente all'interno di tale run, solo i valori di  $u_k$ , in quel range di indici, cresceranno di volta in volta di uno mentre per  $v_k$ , nello stesso range, si avrà sempre lo stesso valore.

**Esempio 21.** Si vede un esempio per chiarire meglio quanto espresso in merito a  $u_k$  e  $v_k$ .

Sia data la seguente colonna:

$$y^5 = 00101111100000000000$$

Si hanno, oltre a  $c[5] = 15$ :

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
$y^5$	0	0	1	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
$u_5$	0	1	2	2	3	3	3	3	3	4	5	6	7	8	9	10	11	12	13	14
$v_5$	0	0	0	1	1	2	3	4	5	5	5	5	5	5	5	5	5	5	5	5

Dove si nota l'alternanza di crescita dei valori sopra descritta.

Grazie a questo comportamento è possibile memorizzare, per ogni indice di testa di run  $i$ , tale che  $i \neq 0$ , solo il valore di  $u_k[i]$  o  $v_k[i]$ , rispettivamente se sia

una run su simboli  $\sigma = 1$  o  $\sigma = 0$ . Questo in quanto, se si analizza una run di zeri si avrà che solo i valori di  $v_k$ , nel range della run, verranno incrementati ad ogni step. Per  $i = 0$  banalmente si ha che  $u_k[i] = v_k[i] = 0$ .

Memorizzando i valori di  $u_k$  e  $v_k$  in un array  $uv_k$ , tale che  $|uv_k| = r$ , con  $r$  numero di run, e dato  $i \in \{0, \dots, r-1\}$ , a seconda che la colonna presenti o meno la prima run con simboli  $\sigma = 0$ , si possono estrarre, in tempo costante, i valori di  $u_k$  e  $v_k$  per una data testa di run. Nel dettaglio, dato  $i \in 0, \dots, r-1$ :

- se  $i = 0$  si ha che  $u_k[p[i]] = v_k[p[i]] = uv_k[0] = 0$
- se  $i \bmod 2 = 0$  si hanno due casi:
  - la prima run è di simboli  $\sigma = 0$  e quindi si ottiene  $u_k[p[i]] = uv_k[i-1]$  e  $v_k[p[i]] = uv_k[i]$
  - la prima run è di simboli  $\sigma = 1$  e quindi si ottiene  $u_k[p[i]] = uv_k[i]$  e  $v_k[p[i]] = uv_k[i-1]$
- se  $i \bmod 2 \neq 0$  si hanno due casi, che sono l'inverso della situazione descritta precedentemente:
  - la prima run è di simboli  $\sigma = 0$  e quindi si ottiene  $u_k[p[i]] = uv_k[i]$  e  $v_k[p[i]] = uv_k[i-1]$
  - la prima run è di simboli  $\sigma = 1$  e quindi si ottiene  $u_k[p[i]] = uv_k[i-1]$  e  $v_k[p[i]] = uv_k[i]$

Tale operazione è eseguibile in tempo costante lo pseudocodice relativo a quanto appena detto è consultabile all'algoritmo 3.2.

Da un punto di vista implementativo, gli array di interi  $p_k$  e  $uv_k$  vengono memorizzati in *vettori di interi bit-compressed*. Dato un array  $v$ , tale che  $x$  è il massimo indice, si calcola:

$$b = \lceil \log(x-1) \rceil + 1 \quad (3.2)$$

e si memorizzano le componenti di  $v$  in un vettore di interi con componenti memorizzate in di  $b$  bit.

Ricapitolando, per la componente MAP-INT, si hanno in memoria, per ogni colonna  $k$ :

- $start_k$ , con il booleano atto a capire il simbolo della prima run
- $p_k$ , con gli indici delle teste di run
- $uv_k$ , coi valori compatti di  $u_k$  e  $v_k$  per le teste di run

**Algoritmo 3.2** Algoritmo per uvtrick con MAP-INT.

---

```

1: function UVTRICK( $k, i$ ) ▷  $k$  indice di colonna,  $i$  indice di run
2:   if  $i = 0$  then
3:     return  $(0, 0)$ 
4:   else if  $i \bmod 2 = 0$  then
5:      $u \leftarrow uv_k[i - 1], v \leftarrow uv_k[i]$ 
6:     if  $start_k$  then
7:       return  $(u, v)$ 
8:     else
9:       return  $(v, u)$ 
10:  else
11:     $u \leftarrow uv_k[i], v \leftarrow uv_k[i - 1]$ 
12:    if  $start_k$  then
13:      return  $(u, v)$ 
14:    else
15:      return  $(v, u)$ 

```

---

- $c[k]$ , per sapere il numero totale di simboli  $\sigma = 0$  nella colonna  $k$  della matrice PBWT

**Esempio 22.** Sia data la seguente colonna:

$$y^5 = 00101111000000000000$$

Per la componente MAP-INT della colonna 5, si hanno in memoria:

$$p_5 = [0, 2, 3, 4, 8]$$

$$uv_5 = [0, 2, 1, 3, 5]$$

$$c[5] = 15$$

La costruzione della componente MAP-INT per una certa colonna, analizzabile nell'algoritmo 3.3, ha costo  $\mathcal{O}(M)$ , avendo che la costruzione avviene scorrendo la colonna  $k$  permutata dal *prefix array*  $a_k$ .

Si hanno quindi le informazioni relative alle teste di run. Per l'implementazione dei vari algoritmi si ha però necessità di usare anche indici con valori  $\{0, M - 1\}$ . Una delle operazioni fondamentali è quindi quella, dato un indice  $i \in \{0, \dots, M - 1\}$ , di computare a quale run esso appartenga, in una certa colonna  $k$ . Tale operazione può essere svolta usando una semplice variante della *ricerca binaria*. Questa variante dell'algoritmo tradizionale, anziché ritornare l'indice di un elemento qualora esista nell'array, restituisce l'ultimo indice iniziale del sottointervallo usato dalla ricerca binaria calcolato prima dell'interruzione dell'esecuzione dell'algoritmo (che

---

**Algoritmo 3.3** Algoritmo per la costruzione della componente *MAP-INT* per la colonna  $k$ .

---

```

1: function BUILD_MAP_INT( $col, pref$ )  $\triangleright pref = a_k$ 
2:    $c \leftarrow 0, u \leftarrow 0, v \leftarrow 0, u' \leftarrow 0, v' \leftarrow 0, run \leftarrow 0$ 
3:    $start \leftarrow \top, begun \leftarrow \top, push_{zero} \leftarrow \perp, push_{one} \leftarrow \perp$ 
4:    $p \leftarrow [], uv \leftarrow []$ 
5:   for every  $k \in [0, M)$  do
6:     if  $k = 0 \wedge col[pref[k]] = 1$  then
7:        $start \leftarrow \perp$ 
8:     if  $col[k] = 0$  then
9:        $c \leftarrow c + 1$ 
10:    if  $start$  then
11:       $push_{one} \leftarrow \top$ 
12:    else
13:       $push_{zero} \leftarrow \top$ 
14:    for every  $k \in [0, M)$  do
15:      if  $begun$  then
16:         $u \leftarrow u', v \leftarrow v'$ 
17:         $begun \leftarrow \perp$ 
18:      if  $col[pref[k]] = 1$  then
19:         $v' \leftarrow v' + 1$ 
20:      else
21:         $u' \leftarrow u' + 1$ 
22:      if  $k = 0 \vee col[pref[k]] \neq col[pref[k - 1]]$  then
23:         $run \leftarrow k$ 
24:      if  $k = M - 1 \vee col[pref[k]] \neq col[pref[k + 1]]$  then
25:        if  $push_{one}$  then
26:           $push(p, run), push(uv, v)$ 
27:           $swap(push_{one}, push_{zero})$ 
28:        else
29:           $push(p, run), push(uv, u)$ 
30:           $swap(push_{one}, push_{zero})$ 
31:         $begun \leftarrow \top$ 
32:    return ( $start, c, p, uv$ )

```

---



avviene secondo l'algoritmo standard). Pur essendo, di fatto, una funzione *rank* si è deciso, per comodità di lettura, di chiamare tale funzione *index\_to\_run*, la quale, come visualizzabile all'algoritmo 3.4, ha complessità in tempo, con  $r = |p_k|$  (ovvero è il numero di run in colonna  $k$ ):

$$\mathcal{O}(\log(r)) \quad (3.3)$$

---

**Algoritmo 3.4** Algoritmo per convertire un indice di colonna in indice di run, con MAP-INT.

---

```

1: function INDEX_TO_RUN( $k, i$ )      ▷  $k$  indice di colonna,  $i$  indice di riga della colonna  $k$ 
2:   if  $i \geq p_k[|p_k| - 1]$  then
3:     return  $|p_k| - 1$ 
4:    $b \leftarrow 0, e \leftarrow |p_k|$ 
5:    $run \leftarrow \frac{e-b}{2}$ 
6:   while  $run \neq e \wedge p_k[run] \neq i$  do
7:     if  $i < p_k[run]$  then
8:        $e \leftarrow run$ 
9:     else
10:      if  $run + 1 = e \vee p_k[run + 1] > i$  then
11:        break
12:       $b \leftarrow run + 1$ 
13:       $run \leftarrow b + \frac{e-b}{2}$ 
14:   return  $run$ 

```

---

Ipotizzando, inoltre, di avere un indice  $i \in \{0, \dots, M-1\}$  è possibile risalire ai valori  $u_k[i]$  e  $v_k[i]$ , sfruttando l'*offset* dell'indice rispetto alla testa della run a cui appartiene. Banalmente, ipotizzando di essere in una run di simboli  $\sigma$  con testa di run all'indice  $p$ , si avranno, avendo ottenuto  $u_k[p]$  e  $v_k[p]$  da  $uv_k[p]$ :

$$\begin{cases} v_k[i] = v_k[p] \\ u_k[i] = u_k[p] + (i - p) \end{cases}, \text{sse } y_p^k[k] = 0 \quad \begin{cases} u_k[i] = u_k[p] \\ v_k[i] = v_k[p] + (i - p) \end{cases}, \text{sse } y_p^k[k] = 1 \quad (3.4)$$

Tenendo eventualmente conto dell'*offset*  $off$ , qualora si abbia un simbolo  $\sigma$  uguale a quello della run in analisi, è quindi possibile riadattare l'algoritmo per il mapping visto per la *PBWT* di Durbin, da intendersi alla stregua del *backward step* visto nel caso della *BWT*, dalla colonna  $k$  alla colonna  $k+1$  guidato da  $z[k+1]$ . Tale soluzione è riportata all'algoritmo 3.5. Ricordando che si può risalire ai valori  $u[p]$  e  $v[p]$  in tempo costante, anche il mapping da una colonna alla successiva avviene in tempo costante.

**Algoritmo 3.5** Algoritmo per il mapping con MAP-INT.

---

```

1: function  $w(k, i, \sigma, o)$   $\triangleright k$  indice di colonna,  $i$  indice di riga,  $\sigma$  simbolo
2:    $run \leftarrow index\_to\_run(k, i)$ 
3:   if  $\sigma = 0 \wedge get\_symbol(start_k, run) = 1$  then
4:      $off \leftarrow 0$ 
5:   else if  $\sigma = 1 \wedge get\_symbol(start_k, run) = 0$  then
6:      $off \leftarrow 0$ 
7:   else
8:      $off \leftarrow i - p_k[run]$ 
9:    $(u, v) \leftarrow uvtrick(k, i)$ 
10:  if  $p_k[i] + off = M$  then
11:    if  $get\_symbol(start_k, i) = 0$  then
12:       $v \leftarrow v - 1$ 
13:    else
14:       $u \leftarrow u - 1$ 
15:  if  $\sigma = 0$  then
16:    return  $u + off$ 
17:  else
18:    return  $c[k] + v + off$ 

```

---

**Mapping con bitvector**

La seconda variante della componente di mapping sfrutta, al posto degli *intvector*, i *bitvector sparsi*, da cui la nomenclatura MAP-BV.

L'idea è quindi quella di sostituire, data una colonna  $k$ , quanto necessario a rappresentare le run (ovvero il vettore  $p_k$  della MAP-INT) e quanto necessario a permettere il mapping (ovvero il vettore  $uv_k$  della MAP-INT).

In primis, per poter localizzare le run nella  $k$ -esima colonna, si è scelto di usare un *bitvector sparso*, che denominiamo per praticità  $h_k$ , tale che  $|h_k| = M$ . Formalmente si ha che:

$$h_k[i] = \begin{cases} 1 & \text{se } y_i^k[k] \neq y_{i+1}^k[k] \vee i = M - 1 \\ 0 & \text{altrimenti} \end{cases}, \forall i \in \{0, \dots, M - 1\} \quad (3.5)$$

Informalmente, quindi, si ha che si ha  $h_k[j] = 1$  sse è l'indice di fine di una run. empiricamente ci si aspettano “poche” run all'interno di una colonna della *matrice PBWT*, per quanto già discusso nella sezione 2.5. Avendo poche run ci si aspetta anche “pochi” 1 all'interno di  $h_k$ , di conseguenza si è optato per usare i *bitvector sparsi* per la memorizzazione in memoria di ogni  $h_k$ , ricordando che, secondo quanto riportato per la libreria *SDSL* [12], tale variante richiede in memoria, indicando

con  $r$  il numero di run:

$$\approx r \left( 2 + \log \frac{M}{r} \right) \text{ bit} \quad (3.6)$$

Pensando ad una correlazione tra MAP-INT e MAP-BV, si ha che  $rank_{h_k}$  fa le veci della funzione `index_to_run` mentre  $select_{h_k}$  equivale a  $p_k$ .

Più elaborata è la rappresentazione dei vettori  $u_k$  e  $v_k$ . In questo caso si è deciso, a differenza della rappresentazione unica vista con la MAP-INT, di optare per due *bitvector sparsi*. In particolare, per il vettore  $u_k$ , tale che  $|u_k| = c[k]$ , si ha che,  $\forall i \in \{0, \dots, |u_k| - 1\}$ :

$$u_k[i] = \begin{cases} 1 & \text{se } i \text{ è il numero di simboli che contiene la } rank_{u_k}(i)\text{-esima run di } 0 \\ 0 & \text{altrimenti} \end{cases} \quad (3.7)$$

Analogamente si definisce  $v_k$ , avendo  $|v_k| = M - c[k]$  e  $\forall i \in \{0, \dots, |v_k| - 1\}$ , come:

$$v_k[i] = \begin{cases} 1 & \text{se } i \text{ è il numero di simboli che contiene la } rank_{v_k}(i)\text{-esima run di } 1 \\ 0 & \text{altrimenti} \end{cases} \quad (3.8)$$

Si noti che:

$$rank_{h_k}(|h_k| - 1) + 1 = (rank_{u_k}(|u_k| - 1) + 1) + (rank_{v_k}(|v_k| - 1) + 1) \quad (3.9)$$

Ovvero il numero di 1 presenti in  $h_k$  è pari alla somma di quelli presenti in  $u_k$  e  $v_k$ . Si noti che i vari +1 sono dovuti al fatto che la funzione  $rank(i)$  esclude dal computo la posizione  $i$  stessa e tutti e tre i bitvector, per costruzione, presentano  $\sigma = 1$  in ultima posizione. Ne segue che, anche per questi ultimi due bitvector, la scelta di usare *bitvector sparsi* per la loro memorizzazione sia giustificata, empiricamente, dalla poca quantità attesa di simboli  $\sigma = 1$ .

**Esempio 23.** *Sia data la seguente colonna:*

$$y^5 = 00101111100000000000$$

*Si ha quindi che:*

$$h_5 = 01110001000000000001$$

*Avendo appunto un numero di run pari a:*

$$rank_{h_5}(|h_5| - 1) + 1 = 4 + 1 = 5$$

*In merito alle run composte da simboli  $\sigma = 0$  si ha che:*

$$u_5 = 0110000000000001$$

*Avendo infatti che si segnalano:*

- la prima run composta da due simboli  $\sigma = 0$
- la seconda run composta da un solo simbolo  $\sigma = 0$
- la terza run composta da dodici simboli  $\sigma = 0$

Parlando invece di  $v_5$  si ha:

$$v_5 = 10001$$

Avendo che:

- la prima run è composta da un solo simbolo  $\sigma = 1$
- la seconda run è composta da quattro  $\sigma = 1$

Si conferma, inoltre, quanto detto nell'equazione 3.9, avendo:

$$\text{rank}_{h_5}(|h_5| - 1) + 1 = 5 = (\text{rank}_{u_5}(13) + 1) + (\text{rank}_{v_5}(4) + 1) = (2 + 1) + (1 + 1) = 5$$

Lo pseudocodice relativo alla costruzione della componente **MAP-BV** per la colonna  $k$ -esima è disponibile all'algoritmo 3.6. Anche in questo caso la costruzione avviene scorrendo la colonna  $k$  permutata dal *prefix array*  $a_k$ .

Assumendo che la complessità in tempo delle costruzioni delle strutture a supporto per le funzioni *rank* e *select* dei tre bitvector sparsi sia limitata superiormente dalla loro lunghezza massima, ovvero  $M$ , si ha che la costruzione della componente **MAP-BV** per una singola colonna avviene in tempo:

$$\mathcal{O}(M) \tag{3.10}$$

Bisogna spiegare come, dato un indice di aplotipo  $i \in \{0, \dots, M-1\}$  e una colonna  $k$ , estrarre  $u'_k[i]$  e  $v'_k[i]$ , ovvero come se si stesse usando la *PBWT* classica, a partire dagli attuali  $u_k[i]$  e  $v_k[i]$ . Ovviamente, se  $i = 0$ , si ha che  $u'_k[0] = v'_k[0] = 0$ . In caso contrario bisogna, in primis, calcolare la run in cui si trova l'indice  $i$ . Questo si ottiene direttamente sfruttando  $h_k$ :

$$\text{run} = \text{rank}_{h_k}(i) \tag{3.11}$$

Una volta calcolato l'indice di run si hanno tre possibilità:

1. si ha che  $\text{run} = 0$  e una run di simboli  $\sigma = b$ , con  $b \in \{0, 1\}$  allora:

$$(u, v) = \begin{cases} (i, 0) & \text{se } b = 0 \\ (0, i) & \text{altrimenti} \end{cases} \tag{3.12}$$

---

**Algoritmo 3.6** Algoritmo per la costruzione della componente MAP-BV per la colonna  $k$ .

---

```

1: function BUILD_MAP_BV( $col, pref$ )  $\triangleright pref = a_k$ 
2:    $c \leftarrow 0, u \leftarrow 0, v \leftarrow 0, u' \leftarrow 0, v' \leftarrow 0, curr_{lcs} \leftarrow 0$ 
3:    $start \leftarrow \top, beg_{run} \leftarrow \top, push_{zero} \leftarrow \perp, push_{one} \leftarrow \perp$ 
4:   for every  $k \in [0, M)$  do
5:     if  $k = 0 \wedge col[pref[k]] = 1$  then
6:        $start \leftarrow \perp$ 
7:       if  $col[k] = 0$  then
8:          $c \leftarrow c + 1$ 
9:        $runs \leftarrow [0..0]$   $\triangleright$  bitvector sparso per le run, di lunghezza  $M + 1$ 
10:       $zeros \leftarrow [0..0]$   $\triangleright$  bitvector sparso per  $u_k$ , di lunghezza  $c[k]$ 
11:       $ones \leftarrow [0..0]$   $\triangleright$  bitvector sparso per  $v_k$ , di lunghezza  $M - c$ 
12:      if  $start$  then
13:         $push_{one} \leftarrow \top$ 
14:      else
15:         $push_{zero} \leftarrow \top$ 
16:      for every  $k \in [0, M)$  do
17:        if  $beg_{run}$  then
18:           $u \leftarrow u', v \leftarrow v', beg_{run} \leftarrow \perp$ 
19:        if  $col[pref[k]] = 1$  then
20:           $v' \leftarrow v' + 1$ 
21:        else
22:           $u' \leftarrow u' + 1$ 
23:        if  $k = M - 1 \vee col[pref[k]] \neq col[pref[k + 1]]$  then
24:           $runs[k] \leftarrow 1$ 
25:          if  $push_{one}$  then
26:            if  $v \neq 0$  then
27:               $ones[k - 1] = 1$ 
28:             $swap(push_{zero}, push_{one})$ 
29:          else
30:            if  $u \neq 0$  then
31:               $zeros[k - 1] = 1$ 
32:             $swap(push_{zero}, push_{one})$ 
33:           $beg_{run} \leftarrow \top$ 
34:        if  $|zeros| \neq 0$  then
35:           $zeros[|zeros| - 1] \leftarrow 1$ 
36:        if  $|ones| \neq 0$  then
37:           $ones[|ones| - 1] \leftarrow 1$ 
38:      costruzione delle strutture per rank/select dei tre bitvector
39:      return ( $start, c, runs, zeros, ones$ )

```

---

2. si ha che  $run = 1$  e una run di simboli  $\sigma = b$ , con  $b \in \{0, 1\}$ . In tal caso bisogna per prima cosa individuare l'indice di inizio della seconda run, sfruttando  $h_k$ :

$$beg = select_{h_k}(1) + 1 \quad (3.13)$$

A questo punto si ha il numero di simboli della prima run, indicizzata a 0, e, calcolando la distanza tra l'indice di riga e quello di inizio della prima run, avendo che:

$$(u, v) = \begin{cases} (beg, i - beg) & \text{se } b = 0 \\ (i - beg, beg) & \text{altrimenti} \end{cases} \quad (3.14)$$

3. si ha che  $run = j$ , con  $j \in \{2, r - 1\}$ . Anche in questo caso si procede calcolando l'indice di inizio della run:

$$beg = select_{h_k}(run) + 1 \quad (3.15)$$

e l'offset rispetto all'indice  $i$  dato:

$$offset = i - beg \quad (3.16)$$

Poi, sfruttando la solita dicotomia fornita dal caso binario in studio, si hanno due casi:

- (a) si è in una run di indice pari. Si sfruttano poi  $u_k$  e  $v_k$  per sapere l'indice della precedente run con simboli  $\sigma = 0$ :

$$pre_u = select_{u_k} \left( \left\lfloor \frac{run}{2} \right\rfloor \right) + 1 \quad (3.17)$$

e quello della run con simboli  $\sigma = 1$ :

$$pre_v = select_{v_k} \left( \left\lfloor \frac{run}{2} \right\rfloor \right) + 1 \quad (3.18)$$

Si noti che si usa  $\frac{run}{2}$  in quanto, essendo in una run di indice pari si hanno precedentemente lo stesso numero di run per  $\sigma = 0$  e per  $\sigma = 1$  e quindi si considera lo stesso numero di “run” nei due *bitvector sparsi*  $u_k$  e  $v_k$ .

A questo punto, sempre per il ragionamento per cui solo uno tra  $u$  e  $v$  non è costante all'interno di una run si ha che o  $pre_u$  o  $pre_v$  è tale costante mentre l'altro valore deve essere calcolato considerando l'offset:

$$(u, v) = \begin{cases} (pre_u + offset, pre_v) & \text{se } b = 0 \\ (pre_u, pre_v + offset) & \text{altrimenti} \end{cases} \quad (3.19)$$

- (b) ci si trova in una run di indice dispari, quindi non si hanno precedentemente lo stesso numero di run per i due simboli. Bisogna quindi calcolare quante siano tali run. Se la prima run è di zeri:

$$run_u = select_{u_k} \left( \left\lfloor \frac{run}{2} \right\rfloor \right) + 1 \quad (3.20)$$

$$run_v = select_{v_k} \left( \left\lfloor \frac{run}{2} \right\rfloor \right) \quad (3.21)$$

mentre se la prima run non è di zeri si devono invertire i due valori. Si sa quindi quali “run” considerare sui due *bitvector sparsi*  $u_k$  e  $v_k$ . Posso quindi procedere come nel caso precedente, avendo:

$$pre_u = select_{u_k}(run_u) + 1 \quad (3.22)$$

$$pre_v = select_{v_k}(run_v) + 1 \quad (3.23)$$

E potendo quindi restituire:

$$(u, v) = \begin{cases} (pre_u, pre_v + offset) & \text{se } b = 0 \\ (pre_u + offset, pre_v) & \text{altrimenti} \end{cases} \quad (3.24)$$

DC Sistemare

**Esempio 24.** Si prendano i dati e i risultati ottenuti all'esempio 23. Si vogliono calcolare  $u$  e  $v$  per  $i = 6$ .

In primis si ha quindi:

$$run = rank_{h_5}(6) = 3$$

:

$$beg = select_{h_5}(3) + 1 = 3 + 1 = 4$$

$$offset = i - beg = 6 - 4 = 2$$

Quindi ci si trova nel terzo caso e, nel dettaglio, avendo una run di indice dispari. Si calcolano quindi:

$$run_u = select_{u_5} \left( \left\lfloor \frac{3}{2} \right\rfloor \right) + 1 = select_{u_5}(1) + 1 = 1 + 1 = 2$$

$$run_v = select_{v_5} \left( \left\lfloor \frac{3}{2} \right\rfloor \right) = select_{v_5}(1) = 0$$

che non andranno invertiti avendo  $start^5 = \top$ .

Si calcolano quindi:

$$pre_u = select_{u_5}(2) + 1 = 2 + 1 = 3$$

$$pre_v = select_{v_5}(0) + 1 = 0 + 1 = 1$$

Avendo infatti, in totale, tre simboli  $\sigma = 0$  e un simbolo  $\sigma = 1$  prima dell'indice 6. Concludendo, avendo  $start^5 = \top$ :

$$(u, v) = (pre_u, pre_v + offset) = (3, 1 + 2) = (3, 3)$$

Lo pseudocodice per il calcolo di  $u_k[i]$  e  $v_k[i]$  è disponibile all'algoritmo 3.7. In merito alla complessità in tempo del calcolo di  $u_k[i]$  e  $v_k[i]$ , si ha che essa è limitata superiormente dal costo della *funzione rank* su *bitvector sparsi*, essendo la *funzione select* disponibile in tempo costante. Ne segue che, avendo  $r$  run nella colonna  $k$ , si ha un tempo proporzionale a:

$$\mathcal{O}\left(\log \frac{M}{r}\right) \quad (3.25)$$

Non dovendo in tal caso considerare esplicitamente l'offset, come nel caso della MAP-INT, il mapping dalla colonna  $k$  alla colonna  $k + 1$ , guidato da  $z[k + 1]$ , viene fatto come nel caso della PBWT, come visualizzabile all'algoritmo 3.8, che presenta quindi la medesima complessità del calcolo di  $u[i]$  e  $v[i]$ , ovvero quello visto all'equazione 3.25.

DC Serve altro?

### 3.3.2 Componente per le threshold

Come discusso per MONI, l'uso delle **threshold** è parte fondamentale di uno dei due modi per ottenere le *matching statistics*.

**Definizione 28.** Data la colonna  $k$ -esima della **matrice PBWT**,  $y^k$ , memorizzata tramite compressione **run-length** e data la run  $j$ -esima, indicizzata da  $i$  a  $i'$ , si definisce **threshold** come l'indice del minimo valore LCP, che ricordiamo essere calcolato sull'ordinamento inverso, compreso negli indici della run, compreso l'eventuale  $LCP_k[i' + 1]$ , qualora  $i' \neq M - 1$ . Si noti che quest'ultimo valore, se esistente, deve essere considerato in quanto per il suo calcolo, come specificato nei preliminari alla sezione 2.5, si prende in considerazione  $y_{i'}^k$  e  $y_{i'+1}^k$ .

Da un punto di vista implementativo, come anticipato, si hanno due soluzioni, una basata su *intvector* e una basata su *bitvector sparsi*. In entrambi i casi il calcolo si può effettuare in parallelo a quello di MAP-INT e MAP-BV.

#### Threshold con intvector

In questo caso la memorizzazione delle threshold avviene in modo molto semplice, usando un *vettore di interi bit-compressed*. Data una colonna  $k$  della PBWT



**Algoritmo 3.7** Algoritmo per uvtrick con MAP-BV.

---

```

1: function UVTRICK( $k, i$ )  $\triangleright k$  indice di colonna,  $i$  indice di riga
2:   if  $i = 0$  then
3:     return (0, 0)
4:    $run \leftarrow rank_h^k(i)$ 
5:   if  $run = 0$  then
6:     if  $start_k$  then
7:       return ( $i, 0$ )
8:     else
9:       return (0,  $i$ )
10:  else if  $run = 1$  then
11:    if  $start_k$  then
12:      return ( $select_h^k(run) + 1, i - (select_h^k(run) + 1)$ )
13:    else
14:      return ( $i - (select_h^k(run) + 1), select_h^k(run) + 1$ )
15:  else
16:    if  $run \bmod 2 = 0$  then
17:       $pre_u \leftarrow select_u^k(\lfloor \frac{run}{2} \rfloor) + 1$ 
18:       $pre_v \leftarrow select_v^k(\lfloor \frac{run}{2} \rfloor) + 1$ 
19:       $offset \leftarrow i - (select_h^k(run) + 1)$ 
20:      if  $start_k$  then
21:        return ( $pre_u + offset, pre_v$ )
22:      else
23:        return ( $pre_u, pre_v + offset$ )
24:    else
25:       $run_u \leftarrow (\lfloor \frac{run}{2} \rfloor) + 1$ 
26:       $run_v \leftarrow \lfloor \frac{run}{2} \rfloor$ 
27:      if  $\neg start_k$  then
28:         $swap(run_u, run_v)$ 
29:       $pre_u \leftarrow select_u^k(run_u) + 1$ 
30:       $pre_v \leftarrow select_v^k(run_v) + 1$ 
31:       $offset \leftarrow i - (select_h^k(run) + 1)$ 
32:      if  $start_k$  then
33:        return ( $pre_u, pre_v + offset$ )
34:      else
35:        return ( $pre_u + offset, pre_v$ )

```

---

---

**Algoritmo 3.8** Algoritmo per il mapping con MAP-BV.

---

```

1: function  $w(k, i, \sigma)$   $\triangleright k$  indice di colonna,  $i$  indice di riga,  $\sigma$  simbolo
2:    $c \leftarrow c[k]$ 
3:    $(u, v) \leftarrow uvtrick(k, i)$ 
4:   if  $\sigma = 0$  then
5:     return  $u$ 
6:   else
7:     return  $c + v$ 

```

---

*matrix*, con  $r$  numero di run, si calcola  $t_k$  tale che  $t_k[i] = j$  sse  $j$  è l'indice della threshold dell' $i$ -esima run.

Lo pseudocodice per la costruzione della componente THR-INT della colonna  $k$  è consultabile all'algoritmo 3.9 e, dovendo scorrere la colonna permutata dal prefix array  $a_k$  e dovendo accedere ai valori di  $l_k$ , tale operazione ha complessità in tempo proporzionale a:

$$\mathcal{O}(M) \tag{3.26}$$

Si noti che, qualora minimo LCP si trovi nella testa della run successiva (da considerare in quanto calcolato anche grazie all'ultimo elemento della run corrente), si può tranquillamente memorizzare l'indice della testa della run successiva come *threshold*.

---

**Algoritmo 3.9** Algoritmo per la costruzione della componente THR-INT.

---

```

function BUILD_THR_INT( $col, pref, div$ )  $\triangleright pref = a_k, div = l_k$ 
   $curr_{lcs} \leftarrow 0, tmp_{thr} \leftarrow 0$ 
   $t \leftarrow []$ 
  for every  $k \in [0, M)$  do
    if  $k = 0 \vee col[pref[k]] \neq col[pref[k-1]]$  then
       $curr_{lcs} \leftarrow div[k], tmp_{thr} \leftarrow k$ 
    if  $div[k] < curr_{lcs}$  then
       $curr_{lcs} \leftarrow div[k], tmp_{thr} \leftarrow k$ 
    if  $k = M-1 \vee col[pref[k]] \neq col[pref[k+1]]$  then
      if  $k \neq M-1 \wedge div[k+1] < div[tmp_{thr}]$  then
         $push(t, k+1)$ 
      else
         $push(t, tmp_{thr})$ 
  return  $t$ 

```

---

### Threshold con bitvector

In questo caso le posizioni delle *threshold* vengono memorizzate tramite un *bitvector sparso* per ogni colonna  $k$ , denotato  $thr_k$ , avendo che  $thr_k[i] = 1$  sse  $i$  è l'indice di una *threshold*. Qualora il minimo *LCP* si ritrovi nell'indice della testa della run successiva, la posizione della *threshold* verrà comunque memorizzata all'indice della coda della run corrente. Purtroppo questa è una situazione di ambiguità, avendo che, seguendo la definizione sopra, avendo la *threshold* a fine run, bisognerebbe scegliere la testa della run successiva, qualora l'indice  $i$  si trovi esattamente a fine run. Invece, qualora la *threshold* sia a fine run a causa del fatto che il minimo *LCP* si trovi nella testa della run successiva, bisogna scegliere la coda della run precedente. L'unico modo per disambiguare è quindi effettuare *random access* al pannello per vedere quale sia la soluzione migliore, ovvero quale tra la coda della run precedente e la testa della run successiva siano relative alla riga del pannello originale con un suffisso comune alla query più lungo.

Purtroppo non è possibile salvare la *threshold* direttamente nella testa della run successiva in quanto questa potrebbe essere anche la posizione della *threshold* della run successiva e avere due *threshold* sovrapposte impedirebbe di capire a quale run appartiene una certa *threshold*, tramite la funzione *rank*.

Tale bitvector deve essere quindi aggiunto alle informazioni memorizzate per ogni singola colonna. Lo pseudocodice per la costruzione della componente **THR-BV** della colonna  $k$  è consultabile all'algoritmo 3.10 e, dovendo scorrere la colonna permutata dal prefix array  $a_k$  e dovendo accedere ai valori di  $l_k$ , tale operazione ha complessità in tempo proporzionale a:

$$\mathcal{O}(M) \tag{3.27}$$

### 3.3.3 Componente per i prefix array samples

Come introdotto parlando delle *matching statistics*, qualora si abbia un cambio di riga da memorizzare si seleziona sempre o quella relativa alla coda della run precedente o quella relativa alla testa della run successiva. Risulta quindi necessario, in colonna  $k$ , memorizzare i valori di  $a_k$  all'inizio e alla fine di ogni run, sotto forma di *vettori di interi bit-compressed*. Tali valori sono quindi un sample dei valori che permettono le permutazioni che costruiscono la *matrice PBWT* e quindi tale componente prende il nome di **PERM**.

All'algoritmo 3.11 è possibile analizzare lo pseudocodice del metodo usato per calcolare la componente **PERM** per la colonna  $k$ -esima. L'algoritmo, dovendo iterare l'intera colonna della *matrice PBWT* ha costo, in tempo:

$$\mathcal{O}(M) \tag{3.28}$$

---

**Algoritmo 3.10** Algoritmo per la costruzione della componente THR-BV.

---

```

function BUILD_THR_BV(col, pref, div) ▷ pref =  $a_k$ , div =  $l_k$ 
  currlcs ← 0, tmpthr ← 0
  thrs ← [0..0] ▷ bitvector sparso di lunghezza  $M$ 
  for every  $k \in [0, M)$  do
    if  $k = 0 \vee col[pref[k]] \neq col[pref[k-1]]$  then
      currlcs ← div[ $k$ ], tmpthr ←  $k$ 
    if div[ $k$ ] < currlcs then
      currlcs ← div[ $k$ ], tmpthr ←  $k$ 
    if  $k = M-1 \vee col[pref[k]] \neq col[pref[k+1]]$  then
      if  $k \neq M-1 \wedge div[k+1] < div[tmp_{thr}]$  then
        thrs[ $k$ ] ← 1
      else
        thrs[tmpthr] ← 1
  costruzione delle strutture rank/select per thr
return thr

```

---

La costruzione può essere fatta in contemporanea a quelle delle componenti già descritte, ovvero: MAP-INT/MAP-BV e THR-INT/THR-BV.

### 3.3.4 Componenti per il random access e le LCE query

Le ultime componenti da descrivere sono quelle atte a garantire il *random access* al testo e, nel caso dell'uso degli *SLP*, permettere il computo delle *LCE query*.

Parlando di strutture per il random access una differenza sostanziale tra l'uso di un *vettore di bitvector*, RA-BV, e quello dell'*SLP*, RA-SLP, è data dai tempi di accesso ai singoli elementi. Infatti, parlando di RA-BV, si ha accesso in tempo costante ad un qualsiasi elemento del pannello mentre, nel caso di RA-SLP, si ha che l'accesso ad ogni elemento è in tempo:

$$\mathcal{O}(\log(NM)) \tag{3.29}$$

La seconda differenza, già ampiamente introdotta e di fatto scontata, è data dalla dimensione delle due strutture dati, avendo che RA-BV memorizza  $\sim NM$  bit, dove il  $\sim$  è dato dai costi in memoria aggiuntivi dati dall'avere un vettore che memorizza i bitvector. Parlando invece di RA-SLP non si può avere una stima teorica dello spazio necessario ma, come si vedrà nel capitolo 4, i risultati quantitativi daranno prova della capacità di compressione degli *SLP*.

Parlando della componente LCE bisogna solo descrivere il modo con cui si ottiene la singola stringa che verrà compressa tramite *SLP*. In primis, le librerie per la costruzione di tale struttura assumono un input “monodimensionale”, ovvero una

---

**Algoritmo 3.11** Algoritmo per la costruzione della componente PERM per la colonna  $k$ .

---

```

1: function BUILD_PERM( $col, pref$ )  $\triangleright pref = a_k$ 
2:    $tmp_{beg} \leftarrow 0, beg_{run} \leftarrow \top$ 
3:    $samples_{beg} \leftarrow [], samples_{end} \leftarrow []$   $\triangleright$  vettori per i prefix array samples ad
   inizio/fine di ogni run
4:   for every  $k \in [0, height)$  do
5:     if  $beg_{run}$  then
6:        $tmp_{beg} \leftarrow pref[k]$ 
7:        $beg_{run} \leftarrow \perp$ 
8:     if  $k = height - 1 \vee col[pref[k]] \neq col[pref[k + 1]]$  then
9:        $push(samples_{beg}, tmp_{beg})$ 
10:       $push(samples_{end}, pref[k])$ 
11:       $beg_{run} \leftarrow \top$ 
12:   return ( $samples_{beg}, samples_{end}$ )

```

---

singola sequenze lineare. Inoltre, anche per permettere la costruzione efficiente della *PBWT*, e conseguentemente della *RLPBWT*, il pannello in input risulta essere trasposto, avendo che le righe nel file in input rappresentano i siti e non gli individui. Bisogna quindi in primis trasporre tale pannello. Per procedere ulteriormente bisogna però ricordare che sull'*SLP* si avrà necessità di effettuare *LCE query* che però, si anticipa, nel nostro pannello, devono essere fatte tra due righe da destra a sinistra (a differenza di quanto visto nel caso standard dove si confrontavano prefissi comuni). Per rendere possibile questa operazione quindi il pannello deve essere sia salvato come un'unica riga, per ottenerne l'*SLP*, che “da destra a sinistra”, per permettere le *LCE query*. Si procede quindi concatenando ogni riga, selezionandole consecutivamente e leggendone i singoli elementi da destra a sinistra.

**Esempio 25.** Si vede quindi un breve esempio.

Si assuma di avere il seguente pannello nel file in input.

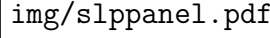
$$X = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Dove però come detto le righe sono i siti e le colonne i sample. Per ottenere l'*SLP*

bisogna quindi, in primis, trasporre la matrice:

$$X^T = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

A questo punto bisogna considerare l'ordine in cui si vorranno effettuare le LCE query. Ad esempio, prendendo la seconda e la terza riga, facendo partire il confronto dall'ultima colonna, avremmo una LCE query lunga 3, terminante nella prima colonna esclusa:



Si procede quindi salvando la sequenza lineare relativa ai pannelli come descritto sopra, ottenendo, con colorate gli stessi risultati della query fatta sopra:

0010 0110 0111 1100 0110

Si noti che qui si sono segnalate le varie righe con uno spazio ma solo per praticità "visiva".

In termini di complessità, si ricorda che, come per il *random access*, per il calcolo delle LCE query con SLP si ha un tempo proporzionale a:

$$\mathcal{O}(\log(NM)) \quad (3.30)$$

### 3.3.5 Componente per la struttura Phi

L'ottenimento dell'array *matching statistics* permette di sapere solo l'indice di una delle righe del pannello per le quali si ha un match con l'aplotipo query. Analogamente a quanto discusso in PHONI [8], anche per la RLPBWT si è pensato a due funzioni,  $\varphi$  e  $\varphi^{-1}$ , per il riconoscimento di tutte le righe del pannello per cui si ha il match. La struttura che permette il calcolo di tali funzioni è la componente denotata PHI.

L'intuizione alla base del ragionamento è molto semplice. Nell'ordinamento alla colonna  $k$ -esima, dato da  $a_k$ , tutte le righe per le quali si ha un match sono poste consecutivamente, questo a causa del fatto che l'ordinamento è lessicografico.

**DC** Qui non ho PLCP quindi calcolo ogni volta

**Definizione 29.** *Dati:*

- un pannello  $X$ , di dimensioni  $N \times M$
- una colonna  $k$ , il prefix array  $a_k$  e la sua permutazione inversa  $\alpha_k$

Si definiscono formalmente:

$$\varphi_k(p) = \begin{cases} \text{null} & \text{se } \alpha_k[p] = 0 \\ a_k[\alpha_k[p] - 1] & \text{altrimenti} \end{cases}, \forall p \in \{0, M-1\}$$

$$\varphi_k^{-1}(p) = \begin{cases} \text{null} & \text{se } \alpha_k[p] = M-1 \\ a_k[\alpha_k[p] + 1] & \text{altrimenti} \end{cases}, \forall p \in \{0, M-1\}$$

In altri termini, avendo  $a_k[j] = p$  si ha che:

$$\varphi_k(p) = \begin{cases} \text{null} & \text{se } j = 0 \\ a_k[j-1] & \text{altrimenti} \end{cases}, \forall p \in \{0, M-1\}$$

$$\varphi_k^{-1}(p) = \begin{cases} \text{null} & \text{se } j = M-1 \\ a_k[j+1] & \text{altrimenti} \end{cases}, \forall p \in \{0, M-1\}$$

#### VERIFICARE DEFINIZIONE IN QUANTO “NUOVA”

**Esempio 26.** *Per praticità si riporta un breve esempio.*

*Si ipotizzi di avere, come per l'esempio 15:*

$$a_6 = [14, 15, 0, 9, 10, 16, 8, 11, 12, 13, 18, 19, 1, 2, 3, 17, 4, 5, 6, 7]$$

$$\alpha_6 = [2, 12, 13, 14, 16, 17, 18, 19, 6, 3, 4, 7, 8, 9, 0, 1, 5, 15, 10, 11]$$

*Si fissa quindi  $p = 3$  e si ottengono:*

$$\varphi(3) = a_6[\alpha_6[3] - 1] = a_6[14 - 1] = a_6[13] = 2$$

$$\varphi^{-1}(3) = a_6[\alpha_6[3] + 1] = a_6[14 + 1] = a_6[15] = 17$$

Avendo quindi  $MS[i].row = p$  e  $MS[i].len = l$  basta iterare le righe a partire a  $p$  in  $a_i$ , che denotiamo con l'indice  $q$ , fino a che si ha  $LCE_k(x_p, x_q) \geq l$ . Ovviamente bisogna iterare in entrambe le direzioni. Tutte le righe  $x_q$  che soddisfano un match di lunghezza  $l$  con l'aplotipo query. L'algoritmo 3.12 rappresenta esattamente quanto detto, avendo che con la funzione *lce\_bounded* si limita il calcolo della *LCE* alla lunghezza  $l$ . La complessità temporale di questo algoritmo varia a seconda della componente **RA** (e della conseguente presenza della componente

LCE). Inoltre è difficile poter dare una stima asintotica in quanto varia sul numero di righe che presentano un certo *SMEM*, che denominiamo qui  $\nu$ , e dalla lunghezza dello stesso, indicata con  $\lambda$ . Quindi, si ha che, qualora si avesse la componente RA-BV, si avrebbe un tempo proporzionale a:

$$\mathcal{O}(\nu\lambda) \quad (3.31)$$

Mentre con l'uso della componente LCE, avendo il pannello in memoria con la componente RA-SLP, si avrebbe una complessità in tempo:

$$\mathcal{O}(\nu\lambda \log NM) \quad (3.32)$$

Questa è la definizione formale delle due funzioni ma, all'atto pratico, in memoria

---

**Algoritmo 3.12** Algoritmo per estendere uno SMEM in colonna  $k$  usando  $\varphi, \varphi^{-1}$ .

---

```

1: function EXTEND_MATCHES( $k, row, len$ )
2:    $haplos \leftarrow []$ 
3:    $check_{down} \leftarrow \top, check_{up} \leftarrow \top$ 
4:   while  $check_{down}$  do
5:      $down_{row} \leftarrow \varphi^{-1}(row, k)$ 
6:     if  $lce\_bounded(k, row, down_{row}, len)$  then
7:        $push(haplos, down_{row})$ 
8:        $row \leftarrow down_{row}$ 
9:     else
10:       $check_{down} \leftarrow \perp$ 
11:   while  $up_{down}$  do
12:      $up_{row} \leftarrow \varphi(row, k)$ 
13:     if  $lce\_bounded(k, row, up_{row}, len)$  then
14:        $push(haplos, up_{row})$ 
15:        $row \leftarrow up_{row}$ 
16:     else
17:       $check_{up} \leftarrow \perp$ 
18:   return  $haplos$ 

```

---

si hanno solo i *prefix array sample*, ad inizio e fine di ogni run, e nessuna informazione in merito alla *permutazione inversa* del *prefix array*. Si è quindi pensato ad una struttura dati, basata anch'essa su *bitvector sparsi*, che permettesse il calcolo delle due funzioni.



### Costruzione della struttura di supporto

L'idea di base per la costruzione della struttura a supporto delle **funzioni**  $\varphi$  e  $\varphi^{-1}$  si basa sul fatto che, data una colonna  $k$  e dati due valori consecutivi  $p$  e  $q$  in  $a_k$  (avendo  $a_k[i] = p$  e  $a_k[i+1] = q$ ), essi rimarranno consecutivi anche in  $a_{k+o}$ , *prefix array* dell'arbitraria colonna  $k+o$ , fino a che che  $x_p[k+o] \neq x_q[k+o]$ , ovvero fino a che, in colonna  $k+o$ , tali righe non corrisponderanno a due simboli diversi. Cruciale è che, in quella colonna,  $p$  sarà memorizzato come *prefix array sample* della fine della run  $r$  mentre  $q$  come *prefix array sample* dell'inizio della run  $r+1$ . Grazie a questa informazione si può costruire una struttura che, data una colonna arbitraria e un arbitrario valore di *prefix array*, permetta di computare  $\varphi$  e  $\varphi^{-1}$ . Tale struttura dati è composta da:

- un vettore di *sparse bitvector* per  $\varphi$ , che denotiamo con  $\Phi$ , tale che  $\Phi[i][j] = 1$  sse la riga  $i$  indicizza una testa di run alla colonna  $j$  nella *matrice PBWT*. Si ha quindi che  $\Phi$  ha dimensione  $M \times N$
- un vettore di *sparse bitvector* per  $\varphi^{-1}$ , che denotiamo con  $\Phi^{-1}$ , tale che  $\Phi^{-1}[i][j] = 1$  sse la riga  $i$  indicizza una coda di run alla colonna  $j$  nella *matrice PBWT*. Si ha quindi che  $\Phi^{-1}$  ha dimensione  $M \times N$
- un vettore di interi a supporto, denotato  $\Phi_{supp}$ , del vettore di *sparse bitvector* per  $\varphi$  che memorizza, per ogni 1 di tale vettore, il *prefix array sample* della coda della run precedente o l'altezza del pannello,  $M$ , qualora non si abbia alcuna run precedente
- un vettore di interi a supporto, denotato  $\Phi_{supp}^{-1}$ , del vettore di *sparse bitvector* per  $\varphi^{-1}$  che memorizza, per ogni 1 di tale vettore, il *prefix array sample* della testa della run successiva o l'altezza del pannello,  $M$ , qualora non si abbia alcuna run successiva.

Si ha quindi che la lunghezza della riga  $i$ -esima di  $\Phi_{supp}$  è uguale al numero di uni presenti nella riga  $i$ -esima di  $\Phi$ . Analogamente si ha per  $\Phi_{supp}^{-1}$ . In entrambi i casi, inoltre, si hanno  $M$  righe.

Al fine della costruzione bisogna, inoltre, sfruttare  $a_{N-1}$  per poter identificare quelle coppie di valori consecutivi non presenti nei vari *prefix array samples*, in modo che sia possibile effettuare le query per qualsiasi valore di *prefix array* in input.

L'algoritmo 3.13 riporta quindi la costruzione della struttura, iterando in primis i vari *prefix array samples* e completando i risultati con  $a_{N-1}$ . Tale algoritmo ha complessità in tempo, nel caso peggiore, pari a:

$$\mathcal{O}(NM) \tag{3.33}$$

Che si ha qualora ogni colonna della *matrice PBWT* abbia un numero di run pari all'altezza stessa della colonna. Indicando con  $\rho$  il numero medio complessivo di run, si può dire che la complessità nel caso medio è:

$$\mathcal{O}(N\rho) \quad (3.34)$$

### CAPIRE SE COMMENTARE ULTERIORMENTE LA COSTRUZIONE

Dal punto di vista delle query, data una colonna  $k$  e un valore di *prefix array*  $p$ ,

---

**Algoritmo 3.13** Algoritmo per la costruzione della componente PHI.

---

```

1: function BUILD_PHI(cols, panel, prefix)                                ▷ prefix is the last prefix array
2:    $\Phi \leftarrow [[0..0]..[0..0]]$ ,  $\Phi^{-1} \leftarrow [[0..0]..[0..0]]$       ▷ sparse bit vector panels for  $\varphi$  and  $\varphi^{-1}$ 
3:    $\Phi_{supp} = []$ ,  $\Phi_{supp}^{-1} = []$                                        ▷ vectors for  $\varphi$  and  $\varphi^{-1}$  row values
4:   for every  $k \in [0, |cols|)$  do
5:     for every  $i \in [0, |samples_{beg}|)$  do
6:        $\Phi[sample_{beg}^k[i]][k] \leftarrow 1$ 
7:       if  $i = 0$  then
8:          $push(\Phi_{supp}[sample_{beg}^k[i]], panel_{height})$ 
9:       else
10:         $push(\Phi_{supp}[sample_{beg}^k[i]], sample_{end}^k[i - 1])$ 
11:       $\Phi^{-1}[sample_{end}^k[i]][k] \leftarrow 1$ 
12:      if  $i = |sample_{beg}^k| - 1$  then
13:         $push(\Phi_{supp}^{-1}[sample_{end}^k[i]], panel_{height})$ 
14:      else
15:         $push(\Phi_{supp}^{-1}[sample_{end}^k[i]], sample_{beg}^k[i + 1])$ 
16:    for every  $k \in [0, |prefix|)$  do
17:      if  $\Phi[k][|\Phi[k]| - 1] = 0$  then
18:         $\Phi[k][|\Phi[k]| - 1] \leftarrow 1$ 
19:      if  $k = 0$  then
20:         $push(\Phi_{supp}[prefix[k]], panel_{height})$ 
21:      else
22:         $push(\Phi_{supp}[prefix[k]], prefix^k[i - 1])$ 
23:      if  $\Phi^{-1}[k][|\Phi[k]| - 1] = 0$  then
24:         $\Phi^{-1}[k][|\Phi[k]| - 1] \leftarrow 1$ 
25:      if  $k = |prefix| - 1$  then
26:         $push(\Phi_{supp}^{-1}[prefix[k]], panel_{height})$ 
27:      else
28:         $push(\Phi_{supp}^{-1}[prefix[k]], prefix^k[i + 1])$ 
29:    costruzione della struttura rank per ogni bitvector sparso  $\Phi$  e  $\Phi^{-1}$ 

```

---

si procede quindi nel seguente modo:

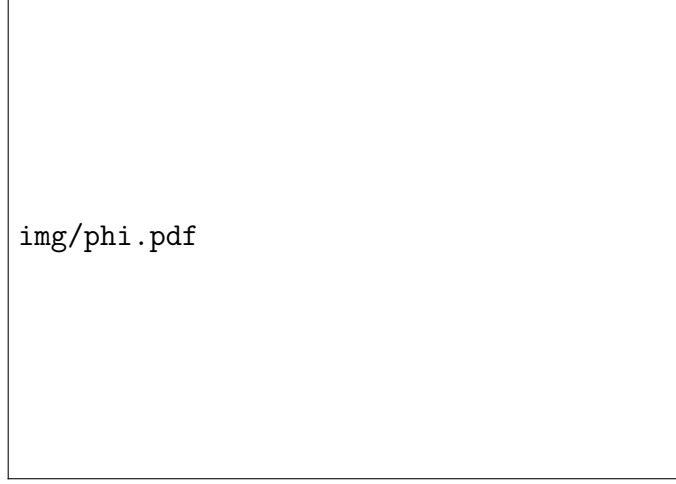
- per la funzione  $\varphi$  si effettua la  $rank^\varphi(k)$  sulla riga  $p$  di  $\Phi$ , avendo che:

$$\varphi_k(p) = \begin{cases} null & \text{se } \Phi_{supp}^p[rank_p^\varphi(k)] = M \\ \Phi_{supp}^p[rank_p^\varphi(k)] & \text{altrimenti} \end{cases}$$

- per la funzione  $\varphi^{-1}$  si effettua la  $rank^{\varphi^{-1}}(k)$  sulla riga  $p$  di  $\Phi^{-1}$ , avendo che:

$$\varphi_k^{-1}(p) = \begin{cases} null & \text{se } \Phi_{supp}^{-1 p}[rank_p^{\varphi^{-1}}(k)] = M \\ \Phi_{supp}^{-1 p}[rank_p^{\varphi^{-1}}(k)] & \text{altrimenti} \end{cases}$$

**Esempio 27.** Si ha la seguente situazione nella matrice PBWT:



Dove si noti che, a parità di colore, si ha lo stesso simbolo tra due indici consecutivi.

In colonna  $k$ , che per praticità assumiamo essere  $k = 0$ , si vorrebbe avere informazione in merito a  $\varphi_k(j)$  e  $\varphi_k^{-1}(m)$ .

Si nota che, per definizione della struttura dati, si ha che (limitandoci alle colonne della figura):

$$\Phi_j = [0, 0, 0, 1, 0, \dots]$$

$$\Phi_m^{-1} = [0, 0, 0, 1, 0, \dots]$$

In quanto, in entrambi i casi, rispettivamente per la riga  $j$  e la riga  $m$ , in colonna  $k + 3$ , si che  $j$  è il prefix array di una testa di run mentre  $m$  di una coda di run. In colonna  $k + 3$  si conoscono anche, rispettivamente,  $i$ , prefix array della coda della run precedente a quella di  $j$ , e  $n$ , prefix array della testa della run successiva quella di  $m$ . Si ottengono quindi:

$$\Phi_{supp} = [i, \dots]$$

$$\Phi_{supp}^{-1} = [n, \dots]$$

Si vogliono quindi calcolare  $\varphi_0(j)$  e  $\varphi_0^{-1}(m)$ . Si ha:

$$\Phi_{supp}^j[rank_j^{\varphi}(0)] = \Phi_{supp}^j[0] = i$$

$$\Phi_{supp}^{-1 m}[rank_m^{\varphi^{-1}}(0)] = \Phi_{supp}^{-1 m}[0] = n$$

Si noti che uguali risultati si avrebbero per  $k + 1$ ,  $k + 2$  e  $k + 3$ .

DC SISTEMA-  
RE UN PO'  
TUTTO

## 3.4 Strutture dati per la RLPBWT

Assemblando le componenti descritte nella sezione precedente si ottengono otto varianti della *RLPBWT*:

DC ref sezione

- due strutture dati composte unicamente dalle componenti dedicate al mapping e dall'intero *array LCP*. Tali strutture sono nominate:

1 MAP-INT + LCP

2 MAP-BV + LCP

Queste soluzioni, che non permettono di sapere quali righe del pannello presentano uno *SMEM*, terminante in una certa colonna, ma solo quante esse siano, sono basate su una re-implementazione dell'*algoritmo 5 di Durbin*

- quattro strutture composte per il calcolo degli *SMEM* tramite la computazione in due passaggi dell'array delle *matching statistics*, in modo analogo a quanto introdotto in *MONI* per la *BWT*, tramite *threshold* e *random access* al pannello. Il pannello, come anticipato, è memorizzato come *vettore di bitvector* o come *SLP*. L'algoritmo necessita, inoltre, sia della componente atta al mapping che di quella relativa ai *sample di prefix array*. Infine, al fine di estendere il riconoscimento a tutte le righe che presentano uno *SMEM* fino ad una certa colonna, si necessita della struttura che permette di "emulare" le *funzioni*  $\varphi$  e  $\varphi^{-1}$ . Tali strutture sono nominate:

3 MAP-INT + THR-INT + RA-BV + PERM + PHI

4 MAP-INT + THR-INT + RA-SLP + PERM + PHI

5 MAP-BV + THR-BV + RA-BV + PERM + PHI

6 MAP-BV + THR-BV + RA-SLP + PERM + PHI

- due strutture composte per il calcolo degli *SMEM* tramite la computazione in un singolo passaggio dell'array delle *matching statistics*, in modo analogo a quanto introdotto in *PHONII* per la *BWT*, tramite l'uso delle *LCE query*, con le quali, rispetto alle quattro precedenti, si sostituisce l'uso delle *threshold* e del *random access* al pannello. Tali strutture sono nominate:

7 MAP-INT + LCE + PERM + PHI

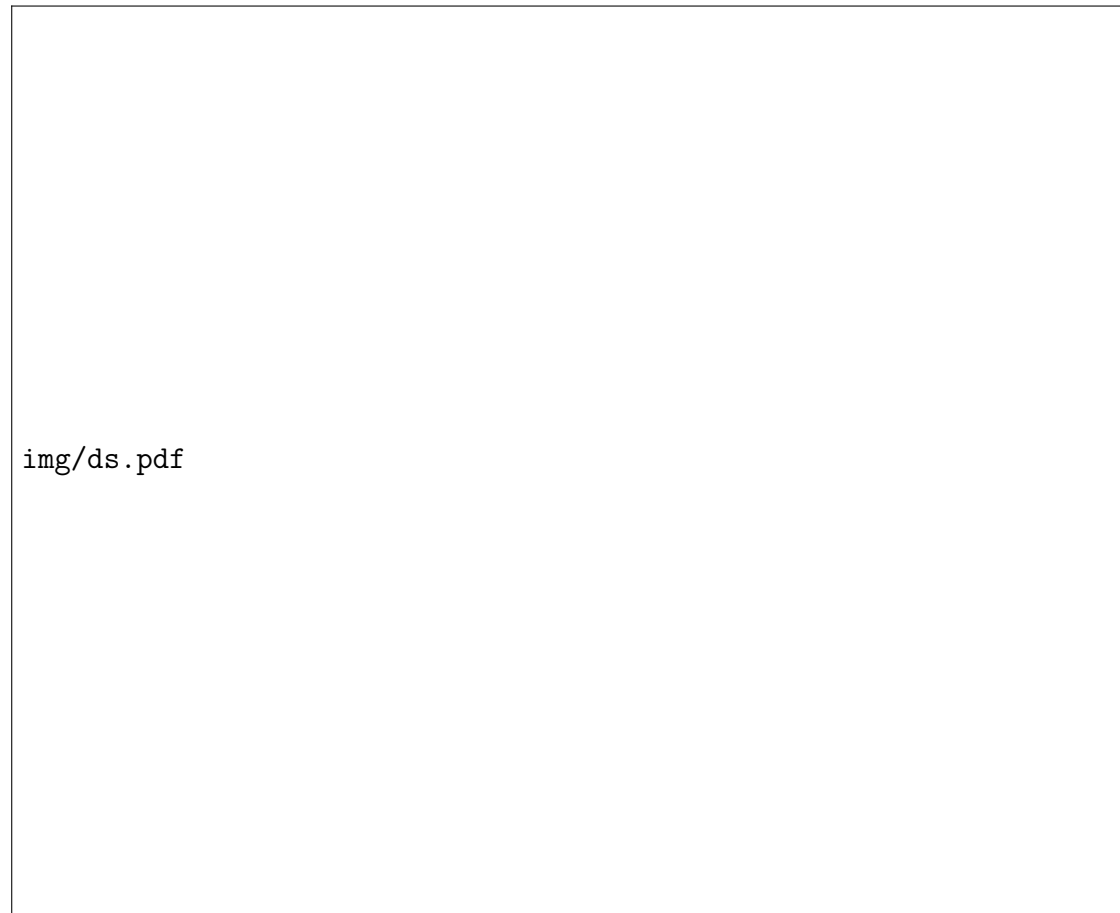


Figura 3.1: Schema grafico dell'ottenimento delle otto strutture dati.

$$8 \text{ MAP-BV} + \text{LCE} + \text{PERM} + \text{PHI}$$

Una visualizzazione grafica di quanto descritto è rappresentata alla figura 3.1.

### 3.4.1 Calcolo degli SMEM con LCP

Questa prima soluzione per il calcolo degli SMEM con un apotipo esterno è quella che può essere effettuata tramite le strutture dati composte:

- MAP-INT + LCP
- MAP-BV + LCP

I due algoritmi riprendono esattamente quanto discusso nell'*algoritmo 5 di Durbin*. Tali algoritmi, di fatto, non sfruttano l'uso delle matching statistics e sono limitati

dal non poter calcolare quali righe presentano un solo SMEM, calcolando solo quante siano. Il secondo limite è dato dal fatto che necessitano di avere interamente in memoria l'*LCP* array. Questo comporta avere in memoria una struttura non run-length encoded occupante  $4NM$  bytes.

Il metodo procede, quindi, con l'aggiornamento dei tre indici  $e_k$ ,  $f_k$  e  $g_k$ , avendo che gli ultimi due possono assumere qualsiasi valore in  $\{0, \dots, M\}$ , come con la *PBWT* classica. Avendo memorizzato solo informazioni relative alle *run* bisogna quindi, ogni volta, ricondurre l'indice alla run corretta. Inoltre Durbin sfruttava il *random access* al pannello, avendo in memoria sia il pannello che il *prefix array*, al fine di aggiornare il valore di  $e_k$ . In entrambe le strutture dati, però, non si ha in memoria né il *prefix array* né il pannello ma solo la rappresentazione compatta della *matrice PBWT*. Si è quindi dovuto pensare ad un metodo che ricomponga data una riga  $x_j$  del pannello  $X$  a partire da un elemento, indicizzato con  $a_{k+1}[i] = j$ , con  $0 \leq i < M$ , alla colonna  $k + 1$ , della *matrice PBWT*, muovendosi da destra a sinistra e seguendo in modo inverso la permutazione che produce il *prefix array*. In altri termini, tale metodo permette un *mapping inverso* che segua una riga del pannello originale nella *matrice PBWT*.

Per ottenere l'indice alla colonna  $k$ -esima da cui “proviene” la riga  $j$ , indicizzata all'indice  $i$  in colonna  $k + 1$ , si inizia analizzando il valore  $c[k]$ . Infatti, se  $i < c[k]$ , allora sicuramente, in colonna  $k$ , è un indice corrispondente a  $\sigma = 0$  quello dal quale proviene, ricordando come la costruzione della colonna  $k + 1$  nella *matrice PBWT* si abbia grazie ad ordinamento stabile. Si sfruttano così o l'array  $p_k$  o le funzioni  $rank_{h_k}$  e  $select_{h_k}$  per risalire all'indice in colonna  $k$ , calcolando prima l'indice di run e l'eventuale offset, per il quale il mapping porta all'indice  $i'$  in colonna  $k + 1$ , seguendo “virtualmente” la riga  $x_j$  del pannello originale. Per quanto riguarda la **MAP-INT + LCP** si ha lo pseudocodice per il mapping inverso consultabile all'algoritmo 3.14 mentre per quanto riguarda la **MAP-BV + LCP** si ha l'algoritmo 3.15. Parlando in termini di complessità in tempo si ha che, nel caso della componente **MAP-INT**, si ha, con  $r$  numero di run alla colonna  $k$ , un caso peggiore proporzionale a:

$$\mathcal{O}(r) \quad (3.35)$$

Nel caso, invece, in cui si ha la componente **MAP-BV**, si ha:

$$\mathcal{O}\left(\log \frac{M}{r}\right) \quad (3.36)$$

Si procede quindi riadattando l'algoritmo di Durbin all'uso delle *run*, ottenendo, ad ogni step, i medesimi valori per  $e_k$ ,  $f_k$  e  $g_k$ . Le uniche differenze sono:

- il calcolo del mapping necessita dell'estrazione dei valori  $u$  e  $v$ , tenendo conto esplicito degli offset nel caso della **MAP-INT + LCP**

DC Approfondire?

---

**Algoritmo 3.14** Algoritmo per il mapping inverso con la MAP-INT + LCP.

---

```

1: function REVERSE_MAP( $k, i$ )            $\triangleright k$  indice di colonna,  $i$  indice di riga
2:   if  $k = 0$  then                        $\triangleright$  by design
3:     return 0
4:    $k \leftarrow k - 1$ 
5:    $c \leftarrow \text{rlpbwt}[k].c$ 
6:    $u \leftarrow 0, v \leftarrow 0, \text{offset} \leftarrow 0, \text{run} \leftarrow 0, \text{found} \leftarrow \perp$ 
7:   if  $i < c$  then
8:      $u \leftarrow i$ 
9:      $\text{prev}_0 \leftarrow 0, \text{next}_0 \leftarrow 0$ 
10:    for every  $j \in [0, |p_k|)$  do
11:       $(\text{prev}_0, \_) \leftarrow \text{uvtrick}(k, j)$ 
12:       $(\text{next}_0, \_) \leftarrow \text{uvtrick}(k, j + 1)$ 
13:      if  $\text{prev}_0 \leq u < \text{next}_0$  then
14:         $\text{run} \leftarrow j, \text{found} \leftarrow \top$ 
15:        break
16:    if  $\neg \text{found}$  then
17:       $\text{run} \leftarrow |p_k| - 1$ 
18:       $(\text{curr}_u, \_) \leftarrow \text{uvtrick}(k, \text{run}), \text{offset} \leftarrow u - \text{curr}_u$ 
19:      return  $p_k[\text{run}] + \text{offset}$ 
20:  else
21:     $v \leftarrow i - c$ 
22:     $\text{prev}_1 \leftarrow 0, \text{next}_1 \leftarrow 0$ 
23:    for every  $j \in [0, |p_k|)$  do
24:       $(\_, \text{prev}_1) \leftarrow \text{uvtrick}(k, j)$ 
25:       $(\_, \text{next}_1) \leftarrow \text{uvtrick}(k, j + 1)$ 
26:      if  $\text{prev}_1 \leq v < \text{next}_1$  then
27:         $\text{run} \leftarrow j, \text{found} \leftarrow \top$ 
28:        break
29:    if  $\neg \text{found}$  then
30:       $\text{run} \leftarrow |p_k| - 1$ 
31:       $(\text{curr}_v, \text{curr}_u) \leftarrow \text{uvtrick}(k, \text{run}), \text{offset} \leftarrow v - \text{curr}_v$ 
32:      return  $p_k[\text{run}] + \text{offset}$ 

```

---

---

**Algoritmo 3.15** Algoritmo per il mapping inverso con la MAP-BV + LCP.

---

```

1: function REVERSE_MAP( $k, i$ )           ▷  $k$  indice di colonna,  $i$  indice di riga
2:   if  $k = 0$  then                       ▷ by design
3:     return 0
4:    $k \leftarrow k - 1$ 
5:    $c \leftarrow rlpbwt[k].c$ 
6:   if  $i < c$  then
7:     if  $start_k$  then
8:        $run \leftarrow rank_u^k(i) \cdot 2$ 
9:     else
10:       $run \leftarrow rank_u^k(i) \cdot 2 + 1$ 
11:     $i_{run} \leftarrow 0$ 
12:    if  $run \neq 0$  then
13:       $i_{run} \leftarrow select_h^k(run) + 1$ 
14:     $(prev_0, \_) \leftarrow uvtrick(k, i_{run})$ 
15:    return  $i_{run} + (i - prev_0)$ 
16:  else
17:    if  $start_k$  then
18:       $run \leftarrow rank_v^k(i) \cdot 2 + 1$ 
19:    else
20:       $run \leftarrow rank_v^k(i) \cdot 2$ 
21:     $i_{run} \leftarrow 0$ 
22:    if  $run \neq 0$  then
23:       $i_{run} \leftarrow select_h^k(run) + 1$ 
24:     $(\_, prev_1) \leftarrow uvtrick(k, i_{run})$ 
25:    return  $i_{run} + (i - (c + prev_1))$ 

```

---



- non si ha *random access* al pannello quindi bisogna procedere ogni volta con il'inverso del mapping e il calcolo del simbolo a partire dall'indice della run
- non si ha il *prefix array* in memoria quindi non è possibile sapere quali siano le righe che stanno matchando fino alla colonna  $k$  ma solo quante, sapendo che sono  $g_k - f_k$

Anche in questo caso lo pseudocodice è consultabile all'algoritmo 3.16. Calcolare la complessità di tale algoritmo non è semplice, come già visto nel caso dell'algoritmo 5 di Durbin. In modo analogo si può comunque intuire come i vari cicli interni siano limitati superiormente dalla larghezza del pannello e dai tempi di mapping. Questo si può stimare in quanto le occorrenze dei cicli interni sono proporzionali al numero di SMEM e al numero di "step" all'indietro necessari a ri-computare il nuovo intervallo, numero di step che scala sul numero di caratteri in overlap tra due SMEM consecutivi. Fatta questa premessa si può stimare che il calcolo degli SMEM con la struttura MAP-INT + LCP è proporzionale, con  $\rho$  numero medio di run per una colonna, a:

$$\mathcal{O}(N \log \rho) \quad (3.37)$$

Nel caso, invece, della struttura MAP-BV + LCP si ha:

$$\mathcal{O}\left(N \log \frac{M}{\rho}\right) \quad (3.38)$$

DC APPRO-  
FONDIRE  
SPIEGAZIONE  
ALGORITMI

### 3.4.2 Calcolo degli SMEM con matching statistics

L'obiettivo di questa tesi era quello di applicare i metodi e gli algoritmi già studiati per la *BWT*, riferendosi al calcolo dei *MEM* a partire dall'array delle *matching statistics*, alla *PBWT*.

Nelle sei strutture dati dedicate al calcolo degli SMEM tramite *matching statistics* si riconoscono le due modalità già descritte con *MONI* e *PHONI*:

1. calcolare l'array MS in due passaggi sfruttando il *random access* al pannello per calcolare i vari  $MS[i].len$
2. calcolare l'array MS in un passaggio sfruttando le *LCE query* sia per scegliere i vari  $MS[i].row$  che per calcolare, in contemporanea, i vari  $MS[i].len$

---

**Algoritmo 3.16** Calcolo degli SMEM con aplotipo esterno per MAP-INT/BV + LCP, con eventuali usi diversificati per MAP-INT e MAP-BV segnalati con “oppure”.

---

```

1: function EXTERNAL_MATCHES( $z$ ) ▷ si assume  $|z| = N$ 
2:    $f \leftarrow 0, f_{run} \leftarrow 0, f' \leftarrow 0$ 
3:    $g \leftarrow 0, g_{run} \leftarrow 0, g' \leftarrow 0$ 
4:    $e \leftarrow 0, nh \leftarrow 0$ 
5:   for every  $k \in [0, |z|)$  do
6:      $f_{run} \leftarrow index\_to\_run(f, k)$  oppure  $f_{run} \leftarrow rank_h^k(f)$ 
7:      $g_{run} \leftarrow index\_to\_run(g, k)$  oppure  $g_{run} \leftarrow rank_h^k(g)$ 
8:      $f' \leftarrow w(k, f, z[k]), g' \leftarrow w(k, g, z[k]), nh \leftarrow g - f$ 
9:     if  $f' < g'$  then
10:       $f \leftarrow f', g \leftarrow g'$ 
11:   else
12:     if  $k \neq 0$  then
13:       memorizzazione degli SMEM tra le colonne  $[e, k - 1]$  con  $nh$  aplotipi
14:     if  $f' = |l_{k+1}|$  then  $e \leftarrow k + 1$  else  $e \leftarrow k - l_{k+1}[f']$ 
15:     if  $(z[e] = 0 \wedge f' > 0) \vee f' = M$  then
16:        $f' \leftarrow g' - 1$ 
17:       if  $e \geq 1$  then
18:          $f_{rev} \leftarrow f', k' \leftarrow k + 1$ 
19:         while  $k' \neq e - 1$  do
20:            $f_{rev} \leftarrow reverse\_map(k', f_{rev}), k' \leftarrow k' - 1$ 
21:            $run \leftarrow index\_to\_run(f_{rev}, k')$  oppure  $run \leftarrow rank_h^{k'}(f_{rev})$ 
22:            $symb \leftarrow get\_symbol(start_{k'}, run)$ 
23:           while  $e > 0 \wedge z[e - 1] = symb$  do
24:              $e \leftarrow e - 1, f_{rev} \leftarrow reverse\_map(e, f_{rev})$ 
25:              $run \leftarrow index\_to\_run(f_{rev}, e - 1)$  oppure  $run \leftarrow rank_h^{e-1}(f_{rev})$ 
26:              $symb \leftarrow get\_symbol(start_{e-1}, run)$ 
27:           while  $f' > 0 \wedge (k + 1) - l_{k+1}[f] \leq e$  do  $f' \leftarrow f' - 1$ 
28:            $f \leftarrow f', g \leftarrow g'$ 
29:       else
30:          $g' \leftarrow f' - 1$ 
31:         if  $e \geq 1$  then
32:            $f_{rev} \leftarrow f', k' \leftarrow k + 1$ 
33:           while  $k' \neq e - 1$  do
34:              $f_{rev} \leftarrow reverse\_map(k', f_{rev}), k' \leftarrow k' - 1$ 
35:              $run \leftarrow index\_to\_run(f_{rev}, k')$  oppure  $run \leftarrow rank_h^{k'}(f_{rev})$ 
36:              $symb \leftarrow get\_symbol(start_{k'}, run)$ 
37:             while  $e > 0 \wedge z[e - 1] = symb$  do
38:                $e \leftarrow e - 1, f_{rev} \leftarrow reverse\_map(e, f_{rev})$ 
39:                $run \leftarrow index\_to\_run(f_{rev}, e - 1)$  oppure  $run \leftarrow rank_h^{e-1}(f_{rev})$ 
40:                $symb \leftarrow get\_symbol(start_{e-1}, run)$ 
41:             while  $e < M \wedge (k + 1) - l_{k+1}[g'] \leq e$  do  $g' \leftarrow g' + 1$ 
42:              $f \leftarrow f', g \leftarrow g'$ 
43:         if  $f < g$  then
44:            $nh \leftarrow g - f$ 
45:           memorizzazione degli SMEM tra le colonne  $[e, |z| - 1]$  con  $nh$  aplotipi

```

---

### Calcolo dell'array delle matching statistics tramite threshold

Questa prima soluzione, necessitando sia della componente THR-INT/THR-BV che della componente RA-BV/RA-SLP, è relativa alle seguenti strutture dati composte:

- MAP-INT + THR-INT + RA-BV + PERM + PHI
- MAP-INT + THR-INT + RA-SLP + PERM + PHI
- MAP-BV + THR-BV + RA-BV + PERM + PHI
- MAP-BV + THR-BV + RA-SLP + PERM + PHI

Tra le quali le uniche differenze si riscontrano nei tempi d'esecuzione e nella memoria richiesta.

Vediamo il funzionamento dell'algoritmo. Sia data  $t$  la posizione della *threshold* nella run corrente, in colonna  $k$ , e si supponga che tale run, con testa all'indice  $h$ , non sia associata al simbolo desiderato, ovvero  $z[k]$ . Si supponga che, con il mapping, si sia arrivati all'indice  $i$  della colonna  $k$ . Si supponga inoltre che la run successiva abbia testa in indice  $e$ . Si hanno due casi possibili, denotando con  $LCS(x, y)$  il *longest common suffix* tra le stringhe  $X$  e  $Y$  e con  $a_k$  il *prefix array* in colonna  $k$ :

**DC** Cercare di capire come legare a R-index

1.  $i < t$  allora, per definizione di *threshold*:

$$LCS(z[0, k], x_{a_k[h-1]}[0, k]) \geq LCS(z[0, k], x_{a_k[e]}[0, k])$$

Quindi si ha che  $MS[k].row = a_k[h-1]$  e il mapping potrà proseguire dall'indice  $h-1$

2.  $i \geq t$  allora, per definizione di *threshold*:

$$LCS(z[0, k], x_{a_k[s-1]}[0, k]) \leq LCS(z[0, k], x_{a_k[e]}[0, k])$$

Quindi si ha che  $MS[k].row = a_k[e]$  e il mapping potrà proseguire dall'indice  $e$

Qualora una colonna presenti solo simboli  $\sigma \neq z[k]$ , per convenzione, si imposta che  $MS[k].row = M$  e si ricomincia, in colonna  $k+1$ , dall'ultima posizione, indicizzata nel pannello originale dal valore finale del *prefix array sample* dell'ultima run.

**Esempio 28.** Si vede, quindi, un esempio di funzionamento delle *threshold* per la scelta della riga da memorizzare in  $MS[i].row$  dopo un mismatch.

Si prenda pannello visto all'esempio 15 e si effettui la permutazione secondo  $a_2$ :

X	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14
00	1	0	0	1	0	0	0	0	0	0	0	1	1	0	1
01	1	0	0	1	1	0	0	1	0	0	0	0	0	1	1
02	1	0	0	1	1	0	0	1	0	0	0	1	0	0	1
03	1	0	0	1	1	0	0	1	0	0	0	1	0	0	1
04	0	1	0	1	0	1	0	0	0	0	0	1	0	0	1
05	0	1	0	1	0	1	0	0	0	0	0	1	0	0	1
06	0	1	0	1	0	1	0	0	0	0	0	1	0	0	1
07	0	1	0	1	0	1	0	0	0	0	0	0	1	0	1
08	0	1	0	0	1	0	0	0	0	1	1	1	0	0	1
09	0	1	0	1	0	0	0	0	1	0	0	0	0	1	1
10	0	1	0	1	0	0	0	0	1	0	0	0	0	1	1
11	0	1	0	0	1	0	0	0	0	0	1	1	0	0	0
12	0	1	0	0	1	0	0	0	1	0	1	1	0	0	1
13	0	1	0	0	1	0	0	0	1	0	1	1	0	0	1
14	0	1	0	0	0	0	0	0	1	0	0	0	1	0	1
15	0	1	0	0	0	0	0	0	1	0	0	0	1	0	1
16	0	1	0	1	0	0	0	0	0	0	0	1	1	0	1
17	0	1	1	0	1	0	0	0	0	0	0	1	0	0	1
18	0	1	1	0	1	0	1	0	0	0	0	0	1	0	1
19	1	1	0	0	0	1	0	0	0	0	0	1	1	0	1

Si prenda la seconda run, di simboli  $\sigma = 1$ , indicizzata tra 17 e 18.

Si supponga che, tramite il mapping, si sia arrivati alla riga 17 ma che si abbia  $z[2] = 0$ . la scelta è quindi tra la coda della run precedente, avendo che  $a_2[16] = 16$  o la testa della run successiva, avendo che  $a_2[19] = 17$ . Si può notare come il minimo LCP si trovi, per la run, all'indice 18 (a causa del fatto che il minimo LCP è all'indice 19, quello della testa della run successiva). Si può quindi proseguire o con la riga. Questo significa che il suffisso comune più lungo con la query si ha con la riga 16 del pannello, per definizione di threshold, avendo che questa sarà memorizzata nell'array  $MS$ :

$$MS[2].row = 16$$

Successivamente, tramite random access al testo, confrontando la riga  $x_{16}$  e la query  $z$ , fino alla colonna  $k = 2$ , si potrà calcolare che  $MS[2].len = 3$ .

Una volta computato tutti i valori  $MS[i].row$  per calcolare i vari  $MS[i].len$  si scorre da sinistra a destra calcolando la lunghezza dello SMEM facendo random access al pannello e confrontando la query  $z$  con la riga  $MS[i].row$ . Si assuma infatti di aver calcolato  $MS[i-1].len$  e di voler calcolare  $MS[i].len$ . Si hanno tre casi possibili:

1.  $MS[i].row = M$  e in tal caso, avendo segnalata l'inesistenza di alcuno SMEM, si ha che  $MS[i].len = 0$

2.  $MS[i].row = MS[i-1].row$ , avendo  $i \neq 0$  e  $MS[i-1].len \neq 0$ , allora si sta seguendo la stessa riga che si seguiva in colonna  $i-1$  e quindi, banalmente,  $MS[i].len = MS[i-1].len + 1$
3. in qualsiasi altro caso bisogna confrontare, a partire dalla colonna  $i$ , la query  $z$  con la riga  $MS[i].row$  del pannello da destra a sinistra, fino a che non si trova un mismatch, calcolando la lunghezza  $l$  del suffisso comune tra esse e memorizzando tale valore, tramite  $MS[i].len = l$

In fase di costruzione delle lunghezze è possibile anche riportare gli *SMEM*, terminanti in colonna  $i$ , qualora:

- $MS[i].len \geq MS[i+1].len \wedge MS[i].len \neq 0$
- si è arrivati a fine array, avendo  $i = N - 1 \wedge MS[i].len \neq 0$

Come si può verificare nell'esempio 20.

L'algoritmo per il match tramite *threshold* è visualizzabile all'algoritmo 3.17. All'algoritmo 3.18 si riporta il metodo di “update” delle informazioni nel passaggio dalla colonna  $k$  alla colonna  $k+1$ , di complessità pari a quelle per effettuare il mapping. Si noti che è possibile usare la funzione *lf*, spiegata in precedenza, in quanto, avendo per costruzione  $y_{curr\_index}^k[k] = z[k]$  nel momento dell'applicazione della funzione, si ha che la funzione “segue” esattamente una certa riga da una colonna alla successiva nella permutazione data dai *prefix array*. Anche in questo caso la stima delle complessità non è di facile ottenimento. Dividendo nelle varie parti l'algoritmo si ha che:

- il calcolo dei valori *row* dell'array delle matching statistics varia a seconda dell'uso della componente MAP-INT o MAP-BV. Il costo della funzione *down*, variabile a seconda della componente RA e dell'eventuale componente LCE, risulta trascurabile vista la bassa frequenza d'uso, in termini probabilistici. Si ha quindi che, con  $\rho$  numero medio di run per colonna, usando MAP-INT si ha tempo proporzionale a:

$$\mathcal{O}(N \log \rho) \quad (3.39)$$

Mentre con MAP-BV è proporzionale a:

$$\mathcal{O}\left(N \log \frac{M}{\rho}\right) \quad (3.40)$$

- il calcolo dei valori *len* dell'array delle matching statistics è il più complesso da stimare in termini di complessità asintotica. Questa difficoltà è dovuta

dal fatto che gli accessi al pannello vengono fatti solo quando  $MS.row[i] \neq MS.row[i - 1]$ . Per semplicità denotiamo con  $\gamma$  il numero di accessi al pannello e, nel caso della componente RA-BV, si ha che il calcolo complessivo delle lunghezze è proporzionale a:

$$\mathcal{O}(N\gamma) \quad (3.41)$$

Mentre con RA-SLP è proporzionale a:

$$\mathcal{O}(N\gamma \log(NM)) \quad (3.42)$$

- si ha infine il calcolo effettivo degli *SMEM* e di tutte le righe del pannello per le quali si hanno tali match. Denotando con  $\delta$  il numero complessivo di accessi alla componente RA-BV o di usi della componente LCE, qualora disponibile, e con  $\mu$  il numero di *SMEM*, si ha che, nel primo caso, la complessità di tale operazione è proporzionale a:

$$\mathcal{O}(\mu\delta) \quad (3.43)$$

mentre nel caso di uso dell'*SLP* e della componente LCE:

$$\mathcal{O}(\mu\delta \log(NM)) \quad (3.44)$$

Facendo una stima complessiva si può ipotizzare come la struttura MAP-BV + THR-BV + RA-SLP + PERM + PHI, a causa della maggior lentezza in fase di mapping e di accesso al pannello per il calcolo delle lunghezze, sia quella con prestazioni peggiori mentre, per il ragionamento inverso, la struttura MAP-INT + THR-INT + RA-BV + PERM + PHI sia quella con le migliori performance dal punto di vista del tempo macchina.

In termini di memoria, invece, la struttura MAP-INT + THR-INT + RA-SLP + PERM + PHI risulta essere la più vantaggiosa mentre la struttura MAP-BV + THR-BV + RA-BV + PERM + PHI la peggiore, per le stime viste nelle sezioni precedenti dedicate alle singole componenti.

### Calcolo dell'array delle matching statistics tramite LCE query

Come anticipato, grazie all'uso delle *LCE query* è possibile calcolare l'array delle *matching statistics* in un solo scorrimento da sinistra a destra. Infatti è possibile usare tali query per calcolare non solo quale nuova sequenza scegliere in caso di mismatch con l'aplotipo query in colonna  $i$ , come si faceva con l'uso delle *threshold*, ma anche di computare la lunghezza del suffisso comune tra essa e l'aplotipo query, calcolando nello stesso momento sia  $MS[i].row$  che  $MS[i].len$ .

Tale soluzione è quindi relativa alle seguenti strutture dati composte:

---

**Algoritmo 3.17** Calcolo degli SMEM con aplotipo esterno con componenti MAP-INT/BV, THR-INT/BV (i cui usi diversificati di entrambe le componenti sono segnalati con “oppure”), RA-BV/SLP, PERM e PHI.

---

```

1: function EXTERNAL_MATCHES( $z$ )
2:    $ms_{row} \leftarrow [0..0]$ ,  $ms_{len} \leftarrow [0..0]$  ▷ vettore  $MS$  di lunghezza  $|z|$ 
3:    $curr_{row} \leftarrow rlpbwt[0].samples_{end}[rlpbwt[0].samples_{end} - 1]$ 
4:    $curr_{index} \leftarrow curr_{row}$ 
5:    $curr_{run} \leftarrow index\_to\_run(curr_{index}, 0)$  oppure  $curr_{run} \leftarrow rank_h^0(curr_{index})$ 
6:    $symb \leftarrow get\_symbol(start_0, curr_{run})$  ▷ Costruzione righe dell'array  $MS$ 
7:   for every  $k \in [0, |z|)$  do
8:     if  $z[k] = symb$  then
9:        $ms_{row}[k] \leftarrow curr_{row}$ 
10:      if  $k \neq |z| - 1$  then  $(curr_{index}, curr_{run}, symb) \leftarrow UPDATE(k, curr_{index}, z)$ 
11:    else
12:       $curr_{thr} \leftarrow t_k[curr_{run}]$  oppure  $curr_{thr} \leftarrow rank_t^k(curr_{index})$ 
13:       $force_{down} \leftarrow \top$  sse l'indice è sovrapposto ad una threshold non in coda di run
14:       $force_{down} \leftarrow \top$  sse l'indice è sovrapposto ad una threshold in coda di run e  $DOWN(\dots) = \top$ 
15:      if  $|samples_{beg}^k| = 1$  then
16:         $ms_{row}[k] \leftarrow M$ 
17:        if  $k \neq |z| - 1$  then
18:           $curr_{row} \leftarrow rlpbwt[k+1].samples_{end}[rlpbwt[k+1].samples_{end} - 1]$ 
19:           $curr_{index} \leftarrow M - 1$ 
20:           $curr_{run} \leftarrow index\_to\_run(curr_{index}, k+1)$  oppure  $curr_{run} \leftarrow rank_h^{k+1}(curr_{index})$ 
21:           $symb \leftarrow get\_symbol(start_{k+1}, curr_{run})$ 
22:        else if  $(curr_{run} \neq 0 \wedge curr_{run} = curr_{thr} \wedge \neg down) \vee curr_{run} = |samples_{beg}^k| - 1$  then
23:           $curr_{index} \leftarrow p_k[curr_{run}] - 1$  oppure  $curr_{index} \leftarrow select_h^k(curr_{run})$ 
24:           $curr_{row} \leftarrow samples_{end}^k[curr_{run} - 1]$ 
25:           $ms_{row}[k] \leftarrow curr_{row}$ 
26:          if  $k \neq |z| - 1$  then  $(curr_{index}, curr_{run}, symb) \leftarrow UPDATE(k, curr_{index}, z)$ 
27:        else
28:           $curr_{index} \leftarrow p_k[curr_{run} + 1]$  oppure  $curr_{index} \leftarrow select_h^k(curr_{run} + 1) + 1$ 
29:           $curr_{row} \leftarrow samples_{beg}^k[curr_{run} + 1]$ 
30:           $ms_{row}[k] \leftarrow curr_{row}$ 
31:          if  $k \neq |z| - 1$  then  $(curr_{index}, curr_{run}, symb) \leftarrow UPDATE(k, curr_{index}, z)$ 
32:      for every  $k \in [0, |z|)$  do ▷ Costruzione lunghezze dell'array  $MS$ 
33:        if  $ms_{row}[k] = M$  then
34:           $ms_{len}[k] \leftarrow 0$ 
35:        else if  $k \neq 0 \wedge ms_{row}[i] = ms_{row}[i-1] \wedge ms_{len}[i-1] \neq 0$  then
36:           $ms_{len}[i] \leftarrow ms_{len}[i-1] + 1$ 
37:        else ▷ ra effettua il random access con la componente RA-BV o RA-SLP
38:           $tmp_{index} \leftarrow i$ ,  $tmp_{len} \leftarrow 0$ 
39:          while  $tmp_{index} \geq 0 \wedge z[tmp_{index}] = ra(ms_{row}[k], tmp_{index})$  do
40:             $tmp_{index} \leftarrow tmp_{index} - 1$ ,  $tmp_{len} \leftarrow tmp_{len} + 1$ 
41:           $ms_{len}[k] \leftarrow tmp_{len}$ 
42:      for every  $k \in [0, |z|)$  do ▷ Calcolo dei match da  $MS$ 
43:        if  $(ms_{len}[k] > 1 \wedge ms_{len}[k] \geq ms_{len}[k+1]) \vee (k = |z| - 1 \wedge ms_{len}[k] \neq 0)$  then
44:          report dello SMEM terminante in colonna  $k$ 
45:          SMEM di lunghezza  $ms_{len}[k]$  con la riga  $ms_{row}[k]$  e quelle estese da essa tramite PHI
46: function DOWN( $pos, prev, next$ )
47:   si usano le LCE queries o il random access per calcolare il suffisso comune più lungo tra quelli delle righe
48:    $pos/prev$  e  $pos/next$  fino alla colonna precedente a quella corrente
49:   se il secondo è maggiore o uguale al primo ritorna  $\top$ , altrimenti  $\perp$ 

```

---

---

**Algoritmo 3.18** Algoritmo per l'update con componenti MAP-INT e MAP-BV.

---

```

1: function UPDATE( $k, curr\_index, z$ )
2:    $curr\_index \leftarrow lf(k, curr\_index, z[k])$ 
3:    $curr\_run \leftarrow index\_to\_run(curr\_index, k + 1)$  oppure  $curr\_run \leftarrow rank_h^{k+1}(curr\_index)$ 
4:    $symb \leftarrow get\_symbol(start_{k+1}, curr\_run)$ 
5:   return ( $curr\_index, curr\_run, symb$ )

```

---

- MAP-INT + LCE + PERM + PHI
- MAP-BV + LCE + PERM + PHI

Con la notazione:

$$LCE(k, x, y) \quad (3.45)$$

Si indica il calcolo della *LCE query* tra le righe di indice  $x$  e indice  $y$  terminante in colonna  $k - 1$  (quindi escludendo la colonna  $k$ -esima).

Si illustra ora come computare l'array delle *matching statistics*. Anche in questo caso, per convenzione, si inizia la computazione dell'ultima riga della prima colonna. Si assuma di avere calcolato l'array  $MS$  di una query  $z$  rispetto al pannello  $X$ . le cui righe si identificano tramite  $x_i, \forall i \in \{0, M\}$ , fino alla colonna  $k - 1$ . Sia  $i$  l'indice di riga sulla *matrice PBWT* al quale si è arrivati mediante il mapping, avendo che tale riga è quella che ha il più lungo suffisso comune con  $z[1, k - 1]$ . Si assuma che l'indice  $i$  appartenga alla run  $r$ , di simboli  $\sigma$ , testa di indice  $h$  e coda di indice  $e - 1$ . Si hanno diversi casi:

1.  $z[k] = \sigma$ , quindi la riga  $i$  può essere usata per estendere il match, avendo che  $MS[k].row = MS[k - 1].row$  e  $MS[k].len = MS[k - 1].len + 1$ , e per proseguire col mapping in colonna  $k + 1$
2.  $z[k] \neq \sigma$  e si ha una sola run in colonna  $k$ , avendo quindi che non si possono avere match. Per convenzione, si imposta che  $MS[k].row = M$  e  $MS[k].len = 0$ . Infine si ricomincia, in colonna  $k + 1$ , dall'ultima posizione, indicizzata nel pannello originale dal valore finale del *prefix array sample* dell'ultima run
3.  $z[k] \neq \sigma$  ma si hanno anche altre run, dovendo quindi scegliere la nuova riga da seguire. Si ha che il più lungo suffisso di  $z[1, k]$  che è anche suffisso di  $x_1[1, k], \dots, x_m[1, k]$  è uno tra:
  - $x_{a_k[h-1]}$ , se  $h \neq 0$ , ovvero la riga del pannello corrispondente alla fine della run precedente a  $r$  nella *matrice PBWT*, se esistente
  - $x_{a_k[e+1]}$ , se  $e \neq M - 1$ , ovvero la riga del pannello corrispondente all'inizio della run successiva a  $r$  nella *matrice PBWT*, se esistente



Avendo quindi i *prefix array sample* che ci dicono a quale riga nel pannello corrispondano tali valori e conoscendo  $MS[k-1].row$  è possibile calcolare  $LCE(k, MS[k-1].row, a_k[h-1])$  e  $LCE(k, MS[k-1].row, a_k[e+1])$ . A questo punto si sceglie il suffisso comune più lungo tra le due, ovvero il maggiore tra i valori ritornati dalla *LCE query* e si sceglie la riga corrispondente per proseguire. Si ha quindi o  $MS[k].row = a_k[h-1]$  o  $MS[k].row = a_k[e+1]$ . In merito alla lunghezza, assumendo che la lunghezza maggiore delle due *LCE query* sia  $l$ , si ha che:

$$MS[k].len = \min(MS[k-1].len, l) + 1$$

In quanto la LCE query potrebbe restituire un valore più lungo dell'effettivo match con al query  $z$  quindi si sceglie il minimo tra le due lunghezze, ottenendo l'effettiva lunghezza del suffisso comune tra  $z$  e la nuova riga scelta fino a  $k-1$ , e lo si incrementa di uno, contando il match ottenuto in colonna  $k$

DC Sistemare esempio.

**Esempio 29.** Riprendiamo l'esempio 28, visto per il calcolo di  $MS[i].row$ , dopo un mismatch, tramite *threshold*.

Senza usare le *threshold*, nella medesima situazione si dovrebbe calcolare, avendo che  $MS[1].row = 19$  e  $MS[1].len = 2$ :

$$LCE(2, x_{19}, x_{16}) = "01" \implies |LCE(2, x_{19}, x_{16})| = 2$$

$$LCE(2, x_{19}, x_{17}) = "1" \implies |LCE(2, x_{19}, x_{17})| = 1$$

Come verificabile dal pannello presente all'esempio 15.

Si ha quindi che  $MS[2].row = 16$ . Inoltre, sempre per quanto detto sopra:

$$MS[2].len = \min(MS[1].len, 2) + 1 = 2 + 1 = 3$$

Con questa soluzione, il cui pseudocodice è consultabile all'algoritmo 3.19, quindi:

- non si necessita di tenere in memoria le informazioni per le *threshold*
- si permette il calcolo dell'array  $MS$  in una singola scansione del pattern
- non si necessita di memorizzare l'intero array  $MS$  ma solamente quattro variabili relative alla coppia  $(row, len)$  corrente e quella precedente. Per facilità di lettura si è lasciato, nello pseudocodice, l'uso dell'intero array  $MS$

Dal punto di vista della complessità temporale, per il calcolo dell'array  $MS$ , si hanno solo due casistiche possibili, al variare della componente di mapping. Nel caso della componente **MAP-INT**, avendo  $\rho$  numero medio di run per colonna, si ha un tempo proporzionale, dovendo iterare la query, fare il mapping e usare la componente **LCE**, a:

$$\mathcal{O}(N(\log \rho + \log(NM))) \quad (3.46)$$

Mentre nel caso dell'uso della componente **MAP-BV** si ha tempo proporzionale a:

$$\mathcal{O}\left(N\left(\log \frac{M}{\rho} + \log(NM)\right)\right) \quad (3.47)$$

Infine, per il calcolo di tutte le righe del pannello per cui si ha uno  $SMEM$  si può fare riferimento all'equazione 3.44, avendo la medesima situazione.

Si deduce quindi come la struttura composta **MAP-INT** + **LCE** + **PERM** + **PHI** sia, a livello di tempo macchina, la soluzione più vantaggiosa usando la componente **LCE**. Tale soluzione risulta, sempre nel contesto delle strutture basate sulla componente **LCE**, essere anche la soluzione più vantaggiosa in termini di memoria.

Si vedrà, sperimentalmente, nel capitolo 4, il confronto con le altre strutture dati.

**DC** Sistemare pseudocodice per non avere salvato intero  $MS$

**Algoritmo 3.19** Calcolo degli SMEM con aplotipo esterno con componenti MAP-INT/BV (i cui usi diversificati sono segnalati con “oppure”), LCE, PERM e PHI.

---

```

1: function MATCHES_MS_LCE( $z$ )
2:    $ms_{row} \leftarrow [0..0]$ ,  $ms_{len} \leftarrow [0..0]$  ▷ array  $MS$  di lunghezza  $|z|$ 
3:    $curr_{row} \leftarrow rlpbwt[0].samples_{end}[|rlpbwt[0].samples_{end}| - 1]$ ,  $curr_{index} \leftarrow curr_{row}$ 
4:    $curr_{run} \leftarrow index\_to\_run(curr_{index}, 0)$  oppure  $curr_{run} \leftarrow rank_h^0(curr_{index})$ 
5:    $symb \leftarrow get\_symbol(start_0, curr_{run})$  ▷ Costruzione dell'array  $MS$ 
6:   for every  $k \in [0, |z|)$  do
7:     if  $z[k] = symb$  then
8:        $ms_{row}[k] \leftarrow curr_{row}$ 
9:       if  $k = 0$  then  $ms_{len}[k] \leftarrow 1$  else  $ms_{len}[k] \leftarrow ms_{len}[k - 1] + 1$ 
10:      if  $k \neq |z| - 1$  then  $(curr_{index}, curr_{run}, symb) \leftarrow UPDATE(k, curr_{index}, z)$ 
11:    else
12:      if  $|samples_{beg}^k| = 1$  then
13:         $ms_{row}[k] \leftarrow M$ 
14:         $ms_{len}[k] \leftarrow 0$ 
15:        if  $k \neq |z| - 1$  then
16:           $curr_{row} \leftarrow rlpbwt[k + 1].samples_{end}[|rlpbwt[k + 1].samples_{end}| - 1]$ 
17:           $curr_{index} \leftarrow M - 1$ 
18:           $curr_{run} \leftarrow index\_to\_run(curr_{index}, k + 1)$  oppure  $curr_{run} \leftarrow rank_h^{k+1}(curr_{index})$ 
19:           $symb \leftarrow get\_symbol(start_{k+1}, curr_{run})$ 
20:      else
21:        if  $curr_{run} = |samples_{beg}^k| - 1$  then
22:           $curr_{index} \leftarrow p_k[curr_{run} - 1]$  oppure  $curr_{index} \leftarrow select_h^k(curr_{run})$ 
23:           $prev_{row} \leftarrow samples_{end}^k[curr_{run} - 1]$ 
24:           $lce \leftarrow LCE(k, curr_{row}, prev_{row})$ 
25:           $ms_{row}[k] \leftarrow prev_{row}$ ,  $curr_{row} \leftarrow prev_{row}$ 
26:          if  $k = 0$  then  $ms_{len}[k] \leftarrow 1$  else  $ms_{len}[k] \leftarrow \min(ms_{len}[k - 1], |lce|) + 1$ 
27:          if  $k \neq |z| - 1$  then  $(curr_{index}, curr_{run}, symb) \leftarrow UPDATE(k, curr_{index}, z)$ 
28:        else if  $curr_{run} = 0$  then
29:           $curr_{index} \leftarrow p_k[curr_{run} + 1]$  oppure  $curr_{index} \leftarrow select_h^k(curr_{run} + 1) + 1$ 
30:           $next_{row} \leftarrow samples_{beg}^k[curr_{run} + 1]$ 
31:           $lce \leftarrow LCE(k, curr_{row}, next_{row})$ 
32:           $ms_{row}[k] \leftarrow next_{row}$ ,  $curr_{row} \leftarrow next_{row}$ 
33:          if  $k = 0$  then  $ms_{len}[k] \leftarrow 1$  else  $ms_{len}[k] \leftarrow \min(ms_{len}[k - 1], |lce|) + 1$ 
34:          if  $k \neq |z| - 1$  then  $(curr_{index}, curr_{run}, symb) \leftarrow UPDATE(k, curr_{index}, z)$ 
35:        else
36:           $prev_{row} \leftarrow samples_{end}^k[curr_{run} - 1]$ ,  $next_{row} \leftarrow samples_{beg}^k[curr_{run} + 1]$ 
37:           $lce \leftarrow \max(|LCE(k, curr_{row}, prev_{row})|, |LCE(k, curr_{row}, next_{row})|)$ 
38:           $curr_{row} \leftarrow lce_{row}$  ▷  $lce_{row}$  segnala l'indice della riga con  $LCE$  query più lunga
39:           $ms_{row}[k] \leftarrow curr_{row}$ 
40:          if  $k = 0$  then  $ms_{len}[k] \leftarrow 1$  else  $ms_{len}[k] \leftarrow \min(ms_{len}[k - 1], |lce|) + 1$ 
41:          if  $k \neq |z| - 1$  then  $(curr_{index}, curr_{run}, symb) \leftarrow UPDATE(k, curr_{index}, z)$ 
42:      for every  $k \in [0, |z|)$  do ▷ Calcolo dei match da  $MS$ 
43:      if  $(ms_{len}[k] > 1 \wedge ms_{len}[k] \geq ms_{len}[k + 1]) \vee (k = |z| - 1 \wedge ms_{len}[k] \neq 0)$  then
44:        report degli SMEM di lunghezza  $ms_{len}[k]$ , terminanti in colonna  $k$ 
45:        con la riga  $ms_{row}[k]$  e quelle estese da essa tramite la componente PHI

```

---

# Capitolo 4

## Risultati sperimentali

Verranno ora riportati alcuni risultati sperimentali relativi all'implementazione della **RLPBWT**. In primis, verranno discusse le modalità di sperimentazione per poi trattare i risultati ottenuti su alcuni pannelli della *phase 3* del **1000 Genome Project (1KGP)** [36], progetto, che ha avuto inizio nel 2008, il quale ha visto lo sforzo della comunità scientifica internazionale per catalogazione delle variazioni geniche umane. Il *1KGP* risulta essere uno dei più importanti progetti del settore. Verranno quindi confrontate le implementazioni degli algoritmi di calcolo degli *SMEM* della *PBWT* di Durbin e delle varianti per la *RLPBWT*.

DC Magari servono dettagli per il 1kgp

**RLPBWT.** In merito alle varianti della *RLPBWT*, sono state testate le otto varianti discusse nel capitolo 3:

1. la struttura dati composta **MAP-INT + LCP** e la struttura dati composta **MAP-BV + LCP**. Si ricorda che tali soluzioni non supportano il riconoscimento delle righe del pannello per cui si ha uno *SMEM* ma solo la cardinalità dell'insieme di tali righe
2. le strutture dati composte basate sull'uso delle *threshold* per il calcolo dell'array delle *matching statistics*, ovvero: **MAP-INT + THR-INT + RA-BV + PERM + PHI**, **MAP-INT + THR-INT + RA-SLP + PERM + PHI**, **MAP-BV + THR-BV + RA-BV + PERM + PHI** e **MAP-BV + THR-BV + RA-SLP + PERM + PHI**
3. le strutture dati composte basate sull'uso delle *LCE query* per il calcolo dell'array delle *matching statistics*, ovvero: **MAP-INT + LCE + PERM + PHI** e **MAP-BV + LCE + PERM + PHI**

L'implementazione è stata fatta in linguaggio C++, usando le già citate librerie esterne:

- **SDSL-lite** per *int vector compressi*, *bitvector*, *bitvector sparsi*, serializzazione e varie utility per il calcolo della memoria
- **BigRePair** e **ShapedSlp** per la costruzione e l'uso degli *SLP*

L'implementazione delle strutture composte per la *RLPBWT* supportano lo studio parallelo di più query tramite **openMP**[37]. Al fine di un più corretto confronto con l'implementazione della *PBWT*, specialmente parlando dell'algoritmo **matchIndexed**, l'intera sperimentazione è stata svolta sfruttando un solo *thread* per volta, tramite la variabile d'ambiente `OMP_NUM_THREADS=1`.

**PBWT.** Al fine di validare più correttamente i confronti tra le varie strutture dati per la *RLPBWT* e la *PBWT* di Durbin, si è scelto di utilizzare l'implementazione originale di quest'ultima<sup>1</sup>, scritta in linguaggio C. L'implementazione fornisce tre algoritmi per il calcolo degli SMEM, avendo  $N$  siti,  $M$  sample e  $Q$  query:

1. **matchNaive**, ovvero un'implementazione naïve del calcolo degli SMEM che non sfrutta la *PBWT*. Questo algoritmo non è utilizzabile in casi reali. La complessità in tempo di tale soluzione è stimata essere  $\mathcal{O}(NMQ)$  mentre quella in spazio è  $\mathcal{O}(NM)$
2. **matchIndexed**, ovvero l'algoritmo 5 del paper originale [4]. La complessità in tempo di tale soluzione è stimata essere  $\mathcal{O}(NQ)$  dopo una fase di preprocessing con complessità  $\mathcal{O}(NM)$ . La complessità in spazio è stimata essere  $\mathcal{O}(NM)$ , ricordando come, in pratica, essa corrisponda a  $13NM$  bytes in memoria
3. **matchDynamic**, ovvero un algoritmo non approfondito nel paper ma solo citato nei risultati sotto il nome di "batch". Pur mancando una descrizione approfondita dell'algoritmo si è dedotto che il suo funzionamento si basi sul creare una trasformata *PBWT* anche delle query, viste sotto forma di pannello, e applicare l'algoritmo per il calcolo degli SMEM interni alla *PBWT* stessa. Inoltre il calcolo dei vari indici viene fatto di colonna in colonna. La complessità in tempo di tale soluzione è stimata essere  $\mathcal{O}(N(M + Q))$  mentre quella in spazio è  $\mathcal{O}(N + M)$ .

Si intuisce fin da subito come l'ultima soluzione, non approfondita nel paper di riferimento e della quale si è avuto riconoscimento solo in fase di sperimentazione, risulti essere la migliore a disposizione. Si hanno solo due piccole limitazioni. La prima è dovuta al fatto che, dovendo di fatto computare la trasformata anche

---

<sup>1</sup><https://github.com/richarddurbin/pbwt>

per il pannello di query ed essendo l'algoritmo studiato per lavorare sulla trasformata stessa, i tempi di calcolo per poche query sono alti rispetto all'algoritmo `matchIndexed` e rispetto alle varie soluzioni per la *RLPBWT*. Il secondo limite è che i risultati non sono ordinati. Tutti gli altri algoritmi presentano i risultati query per query, in ordine, mentre l'algoritmo `matchDynamic`, studiando la trasformata, presenta tutti i risultati permutati secondo la stessa. Si rileva, in ogni caso, come tali limiti possano essere per lo più trascurati, nonostante il problema su cui si concentrano gli studi di questa tesi fosse la ricerca degli SMEM tra una singola query e un pannello di aplotipi.

## 4.1 Pannelli del 1000 Genome Project

Come anticipato, al fine di valorizzare i risultati teorici ottenuti in questo progetto, si è deciso di procedere con lo studio di dati reali, relativi alla *phase 3* del **1000 Genome Project** <sup>2</sup> [36].

Tali pannelli, disponibili in formato *VCF*, presentano un numero costante di sample, ovvero 5.008, mentre a variare è il numero di siti. Essendo dati reali, si ha anche la presenza di siti multiallelici. Si è quindi proceduto alla selezione dei soli siti biallelici, ottenendo quindi pannelli costruiti su un alfabeto binario  $\Sigma = \{0, 1\}$ , tramite l'uso di `bcftools` [38], tramite il comando `bcftools view -m2 -M2 -v snps`.

La prima selezione dei pannelli è stata dettata dalla volontà di studiare, per praticità, pannelli non troppo estesi. Si sono, quindi, scelti i pannelli relativi ai cromosomi 22 (*chr22*), 20 (*chr20*), 18 (*chr18*), 16 (*chr16*) e 1 (*chr1*). Si noti che l'ordine è dato dal numero crescente di siti. La scelta di includere il cromosoma 1 è dettata dal fatto che risulta essere il più grande cromosoma umano, implicando quindi che anche il relativo pannello delle variazioni geniche sia tra quelli col maggior numero di siti.

Trattandosi di pannelli reali, è risultata interessante una preliminare indagine esplorativa sulla natura di tali pannelli in termini di *sparsità* degli alleli e di conseguente numerosità delle run. Si è quindi calcolato, per i cinque pannelli, il numero di simboli  $\sigma = 0$  e  $\sigma = 1$ , notando come il numero di simboli  $\sigma = 1$  fosse molto ridotto rispetto al totale, essendo il  $\sim 0.03\%$  del totale dei simboli in tutti e cinque i casi. Una tale sparsità del dato ha diretta conseguenza sul numero di run di ogni colonna, avendo pochi simboli  $\sigma = 1$  in ogni colonna, simboli che possono anche essere nella medesima run dopo la permutazione data dalla *PBWT*. Si ricordi, inoltre, che tale permutazione, come la *BWT*, è studiata per essere maggiormente efficiente nel caso del dato biologico, comportando un'alta probabilità di produrre run del medesimo carattere. In tabella 4.1, quindi, si riportano il numero di siti

<sup>2</sup><https://ftp.1000genomes.ebi.ac.uk/vol1/ftp/release/20130502/>

Tabella 4.1: Informazioni quantitative relative ai cinque pannelli in analisi.

Chr	#Siti	#Run totale	Max run	Media run
Chr22	1.055.454	14.772.105	2.450	14
Chr20	1.739.315	19.966.504	2.176	11
Chr18	2.171.378	24.288.263	2.365	11
Chr16	2.596.072	31.187.856	2.330	12
Chr1	6.196.151	69.671.952	2.721	11

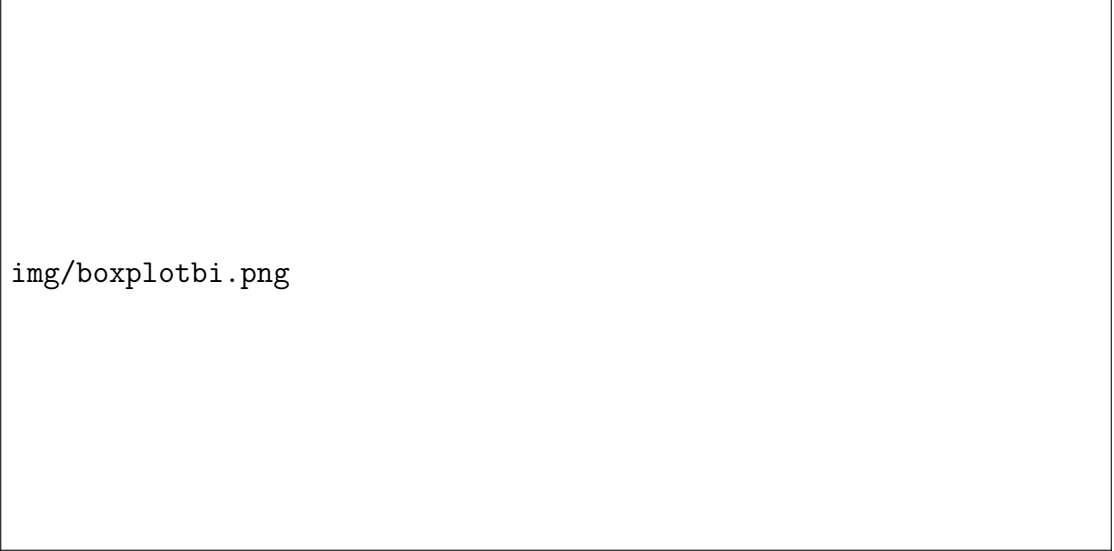
di ogni cromosoma, il numero medio di run per colonna, il numero massimo di run in una colonna e il totale delle run. Si segnala inoltre come la mediana abbia valore 3 per tutti e tre i pannelli. I valori quantitativi sono stati calcolati a partire dai pannelli con un numero di sample pari a 4908 in quanto, si anticipa, 100 righe sono state utilizzate come query nelle successive fasi della sperimentazione. Si conferma quindi il risultato atteso, in merito alla sparsità del dato e al conseguente basso numero medio di run per colonna, risultato che è a favore, in termini di complessità in spazio, della *RLPBWT* in quanto tutte le componenti sono proporzionali al numero, sperimentalmente basso, di run. In figura 4.1 si riportano i risultati statistici, sotto forma di *boxplot*, relativi alla distribuzione delle run nei cinque pannelli studiati. Il forte numero di outlier è dovuto al fatto che media e soprattutto mediana del numero di run per colonna risultano essere molto piccoli rispetto al numero massimo di run riscontrabili in una colonna.

### 4.1.1 Riproducibilità degli esperimenti

Al fine di rendere riproducibili gli esperimenti, si è costruita una pipeline per l'esecuzione dei vari algoritmi e l'estrazione dei dati quantitativi relativi alle performance.

L'intera pipeline è stata gestita tramite **Snakemake** [39], un *workflow management system*, uno strumento molto usato in *bioinformatica* per creare analisi dati scalabili e riproducibili. Nel dettaglio la pipeline comprende, come visualizzabile in figura 4.2, avendo in input un pannello ed una lista di quantità di apotipi di query:

- scaricamento dei tool e delle loro dipendenze per la *PBWT* di Durbin e la *RLPBWT* proposta in questa tesi
- produzione dell'input per la *PBWT* e della *RLPBWT* per ogni quantità di query richiesta
- produzione delle strutture dati



img/boxplotbi.png

Figura 4.1: Boxplot della distribuzione delle run per i pannelli dei cinque cromosomi studiati. Il grafico (a) presenta uno zoom che esclude la maggior parte degli outlier mentre il grafico (b) presenta, in scala logaritmica, il boxplot completo con tutti gli outlier.

- esecuzione degli algoritmi per il calcolo degli *SMEM*

Al fine di ottenere risultati non banali, si è deciso di partire da un pannello iniziale fisso ed estrarre un numero di righe pari al numero di query richieste, righe che, a loro volta, andranno a formare il pannello di query.

*La sperimentazione è stata effettuata su una macchina con processore Intel Xeon E5-2640 V4 (2.40 GHz), 756 GB di RAM, 768 GB di swap e sistema operativo Ubuntu 20.04.4 LTS. Tale macchina è stata gentilmente messa a disposizione dalla University of Florida.*

## 4.2 Risultati della sperimentazione

Si presentano ora i risultati quantitativi della sperimentazione effettuata sui cinque pannelli scelti tra quelli ottenuti nella *phase 3* del *1000 Genome Project*. Come anticipato, al fine del computo degli *SMEM*, avendo un numero ridotto di sample a disposizione, si è scelto di estrarre da ognuno 100 sample da usare come query, riducendo quindi il numero di sample a 4.908.



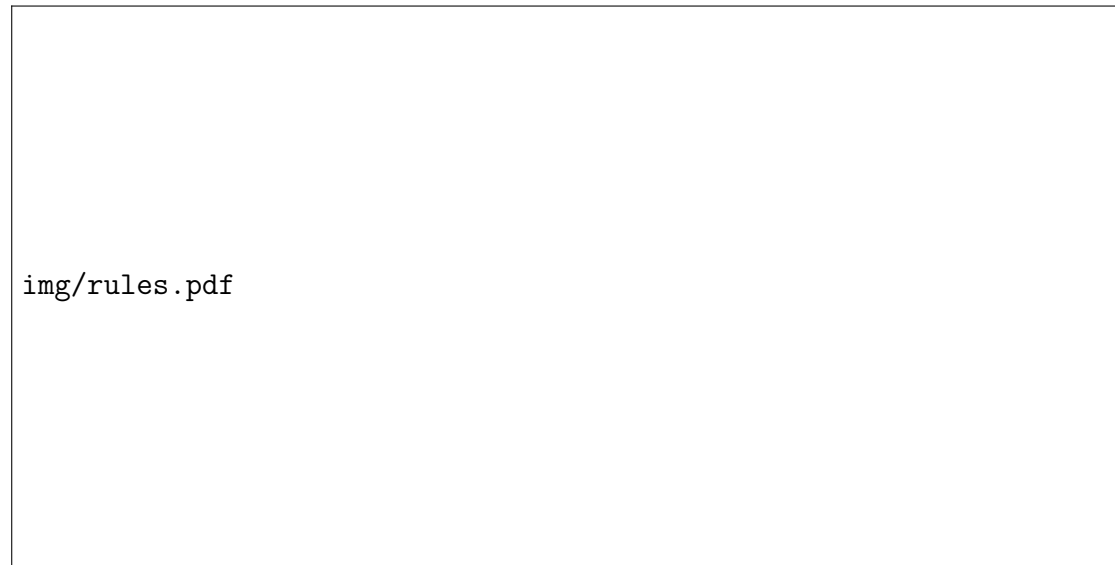


Figura 4.2: Regole usate in Snakemake per la sperimentazione.

### 4.2.1 Costruzione delle strutture e calcolo degli SMEM

Viste le dimensioni di tali pannelli si ritiene necessario studiare, dal punto di vista del tempo macchina e dei picchi di memoria necessaria, le varie fasi della sperimentazione, ovvero:

- la fase di *preprocessing*, necessaria per la preparazione dei vari input della *RLPBWT*, comprendente:
  - la conversione dei file in formato VCF nei file in formato MACs
  - l'estrazione del pannello delle query e la creazione del nuovo pannello di aplotipi reference
  - la produzione dell'*SLP* del pannello di reference, comprendente sia la produzione della stringa unica monodimensionale che l'esecuzione di *BigRepair* e *ShapedSlp*
- la costruzione e serializzazione delle varie strutture dati composte per la *RLPBWT* e dei file ad hoc per la *PBWT*
- il calcolo degli *SMEM*

**Preprocessing.** In figura 4.3 si possono analizzare le prestazioni delle tre fasi di preprocessing. I risultati quantitativi sono consultabili alla tabella 4.2 e alla tabella 4.3. La separazione del pannello con le query risulta essere assolutamente

Tabella 4.2: Risultati quantitativi dei picchi di memoria (gigabytes) relativi alle fasi di preprocessing per l'input delle varianti della *RLPBWT*

Chr	Costruzione SLP	Conversione VCF a MACs	Estrazione query
chr22	111	5	0,0089
chr20	183	9	0,0090
chr18	215	11	0,0090
chr16	272	13	0,0090
chr1	644	31	0,0094

Tabella 4.3: Risultati quantitativi dei tempi (secondi) relativi alle fasi di preprocessing per l'input delle varianti della *RLPBWT*

Chr	Costruzione SLP	Conversione VCF a MACs	Estrazione query
chr22	2542	207	28
chr20	3950	320	47
chr18	4890	405	64
chr16	6104	500	71
chr1	14430	1098	157

ininfluente e, di fatto, anche la conversione tra i due formati non necessita particolari considerazioni, come inferibile dai risultati numerici delle performance. Tale conversione diventerebbe non necessaria implementando l'input direttamente da file VCF per le varie strutture dati relative alla *RLPBWT*. Inoltre, in un contesto reale, la costruzione del pannello di query non sarebbe un'operazione necessaria. Bisogna, però, analizzare la costruzione dell'*SLP*. Per quanto quest'operazione sia da svolgersi *una tantum*, le richieste in termini di memoria sono nell'ordine delle centinaia di gigabytes di RAM mentre i tempi di calcolo sono nell'ordine delle ore. Prendendo in analisi il cromosoma 1 si ha infatti che è richiesto un picco di 644GB di RAM, avendo che l'intera esecuzione richiede circa 4 ore. D'altro canto, bisogna considerare che tutti gli strumenti computazionali per la produzione dell'*SLP* sono studiati per partire da una singola stringa e non da una matrice. Nuovi sviluppi in questa direzione potrebbero lasciar spazio a diverse ottimizzazioni. Bisogna, infine, considerare che questa fase è necessaria solo per quattro delle sei soluzioni studiate per la *RLPBWT* basate sul calcolo delle *matching statistics* e, come già detto, il fatto che sia necessaria solo una volta è cruciale nell'ottica di un confronto con, ad esempio, lo spazio richiesto dall'*algoritmo 5* di Durbin, che richiede  $13NM$  bytes per il calcolo degli *SMEM* ad ogni esecuzione.

In generale, la crescita di memoria richiesta e di tempi per le tre fasi di preprocessing sembrano essere proporzionali al numero di siti dei pannelli, come ci si potrebbe attendere. In figura 4.4 si può osservare il vantaggio in termini di

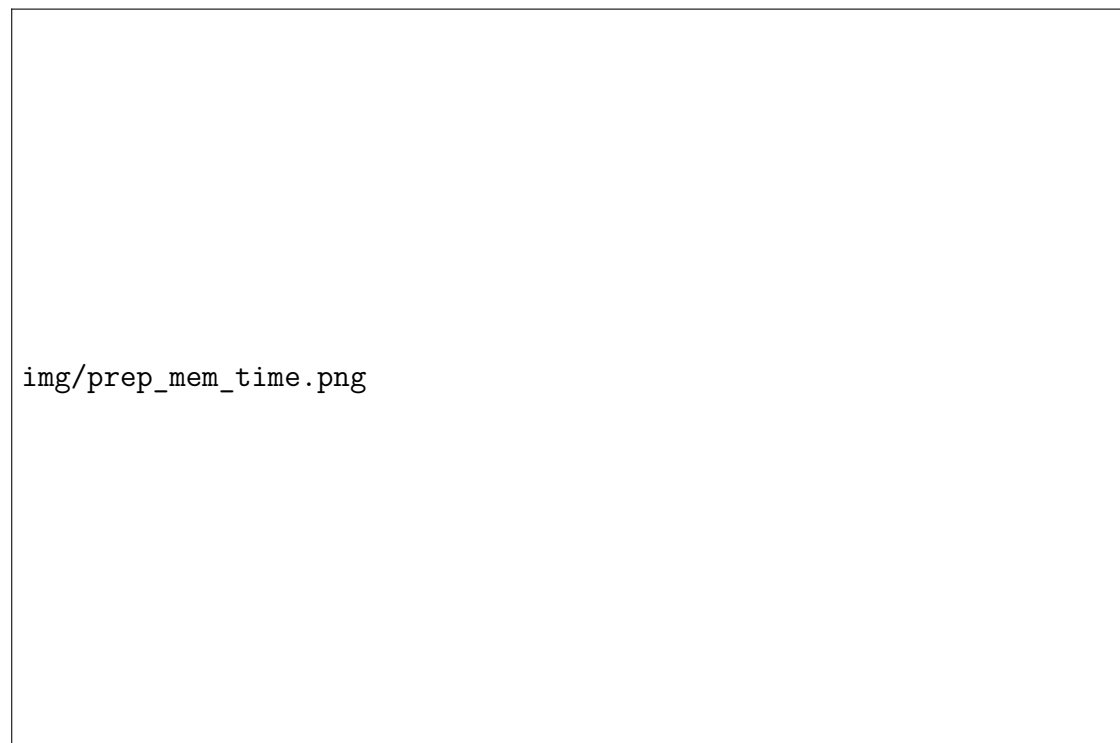
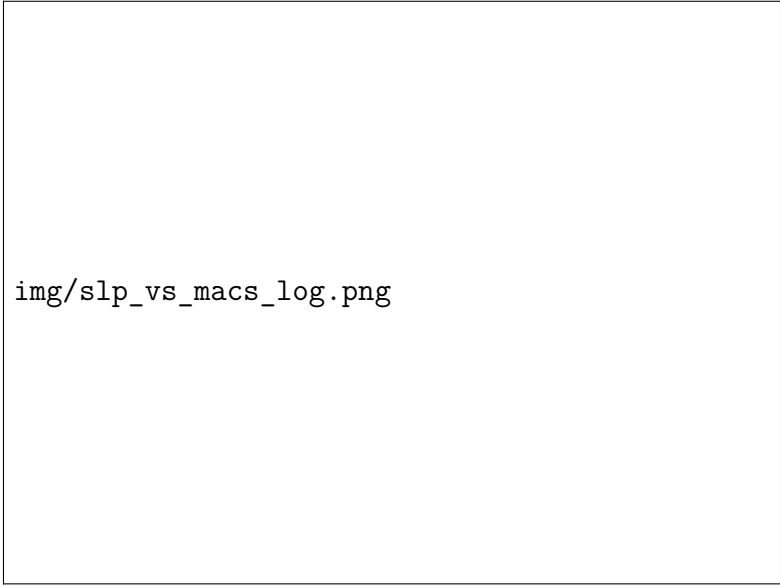


Figura 4.3: Picchi di memoria (a) e tempo richiesto (b) per le tre fasi di preprocessing dell'input per la *RLPBWT*, in scala logaritmica.

Tabella 4.4: Dimensioni, in gigabytes, degli *SLP* e dei file *MACs* per i vari pannelli del 1000 Genome Project.

Chr	SLP	MACs	%
chr22	0,04	4,84	0,9
chr20	0,06	7,98	0,76
chr18	0,08	9,97	0,79
chr16	0,1	11,91	0,81
chr1	0,22	28,44	0,78



img/slp\_vs\_macs\_log.png

Figura 4.4: Confronto tra la memoria richiesta dai file *MACs* e dagli *SLP* per i pannelli del 1000 Genome Project, in scala logaritmica.

memoria che si ha con l'uso degli *SLP*, confrontando il peso dei file *MACs* con il peso delle grammatiche compresse. Si segnala, in ogni caso, che il peso dei vari file *MACs* include anche i diversi header. In tabella 4.4 si possono confrontare quantitativamente tali risultati. È possibile, grazie a tali risultati, apprezzare la compressione di tali grammatiche. Si noti che, essendo la capacità di compressione di un *SLP* direttamente correlata alle ripetizioni presenti nella stringa da comprimere, la dimensione dell'*SLP* non è perfettamente proporzionale al numero di siti dei pannelli in analisi.

**Costruzione della struttura.** Passiamo ora ad analizzare tempi e picchi di memoria per la costruzione delle strutture dati. Bisogna ricordare che:

- nel caso della *RLPBWT*, per ognuna delle strutture dati composte, questa fase prevede la costruzione e la serializzazione dell'intera struttura dati
- nel caso della *PBWT* questa fase crea unicamente un file compresso ad hoc, contenente le strutture base delle *PBWT*. A partire da tale file, in fase di calcolo degli *SMEM*, verranno calcolati anche tutti gli altri indici al calcolo degli stessi, a seconda dell'algoritmo usato

Fatte queste doverose premesse, passiamo ad analizzare i risultati. In figura 4.5 (a) vengono visualizzati i picchi di memoria richiesti mentre in figura 4.5 (b) i tempi di calcolo delle strutture. Alle tabelle 4.5 e 4.6 si riportano i risultati quantitativi in termini di gigabytes e secondi.

Come anticipato, l'implementazione della *PBWT* non calcola e memorizza tutti gli indici necessari al calcolo degli *SMEM* in fase di costruzione, avendo quindi una bassissima richiesta di memoria in questa fase. Discorso diverso si ha parlando delle strutture dati per la *RLPBWT*. Le strutture dati composte **MAP-INT + LCP** e **MAP-BV + LCP**, dovendo memorizzare l'intero insieme degli *array LCP*, hanno un elevato consumo di memoria. Pur utilizzando degli *int vector compressi*, in modo analogo a quanto visto, ad esempio, per la componente **PERM**, si ha necessità di salvare *NM* valori interi, valori che, nel caso peggiore, sono pari al numero di siti del pannello. Risulta ovvio notare come, in termini di costruzione, siano le due strutture che non scalano sul numero di run ad aver maggiori richieste di memoria. Proseguendo nell'analisi si ha che l'utilizzo della componente **MAP-BV**, come atteso, richiede maggior memoria (approssimativamente tra il 15% e il 20% in più) della componente **MP-INT**. In merito invece ai tempi di costruzione delle due strutture si segnala come i tempi di calcolo della componente **MAP-BV** siano superiori rispetto a quelli della **MAP-INT** (anche in questo caso approssimativamente tra il 15% e il 20% in più), dovendo, ad esempio, calcolare anche le strutture per le funzioni *rank/select* per i *bitvector sparsi*. Nel caso della componente **MAP-INT**, invece, l'unica operazione aggiuntiva rispetto a quelle attese per una classica popolazione di array di interi è la fase di compressione degli stessi. Le analisi effettuate sulle componenti di *mapping* sono da ritenersi analoghe per le componenti dedicate alle *threshold*, in termini di uso di *bitvector sparsi* e *int vector compressi*. Tali considerazioni sono valide anche per le altre strutture composte per la *RLPBWT*. Infine, confrontando le strutture dati in grado di computare l'array *MS*, si aggiungono le considerazioni sulle possibili strutture per il *random access* e, nel caso dell'*SLP*, per l'uso delle *LCE query*. Si nota come l'uso della componente **RA-BV** comporta, come atteso, una maggior richiesta di memoria, pur limitata dall'uso dei *bitvector*. Sempre in termini di tempi di esecuzione, si ha che la componente **RA-BV** deve essere computata in fase di costruzione delle strutture, comportando un aumento dei tempi di calcolo (meno del 5% in più). Confrontando i tempi di tutte le varianti si ha che tutti gli algoritmi di costruzione sono in tempo proporzionale

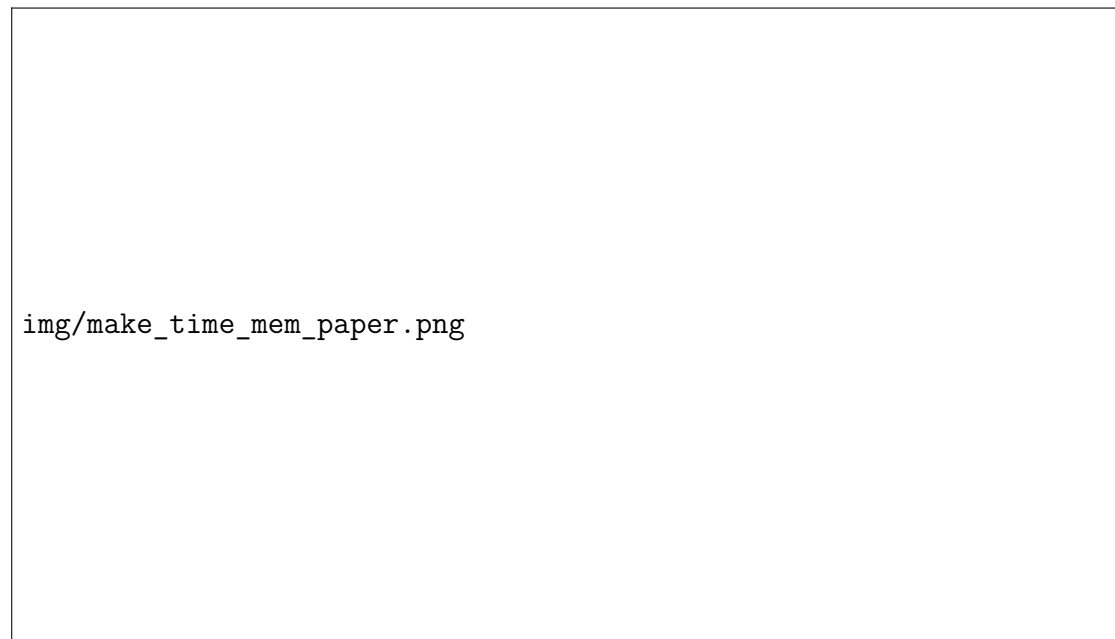


Figura 4.5: Picchi di memoria (a) e tempi di calcolo (b) per la costruzione delle varianti della RLPBWT e per la PBWT.

a  $\mathcal{O}(NM)$  ma, come detto, le varianti della *RLPBWT* includono, in questa fase, anche il calcolo delle strutture utili al calcolo degli SMEM. Si noti come, parlando di *RLPBWT*, la struttura composta **MAP-INT + LCE + PERM + PHI** risulti essere la meno costosa in termini di memoria, usando la struttura di mapping tramite *int vector compressi* e l'*SLP* (per le *LCE query*). Anche in termini di tempo, per i discorsi fatti sui tempi di calcolo delle singole componenti, risulta essere la soluzione più vantaggiosa.

In generale, si può concludere che questa frase conferma quanto discusso nel capitolo 3.

Sfruttando i metodi offerti da *SDSL-LITE*, è possibile studiare l'occupazione di memoria delle singole componenti trattate nel capitolo 3. In tabella 4.8 e in figura 4.6 si riportano, in megabytes, le dimensioni di tali componenti. Si può, innanzitutto, apprezzare il vantaggio dell'uso della componente **RA-SLP/LCE** rispetto alla componente **RA-BV**, avendo che si ha un risparmio di memoria superiore al 90%. Numericamente tale vantaggio è riportato in tabella 4.7. Confermando nuovamente quando detto diverse volte nella tesi, si segnala il forte vantaggio in memoria nell'utilizzo delle componenti basate su *int vector compressi*, rispetto che a quelle basate su *sparse bitvector*.

Si nota, infine, come le componenti **PERM** e **PHI**, non presentino particolari criticità dal punto di vista della memoria richiesta. Infine, terminando l'analisi di

DC Controllare  
tutti questi dati

Tabella 4.5: Risultati quantitativi dei picchi di memoria (gigabytes) di costruzione delle strutture dati.

Tabella A: risultati relativi alla *PBWT* e alle varianti basate su LCP per la *RLPBWT*.

Chr	PBWT	MAP-INT+ LCP	MAP-BV+ LCP
chr22	0.1	9	11
chr20	0.2	15	18
chr18	0.2	19	23
chr16	0.3	23	27
chr1	0.6	55	65

Tabella B: risultati relativi alle varianti basate su *matching statistics* e *bitvector sparsi* per la *RLPBWT*.

Chr	MAP/THR-BV + RA-BV ...	MAP/THR-BV + RA-SLP...	MAP-BV + LCE ...
chr22	5	4	4
chr20	8	7	7
chr18	10	9	8
chr16	12	10	10
chr1	28	24	23

Tabella C: risultati relativi alle varianti basate su *matching statistics* e *int vector compressi* per la *RLPBWT*.

Chr	MAP/THR-INT + RA-BV ...	MAP/THR-INT + RA-SLP ...	MAP-INT + LCE ...
chr22	3	2	2
chr20	4	3	3
chr18	5	4	4
chr16	6	5	5
chr1	14	12	11

Tabella 4.6: Risultati quantitativi dei tempi (secondi) di costruzione delle strutture dati.

Tabella A: risultati relativi alla *PBWT* e alle varianti basate su LCP per la *RLPBWT*.

Chr	PBWT	MAP-INT+ LCP	MAP-BV+ LCP
chr22	136	210	250
chr20	233	349	407
chr18	290	417	488
chr16	412	511	613
chr1	792	1145	1393

Tabella B: risultati relativi alle varianti basate su *matching statistics* e *bitvector sparsi* per la *RLPBWT*.

Chr	MAP/THR-BV + RA-BV ...	MAP/THR-BV + RA-SLP ...	MAP-BV + LCE ...
chr22	262	255	240
chr20	439	424	363
chr18	521	503	431
chr16	660	628	533
chr1	1472	1400	1248

Tabella C: risultati relativi alle varianti basate su *matching statistics* e *int vector compressi* per la *RLPBWT*.

Chr	MAP/THR-INT + RA-BV ...	MAP/THR-INT + RA-SLP ...	MAP-INT + LCE ...
chr22	143	137	115
chr20	232	219	192
chr18	288	275	238
chr16	347	330	285
chr1	778	739	652

Tabella 4.7: Vantaggio percentuale dell'uso delle componenti RA-SLP/LCE rispetto alla componente RA-BV.

Chr	RA-SLP/LCE (MB)	RA-BV (MB)	%
chr22	44	628	7
chr20	61	1.035	6
chr18	80	1.292	6
chr16	98	1.544	6
chr1	226	3.687	6



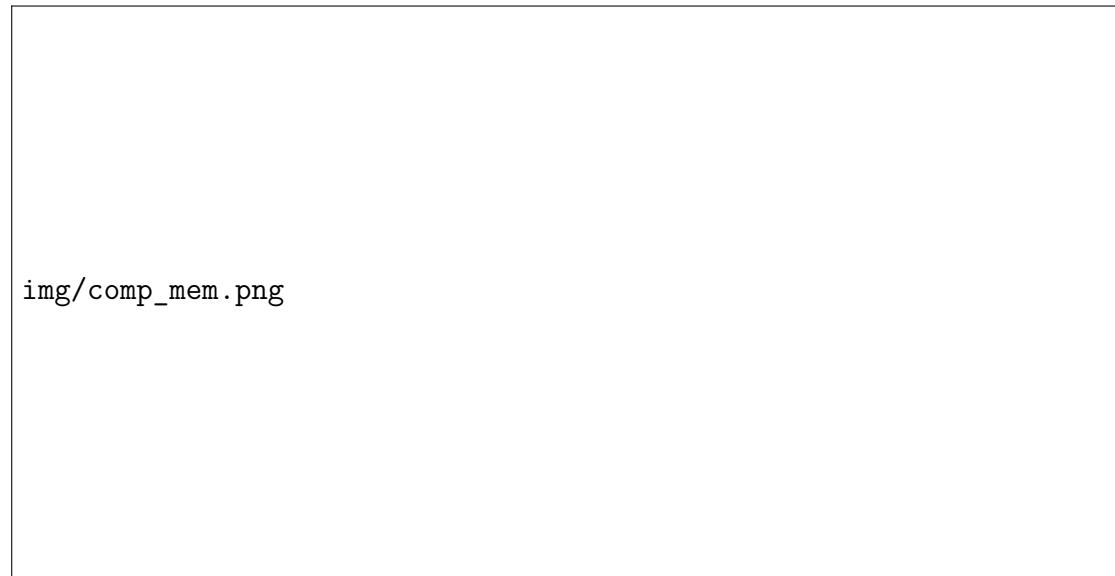


Figura 4.6: Memoria occupata dalle singole componenti, avendo sulle ascisse in (a) il numero di siti e in (b) il numero di run.

tali risultati, senza trattare nuovamente le componenti per il *random access*, si ha conferma della richiesta eccessiva in memoria della componente LCP.

DC Serve altro?

**Calcolo degli SMEM.** In seguito si ha la discussione dei risultati ottenuti per il calcolo degli *SMEM*.

In figura 4.7 (a) si riportano i risultati i termini di picchi di memoria durante la computazione degli *SMEM*. Tali risultati sono consultabili numericamente in tabella 4.9. Come previsto, l'algoritmo *matchDynamic* della *PBWT* ha le migliori prestazioni in spazio, calcolando dinamicamente i vari indici necessari al calcolo degli *SMEM* "interni". D'altro canto per quanto riguarda l'*algoritmo 5* di Durbin, ovvero l'algoritmo *matchIndexed*, si confermano le previsioni fatte dall'autore stesso, avendo che la memoria utilizzata è circa  $13NM$  bytes. Escludendo le strutture *MAP-INT + LCP* e *MAP-BV + LCP*, si parla di circa un'intero ordine di grandezza in più di memoria rispetto alle strutture dati composte per la *RLPBWT*. Parlando di queste ultime, la differenza tra le varie strutture dati che supportano il calcolo dell'array *MS* è dovuta, a parità di componente per il mapping (e conseguentemente della componente per le threshold), dall'uso della componente *RA-BV* o della componente *RA-SLP* (o della componente *LCE*), in modo analogo a quanto visto discutendo il peso in memoria delle singole componenti.

In figura 4.7 (b) si riportano, invece, i risultati i termini di tempo di calcolo. Tali risultati sono consultabili numericamente in tabella 4.10. Anche in questo caso

Tabella 4.8: Dimensioni, in megabytes, delle singole componenti per la *RLPBWT*.Tabella A: dimensioni delle componenti di *mapping* e *threshold* usate nelle strutture dati per la *RLPBWT*.

Chr	MAP-INT	MAP-BV	THR-INT	THR-BV
22	74	543	36	199
20	109	882	53	322
18	137	1.100	61	402
16	167	1.320	81	483
1	384	3.133	185	1.146

Tabella B: dimensioni delle componenti di *random access*, *prefix array samples* e struttura  $\phi$ .

Chr	RA-BV	RA-SLP/LCE	PERM	PHI	LCP
22	628	44	71	89	9.095
20	1.035	62	104	118	15.468
18	1.292	80	128	147	19.223
16	1.545	99	159	186	22.888
1	3.687	227	366	411	54.588

l'algoritmo `matchDynamic` risulta essere il più performante, in quanto studia contemporaneamente l'intero pannello di query. Parlando di *RLPBWT*, la struttura MAP-INT + THR-INT + RA-SLP + PERM + PHI e la struttura MAP-BV + THR-BV + RA-SLP + PERM + PHI, a causa delle frequenti operazioni di *random access* con la componente RA-SLP, sia per il calcolo delle lunghezze delle matching statistics che per la fase di "disambiguazione", richiede più tempo di tutte le altre varianti, soprattutto se si pensa alla corrispondente variante con componente RA-BV. L'uso della componente RA-SLP comporta, infatti, circa venti volte i tempi di calcolo usando i *bitvector sparsi* e addirittura di circa 45 volte usando gli *int vector compressi*. La struttura MAP-INT + LCE + PERM + PHI e la struttura MAP-BV + LCE + PERM + PHI risultano essere, al massimo, il doppio più lenta rispetto all'algoritmo `matchIndexed`, con una differenza che diventa quasi trascurabile all'aumentare delle dimensioni del pannello. Questo è un risultato molto interessante se si tiene in considerazione la memoria necessaria per il calcolo degli *SMEM*.

Confrontando l'uso delle *LCE query* con l'uso della componente RA-BV si hanno, invece, tempi triplicati nel caso di *mapping* e *threshold* con *bitvector sparsi*. Nel caso degli *int vector compressi* si hanno, d'altro canto, circa sette volte i tempi di computazione. A priori delle componenti per *mapping* e *threshold*, l'uso della componente RA-SLP comporta circa sei volte il tempo d'uso delle *LCE query*. Queste ultime analisi comportano che l'uso del *random access* su *SLP* è la peggior

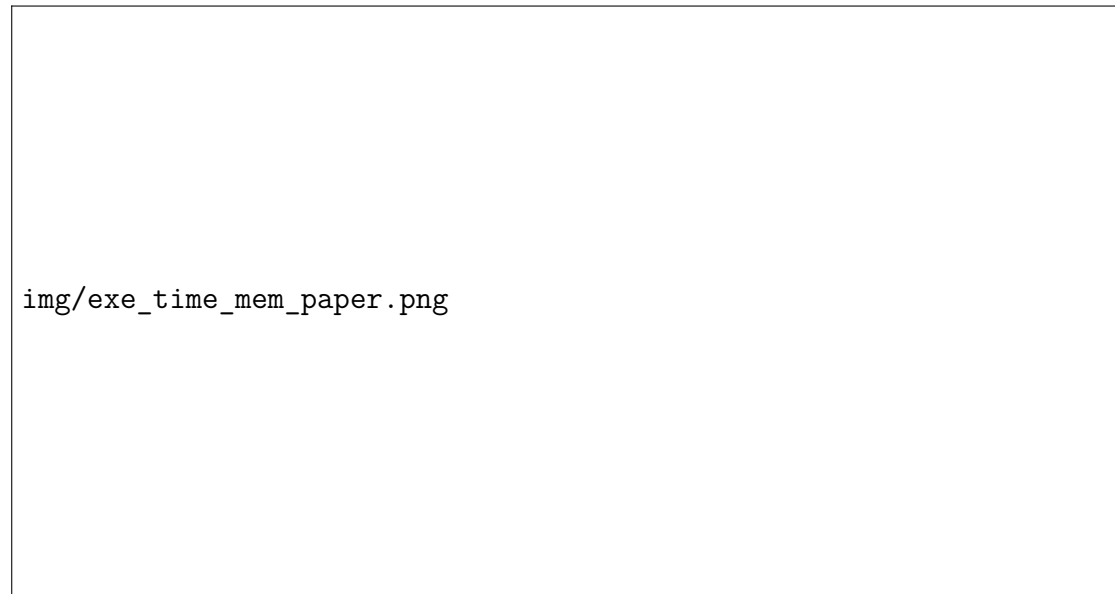


Figura 4.7: Picchi di memoria (a) e tempi di esecuzione (b) per il calcolo degli SMEM.

soluzione in ottica calcolo delle *matching statistics*.

Concludendo, si può notare come la struttura composta MAP-INT + THR-INT + RA-BV + PERM + PHI, tra quelle per la *RLPBWT*, risulti essere la migliore in termini di tempi di calcolo mentre la struttura composta MAP-INT + LCE + PERM + PHI sia la migliore in termini di memoria richiesta. Notando come quest'ultima sia circa 10 volte più lenta si può inferire come la scelta della miglior soluzione per la *RLPBWT* debba ricadere sulla MAP-INT + THR-INT + RA-BV + PERM + PHI, salvo situazioni in il risparmio di memoria sia un limite fondamentale in fase di analisi dati.

DC Controllare  
tutti questi dati

### 4.2.2 Tempo di una singola query

Infine, per completare lo studio delle prestazioni temporali, si è deciso di isolare il calcolo degli SMEM con ogni singola query, valutando media e deviazione standard delle 100 query. A tal fine la misurazione è stata effettuata sfruttando la libreria `time.h` presente nello standard del linguaggio C, al fine di avere le medesime misurazioni sia con la *PBWT* che con la *RLPBWT*. La misurazione è stata fatta misurando unicamente le istruzioni atte a cercare gli *SMEM*, escludendo quelle per il computo degli indici o del caricamento delle strutture dati. Si segnala che nel caso dell'algoritmo `matchDynamic` non si è potuto, per natura stessa dell'algoritmo, isolare il computo degli indici all'avanzamento alla colonna successiva. Resta

Tabella 4.9: Risultati quantitativi dei picchi di memoria (gigabytes) di costruzione delle strutture dati.

Tabella A: risultati relativi alla *PBWT* e alle varianti basate su LCP per la *RLPBWT*.

Chr	matchIndexed	matchDynamic	MAP-INT+ LCP	MAP-BV+ LCP
chr22	63	0.02	9	11
chr20	104	0.03	16	19
chr18	129	0.04	20	23
chr16	155	0.05	24	28
chr1	369	0.1	56	66

Tabella B: risultati relativi alle varianti basate su *matching statistics* e *bitvector sparsi* per la *RLPBWT*.

Chr	MAP/THR-BV + RA-BV ...	MAP/THR-BV + RA-SLP ...	MAP-BV + LCE ...
chr22	4	3	3
chr20	6	5	5
chr18	8	6	6
chr16	9	8	7
chr1	22	18	17

Tabella C: risultati relativi alle varianti basate su *matching statistics* e *int vector compressi* per la *RLPBWT*.

Chr	MAP/THR-INT + RA-BV ...	MAP/THR-INT + RA-SLP ...	MAP-INT + LCE ...
chr22	1,4	0,8	0,8
chr20	2,2	1,2	1,2
chr18	2,8	1,6	1,5
chr16	3,4	1,9	1,8
chr1	8,0	4,4	4,3

Tabella 4.10: Risultati quantitativi dei tempi (secondi) di calcolo degli *SMEM*.Tabella A: risultati relativi alla *PBWT* e alle varianti basate su LCP per la *RLPBWT*.

Chr	matchIndexed	matchDynamic	MAP-INT+ LCP	MAP-BV+ LCP
chr22	144	17	61	114
chr20	243	28	89	210
chr18	305	35	119	255
chr16	424	42	140	299
chr1	1026	93	298	616

Tabella B: risultati relativi alle varianti basate su *matching statistics* e *bitvector sparsi* per la *RLPBWT*.

Chr	MAP/THR-BV + RA-BV ...	MAP/THR-BV + RA-SLP ...	MAP-BV + LCE ...
chr22	76	1305	260
chr20	121	2097	385
chr18	149	2509	442
chr16	189	3252	640
chr1	419	7531	1278

Tabella C: risultati relativi alle varianti basate su *matching statistics* e *int vector compressi* per la *RLPBWT*.

Chr	MAP/THR-INT + RA-BV ...	MAP/THR-INT + RA-SLP ...	MAP-INT + LCE ...
chr22	29	1344	250
chr20	49	2103	375
chr18	58	2483	421
chr16	68	3092	606
chr1	150	7234	1142

esclusa, in ogni caso, la costruzione della struttura base della *PBWT*. Tale risultato è visualizzabile in figura 4.8, dove si è deciso di escludere le strutture *MAP-INT + LCP* e *MAP-BV + LCP* in quanto non in grado di computare quali righe presentino un certo *SMEM*. I risultati quantitativi sono consultabili in tabella 4.11. Anche in questo caso, si conferma molto di quanto ipotizzato e discusso precedentemente. Caso a parte è dato dall'algoritmo *matchDynamic*, che risulta avere le performance peggiori, impiegando fino a cento volte il tempo dell'*algoritmo 5* di Durbin, ovvero *matchIndexed*. Per natura stessa dell'algoritmo, le operazioni sono studiate al fine di essere ottimizzate per pannelli di query e non per una query singola. Nel caso in analisi, infatti, si hanno molte operazioni che potrebbero essere ottimizzate per il caso della singola query, caso invece dell'algoritmo *matchIndexed*. Sperimentalmente si è quindi notato che una query o un centinaio di query presentano all'incirca i medesimi tempi di calcolo. Infatti, prendendo ad esempio il cromosoma 1 si ha che tale algoritmo impiega 93s per il calcolo con 100 query e una media di circa 88s (con una deviazione standard di ben 7s) per una singola query. Per quanto riguarda la *RLPBWT*, con l'uso della componente *RA-SLP*, si rilevano gli stessi problemi relativi all'random access, precedentemente descritti. Questi problemi sono risolti con l'uso della componente *RA-BV*. Inoltre, a parità di componenti per il mapping (e conseguenti componenti per le threshold), l'uso della componente *LCE* risulta più lenta dell'uso della componente *RA-BV*, a causa dei costi di calcolo delle *LCE query* stesse. Tutti questi sono risultati coerenti con quanto visto nel caso di 100 query, anche in termini di migliori strutture composte parlando di *RLPBWT*. Si segnala che, oltre al fatto che non sono qui misurate le tempistiche di caricamento delle strutture, le acquisizioni dei dati per la singola query sono state ottenute in un momento diverso da quelle per 100 query, avendo, di conseguenza, una non perfetta proporzione tra i risultati.

DC Non se dire  
questa cosa

Tabella 4.11: Risultati quantitativi dei tempi (secondi) di calcolo degli *SMEM* su singola query. I risultati sono nella forma “media  $\pm$  deviazione standard”.

Tabella A: risultati relativi alla *PBWT*.

Chr	matchIndexed	matchDynamic
chr22	0,15 $\pm$ 0,01	18,94 $\pm$ 0,58
chr20	0,25 $\pm$ 0,01	28,9 $\pm$ 1,99
chr18	0,33 $\pm$ 0,02	37,24 $\pm$ 1,08
chr16	0,38 $\pm$ 0,01	45,37 $\pm$ 3,49
chr1	1,01 $\pm$ 0,10	88,73 $\pm$ 7,08

Tabella B: risultati relativi alle varianti basate su *matching statistics* e *bitvector sparsi* per la *RLPBWT*.

Chr	MAP/THR-BV + RA-BV ...	MAP/THR-BV + RA-SLP ...	MAP-BV + LCE ...
chr22	0,75 $\pm$ 0,28	16,51 $\pm$ 1,34	2,92 $\pm$ 0,55
chr20	1,08 $\pm$ 0,05	22,85 $\pm$ 2,21	4,57 $\pm$ 0,82
chr18	1,36 $\pm$ 0,05	27,23 $\pm$ 2,50	4,96 $\pm$ 0,74
chr16	1,54 $\pm$ 0,05	38,93 $\pm$ 2,70	7,65 $\pm$ 1,14
chr1	3,41 $\pm$ 0,07	74,54 $\pm$ 3,40	12,75 $\pm$ 1,82

Tabella C: risultati relativi alle varianti basate su *matching statistics* e *int vector compressi* per la *RLPBWT*.

Chr	MAP/THR-INT + RA-BV ...	MAP/THR-INT + RA-SLP ...	MAP-INT + LCE ...
chr22	0,24 $\pm$ 0,02	13,52 $\pm$ 1,26	2,44 $\pm$ 0,54
chr20	0,37 $\pm$ 0,03	20,55 $\pm$ 2,16	3,68 $\pm$ 0,74
chr18	0,45 $\pm$ 0,03	24,43 $\pm$ 2,38	3,62 $\pm$ 0,70
chr16	0,54 $\pm$ 0,03	30,86 $\pm$ 2,35	5,93 $\pm$ 1,01
chr1	1,21 $\pm$ 0,06	68,22 $\pm$ 3,02	11,12 $\pm$ 1,52

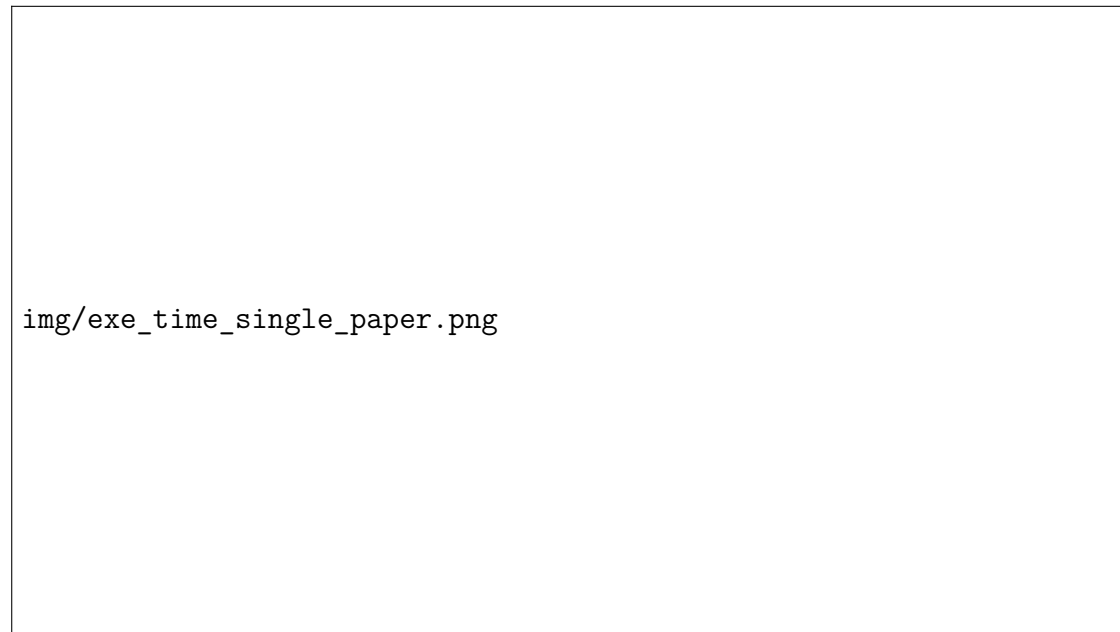


Figura 4.8: Tempo medio di esecuzione del calcolo degli SMEM per una singola query. Il grafico di destra è in scala logaritmica e, in entrambi, le barre d'errore rappresentano la deviazione standard.



# Capitolo 5

## Conclusioni

Fissato l'iniziale obiettivo di risolvere le problematiche relative alla memoria richiesta dall'*algoritmo 5* di Durbin, l'implementazione della **RLPBWT**, principalmente nelle strutture dati composte segnalate nel Capitolo 4, ha riportato risultati molto incoraggianti. Come descritto nel medesimo capitolo, la quantità di memoria richiesta risulta essere incredibilmente inferiore rispetto a quella richiesta dall'algoritmo **matchIndexed**. D'altro canto, l'algoritmo **matchDynamic** di Durbin, per quanto non approfondito nell'articolo del 2014 [4], risulta essere ancor meno esoso di risorse, nonché incredibilmente più veloce dal punto di vista dei tempi di calcolo, ad eccezione del caso limite di avere un numero esiguo di query. Lo svantaggio si ritrova anche nell'ordinamento dei risultati e nella creazione di un'ulteriore *PBWT* che, giudicando la letteratura degli ultimi anni le cui trattazioni si basano sempre sull'algoritmo 5, sembrano implicare un non facile riadattamento per la risoluzione di altri task.

Si possono comunque rilevare alcune possibili migliorie in merito alle varie implementazioni della *RLPBWT* presentate in questa tesi, riferendosi essenzialmente alle soluzioni che prevedono il calcolo dell'array delle *matching statistics*:

- si potrebbe pensare ad un metodo per gestire in modo efficiente lo studio di più query contemporaneamente, migliorando i tempi di calcolo complessivi. Studiando contemporaneamente tutte le query si potrebbe, alla stregua di quanto visto con l'algoritmo **matchDynamic**, caricare in memoria, di volta in volta, solo la colonna necessaria ad un dato passo di computazione o comunque un sottoinsieme di colonne. In tal modo, si ridurrebbe l'uso massimo di memoria
- studiare eventuali ottimizzazioni per la componente **MAP-BV** al fine di comprendere se sia possibile tenere in memoria un solo bitvector  $uv_k$  che funzioni in modo simile a quanto si ha con la componente **MAP-INT**

Nonostante queste possibili migliorie, la qualità dei risultati è sufficiente per stabilire che una variante *run-length encoded* della *PBWT*, alla stregua di quanto analizzato negli ultimi anni sulla *RLBWT* con *MONI* [7] e *PHONI* [8], sia possibile e possa permettere, nel prossimo futuro, la memorizzazione compatta delle informazioni necessarie allo studio di grandi pannelli di aplotipi. In un futuro in cui le tecnologie di sequencing produrranno sempre più dati, provenienti da sempre più individui, avere a disposizione strutture dati efficienti dal punto di vista della memorizzazione permetterà uno studio sempre più approfondito dei dati stessi, nei campi dei *Genome-Wide Association Studies (GWAS)*, della *medicina personalizzata* etc...

## 5.1 Sviluppi futuri

Ovviamente questa prima implementazione completa della *RLPBWT*, declinata nelle possibili strutture composte, non è da considerarsi come un punto di arrivo. Come accaduto per la *PBWT*, infatti, si potranno sviluppare nuove strutture dati basate su di essa per la gestione di pannelli di varia natura. Principalmente si può pensare a due casi, già anticipati nella sezione 2.5:

- **pannelli multi-allelici**, ovvero costruiti su un alfabeto  $\Sigma$  arbitrario e non limitato ai simboli  $\sigma = 0$  e  $\sigma = 1$
- **pannelli con dati mancanti**, ovvero pannelli costruiti direttamente da *dati reali* che possono contenere siti, per certi individui, per i quali non si ha certezza in merito all'allele

Inoltre, allo stato attuale, la struttura dati è stata sviluppata per permettere unicamente il calcolo dei match massimali con un aplotipo esterno. Anche in questo caso, quindi, si potrebbe avere lo sviluppo di nuovi algoritmi che rispondano a task diversi, come il calcolo dei match interni al panel, i cosiddetti *blocchi*, o anche il calcolo di tutti i match con un aplotipo esterno di lunghezza maggiore ad una fissata o che includano un numero stabilito di sequenze di aplotipi nel pannello.

**RLPBWT multi-allelica.** Per quanto i pannelli di aplotipi prodotti dal sequencing del genoma umano raramente presentino siti multi-allelici si ha una presenza stimata, al momento, di circa il 2% di siti tri-allelici [40]. Inoltre, all'aumentare della disponibilità di dati genomici, si ha in letteratura la propensione a credere che tale percentuale di siti sia non solo sottostimata (stimando che sia stimato circa un terzo dei reali siti tri-allelici) ma anche destinata a crescere in modo non lineare rispetto al numero di individui sequenziati [41]. Inoltre, molte specie, soprattutto vegetali, sono già riconosciute essere poliploidi, quindi una struttura dati efficiente

in memoria in grado di gestire pannelli costruiti su un alfabeto arbitrario risulterà necessaria nel breve futuro.

Ipotizzando un possibile funzionamento della **RLPBWT multi-allelica** (*m-RLPBWT*) si può pensare ad una soluzione molto simile a quanto visto per la *RLPBWT*. Infatti, per ogni colonna, si potrebbero memorizzare:

- una stringa che memorizzi quale simbolo corrisponda ad una certa run, non potendo sfruttare l'alternanza di simboli vista nel caso binario
- una rivisitazione delle strutture necessarie al mapping, tenendo in memoria vettori di *bitvector sparsi* o valori interi alla stregua della componente MAP-INT. Si segnala che si attende un'inversione di tendenza in termini di memoria, avendo che, in tal caso, l'uso di *int vector compressi* potrebbe rivelarsi meno efficiente dei *bitvector sparsi*
- riadattamento del calcolo dell'array delle *matching statistics*

In merito allo spazio richiesto e ai tempi di calcolo bisognerà considerare la grandezza dell'alfabeto su cui è costruito il pannello, che ci si aspetta comune inferiore a 10 nella maggioranza dei casi di studio biologico.

Nonostante, allo stato dell'arte, ci siano pochissimi studi in merito si ritiene possibile generalizzare, in modo computazionalmente efficiente, la *RLPBWT* anche a questa casistica.

**RLPBWT con dati mancanti.** La maggior parte delle soluzioni attualmente sviluppate sono basate su una forte assunzione: i dati in input sono corretti e senza dati mancanti. Ovviamente, limitandosi a studiare pannelli simulati o comunque “riempiti” in una fase di preprocessing, si rischia di non poter comprendere a fondo l'efficacia dei metodi su dati reali, oltre che a limitare l'inferenza dai pannelli stessi.

Come anticipato alla sezione 2.5, si sono iniziate a sviluppare estensioni della *PBWT* che ammettano wildcard, ovvero simboli nel pannello che possono assumere qualsiasi valore dell'alfabeto  $\Sigma$ , su cui è costruito il pannello stesso.

Uno degli sviluppi futuri sarebbe quindi quello di generalizzare la *RLPBWT*, ma anche l'eventuale *m-RLPBWT*, per la gestione di dati mancanti nel pannello. Inoltre si potrebbero sviluppare algoritmi in grado di gestire le wildcard anche all'interno delle query stesse.

Sempre in via ipotetica, l'uso di *algoritmi parametrici* (ma anche di *algoritmi approssimati*) adattati al funzionamento della *RLPBWT* potrebbero portare a soluzioni interessanti per la gestione di pannelli reali.

**K-SMEM.** Come anticipato, oltre che variare le caratteristiche del pannello in analisi, si possono studiare anche algoritmi per risolvere nuovi task con la *RLPBWT*.

Di recente, Gagie [42] ha proposto un articolo in cui dimostra come la struttura implementata in *MONI* [7] sia già predisposta al calcolo dei **k-MEM**, ovvero match massimali tra sotto-stringhe di un pattern e un testo che occorrono esattamente  $k$  volte nel testo stesso.

In merito alla *RLPBWT* si potrebbe adattare l'idea di Gagie al calcolo di **K-SMEM** tra sotto-stringhe dell'aplotipo query e il pannello che comportino il match con esattamente  $k$  righe del pannello stesso. La correlazione tra la *RLBWT* e la *RLPBWT* porta a pensare che tale problema sia risolvibile anche con la nuova definizione di *matching statistics* per la *RLPBWT*.

Nulla è stato sviluppato al momento ma si ritiene questo un'interessante sviluppo futuro in quanto permetterebbe studi statistici, molto comuni nei *GWAS*, in merito alla presenza di sotto-sequenze di un aplotipo esterno all'interno di un pannello di aplotipi.

*La tematica della pangenomica è praticamente nuova e il numero di problemi aperti è incredibilmente grande. I dati aumentano sempre di più e gli studi informatici devono evolversi per “stare al passo” con questa mole d'informazioni. Gli sviluppi futuri sono, da diversi punti di vista, anche imprevedibili. Risulta quindi difficile elencare in modo completo le possibilità future dietro questa branca della bioinformatica e dell'algoritmica sperimentale.*

DC Frase conclusiva da modificare fortemente

# Bibliografia

- [1] Hervé Tettelin, Vega Masignani, Michael J Cieslewicz, Claudio Donati, Duccio Medini, Naomi L Ward, Samuel V Angiuoli, Jonathan Crabtree, Amanda L Jones, A Scott Durkin, et al. Genome analysis of multiple pathogenic isolates of streptococcus agalactiae: implications for the microbial “pan-genome”. *Proceedings of the National Academy of Sciences*, 102(39):13950–13955, 2005.
- [2] Jasmijn A Baaijens, Paola Bonizzoni, Christina Boucher, Gianluca Della Vedova, Yuri Pirola, Raffaella Rizzi, and Jouni Sirén. Computational graph pangenomics: a tutorial on data structures and their applications. *Natural Computing*, pages 1–28, 2022.
- [3] The Computational Pan-Genomics Consortium. Computational pan-genomics: status, promises and challenges. *Briefings in Bioinformatics*, 19(1):118–135, 10 2016.
- [4] Richard Durbin. Efficient haplotype matching and storage using the positional Burrows–Wheeler transform (PBWT). *Bioinformatics*, 30(9):1266–1272, 01 2014.
- [5] Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. In *Annual Symposium on Combinatorial Pattern Matching*, pages 45–56. Springer, 2005.
- [6] Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in bwt-runs bounded space. *Journal of the ACM (JACM)*, 67(1):1–54, 2020.
- [7] Massimiliano Rossi, Marco Oliva, Ben Langmead, Travis Gagie, and Christina Boucher. MONI: A pangenomic index for finding maximal exact matches. *Journal of Computational Biology*, 02 2022.
- [8] Christina Boucher, Travis Gagie, I Tomohiro, Dominik Köppl, Ben Langmead, Giovanni Manzini, Gonzalo Navarro, Alejandro Pacheco, and Massimiliano

- Rossi. PHONI: Streamed matching statistics with multi-genome references. In *2021 Data Compression Conference (DCC)*, pages 193–202. IEEE, 2021.
- [9] Guy Joseph Jacobson. *Succinct static data structures*. Carnegie Mellon University, 1988.
- [10] Gonzalo Navarro. *Compact data structures: A practical approach*. Cambridge University Press, 2016.
- [11] Guy Jacobson. Space-efficient static trees and graphs. In *30th annual symposium on foundations of computer science*, pages 549–554. IEEE Computer Society, 1989.
- [12] 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.
- [13] Markus Lohrey. Algorithmics on SLP-compressed strings: A survey. *Groups-Complexity-Cryptology*, 4(2):241–299, 2012.
- [14] 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.
- [15] Travis Gagie, Giovanni Manzini, Gonzalo Navarro, Hiroshi Sakamoto, Yoshimasa Takabatake, et al. Rpair: rescaling repair with rsync. In *International Symposium on String Processing and Information Retrieval*, pages 35–44. Springer, 2019.
- [16] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948, 1993.
- [17] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.
- [18] Juha Kärkkäinen, Giovanni Manzini, and Simon J Puglisi. Permuted longest-common-prefix array. In *Annual Symposium on Combinatorial Pattern Matching*, pages 181–192. Springer, 2009.
- [19] Kunihiko Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *SODA*, volume 2, pages 225–232, 2002.

- [20] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Annual Symposium on Combinatorial Pattern Matching*, pages 181–192. Springer, 2001.
- [21] Michael Burrows and David Wheeler. A block-sorting lossless data compression algorithm. 1994.
- [22] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proceedings 41st annual symposium on foundations of computer science*, pages 390–398. IEEE, 2000.
- [23] Alberto Policriti and Nicola Prezza. LZ77 computation based on the run-length encoded BWT. *Algorithmica*, 80(7):1986–2011, 2018.
- [24] Alan Kuhnle, Taher Mun, Christina Boucher, Travis Gagie, Ben Langmead, and Giovanni Manzini. Efficient construction of a complete index for pan-genomics read alignment. *Journal of Computational Biology*, 27(4):500–513, 2020.
- [25] Taher Mun, Alan Kuhnle, Christina Boucher, Travis Gagie, Ben Langmead, and Giovanni Manzini. Matching reads to many genomes with the r-index. *Journal of Computational Biology*, 27(4):514–518, 2020.
- [26] Christina Boucher, Travis Gagie, Alan Kuhnle, Ben Langmead, Giovanni Manzini, and Taher Mun. Prefix-free parsing for building big BWTs. *Algorithms for Molecular Biology*, 14(1):1–15, 2019.
- [27] Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, and David J Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, 1990.
- [28] Hideo Bannai, Travis Gagie, and I Tomohiro. Refining the r-index. *Theoretical Computer Science*, 812:96–108, 2020.
- [29] Richard A Gibbs, John W Belmont, Paul Hardenbol, Thomas D Willis, FL Yu, HM Yang, Lan-Yang Ch’ang, Wei Huang, Bin Liu, Yan Shen, et al. The international HapMap project. *Nature*, 2003.
- [30] Ahsan Sanaullah, Degui Zhi, and Shaojie Zhang. d-PBWT: dynamic positional Burrows–Wheeler transform. *Bioinformatics*, 37(16):2390–2397, 02 2021.
- [31] Ardalan Naseri, Degui Zhi, and Shaojie Zhang. Multi-allelic positional Burrows–Wheeler transform. *BMC bioinformatics*, 20(11):1–8, 2019.

- [32] Ardalan Naseri, Erwin Holzhauser, Degui Zhi, and Shaojie Zhang. Efficient haplotype matching between a query and a panel for genealogical search. *Bioinformatics*, 35(14):i233–i241, 2019.
- [33] Lucia Williams and Brendan Mumey. Maximal perfect haplotype blocks with wildcards. *Iscience*, 23(6):101149, 2020.
- [34] Simone Rubinacci, Olivier Delaneau, and Jonathan Marchini. Genotype imputation using the positional Burrows Wheeler transform. *PLoS genetics*, 16(11):e1009049, 2020.
- [35] Nathaniel K Brown, Travis Gagie, and Massimiliano Rossi. RLBWT tricks. *arXiv preprint arXiv:2112.04271*, 2021.
- [36] 1000 Genomes Project Consortium et al. A global reference for human genetic variation. *Nature*, 526(7571):68, 2015.
- [37] Leonardo Dagum and Ramesh Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.
- [38] Petr Danecek, James K Bonfield, Jennifer Liddle, John Marshall, Valeriu Ohan, Martin O Pollard, Andrew Whitwham, Thomas Keane, Shane A McCarthy, Robert M Davies, and Heng Li. Twelve years of SAMtools and BCFtools. *GigaScience*, 10(2), 02 2021. giab008.
- [39] Felix Mölder, Kim Philipp Jablonski, Brice Letcher, Michael B Hall, Christopher H Tomkins-Tinch, Vanessa Sochat, Jan Forster, Soohyun Lee, Sven O Twardziok, Alexander Kanitz, et al. Sustainable data analysis with snakemake. *F1000Research*, 10, 2021.
- [40] Alan Hodgkinson and Adam Eyre-Walker. Human triallelic sites: evidence for a new mutational mechanism? *Genetics*, 184(1):233–241, 2010.
- [41] Ian M Campbell, Tomasz Gambin, Shalini N Jhangiani, Megan L Grove, Narayanan Veeraraghavan, Donna M Muzny, Chad A Shaw, Richard A Gibbs, Eric Boerwinkle, Fuli Yu, et al. Multiallelic positions in the human genome: challenges for genetic analyses. *Human mutation*, 37(3):231–234, 2016.
- [42] Travis Gagie. MONI can find k-MEMs. *arXiv preprint arXiv:2202.05085*, 2022.