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.ssa 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

Capitolo 1

Introduzione

Negli ultimi anni si è assistito a 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 di 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, a meno di mutazioni, 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 ha quindi permesso lo sviluppo di diverse strutture dati composte per la variante **run-length encoded** della **PBWT** (RLPBWT), efficienti dal punto di vista della memoria utilizzata. Tale progetto è stato svolto in collaborazione con due tra gli autori dei principali risultati ottenuti per la **trasformata di Burrows–Wheeler run-length encoded** (RLBWT) [5] [6] [7] [8]: il prof. Gagie (Dalhousie University) e la prof.ssa Boucher (University of Florida).
Parte delle strutture dati e degli algoritmi presentati in questa tesi sono inseriti in un articolo, dal titolo *Compressed data structures for population-scale positional Burrows–Wheeler transforms* [9]. Al momento della stesura di questa tesi, tale articolo è in fase di review per la rivista *Genome Research* (*Cold Spring Harbor Laboratory Press*).

Struttura della tesi

Nel Capitolo 2, si introdurranno i concetti di base, di ambito computazionale e bioinformatico, 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 composte per la RLPBWT e al calcolo degli SMEM. Nel Capitolo 4, si discuteranno i risultati ottenuti durante la sperimentazione sui dati reali della *phase 3* del **1000 Genome Project** [38], progetto, iniziato nel 2008, che ha visto lo sforzo della comunità scientifica internazionale per la catalogazione delle varianti geniche umane. Infine, nel Capitolo 5, si trarranno le conclusioni di questo progetto di tesi, discutendone gli sviluppi futuri.

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

A livello di notazione, si specifica che, con $T[i, j]$ si intende la sottostringa del testo/sequenza/riga/colonna T, iniziante all'indice i e terminante all'indice j incluso. Qualora si avesse $j > i$ si identifica la sottostringa nulla ε.

2.1 Bitvector

Nonostante qualche primo risultato isolato, si identifica l'inizio dello studio delle **strutture dati succinte** con la tesi di dottorato di Jacobson del 1988 [10]. Jacobson, con questo termine, denota quelle strutture dati che usano $\log N + o(\log N)$ bit, con N numero dei differenti oggetti da memorizzare. Ad esempio, assumendo un array di n bit, una struttura dati succinta utilizza $n + o(n)$ bit, avendo infatti $N = 2^n$ [11].
Un anno dopo, Jacobson [12] 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:*

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

In alternativa si potrebbe avere, come formalismo:

$$B[i] \in \{\bot, \top\},\ \forall i\ t.c.\ 0 \leq i < n \tag{2.2}$$

Nel corso degli ultimi anni, si sono sviluppate diverse varianti dei bitvector, finalizzate a offrire diversi costi di complessità spaziale e diversi tempi computazionali per le principali funzioni offerte.
Il primo vantaggio di questa struttura dati è quello di garantire **random access** in tempo costante, pur sfruttando varie tecniche per la memorizzazione efficiente della stessa in memoria. Lo spazio necessario per l'implementazione delle principali varianti nella **Succinct Data Structure Library (*SDSL*)** [13] (una delle principali librerie, scritta in C++11, per strutture dati succinte) è visualizzabile in tabella 2.1. Il secondo vantaggio consiste nel fatto che i bitvector permettono l'implementazione efficiente di due funzioni: la funzione rank e la funzione select. Un'implementazione naïve 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 della SDSL, le stime asintotiche 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 (bit) per la memorizzazione di alcune varianti di *bitvector*. Si assume un bitvector di lunghezza n con un numero di simboli $\sigma = 1$ (o $\sigma = \top$) pari a m. K indica la *block size* per l'*interleave*.

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

2.1.1 Funzione rank

La prima funzione che si approfondisce è la funzione rank, che permette di calcolare il *rango* 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 valori uguali a 1 presenti fino a quella data posizione esclusa. Più formalmente, si ha che:*

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

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 del fatto che si ha una quantità diversa di bit aggiuntivi salvati in memoria. In tabella 2.2 sono visualizzabili le complessità temporali stimate della funzione rank, per le varianti di bitvector implementate nella SDSL.

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 simboli $\sigma = 1$ pari a m, e un numero k di *rank sample*.

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

2.1.2 Funzione select

La seconda funzione fondamentale è la funzione select, che 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. Più formalmente, si ha che:*

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

Anche in questo caso vale lo stesso discorso fatto per la funzione rank in merito alla complessità temporale e ai bit aggiuntivi. In tabella 2.3 sono visualizzabili le complessità temporali stimate della funzione select, per le varianti di bitvector

implementate nella SDSL.

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

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:

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

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

Si vedrà, in questo progetto di tesi, come l'uso di tali strutture, nelle varianti *plain bitvector* e *bitvector sparsi*, sia fondamentale per lo studio delle due strutture run-length encoded.

2.2 Straight-Line Program

In ambito bioinformatico, una delle principali problematiche è la gestione di testi molto estesi. Si pensi, ad esempio, al caso umano, dove il primo cromosoma, il più lungo, conta circa 247.249.719 *pb* (paia di basi), nonostante l'uomo non sia 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 pensato all'uso dei cosiddetti **Straight-Line Program** (SLP). In termini generici, un SLP è una *grammatica context-free* che genera una e una sola parola [14]. Si parla, quindi, di *grammar-based compression*.

Definizione 4. *Sia dato un alfabeto finito Σ di simboli terminali. Sia data una stringa $s = a_1, a_2, \ldots, a_n \in \Sigma^*$, lunga n e costruita sull'alfabeto Σ, avendo $a_i \in \Sigma$, $\forall\, i$ t.c. $1 \leq i \leq n$. Si denota con $alph(s) = \{a_1, a_2, \ldots a_n\}$ l'insieme dei simboli della stringa s.*
Si definisce SLP*, costruito sull'alfabeto Σ, una grammatica context-free $\mathcal{A}$ tale che:*

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

Dove:

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

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

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, α) può anche essere indicata con $A \to \alpha$)*
2. *la relazione $\{(A, B) \mid (A, \alpha) \in \mathcal{P},\ B \in alph(\alpha)\}$ è aciclica*

DC verificare questo secondo punto

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

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

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

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 con un simbolo di $\mathcal{V} \cup \Sigma$ e ogni foglia con un simbolo di Σ.

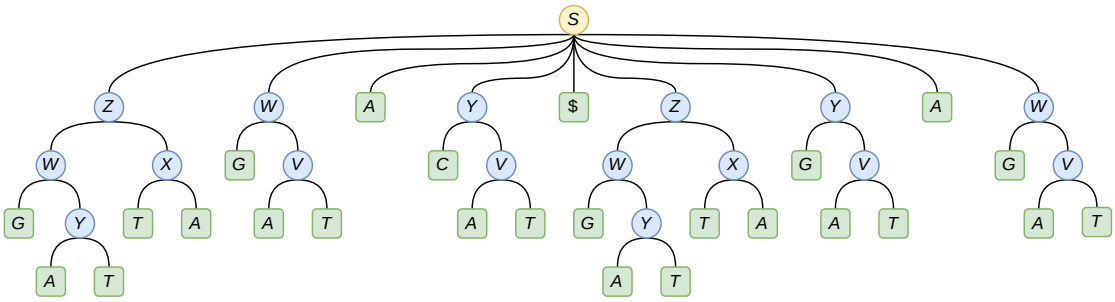
Esempio 2. *Si prenda, ad esempio [15], la seguente stringa:*

$$s = GATTAGATACAT\$GATTACATAGAT$$

Si potrebbe produrre il seguente SLP*:*

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

Al quale corrisponde il seguente albero di derivazione:



Si noti che il simbolo iniziale non terminante, ovvero la radice, è indicato con un cerchio giallo, i simboli non terminanti, ovvero i nodi interni, sono indicati dai cerchi blu mentre i simboli terminanti, ovvero le foglie, sono indicati dai quadrati verdi.

Nel 2020, Gagie et al. [15] 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:

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

L'uso degli *SLP* è stato cruciale, come si vedrà più avanti in questa tesi, per la costruzione della variante run-length encoded sia della BWT che della PBWT.

2.2.1 Longest Common Extension

Oltre a permettere il *random access* al testo compresso, l'uso degli SLP permette di effettuare un'altra operazione in modo efficiente, ovvero il calcolo delle **Longest Common Extension** (LCE) **query**.

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 sottostringhe che hanno come indice di partenza i e j, ovvero 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 agli indici i e j del testo compresso, per poi "risalire" l'albero di derivazione al fine

di computare il prefisso comune tra le due sottostringhe.
Avendo l'SLP di un testo T lungo n, si stima che il calcolo di una LCE query di lunghezza l sia effettuabile in tempo:

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

Si noti che, normalmente, la lunghezza dell'LCE è trascurabile rispetto alla lunghezza del testo.
In questa tesi, i due concetti di SLP ed LCE query verranno generalizzati all'uso su matrici, permettendo una rappresentazione compatta in memoria per un pannello di aplotipi, garantendo random access.

Librerie

Da un punto di vista implementativo, l'oggetto contenente l'SLP del pannello viene costruito e interrogato mediante l'uso della libreria **ShapedSlp**[1], implementazione dei risultati teorici ottenuti da Gagie et al. [15]. Inoltre, tale libreria basa il suo funzionamento sull'uso di un'altra libreria, detta **BigRePair**[2], che implementa ulteriori risultati teorici di Gagie et al. [16] in merito alla compressione, via uso di grammatiche, di file con frequenti ripetizioni (come, nel nostro caso, i pannelli binari di aplotipi).
In termini di pipeline, si procede:

1. generando la grammatica tramite BigRePair, che accetta come file di input un file `txt` "raw" oppure file in formato standard nel campo della bioinformatica, come i `FASTA`

2. generando l'SLP tramite ShapedSlp specificatamente a partire dai risultati di BigRePair (si segnala che la libreria accetta anche grammatiche prodotte da altri software).

2.3 Suffix Array

Nel 1976, Manber e Myers [17] 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 nominata **Suffix Array** (SA).

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 array di interi lungo n, tale*

[1] `ttps://github.com/itomomoti/ShapedSlp`
[2] `https://gitlab.com/manzai/bigrepair`

che $\mathsf{SA}_T[i] = j$ *se e solo se (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* $0 \leq i < i' < n$ *e indicando con* $\prec$ *l'ordinamento lessicografico:*

$$T[\mathsf{SA}_T[i], n-1] \prec T[\mathsf{SA}_T[i'], n-1] \tag{2.10}$$

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

Esempio 3. *Data la stringa:*

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

Si producono i seguenti suffissi e il loro riordinamento:

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

$\Longrightarrow$

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

Avendo che:

$$\mathsf{SA}_T = [11, 10, 7, 4, 1, 0, 9, 8, 6, 3, 5, 2]$$

2.3.1 Longest common prefix

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

Definizione 7. *Si definisce il Longest Common Prefix 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* $\mathsf{lcp}(x, y)$ *il più lungo prefisso comune tra le stringhe* x *e* y*:*

$$\mathsf{LCP}_T[i] = \begin{cases} -1 & \textit{se } i = 0 \vee i = n \\ \left|\mathsf{lcp}(T[\mathsf{SA}_T[i-1], n], T[\mathsf{SA}_T[i], n])\right| & \textit{altrimenti} \end{cases} \tag{2.11}$$

Esempio 4. *Riprendendo l'esempio precedente si ha:*

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

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

$$\mathcal{O}(m \log n) \tag{2.12}$$

Con l'uso dell' LCP, la complessità temporale si riduce a:

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

Per ulteriori approfondimenti, in merito agli algoritmi di pattern matching basati su suffix array e ai relativi "acceleratori", si rimanda al testo di Gusfield [18].

2.3.2 Inverse suffix array

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:*

$$\mathsf{ISA}_T[i] = j \iff \mathsf{SA}_T[j] = i, \ \forall\, i \in \{0, n-1\} \tag{2.14}$$

Esempio 5. *Riprendendo l'esempio precedente si ha:*

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 Permuted longest common prefix

Un'altra permutazione che bisogna introdurre è il **permuted longest common prefix** (PLCP) **array** [19]. Tale permutazione consente una rappresentazione succinta in memoria dell'LCP [20], permettendo di ottenere gli stessi risultati di quest'ultimo.

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

$$\mathsf{PLCP}_T[p] = \begin{cases} -1 & \textit{se } \mathsf{ISA}_T[p] = 0 \\ \mathsf{LCP}_T[\mathsf{ISA}_T[p]] & \textit{altrimenti} \end{cases}, \ \forall p \in \{0, n-1\} \tag{2.15}$$

Si noti che 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 [19]:* $\mathsf{PLCP}_T[\mathsf{SA}_T[p]] = \mathsf{LCP}_T[p], \ \forall p \in \{1, n-1\}$

Esempio 6. *Riprendendo l'esempio precedente si ha:*

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	sissippi$
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 [21].

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

$$\mathsf{PLCP}_T[i] \geq \mathsf{PLCP}_T[i-1] - 1, \ \forall\, i \in \{1, n-1\} \tag{2.16}$$

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 [21], si definisce array PLCP sparso, lungo $\left\lfloor \frac{n}{q} \right\rfloor$ e denotato PLCP_q, come l'array che memorizza ogni q-esimo valore del PLCP, avendo che:*

$$\mathsf{PLCP}_q[i] = \mathsf{PLCP}_T[iq] \tag{2.17}$$

2.3.4 Funzione phi

L'ultimo concetto che si introduce sono le **funzioni $\boldsymbol{\varphi}$** e **$\boldsymbol{\varphi^{-1}}$**, usate per poter identificare i valori precedenti e successivi di un dato valore in SA_T. Esse sono utili per ricostruire efficientemente il PLCP di un testo T (per i dettagli si rimanda all'articolo di Kärkkäinen [19]) e per permettere, come si vedrà più avanti nella sezione 2.4, il riconoscimento di tutte le occorrenze di un **maximal exact match** (MEM) in T [8].

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

$$\varphi(p) = \begin{cases} \mathsf{null} & \text{se } \mathsf{ISA}_T[p] = 0 \\ \mathsf{SA}_T[\mathsf{ISA}_T[p] - 1] & \text{altrimenti} \end{cases}, \ \forall\, p \in \{0, n-1\} \tag{2.18}$$

$$\varphi^{-1}(p) = \begin{cases} \mathsf{null} & \text{se } \mathsf{ISA}_T[p] = n-1 \\ \mathsf{SA}_T[\mathsf{ISA}_T[p] + 1] & \text{altrimenti} \end{cases}, \ \forall\, p \in \{0, n-1\} \tag{2.19}$$

Si noti che si ha il valore null quando si studia il primo e l'ultimo valore del suffix array in quanto non hanno, rispettivamente, l'antecedente e il successore.
Analogamente, coi medesimi vincoli, le due funzioni possono essere definite come [19]:

$$\varphi(\mathsf{SA}[p]) = \mathsf{SA}[p-1] \tag{2.20}$$

$$\varphi^{-1}(\mathsf{SA}[p]) = \mathsf{SA}[p+1] \tag{2.21}$$

Esempio 7. *Riprendendo l'esempio precedente si ha:*

Indice	SA_T	ISA_T	φ	φ^{-1}	**Suffisso**
0	11	5	1	9	\$
1	10	4	4	0	i\$
2	7	11	5	null	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	null	10	ssissippi\$

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** (BWT) [22] è divenuta ormai uno standard nell'algoritmica su stringhe e nella bioinformatica, grazie ai suoi molteplici vantaggi, sia dal punto di vista della complessità temporale che da quello dell'efficienza in memoria.
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. La proprietà per cui caratteri uguali tendono ad essere posti consecutivamente all'interno della stringa prodotta dalla trasformata 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 trasformata di Burrows–Wheeler 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 di* T *nel riordinamento lessicografico. Più formalmente, si ha che:*

$$\mathsf{BWT}_T[i] = \begin{cases} T[\mathsf{SA}_T[i] - 1] & \textit{se } \mathsf{SA}_T[i] \neq 1 \\ \$ & \textit{altrimenti} \end{cases}, \ \forall\, i \in \{0, n-1\} \tag{2.22}$$

In termini operativi, 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 $\mathsf{ROT}_T(i)$, 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:*

$$\mathsf{ROT}_T(i) = T[i, n-1] \cdot T[0, i-1],\ \forall\, i \in \{0, n-1\} \tag{2.23}$$

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 di SA_T a cui viene concatenata la parte restante del testo.
Un altro array spesso utilizzato insieme alla BWT è il cosiddetto array F, lungo $|T|$ e formato dalla prima colonna della BWM. In pratica, l'array F_T è formato dal riordinamento lessicografico dei caratteri del testo T.

Esempio 8. *Data la stringa:*

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

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 al fatto che sia reversibile, implicando che, a partire da BWT_T, sia possibile ricostruire T. Questo è possibile grazie ad una proprietà intrinseca della trasformata, riassunta nel concetto di **LF-mapping**.

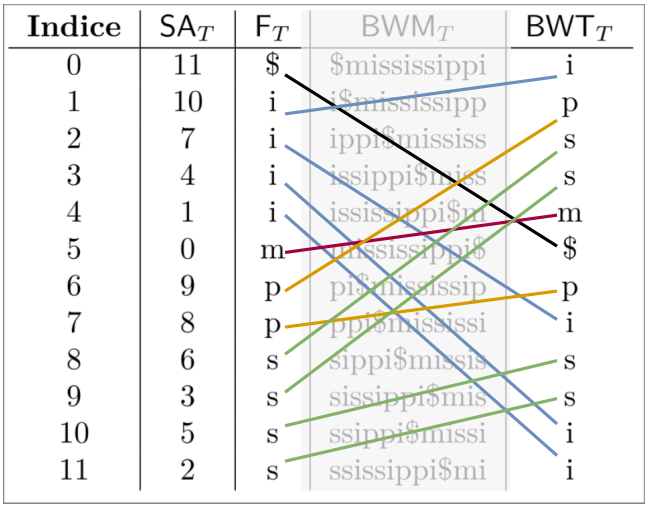
Definizione 14. *Dato un testo T, tale che $|T| = n$, data la sua BWT_T e il suo array F_T, si definisce LF-mapping come la proprietà per la quale l'i-esima occorrenza di un carattere σ in BWT_T corrisponde all'i-esima occorrenza dello stesso carattere in F_T.*

Grazie a questa definizione è possibile partire dall'ultimo carattere del testo, ovvero $\sigma = \$$, e ricostruire l'intero testo a ritroso.

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

$$\mathsf{BWT}_T = ipssm\$pissii \text{ e } \mathsf{F}_T = \$iiiimppssss$$

Si hanno i seguenti caratteri associati dall'LF-mapping:



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* $\mathsf{BWT}_T[0] =$ *'i' in* T*. Quindi 'i' precederà '\$' in* T*:*

$$T = \ldots i\$$$

Inoltre, si ha 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 carattere allo stesso indice di tale 'i' nella* BWT_T*, ovvero 'p', sarà il simbolo che precede 'i' nel testo:*

$$T = \ldots pi\$$$

Proseguendo a ritroso, si ricostruisce l'intero testo:

$$T = mississippi\$$$

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 e viene esteso usando il carattere $P[i-1]$, selezionando il nuovo intervallo sul suffix array. Esso 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 $P[i-1, |P|-1]$.
Con 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 Σ, si definisce C come una funzione:*

$$\mathsf{C} : \Sigma \cup \$ \rightarrow \mathbb{N} \tag{2.24}$$

Dato un simbolo $\sigma \in \Sigma$, $\mathsf{C}(\sigma)$ restituisce il numero di occorrenze dei caratteri lessicograficamente più piccoli di σ in T.

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

$$\mathsf{Occ} : \Sigma \cup \$ \times \{0, n\} \rightarrow \mathbb{N} \tag{2.25}$$

Dato un simbolo $\sigma \in \Sigma$ e una posizione i della BWT_T, $\mathsf{Occ}(\sigma, i)$ restituisce il numero di occorrenze del carattere σ nei primi i elementi di BWT_T.

Questa coppia di funzioni prende il nome di **FM-index** [23], il quale è definito essere un self index in quanto è possibile tenere in memoria solo tale indice per ottenere i risultati medesimi della BWT_T.

Esempio 10. *Dato il seguente testo e la corrispondente BWT_T:*

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

$$\mathsf{BWT}_T = ipssm\$pissii$$

Si hanno, per $\mathsf{C}(\sigma)$ *e* $\mathsf{Occ}(\sigma, i)$*:*

σ	$\mathsf{C}(\sigma)$
\$	0
i	1
m	5
p	6
s	8

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/σ	\$	i	m	p	s

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

$$f' = \mathsf{C}(\sigma) + \mathsf{Occ}(\sigma, f), \quad g' = \mathsf{C}(\sigma) + \mathsf{Occ}(\sigma, g) \tag{2.26}$$

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 si basa sull'LF-mapping. Infatti, partendo da un intervallo su SA_T (che è anche un intervallo su BWT_T), si identificano i suffissi che 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 $\mathsf{C}(\sigma)$ e $\mathsf{Occ}(\sigma, i)$ per trovare tali caratteri su F_T, calcolando il nuovo intervallo $[f, g)$.

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

$$f' = \mathsf{C}(s) + \mathsf{Occ}(s, 0) = 8 + 0 = 8$$

$$g' = \mathsf{C}(s) + \mathsf{Occ}(s, 12) = 8 + 4 = 12$$

Ottenendo l'intervallo $[8, 12)$ *sul suffix array.*
Si prosegue leggendo il carattere $\sigma = P[1] = s$*:*

$$f' = \mathsf{C}(s) + \mathsf{Occ}(s, 8) = 8 + 2 = 10$$

$$g' = \mathsf{C}(s) + \mathsf{Occ}(s, 12) = 8 + 4 = 12$$

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

$$f' = \mathsf{C}(i) + \mathsf{Occ}(i, 10) = 1 + 2 = 3$$

$$g' = \mathsf{C}(i) + \mathsf{Occ}(i, 12) = 1 + 4 = 5$$

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

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

$\Longrightarrow$

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

$\Downarrow$

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

$\Longrightarrow$

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

Si ottiene che le occorrenze del pattern $P =$ *iss iniziano alle posizioni* $\mathsf{SA}_T[3] = 4$ *e* $\mathsf{SA}_T[4] = 1$ *del testo.*

2.5 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 a una tecnica 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. *data la seguente stringa:*

$$s = aaaacctgggggg$$

La sua memorizzazione run-length encoded è:

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

2.5.1 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*:*

$$|\mathsf{RLBWT}_T| = r \tag{2.27}$$

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 memorizzare in modo compatto la RLBWT è quella di salvare:

- 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 $\mathsf{BWT}_T[i]$ è il primo carattere, detto anche testa, di una run

Esempio 13. *Data la seguente* BWT_T*:*

$$\mathsf{BWT}_T = acggtcccaa$$

Si hanno:

$$c = acgtca \qquad bv = 1110110010$$

Mäniken e Navarro hanno proposto anche il seguente teorema.

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

$$\mathcal{O}(m \log n) \tag{2.28}$$

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

- la RLBWT
- dei suffix array sample, 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 $\mathcal{O}(m \log n)$ e in spazio $\mathcal{O}(r)$
- localizzare tali occorrenze nel testo (*locate query*) in tempo $\mathcal{O}(s)$ e in spazio $\mathcal{O}\left(\frac{r}{s}\right)$, avendo s come distanza tra due SA *sample*

Nel 2017, Policriti and Prezza [24] 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:*

$$\mathcal{O}(m \log \log n) \tag{2.29}$$

Questo risultato dimostra come identificare un solo 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 sample in quell'intervallo.
Nel 2020 Gagie et al [6], combinando la RLBWT, il Toehold lemma e la già introdottta funzione φ, hanno trovato una soluzione a questo problema, permettendo 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 $\mathcal{O}(r)$, avendo che si possano trovare tutte le k occorrenze di un pattern P, lungo m, in tempo:*

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

Nel dettaglio, i risultati di Gagie hanno portato a ridefinire l'r-index tramite l'uso dei valori del SA all'inizio e alla fine di ogni run come SA sample. Si è quindi ottenuto che i SA sample possono essere memorizzati in spazio proporzionale al numero di run, pur permettendo in modo efficiente le locate query.
Per ulteriori dettagli in merito alla costruzione dell'r-index si rimanda anche ai paper di Kuhnle et al. [25], di Mun et al. [26] e di Boucher et al. [27].

2.5.2 Match massimali con RLBWT

Dopo aver introdotto l'r-index, bisogna descrivere 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, essendo quindi massimali.

Definizione 18. *Dato un testo T, con $|T| = n$, e un pattern P, con $|P| = m$, si definisce* MEM *di P in T una sottostringa $P[i, i + l - 1]$, di lunghezza l, 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 del calcolo dei match massimali esatti si ritrova nel loro uso nei metodi di allineamento basati sul *paradigma seed-and-extend*. Tale paradigma si basa sul trovare MEM di piccola lunghezza, detti seed, per poi continuare l'allineamento tramite algoritmi più sofisticati, spesso basati sulla programmazione dinamica, ed è sfruttato in algoritmi di allineamento come BLAST [28], uno degli allineatori più usati al mondo.
Nel 2020, Bannai et al. [29] hanno mostrato come il calcolo dei MEM sia equivalente al calcolo delle **Matching Statistics** (MS), un concetto teorico molto usato in bioinformatica.

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* $(\mathsf{pos}, \mathsf{len})$*, lungo quanto il pattern, tale che:*

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

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 $P[i, |P| - 1]$.
Si ha il seguente lemma.

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

$$P[i, i + l - 1], \forall\, 0 < i \leq m \tag{2.31}$$

è un MEM*, di lunghezza l, in T sse:*

$$\mathsf{MS}[i].\mathsf{len} = l \wedge \mathsf{MS}[i-1].\mathsf{len} \leq \mathsf{MS}[i].\mathsf{len} \tag{2.32}$$

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

$$\mathsf{MS}[0].\mathsf{len} = 1 \wedge \mathsf{MS}[0].\mathsf{len} \geq \mathsf{MS}[1].\mathsf{len} \tag{2.33}$$

Per costruire l'array delle matching statistics l'approccio naïve è quello di sfruttare interamente l'array LCP ma, nell'articolo di Bannai et al. [29], si è presentato una semplice concetto in grado di ottimizzare il processo, quello delle **threshold**. Questa piccola struttura dati memorizza il primo valore minimo dell'array LCP tra due run consecutive del medesimo simbolo nella BWT.

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

$$j < i \leq k \text{ t. c. } i \text{ è l'indice del primo valore minimo in } \mathsf{LCP}_T[j+1, k] \tag{2.34}$$

Rossi et al., nel 2021, hanno sfruttato 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 la costruzione dell'array MS, in due sweep sul pattern P (lungo m), tramite l'**algoritmo di Bannai**. Infatti, si ha:

- un primo sweep che computa i valori $\mathsf{MS}[i].\mathsf{pos}$, $\forall i \in \{0, m-1\}$
- un secondo sweep che, tramite random access sul testo T computa i valori $\mathsf{MS}[i].\mathsf{len}$, $\forall i \in \{0, m-1\}$, 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 $\mathsf{MS}[i].\mathsf{pos}$, $\forall i \in \{0, m-1\}$, si procede scorrendo il pattern P, lungo m, da destra a sinistra. Brevemente, i passi dell'algoritmo sono:

1. si inizia cercando l'ultima occorrenza, di indice q, di $P[m-1]$, in BWT_T, 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. a questo punto si hanno due alternative:

   - se $\mathsf{BWT}_T[q] = P[i-1]$ si procede con il mapping come in 2, dopo aver memorizzato $\mathsf{MS}[i].\mathsf{pos} = \mathsf{SA}_T[q]$
   - se $\mathsf{BWT}_T[q] \neq P[i-1]$ si deve selezionare un nuovo q tale per cui $\mathsf{BWT}_T[q] = P[i-1]$. Questo può essere l'indice della coda della run precedente di simboli $P[i-1]$ o la testa della run successiva di simboli $P[i-1]$. Qualora la run attuale non sia 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. Avendo $q < t$, 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 si seleziona il suffisso più lungo che presenta un match con il suffisso corrente del pattern, esteso a sinistra con $P[i-1]$. Questo è garantito dal riordinamento lessicografico che costruisce BWT_T. Una volta scelto il nuovo q si procede con il mapping come in 2, dopo aver memorizzato $\mathsf{MS}[i].\mathsf{pos} = \mathsf{SA}_T[q]$

4. si itera fino a 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, 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]. Una corretta stima della complessità temporale risulta difficile in quanto dipendente dalla struttura con cui si memorizza il testo T sul quale fare random access. Nell'articolo si riporta che i valori $\mathsf{MS}[i].\mathsf{pos}$, $\forall i \in \{0, m-1\}$, sono calcolabili in tempo $\mathcal{O}(m \log\log n)$, per un testo T lungo n e un pattern P lungo m, assumendo che si possa accedere alla posizione delle threshold in tempo $\mathcal{O}(\log\log n)$ e che i backward-step siano effettuabili in tempo $\mathcal{O}(\log\log n)$. Analogamente, ipotizzando di avere random access su T in tempo $\mathcal{O}(\log\log n)$ (tramite una struttura compatta basata sul Tabix index [30]), i valori $\mathsf{MS}[i].\mathsf{len}$, $\forall i \in \{0, m-1\}$, sono calcolabili in tempo

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 ← [(pos : 0, len : 0) ... (0, 0)]                ▷ |P| = m, |T| = n, |MS| = m
3:     q ← posizione dell'ultima occorrenza di P[m − 1] in BWT_T
4:     pos ← SA_T[q]                                        ▷ calcolo MS[i].pos
5:     for every i ∈ [0, m − 1] do
6:         if BWT_T[q] ≠ P[i] then
7:             if BWT_T[q] è prima della relativa threshold per P[i] then
8:                 q ← posizione dell'occorrenza precedente di P[i] in BWT_T
9:             else
10:                q ← posizione dell'occorrenza successiva di P[i] in BWT_T
11:            pos ← SA_T[q]
12:        MS[i].pos ← pos
13:        q ← LF(q), pos ← pos − 1
14:    for every i ∈ [0, m − 1] do                          ▷ calcolo MS[i].len
15:        MS[i].len ← MS[i − 1].len − 1
16:        while P[i + MS[i].len] = T[MS[i].pos + MS[i].len] do
17:            MS[i].len ← MS[i].len + 1
18:    return MS
```

$\mathcal{O}(m \log\log n)$. Tali stime sono fortemente teoriche e si rimanda all'articolo per ulteriori dettagli.

2.5.3 Uso delle LCE query

Nel 2021, Boucher, Gagie, Rossi et al. hanno proposto un ulteriore miglioramento di quanto fatto in MONI, con **PHONI: *Streamed Matching Statistics with Multi-Genome References*** [8].
In questo progetto, non solo si è sostituito l'uso delle thresholds con l'uso delle LCE query, riducendo l'algoritmo a un solo sweep sull'array delle matching statistics (permettendo un uso "online" dell'algoritmo), ma si è esplicitato anche l'utilizzo delle funzioni φ e φ^{-1} e di PLCP_T per il riconoscimento di tutte le occorrenze di ogni MEM tra un pattern e un testo, come riportato all'algoritmo 2.2 [8].
A tal fine, si sfrutta 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 possono computare $\varphi(p)$, $\varphi^{-1}(p)$ e $\mathsf{PLCP}[p]$ in tempo:*

$$\mathcal{O}(\log\log n) \tag{2.35}$$

Si è potuto migliorare e semplificare l'algoritmo di Bannai usato in MONI. Sfruttando le LCE query e avendo il testo T in memoria sotto forma di SLP, è possibile computare contemporaneamente sia i valori $\mathsf{MS}[i].\mathsf{pos}$ che i valori $\mathsf{MS}[i].\mathsf{len}$. Infatti, a differenza di quanto visto in MONI, si usa il risultato delle LCE query per scegliere una nuova posizione dopo un mismatch. In tal modo, si possono computare i valori $\mathsf{MS}[i].\mathsf{len}$, $\forall \in \{0, m-1\}$ contemporaneamente. Sfruttando, $\mathsf{MS}[i+1].\mathsf{len}$ e la lunghezza del risultati della LCE query è possibile tenere conto di eventuali overlap tra i match e computare $\mathsf{MS}[i].\mathsf{len}$. Alternativamente, per proseguire avendo un match tra $\mathsf{BWT}_T[q]$ e $P[i-1]$, il calcolo $\mathsf{MS}[i].\mathsf{len}$ avviene a partire da $\mathsf{MS}[i+1].\mathsf{len}$, incrementandolo di uno, grazie all'aggiunta del carattere a sinistra. Inoltre, come nel caso dell'algoritmo di Bannai, si ha il computo dei MEM in contemporanea a quello dei valori $\mathsf{MS}[i].\mathsf{len}$. Il riconoscimento della run a cui appartiene un certo indice e degli indici delle teste delle run avviene tramite l'uso di bitvector, che tengono traccia le teste di run per ogni simbolo dell'alfabeto. Infatti, $\forall\, \sigma \in \Sigma$, con Σ alfabeto in uso, si ha un bitvector (sparso) B_σ, lungo n (ovvero quanto il testo), tale che:

$$B_\sigma[i] = \begin{cases} 1 & \text{se } \mathsf{BWT}_T[i] = \sigma \wedge \mathsf{BWT}_T[i] \neq \mathsf{BWT}_T[i-1] \\ 0 & \text{altrimenti} \end{cases} \tag{2.36}$$

L'algoritmo 2.3 [8] riporta il calcolo completo dell'array delle matching statistics presente in PHONI, la cui complessità temporale è stimata in $\mathcal{O}(m \log\log n)$. Anche in questo caso, la stima asintotica è difficile da caratterizzare in modo corretto ed è fortemente dipendente dalle strutture dati in uso. Per ulteriori approfondimenti si rimanda al paper di riferimento [8].

Esempio 14. *Dati il testo T = mississippi\$ e il pattern P = miss, si vuole studiare il calcolo dell'array delle matching statistics sia con MONI che con PHONI. Si assume che $|T| = n$ e $|P| = m$. Avendo, in aggiunta in ultima colonna, i cinque bitvector relativi alle threshold di ogni simbolo, si hanno:*

Indice	SA_T	F_T	BWM_T	BWT_T	$B_\$$	B_i	B_m	B_p	B_s	\$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

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 ← MS[i].pos
5:      occ ← [ ]
6:      push(occ, p)
7:      while PLCP[p] ≥ j − i + 1 do
8:          p ← φ(p)
9:          push(occ, p)
10:     p ← φ^{-1}(MS[i].pos)
11:     while p ≠ null ∧ PLCP[p] ≥ j − i + 1 do
12:         push(occ, p)
13:         p ← φ^{-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 $\mathsf{LF}(\cdot)$ si intende il calcolo dell'LF-mapping.

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

Si inizia con l'algoritmo visto in MONI.
Dato $P[m-1] =$ *'s', ne segue che* $q = 9$*, utilizzando la notazione precedente e cercando l'ultima occorrenza di 's' in* BWT_T*. Si procede quindi con l'LF-mapping, avendo che* $\mathsf{LF}(9) = 11$*. A questo punto si calcola il valore di* $\mathsf{MS}[m-1].pos$*:*

$$\mathsf{MS.pos} = ??? \mathsf{SA}_T[11] = ???2$$

Si ha poi che $\mathsf{BWT}_T[11] =$ *'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 coda della precedente run di simboli 's', ovvero* $q = 9$*, avendo* $\mathsf{LF}(9) = 11$*, e si aggiorna l'array* $\mathsf{MS.pos}$*:*

$$\mathsf{MS.pos} = ?? \mathsf{SA}_T[11]2 = ??22$$

Si ha, a questo punto, che $\mathsf{BWT}_T[11] =$ *'i'* $= P[m-3] =$ *'i', ottenendo* $\mathsf{LF}(11) = 4$ *e aggiornando* MS*:*

$$\mathsf{MS.pos} = ? \mathsf{SA}_T[4]22 = ?122$$

Infine, avendo $\mathsf{BWT}_T[4] =$ *'i'* $\neq P[m-4] =$ *'m', si conclude il calcolo dell'array* MS *con l'ultimo l'LF-mapping,* $\mathsf{LF}(4) = 5$*, ottenendo:*

$$\mathsf{MS.pos} = \mathsf{SA}_T[5]122 = 0122$$

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

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

Secondo il lemma 2 visto per il calcolo dei MEM*, si ha 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 $q = \mathsf{select}_{P[m-1]}(1) = \mathsf{select}_{'s'}(1) = 2$*, ovvero ponendo q pari all'indice della prima occorrenza di* $P[m-1]$ *in* BWT_T*. Seguendo l'algoritmo si ottiene, dato che* $\mathsf{SA}_T[2] = 7$*:*

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

Si procede con l'LF-mapping: $\mathsf{LF}(2) = 8$. *Si ha che* $\mathsf{BWT}_T[8] = P[m-2]$ *e quindi, essendo* $\mathsf{SA}_T[8] = 6$, *si ottiene:*

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

Anche in questo caso, essendo $\mathsf{LF}(8) = 10$ *e* $\mathsf{BWT}_T[10] = P[m-3]$, *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 $\mathsf{LF}(10) = 3$, *si ha* $\mathsf{BWT}_T[3] \neq P[m-4]$. *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* $\mathsf{rank}_{'m'}(3) = 0$. *A questo punto si selezionano, tramite* $\mathsf{select}_{'m'}$, *gli indici della coda della run precedente di caratteri 'm' (che in questo caso non esiste e gli si assegna, per comodità, il valore 0) e della testa della run successiva:*

$$q' = \mathsf{select}_{'m'}(3) = 0 \qquad q'' = \mathsf{select}_{'m'}(4) = 4$$

Seguendo l'algoritmo si ha che:

$$l' = \min(3, |\mathsf{LCE}(\mathsf{SA}_T[0], 4)|) = \min(3, |\mathsf{LCE}(11, 4)|) = \min(3, 0) = 0$$

$$l'' = \min(3, |\mathsf{LCE}(\mathsf{SA}_T[4], 4)|) = \min(3, |\mathsf{LCE}(1, 4)|) = \min(3, 4) = 3$$

Avendo $l'' \geq l'$ *si aggiorna* MS *di conseguenza, avendo* $\mathsf{SA}_T[q''] = \mathsf{SA}_T[4] = 1$*:*

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

Infine, sempre per il lemma 2, si ha che $P[0, 4-1] = P[0, 3]$ *è un* MEM *di* P *in* T.

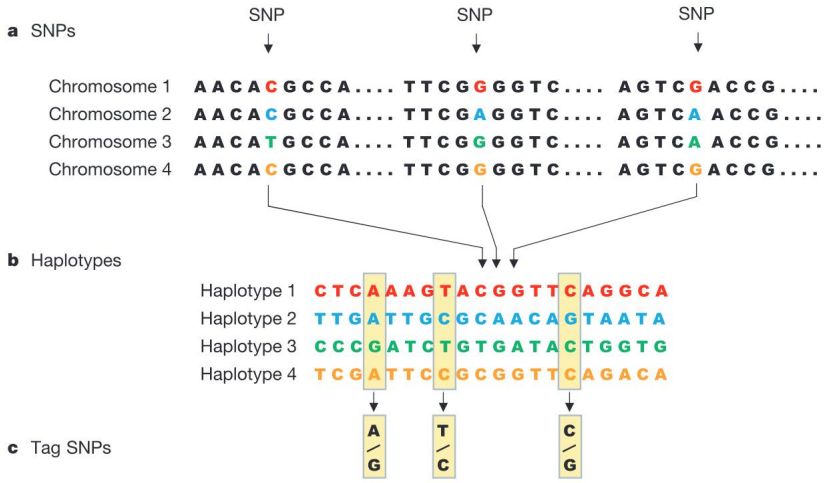


Figura 2.1: Schema di ottenimento del pannello di aplotipi.

2.6 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 dati efficiente per la memorizzazione e l'interrogazione di pannelli di aplotipi.
La costruzione di tali pannelli avviene tramite il riconoscimento delle varianti di un singolo nucleotide tra le sequenze genomiche di diversi individui, ovvero dei cosiddetti Single-Nucleotide Polymorphism (SNP). Ogni variante, 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 [31], la costruzione parte da diversi sequenziamenti (nell'immagine relativi a diversi cromosomi ma il procedimento è uguale partendo da diversi individui) grazie a cui si identificano le varianti alleliche. Da queste ultime si costruiscono gli aplotipi, utilizzati per estrarre i cosiddetti tag SNP, ovvero le possibili alternative per una certa variante allelica. Questi ultimi, normalmente 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 **genotipo**. Formalmente, si considera un pannello X di M aplotipi x_i, con $i = 0, \ldots, M-1$, con N siti, indicizzati tramite $k = 0, \ldots, N-1$, 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 $0 \prec 1$, avendo la

sostituzione dei tag SNPs, per un certo sito, con tale alfabeto. Ne segue che:

$$x_i[k] = \{0, 1\} \tag{2.37}$$

Prima di proseguire con la trattazione, è bene specificare alcuni formalismi utilizzati:

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

$$x_i[k_1, k_2) = x_j[k_1, k_2) \tag{2.38}$$

- 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:

$$(k_1 = 0 \vee x_i[k_1 - 1] \neq x_j[k_1 - 1]) \wedge (k_2 = N \vee x_i[k_2] \neq x_j[k_2]) \tag{2.39}$$

- comparando una sequenza (ovvero un aplotipo esterno) z a un pannello di aplotipi X, si ha che un match localmente massimale è un set-maximal exact match (SMEM) di z contro $x_i \in X$, iniziante alla colonna k_1 e terminante alla colonna $k_2 - 1$, sse non si ha alcun altro match localmente massimale di z con un altro x_j che include ed estende l'intervallo $[k_1, k_2)$. La sequenza z può avere uno SMEM, tra k_1 e $k_2 - 1$, con più di un aplotipo del pannello. Per praticità, si è introdotta la definizione di SMEM usando un aplotipo esterno, ma tale definizione può essere riferita anche a due righe interne al pannello X

Si noti che il match tra due sequenze nella PBWT è tale sse entrambi iniziano e terminano nella stessa colonna. Questo vincolo, da cui deriva il termine "posizionale" e che 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, a ogni colonna k, sul riordinamento lessicografico delle sequenze di aplotipi, nel dettaglio sull'ordinamento inverso dei prefissi terminanti in colonna $k - 1$. I valori presenti in colonna k, dopo il riordinamento, sono quelli 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 inversi alla k-esima colonna permette di identificare i match con maggior facilità in quanto aplotipi con suffisso comune (o prefisso comune inverso)

saranno "virtualmente" in posizioni consecutive a ogni riordinamento della trasformata.
Il calcolo di tutti i riordinamenti non presenta difficoltà dal punto di vista computazionale. Infatti, conoscendo l'ordinamento in colonna k, si può derivare facilmente quello in colonna $k+1$, studiando solo i valori riordinati alla colonna precedente. Si ha infatti un ordinamento stabile ad ogni colonna.

Definizione 21. *Dato un pannello X e un indice di colonna k, si definisce il* ***prefix array*** *a_k come una permutazione degli indici $\{0, \ldots, M-1\}$ tale per cui $a_k[i] = j$ sse x_j è l'i-esimo aplotipo di X nell'ordinamento inverso dei prefissi ottenuto alla colonna k.*

Data questa definizione, si ha che la matrice PBWT si ottiene direttamente usando, 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 l'i-esima sequenza nell'ordinamento inverso necessario a ottenere la colonna k della matrice PBWT, come:

$$y_i^k = x_{a_k[i]} \tag{2.40}$$

In termini di colonne, con la notazione $y^k[k]$ si identifica l'intera colonna k della matrice PBWT mentre l'elemento i-esimo di tale colonna è indicizzato da $y_i^k[k]$.
L'ordinamento degli elementi per a_{k+1} si ottiene a partire dall'ordinamento in a_k, considerando i valori $y_i^k[k]$, $\forall i \in \{0, M-1\}$, 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 prefissi inversi comuni più lunghi tra prefissi adiacenti nei riordinamenti.

Definizione 22. *Si definisce* ***divergence array*** *l'array d_k, tale che $d_k[i]$ è l'indice della colonna iniziale del match massimale a sinistra, terminante a destra in colonna $k-1$, tra l'i-esimo aplotipo e il suo precedente nell'ordinamento ottenuto per la colonna k-esima. Formalmente, dato $i > 0$, si definisce $d_k[i]$ come il più piccolo j tale che:*

$$y_i^k[j, k) = y_{i-1}^k[j, k) \tag{2.41}$$

Ne segue che:

$$y_i^k[k-1] \neq y_{i-1}^k[k-1] \implies d_k[i] = k \tag{2.42}$$

Per definizione, non avendo una riga precedente con cui effettuare il confronto, si ha che $d_k[0] = k$.

Si noti che, per le proprietà del divergence array, l'inizio di qualsiasi match massimale, terminante in colonna k, tra qualsiasi y_i^k e y_j^k, con $i < j$, è dato da:

$$\max_{i<m\leq j} d_k[m] \tag{2.43}$$

Tabella 2.4: Tabella di confronto tra le strutture dati relative alla BWT e alla PBWT.

BWT	PBWT
SA_T	a_k
ISA_T	α_k
LCP_T	d_k o l_k

Al posto del divergence array si può usare anche una variante "posizionale" del Longest Common Prefix array.

Definizione 23. *Si definisce Reverse Longest Common Prefix (*RLCP*) 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 che:*

$$l_k[i] = k - d_k[i] \tag{2.44}$$

Fatte queste premesse possiamo quindi fornire una definizione formale di PBWT.

Definizione 24. *Dato un pannello $X = \{x_0, x_1, \ldots, x_{M-1}\}$, di M aplotipi con ciascuno N siti, si definisce* PBWT *di X una collezione di $N+1$ coppie di array (a_k, d_k), con $0 \leq k \leq N$.*

La procedura 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 noti che il costo della costruzione dei due insiemi di array è:

$$\mathcal{O}(NM) \tag{2.45}$$

Ai fini della trattazione dell'algoritmo di computo degli SMEM per un aplotipo esterno, ricordiamo un'ulteriore definizione, analoga a quanto visto con il suffix array e la sua permutazione inversa.

Definizione 25. *Si definisce α_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 denotare alcune forti correlazioni, esplicitate in tabella 2.4, che sussistono tra BWT e PBWT (e le rispettive varianti run-length encoded), fattori chiave nello sviluppo di questa tesi.

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

Esempio 15. *Dato il seguente pannello X si vuole 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 è la sesta colonna del pannello così riordinato. Ne segue che, secondo il riordinamento mostrato nel seguente pannello, a_6 è la colonna degli indici, già riordinati, mentre d_6 è specificato tramite le righe in verde:

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

Si ha, nel complesso:

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

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

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

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

Permutando X con tutti gli a_k, si ottiene la seguente matrice PBWT*:*

PBWT	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

2.6.1 Set-maximal exact match 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 a X più lunghi di una lunghezza minima L e uno per la ricerca di tutti gli SMEM interni a X, in tempo lineare.
Di interesse per questa tesi è il cosiddetto **algoritmo 5**, usato per computare tutti gli SMEM tra il panello X e un aplotipo esterno z, assumendo che z abbia lo stesso numero di siti del pannello. Quest'ultimo vincolo è dovuto al fatto 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. Invece, l'intervallo $[f_k, g_k) \subseteq [0, \ldots, M)$ identifica il sotto-intervallo di a_k contenente gli indici degli aplotipi che presentano tale match. Si noti come viene ripresa 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. In questo caso, però, tratta di una "forward search".

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 uno* SMEM*, iniziante in colonna e_k e terminante il colonna k, tra la query z e tutte le righe del pannello indicizzate dai valori compresi nell'intervallo $[f_k, g_k)$ in a_k, sse:*

$$z[e_k, k) = y_i^k[e_k, k) \wedge z[e_k - 1] \neq y_i^k[e_k - 1], \forall\, i \text{ t.c. } f_k \leq i < g_k \tag{2.46}$$

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

Bisogna capire come aggiornare e_k, f_k e g_k passando dalla colonna k alla colonna $k + 1$. Per calcolare $[f_{k+1}, g_{k+1})$, si procede esattamente come visto per la backward search nella BWT, selezionando il sottointervallo non stretto 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 si hanno 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 si può concludere che quelli terminanti in colonna k erano effettivamente SMEM. In questo secondo caso, bisogna aggiornare e_{k+1}, ottenendo i relativi nuovi valori f_{k+1} e g_{k+1}, al fine di trovare la colonna da cui parte lo SMEM successivo e le righe del pannello per le quali si ha tale SMEM.
A questo punto, bisogna capire come funziona la variante del backward-step introdotta per la BWT, avendo che nella PBWT si può parlare di forward-step/mapping. Tale funzione, guidata 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 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 valori pari a 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 valori pari a 0 prima dell'indice i in $y^k[k]$
3. l'array v_k tale per cui, alla colonna k-esima, $v_k[i] = j$ sse j è il numero di occorrenze di valori pari a 1 prima dell'indice i in $y^k[k]$

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 l'array c basterebbe tenere traccia del numero di 0 incontrati nella colonna k-esima o sfruttare direttamente l'array u).
Sfruttando i valori di questi tre array possiamo quindi effettuare lo step/mapping alla colonna successiva, definito da una funzione:

$$w_k : \{0, \ldots, M\} \times \Sigma \to \{0, \ldots, M\} \tag{2.47}$$

tale per cui:

$$w_k(i, \sigma) = \begin{cases} u_k[i] & \text{se } \sigma = 0 \\ v_k[i] + c[k] & \text{se } \sigma = 1 \end{cases} \tag{2.48}$$

Tale funzione è rappresentabile in pseudocodice come nell'algoritmo 2.5.
Risulta interessante notare che:

$$a_{k+1}\left[w_k\left(i, y_i^k[k]\right)\right] = a_k[i] \tag{2.49}$$

Quindi, tale mapping permette di "seguire" una determinata riga all'interno delle permutazioni dettate dai valori dei prefix array.

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. *Si riprende 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, con $k = 5$ e $i = 2$, che:

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

Avendo:

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

Avendo anche che:

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

Si conferma come con tale funzione si possa seguire la riga 17, capendo da quale riga della colonna permutata precedente arrivi.

Pensando alla permutazione inversa del prefix array, si ottiene un altro risultato interessante, che lega tale permutazione alla funzione w_k:

$$\alpha_{k+1}[i] = w_k(\alpha_k[i], x_i[k]) \tag{2.50}$$

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

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

Come volevasi dimostrare.

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

$$f_{k+1} = w_k(f_k, z[k]) \tag{2.51}$$

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:

$$g_{k+1} = w_k(g_k, z[k]) \tag{2.52}$$

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

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

2. $f_{k+1} = g_{k+1}$. In questo caso non ci si sono match, inizianti in e_k e terminanti in k, che sono anche estendibili in $k+1$. Bisogna, quindi, annotare i match terminanti in $k-1$, ovvero gli SMEM con le righe indicizzate 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 l'aplotipo z si trova virtualmente o subito prima o subito dopo l'insieme di aplotipi indicizzati da $[f_k, g_k)$ su a_k, in colonna k secondo il riordinamento inverso. Di conseguenza, si può inferire che, avendo $f_{k+1} = g_{k+1}$:

$$y^{k+1}_{f_{k+1}-1} \prec z \prec y^{k+1}_{f_{k+1}} \tag{2.53}$$

Ne segue direttamente che:

$$e_{k+1} \leq d_{k+1}[f_{k+1}] \tag{2.54}$$

Il nuovo indice di partenza del match sarà almeno nella colonna indicata da $d_{k+1}[f_{k+1}]$, essendo esso calcolato tra $y^{k+1}_{f_{k+1}-1}$ e $y^{k+1}_{f_{k+1}}$, fra le quali sequenze è virtualmente compresa la query z.
Quindi, si considera come punto di partenza:

$$e_{k+1} = d_{k+1}[f_{k+1}] - 1 \tag{2.55}$$

Studiando $z[e_{k+1}]$, grazie al fatto che, per la nozione di divergence array e di ordinamento dei prefissi inversi con $0 \prec 1$:

$$y^{k+1}_{f_{k+1}-1}[e_{k+1}] = 0 \neq y^{k+1}_{f_{k+1}}[e_{k+1}] = 1 \tag{2.56}$$

si hanno due casi possibili:

   (a) se tale valore è 0, allora, z ha un match migliore con $y^{k+1}_{f_{k+1}-1}$ rispetto che con $y^{k+1}_{f_{k+1}}$. Si aggiorna, quindi, e_{k+1}, decrementandolo fino a che si ha match tra $z[e_{k+1}-1]$ e $y^{k+1}_{f_{k+1}-1}[e_{k+1}-1]$. Infine, si decrementa f_{k+1} fino a che $d_{k+1}[f_{k+1}] \leq e_{k+1}$, trovando quelle righe per le quali 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 $y^{k+1}_{g_{k+1}}$ presenta un mismatch in colonna $k+1$

   (b) se tale valore è 1, allora, z ha un match migliore con $y^{k+1}_{f_{k+1}}$ rispetto che con $y^{k+1}_{f_{k+1}-1}$. Si aggiorna, quindi, e_{k+1}, decrementandolo fino a che si ha match tra $z[e_{k+1}-1]$ e $y^{k+1}_{f_{k+1}-1}[e_{k+1}-1]$. 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 è possibile 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 $y^{k+1}_{f_{k+1}}$ presenta un mismatch in colonna $k+1$

In termini di inizializzazione, per permettere il funzionamento dell'algoritmo, si hanno: $f_0 = g_0 = e_0 = 0$.

Esempio 18. *Mostrare un esempio completo di esecuzione richiederebbe troppo spazio quindi ci si limita a mostrare cosa succede nel caso in cui, a un certo punto dell'esecuzione,* $f_{k+1} = g_{k+1}$.
Assumendo il pannello e la matrice PBWT *visti all'esempio 15, con una query* z*, si identificano i seguenti* SMEM*:*

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
z	0	1	0	0	1	0	1	0	0	0	1	1	1	0	1

Assumendo di essere in colonna $k = 6$ *e avendo, dopo i calcoli fatti in colonna* $k = 5$*, si ha che:*

$$f_6 = 6, \qquad g_6 = 10, \qquad e_6 = 0$$

Tali valori segnalano che, a partire dalla colonne 0 *fino alla colonna* $6 - 1 = 5$*, si hanno le righe nel range* $[6, 10)$ *di* a_6 *che presentano un match con* $z[0, 5]$*. Tali*

righe sono, nel dettaglio, quelle indicizzate $\{8, 11, 12, 13\}$.
Bisogna quindi computare f_7 *e* g_7. *Assumendo che* $z[6] = 1$ *e che:*

$$y^6 = 00000000000100000000, \;\; c[6] = 19$$

si calcolano:

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

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

Avendo $f_7 = g_7$ *si procede, in primis, annotando gli* SMEM *terminanti in* $k = 5$.
Seguendo l'algoritmo si ha un primo aggiornamento di e_{k+1}, *che, avendo in memoria* d_7, *viene posto pari a:*

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

Questo viene fatto in quanto l'aplotipo z *si trova o subito prima o subito dopo il blocco di aplotipi* $[f_k, g_k)$.
Essendo, inoltre, $z[e_7] = z[6] = 1$, *si procede aggiornando* g_7 *e tenendo fisso* f_7, *avendo* $z[6] = 1$. *Si calcola* g_7*:*

$$g_7 = f_7 + 1 = 20$$

A questo punto, si segue la riga 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 presentanti un match con* $z[e_7, 6]$.

Secondo i calcoli di Durbin, il suo algoritmo 5, consultabile all'algoritmo 2.6, ha complessità:

$$\mathcal{O}(N + c) \tag{2.57}$$

Tale risultato è così stimato in quanto si ritiene che il numero di accessi ai cicli `while` interni sia limitato dalla costante c, rappresentante il numero di SMEM. 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 [32], 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 precisata una nuova misura per la complessità dell'algoritmo ma solo che la stima di Durbin è empiricamente accettabile come *caso medio*:

$$Avg. \;\; \mathcal{O}(N + c) \tag{2.58}$$

In ogni caso, una soluzione naïve, impiegherebbe tempo:

$$\mathcal{O}(N^2 M) \tag{2.59}$$

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 per il calcolo degli SMEM con aplotipo esterno.

```
1:  function FIND_SET_MAXIMAL_MATCHES_WITH_Z(z)
2:      e ← 0,  f ← 0,  f ← 0
3:      for k ← 0 to N do
4:          e, f, g ← Update_Matches(k, z, e, f, g)
5:
6:  function UPDATE_MATCHES(k, z, e, f, g)
7:      f' ← w(k, f, z[k])
8:      g' ← w(k, g, z[k])
9:      if f' < g' then              ▷ se k è N − 1 report degli SMEM da e_k a N − 1
10:         e' ← e_k
11:     else                                          ▷ report degli SMEM da e_k a k
12:         e' ← d_{k+1}[f'] − 1
13:         if z[e'] = 0 and f' > 0 then
14:             f' ← g' − 1
15:             while z[e' − 1] = y^{k+1}_{f'}[e' − 1] do e' ← e' − 1
16:             while d_{k+1}[f'] ≤ e' do f' ← f' − 1
17:         else
18:             g' ← f' + 1
19:             while z[e' − 1] = y^{k+1}_{f'}[e' − 1] do e' ← e' − 1
20:             while g' < M and d_{k+1}[g'] ≤ e' do g' ← g' + 1
21:     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, tutti gli array necessari a costruire la PBWT e a permettere di computare la funzione $w(i, \sigma)$, comportando un'incremento dal punto di vista temporale per studiare più query z.
Ricapitolando, per poter eseguire l'algoritmo 5, è necessario avere in memoria con random access:

- il pannello X, di dimensione NM

- l'insieme dei prefix array a, di dimensione NM
- l'insieme dei divergence array d, di dimensione NM
- gli array u_k e v_k, per ogni colonna k, complessivamente di dimensione $2NM$
- l'array c, di dimensione N

Quindi, si ha una complessità in spazio pari a:

$$\mathcal{O}(NM) \tag{2.60}$$

Nel dettaglio, Durbin stesso ha proposto una stima quantitativa della memoria richiesta, ovvero[3]:

$$13NM \text{ byte} \tag{2.61}$$

Per poter capire meglio la problematica conseguente a tali richieste di spazio, prendiamo, ad esempio, un pannello di dimensioni $N = 6.196.151$ e $M = 4.908$. Ne segue che si necessitano 368GB di memoria. Una stima sperimentale di tale richiesta di memoria può essere confermata con l'esecuzione dell'implementazione ufficiale della PBWT[4]. Infatti, monitorando con `time` il picco di memoria durante l'esecuzione, si ha che esso corrisponde a 369GB.
L'alto uso di memoria richiesto dall'algoritmo 5 è stata la motivazione principale per cui si è sviluppata, in questa tesi magistrale, una versione run-length encoded della PBWT che permettesse il calcolo degli SMEM con un aplotipo esterno, in modo efficiente dal punto di vista della memoria richiesta.

2.6.2 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 delle prospettive futuri, è bene citarne i principali.

PBWT multiallelica

La prima variante che si introduce è la **PBWT multiallelica** (mPBWT), proposta da Naseri et al. [33] nel 2019. Questo lavoro estende la PBWT di Durbin, generalizzandola a un alfabeto arbitrario.

[3] `https://github.com/richarddurbin/pbwt/blob/0de8d02df1b77146ded81e9e196991fdab520767/pbwtMatch.c#L252`

[4] `https://github.com/richarddurbin/pbwt`

Dal punto di vista delle motivazioni biologiche, questa soluzione risulta fondamentale in quanto gli studi riportano come, nell'uomo, la presenza di siti triallelici sia sotto stimata [44] [45], oltre che per lo studio di specie multialleliche (soprattutto nel mondo vegetale).
Dal punto di vista algoritmico, si sono generalizzati i concetti di c, u_k e v_k visti nella PBWT, ottenendo un vero e proprio FM-index in grado di lavorare su alfabeto arbitrario Σ, con conseguente forte aumento dello spazio richiesto in memoria. Invece, dal punto di vista della complessità temporale, si ha che le stime asintotiche degli algoritmi devono tenere conto della grandezza dell'alfabeto stesso, considerando che questo fatto non comporta particolari problematiche dal punto di vista dei tempi di calcolo, essendo l'alfabeto di dimensioni ridotte. Infatti, le complessità temporali della mPBWT sono moltiplicate di un fattore $t = |\Sigma|$. Se tale valore è assunto costante a inizio computazione, la complessità temporale non subisce variazioni considerevoli poiché difficilmente $t >> 2$.

PBWT con struttura LEAP

Sempre nel 2019, Naseri et al. [34] hanno proposto una variante della PBWT che permetta il calcolo di qualsiasi match, tra un aplotipo esterno e un pannello, di lunghezza maggiore uguale a una lunghezza arbitraria L. Tale algoritmo è stato nominato *PBWT-query*. Inoltre, nello stesso articolo, hanno proposto una struttura dati aggiuntiva, detta *LEAP* (Linked Equal/Alternating Position), che ottimizzava i tempi dell'algoritmo per la PBWT-query, ottenendo l'algoritmo *L-PBWT-query*, al costo della memorizzazione di otto array aggiuntivi che permettono 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. Da un punto di vista computazionale, si noti che la complessità in tempo dell'algoritmo per la PBWT query, con match di lunghezza minima L, è:

$$\mathcal{O}(N + c(R - L + 1)) \tag{2.62}$$

con:

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

In merito alla complessità in tempo dell'algoritmo L-PBWT-query si ha che, al costo di $8NM$ interi aggiuntivi in memoria, con N e M dimensioni del pannello, essa è proporzionale a:

$$\mathcal{O}(N + c) \tag{2.63}$$

PBWT dinamica

Sanaullah et al. [32], nel 2021, hanno proposto la **Dynamic PBWT** (dPBWT), al fine di superare le limitazioni imposte dalle strutture statiche usate nella PBWT di Durbin. Si è quindi pensato di sostituire l'uso degli array con l'uso di linked list, ovvero strutture dati dinamiche. Questo ha portato all'aggiornamento efficiente della matrice PBWT, all'aggiunta di un nuovo aplotipo nel pannello o alla rimozione di uno già presente.
Dal punto di vista computazionale, è interessante notare come le implementazioni degli algoritmi di Durbin presentano la medesima complessità asintotica di quelli basati sull'uso di tali strutture dinamiche. Ad esempio, la creazione della dPBWT richiede tempo:

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

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

$$Avg.\ \mathcal{O}(N) \tag{2.65}$$

PBWT con wildcard

La tematica dei dati mancanti è una tematica aperta in bioinformatica. I sequenziatori presentano un range d'errore dall'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 possa accadere che non si sappia quale sia l'allele corretto per un individuo, riferendosi a un certo sito.
Nel 2020, Williams e Mumey [35] hanno proposto l'uso della PBWT **con wildcard** al fine di disegnare un algoritmo in grado di calcolare determinati match interni a un pannello biallelico con dati mancanti, rappresentati come wildcard mediante il simbolo "*" (avendo quindi, come alfabeto, $\Sigma = \{0, 1, *\}$).
In termini computazionali, gli autori sono riusciti a formulare un algoritmo in grado di calcolare questi T match interni al pannello in tempo:

$$\mathcal{O}(NMT) \tag{2.66}$$

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 di una malattia.
A tal fine, nel 2020, Rubinacci et al. [36] hanno proposto **IMPUTE5**, un metodo basato sulla PBWT per la genotype imputation, in grado di studiare, in ottica GWAS, pannelli di grandi dimensioni.

2.6.3 Una prima proposta run-length encoded

A fine 2021, Gagie et al. [37] hanno iniziato a teorizzare una variante run-length encoded della PBWT, cercando di basarsi sui risultati già ottenuti sulla BWT 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 π_k, $\forall\, 0 \leq k < N$. Ipotizziamo 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:

$$i, \pi_0(i), \pi_1(\pi_0(i)), \ldots, \pi_{N-1}(\cdots(\pi_1(\pi_0(i)))\cdots) \tag{2.67}$$

Il punto fondamentale, relativo al run-length encoding, si ritrova nel fatto che l'autore asserisce:

Notice π_k can be stored in space proportional to the number of runs in the kth column of the PBWT *...*

Nell'articolo si propone 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:

$$\pi_k(p) = \begin{cases} p - v_k[p] & \text{if } y_p^k[k] = 0 \\ c[k] + v_k[p] - 1 & \text{if } y_p^k[k] = 1 \end{cases} \tag{2.68}$$

- l'indice della run contenente il simbolo $\pi_k(p)$ nella colonna $k{+}1$ della matrice PBWT
- un booleano che specifica se la prima run è composta da simboli $\sigma = 0$. Si noti che tale valore non è esplicitato nell'articolo, ma risulta necessario per ottenere il simbolo corrispondente a una qualsiasi run

Il paper presenta anche il metodo per l'estrazione della i-esima riga. Inizialmente, si cerca la riga relativa alla run nella prima "tabella", con indice di testa p, contenente l'indice i. Si noti che tale "tabella", relativa alla colonna $k = 0$, non presenta permutazioni e quindi l'indice i del pannello è anche l'indice i della matrice PBWT. Si può calcolare, quindi, la permutazione per l'indice i (alla prima operazione si avrà $k = 0$):

$$\pi_k(i) = \pi_k(p) + i - p \tag{2.69}$$

Si identifica, 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 $k = 0$). Infine, si estrae il simbolo relativo a tale run. Ripetendo la procedura per ogni colonna k, a partire dal computo della permutazione, si può calcolare la riga i-esima del pannello.
Inoltre, vengono proposte ulteriori ottimizzazioni, basate sul metodo detto *fractional cascading*. Con tale rappresentazione, si riesce a ridurre il numero di run che devono essere scansionate, al costo di una maggior richiesta di spazio. Infatti, aumentando il numero totale di righe in tutte le tabelle di un fattore al più $\left(1 + \frac{1}{d}\right)$, è possibile garantire che si avranno al più d iterazioni, in ogni tabella, per ottenere l'estrazione del simbolo desiderato. Per ulteriori dettagli si rimanda al paper di riferimento [37].

Esempio 19. *Supponendo di voler ricostruire la riga $i = 9$ e assumendo la seguente matrice* PBWT*, con in rosso i simboli appartenenti alla riga 9:*

X	00	01	02	03	04	05	06	07	08	09	10	11
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 le seguenti tabelle [37]:

	table 00			table 01			table 02			table 03			table 04			table 05		
0	0	9	3	0	11	4	0	0	0	0	0	0	0	0	0	0	14	2
1	8	0	0	4	0	0	2	15	2	11	19	2	11	17	5	2	0	0
2	9	17	5	7	15	4	3	2	0	12	11	1	14	11	5	3	16	2
3	11	1	0	9	3	2	4	16	2							5	1	0
4	16	19	5	10	17	4	8	3	0							8	18	2
5	17	6	1	13	4	3										10	4	2

	table 06			table 07			table 08			table 09			table 10			table 11
0	0	0	0	0	0	0	0	7	4	0	13	1	0	17	2	0
1	2	19	3	2	16	4	7	0	0	2	0	0	3	0	0	6
2	3	2	1	3	2	0	10	14	7	3	15	1				7
3				17	17	4	12	3	2	5	1	0				
4							16	16	7	7	17	1				
5										9	3	1				
6										10	19	1				
7										11	4	1				

Nelle tabelle, in rosso, si hanno le varie $\pi_k(i)$, ottenute, se necessario, iterando a partire dalle $\pi_k(p)$, segnalate in azzurro.
Numericamente si hanno i seguenti calcoli, ovvero i diversi $\pi_j(i) = \pi_j(p) + i - p$ relativi alle permutazioni in colonna k, per l'estrazione della riga 9:

- $\pi_0(9) = 17 + 9 - 9 = 17$
- $\pi_1(17) = 4 + 17 - 13 = 8$
- $\pi_2(8) = 4 + 8 - 8 = 3$
- $\pi_3(3) = 0 + 3 - 0 = 3$
- $\pi_4(3) = 0 + 3 - 0 = 3$
- $\pi_5(3) = 16 + 3 - 3 = 16$
- $\pi_6(16) = 2 + 16 - 3 = 15$
- $\pi_7(15) = 2 + 15 - 3 = 14$
- $\pi_8(14) = 3 + 14 - 12 = 5$
- $\pi_9(5) = 1 + 5 - 5 = 1$
- $\pi_{10}(1) = 17 + 1 - 0 = 18$

Sfruttando il valore booleano (non rappresentato nelle tabelle) che indica con che simbolo inizia una colonna della matrice PBWT *e sapendo che si alternano run con simboli $\sigma = 0$ e $\sigma = 1$ (pannello binario), si può ricostruire la riga 9 del pannello originale:* $x_9 = 100001000011$.

Si segnala che, nel paper, non vengono specificati metodi per effettuare query a questa struttura dati, ma viene solo indicata la possibilità di interrogare tale struttura a tabelle.

Capitolo 3

Metodi

In questo capitolo verranno illustrate le metodologie usate in questa tesi, trattando tutte le soluzioni che hanno portato alla costruzione di diverse varianti della RLPBWT.
Verranno discussi gli usi delle singole componenti, le stime asintotiche sia in tempo che in spazio, gli algoritmi di costruzione e di successivo calcolo degli SMEM e i pro/contro delle varie strutture dati ottenibili da tali componenti.

3.1 Perché la compressione run-length

Prima di proseguire con la spiegazione delle varianti della RLPBWT, è bene dare un'ulteriore motivazione al perché si sia ritenuto utile sviluppare una variante run-length encoded della PBWT.
Una motivazione la si ha citando direttamente il paper di Durbin sulla PBWT [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, consecutivi nel riordinamento inverso (che è fino alla colonna $k-1$), è molto probabile presentino lo stesso allele nella colonna k. Ne segue che, all'interno della matrice PBWT, si hanno lunghe run di simboli $\sigma = 0$ e di simboli $\sigma = 1$. Si ottiene il medesimo risultato avuto con la BWT, dove caratteri uguali è molto probabile siano posti in posizioni consecutive all'interno della trasformata stessa. Si hanno le medesime premesse che hanno portato alla RLBWT, considerando, inoltre, che 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 che, per una certa colonna della matrice PBWT, il numero di run sia molto minore della lunghezza della colonna stessa, si deduce che l'uso della compressione run-length possa comportare una riduzione significativa della memoria necessaria.

3.2 Matching Statistics per la RLPBWT

Bisogna 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, con M individui/righe aventi N siti/colonne, e un aplotipo esterno/pattern z, tale che $|z| = N$, si definisce matching statistics di z su X un array* MS *di coppie* $(\mathsf{row}, \mathsf{len})$*, di lunghezza N, tale che (avendo che x_i indica l'i-esima riga del pannello X):*

- $x_{\mathsf{MS}[i].\mathsf{row}}[i - \mathsf{MS}[i].\mathsf{len} + 1, i] = z[i - \mathsf{MS}[i].\mathsf{len} + 1, i]$*, ovvero si ha che l'aplotipo esterno ha un match, lungo* $\mathsf{MS}[i].\mathsf{len}$*, terminante in colonna i, con la riga* $\mathsf{MS}[i].\mathsf{row}$*-esima del pannello*
- $z[i - \mathsf{MS}[i].\mathsf{len}, i]$ *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*

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/righe e N siti/colonne, un aplotipo esterno/pattern z, tale che $|z| = N$, e il corrispondente array di matching statistics* MS *si ha che $z[i - l + 1, i]$ presenta uno* SMEM *di lunghezza l con la riga* $\mathsf{MS}[i].\mathsf{row}$*-esima del pannello X sse:*

$$\mathsf{MS}[i].\mathsf{len} = l \wedge (i = N - 1 \vee \mathsf{MS}[i].\mathsf{len} \geq \mathsf{MS}[i + 1].\mathsf{len}) \tag{3.1}$$

Si vedrà in sezione 3.3.5 come calcolare, a partire da tali SMEM*, tutte le righe del panello per le quali si ha il medesimo* 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**

Dati 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:

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

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 coppia di valori $\mathsf{MS}[i]$, $\forall i \in [0, N)$:

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

Per convenzione, si ha che, nella prima colonna, l'iterazione parte dall'ultima riga. In modo analogo, qualora si abbia una colonna k di un solo carattere, non corrispondente a quello della query, si sceglie l'ultima riga della colonna successiva, dalla quale si riprende il calcolo dell'array delle matching statistics, dopo aver memorizzato $\mathsf{MS}[k].\mathsf{row} = M$ (un valore sentinella non esistendo la riga di indice M) e $\mathsf{MS}.\mathsf{len}[k] = 0$.
In termini più "pratici", il calcolo dell'array MS avviene nel seguente modo:

- si parte da una riga arbitraria i della prima colonna
- se si ha $x_i[0] = z[0]$, si procede salvando $\mathsf{MS}[0].\mathsf{row} = i$
- mentre se si ha $x_i[0] \neq z[0]$, si seleziona l'ultima riga della run precedente o la prima riga della run successiva a quella a cui appartiene la riga i. Tale riga verrà salvata come $\mathsf{MS}[0].\mathsf{row} = j$ e da essa si proseguirà l'esecuzione dell'algoritmo
- 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 precedenti

Per calcolare i valori $\mathsf{MS}[i].\mathsf{len}$, si hanno due soluzioni (approfondite in seguito), che rispecchiano quanto visto con MONI [7] e PHONI [8] per la RLBWT:

1. si possono usare le threshold (definite in seguito), per capire quale nuova riga selezionare in caso di mismatch. In tal caso, i valori MS$[i]$.len devono essere calcolati successivamente a quelli MS$[i]$.row, tramite random access al panello

2. si possono usare le LCE query, per capire quale nuova riga selezionare in caso di mismatch. In tal caso, il calcolo dei valori MS$[i]$.len avviene in contemporanea a quelli MS$[i]$.row

Esempio 20. *Per vedere un esempio di calcolo dell'array* MS*, si riprende l'esempio 18, con i seguenti* SMEM *tra la query z e il pannello X:*

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
z	0	1	0	0	1	0	1	0	0	0	1	1	1	0	1

In tal caso l'array MS *è, iniziando il computo dalla riga 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

In MS *si possono riconoscere i vari* SMEM*, la cui colonna di fine è segnalata con lo stesso colore con cui si rappresentano gli* SMEM *nel pannello.*

3.3 Componenti per la RLPBWT

Dovendo utilizzare 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 si sarebbero potute smentire in fase sperimentale. Una sola caratterizzazione asintotica avrebbe potuto comportare sottostime e sovrastime, sia in termini di tempo che di spazio. Quindi, si è deciso di sviluppare diverse varianti della RLPBWT.
Si è deciso di suddividere le stesse in componenti che, adeguatamente assemblate, permetteranno la costruzione di strutture dati composte atte al calcolo degli SMEM. Tali componenti, che verranno dettagliate in seguito, sono:

- la componente per il mapping tra la colonna k-esima e la colonna $k+1$ della matrice PBWT, ovvero, riprendendo la notazione di Durbin, le strutture run-length encoded per gli array c, u_k e v_k. Nel dettaglio, si hanno due varianti:
    1. mapping tramite intvector compressi (`MAP-INT`)
    2. mapping tramite bitvector sparsi (`MAP-BV`)
- la componente per la memorizzazione delle threshold, proporzionali al numero di run. Anche in questo caso si hanno due varianti, corrispondenti alle due varianti per il mapping:
    1. threshold con intvector compressi (`THR-INT`)
    2. threshold con bitvector sparsi (`THR-BV`)
- la componente per la memorizzazione compatta della permutazione a ogni colonna della matrice PBWT, tramite i sample 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 LCE query (`LCE`)
- la componente per l'intero reverse longest common prefix array (`RLCP`), già descritto nella sezione 2.6
- la componente per permettere il calcolo delle funzioni φ e φ^{-1} (`PHI`)

3.3.1 Componente per il mapping

La prima componente che si descrive è quella relativa al mapping tra una colonna e la sua successiva nella matrice PBWT. Bisogna studiare come effettuare il forward-step/mapping nella RLPBWT e memorizzare, per ogni colonna k e in modo proporzionale al numero di run della stessa, le informazioni necessarie a ottenere i medesimi risultati della funzione $w(i, \sigma)$ (secondo la notazione di Durbin).

Mapping con intvector compressi

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 [37]. Riprendendo quanto descritto al termine della sezione 2.6, si è deciso di memorizzare gli indici delle teste di run, ma quest'informazione non è sufficiente per poter sapere se una run sia composta da simboli $\sigma = 0$ o simboli $\sigma = 1$. Essendo lo studio limitato a pannelli costruiti su alfabeto binario $\Sigma = \{0, 1\}$, si è potuto sfruttare il fatto che le run si alternino 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 costituita da simboli $\sigma = 0$. 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. L'implementazione di questo concetto si ritrova nella funzione get_symbol, la quale è visualizzabile all'algoritmo 3.1 e richiede tempo costante.

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

```
1: function GET_SYMBOL(k, r)                ▷ k indice di colonna, r indice di run
2:     if start_k then
3:         if r mod 2 = 0 then return 0 else return 1
4:     else
5:         if r mod 2 = 0 then return 1 else return 0
```

Si memorizzano gli indici delle teste di run in un array p_k, di lunghezza pari al numero di run in colonna k, avendo che un indice $i \in \{0, M-1\}$ è una testa di run sse:

$$i = 0 \vee y_{i-1}^{k}[k] \neq y_{i}^{k}[k] \tag{3.2}$$

Il passaggio successivo è stato quello di capire se le informazioni memorizzate per il mapping siano tutte necessarie, ovvero se, data la colonna k nella matrice PBWT, siano necessari interamente $c[k]$, u_k e v_k. In merito al valore $c[k]$, si è deciso che si possa calcolarlo in fase di costruzione delle RLPBWT e memorizzarlo esattamente come per la PBWT, avendo in totale N valori per l'intera trasformata. In merito, invece, ai vettori u_k e v_k si è cercato un modo per ottenerne una rappresentazione che implichi memorizzare un solo valore per ogni run della colonna. In altri termini,

si è cercato di capire se sia possibile tenere in memoria r valori (con r numero di run in una colonna) che permettano di effettuare comunque il mapping, a partire da un indice arbitrario $i \in \{0, \ldots, M-1\}$. L'alternanza data dal caso binario ha permesso di trovare una semplice soluzione, avendo che i valori di u_k e v_k crescono in modo alternato. Infatti, a seconda del simbolo σ 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, a ogni passo, di un'unità. Facendo un esempio, se in una run di simboli $\sigma = 0$ si itera 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. *Data la seguente colonna:*

$$y^5 = 00101111000000000000$$

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, come sopra descritto.

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 $\sigma = 0$. Per $i = 0$ si ha $u_k[i] = v_k[i] = 0$.
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 $i \in \{0, \ldots, r-1\}$, 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 $i \in 0, \ldots, r-1$:

- se $i = 0$ si ha che $u_k[p[i]] = v_k[p[i]] = uv_k[0] = 0$
- se $i \mod 2 = 0$ 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 $i \mod 2 \neq 0$ 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, chiamata uvtrick, è eseguibile in tempo costante e lo pseudocodice relativo a quanto appena descritto è consultabile all'algoritmo 3.2.

Algoritmo 3.2 Algoritmo per la funzione 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 mod 2 = 0 then
5:         u ← uv_k[i − 1], v ← uv_k[i]
6:         if start_k then return (u, v) else return (v, u)
7:     else
8:         u ← uv_k[i], v ← uv_k[i − 1]
9:         if start_k then return (u, v) else return (v, u)
```

Da un punto di vista implementativo, gli array di interi p_k e uv_k vengono memorizzati in vettori di interi bit-compressed (intvector compressi), disponibili nella libreria SDSL. Dato un array v, si memorizzano le componenti di v in un vettore di interi con componenti memorizzate in b bit, avendo:

$$b = \lceil \log(\max_i v[i] - 1) \rceil + 1 \tag{3.3}$$

Quindi, per la componente `MAP-INT`, si hanno in memoria, per ogni colonna k:

- $start_k$, ovvero il booleano necessario per identificare il simbolo della prima run
- p_k, ovvero gli indici delle teste di run.
- uv_k, ovvero i valori compatti di u_k e v_k per le teste di run
- $c[k]$, ovvero il numero totale di simboli $\sigma = 0$ nella colonna k della matrice PBWT

Facendo una prima stima della memoria occupata da questa componente si ha che, avendo ρ numero medio di run per colonna e considerando che un elemento di un intvector compresso necessita $\frac{\lceil \log(M-1) \rceil + 1}{8}$ byte, essa è:

$$\approx N \left(\rho \frac{\lceil \log(M-1) \rceil + 1}{4} + 5 \right) \text{ byte} \tag{3.4}$$

DC Ricontrollare conto

DC Spiegare algoritmo?

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

```
1: function BUILD_MAP_INT(col, pref)                                  ▷ pref = a_k
2:     c ← 0, u ← 0, v ← 0, u' ← 0, v' ← 0, run ← 0
3:     start ← ⊤, beg_run ← ⊤, push_zero ← ⊥, push_one ← ⊥
4:     p ← [], uv ← []
5:     for every k ∈ [0, M) do
6:         if k = 0 ∧ col[pref[k]] = 1 then start ← ⊥
7:         if col[pref[k]] = 0 then c ← c + 1
8:     if start then push_one ← ⊤ else push_zero ← ⊤
9:     for every k ∈ [0, M) do
10:        if beg_run then
11:            u ← u', v ← v'
12:            beg_run ← ⊥
13:        if col[pref[k]] = 1 then v' ← v' + 1 else u' ← u' + 1
14:        if k = 0 ∨ col[pref[k]] ≠ col[pref[k − 1]] then
15:            run ← k
16:        if k = M − 1 ∨ col[pref[k]] ≠ col[pref[k + 1]] then
17:            if push_one then
18:                push(p, run), push(uv, v)
19:                swap(push_one, push_zero)
20:            else
21:                push(p, run), push(uv, u)
22:                swap(push_one, push_zero)
23:            beg_run ← ⊤
24:    return (start, c, p, uv)
```

Esempio 22. *Data la seguente colonna:*

$$y^5 = 00101111000000000000$$

per la componente `MAP-INT` *della colonna 5, si hanno in memoria:*

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

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

$$c[5] = 15, \quad start_5 = \top$$

La costruzione della componente `MAP-INT` per una certa colonna, analizzabile nell'algoritmo 3.3, ha costo temporale $\mathcal{O}(M)$, avendo che la costruzione avviene scorrendo la colonna k, permutata dal prefix array a_k.
Per l'implementazione dei vari algoritmi si ha necessità di usare anche indici con valori $i \in \{0, M-1\}$ e non solo $i \in \{0, r-1\}$. Una delle operazioni fondamentali è quindi quella, dato un indice $i \in \{0, \ldots, M-1\}$, di computare a quale run esso appartenga, in una certa colonna k. Tale operazione può essere svolta usando una semplice variante della ricerca binaria che, anziché ritornare l'indice di un elemento nell'array, restituisce l'ultimo indice iniziale del sottointevallo usato dalla ricerca binaria, calcolato prima dell'interruzione dell'esecuzione dell'algoritmo. Pur essendo una funzione rank, si è deciso di chiamare tale funzione index_to_run, che, come visualizzabile all'algoritmo 3.4, ha complessità in tempo:

$$\mathcal{O}(\log r) \tag{3.5}$$

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 ≥ p_k[|p_k| − 1] then return |p_k| − 1
3:     b ← 0, e ← |p_k|
4:     run ← (e−b)/2
5:     while run ≠ e ∧ p_k[run] ≠ i do
6:         if i < p_k[run] then
7:             e ← run
8:         else
9:             if run + 1 = e ∨ p_k[run + 1] > i then
10:                break
11:            b ← run + 1
12:        run ← b + (e−b)/2
13:    return run
```

Ipotizzando di avere un indice $i \in \{0, \ldots, M-1\}$, 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. Ipotizzando di essere in una run di simboli σ con testa di run all'indice p, calcolati $u_k[p]$ e $v_k[p]$ da $uv_k[p]$, si hanno:

$$\begin{cases} v_k[i] = v_k[p] \\ u_k[i] = u_k[p] + (i-p) \end{cases} \text{, sse } y_p^k[k] = 0 \qquad \begin{cases} u_k[i] = u_k[p] \\ v_k[i] = v_k[p] + (i-p) \end{cases} \text{, sse } y_p^k[k] = 1 \tag{3.6}$$

Tenendo conto dell'offset off, qualora si abbia un simbolo σ uguale a quello della run in analisi, è possibile riadattare l'algoritmo per il mapping visto per la PBWT di Durbin. 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, usando la componente `MAP-INT`.

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

```
1:  function W(k, i, σ, o)        ▷ k indice di colonna, i indice di riga, σ simbolo
2:      run ← index_to_run(k, i)
3:      if σ = 0 ∧ get_symbol(start_k, run) = 1 then
4:          off ← 0
5:      else if σ = 1 ∧ get_symbol(start_k, run) = 0 then
6:          off ← 0
7:      else
8:          off ← i − p_k[run]
9:      (u, v) ← uvtrick(k, i)
10:     if p_k[i] + off = M then
11:         if get_symbol(start_k, i) = 0 then v ← v − 1 else u ← u − 1
12:     if σ = 0 then return u + off else return c[k] + v + off
```

Mapping con bitvector

La seconda variante della componente di mapping, al posto degli intvector compressi, utilizza i bitvector sparsi, da cui la nomenclatura `MAP-BV`.
L'idea è quella di sostituire, data una colonna k, quanto necessario a rappresentare le run e quanto necessario a permettere il mapping (ovvero i vettori p_k e uv_k della `MAP-INT`), tramite bitvector sparsi.
In primis, per poter localizzare le run nella k-esima colonna, si è scelto di usare un bitvector sparso, denominato h_k, tale che $|h_k| = M$. Formalmente, si ha che:

$$h_k[i] = \begin{cases} 1 & \text{se } y_i^k[k] \neq y_{i+1}^k[k] \vee i = M-1 \\ 0 & \text{altrimenti} \end{cases} \text{, } \forall i \in \{0, \ldots, M-1\} \tag{3.7}$$

Quindi, si ha che si ha $h_k[j] = 1$ sse j è l'indice di fine di una run.
In questa struttura dati succinta ci si aspettano poche run all'interno di una colonna della matrice PBWT, per quanto già discusso nella sezione 2.6. Avendo poche run, ci si aspetta di avere anche pochi simboli $\sigma = 1$ all'interno di h_k, ottimizzando al meglio l'uso dei bitvector sparsi. Si ricorda che, secondo quanto riportato per la libreria *SDSL* [13], tale variante richiede in memoria, indicando con r il numero di run:

$$\approx r \left(2 + \log \frac{M}{r}\right) \text{ bit} \tag{3.8}$$

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 ad accedere ai valori di p_k.
Per la rappresentazione dei vettori u_k e v_k si è deciso di optare per due bitvector sparsi, a differenza della rappresentazione unica vista con la `MAP-INT`. In particolare, per il vettore u_k, tale che $|u_k| = c[k]$, si ha che, $\forall i \in \{0, \ldots, |u_k| - 1\}$:

$$u_k[i] = \begin{cases} 1 & \text{se } i \text{ è il numero di simboli che contiene la } \mathsf{rank}_{u_k}(i)\text{-esima run di 0} \\ 0 & \text{altrimenti} \end{cases} \tag{3.9}$$

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

$$v_k[i] = \begin{cases} 1 & \text{se } i \text{ è il numero di simboli che contiene la } \mathsf{rank}_{v_k}(i)\text{-esima run di 1} \\ 0 & \text{altrimenti} \end{cases} \tag{3.10}$$

Si noti che:

$$\mathsf{rank}_{h_k}(|h_k| - 1) + 1 = (\mathsf{rank}_{u_k}(|u_k| - 1) + 1) + (\mathsf{rank}_{v_k}(|v_k| - 1) + 1) \tag{3.11}$$

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

Esempio 23. *Data la seguente colonna:*

$$y^5 = 00101111000000000000$$

si ha che:

$$h_5 = 01110001000000000001$$

avendo appunto un numero di run pari a:

$$\mathsf{rank}_{h_5}(|h_5| - 1) + 1 = 4 + 1 = 5$$

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

$$u_5 = 011000000000001$$

avendo:

- *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 che:

$$v_5 = 10001$$

avendo:

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

Si conferma, inoltre, quanto indicato nell'equazione 3.11:

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

Lo pseudocodice relativo alla costruzione della componente `MAP-BV` per la colonna k-esima è disponile 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 della costruzione 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:

$$\mathcal{O}(M) \tag{3.12}$$

È necessario spiegare come, dato un indice di riga $i \in \{0, \ldots, M-1\}$ e una colonna k, computare $u'_k[i]$ e $v'_k[i]$ (corrispondenti ai valori $u_k[i]$ e $v_k[i]$ della PBWT) a partire dagli attuali $u_k[i]$ e $v_k[i]$ (elementi dei due bitvector sparsi della `MAP-BV`). A differenza della componente `MAP-INT`, risulta difficile formulare un'unica formula per il calcolo di tali valori. Si è quindi deciso di offrire una spiegazione operazionale. Se $i = 0$, si ha che $u'_k[0] = v'_k[0] = 0$, in caso contrario, bisogna calcolare la run in cui si trova l'indice i. Questo si ottiene direttamente sfruttando h_k:

$$run = \mathsf{rank}_{h_k}(i) \tag{3.13}$$

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

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

```
1: function BUILD_MAP_BV(col, pref)                                  ▷ pref = a_k
2:     c ← 0, u ← 0, v ← 0, u' ← 0, v' ← 0, curr_lcs ← 0
3:     start ← ⊤, beg_run ← ⊤, push_zero ← ⊥, push_one ← ⊥
4:     for every k ∈ [0, M) do
5:         if k = 0 ∧ col[pref[k]] = 1 then start ← ⊥
6:         if col[pref[k]] = 0 then c ← c + 1
7:     runs ← [0..0]                  ▷ bitvector sparso per le run, di lunghezza M + 1
8:     zeros ← [0..0]                       ▷ bitvector sparso per u_k, di lunghezza c[k]
9:     ones ← [0..0]                      ▷ bitvector sparso per v_k, di lunghezza M − c
10:    if start then push_one ← ⊤ else push_zero ← ⊤
11:    for every k ∈ [0, M) do
12:        if beg_run then
13:            u ← u', v ← v', beg_run ← ⊥
14:        if col[pref[k]] = 1 then v' ← v' + 1 else e u' ← u' + 1
15:        if k = M − 1 ∨ col[pref[k]] ≠ col[pref[k + 1]] then
16:            runs[k] ← 1
17:            if push_one then
18:                if v ≠ 0 then ones[k − 1] = 1
19:                swap(push_zero, push_one)
20:            else
21:                if u ≠ 0 then zeros[k − 1] = 1
22:                swap(push_zero, push_one)
23:            beg_run ← ⊤
24:    if |zeros| ≠ 0 then zeros[|zeros| − 1] ← 1
25:    if |ones| ≠ 0 then ones[|ones| − 1] ← 1
26:    costruzione delle strutture per rank/select dei tre bitvector
27:    return (start, c, runs, zeros, ones)
```

1. si ha $run = 0$ e una run di simboli $\sigma = b$, con $b \in \{0, 1\}$. In tal caso, semplicemente:
$$(u, v) = \begin{cases} (i, 0) & \text{se } b = 0 \\ (0, i) & \text{altrimenti} \end{cases} \tag{3.14}$$

2. si ha $run = 1$ e una run di simboli $\sigma = b$, con $b \in \{0, 1\}$. In tal caso, bisogna individuare l'indice di inizio della seconda run, sfruttando h_k:
$$beg = \mathsf{select}_{h_k}(1) + 1 \tag{3.15}$$
A questo punto si ha il numero di simboli della prima run, e, calcolando la distanza tra l'indice di riga e quello di inizio della prima run, si ottiene che:
$$(u, v) = \begin{cases} (beg, i - beg) & \text{se } b = 0 \\ (i - beg, beg) & \text{altrimenti} \end{cases} \tag{3.16}$$

3. si ha $run = j$, con $j \in \{2, r-1\}$. Anche in questo caso si procede calcolando l'indice di inizio della run:
$$beg = \mathsf{select}_{h_k}(run) + 1 \tag{3.17}$$
e l'offset rispetto all'indice i dato da:
$$offset = i - beg \tag{3.18}$$
Si hanno due casi:

   (a) si è in una run di indice pari e si sfruttano poi u_k e v_k per sapere l'indice della precedente run con simboli $\sigma = 0$:
   $$pre_u = \mathsf{select}_{u_k}\left(\left\lfloor \frac{run}{2} \right\rfloor\right) + 1 \tag{3.19}$$
   Analogamente si calcola l'indice della run precedente con simboli simboli $\sigma = 1$:
   $$pre_v = \mathsf{select}_{v_k}\left(\left\lfloor \frac{run}{2} \right\rfloor\right) + 1 \tag{3.20}$$
   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 costante mentre l'altro valore deve essere calcolato considerando l'offset:

$$(u, v) = \begin{cases} (pre_u + offset, pre_v) & \text{se } b = 0 \\ (pre_u, pre_v + offset) & \text{altrimenti} \end{cases} \tag{3.21}$$

(b) ci si trova in una run di indice dispari, quindi non si ha lo stesso numero di run che precedono i due simboli e bisogna calcolare quante siano. Se la prima run è di zeri si ha che:

$$run_u = \mathsf{select}_{u_k}\left(\left\lfloor \frac{run}{2} \right\rfloor\right) + 1 \tag{3.22}$$

$$run_v = \mathsf{select}_{v_k}\left(\left\lfloor \frac{run}{2} \right\rfloor\right) \tag{3.23}$$

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, potendo procedere come nel caso precedente:

$$pre_u = \mathsf{select}_{u_k}(run_u) + 1 \tag{3.24}$$

$$pre_v = \mathsf{select}_{v_k}(run_v) + 1 \tag{3.25}$$

Infine, si calcola:

$$(u, v) = \begin{cases} (pre_u, pre_v + offset) & \text{se } b = 0 \\ (pre_u + offset, pre_v) & \text{altrimenti} \end{cases} \tag{3.26}$$

Esempio 24. *Si riprendono i dati e i risultati ottenuti all'esempio 23 e si vogliono calcolare $u[i]$ e $v[i]$ per $i = 6$.*
Si hanno:

$$run = \mathsf{rank}_{h_5}(6) = 3$$

$$beg = \mathsf{select}_{h_5}(3) + 1 = 3 + 1 = 4$$

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

Quindi ci si trova nel terzo caso e in una run di indice dispari. Si calcolano:

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

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

Tali valori non vanno invertiti avendo $start_5 = \top$.
Si calcolano:

$$pre_u = \mathsf{select}_{u_5}(2) + 1 = 2 + 1 = 3$$

$$pre_v = \mathsf{select}_{v_5}(0) + 1 = 0 + 1 = 1$$

avendo, in totale, tre simboli $\sigma = 0$ *e un simbolo* $\sigma = 1$ *prima dell'indice 6.*
Avendo $start_5 = \top$*, si ha:*

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

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, 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:

$$\mathcal{O}\left(\log \frac{M}{r}\right) \tag{3.27}$$

Non dovendo considerare esplicitamente l'offset, come nel caso della `MAP-INT`, il mapping dalla colonna k alla colonna $k+1$ viene fatto come nel caso della PBWT (visualizzabile all'algoritmo 3.8), che presenta quindi la medesima complessità del calcolo di $u[i]$ e $v[i]$, ovvero quello visto all'equazione 3.27. Facendo una prima stima della memoria occupata da questa componente, dato ρ numero medio di run per colonna, essa è:

$$\approx N \left(\frac{\rho \left(2 + \log \frac{M}{\rho}\right)}{4} + 29 \right) \text{ byte} \tag{3.28}$$

È possibile stimare un confronto tra la memoria richiesta da un bitvector sparso e da un intvector compresso.

DC Ricontrollare conto

Si noti che è molto complesso risolvere analiticamente la seguente equazione, relativa a un solo bitvector sparso, comprendente le strutture per le funzioni rank e select, e a un solo intvector compresso:

$$\rho \left(2 + \log \frac{M}{\rho}\right) + 128 = \rho \lceil \log(M - 1) \rceil + 1 \tag{3.29}$$

Bisogna considerare come difficoltà aggiuntiva il fatto che M, $\rho \in \mathbb{N}$ e che entrambi i lati dell'equazione hanno valori nell'insieme dei naturali.

DC Ricontrollare conto

Solo valutazioni sperimentali hanno mostrato come, con valori di M e ρ relativi ai pannelli di aplotipi studiati, l'uso degli intvector compressi sia più vantaggioso in termini di memoria richiesta, nei casi testati. Una stima derivata da tali conti

Algoritmo 3.7 Algoritmo per la funzione uvtrick con `MAP-BV`.

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

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

1: **function** W($k,\ i,\ \sigma$) ▷ k indice di colonna, i indice di riga, σ simbolo
2: $c \leftarrow c[k]$
3: $(u, v) \leftarrow$ uvtrick($k,\ i$)
4: **if** $\sigma = 0$ **then return** u **else return** $c + v$

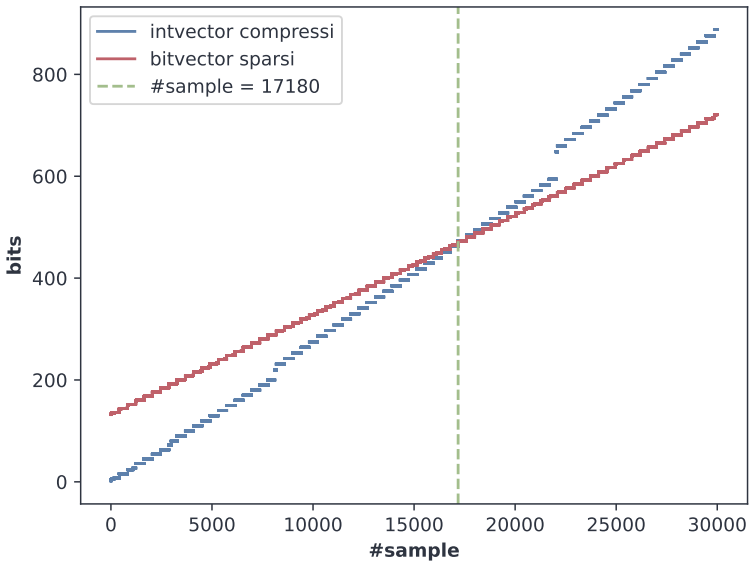


Figura 3.1: Confronto tra la stima di memoria in bit necessaria a un bitvector sparso e a un intvector compresso, al variare dell'altezza del pannello. Con la linea verde si segnala il numero di sample dopo il quale si ha un vantaggio in memoria nell'usare un bitvector sparso.

è visibile in figura 3.1, da cui si evince che, qualora la media del numero di run resti proporzionale a quanto visto con 4.908 sample (ovvero una media di 12 run come verrà analizzato nel Capitolo 4), l'uso di un bitvector sparso diventerebbe favorevole solo con pannelli con più di circa 17.180 sample/aplotipi/righe.

3.3.2 Componente per le threshold

Come discusso con MONI per la RLBWT, l'uso delle threshold è uno dei due modi per computare i campi len dell'array delle 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 l'indice del primo minimo valore dell'*RLCP *(o l'indice del primo massimo valore del divergence array), compreso negli indici della run, considerando anche l'eventuale* $\mathsf{RLCP}_k[i'+1]$*, qualora $i' \neq M-1$. Si noti che quest'ultimo valore, se esistente, deve essere considerato in quanto calcolato prendendo in considerazione $y^k_{i'}$ (valore appartenente alla run in analisi) e $y^k_{i'+1}$.*

Da un punto di vista implementativo, come anticipato, si hanno due soluzioni, una basata su intvector compressi e una basata su bitvector sparsi. In entrambi

i casi il calcolo si può effettuare in parallelo a quello delle componenti `MAP-INT` e `MAP-BV`.

Threshold con intvector compressi

Con questa componente, la memorizzazione delle threshold avviene in modo molto semplice, usando un vettore di interi bit-compressed. Data una colonna k della matrice PBWT, 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 accedere ai valori di l_k, tale operazione ha complessità in tempo proporzionale a:

$$\mathcal{O}(M) \tag{3.30}$$

Si noti che, qualora il minimo valore dell' RLCP si trovi nella testa della run successiva, si memorizza, come threshold, l'indice della testa della run successiva.

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

```
function BUILD_THR_INT(col, pref, div)          ▷ pref = a_k, div = l_k
    curr_lcs ← 0, tmp_thr ← 0
    t ← []
    for every k ∈ [0, M) do
        if k = 0 ∨ col[pref[k]] ≠ col[pref[k − 1]] then
            curr_lcs ← div[k], tmp_thr ← k
        if div[k] < curr_lcs then
            curr_lcs ← div[k], tmp_thr ← k
        if k = M − 1 ∨ col[pref[k]] ≠ col[pref[k + 1]] then
            if k ≠ M − 1 ∧ div[k + 1] < div[tmp_thr] then
                push(t, k + 1)
            else
                push(t, tmp_thr)
    return t
```

Threshold con bitvector

Con questa componente, le posizioni delle threshold vengono memorizzate, per ogni colonna k, tramite un bitvector sparso, denotato t_k, avendo che $t_k[i] = 1$ sse i è l'indice di una threshold. Qualora il minimo RLCP si trovi nell'indice della

testa della run successiva, la posizione della threshold verrà memorizzata all'indice della coda della run corrente. Purtroppo questa è una situazione di ambiguità. Infatti, come si vedrà poi durante lo studio dell'algoritmo di calcolo delle matching statistics tramite threshold, il fatto che una threshold, posta a fine run corrisponda effettivamente a un minimo RLCP in quella posizione o in quella successiva, comporta differenze dal punto di vista del calcolo dell'array MS. 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. Avere due threshold sovrapposte impedirebbe di capire a quale run appartiene una certa threshold, tramite la funzione rank sul corrispondente bitvector sparso.
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 accedere ai valori di l_k, tale operazione ha complessità in tempo proporzionale a:

$$\mathcal{O}(M) \tag{3.31}$$

In merito al confronto tra `THR-INT` e `THR-BV`, in termini di uso di memoria, valgono le stesse considerazioni fatte per la componente per il mapping.

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

function BUILD_THR_BV($col,\ pref,\ div$) ▷ $pref = a_k,\ div = l_k$
  $curr_{lcs} \leftarrow 0,\ tmp_{thr} \leftarrow 0$
  $t \leftarrow [0..0]$ ▷ 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**
      $curr_{lcs} \leftarrow div[k],\ tmp_{thr} \leftarrow k$
    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**
        $t[k] \leftarrow 1$
      else
        $t[tmp_{thr}] \leftarrow 1$
  costruzione delle strutture rank/select *per* t
  return t

3.3.3 Componente per i prefix array sample

Come introdotto parlando delle matching statistics, qualora si abbia un cambio di riga da memorizzare, si seleziona sempre quella relativa alla coda della run prece-

dente 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. Anche in questo caso si sono scelti gli intvector compressi. Tali valori sono 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 tale componente per la colonna k-esima. L'algoritmo, dovendo iterare l'intera colonna della *matrice PBWT* ha costo, in tempo:

$$\mathcal{O}(M) \tag{3.32}$$

La costruzione può essere fatta in contemporanea a quelle delle componenti già descritte, ovvero: `MAP-INT`/`MAP-BV` e `THR-INT`/`THR-BV`.
In termini di memoria necessaria per questa componente, si hanno le medesime considerazioni fatte nel caso della componente `MAP-INT`, per l'uso degli intvector compressi.

Algoritmo 3.11 Algoritmo per la costruzione della componente `PERM` per la colonna k.

```
1: function BUILD_PERM(col, pref)                          ▷ pref = a_k
2:     tmp_beg ← 0, beg_run ← ⊤
3:     samples_beg ← []          ▷ vettore per i prefix array sample ad inizio di ogni run
4:     samples_end ← []            ▷ vettore per i prefix array sample a fine di ogni run
5:     for every k ∈ [0, height) do
6:         if beg_run then
7:             tmp_beg ← pref[k]
8:             beg_run ← ⊥
9:         if k = height − 1 ∨ col[pref[k]] ≠ col[pref[k + 1]] then
10:            push(samples_beg, tmp_beg)
11:            push(samples_end, pref[k])
12:            beg_run ← ⊤
13:    return (samples_beg, samples_end)
```

3.3.4 Componenti per il random access e le LCE query

Si descrivono ora le componenti atte a garantire il random access al testo e, nel caso dell'uso di un 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 a un qualsiasi elemento del pannello mentre, nel caso di `RA-SLP`, si ha che

l'accesso a ogni elemento è in tempo:

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

La seconda differenza 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 dovuti al vettore che memorizza i bitvector. Parlando invece di `RA-SLP`, non si può avere una stima a priori dello spazio necessario ma, come si vedrà nel Capitolo 4, i risultati quantitativi daranno prova della capacità di compressione di tale grammatica.
Parlando della componente `RA-SLP` e della componente `LCE`, bisogna descrivere la metodologia 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 singola sequenza lineare. Inoltre, per permettere la costruzione efficiente della PBWT, e conseguentemente della RLPBWT, il pannello in input risulta essere trasposto, avendo che ogni record consecutivo nel file contiene una colonna del pannello e non una riga dello stesso. Bisogna quindi trasporre tale pannello in input. Si anticipa che sull' SLP si avrà necessità di effettuare LCE query che devono essere fatte tra due righe da destra a sinistra (a differenza di quanto visto nel caso standard dove si confrontavano, da sinistra a destra, prefissi comuni a partire da due posizioni del testo). Per rendere possibile questa operazione, il pannello deve essere sia salvato come un'unica stringa, in modo da ottenerne l' SLP, che essere letto 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. *Dato il seguente pannello trasposto nel file in input:*

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

*si ha che le righe sono i siti e le colonne i sample. Per ottenere l'*SLP *bisogna 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 vogliono effettuare le LCE *query. Ad esempio, prendendo la seconda e la terza riga, facendo partire il confronto*

dall'ultima colonna, si ha una LCE *query lunga 3, terminante nella prima colonna esclusa:*

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

Si procede quindi salvando la sequenza lineare relativa al pannelli come descritto sopra. Si ottiene, con colorati gli stessi risultati della LCE *query fatta sopra:*

0010 0110 0111 1100 0110

In merito al random access, considerando tale memorizzazione monodimensionale e invertita, si ha, per accedere alla colonna k della riga i-esima del pannello (con N colonne):

$$x_i[k] = \mathsf{SLP}[N(i+1) - k - 1] \tag{3.34}$$

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

$$\mathcal{O}(\log(NM)) \tag{3.35}$$

3.3.5 Componente per la struttura phi

L'ottenimento dell'array delle matching statistics permette di sapere solo l'indice di una della righe del pannello per le quali si ha uno SMEM con l'aplotipo query. Analogamente a quanto discusso in PHONI [8], anche per la RLPBWT si è pensato a due funzioni, φ e φ^{-1}, per il riconoscimento di tutte le righe del pannello per cui si ha il medesimo SMEM. La componente che permette il calcolo di tali funzioni è la componente denotata `PHI`.
Nell'ordinamento alla colonna k-esima, dato da a_k, tutte le righe, per le quali si ha un certo SMEM, sono poste consecutivamente, a causa dell'ordinamento lessicografico inverso.

Definizione 29. *Dati un pannello X, di dimensioni $M \times N$, e una colonna k, avendo prefix array a_k e permutazione inversa del prefix array α_k, si definiscono formalmente:*

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

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

In altri termini, si ha che:

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

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

Quindi, dato un elemento di a_k, le due funzioni restituiscono il valore antecedente e il valore successivo a esso, se esistenti nel prefix array. In caso di inesistenza di tali valori, rispettivamente a inizio e fine del prefix array, si restituisce null*.*

Esempio 26. *Si ipotizza 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 $p = 3$, ottenendo:

$$\varphi_6(3) = a_6[\alpha_6[3] - 1] = a_6[14 - 1] = a_6[13] = 2$$

$$\varphi_6^{-1}(3) = a_6[\alpha_6[3] + 1] = a_6[14 + 1] = a_6[15] = 17$$

Avendo $\mathsf{MS}[i].\mathsf{row} = p$ e $\mathsf{MS}[i].\mathsf{len} = l$, è sufficiente iterare, in entrambe le direzioni, le righe a partire da p in a_i, che denotiamo con l'indice q, fino a che si ha $\mathsf{LCE}_k(x_p, x_q) \geq l$. Tutte le righe x_q che soddisfano tale condizione presentano uno SMEM di lunghezza l con l'aplotipo query. L'algoritmo 3.12 rappresenta quanto appena descritto, avendo che la funzione lce_bounded limita il calcolo della LCE alla lunghezza desiderata l, escludendo computazioni oltre tale lunghezza. Tale funzione è riproducile, per mezzo di iterazioni, anche sulla componente `RA-BV`. La complessità temporale di questo algoritmo varia a seconda della componente per il random access (e della conseguente presenza della componente `LCE`). Inoltre, è difficile poter dare una stima asintotica in quanto varia sul numero di righe ν che presentano un certo SMEM. Quindi, si ha, con la componente `RA-BV`, un tempo proporzionale a:

$$\mathcal{O}(\nu N) \tag{3.36}$$

Mentre con l'uso della componente `LCE`, avendo il pannello in memoria sotto forma di SLP, si ha complessità in tempo:

$$\mathcal{O}(\nu \log(NM)) \tag{3.37}$$

Entrambe le stime assumono, solo per il momento, che sia possibile calcolare il valore delle funzioni φ e φ^{-1} in tempo $\mathcal{O}(1)$, avendo in memoria l'insieme dei prefix array (e l'insieme delle permutazioni inverse dei prefix array) con random access in tempo costante.
Con la RLPBWT, si hanno in memoria solo i prefix array sample e nessuna informazione in merito alla permutazione inversa del prefix array. Non si ha in memoria nemmeno l'RLCP, che, in via teorica, come per la BWT, potrebbe rendere più efficiente il computo delle funzioni φ e φ^{-1}. Quindi, si è pensato a una struttura dati, basata su bitvector sparsi e intvector compressi, che permetta il calcolo delle due funzioni senza mantenere informazioni complete in memoria.

Algoritmo 3.12 Algoritmo per il calcolo di ogni SMEM in colonna k tramite la componente `PHI`.

```
1: function EXTEND_MATCHES(k, row, len)
2:     haplos ← []
3:     check_down ← ⊤, check_up ← ⊤
4:     while check_down do
5:         down_row ← φ^{-1}(row, k)
6:         if lce_bounded(k, row, down_row, len) then
7:             push(haplos, down_row)
8:             row ← down_row
9:         else
10:            check_down ← ⊥
11:    while up_down do
12:        up_row ← φ(row, k)
13:        if lce_bounded(k, row, up_row, len) then
14:            push(haplos, up_row)
15:            row ← up_row
16:        else
17:            check_up ← ⊥
18:    return haplos
```

Costruzione della struttura di supporto

L'idea per la costruzione della struttura a supporto delle funzioni φ e φ^{-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 corrisponderanno a due simboli diversi, consecutivi nella matrice PBWT. 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 e valore di prefix array arbitrari, permetta di computare φ e φ^{-1}.
Tale struttura dati è composta da:

- un vettore di bitvector sparsi per φ, che denotiamo con Φ, 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 Φ ha dimensione $M \times N$
- un vettore di bitvector sparsi per φ^{-1}, che denotiamo con Φ^{-1}, tale che $\Phi[i][j] = 1$ sse la riga i indicizza una coda di run alla colonna j, nella matrice PBWT. Si ha quindi che Φ^{-1} ha dimensione $M \times N$
- un vettore di intvector compressi, denotato Φ_{supp}, a supporto del vettore Φ, che memorizza, per ogni simbolo $\sigma = 1$ 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 intvector compressi, denotato Φ^{-1}_{supp}, a supporto del vettore Φ^{-1}, che memorizza, per ogni simbolo $\sigma = 1$ 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 che la lunghezza della riga i-esima di Φ_{supp} (di M righe totali) è uguale al numero di simboli $\sigma = 1$ presenti nella riga i-esima di Φ. Analogamente si ha per Φ^{-1}_{supp}. Queste osservazioni si ripercuotono sul costo in memoria della componente `PHI`, avendo che può essere stimata con quelli dei bitvector sparsi e degli intvector compressi, come fatto per le precedenti componenti. Si noti che in questo caso i bitvector sparsi sono ulteriormente ottimizzati, avendo un rapporto davvero basso di simboli $\sigma = 1$ sul totale di simboli N (non M come negli altri casi, segnalando che, in pannelli reali, $N >> M$).
Bisogna anche sfruttare a_{N-1} per poter identificare quelle coppie di valori consecutivi non presenti nei vari prefix array sample, in modo che sia possibile effettuare le query per qualsiasi valore di prefix array in input.
L'algoritmo 3.13 riporta la costruzione della struttura, iterando prima i vari prefix array sample e completando i risultati con a_{N-1}. Tale algoritmo ha complessità in tempo, nel caso peggiore, pari a:

$$\mathcal{O}(NM) \tag{3.38}$$

Tale caso peggiore si ha qualora ogni colonna della matrice PBWT abbia un numero di run pari all'altezza stessa della colonna (un caso irrealistico). Indicando con ρ il numero medio di run per colonna, si ha che la complessità nel caso medio è:

$$\Theta(N\rho) \tag{3.39}$$

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

function BUILD_PHI($cols, panel, prefix$) ▷ $prefix = a_{N-1}$
  $\Phi \leftarrow [[0..0]..[0..0]]$, $\Phi^{-1} \leftarrow [[0..0]..[0..0]]$ ▷ vettori di bitvector sparsi per φ e φ^{-1}
  $\Phi_{supp} = []$, $\Phi^{-1}_{supp} = []$ ▷ vettori di intvector compressi di supporto per φ e φ^{-1}
  for *every* $k \in [0, |cols|)$ **do** ▷ costruzione da prefix array sample
    for *every* $i \in [0, |samples_{beg}|)$ **do**
      $\Phi[sample^k_{beg}[i]][k] \leftarrow 1$
      if $i = 0$ **then**
        $push(\Phi_{supp}[sample^k_{beg}[i]], panel_{height})$
      else
        $push(\Phi_{supp}[sample^k_{beg}[i]], sample^k_{end}[i-1])$
      $\Phi^{-1}[sample^k_{end}[i]][k] \leftarrow 1$
      if $i = |sample^k_{beg}| - 1$ **then**
        $push(\Phi^{-1}_{supp}[sample^k_{end}[i]], panel_{height})$
      else
        $push(\Phi^{-1}_{supp}[sample^k_{end}[i]], sample^k_{beg}[i+1])$
  for *every* $k \in [0, |prefix|)$ **do** ▷ costruzione da ultimo prefix array
    if $\Phi[k][|\Phi[k]| - 1] = 0$ **then**
      $\Phi[k][|\Phi[k]| - 1] \leftarrow 1$
      if $k = 0$ **then**
        $push(\Phi_{supp}[prefix[k]], panel_{height})$
      else
        $push(\Phi_{supp}[prefix[k]], prefix^k[i-1])$
    if $\Phi^{-1}[k][|\Phi[k]| - 1] = 0$ **then**
      $\Phi^{-1}[k][|\Phi[k]| - 1] \leftarrow 1$
      if $k = |prefix| - 1$ **then**
        $push(\Phi^{-1}_{supp}[prefix[k]], panel_{height})$
      else
        $push(\Phi^{-1}_{supp}[prefix[k]], prefix^k[i+1])$
  costruzione della struttura `rank` *per ogni bitvector sparso* Φ e Φ^{-1}

Dal punto di vista delle query, data una colonna k e un valore di prefix array p, per la funzione φ si effettua $\mathsf{rank}^{\varphi}(k)$ sulla riga p di Φ, avendo che:

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

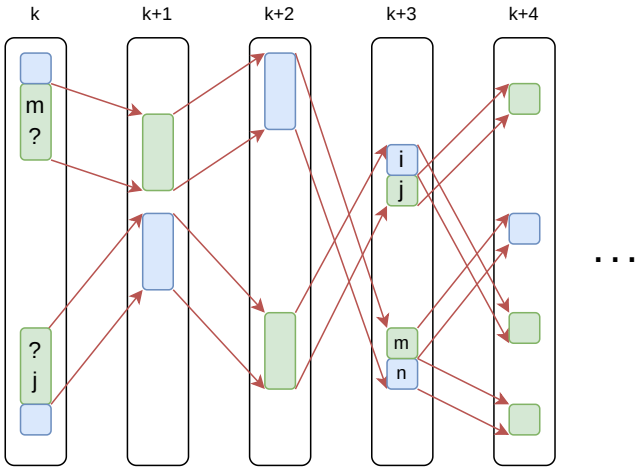
Analogamente, per la funzione φ^{-1} si effettua la $rank^{\varphi^{-1}}(k)$ sulla riga p di Φ^{-1}, avendo che:

$$\varphi^{-1}_k(p) = \begin{cases} \mathsf{null} & \text{se } \Phi^{-1\,p}_{supp}[\mathsf{rank}^{\varphi^{-1}}_{p}(k)] = M \\ \Phi^{-1\,p}_{supp}[\mathsf{rank}^{\varphi^{-1}}_{p}(k)] & \text{altrimenti} \end{cases}$$

In termini di complessità, si ha che tale calcolo è limitato da quella della funzione rank, avendo che il numero m di simboli $\sigma = 1$ in ogni bitvector (lungo N) equivale al numero di volte in cui la corrispondente riga è testa/coda di una run:

$$\mathcal{O}\left(\log \frac{N}{m}\right) \tag{3.40}$$

Esempio 27. *Sia dato il seguente caso nella matrice* PBWT*:*



dove, a parità di colore, si ha lo stesso simbolo tra due indici consecutivi.
In colonna k, che assumiamo essere $k = 0$, si vogliono calcolare $\varphi_k(j)$ e $\varphi^{-1}_k(m)$.
Si nota che, per definizione della struttura dati, si ha (limitandoci alle colonne della figura):

$$\Phi_j = [0, 0, 0, 1, 0, \ldots]$$
$$\Phi^{-1}_m = [0, 0, 0, 1, 0, \ldots]$$

questo in quanto, in colonna $k+3$, si ha che j è il valore del prefix array di una testa di run mentre m di una coda di run. Nella stessa colonna si conoscono anche i, valore del prefix array della coda della run precedente a quella di j, e n, valore del prefix array della testa della run successiva quella di m. Si ottengono:

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

$$\Phi^{-1}_{supp} = [n, \ldots]$$

Per calcolare $\varphi_0(j)$ e $\varphi_0^{-1}(m)$, si ha:

$$\Phi^{j}_{supp}[\mathsf{rank}^{\varphi}_{j}(0)] = \Phi^{j}_{supp}[0] = i$$

$$\Phi^{-1\ m}_{supp}[\mathsf{rank}^{\varphi^{-1}}_{m}(0)] = \Phi^{-1\ m}_{supp}[0] = n$$

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

3.4 Strutture dati per la RLPBWT

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

- due strutture dati composte unicamente dalle componenti dedicate al mapping e dall'intero RLCP, dette:

  1 `MAP-INT + RLCP`

  2 `MAP-BV + RLCP`

  Queste soluzioni non permettono di sapere quali righe del pannello presentino uno SMEM, terminante in una certa colonna, ma solo quante esse siano. Non avendo traccia dei valori del prefix array, non si ha modo di usare la componente `PHI`. Il funzionamento dell'algoritmo è basato 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 [7] per la RLBWT, tramite threshold e random access al pannello. Quest'ultimo può essere memorizzato come vettore di bitvector o come SLP e l'algoritmo necessita sia della componente atta al mapping che di quella relativa ai prefix array sample. Infine, per estendere il riconoscimento a tutte le righe che presentano un medesimo SMEM fino a una certa colonna, è necessaria la struttura che permette di calcolare le funzioni φ e φ^{-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 PHONI [8] per la RLBWT, tramite l'uso delle LCE query. Queste ultime sostituiscono l'uso delle threshold e del random access al pannello. Tali strutture sono nominate:

  7 `MAP-INT + LCE + PERM + PHI`

  8 `MAP-BV + LCE + PERM + PHI`

Una visualizzazione grafica di quanto descritto è rappresentata alla figura 3.2.

3.4.1 Calcolo degli SMEM con RLCP

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

- `MAP-INT + RLCP`
- `MAP-BV + RLCP`

I due algoritmi riprendono quanto discusso per l'algoritmo 5 di Durbin, non sfruttano l'uso delle matching statistics e sono limitati dal non poter calcolare quali righe presentano uno SMEM, ma solo quante siano. Il secondo limite è dato dal fatto che necessitano di avere interamente in memoria l' RLCP array, una struttura non run-length encoded.
Il metodo procede con l'aggiornamento dei tre indici e_k, f_k e g_k, avendo che gli ultimi due possono assumere qualsiasi valore in $\{0, \ldots, M\}$, come con la PBWT. Inoltre, Durbin sfrutta il random access al pannello in input, avendolo in memoria insieme al prefix array e al divergence array, per aggiornare il valore di e_k. In entrambe le strutture dati run-length encoded non si ha in memoria né il prefix array né il pannello, ma solo la rappresentazione compatta della matrice PBWT. Quindi, si è dovuto pensare a un metodo che ricomponga una data riga x_j del pannello X a partire da un elemento, indicizzato alla colonna $k+1$ della matrice PBWT, con $a_{k+1}[i] = j$ e $0 \leq i < M$. Questo risultato si può ottenere muovendosi da destra a sinistra, seguendo in modo inverso la permutazione che produce il prefix array. In altri termini, tale metodo permette un mapping inverso, che segue una riga del

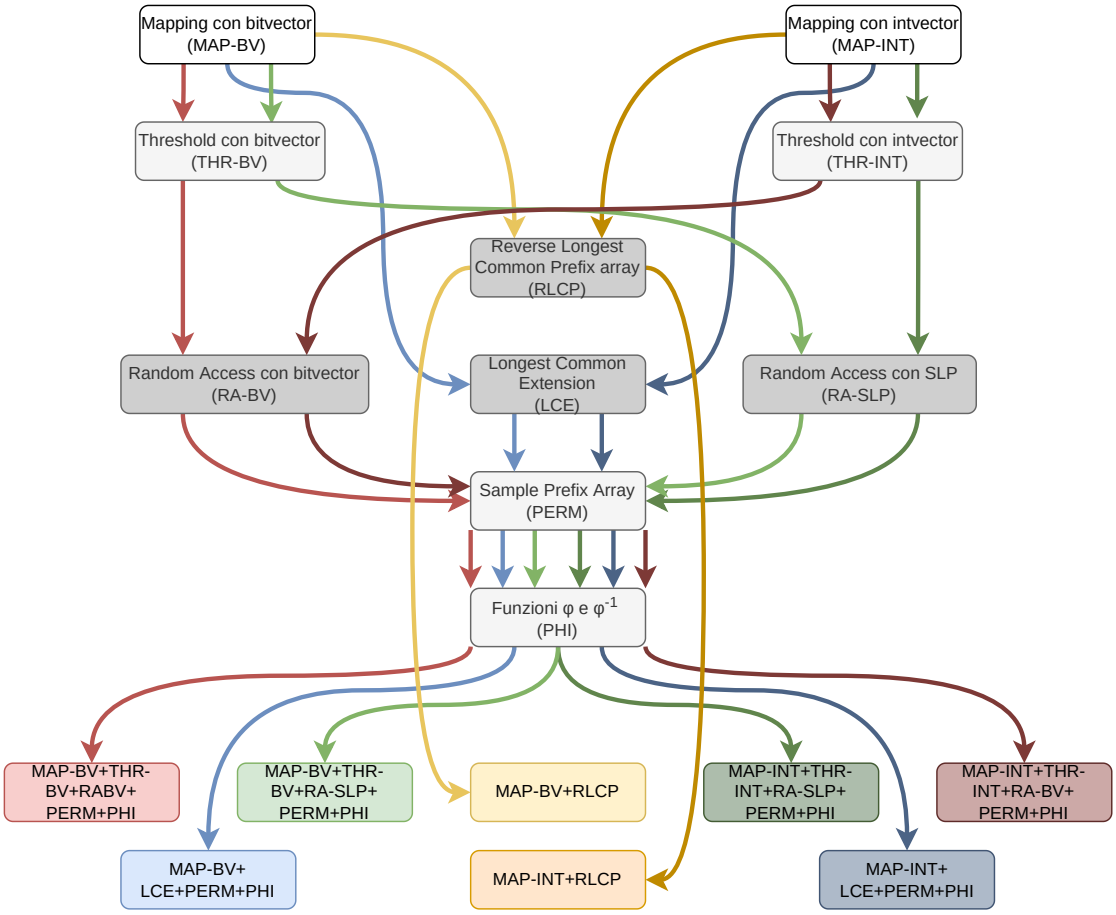


Figura 3.2: Schema grafico dell'ottenimento delle otto strutture dati a partire dalle varie componenti.

pannello originale nella matrice PBWT a ritroso.
Per ottenere l'indice alla colonna k-esima della matrice PBWT, 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, i è un indice di riga corrispondente a un simbolo $\sigma = 0$, ricordando come la costruzione della colonna $k+1$ nella matrice PBWT si abbia grazie a un ordinamento stabile. In caso contrario, i corrisponde a un simbolo $\sigma = 1$ in colonna k. Si sfruttano così 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 $k+1$, e poi seguendo, virtualmente, la riga x_j del pannello originale. Per quanto riguarda la struttura composta `MAP-INT + RLCP`, si ha lo pseudocodice per il mapping inverso consultabile all'algoritmo 3.14 mentre, per la struttura composta `MAP-BV + RLCP`, si ha l'algoritmo 3.15. Parlando in termini di complessità in tempo, usando la componente `MAP-INT`, si ha, con r numero di run alla colonna k, un caso peggiore proporzionale a:

$$\mathcal{O}(r) \tag{3.41}$$

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

$$\mathcal{O}\left(\log \frac{M}{r}\right) \tag{3.42}$$

Si ha un riadattamento dell'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 esplicitamente conto degli offset nel caso della `MAP-INT + RLCP`
- non si ha random access al pannello quindi bisogna procedere, ogni volta, con l'inverso del mapping e il calcolo del simbolo corrispondente alla run
- non si ha il prefix array in memoria, quindi non è possibile sapere quali siano le righe per cui si ha uno SMEM fino alla colonna k, ma è possibile conoscere solo quante siano: $g_k - f_k$

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 a quanto visto per quest'ultimo, 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 computare la nuova tripla e_k, f_k e g_k. Tale numero di step scala sulla cardinalità dei caratteri in overlap tra due SMEM consecutivi.

Algoritmo 3.14 Algoritmo per il mapping inverso con la `MAP-INT + RLCP`.

```
1:  function REVERSE_MAP(k, i)          ▷ k indice di colonna, i indice di riga
2:      if k = 0 then return 0                                    ▷ by design
3:      k ← k − 1
4:      c ← rlpbwt[k].c
5:      u ← 0, v ← 0, offset ← 0, run ← 0, found ← ⊥
6:      if i < c then
7:          u ← i
8:          prev_0 ← 0, next_0 ← 0
9:          for every j ∈ [0, |p_k|) do
10:             (prev_0, _) ← uvtrick(k, j)
11:             (next_0, _) ← uvtrick(k, j + 1)
12:             if prev_0 ≤ u < next_0 then
13:                 run ← j, found ← ⊤
14:                 break
15:         if ¬found then run ← |p_k| − 1
16:         (curr_u, _) ← uvtrick(k, run), offset ← u − curr_u
17:         return p_k[run] + offset
18:     else
19:         v ← i − c
20:         prev_1 ← 0, next_1 ← 0
21:         for every j ∈ [0, |p_k|) do
22:             (_, prev_1) ← uvtrick(k, j)
23:             (_, next_1) ← uvtrick(k, j + 1)
24:             if prev_1 ≤ v < next_1 then
25:                 run ← j, found ← ⊤
26:                 break
27:         if ¬found then run ← |p_k| − 1
28:         (curr_v, curr_u) ← uvtrick(k, run), offset ← v − curr_v
29:         return p_k[run] + offset
```

Algoritmo 3.15 Algoritmo per il mapping inverso con la `MAP-BV + RLCP`.

```
1:  function REVERSE_MAP(k, i)          ▷ k indice di colonna, i indice di riga
2:      if k = 0 then return 0                                          ▷ by design
3:      k ← k − 1
4:      c ← c[k]
5:      if i < c then
6:          if start_k then
7:              run ← rank_u^k(i) · 2
8:          else
9:              run ← rank_u^k(i) · 2 + 1
10:         i_run ← 0
11:         if run ≠ 0 then i_run ← select_h^k(run) + 1
12:         (prev_0, _) ← uvtrick(k, i_run)
13:         return i_run + (i − prev_0)
14:     else
15:         if start_k then
16:             run ← rank_v^k(i) · 2 + 1
17:         else
18:             run ← rank_v^k(i) · 2
19:         i_run ← 0
20:         if run ≠ 0 then i_run ← select_h^k(run) + 1
21:         (_, prev_1) ← uvtrick(k, i_run)
22:         return i_run + (i − (c + prev_1))
```

Fatta questa premessa, si può stimare che il calcolo dei c SMEM con la struttura `MAP-INT + RLCP` è proporzionale, con ρ numero medio di run per una colonna, a:

$$\mathcal{O}(N \log \rho + c) \tag{3.43}$$

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

$$\mathcal{O}\left(N \log \frac{M}{\rho} + c\right) \tag{3.44}$$

3.4.2 Calcolo degli SMEM con matching statistics

L'obiettivo principale di questa tesi è quello di applicare i metodi e gli algoritmi già studiati per la RLBWT alla RLPBWT.
Nelle sei strutture dati dedicate al calcolo degli SMEM tramite matching statistics, si riconoscono le due modalità già descritte con MONI [7] e PHONI [8]:

1. calcolare l'array MS, in due passaggi, sfruttando le threshold per computare i valori $\mathsf{MS}[i].\mathsf{row}$ e il random access al pannello per calcolare i valori $\mathsf{MS}[i].\mathsf{len}$
2. calcolare l'array MS, in un passaggio, sfruttando le LCE query sia per scegliere le righe da memorizzare in $\mathsf{MS}[i].\mathsf{row}$ che per calcolare, in contemporanea, i valori $\mathsf{MS}[i].\mathsf{len}$

Grazie a queste due strategie, si possono computare tutti gli SMEM senza tenere in memoria il divergence array.

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`

Algoritmo 3.16 Calcolo degli SMEM con aplotipo esterno per `MAP-INT/BV + RLCP`, 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: $f_{run} \leftarrow \mathsf{index\_to\_run}(f, k)$ **oppure** $f_{run} \leftarrow \mathsf{rank}_h^k(f)$
7: $g_{run} \leftarrow \mathsf{index\_to\_run}(g, k)$ **oppure** $f_{run} \leftarrow \mathsf{rank}_h^k(g)$
8: $f' \leftarrow \mathsf{w}(k,\ f,\ z[k]),\ g' \leftarrow \mathsf{w}(k,\ g,\ z[k]),\ nh \leftarrow g - f$
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: $f_{rev} \leftarrow \mathsf{reverse\_map}(k',\ f_{rev}),\ k' \leftarrow k' - 1$
21: $run \leftarrow \mathsf{index\_to\_run}(f_{rev}, k')$ **oppure** $run \leftarrow \mathsf{rank}_h^{k'}(f_{rev})$
22: $symb \leftarrow \mathsf{get\_symbol}(start_{k'}, run)$
23: **while** $e > 0 \wedge z[e-1] = symb$ **do**
24: $e \leftarrow e - 1,\ f_{rev} \leftarrow \mathsf{reverse\_map}(e,\ f_{rev})$
25: $run \leftarrow \mathsf{index\_to\_run}(f_{rev}, e-1)$ **oppure** $run \leftarrow \mathsf{rank}_h^{e-1}(f_{rev})$
26: $symb \leftarrow \mathsf{get\_symbol}(start_{e-1}, run)$
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: $f_{rev} \leftarrow \mathsf{reverse\_map}(k',\ f_{rev}),\ k' \leftarrow k' - 1$
35: $run \leftarrow \mathsf{index\_to\_run}(f_{rev}, k')$ **oppure** $run \leftarrow \mathsf{rank}_h^{k'}(f_{rev})$
36: $symb \leftarrow \mathsf{get\_symbol}(start_{k'}, run)$
37: **while** $e > 0 \wedge z[e-1] = symb$ **do**
38: $e \leftarrow e - 1,\ f_{rev} \leftarrow \mathsf{reverse\_map}(e,\ f_{rev})$
39: $run \leftarrow \mathsf{index\_to\_run}(f_{rev}, e-1)$ **oppure** $run \leftarrow \mathsf{rank}_h^{e-1}(f_{rev})$
40: $symb \leftarrow \mathsf{get\_symbol}(start_{e-1}, run)$
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

Tra queste soluzioni, le uniche differenze si riscontrano nei tempi d'esecuzione e nella memoria richiesta.
Si supponga di aver computato l'array delle matching statistics fino alla colonna $k-1$ e di processare la colonna corrente k. Sia i la riga della matrice PBWT che ha un match con il più lungo suffisso di $z[0,k-1]$ che è suffisso di almeno uno tra $x_0[0,k-1]$, ..., $x_{M-1}[0,k-1]$. Sia p la corrispondente riga, sul pannello in input X, della riga i sulla matrice PBWT. Tale valore è ottenibile tramite il prefix array, avendo $p = a_k[i]$. Si ha, denotando con $\mathsf{lcs}(A,B)$ il più lungo suffisso comune tra le stringhe A e B, che:

$$|\mathsf{lcs}(z[0,k-1], x_p[0,k-1])| \geq |\mathsf{lcs}(z[0,k-1], x_{a_k[j]}[0,k-1])|, \ \forall\, j \in [0,M-1] \quad (3.45)$$

Qualora si abbia $y_i^k[k] = z[k]$, ovvero un match tra il k-esimo carattere della query e il carattere in riga i della matrice PBWT, si avrebbe che $\mathsf{MS}[k].\mathsf{row} = a_k[i] = p$ e $\mathsf{MS}[k].\mathsf{len} = \mathsf{MS}[k-1].\mathsf{len} + 1$, in quanto si seguirebbe la medesima riga dello step precedente del calcolo delle matching statistics.
In caso contrario, si avrebbe $y_i^k[k] \neq z[k]$, ovvero un mismatch tra il k-esimo carattere della query è il carattere in riga i della matrice PBWT. Bisogna, quindi, scegliere un nuovo indice i' nella matrice PBWT, corrispondente all'indice p' nella pannello in input, che, a sua volta, corrisponde alla riga con il suffisso più lungo possibile comune con la query, che sia estendibile anche in colonna k. Sia $y^k_{[s,e]}[k]$ l'intervallo corrispondente alla run che contiene l'indice i, nella matrice PBWT. Il più lungo suffisso di $z[0,k]$, che è suffisso di almeno uno tra $x_0[0,k]$, ..., $x_{M-1}[0,k]$, corrisponde alla fine della run precedente di simboli $\sigma = z[k]$ o all'inizio della run successiva di simboli $\sigma = z[k]$, nella colonna k. Formalmente, tale suffisso corrisponde a uno tra $X_{a_k[s-1]}[0,k]$, se $s > 0$, e $X_{a_k[e+1]}[0,k]$, se $e < M-1$. L'uso delle threshold permette di capire quale tra le righe $a_k[s-1]$ e $a_k[e+1]$ del pannello in input, se esistenti, abbia il più lungo suffisso comune con $z[0,k]$.
Sia t l'indice della threshold nella run corrente. Si hanno due casi possibili:

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

$$|\mathsf{lcs}(z[0,k], x_{a_k[s-1]}[0,k])| \geq |\mathsf{lcs}(z[0,k], x_{a_k[e+1]}[0,k])| \quad (3.46)$$

   Quindi si ha che $\mathsf{MS}[k].\mathsf{row} = a_k[s-1] = p$ e il mapping potrà proseguire dall'indice $s-1$

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

$$|\mathsf{lcs}(z[0,k], x_{a_k[s-1]}[0,k])| \leq |\mathsf{lcs}(z[0,k], x_{a_k[e+1]}[0,k])| \quad (3.47)$$

   Quindi si ha che $\mathsf{MS}[k].\mathsf{row} = a_k[e+1]$ e il mapping potrà proseguire dall'indice $e+1$

Si ricorda il caso in cui la threshold sia posta a fine run, nel caso della componente `THR-BV`. In tale situazione, bisognerebbe scegliere la testa della run successiva, qualora l'indice i si trovi esattamente a fine run. Invece, bisognerebbe scegliere la coda della run precedente qualora la threshold sia a fine run, a causa del fatto che il minimo RLCP si trovi nella testa della run successiva. L'unico modo per disambiguare è effettuare random access al pannello o calcolare una LCE query, per vedere quale, tra la coda della run precedente e la testa della run successiva, sia relativa alla riga del pannello originale con un suffisso comune alla query più lungo.
Una volta computati tutti i valori $\mathsf{MS}[i].\mathsf{row}$, per calcolare i valori $\mathsf{MS}[i].\mathsf{len}$, si scorre da sinistra a destra calcolando la lunghezza del match fino alla colonna i, facendo random access al pannello e confrontando la query z con la riga $\mathsf{MS}[i].\mathsf{row}$. Si assuma di aver calcolato $\mathsf{MS}[i-1].\mathsf{len}$ e di voler calcolare $\mathsf{MS}[i].\mathsf{len}$. Si hanno tre casi possibili:

1. $\mathsf{MS}[i].\mathsf{row} = M$. In tal caso, avendo segnalata l'inesistenza di un match terminante in colonna i, si ha che $\mathsf{MS}[i].\mathsf{len} = 0$

2. $\mathsf{MS}[i].\mathsf{row} = \mathsf{MS}[i-1].\mathsf{row}$, avendo $i \neq 0$ e $\mathsf{MS}[i-1].\mathsf{len} \neq 0$. In tal caso, si sta seguendo la medesima riga ottenuta in colonna $i-1$ e quindi $\mathsf{MS}[i].\mathsf{len} = \mathsf{MS}[i-1].\mathsf{len} + 1$, dovendo conteggiare il carattere della colonna corrente

3. in qualsiasi altro caso bisogna confrontare, a partire dalla colonna i, la query z con la riga $\mathsf{MS}[i].\mathsf{row}$ 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 come $\mathsf{MS}[i].\mathsf{len} = l$

Esempio 28. *Si vede un esempio di funzionamento delle threshold per la scelta della riga da memorizzare in* $\mathsf{MS}[i].\mathsf{row}$ *dopo un mismatch.*
Si prende il pannello visto all'esempio 15 e si effettua 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 prende la seconda run, di simboli $\sigma = 1$, indicizzata tra 17 e 18.
Si suppone 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 RLCP *si trovi, per la run, all'indice 18 (a causa del fatto che il minimo* RLCP *è all'indice 19, quello della testa della run successiva). L'indice in cui ci si trova, il 17, è quindi sopra la threshold. Questo significa che il suffisso comune più lungo con la query si ha con la riga 16 del pannello, per la definizione di threshold, avendo che questa sarà memorizzata nell'array* MS *(*$\mathsf{MS}[2].\mathsf{row} = 16$*). Successivamente, sfruttando* $\mathsf{MS}[1].\mathsf{len}$ *o tramite random access al testo, confrontando la riga x_{16} e la query z fino alla colonna $k = 2$, si può calcolare che* $\mathsf{MS}[2].\mathsf{len} = 3$.

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

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

Queste condizioni segnalano che non è possibile estendere a destra il più lungo suffisso comune, terminante in colonna i, tra la query e una qualsiasi riga del pannello di input. Questo si può verificare anche nell'esempio 20.

L'algoritmo per il il calcolo degli SMEM 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 quella per effettuare il mapping. Si noti che è possibile usare la funzione w, spiegata in precedenza, in quanto, avendo per costruzione $y^k_{curr\_index}[k] = z[k]$, si ha che la funzione segue esattamente una certa riga da una colonna alla successiva nella matrice PBWT.
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 $MS[i]$.row varia a seconda dell'uso della componente `MAP-INT` o `MAP-BV`. Il costo della funzione `down`, che risolve l'eventuale ambiguità della threshold a fine run, è variabile a seconda della componente usata per il random access (e dell'eventuale componente `LCE`) e risulta trascurabile vista la bassa frequenza d'uso, in termini probabilistici. Si ha, quindi, che, con ρ numero medio di run per colonna, usando `MAP-INT`, il tempo è proporzionale a:
$$\mathcal{O}(N \log \rho) \tag{3.48}$$
Mentre, usando la componente `MAP-BV`, è proporzionale a:
$$\mathcal{O}\left(N \log \frac{M}{\rho}\right) \tag{3.49}$$

- il calcolo dei valori MS$[i]$.len (e degli SMEM direttamente calcolabili da essi) 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$[i-1]$. Ipotizzando un caso peggiore dove si ha necessità di accedere al pannello in ogni colonna, nel caso della componente `RA-BV`, il calcolo complessivo delle lunghezza è proporzionale a:
$$\mathcal{O}(N^2) \tag{3.50}$$
Mentre, con l'uso della componente `RA-SLP`, la complessità in tempo è proporzionale a:
$$\mathcal{O}\left(N^2 \log(NM)\right) \tag{3.51}$$
In aggiunta bisogna considerare i costi della componente `PHI` per il computo di tutti gli SMEM.

Tali complessità teoriche sono fortemente sovrastimate.
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 in tempo. Mentre, per il ragionamento inverso, la struttura `MAP-INT + THR-INT + RA-BV + PERM + PHI` si ritiene sia quella con le migliori performance, dal punto di vista del tempo macchina.
In termini di memoria la struttura `MAP-INT + THR-INT + RA-SLP + PERM + PHI` risulta essere la più vantaggiosa mentre quella `MAP-BV + THR-BV + RA-BV + PERM + PHI` la peggiore, per le stime sulle singole componenti (usando i bitvector sparsi per mapping/threshold e non usando l'SLP per il random access).

Calcolo dell'array delle matching statistics tramite LCE query

Grazie all'uso delle LCE query, è possibile calcolare l'array delle matching statistics in un solo scorrimento da sinistra a destra sull'aplotipo query. Infatti, è possibile usare le LCE query per calcolare non solo quale nuova riga scegliere in caso di mismatch con l'aplotipo query ma anche di computare la lunghezza del suffisso comune tra essa e l'aplotipo query. In tal modo, si calcolano nello stesso momento sia i valori di $\mathsf{MS}[i].\mathsf{row}$ che di $\mathsf{MS}[i].\mathsf{len}$ (e di conseguenza anche gli SMEM).
Tale soluzione è quindi relativa alle seguenti strutture dati composte:

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

Con la notazione $\mathsf{lce}(k, x, y)$, si indica il calcolo della LCE query, terminante in colonna $k-1$ (quindi escludendo la colonna k-esima), tra le righe del pannello di indice x e indice y.
Per convenzione, si inizia la computazione dall'ultima riga della prima colonna. Si supponga di avere calcolato l'array MS, per una query z rispetto al pannello X, 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 corrisponde a quella, in X, che ha il più lungo suffisso comune con $z[0, k-1]$. Assumendo che l'indice i appartenga alla run r, di simboli σ, con testa di indice s e coda di indice e, si hanno diversi casi:

1. $z[k] = y_i^k[k] = \sigma$, quindi la riga i può essere usata per estendere il match e per proseguire col mapping in colonna $k+1$, avendo che $\mathsf{MS}[k].\mathsf{row} = \mathsf{MS}[k-1].\mathsf{row}$ e $\mathsf{MS}[k].\mathsf{len} = \mathsf{MS}[k-1].\mathsf{len} + 1$
2. $z[k] \neq y_i^k[k] = \sigma$ e si ha una sola run in colonna k, avendo che non si possono avere match in quella colonna. Per convenzione, si memorizzano $\mathsf{MS}[k].\mathsf{row} = M$ e $\mathsf{MS}[k].\mathsf{len} = 0$ e si ricomincia, in colonna $k+1$, dall'ultima posizione, che corrisponde, nel pannello originale, alla riga specificata dal valore del prefix array sample della coda dell'ultima run

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`.

function EXTERNAL_MATCHES(z)
  $ms_{row} \leftarrow [0..0],\ ms_{len} \leftarrow [0..0]$ ▷ vettore MS di lunghezza $|z|$
  $curr_{row} \leftarrow rlpbwt[0].samples_{end}[|rlpbwt[0].samples_{end}| - 1]$
  $curr_{index} \leftarrow curr_{row}$
  $curr_{run} \leftarrow \mathsf{index\_to\_run}(curr_{index}, 0)$ **oppure** $curr_{run} \leftarrow \mathsf{rank}_h^0(curr_{index})$
  $symb \leftarrow \mathsf{get\_symbol}(start_0, curr_{run})$ ▷ **Costruzione righe dell'array** MS
  for *every* $k \in [0, |z|)$ **do**
    if $z[i] = symb$ **then**
      $ms_{row}[k] \leftarrow curr_{row}$
      if $k \neq |z| - 1$ **then** $(curr_{index},\ curr_{run},\ symb) \leftarrow \mathsf{update}(k, curr_{index}, z)$
    else
      $curr_{thr} \leftarrow t_k[curr_{run}]$ **oppure** $curr_{thr} \leftarrow \mathsf{rank}_t^k(curr_{index})$
      $force_{down} \leftarrow \top$ *sse l'indice è sovrapposto ad una threshold non in coda di run*
      $force_{down} \leftarrow \top$ *sse l'indice è sovrapposto ad una threshold in coda di run e* `DOWN`$(\ldots) = \top$
      if $|samples^k_{beg}| = 1$ **then**
        $ms_{row}[k] \leftarrow M$
        if $k \neq |z| - 1$ **then**
          $curr_{row} \leftarrow rlpbwt[k+1].samples_{end}[|rlpbwt[k+1].samples_{end}| - 1]$
          $curr_{index} \leftarrow M - 1$
          $curr_{run} \leftarrow \mathsf{index\_to\_run}(curr_{index}, k+1)$ **oppure** $curr_{run} \leftarrow \mathsf{rank}_h^{k+1}(curr_{index})$
          $symb \leftarrow \mathsf{get\_symbol}(start_{k+1}, curr_{run})$
      else if $(curr_{run} \neq 0 \wedge curr_{run} = curr_{thr} \wedge \neg down) \vee curr_{run} = |samples^k_{beg}| - 1$ **then**
        $curr_{index} \leftarrow p_k[curr_{run}] - 1$ **oppure** $curr_{index} \leftarrow \mathsf{select}_h^k(curr_{run})$
        $curr_{row} \leftarrow samples^k_{end}[curr_{run} - 1]$
        $ms_{row}[k] \leftarrow curr_{row}$
        if $k \neq |z| - 1$ **then** $(curr_{index},\ curr_{run},\ symb) \leftarrow UPDATE(k, curr_{index}, z)$
      else
        $curr_{index} \leftarrow p_k[curr_{run} + 1]$ **oppure** $curr_{index} \leftarrow \mathsf{select}_h^k(curr_{run} + 1) + 1$
        $curr_{row} \leftarrow samples^k_{beg}[curr_{run} + 1]$
        $ms_{row}[k] \leftarrow curr_{row}$
        if $k \neq |z| - 1$ **then** $(curr_{index},\ curr_{run},\ symb) \leftarrow \mathsf{update}(k, curr_{index}, z)$
  for *every* $k \in [0, |z|)$ **do** ▷ **Costruzione lunghezze dell'array** MS
    if $ms_{row}[k] = M$ **then**
      $ms_{len}[k] \leftarrow 0$
    else if $k \neq 0 \wedge ms_{row}[i] = ms_{row}[i-1] \wedge ms_{len}[i-1] \neq 0$ **then**
      $ms_{len}[i] \leftarrow ms_{len}[i-1] + 1$
    else ▷ ra effettua il random access con la componente `RA-BV` o `RA-SLP`
      $tmp_{index} \leftarrow i,\ tmp_{len} \leftarrow 0$
      while $tmp_{index} \geq 0 \wedge z[tmp_{index}] = ra(ms_{row}[k], tmp_{index})$ **do**
        $tmp_{index} \leftarrow tmp_{index} - 1,\ tmp_{len} \leftarrow tmp_{len} + 1$
      $ms_{len}[k] \leftarrow tmp_{len}$
  for *every* $k \in [0, |z|)$ **do** ▷ **Calcolo dei match da** MS
    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**
      report dello SMEM terminante in colonna k
      SMEM di lunghezza $ms_{len}[k]$ *con la riga* $ms_{row}[k]$ *e quelle estese da essa tramite* `PHI`
function DOWN($pos, prev, next$)
  si usano le LCE queries o il random access per calcolare il suffisso comune più lungo tra quelli delle righe pos/prev e pos/next fino alla colonna precedente a quella corrente
  se il secondo è maggiore o uguale al primo ritorna $\top$*, altrimenti* $\bot$

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

```
1: function UPDATE(k, curr_index, z)
2:     curr_index ← w(k, curr_index, z[k])
3:     curr_run ← index_to_run(curr_index, k + 1) oppure curr_run ← rank_h^{k+1}(curr_index)
4:     symb ← get_symbol(start_{k+1}, curr_run)
5:     return (curr_index, curr_run, symb)
```

3. $z[k] \neq y_i^k[k] = \sigma$ ma si hanno anche altre run, dovendo quindi scegliere la nuova riga da seguire. Si ha che il più lungo suffisso di $z[0,k]$, che è anche suffisso di $x_0[0,k], \ldots, x_{M-1}[0,k]$, è uno tra:

   - $x_{a_k[s-1]}$, se $s \neq 0$, ovvero la riga del pannello che corrisponde alla coda della run precedente a quella corrente
   - $x_{a_k[e+1]}$, se $e \neq M-1$, ovvero la riga del pannello che corrisponde alla testa della run successiva a quella corrente

   Questo fatto è dovuto all'ordinamento lessicografico inverso, che si ha per la costruzione della PBWT. Avendo quindi i prefix array sample, che ci dicono a quale riga nel pannello corrispondano tali valori, e conoscendo $\mathsf{MS}[k-1].\mathsf{row}$, è possibile calcolare $\mathsf{lce}(k, \mathsf{MS}[k-1].\mathsf{row}, a_k[s-1])$ e $\mathsf{lce}(k, \mathsf{MS}[k-1].\mathsf{row}, a_k[e+1])$. Si sceglie il suffisso comune più lungo tra le due, ovvero il più lungo risultato tra le due funzioni lce, e la riga corrispondente per proseguire. Si ha quindi $\mathsf{MS}[k].\mathsf{row} = a_k[s-1]$ o $\mathsf{MS}[k].\mathsf{row} = a_k[e+1]$ (in memoria nella componente `PERM`). In merito al campo len, assumendo che la lunghezza maggiore delle due LCE query sia l, si ha che:

$$\mathsf{MS}[k].\mathsf{len} = \min(\mathsf{MS}[k-1].\mathsf{len}, l) + 1 \tag{3.52}$$

   Questa assegnazione si ha in quanto la LCE query potrebbe restituire un valore più lungo dell'effettivo match con la query z. Si sceglie, di conseguenza, il minimo tra le due lunghezze per considerare l'overlap, ottenendo l'effettiva lunghezza del suffisso comune tra z e la nuova riga scelta, fino alla colonna $k-1$, incrementandolo di uno per conteggiare il match ottenuto in colonna k

Esempio 29. *Si riprende l'esempio 28, visto per il calcolo di* $\mathsf{MS}[i].\mathsf{row}$*, dopo un mismatch, tramite threshold.*
Senza usare le threshold, si dovrebbero calcolare, avendo $\mathsf{MS}[1].\mathsf{row} = 19$ *e* $\mathsf{MS}[1].\mathsf{len} = 2$*:*

$$\mathsf{lce}(2, x_{19}, x_{16}) = \text{"01"} \implies |\mathsf{lce}(2, x_{19}, x_{16})| = 2$$
$$\mathsf{lce}(2, x_{19}, x_{17}) = \text{"1"} \implies |\mathsf{lce}(2, x_{19}, x_{17})| = 1$$

Come verificabile dal pannello presente all'esempio 15.
Si ha quindi che $\mathsf{MS}[2].\mathsf{row} = 16$ *e che:*

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

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

- non si necessita di tenere in memoria le informazioni per le threshold
- è possibile il calcolo dell'array MS in una singola scansione della query
- non si necessita di memorizzare l'intero array MS, ma solamente quattro variabili relative alla coppia (row, len) corrente e a quella precedente. Infatti, per computare i valori in colonna $k+1$ dell'array MS e gli SMEM terminanti in colonna $k+1$, si necessita solo delle informazioni in colonna k.
  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 tramite LCE query, si hanno variazioni solo in base alla componente per il mapping. Nel caso della componente `MAP-INT`, avendo ρ numero medio di run per colonna, dovendo iterare la query, fare il mapping e usare la componente `LCE`, si ha un tempo proporzionale a:

$$\mathcal{O}(N(\log \rho + \log(NM))) \tag{3.53}$$

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

$$\mathcal{O}\left(N\left(\log \frac{M}{\rho} + \log(NM)\right)\right) \tag{3.54}$$

Infine, per il calcolo di tutte le righe del pannello per cui si ha uno SMEM, bisogna considerare quando analizzato per la componente `PHI`.
Si deduce 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 essere anche la migliore in termini di memoria, per quanto discusso in merito a intvector compressi e bitvector sparsi nella sezione 3.3.
Si vedrà, sperimentalmente, nel Capitolo 4, il confronto con le altre strutture dati. Una prima intuizione è quella che, usando le LCE query, si hanno sicuramente, a parità di componenti per il mapping, tempi peggiori rispetto all'uso della componente `RA-BV`, come mostrato dalle complessità temporali. Un confronto con le strutture basate su `RA-SLP` risulta più complesso da analizzare, limitandosi alle stime asintotiche e quindi è necessaria un'analisi sperimentale.

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: $curr_{run} \leftarrow \mathsf{index\_to\_run}(curr_{index}, 0)$ **oppure** $curr_{run} \leftarrow rank_h^0(curr_{index})$
5: $symb \leftarrow \mathsf{get\_symbol}(start_0, curr_{run})$ ▷ **Costruzione dell'array** MS
6: **for** *every* $k \in [0, |z|)$ **do**
7: **if** $z[i] = symb$ **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** $(curr_{index},\ curr_{run},\ symb) \leftarrow \mathsf{update}(k, curr_{index}, z)$
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: $curr_{run} \leftarrow \mathsf{index\_to\_run}(curr_{index}, k+1)$ **oppure** $curr_{run} \leftarrow rank_h^{k+1}(curr_{index})$
19: $symb \leftarrow \mathsf{get\_symbol}(start_{k+1}, curr_{run})$
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: $lce \leftarrow \mathsf{lce}(k, curr_{row}, prev_{row})$
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** $(curr_{index},\ curr_{run},\ symb) \leftarrow \mathsf{update}(k, curr_{index}, z)$
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: $lce \leftarrow \mathsf{lce}(k, curr_{row}, next_{row})$
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** $(curr_{index},\ curr_{run},\ symb) \leftarrow \mathsf{update}(k, curr_{index}, z)$
35: **else**
36: $prev_{row} \leftarrow samples_{end}^k[curr_{run} - 1],\ next_{row} \leftarrow samples_{beg}^k[curr_{run} + 1]$
37: $lce \leftarrow \max(|\mathsf{lce}(k, curr_{row}, prev_{row})|, |\mathsf{lce}(k, curr_{row}, next_{row})|)$
38: $curr_{row} \leftarrow lce_{row}$ ▷ lce_{row} è 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** $(curr_{index},\ curr_{run},\ symb) \leftarrow \mathsf{update}(k, curr_{index}, z)$
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

Si riportano alcuni risultati sperimentali relativi alle diverse implementazioni 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) [38], progetto, nato nel 2008, il quale ha visto lo sforzo della comunità scientifica internazionale per la catalogazione delle varianti geniche umane. Il 1KGP risulta essere uno dei più importanti progetti nell'ambito bioinformatico. 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 strutture dati composte 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 per intvector compressi, bitvector, bitvector sparsi, serializzazione e varie utility per il calcolo della memoria delle strutture dati
- BigRePair e ShapedSlp per la costruzione e l'uso degli SLP

L'implementazione delle strutture composte per la RLPBWT supporta lo studio parallelo di più query tramite openMP [39]. Al fine di un più corretto confronto con l'implementazione della PBWT, 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[1]. Tale implementazione è scritta in linguaggio C. L'implementazione fornisce tre algoritmi per il calcolo degli SMEM, avendo N siti, M sample e Q query, per i quali si riportano le complessità asintotiche specificate nei commenti del codice:

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 che, in pratica, essa corrisponda a $13NM$ byte 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, dovendo fare, a differenza dell'algoritmo `matchIndexed` e degli algoritmi per la RLPBWT, una sola scansione della struttura dati per tutte le query. 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,

[1] `https://github.com/richarddurbin/pbwt`

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 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 anche delle query, 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 [2] [38].
Tali pannelli, disponibili in formato VCF (Variant Call Format) [40], 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 pannelli costruiti su un alfabeto binario $\Sigma = \{0, 1\}$, tramite l'uso di *bcftools* [41], 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, ad eccezione di uno molto grande. 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 che anche il relativo pannello delle varianti 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 attesa 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. Infatti, avendo pochi simboli $\sigma = 1$ in ogni colonna, simboli che possono anche essere nella medesima run dopo la permutazione data dalla PBWT, si producono, nel complesso, poche run. Si ricordi, inoltre, che tale permutazione,

[2] `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

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 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 del numero di run per colonna 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 4.908 in quanto, si anticipa, 100 righe sono state utilizzate come query nelle successive fasi della sperimentazione. Si conferma 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 di run (ad eccezione della componente RLCP). 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 sopratutto mediana del numero di run per colonna risultino 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[3].
L'intera pipeline è stata gestita tramite Snakemake [42] (un workflow management system), che è 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 una lista di pannelli con associato il numero di query:

- lo scaricamento dei tool e delle dipendenze per la PBWT di Durbin e la RLPBWT proposta in questa tesi

[3] https://github.com/dlcgold/rlpbwt-test

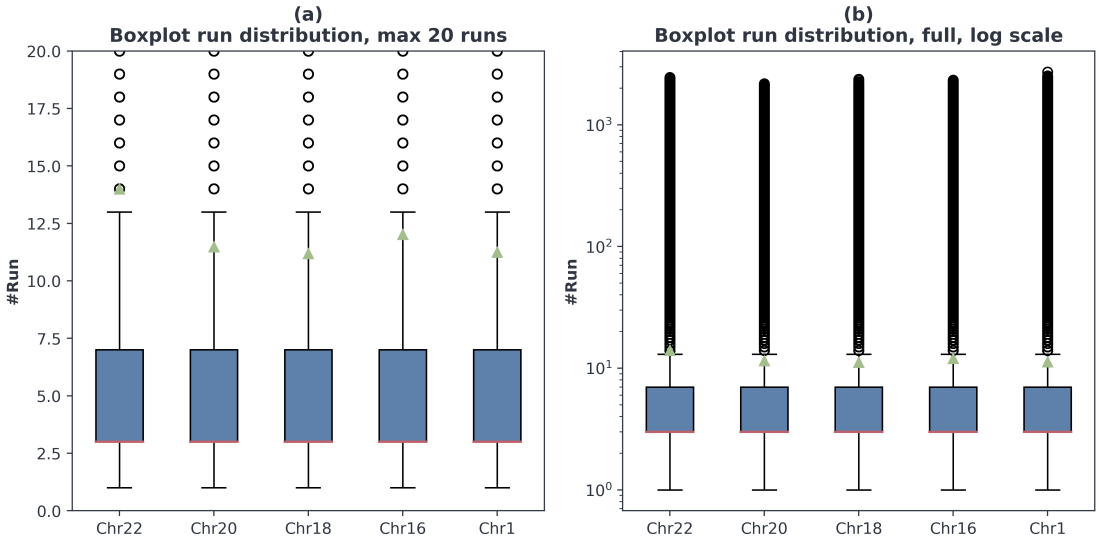


Figura 4.1: Boxplot della distribuzione delle run per i pannelli dei cinque cromosomo studiati. Il grafico (a) presenta uno zoom che esclude la maggior parte degli outlier mentre il grafico (b) presenta, in scala logaritmica, il boxplot completo con tutti gli outlier.

- la produzione dell'input per la PBWT e della RLPBWT per la quantità di query richiesta
- la produzione delle strutture dati
- l'esecuzione degli algoritmi per il calcolo degli SMEM

Al fine di ottenere risultati non banali, si è deciso di estrarre dai pannelli 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 (2,40GHz), 756GB di RAM, 768GB di swap e sistema operativo Ubuntu 20.04.4 LTS. Tale macchina è stata gentilmente messa a disposizione dalla University of Florida.

DC la pic va aggiornata e va aggiunto link github dopo sistemazione pipeline

4.2 Risultati della sperimentazione

Si presentano ora i risultati quantitativi della sperimentazione effettuata sui cinque pannelli scelti. Come anticipato, al fine del computo degli SMEM, avendo un

Figura 4.2: Regole usate in Snakemake per la sperimentazione. Si hanno il download dei vari software, la compilazione degli stessi, la produzione delle strutture dati, il calcolo degli SMEM, l'estrazione dei risultati quantitativi e la produzione dei file CSV finali.

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.

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 [43], usato come formato di input nella RLPBWT
  - l'estrazione del pannello delle query e la creazione del nuovo pannello di aplotipi
  - la produzione dell'SLP del pannello di aplotipi, 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
- l'esecuzione degli algoritmi per 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 ininfluente e, di fatto, anche la conversione tra i due formati per l'input non necessita particolari considerazioni, come inferibile dai risultati numerici delle performance. Si segnala che 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 gigabyte di RAM mentre i tempi di calcolo sono nell'ordine delle ore. Prendendo in analisi il cromosoma 1, si ha che è richiesto un picco di 644GB di RAM, avendo che l'intera esecuzione richiede circa 4 ore. 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

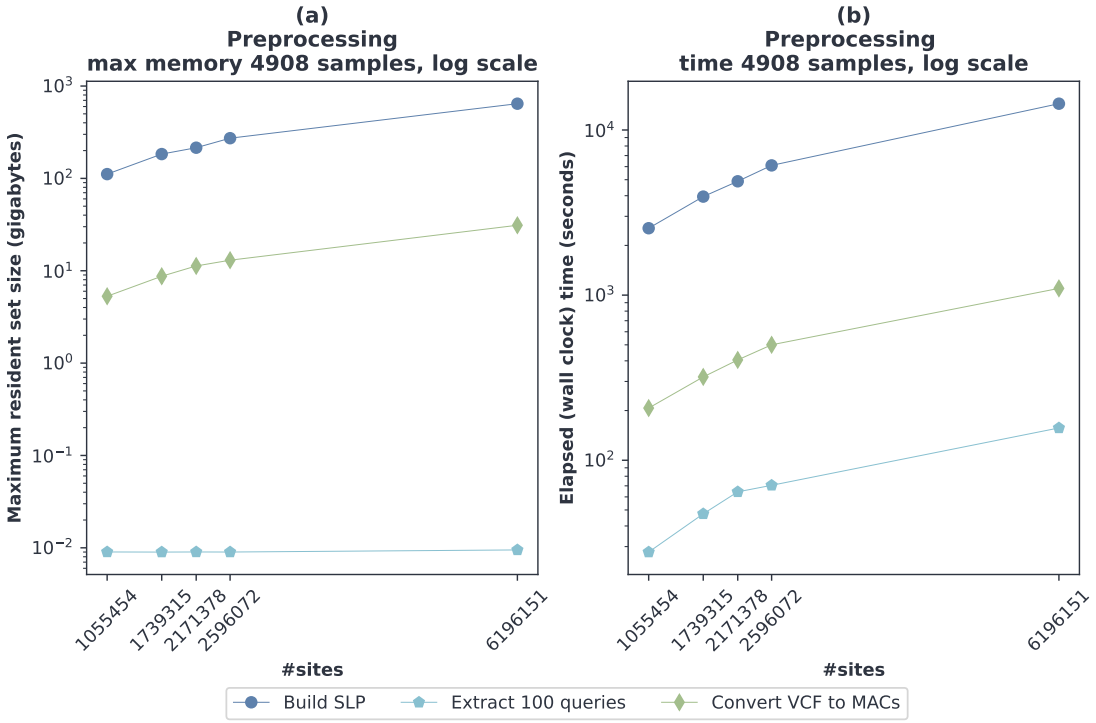


Figura 4.3: Picchi di memoria (a) e tempo richiesto (b) per le tre fasi di preprocessing dell'input per la RLPBWT, in scala logaritmica.

direzione potrebbero lasciar spazio a diverse ottimizzazioni. Bisogna, infine, considerare che questa fase è necessaria solo per quattro delle otto soluzioni studiate per la RLPBWT, ovvero quelle che utilizzano o la componente `RA-SLP` o la componente `LCE`. Inoltre, il fatto che questa costruzione sia necessaria solo una volta è cruciale nell'ottica di un confronto con lo spazio richiesto dall'algoritmo 5 di Durbin, che richiede $13NM$ byte ad ogni esecuzione.
In generale, la crescita di memoria richiesta e di tempo, per le tre fasi di preprocessing, sembra essere proporzionale al numero di siti dei pannelli, come atteso.
In figura 4.4 si può osservare il vantaggio in termini di memoria che si ha con l'uso degli SLP, confrontando il peso dei file MACs con il peso delle grammatiche compresse. Si segnala che il peso dei vari file MACs include anche diversi header. In tabella 4.4 si possono confrontare quantitativamente tali risultati, notando come usare l'SLP richieda circa l'1% della memoria necessaria al file non compresso. È possibile, quindi, 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. Per riferimento, comprime-

Tabella 4.2: Risultati quantitivi dei picchi di memoria (gigabyte) 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 quantitivi 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

Tabella 4.4: Dimensioni, in gigabyte, degli SLP e dei file MACs per i vari pannelli del 1000 Genome Project.

Chr	**SLP**	**MACs**
chr22	0,04	4,84
chr20	0,06	7,98
chr18	0,08	9,97
chr16	0,10	11,91
chr1	0,22	28,44

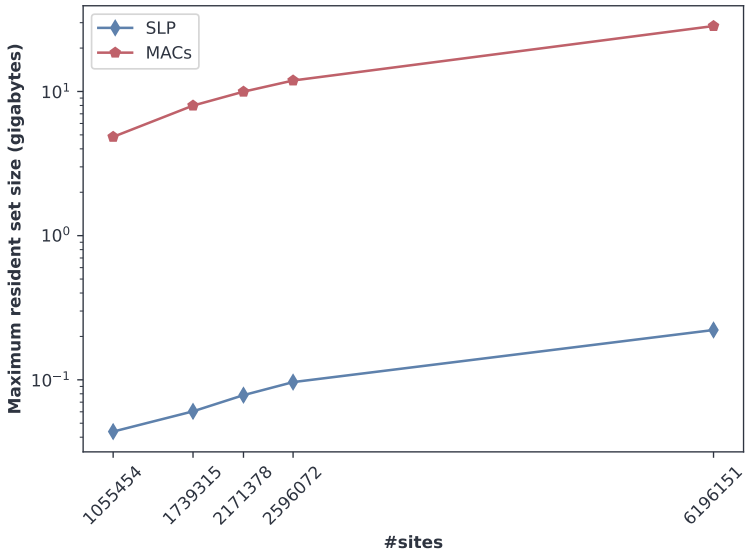


Figura 4.4: Confronto tra la memoria richiesta dai file MACs e dagli SLP per i pannelli del 1000 Genome Project, in scala logaritmica.

re il pannello relativo al cromosoma 1, con una tecnica standard (GZip), richiede 0,5GB.

Costruzione della struttura. Si analizzano, ora, 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 necessari 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. Alla tabelle 4.5 e 4.6 si riportano i risultati quantitativi in termini di gigabyte 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 + RLCP`

e `MAP-BV + RLCP`, dovendo memorizzare l'intero insieme degli array RLCP, hanno un elevato consumo di memoria. Pur utilizzando degli intvector compressi, in modo analogo a quanto visto, ad esempio, per la componente `PERM`, si ha necessità di salvare NM valori interi. 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` richiede maggior memoria (approssimativamente tra il 15% e il 20% in più) della componente `MAP-INT`, coerentemente con quanto analizzato in Sezione 3.3. 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 sui 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 d'uso di bitvector sparsi e intvector 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 d'uso dell' SLP, per l'uso delle LCE query. Si nota come l'uso della componente `RA-BV` comporti, 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) rispetto alle strutture basate su `RA-SLP/LCE`. Confrontando i tempi di tutte le varianti, si ha che tutti gli algoritmi di costruzione sono in tempo proporzionale a $\mathcal{O}(NM)$ ma, come detto, le varianti della RLPBWT includono, in questa fase, anche il calcolo degli indici utili al calcolo degli SMEM.
Si conclude che, 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 intvector 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 questo risultato conferma quanto discusso nel capitolo 3.

Sfruttando i metodi offerti dalla SDSL, 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 megabyte, le dimensioni delle varie 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%.

DC Controllare tutti questi dati

Tabella 4.5: Risultati quantitativi dei picchi di memoria (gigabyte) di costruzione delle strutture dati.

Tabella A: risultati relativi alla *PBWT* e alle varianti basate su RLCP per la RLPBWT.

Chr	**PBWT**	`MAP-INT + RLCP`	`MAP-BV + RLCP`
`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` $\cdots$	`MAP/THR-BV + RA-SLP` $\cdots$	`MAP-BV + LCE` $\cdots$
`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 intvector compressi per la RLPBWT.

Chr	`MAP/THR-INT + RA-BV` $\cdots$	`MAP/THR-INT + RA-SLP` $\cdots$	`MAP-INT + LCE` $\cdots$
`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 RLCP per la RLPBWT.

Chr	**PBWT**	`MAP-INT + RLCP`	`MAP-BV + RLCP`
`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` $\cdots$	`MAP/THR-BV + RA-SLP` $\cdots$	`MAP-BV + LCE` $\cdots$
`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 intvector compressi per la RLPBWT.

Chr	`MAP/THR-INT + RA-BV` $\cdots$	`MAP/THR-INT + RA-SLP` $\cdots$	`MAP-INT + LCE` $\cdots$
`chr22`	143	137	115
`chr20`	232	219	192
`chr18`	288	275	238
`chr16`	347	330	285
`chr1`	778	739	652

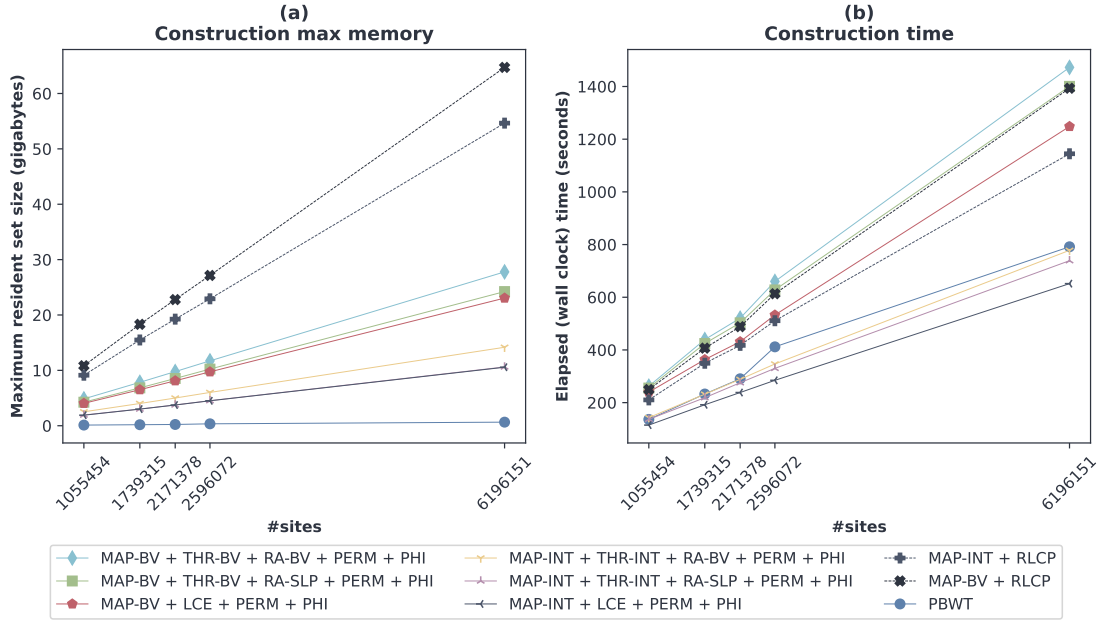


Figura 4.5: Picchi di memoria (a) e tempi di calcolo (b) per la costruzione delle varianti della RLPBWT e per la PBWT.

Numericamente tale vantaggio è riportato in tabella 4.7. Si segnala, nuovamente, il forte vantaggio in memoria nell'utilizzo delle componenti basate su intvector 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. Terminando l'analisi di tali risultati, senza trattare nuovamente le componenti per il random access, si ha conferma della richiesta eccessiva in memoria della componente `RLCP`.

DC Serve altro?

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

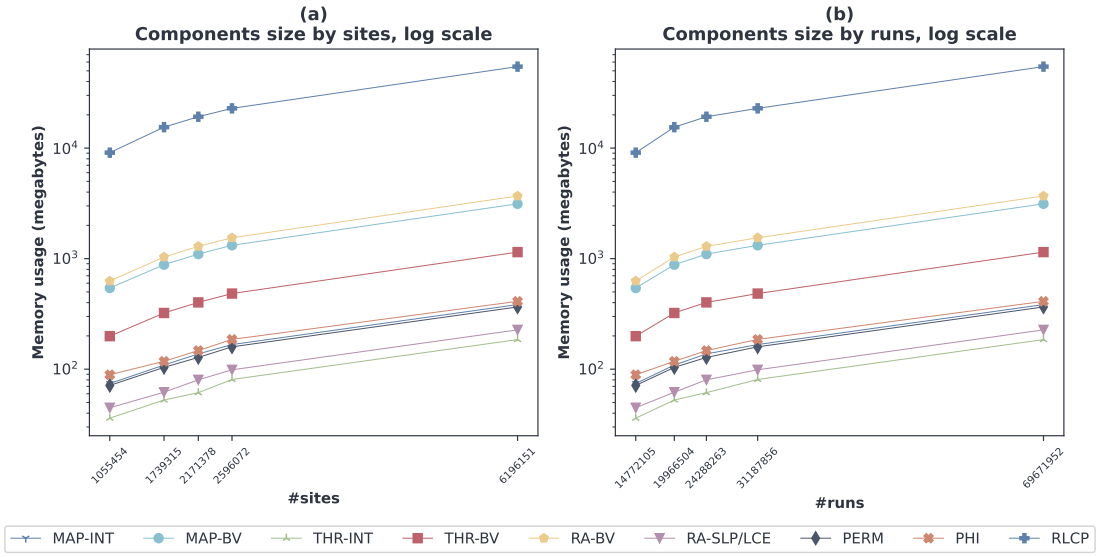


Figura 4.6: Memoria occupata dalle singole componenti, avendo sulle ascisse in (a) il numero di siti e in (b) il numero di run.

Tabella 4.8: Dimensioni, in megabyte, 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
chr22	74	543	36	199
chr20	109	882	53	322
chr18	137	1.100	61	402
chr16	167	1.320	81	483
chr1	384	3.133	185	1.146

Tabella B: dimensioni delle componenti di random access, prefix array sample, struttura ϕ e reverse longest common prefix.

Chr	RA-BV	RA-SLP/LCE	PERM	PHI	RLCP
chr22	628	44	71	89	9.095
chr20	1.035	62	104	118	15.468
chr18	1.292	80	128	147	19.223
chr16	1.545	99	159	186	22.888
chr1	3.687	227	366	411	54.588

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 quantitativamente 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. Invece, 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 $13NM$ byte. Escludendo le strutture `MAP-INT + RLCP` e `MAP-BV + RLCP`, si rileva 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/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 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), richiedono più tempo di tutte le altre varianti, soprattutto se si pensa alle corrispondenti varianti con componente `RA-BV`. L'uso della componente `RA-SLP` comporta, infatti, circa venti volte i tempi di calcolo che si hanno usando le componenti basate su bitvector sparsi e addirittura di circa 45 volte quelli che si hanno usando le strutture basate su intvector compressi per mapping e threshold. La struttura `MAP-INT + LCE + PERM + PHI` e la struttura `MAP-BV + LCE + PERM + PHI` risultano essere, al massimo, il doppio più lente rispetto all'algoritmo `matchIndexed`, con una differenze che diventa quasi trascurabile all'aumentare delle dimensioni del pannello. Questo è un risultato molto interessante, considerando la memoria necessaria per il calcolo degli SMEM delle varie implementazioni.
Confrontando l'uso delle LCE query con l'uso della componente `RA-BV` si hanno, invece, tempi triplicati nel caso d'uso di componenti per mapping e threshold con bitvector sparsi. Nel caso d'uso degli intvector 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

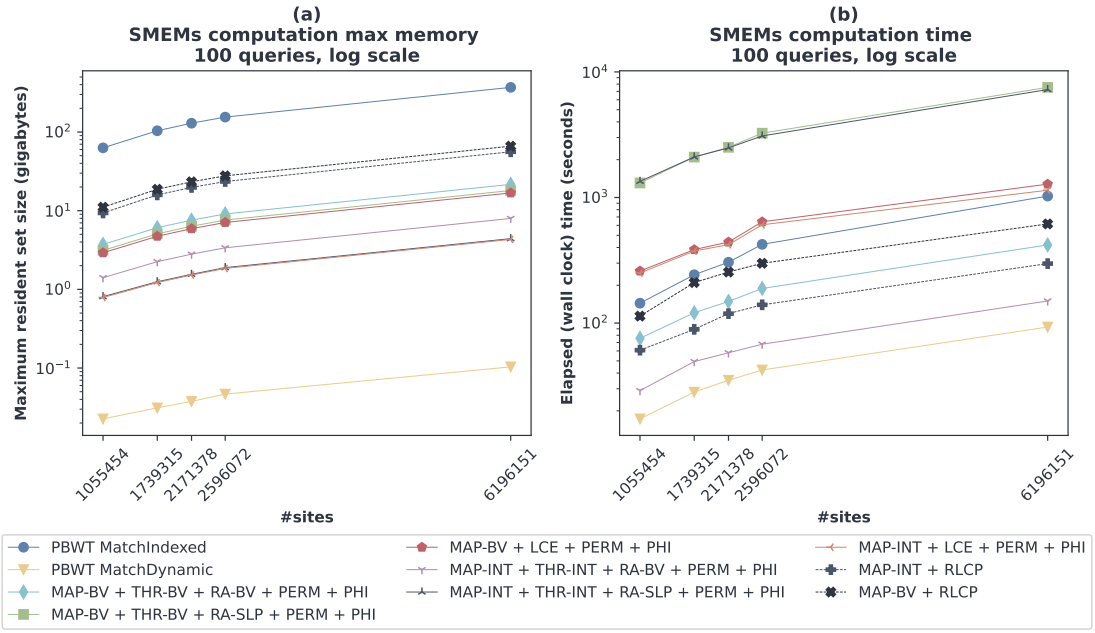


Figura 4.7: Picchi di memoria (a) e tempi di esecuzione (b) per il calcolo degli SMEM.

SLP è la peggior soluzione in ottica di 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. Questo risultato è coerente con quanto analizzato nel Capitolo 3. Notando come quest'ultima sia circa 10 volte più lenta della soluzione con `THR-INT` e `RA-BV`, 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 cui il risparmio di memoria 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 dal punto di vista del tempo macchina, si è deciso di isolare il calcolo degli SMEM per ogni singola query, valutando media e deviazione standard dell'esecuzione su 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 istruzioni per le 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 indi-

Tabella 4.9: Risultati quantitativi dei picchi di memoria (gigabyte) di costruzione delle strutture dati.

Tabella A: risultati relativi alla PBWT e alle varianti basate su RLCP per la RLPBWT.

Chr	`matchIndexed`	`matchDynamic`	`MAP-INT + RLCP`	`MAP-BV + RLCP`
`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	0,10	56	66

Tabella B: risultati relativi alle varianti basate su matching statistics e bitvector sparsi per la RLPBWT.

Chr	`MAP/THR-BV + RA-BV` $\cdots$	`MAP/THR-BV + RA-SLP` $\cdots$	`MAP-BV + LCE` $\cdots$
`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 intvector compressi per la RLPBWT.

Chr	`MAP/THR-INT + RA-BV` $\cdots$	`MAP/THR-INT + RA-SLP` $\cdots$	`MAP-INT + LCE` $\cdots$
`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 RLCP per la RLPBWT.

Chr	`matchIndexed`	`matchDynamic`	`MAP-INT + RLCP`	`MAP-BV + RLCP`
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` $\cdots$	`MAP/THR-BV + RA-SLP` $\cdots$	`MAP-BV + LCE` $\cdots$
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 intvector compressi per la RLPBWT.

Chr	`MAP/THR-INT + RA-BV` $\cdots$	`MAP/THR-INT + RA-SLP` $\cdots$	`MAP-INT + LCE` $\cdots$
chr22	29	1344	250
chr20	49	2103	375
chr18	58	2483	421
chr16	68	3092	606
chr1	150	7234	1142

ci 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 esclusa, in ogni caso, la costruzione della struttura base della PBWT.

DC Non se dire questa cosa

Tale risultato è visualizzabile in figura 4.8, dove si è deciso di escludere le strutture `MAP-INT + RLCP` e `MAP-BV + RLCP` 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 nel Capitolo 3 e nella sezione precedente. Nel caso dell'algoritmo `matchIndexed`, non misurando le istruzioni per computare tutti gli array necessari, si hanno tempi di calcolo degli SMEM migliori di quanto visto sull'esecuzione completa. 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 l'algoritmo `matchIndexed`. Infatti, per natura stessa dell'algoritmo, le operazioni sono studiate al fine di essere ottimizzate per pannelli di query e non per una query singola, avendo molte operazioni che potrebbero essere ottimizzate per il caso della singola query. Sperimentalmente, si è osservato 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 quantitativi delle due fasi di sperimentazione.

Tabella 4.11: Risultati quantitativi dei tempi (secondi) di calcolo degli SMEM su singola query. I risultati sono nella forma "media ± deviazione standard".

Tabella A: risultati relativi alla PBWT.

Chr	`matchIndexed`	`matchDynamic`
`chr22`	0,15 ± 0,01	18,94 ± 0,58
`chr20`	0,25 ± 0,01	28,90 ± 1,99
`chr18`	0,33 ± 0,02	37,24 ± 1,08
`chr16`	0,38 ± 0,01	45,37 ± 3,49
`chr1`	1,01 ± 0,10	88,73 ± 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 ± 0,28	16,51 ± 1,34	2,92 ± 0,55
`chr20`	1,08 ± 0,05	22,85 ± 2,21	4,57 ± 0,82
`chr18`	1,36 ± 0,05	27,23 ± 2,50	4,96 ± 0,74
`chr16`	1,54 ± 0,05	38,93 ± 2,70	7,65 ± 1,14
`chr1`	3,41 ± 0,07	74,54 ± 3,40	12,75 ± 1,82

Tabella C: risultati relativi alle varianti basate su matching statistics e intvector compressi per la RLPBWT.

Chr	`MAP/THR-INT + RA-BV ···`	`MAP/THR-INT + RA-SLP ···`	`MAP-INT + LCE ···`
`chr22`	0,24 ± 0,02	13,52 ± 1,26	2,44 ± 0,54
`chr20`	0,37 ± 0,03	20,55 ± 2,16	3,68 ± 0,74
`chr18`	0,45 ± 0,03	24,43 ± 2,38	3,86 ± 0,70
`chr16`	0,54 ± 0,03	30,86 ± 2,35	5,93 ± 1,01
`chr1`	1,21 ± 0,06	68,22 ± 3,02	11,12 ± 1,52

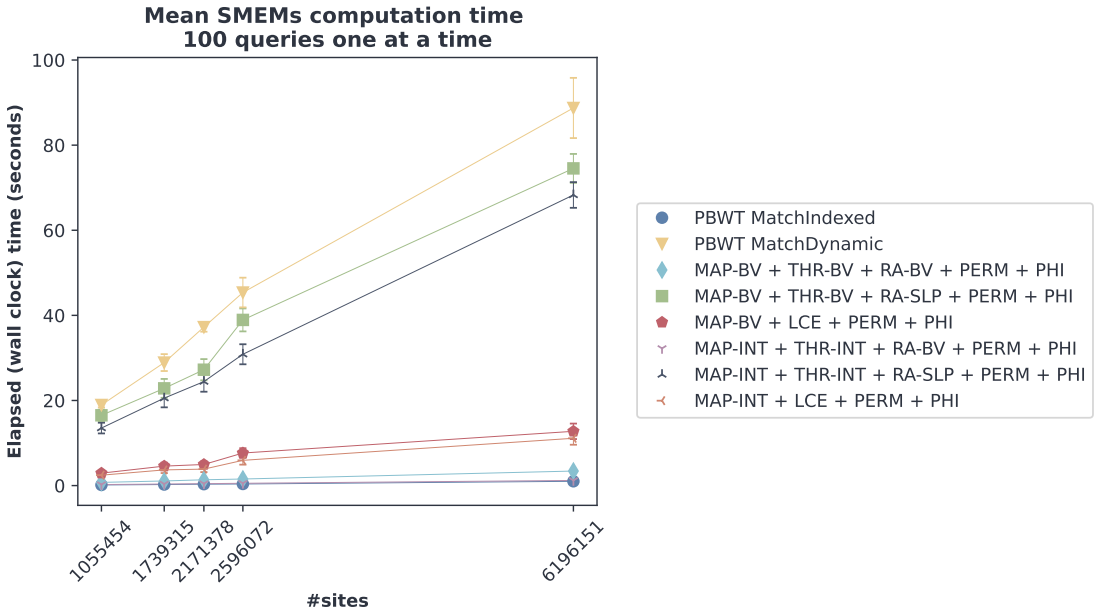


Figura 4.8: Tempo medio di esecuzione del calcolo degli SMEM per una singola query. Il grafico di destra è in scala logaritmica e, in entrambi, le barre d'errore rappresentano la deviazione standard.

Capitolo 5

Conclusioni

Fissato l'iniziale obbiettivo di risolvere le problematiche relative alla memoria richiesta dall'algoritmo 5 di Durbin, le implementazioni della RLPBWT proposte in questa tesi, principalmente nelle strutture dati composte segnalate nel Capitolo 4, hanno riportato risultati molto incoraggianti. Come descritto nel medesimo capitolo, la quantità di memoria richiesta per il calcolo degli SMEM risulta essere molto 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.
In termini di casi d'uso, ipotizzando un possibile futuro in cui diventi comune interrogare pannelli di aplotipi frequentemente, magari con singole query da un'interfaccia web o tramite API (alla stregua di quanto avviene quotidianamente, ad esempio, con BLAST), l'uso della RLPBWT, in una delle sue migliori varianti, avrebbe evidenti vantaggi in termini di tempo di calcolo degli SMEM, non solo rispetto all'algoritmo 5 di Durbin ma anche rispetto all'algoritmo `matchDynamic`, alla luce dei risultati su singole query. Inoltre, quest'ultimo richiederebbe ogni volta la ricostruzione della permutazione, a differenza delle implementazioni della RLPBWT che potrebbero essere costruite e caricate in memoria *una tantum*.
Si possono 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:

DC Frase da rivedere

- 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, qualora non fosse necessario tenere in memoria la struttura per rispondere a diverse query provenienti da diverse fonti

- 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 similare a quanto si ha con la componente `MAP-INT`

Inoltre, risulterebbe interessante, dal solo punto di vista teorico, una più approfondita caratterizzazioni delle complessità asintotiche, sia in spazio che in tempo. Questa analisi, a causa degli algoritmi stessi e dell'uso di strutture dati succinte, sarebbe sicuramente molto difficile ma potrebbe portare a nuovi spunti di riflessione per il campo dell'informatica teorica, oltre che della bioinformatica.
Nonostante quanto detto, la qualità dei risultati è sufficiente per stabilire che una variante run-length encoded della PBWT, come analizzato negli ultimi anni per la 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 2.6:

- lo studio di pannelli multiallelici, ovvero costruiti su un alfabeto Σ arbitrario e non limitato ai simboli $\sigma = 0$ e $\sigma = 1$

- lo studio di pannelli con dati mancanti, ovvero pannelli costruiti 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 degli SMEM con un aplotipo esterno. Anche in questo caso, quindi, si potrebbe avere lo sviluppo di nuovi algoritmi che rispondano a task diversi, come, ad esempio, il calcolo degli SMEM interni al panello, il calcolo dei 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 panello.

DC Serve definire i blocchi?

RLPBWT multiallelica. Per quanto i pannelli di aplotipi, prodotti dal sequencing del genoma umano. raramente presentino siti multiallelici, si ha una presenza stimata, al momento, di circa il 2% di siti triallelici [44]. 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 (evidenziando che sia stimato circa un terzo dei reali siti triallelici) ma anche destinata a crescere in modo non lineare rispetto al numero di individui sequenziati [45]. Inoltre, molte specie, soprattutto vegetali, sono già riconosciute essere poliploidi. 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 multiallelica (mRLPBWT), si può pensare ad una soluzione molto simile a quanto visto per la RLBWT. 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 di intvector compressi, al fine di poter computare la funzione $w(i, \sigma)$ anche nel caso multiallelico. Si segnala che si attende un inversione di tendenza in termini di memoria, avendo che, in tal caso, l'uso di intvector compressi potrebbe rivelarsi meno efficiente dell'uso dei bitvector sparsi, anche con pochi sample
- un 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, fortunatamente, inferiore a 10 nella maggioranza dei casi di studio biologico.
Quindi, 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 corretti in una fase di preprocessing, si rischia di limitare la capacità di inferenza dai pannelli stessi.
Come anticipato alla sezione 2.6, si sono iniziate a sviluppare estensioni della PBWT che ammettano wildcard, ovvero simboli nel pannello che possono assumere qualsiasi valore dell'alfabeto Σ, su cui è costruito il pannello stesso.
Uno degli sviluppi futuri sarebbe quindi quello di generalizzare la RLPBWT, ma anche l'eventuale mRLPBWT, 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 non preprocessati.

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 [46] ha proposto un articolo in cui dimostra come la RLBWT, implementata in MONI [7], sia già predisposta al calcolo dei K-MEM, ovvero MEM, tra sottostringhe 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 dei K-SMEM, tra sottostringhe dell'aplotipo query e il pannello, che comportino SMEM 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 presentata in questa tesi.
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 sottostringhe di un aplotipo esterno all'interno di un pannello di aplotipi.

La tematica della pangenomica *è innovativa 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 pangenomics: 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] Paola Bonizzoni, Christina Boucher, Davide Cozzi, Travis Gagie, Sana Kashgouli, Dominik Köppl, and Massimiliano Rossi. Compressed data structures for population-scale positional Burrows–Wheeler transforms. *bioRxiv*, 09 2022.

[10] Guy Joseph Jacobson. *Succinct static data structures*. Carnegie Mellon University, 1988.

[11] Gonzalo Navarro. *Compact data structures: A practical approach*. Cambridge University Press, 2016.

[12] Guy Jacobson. Space-efficient static trees and graphs. In *30th annual symposium on foundations of computer science*, pages 549–554. IEEE Computer Society, 1989.

[13] 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.

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

[15] 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.

[16] 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.

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

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

[19] 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.

[20] Kunihiko Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *SODA*, volume 2, pages 225–232, 2002.

[21] 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.

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

[23] 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.

[24] Alberto Policriti and Nicola Prezza. LZ77 computation based on the run-length encoded BWT. *Algorithmica*, 80(7):1986–2011, 2018.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] Hideo Bannai, Travis Gagie, and I Tomohiro. Refining the r-index. *Theoretical Computer Science*, 812:96–108, 2020.

[30] Heng Li. Tabix: fast retrieval of sequence features from generic tab-delimited files. *Bioinformatics*, 27(5):718–719, 2011.

[31] 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.

[32] Ahsan Sanaullah, Degui Zhi, and Shaojie Zhang. d-PBWT: dynamic positional Burrows–Wheeler transform. *Bioinformatics*, 37(16):2390–2397, 02 2021.

[33] Ardalan Naseri, Degui Zhi, and Shaojie Zhang. Multi-allelic positional Burrows-Wheeler transform. *BMC bioinformatics*, 20(11):1–8, 2019.

[34] 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.

[35] Lucia Williams and Brendan Mumey. Maximal perfect haplotype blocks with wildcards. *Iscience*, 23(6):101149, 2020.

[36] Simone Rubinacci, Olivier Delaneau, and Jonathan Marchini. Genotype imputation using the positional Burrows Wheeler transform. *PLoS genetics*, 16(11):e1009049, 2020.

[37] Nathaniel K Brown, Travis Gagie, and Massimiliano Rossi. RLBWT tricks. *arXiv preprint arXiv:2112.04271*, 2021.

[38] 1000 Genomes Project Consortium et al. A global reference for human genetic variation. *Nature*, 526(7571):68, 2015.

[39] Leonardo Dagum and Ramesh Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[40] Petr Danecek, Adam Auton, Goncalo Abecasis, Cornelis A Albers, Eric Banks, Mark A DePristo, Robert E Handsaker, Gerton Lunter, Gabor T Marth, Stephen T Sherry, et al. The variant call format and VCFtools. *Bioinformatics*, 27(15):2156–2158, 2011.

[41] 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.

[42] 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.

[43] Gary K. Chen. MaCS. `https://github.com/gchen98/macs`, 2019.

[44] Alan Hodgkinson and Adam Eyre-Walker. Human triallelic sites: evidence for a new mutational mechanism? *Genetics*, 184(1):233–241, 2010.

[45] 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.

[46] Travis Gagie. MONI can find k-MEMs. *arXiv preprint arXiv:2202.05085*, 2022.