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: *Dott. 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!

Abstract

Negli ultimi anni, a partire dall'articolo di Durbin del 2014, la **Trasformata di Burrows-Wheeler Posizionale (*PBWT*)** è stata al centro delle ricerche riguardanti il disegno di algoritmi efficienti per il pattern matching su grandi collezioni di aplotipi. Come indicato da Durbin stesso, una **rappresentazione run-length encoded della PBWT** risulta essere molto efficiente dal punto di vista della memorizzazione della stessa.
In questa tesi, svolta in collaborazione con il laboratorio di ricerca **BIAS** del **Dipartimento di Informatica Sistemistica e Comunicazione *(DISCo)***, con professori e ricercatori dell'**University of Florida (*UFL*)** e della **Dalhousie University (Canada)**, si è quindi implementata una variante della **RLPBWT**, ispirata ai risultati già ottenuti con la **variante run-length encoded della BWT** tradizionale, che permettesse di risolvere il problema del matching tra un aplotipo esterno e un pannello di aplotipi.
A tal fine si sono selezionate le informazioni minimali da memorizzare per ogni run, utilizzando strutture dati succinte (come gli sparse bit-vectors) al fine di ottimizzare la complessità spaziale della struttura dati, e costruendo un efficiente algoritmo per effettuare query alla struttura stessa.

Indice

Capitolo 1

Introduzione

Capitolo 2

Preliminari

In questo capitolo verranno specificate tutti i concetti fondamentali atti a comprendere i metodi usati in questa tesi che le motivazioni della stessa. In primis, verranno introdotte le *motivazioni biologiche*, al fine di dare uno scopo pratico agli algoritmi e alle strutture dati introdotte, per procedere poi con un breve excursus dei fondamenti teorici presenti allo stato dell'arte.
Dal punto di vista tecnico verranno quindi introdotti i *bit vector* e gli *straight-line programs*, fondamentali sia per la *run-length encoded Burrows-Wheeler Transform*. Anche quest'ultima verrà introdotta, dopo la versione classica della trasformata, per la quale saranno necessari i concetti di *suffix array* e *longest common prefix array*. Si introdurrà fine la *Positional Burrows-Wheeler Transform*, avendo che la tesi tratta una sua versione *run-length encoded*.

2.1 Motivazioni Biologiche

2.2 Bitvector

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

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

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

In alternativa si potrebbe avere come formalismo:

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

Nel corso degli ultimi anni si sono sviluppate diverse varianti dei *bit vector*, finalizzate ad offrire diversi costi di complessità spaziale e diversi tempi computazionali per le principali funzioni offerte.
Il primo vantaggio di questa struttura dati, nelle varianti che si andranno poi a nominare, è quella di garantire *random access* in tempo costante pur sfruttando varie tecniche per la memorizzazione efficiente della stessa in memoria. Lo spazio necessario per l'implementazione, presente nella **Succinct Data Structure Library (*SDSL*)** [1], una delle principali librerie (scritta in C++11) per strutture dati succinte, delle principali varianti è visualizzabile in tabella 2.1. Il secondo vantaggio consiste nel fatto che i *bit vector* permettono l'implementazione efficiente di due funzioni:

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

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

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

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

2.2.1 Funzione rank

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

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

Come detto, da un punto di vista teorico, al costo di $\mathcal{O}(n)$ bit aggiuntivi in memoria tale funzione sarebbe supportata in tempo $\mathcal{O}(1)$. Questo però non risulta vero nelle principali implementazioni. La complessità temporale varia infatti a seconda dell'implementazione, anche in conseguenza al fatto che si ha una quantità diversa di bit aggiuntivi salvati in memoria. La tabella con le complessità temporali stimate della *funzione rank*, per le varianti di *bit vector* implementate in *SDSL*, è visualizzabile in tabella 2.2.

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

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

2.2.2 Funzione select

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

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

Anche in questo caso vale lo stesso discorso fatto per la *funzione rank* in merito alla complessità temporale e ai bit aggiuntivi. La tabella con le complessità temporali stimate della *funzione select*, per le varianti di *bit vector* implementate in *SDSL*, è visualizzabile in tabella 2.3.

Tabella 2.3: Complessità temporali stimate della *funzione select* per alcune varianti di *bit vector*, con la quantità di bit aggiuntivi richiesta. Si assume un bit vector di lunghezza n, con un numero di bit posti pari a 1 (o $\top$) pari a m.

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

Si può quindi vedere un semplice esempio esplicativo.

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

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

Si ha che, per esempio:

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

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

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

2.3 Straight-Line Program

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

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

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

dove:

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

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

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

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

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

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

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

Esempio 2. *Si prenda la seguente stringa:*

$$s = GATTAGATACAT\$GATTACATAGAT$$

Si potrebbe produrre il seguente SLP*:*

$$S \to ZWAY\$ZYAW$$

$$Z \to WX$$

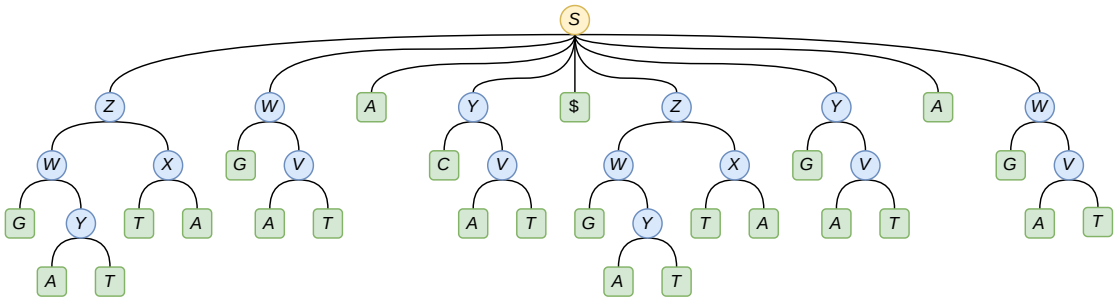
$$Y \to CV$$

$$X \to TA$$

$$W \to GV$$

$$V \to AT$$

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



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

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

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

2.3.1 Longest Common Extension

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

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

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

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

VERIFICARE BENE COSA SIA n.

2.4 Suffix Array

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

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

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

$$T[SA_T[i] : n-1] \prec T[SA_T[i'] : n-1]$$

Si vede quindi esempio chiarificatore.

Esempio 3. *Si prenda la stringa:*

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

Si producono quindi i seguenti suffissi e il loro riordinamento:

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

Ottenendo quindi che:

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

2.4.1 Longest common prefix

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

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

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

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

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

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

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

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

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

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

2.4.2 SA inverso

Ai fini di poter comprendere future definizioni si presenta brevemente anche l'**Inverse Suffix Array (*ISA*)**, ovvero la permutazione inversa dei valori del *suffix array*.

Definizione 6. *Dato il* ***suffix array*** SA_T*, costruito su un testo* T *di lunghezza* n*, si definisce l'****inverse suffix array*** ISA_T *come:*

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

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

Indice	$\mathbf{SA_T}$	$\mathbf{ISA_T}$	**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$

2.4.3 LCP permutato

Un'altra permutazione che bisogna introdurre è il **permuted longest-common-prefix array (*PLCP*)** [6]. Tale permutazione permette una rappresentazione succinta in memoria [7], permettendo di ottenere gli stessi risultati di quest'ultimo. Un'altro vantaggio è che la sua ricostruzione richiede un minor costo computazionale.

Definizione 7. *Si definisce il* ***permuted longest-common-prefix array*** $PLCP_T$*, construito a partire da un testo* T *di lunghezza* n*, come un array tale per cui [8]:*

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

ovvero dove i valori sono in ordine di posizione e non lessicografico. In altri termini [6]:

$$PLCP[SA[p]] = LCP[p], \forall p \in \{1, n-1\}$$

Quindi, essendo i valori in $PLCP$ *i medesimi in* LCP *ma solo con un ordine diverso si ha a che fare con una* permutazione*.*

Ciò che permette una rappresentazione compatta del $PLCP$ è descritto nel seguente lemma [9].

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

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

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

Indice	$\mathbf{SA_T}$	$\mathbf{ISA_T}$	$\mathbf{LCP_T}$	$\mathbf{PLCP_T}$	**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	0/-1	ssissippi$

In merito a $PLCP_T[11]$ *si ha che esso, a seconda delle definizioni, può essere sia* 0 *che* -1.

2.4.4 Funzioni phi e phi invertito

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 al fine di poter sia ricostruire efficientemente il $PLCP$ di un testo T (per i dettagli si rimanda all'articolo di Kärkkäinen [6]) che di permettere, come si vedrà brevemente più avanti nella sezione 2.5, il riconoscimento di tutte le occorrenze di un match in T [8].

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

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

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

Si noti che si assume il valore null quando, rispettivamente, si studia il primo e l'ultimo valore del suffix array.
Analogamente possono essere definite come [6]:

$$\varphi[SA[p]] = SA[p-1]$$

$$\varphi^{-1}[SA[p]] = SA[p+1]$$

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

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

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

CAPIRE SE SERVE DIRE ALTRO

2.5 Trasformata di Burrows-Wheeler

Introdotta nel 1994 da Burrows e Wheeler con lo scopo di comprimere testi, la **Burrows-Wheeler Transform** [10] è divenuta ormai uno standard nel campo dell'*algoritmica su stringhe* e della *bioinformatica*, grazie ai suoi molteplici vantaggi sia dal punto di vista della complessità temporale che da quello della complessità spaziale.
Nel dettaglio la BWT è una *trasformata reversibile* che permette una *compressione lossless*, quindi senza perdita d'informazione. Tale trasformazione vien costruita a partire dal riordinamento dei caratteri del testo in input, fattore che ha portato all'evidenza per cui caratteri uguali tendono ad essere posti consecutivamente all'interno della stringa prodotta dalla trasformata.

Definizione 9. *Dato un testo* T *\$-terminato, tale che* $|T| = n$*, si definisce la* ***Burrows-Wheeler Transform (BWT)*** *di* T*, denotata con* BWT_T*, come un array di caratteri lungo* n *dove l'elemento* i*-esimo è il carattere che precede l'*i*-esimo suffisso* T *nel riordinamento lessicografico. Più formalmente si ha che, con* $0 \leq i < n$*:*

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

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

Definizione 10. *Si definisce* ***rotazione* i-*esima***, *denotata con* $rot_T(i)$ *di un testo* T, *tale che* $|T| = n$, *come la stringa ottenuta dalla concatenazione del suffisso i-esimo con la restante porzione del testo. Più formalmente si ha che, avendo* $0 \leq i < n$:

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

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

Esempio 8. *Si prenda la stringa:*

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

Si produce la seguente matrice delle rotazioni riordinate:

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

Avendo quindi:

$$F_T = \$iiiimppssss \text{ e } BWT_T = ipssm\$pissii$$

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

Definizione 11. *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, \$, e ricostruire l'intero testo a ritroso. Si vede quindi un breve esempio.

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

$$BWT_T = ipssm\$pissii \text{ e } F_T = \$iiiimppssss$$

Si comincia dal simbolo \$ in BWT_T, che è l'ultimo carattere di T. Si inoltre ha che esso corrisponde al primo, e unico \$ in F_T, all'indice 0. Tale simbolo, per l'ovvia proprietà delle rotazioni è preceduto dal simbolo $BWT_T[0] = i$ in T. Quindi i precederà \$ in T. Si sa inoltre che tale i è il primo i in BWT_T. Si cerca quindi il primo i in F_T, sapendo che sono lo stesso simbolo nel testo. A questo punto il simbolo allo stesso indice di tale i nella BWT_T sarà il simbolo che precede i nel testo. Proseguendo a ritroso si ricostruisce l'intero testo:

$$T = mississippi\$$$

2.5.1 Trasformata di Burrows-Wheeler run-length encoded

Come già introdotto con la BWT caratteri uguali tendono ad essere messi in posizioni consecutive all'interno della trasformata stessa. Si è quindi pensato, fin da subito, ad un modo efficiente per memorizzare in modo compresso testi mediante l'uso del *run-length encoding* memorizzando le cosiddette **run** di caratteri consecutivi uguali mediante coppie:

$$(\text{carattere}, \text{lunghezza della run})$$

Esempio 10. *Vediamo un breve esempio.*
Si ipotizzi di avere la seguente stringa:

$$s = aaaacctgggggg$$

Una sua memorizzazione run-length sarebbe:

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

2.5.2 R-index

In questa direzione, nel 2005, Mäniken e Navarro proposero la **Run-Length encoded Burrows–Wheeler Transform (*RLBWT*)** [11]. Dato un testo T, tale che $|T| = n$, la **RLBWT** di T è la rappresentazione *run-length encoded* della BWT_T ed è denotata come $RLBWT_T$. Si ha che $|RLBWT_T| = r$, dove r è il numero di run.
Una strategia per la memorizzazione in modo compatto la $RLBWT$, di r run, è quella di memorizzare:

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

Esempio 11. *Si prenda ad esempio la seguente BWT_T:*

$$BWT_T = acggtcccaa$$

Si hanno:

$$c = acgtca$$

$$bv = 1110110010$$

Grazie al lavoro di Mäniken e Navarro si giunse al seguente teorema.

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

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

Nonostante questi ottimi risultati, per poter computare l'*FM-index*, si richiedeva anche la costruzione dei *suffix array samples* in spazio $\mathcal{O}(r)$.
Grazie al loro indice la struttura era in grado di, dato un testo T, tale che $|T| = n$, e dato un pattern P, tale che $|P| = m$:

- conteggiare le occorrenze (*count query*) del pattern nel testo, in tempo $\mathcal{O}(m \log n)$, con spazio $\mathcal{O}(r)$
- localizzare tali occorrenze (*locate query*) in tempo $\mathcal{O}(s)$, con spazio $\mathcal{O}\left(\frac{r}{s}\right)$, avendo s come distanza tra due *SA samples*

Si ha quindi che i *SA samples* di un ordine di grandezza maggiore, in termini di memoria, rispetto alla *RLBWT*.
A tal proposito, nel 2017, Policriti and Prezza [12] mostrarono come, dato un testo T, tale che $|T| = n$, e dato un pattern P, tale che $|P| = m$, trovare l'intervallo nella BWT_T contenente i caratteri in *occ* che precedono le occorrenze di P in T in spazio $\mathcal{O}(r)$ e in tempo:

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

Questo risultato è ora noto in letteratura come **Toehold Lemma** e dimostra come identificare **un** *SA sample* nell'intervallo contenente una il pattern P. Il limite è dato dal fatto che non si supporta la localizzazione di tutte le k occorrenze degli *SA samples* in quell'intervallo.
Nel 2020 Gagie et al [13] trovarono soluzione a questo problema, mediante la definizione della funzione φ (che nel dettaglio si dettaglierà più avanti) che ha permesso di avere le *locate query* in spazio $\mathcal{O}(r)$. Tale risultato si riassume nel seguente teorema.

Teorema 2. *Dato un testo T, tale che $|T| = n$, si può memorizzare T in spazio $\mathcal{O}(r)$ tale che si possano trovare tutte le k occorrenze di un pattern P, tale che $|P| = m$, in tempo:*

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

La struttura dati dietro questo risultato è stata denotata **R-index**, un'evoluzione dell'*FM-index*, consistente in:

- la *RLBWT*
- i *SA sample*, ovvero i valori di *SA* all'inizio e alla fine di ogni run. Si noti quindi che sono memorizzati in spazio proporzionale al numero di run

Per i dettagli in merito alla costruzione dell'indice si rimanda ai paper di Kuhnle et al. [14], di Mun et al. [15] e di Boucher et al. [16].

2.5.3 Match massimali con RLBWT

Dopo aver introdotto l'**R-index** bisogna brevemente spiegare come avvenga effettivamente la ricerca dei match di un pattern P, lungo m, in un testo T, lungo M. Il fine è quindi quello di calcolare i cosiddetti **Maximal exact matches (*MEM*)**, ovvero match esatti tra P e T che non possono essere estesi in alcuna direzione.

Definizione 12. *Dato un testo T, con $|T| = n$, e un pattern P, con $|P| = m$, si definisce una sottostringa $P[i : i + l - 1]$, di lunghezza l, **MEM** di P in T se:*

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

L'importanza nel calcolo dei match massimali esatti si ritrova nel loro uso nei metodi di allineamento basati sul **paradigma seed-and-extend**. Tale paradigma, sfruttato in algoritmi di allineamento come **BLAST** [17], uno degli allineatori più usati al mondo, si basa sul trovare *MEM* di piccola lunghezza, i *seed* appunto, per poi continuare l'allineamento tramite algoritmi più sofisticati, spesso basati sulla *programmazione dinamica*.
Nel 2020, Bannai et al. [18] mostrarono come il calcolo dei *MEM* fosse equivalente al calcolo delle **Matching Statistics (*MS*)**, un concetto teorico molto usato in *bioinformatica*. Informalmente, per ogni posizione i di un pattern P, le *MS* riportano la lunghezza della più lunga sottostringa comune, iniziante in i, tra P e un testo T, e l'indice di inizio di tale sottostringa in T.

Definizione 13. *Dato un testo* T*, con* $|T| = n$*, e un pattern* P*, con* $|P| = m$*, si definisce* ***matching statistics*** *di* R *su* T *un array* MS*, tale che* $|MS| = m$ *di coppie* (pos, len) *tale che:*

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

Una volta calcolato *MS* si ha il seguente lemma.

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

$$P[i : i + l - 1], \forall 1 < i \leq m$$

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

$$MS[i].len = l \wedge MS[i - 1].len \leq MS[i].len$$

CAPIRE SE METTERE DETTAGLI DI CALCOLO CON LCP.
Per costruire l'array *MS* l'approccio naive è quello di sfruttare interamente l'*LCP array* ma, sempre nell'articolo di Bannai et al.[18], si è presentato una semplice concetto in grado di ottimizzare il processo, quello delle **threshold**. Questa piccola struttura dati memorizza il minimo valore dell'"*LCP array* tra due run consecutive nel medesimo simbolo nella *BWT*.

Definizione 14. *Dato un testo T e date $BWT_T[j' : j]$ e $BWT_T[k : k']$ due run consecutive dello stesso carattere in BWT_T. Si definisce la* ***threshold*** *posizione:*

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

Rossi et al., nel 2021, sfruttarono tutte le conoscenze relative alla **RLBWT**, all'**R-index** e alle **matching statistics** per ideare **MONI:** ***A Pangenomics Index for Finding MEMs*** [19]. In questa soluzione si ha quindi la costruzione, in due *sweep*, tramite l'**algoritmo di Bannai**, dell'array delle *matching statistics*. Infatti si ha:

- un primo sweep che computa i vari $MS[i].pos$
- un secondo sweep che, tramite random access sul testo T, computa i vari $MS[i].len$ e, contemporaneamente, annota i match

APPROFONDIRE ULTIMA FRASE
CAPIRE SE METTERE ESEMPIO
Questa pubblicazione è stata uno dei punti di partenza per riadattare quanto studiato sulla BWT classica al fine di ottenere risultati analoghi per la $PBWT$.
Per ulteriori dettagli sull'implementazione, sul calcolo delle *threshold* e sui risultati si rimanda direttamente al paper di *MONI* [19].

2.5.4 PHONI

Nel 2021, Boucher, Gagie, Rossi et al. proposero un ulteriore miglioramento di quanto fatto in *MONI*, con **PHONI:** ***Streamed Matching Statistics with Multi-Genome References***.
In questo progetto non solo si sostituì l'uso delle *thresholds* con l'uso delle **LCE queries**, riducendosi ad un solo *sweep* sull'array MS (permettendo un uso "online" dell'algoritmo), ma si esplicitò anche l'uso delle *funzioni* φ e φ^{-1} e dell'$PLCP_T$ per il riconoscimento di tutte le occorrenze di ogni *MEM* tra un pattern e un testo. A tal fine si sfrutta infatti il seguente teorema [13].

Teorema 3. *Dato un testo T, tale che $|t| = n$, si può memorizzare T in $\mathcal{O}(r)$, con r numero di run, tale che, dato un indice $p \in \{0, n-1\}$ si possano computare $\varphi(p)$, $\varphi^{-1}(p)$ e $PLCP[p]$ in tempo:*

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

Si è quindi potuto migliorare e semplificare l'**algoritmo di Bannai** usato in *MONI* sfruttando un solo *sweep*. Infatti, sfruttando le *LCE query*, avendo il testo T in memoria sotto forma di *SLP*, è possibile computare contemporaneamente

sia i vari $MS[i].pos$ che i vari $MS[i].len$, avendo, come nel caso dell'*algoritmo di Bannai*, anche il computo dei match nel momento in cui si hanno a disposizione i valori $MS[i].len$, avendo che si ha un MEM sse $MS[i].len = l \wedge MS[i-1].len \leq MS[i].len$.
Per ulteriori approfondimenti si rimanda al paper di *PHONI* [8]. **CAPIRE SE SERVE APPROFONDIMENTO SU COMPLESSITÀ CHE PERÒ NON SONO BEN DEFINITI IN QUANTO DIPENDENTI DALLA STRUTTURA SOTTOSTANTE**

2.6 Trasformata di Burrows-Wheeler posizionale

Presentata nel 2014 da Richard Durbin la **Positional Burrows-Wheeler Transform (*PBWT*)** [20], traducibile con *trasformata di Burrows-Wheeler posizionale*, è una struttura efficiente per la memorizzazione e l'interrogazione di pannelli di aplotipi.
Formalmente si considera un pannello X di M aplotipi x_i, $i = 0, \ldots, M-1$, su N siti, indicizzati tramite $k = 0, \ldots, N-1$, tale che tutti i siti sono considerati biallelici. Da un punto di vista computazionale quest'ultima assunzione comporta che il pannello X è costruito sull'alfabeto $\Sigma = \{0, 1\}$, avendo quindi che:

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

Si consideri che l'alfabeto è *ordinato*, avendo che $0 \prec 1$.
Prima di proseguire con la trattazione è bene fornire la descrizione di alcuni formalismi utilizzati:

- si denota, per una qualsiasi sequenza s, con $s[k_1, k_2)$ la **sottostringa** di s che inizia alla colonna k_1 e termina alla colonna $k_2 - 1$
- date due sequenze t e s, si definisce un **match** tra le due sequenze sse $s[k_1, k_2) = t[k_1, k_2)$, avendo che tale match inizia alla colonna k_1 e termina alla colonna $k_2 - 1$
- un match tra due sequenze s e t, come definito al punto precedente, è definito **localmente massimale** sse non si ha alcuna estensione a destra o sinistra che comporti un ulteriore match, avendo quindi che:

$$(k_1 = 0 \vee s[k_1 - 1] \neq t[k_1 - 1]) \wedge (k_2 = N \vee s[k_2] \neq t[k_2])$$

- comparando una sequenza s ad un pannello di aplotipi X si definisce che s ha un **set-maximal match** con x_i, che inizia alla colonna k_1 e termina alla colonna $k_2 - 1$, sse tale match è *localmente massimale* e

non si ha alcun altro match di s con un altro x_j che include l'intervallo $[k_1, k_2)$

La costruzione di questa struttura dati si basa, ad ogni colonna k, sul riordinamento lessicografico delle sequenze di aplotipi basato sull'ordinamento inverso dei prefissi terminanti in colonna $k-1$. I valori presenti in colonna k dopo il riordinamento altro non sono che i valori che andranno a popolare la cosiddetta **matrice PBWT**, che rappresenta la vera e propria trasformata. Si noti che avere le sequenze ordinate in base ai prefissi invertiti alla k-esima colonna permette di identificare i match con maggior facilità in quanto, ad ogni colonna, aplotipi con suffisso comune (o prefisso comune in ordine inverso) saranno in posizioni consecutive all'interno della trasformata.
La computazione di tutti i riordinamenti non presenta difficoltà dal punto di vista computazionale in quanto, conoscendo l'ordinamento in colonna k, si può derivare facilmente l'ordinamento in colonna $k+1$, studiando solo i valori alla colonna precedente ed effettuando uno **step di radix sort**.
Più formalmente si denota con $a_k[i] = m$, con $m < M$, l'indice della sequenza x_m del pannello X da cui deriva il prefisso i-esimo nell'ordine inverso in colonna k. Si ottiene quindi che l'array a_k, detto **prefix array**, altro non è che una permutazione degli indici $0, \ldots, M-1$.

Definizione 15. *Dato un aplotipo i, appartenente al pannello X, e un indice di colonna k, si definisce il **prefix array** a_k come una permutazione degli indici $0, \ldots, M-1$ tale che $a_k[i] = j$ sse x_j è l'i-esimo aplotipo di X nell'ordinamento inverso dei prefissi ottenuto alla colonna k.*

Data questa definizione ne segue che la *matrice PBWT* si ottiene direttamente andando a vedere, per ogni colonna, gli indici del *prefix array* e prendendo i valori del pannello X secondo l'ordine espresso da quell'array.
Per comodità di rappresentazione definiamo formalmente i valori della *matrice PBTW* con il seguente formalismo:

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

avendo quindi che y_i^k denota la sequenza i-esima secondo l'ordinamento ottenuto per la colonna k. Possiamo quindi meglio spiegare perché risulti semplice computare i vari *prefix array*. Infatti, si ha quindi che l'ordinamento degli elementi per a_{k+1} è lo stesso degli elementi per a_k, al più di "guidare" internamente il riordinamento tramite i valori di $y_i^k[k]$, seguendo l'ordinamento dato dall'alfabeto. A breve, tramite un esempio, si chiarirà meglio quanto detto.
SPIEGARE MOLTO MEGLIO QUANTO DETTO
Come anticipato prefissi simili saranno consecutivi nei riordinamenti fino alla colonna k-esima risulta quindi utile tenere traccia della posizione iniziale dei match

tra prefissi vicini. Formalmente, dato $i > 0$, si definisce il $d_k[i]$ come il più piccolo j tale che $y_i^k[j,k) = y_{i-1}^k[j,k)$. Ne segue ovviamente che, se $y_i^k[k-1] \neq y_{i-1}^k[k-1]$, allora $d_k[i] = k$. Per definizione, inoltre, $d_k[i] = k$ se $i = 0$. L'array d_k è detto **divergence array**.

Definizione 16. *Si definisce **divergence array** l'array d_k tale che $d_k[i]$ è l'indice colonna iniziale del match massimale a sinistra terminante in k tra l'i-esimo aplotipo e il suo precedente nell'ordinamento ottenuto alla colonna k-esima.*

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

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

Si noti che al posto del **divergence array** si può usare anche una variante del **Longest Common Prefix (*LCP*) array**, denotato 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 $l_k[i] = k - d_k[i]$.
FORSE SERVE DEFINIZIONE FORMALE.
Fatte queste premesse possiamo quindi fornire una definizione formale di **PBWT**.

Definizione 17. *Dato $X = \{x_1, x_2, \ldots, x_M\}$ un insieme/pannello di M aplotipi con N siti, la **PBWT** di X è una collezione di $N + 1$ coppie di array (a_k, d_k), con $0 \leq k \leq N$, dove ogni a_k è detto **prefix array** e ogni d_k è detto **divergence array**.*

L'algoritmo per la costruzione di a_{k+1} e d_{k+1} a partire da a_k e d_k è disponibile all'algoritmo A.1.
Ai fini della trattazione dell'algoritmo di match con un'aplotipo esterno ricordiamo un'ulteriore definizione.

Definizione 18. *Definiamo α_k come l'inverso della permutazione data dal **prefix array** a_k, avendo che:*

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

Esempio 12. *Vediamo quindi un esempio chiarificatore.*
Si assuma il seguente 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

Volendo calcolare y^6 riordiniamo il pannello con l'ordine inverso alla quinta colonna e y^6 altro non è che la sesta colonna del pannello riordinato, a_6 la colonna degli indici e d_6 la colonna iniziale in cui terminano le sottolineature:

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

Ottenendo quindi:

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

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

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

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

Nel complesso, permutando con tutti i vari prefix array, si otterrebbe la seguente ***matrice PBTW****:*

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

2.6.1 Match con aplotipo esterno

Durbin, nel suo articolo, propone diversi algoritmi, ad esempio per il calcolo di match interni ad X più lunghi di una lunghezza minima L o per la ricerca di tutti i *set-maximal match* interni ad X in tempo lineare. Di interesse per questa tesi è però il cosiddetto *algoritmo 5*, quello che si propone di trovare tutti i *set-maximal match* tra il panello X e un aplotipo esterno z, assumendo che $|z| = M$.
L'idea dietro l'algoritmo è quella di usare tre indici: e_k, f_k e g_k. Nel dettaglio e_k tiene traccia dell'inizio del più lungo match, terminante in colonna k, tra z e un qualche y_i^k. L'intervallo $[f_k, g_k) \subseteq [0, \ldots, M)$ invece identifica il sotto-intervallo di

a_k contenente gli indici degli aplotipi appartenetenti a tale match. Formalmente si ha quindi che:

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

Si noti che $g_k = M$ sse y_{M-1}^k appartiene alle sequenze per cui si ha tale match più lungo.
Bisogna quindi capire come aggiornare e_k, f_k e g_k passando dalla colonna k alla colonna $k+1$. L'idea è quella per cui, avendo $f_{k+1} < g_{k+1}$ allora sicuramente ho ancora delle righe che presentano un match che parte da $e_k = e_{k+1}$ e termina in k che può essere esteso in $k+1$. In caso contrario, avendo $f_{k+1} = g_{k+1}$, non si hanno match estendibili e quindi si può concludere che quelli terminanti in colonna k erano match massimali, dovendo poi aggiornare e_{k+1} ottenedo i relativi f_{k+1} e g_{k+1}. Bisogna quindi capire come funzioni la variante dell'**LF-mapping**, guidato dal carattere corrente dell'aplotipo query, all'interno della **PBWT**, per ottenere f_{k+1} e g_{k+1} a partire da f_k e g_k (e di conseguenza e_{k+1}).
Per effettuare il mapping abbiamo bisogno di tre componenti:

1. l'array c tale per cui $c[k] = j$ sse la colonna k contiene j occorrenze di 0

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

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

Tali valori possono essere computati e memorizzati in fase di costruzione della **PBWT**, come visibile direttamente nell'algoritmo A.1 per quanto riguarda u e v, avendone già la computazione. Per quanto riguarda c si ha che potrebbe essere banalmente calcolato anch'esso in fase di costruzione della **PBWT**, tenendo ogni volta traccia del numero di 0 incontrati nella colonna k-esima.
Sfruttando i valori di questi 3 array possiamo quindi effettuare il mapping, definito per comodità da una funzione, rappresentabile in pseudocodice come nell'algoritmo A.2:

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

tale per cui:

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

Infatti, come confermato anche dall'algoritmo di costruzione stesso, si ha che:

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

Esempio 13. *Vediamo un piccolo esempio chiarificatore, riprendendo il precedente.*
Per praticità riporta 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 quindi, ad esempio, con $k = 5$ e $i = 2$, che:

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

Avendo:

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

Si ha che:

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

Ai fini dell'algoritmo serve però il "passaggio inverso" rispetto a quello indicato da questa equazione, ovvero passare dalla colonna k alla colonna $k + 1$. Quindi, pensando alla permutazione inversa del **prefix array**, si ha che:

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

Esempio 14. *Si riprendono i dati dell'esempio precedente e si vuole calcolare, sempre con $k = 5$ e $i = 2$:*

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

Come volevasi dimostrare.

L'ultima equazione ci suggerisce quindi che l'LF-mapping sopra definito consente il corretto aggiornamento di f_k e g_k. Definendo quindi:

$$f_{k+1} = w_k(f, z[k])$$

si ha che f_{k+1} sarà l'indice, in a_{k+1}, della prima sequenza y_j^k, con $j \geq f$, per la quale $y_j^k[k] = z[k]$. Analogamente si ha anche:

$$g_{k+1} = w_k(g, z[k])$$

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

1. si ha che $f_{k+1} < g_{k+1}$, quindi si hanno ancora match che partono da e_k e terminano in k che si estendono anche in $k+1$. In tal caso quindi $e_{k+1} = e_k$

2. si ha che $f_{k+1} = g_{k+1}$, quindi non si hanno match che partono da e_k e terminano in k che sono anche estendibili in $k+1$. Bisogna quindi annotare i match terminanti in k, nell'intervallo $[f_k, g_k)$ su a_k, e poi calcolare i nuovi e_k, f_k e g_k. La chiave per questo calcolo è che, virtualmente, l'aplotipo z si trova o subito prima del blocco di aplotipi $[f_k, g_k)$ in colonna k, secondo l'ordinamento dato dalla medesima colonna, o subito dopo. Si ha quindi che, essendo nell'ordinamento o subito prima di f_k o subito dopo g_k:

   $$y^{k+1}_{f_{k+1}-1} < z < y^{k+1}_{f_{k+1}}$$

   Diventa quindi possibile inferire che:

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

   Si considera quindi, come punto di partenza $e_{k+1} = d_{k+1}[f_{k+1}] - 1$, studiando di conseguenza $z[e_{k+1}]$, avendo due casi possibili:

   (a) se tale valore è 0 allora, per l'ordinamento, 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 il quale il **divergence array** non supera il valore di e_{k+1}. Si ottengono in tal modo le sequenze, nel riordinamento in $k+1$, che hanno un match da e_{k+1} a $k+1$. Invece g_{k+1} resta fisso

   (b) se tale valore è 1 allora, per l'ordinamento, 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. Invece f_{k+1} resta fisso

Si noti inoltre che, a livello di inizializzazione, si hanno:

$$f_0 = g_0 = e_0 = 0$$

Quindi il primo step sarà già un caso in cui $f_k = g_k$ qualora $x_0[0] = y^0_0 \neq z[0]$.

Esempio 15. *Mostrare un esempio completo di esecuzione richiederebbe troppo tempo quindi ci si limita a mostrare cosa succede nel caso in cui, ad un certo punto dell'esecuzione si hanno* $f_{k+1} = g_{k+1}$.
Si assuma il pannello e la matrice PBWT visti all'esempio 12 con una query z. *Nel complesso si identificherebbero i seguenti match:*

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

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

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

Avendo quindi che, a partire dalla colonne 0 *fino alla colonna* $6 - 1 = 5$ *si hanno le righe nel range* $[6, 10)$ *di* a_5 *che matchano con* $z[0, 5]$.
Bisogna quindi aggiornare f *e* g. *Si assuma che* $z[6] = 1$ *e che:*

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

Si calcolano quindi:

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

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

Avendo quindi $f_7 = g_7$ si procede, in primis, annotando i match terminanti in k_5. Seguendo l'algoritmo si ha quindi un primo aggiornamento di e_{k+1}, che viene inizializzato a, avendo in memoria d_7 con random access in tempo costante:

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

Questo viene fatto in quanto, come detto, l'aplotipo z si trova o subito prima del blocco di aplotipi $[f_k, g_k)$.
Essendo inoltre $z[e_7] = z[6] = 1$ si procede aggiornando g e tenendo fermo f, avendo che, essendo $z[6] = 1$, gli aplotipi che matchano seguiranno nel mapping tramite il simbolo $\sigma = 1$. Si procede quindi inizializzando il nuovo g:

$$g_7 = f_7 + 1 = 20$$

ricordando che g può "superare" le dimensioni del pannello essendo escluso in $[f_k, g_k)$.
A questo punto si segue la linea specificata da f_7 in a_7 a ritroso, partendo da $e_7 - 1$, fino a che si hanno match con z, aggiornando così il valore di e_7.
In questo caso non si hanno altre operazioni, in quanto $g_7 = M$ ma, qualora non lo fosse stato, si sarebbe incrementato g_7 fino a che il corrispondente $d_7[g_7]$ sarebbe stato minore o uguale di e_7, identificando tutte le nuove righe che hanno un match da e_7 a $k = 6$ con z.

SISTEMARE ESEMPIO E CAPIRE SE METTERE LE IMMAGINI
L'algoritmo 5 è visualizzabile all'algoritmo 2.1 e, secondo i calcoli di Durbin, ha complessità $\mathcal{O}(NM)$, in quanto si ritiene che il numero di accessi ai loop interni sia limitato dalla costante rappresentante il numero di match, c. Nonostante ciò tale complessità temporale è ancora in corso di studio in quanto si hanno in letteratura evidenze della sua non correttezza. Un esempio è il paper di Naseri [21], dove si afferma che l'intuizione per cui tale costante c limiti superiormente gli accessi ai loop innestati sia falsa. Si noti che nell'articolo non viene però precisata una nuova misura per la complessità dell'algoritmo. **VERIFICARE ULTIMA FRASE (MA ANCHE TUTTO IL RESTO CHE VA SCRITTO MEGLIO)**

Limiti spaziali

Bisogna affrontare la tematica della complessità in spazio di tale algoritmo. Ipotizzando di non ricalcolare colonna per colonna tutti i dati necessari (comportando

Algoritmo 2.1 Algoritmo 5 di Durbin.

```
function FIND_SET_MAXIMAL_MATCHES_FROM_Z(z)
    for k ← 0 to N do
        e, f, g ← Update_Z_Matches(k, z, e, f, g)
function UPDATE_Z_MATCHES(k, z, e, f, g)
    f' ← w(k, f, z[k])
    g' ← w(k, g, z[k])
    if f' < g' then                          ▷ se k è N − 1 match da e_k a N − 1
        e' ← e_k
    else                                                  ▷ match da e_k a k
        e' ← d_{k+1}[f'] − 1
        if z[e'] = 0 and f' > 0 then
            f' ← g' − 1
            while z[e' − 1] = y^{k+1}_{f'}[e' − 1] do e' ← e' − 1
            while d_{k+1}[f'] ≤ e' do f' ← f' − 1
        else
            g' ← f' + 1
            while z[e' − 1] = y^{k+1}_{f'}[e' − 1] do e' ← e' − 1
            while g' < M and d_{k+1}[g'] ≤ e' do g' ← g' + 1
    return e', f', g'
```

un'incremento dal punto di vista temporale).
Ricapitolando, per poter eseguire l'algoritmo 5, si necessita di avere in memoria, con *random access* in tempo costante:

- il **pannello** X, di dimensione NM
- il **prefix array** a_k, di dimensione NM
- il **divergence array** d_k, di dimensione NM
- i **vettori** u_k e v_k, complessivamente di dimensione $2NM$
- il **vettore** c_k, di dimensione N

Possiamo quindi dire che si ha una complessità in memoria pari a $\mathcal{O}(NM)$ e, nel dettaglio, Durbin stima si tratti di $13NM$ byte[1].
Per poter capire meglio la problematica prendiamo ad esempio un pannello di medie dimensioni, con $N = 30000$ e $M = 100000$. Ne segue che, secondo la stima di Durbin, si necessitano ~ 36.32 gigabytes di memoria. Inoltre, una stima sperimentale di tale richiesta di memoria può essere confermata con l'esecuzione dell'implementazione della **PBWT** di Durbin stesso. Infatti, monitorando con `time` il picco di memoria durante l'esecuzione si ha che esso corrisponde a ~ 40.76 gigabytes (comprensivi anche di tutto ciò che è "a contorno" all'algoritmo stesso). I dati quindi sembrano confermare le stime di Durbin e confermano l'alto uso di memoria richiesto dall'algoritmo 5. Questa è stata la motivazione principale per cui si è sviluppata, in questa tesi magistrale, una versione **run-length encoded** della struttura dati che permettesse di effettuare query con un aplotipo esterno.

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 dei prospetti futuri, è bene citarne i principali.

PBWT multiallelica

La prima variante che si introduce è la **PBWT multiallelica (*mPBWT*)**, proposta da Naseri et al. nel 2019 [22]. Questo lavoro estende la $PBWT$ di Durbin generalizzandola ad un alfabeto arbitrario.

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

Dal punto di vista delle motivazioni biologiche, questa soluzione risulta fondamentale, oltre che per lo studio di specie multialleliche (soprattutto nel mondo vegetale) in quanto gli studi riportano come, nell'uomo, la presenza di siti triallelici sia sotto stimata.
Da un punto di vista prettamente algoritmico si sono quindi estesi i concetti di c, u_k e v_k visti nella *PBWT* per ottenere un vero e proprio *FM-index* in grado di lavorare su alfabeto arbitrario Σ, con conseguente forte aumento dello spazio richiesto in memoria. Da un punto di vista della complessità temporale, invece, si ha che le complessità degli algoritmi devono tenere conto anche della grandezza dell'alfabeto stesso, avendo però che, essendo esso tendenzialmente di dimensioni ridotte, questo fatto non comporti, in media, particolari problematiche dal punto di vista dei tempi di calcolo. Le complessità temporali della *mPBWT* infatti sono incrementate di un fattore t, con $t = |\sigma|$, e se tale valore è assunto costante ad inizio computazione, avendo che difficilmente si ha $t >> 2$, la complessità temporale non subisce variazioni considerevoli.

PBWT con struttura LEAP

Sempre nel 2019 Naseri et al. proposero anche una variante della *PBWT* che permettesse il calcolo non solo dei match massimali, come per l'algoritmo 5 di Durbin, ma anche qualsiasi match di lunghezza maggiore uguale ad una lunghezza arbitraria L [23]. Tale algoritmo fu nominato **PBWT-query**. Inoltre, nello stesso articolo, proposero una struttura dati aggiuntiva, detta **LEAP (*Linked Equal/Alternating Position*)**, che, al costo della memorizzazione di otto array aggiuntivi che permettessero di effettuare dei salti nella *matrice PBWT* (salvando gli indici del precedente/prossimo valore nella colonna uguale/diverso)e di memorizzare gli indici dei valori nel *divergence array* relativi a tali indici, che ottimizzava i tempi dell'algoritmo per la *PBWT-query* ottenendo l'algoritmo detto (***L-PBWT-query***).
CAPIRE SE SPIEGARE MEGLIO, APPROFONDIRE E FORMALIZZARE GLI 8 ARRAY
Da un punto di vista computazionale si noti che la complessità dell'algoritmo per la *PBWT query*, con match di lunghezza minima L è:

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

Avendo:

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

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

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

VERIFICARE SIANO BYTE

PBWT dinamica

Sanaullah et al., nel 2021, proposero la **Dynamic PBWT (*d-PBWT*)** [21], col fine di superare le limitazioni imposte dalle strutture statiche usate nella *PBWT* di Durbin. Si è quindi pensato di sostituire, per i vari elementi della *PBWT*, l'uso degli array, statici, con l'uso di *linked list*, dinamiche.
Grazie a alle *linked list* si è quindi reso possibile l'aggiornamento efficiente della *matrice PBWT* all'aggiunta di un nuovo aplotipo nel pannello o alla rimozione di uno.
Da un punto di vista computazionale è interessante notare come le implementazioni degli algoritmi di Durbin presentino la medesima complessità computazionale, a partire dalla creazione della *d-PBWT* in $\mathcal{O}(NM)$ e che l'aggiunta e la rimozione di un aplotipo siano entrambe in tempo:

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

PBWT con wildcard

La tematica dei dati mancanti è una tematica aperta in *bioinformatica*. I sequenziatori infatti presentano un range d'errore dal 1% al 15%, si ha a volte un basso *coverage* (ovvero il numero di read che contengono la base sequenziata per un certo locus del genoma) e la fase di assemblaggio del genoma può comportare errori, Questo, in fase di produzione dei pannelli, implica che in alcuni casi non si sappia quale sia l'allele corretto per un individuo riferendosi ad un sito.
Williams e Mumey, nel 2020, proposero quindi l'uso della **PBWT con wildcard** al fine di disegnare un algoritmo in grado di trovare i match interni ad un pannello biallelico con dati mancanti, rappresentati come *wildcard* mediante il simbolo "*" (avendo quindi $\Sigma = \{0, 1, *\}$) [24].
In termini computazionali gli autori sono riusciti a formulare un algoritmo in grado tutti i match interni (ovvero i *blocchi*) massimali al pannello in tempo, con T numero di blocchi:

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

CAPIRE SE PARLARE DI IMPUTE5

IMPUTE5

Per citare un uso della *PBWT* si può parlare di **genotype imputation**, ovvero il processo con il quale si predicono genotipi non ancora osservati in un campione dio 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 di esaminare multipli genomi alla ricerca di associazioni tra varianti genetiche e malattie (o outcome specifici delle stesse), identificando varianti genomiche che sono statisticamente associati al rischio per una malattia.
A tal fine, nel 2020, Rubinacci et al. proposero **IMPUTE5** [25], un metodo basato sulla *PBWT* per la *genotype imputation* in grado di studiare pannelli di grandi dimensioni.
CAPIRE SE DIRE ALTRO

2.6.3 Una prima proposta run-length encoded

A fine 2021, Gagie et al. [26] inizio a teorizzare una variante **run-length encoded** della **PBWT**, basandosi sui risultati ottenuti sulla *BWT* classica.
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, $\forall 1 \leq k < N$. Ipotizziamo ora di voler studiare la riga i-esima del pannello originale. Si ha che, al variare della colonna k sulla *matrice PBWT*, la posizione della riga i è ricostruibile applicando le varie permutazioni (che, nei termini già presentati nella sezione, sarebbe uguale ad effettuare il *mapping* dalla colonna K alla colonna $k+1$):

$$i, \pi_1(i), \pi_2(\pi_1(i)), \ldots, \pi_{N-1}(\cdots(\pi_2(\pi_1(i)))\cdots)$$

Il punto fondamentale 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 quindi una struttura dati formata da N "tabelle" dove, la j-esima riga della k tabella contiene:

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

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

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

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

1. si cerca della prima "tabella" la riga relativa alla run, con indice di testa p, contenente l'indice i, avendo che la prima "tabella", relativa alla colonna $k = 0$ non presenta permutazioni e quindi l'indice i del pannello è anche l'indice i della *matrice PBWT*
2. si calcola poi la permutazione per l'indice i (alla prima operazione si avrà $k = 1$):
$$\pi_k(i) = \pi_k(p) + i - p$$
3. si cerca poi la riga relativa alla run contenente il simbolo $\pi_k(p)$ nella "tabella" successiva e si scansionano le righe di tale tabella a partire da quella appena identificata fino a trovare la run che contiene $\pi_k(i)$ e vedere il simbolo relativo a tale run (alla prima operazione si avrà $k = 1$)
4. si ripete la procedura dal punto 2) per ogni colonna k

Vediamo un esempio, estratto dal paper.

Esempio 16. *Si assuma la seguente matrice PBWT:*

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

Supponendo di voler ricostruire la riga $i = 9$, segnalata in rosso nella matrice PBWT, si costruiscono le seguenti tabelle [26]:

	table 1			table 2			table 3			table 4			table 5			table 6		
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 7			table 8			table 9			table 10			table 11			table 12
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				

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

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

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

$$x_9 = 100001000011$$

Nel paper non si trattano metodi per effettuare query a tale struttura dati, indicando solo che dovrebbe essere possibile farlo.

Capitolo 3

Metodo

In questo capitolo verranno illustrate le metodologie usate in questa tesi, trattando, sia dal punto di vista teorico che sperimentale, tutte le soluzioni che hanno portato alla costruzione della **RLPBWT**.
Nel dettaglio, si approfondiranno tutte le varianti della **RLPBWT** ottenute durante lo studio, evidenziandone pro e contro

Prima di proseguire con la spiegazione dettagliata delle varianti della **RLPBWT** è bene dare una prima motivazione al perché si sia ritenuto utile sviluppare una variante **run-length encoded** della **PBWT**.
Citando direttamente il paper di Durbin del 2014 [20], in cui si introduce la struttura:

> *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.*

Dove, con la dicitura *y arrays*, si indicano le colonne già permutate della **matrice PBWT**.
Quindi il risultato atteso è quello per cui aplotipi simili, che ad ogni step saranno consecutivi nel riordinamento, è molto probabile presentino lo stesso allele nella colonna di cui si sta in quel momento calcolando la permutazione. Ne segue che, all'interno della **matrice PBWT**, è molto probabile che si abbiano, consecutivamente, lunghe run di 0 e di 1.
Si noti quindi che si ottiene quindi il medesimo risultato atteso che si ha con la **BWT**, avendo che caratteri uguali è probabile che vengano posti in modo consecutivo all'interno della BWT stessa. Si hanno quindi le stesse premesse che hanno

portato alla **RLBWT**, considerando inoltre che, come in quel caso, non si tratta solo di memorizzare la struttura con compressione run-length ma di lavorare direttamente con la struttura dati compressa, risolvendo il problema del pattern matching senza decomprimere la struttura dati.

3.1 Introduzione alle varianti della RLPBWT

Lo sviluppo di questo progetto di tesi è stato tale per cui si sono sviluppate varie implementazioni della **RLPBWT**. Tali varianti non sono da intendersi ugualmente valide ma corrispondono al percorso evolutivo che c'è stato nell'ultimo anno di studio e ricerca in merito. Riassumendo il tutto si vedranno:

- una prima implementazione *naive*, detta appunto **RLPBWT naive**, che corrisponde al primo tentativo di studio. Questa soluzione non permette di sapere quali righe del pannello stanno matchando ma solo quali
- si è quindi iniziato ad introdurre l'uso dei *bitvectors*, con la **RLPBWT con bitvectors**, il cui funzionamento è pressoché analogo alla versione *naive* al più dell'uso di tali strutture succinte per il funzionamento del mapping. Questa soluzione non permette di sapere quali righe del pannello stanno matchando ma solo quali
- il primo sostanziale "cambio di paradigma", si ha avuto con la **RLPBWT con pannello denso**, variante in cui, oltre all'uso dei *bitvectors* si è proceduto al calcolo dei match tramite *matching statistics* e *LCE query*. Questa soluzione permette di sapere l'indice di una sola riga per la quale si sta avendo un match con il pattern
- migliorando la soluzione precedente con l'uso dell'*SLP* per la memorizzazione del pannello si è ottenuta la **RLPBWT con SLP**. Questa soluzione permette di sapere l'indice di una sola riga per la quale si sta avendo un match con il pattern
- con l'implementazione della **funzione** φ per la **RLPBWT** si è permesso di estendere i risultati delle ultime due varianti in modo da ottenere tutti i match con tutti gli indici delle righe per cui si hanno tali match con il pattern

Si può quindi iniziare ad apprezzare il percorso evolutivo e incrementale vissuto con questo progetto.

3.2 RLPBWT naive

Un primo approccio alla **compressione run-length** è stato quello di semplicemente "adattare" quanto presentato da Durbin. Soprattutto a causa di questo fattore tale approccio è stato nominato **RLPBWT naive**.
L'idea è stata quella di capire quali informazioni fossero necessarie al fine di poter calcolare i match. Si è quindi partiti studiando quanto memorizzato da Durbin stesso, pensando ad eventuali alternative.
Il dato fondamentale che la *PBWT* tiene in memoria è *il pannello X, con random access*. Ovviamente memorizzare l'intero pannello non era possibile. D'altro canto l'idea dietro la **RLPBWT** è quella di memorizzare con *compressione run-length* la *matrice PBWT*. La soluzione iniziale è stata quindi quella di memorizzare gli indici delle *teste di run*, ovvero gli indici iniziali di ogni run. Ovviamente questa informazione non è sufficiente per poter sapere se una run sia composta da simboli $\sigma = 0$ o simboli $\sigma = 1$. Fortunatamente, essendo lo studio limitato, come per la *PBWT*, a pannelli costruiti su alfabeto binario, $\Sigma = \{0, 1\}$, si è potuto sfruttare il fatto che le run si alternano tra un carattere e l'altro. Basta quindi tenere in memoria anche un valore booleano, nominato $start^k$, che permetta di capire se, in colonna k, la prima run sia una run di simboli $\sigma = 0$. Infatti le run di indice pari presentano lo stesso simbolo della prima run e quindi, dato un qualsiasi indice di run, è possibile sapere quale sia il simbolo di tale run.
Si memorizzano gli indici delle teste di run in un array p_k, di lunghezza parti al numero di run in colonna k. Si riconoscono alcune delle informazioni viste per la "bozza" di *RLPBWT* vista alla sottosezione 2.6.3.
Il passaggio successivo è stato quello di capire se le informazioni necessarie al mapping fossero tutte necessarie. In altri termini se, data la colonna k nella *matrice PBWT*, fossero necessari $c[k]$, u_k e v_k. In merito al valore $c[k]$, per quanto calcolabile in tempo $\mathcal{O}(r)$, dove r è il numero di run della colonna k-esima, si è deciso che si potesse calcolarlo in fase di costruzione delle *RLPBWT* e memorizzarlo esattamente come per la *PBWT*. In merito invece ai vettori u_k e v_k si è cercato un modo per ottenerne una rappresentazione che implicasse avere un solo valore per ogni run della colonna. In altri termini si è cercato di capire se fosse possibile tenere in memoria r valori che permettessero di effettuare comunque il mapping, a partire da un indice arbitrario $i \in \{0, \ldots, N-1\}$. Anche in questo caso l'alternanza data dal caso binario ha permesso di trovare una semplice soluzione. I valori di u_k e v_k crescono infatti in modo alternato. A seconda del simbolo σ rappresentato in una data run infatti si avrà che solo i valori dell'array relativo a tale simbolo, nel range di indici di quella run, verranno incrementati ad ogni passo di una unità. Per fare un semplice esempio, se siamo in una run di 0 e iteriamo virtualmente all'interno di tale run, solo i valori di u_k, in quel range di indici, cresceranno di volta in volta di uno mentre per v_k, nello stesso range, si avrà sempre lo stesso valore.

Esempio 17. *Si vede un esempio per chiarire meglio quanto espresso in merito a* u_k *e* v_k*.*
Sia 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

Grazie a questa alternanza è quindi possibile memorizzare, per ogni indice di testa di run i, tale che $i \neq 0$, solo il valore di $u_k[i]$ o $v_k[i]$, rispettivamente se sia una run su simboli $\sigma = 1$ o $\sigma = 0$. Questo in quanto, se si analizza una run di zeri si avrà che solo i valori di v_k, nel range della run, verranno incrementati ad ogni step. Per $i = 0$ banalmente si ha che $u_k[i] = v_k[i] = 0$.
Memorizzando i valori di u_k e v_k in un array uv_k, tale che $|uv_k| = |r|$, dove r è il numero di run alla colonna k-esima, 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:
  - 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 $v_k[p[i]] = uv_k[i]$

Lo pseudocodice relativo a quanto appena detto è consultabile all'algoritmo A.6.
In questa prima soluzione, infine, si è deciso di non mantenere in memoria il *prefix array* e di mantenere completamente il *divergence array*, sotto forma di *LCP array*,

per poter, da un punto di vista informale, reimplementare l'algoritmo 5 di Durbin solo con meno informazioni in memoria. Il non memorizzare il *prefix array*, d'altro canto, impedisce di identificare con precisione le righe del pannello per cui si ha un match quindi l'algoritmo, che verrà presentato più avanti nella tesi, è limitato al poter sapere quante righe matchano e non quali.
INSERIRE ALGORITMO DI COSTRUZIONE
In conclusione si riporta quindi un esempio dei dati memorizzati, per una data colonna, nella *RLPBWT naive*.

Esempio 18. *Sia data la seguente colonna:*

$$y^5 = 00101111000000000000$$

Per la RLPBWT naive si hanno in memoria:

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

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

$$c[5] = 15$$

$$l_5 = [0, 5, 4, 1, 3, 5, 5, 5, 5, 5, 5, 0, 5, 5, 5, 2, 5, 1, 5, 5]$$

QUI MANCA SPEIGAZIONE OFFSET
Si ha quindi già una forte riduzione dello spazio in memoria occupato dalla struttura, questo nonostante la memorizzazione completa dell*LCP array*. Riprendendo quindi l'esempio già visto per la *PBWT*, dato un panello di medie dimensioni, con $N = 30000$ e$M = 100000$, si ha che l'uso della *RLPBWT naive* richiede ~ 8.17 gigabytes di memoria (rispetto ai ~ 40.76 gigabytes della *PBWT*).
La spiegazione dell'algoritmo di match è rimandata a dopo l'introduzione della seconda variante, ovvero della ***RLPBWT con bitvector****, in quanto le due versioni condividono, ad un alto livello di astrazione, il medesimo procedimento per il calcolo dei match.*

3.3 RLPBWT con bitvectors

Al fine di compiere un ulteriore passo verso la formulazione di una struttura dati efficiente dal punto di vista dello spazio in memoria per la **RLPBWT**, si è provveduto a modificare la versione *naive* al fine di introdurre l'uso dei **bitvectors**. Questo è stato fatto al fine di ottenere una rappresentazione in memoria della stessa che fosse ancora più efficiente. Come si vedrà questa versione intermedia non comporterà un miglioramento effettivo del consumo di memoria ma permetterà di avere la base su cui costruire le versioni successive.

L'idea è quindi quella di sostituire, data una colonna k, quanto necessario a rappresentare le run (ovvero il vettore p_k della variante naive) e quanto necessario a permettere il mapping (ovvero il vettore uv_k).
In primis, per poter localizzare le run nella k-esima colonna, si è scelto di usare un *bitvector*, che denominiamo per praticità h_k, tale che $|h_k| = N$. Formalmente si ha che:

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

Informalmente, quindi, si ha che si ha 1 in h_k in tutti gli indici corrispondenti alla fine di una run.
Empiricamente ci si aspettano "poche" run all'interno di una colonna della *matrice PBWT*, per quanto già discusso nella sezione 2.6. Avendo poche run ci si aspetta anche "pochi" 1 all'interno di h_k, di conseguenza si è optato per usare gli **sparse bitvector** per la memorizzazione in memoria di ogni h_k, ricordando che, secondo quanto riportato per la libreria *SDSL* [1], tale variante richiede in memoria, indicando con r il numero di run:

$$\approx r \left(2 + \log \frac{|h_k|}{r}\right) \text{ bit}$$

VERIFICARE CHE SIANO BITS
Più elaborata è la rappresentazione dei vettori u_k e v_k. In questo caso si è deciso, a differenza della rappresentazione unica vista nella *RLPBWT naive*, di optare per due *sparse bitvector*. In particolare, per il vettore u_k, tale che $|u_k| = c[k]$, si ha che:

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

Analogamente si definisce v_k, tale che $|v_k| = N - c[k]$ come:

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

Si noti che:

$$rank_{h_k}(|h_k - 1|) + 1 = ([rank_{u_k}(|u_k - 1|) + 1) + [rank_{v_k}(|v_k - 1|) + 1$$

Ovvero il numero di 1 presenti in h_k è pari alla somma di quelli presenti in u_k e v_k. Ne segue che, anche per questi ultimi due vettori, la scelta di usare *sparse bitvector*

per la loro memorizzazione sia giustificata dalla poca quantità, empiricamente, di simboli $\sigma = 1$.
Si vede un esempio chiarificatore.

Esempio 19. *Sia data la seguente colonna:*

$$y^5 = 00101111000000000000$$

Si ha quindi che:

$$h_5 = 01110001000000000001$$

Avendo appunto un numero di run pari a:

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

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

$$u_5 = 011000000000001$$

Avendo infatti che si segnalano:

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

Parlando invece di v_5 si ha:

$$v_5 = 10001$$

Avendo che:

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

Le restanti informazioni, ovvero, per la colonna k, il valore $c[k]$, il booleano $start^k$ e l'LCP array l_k sono le medesime della variante *naive* della *RLPBWT* (motivo per quale solo a breve si tratterà l'algoritmo di match).
Lo pseudocodice relativo alla costruzione della colonna k-esima della **RLPBWT con bitvector** è disponile all'algoritmo A.4 (dove sono presenti anche le istruzioni per le varianti che verranno trattate in seguito).
CAPIRE SE METTERE UNO PSEUDO A PARTE
Bisogna spiegare come, data un indice di aplotipo $i \in \{0, \ldots, N-1\}$ e una colonna k, estrarre $u'_k[i]$ e $v'_k[i]$, ovvero come se si stesse usando la $PBWT$ classica, a partire dagli attuali $u_k[i]$ e $v_k[i]$. Ovviamente, se $i = 0$, si ha che $u'_k[0] = v'_k[0] = 0$. In

caso contrario bisogna capire la run in cui si trova l'indice i. Questo si ottiene direttamente sfruttando h_k:

$$run = rank_{h_k}(i)$$

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

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

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

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

$$beg = select_{h_k}(1) + 1$$

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

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

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

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

e l'offset rispetto all'indice i dato:

$$offset = i - beg$$

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

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

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

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

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

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

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

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

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

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

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

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

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

E potendo quindi restituire:

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

SPIEGAZIONE DA MIGLIORARE

Esempio 20. *Si prendano i dati e i risultati ottenuti all'esempio 19. Si vogliono calcolare u e v per $i = 6$.*
In primis si ha quindi:

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

:

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

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

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

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

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

che non andranno invertiti avendo $start^5 = \top$.
Si calcolano quindi:

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

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

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

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

L'algoritmo per il calcolo di u e v, tenendo in considerazione che tale metodo verrà usato anche nelle varianti che verranno presentate in seguito della *RLPBWT*, è disponibile all'algoritmo A.7.

3.4 Algoritmo per match massimali

Avendo presentato le prime due varianti della **RLPBWT**, concettualmente simili tra loro e diverse dalle successive due varianti, è possibile discutere dell'algoritmo di match con un aplotipo esterno, che riprende esattamente quanto discusso nell'algoritmo 5 di Durbin.
Il metodo procede quindi con l'aggiornamento dei tre indici e_k, f_k e g_k, avendo che gli ultimi due possono assumere qualsiasi valore in $\{0, M\}$, come con la *PBWT* classica. Avendo memorizzato solo informazioni relative alle *run* bisogna quindi, ogni volta, ricondurre l'indice alla run corretta:

- nella *RLPBWT naive* si risale all'indice di run a cui appartiene un certo indice, in colonna k, in $\{0, M\}$ scorrendo l'array p_k
- nella *RLPBWT con bitvector* la medesima operazione viene risolta usando la funzione $rank_{h_k}$

Inoltre Durbin sfruttava il *random access* al pannello, avendo in memoria sia il pannello che il *prefix array*, al fine di aggiornare il valore di e_k. In entrambe le versioni già presentate della *RLPBWT* non si ha in memoria né il *prefix array* né il pannello ma solo solo la rappresentazione compatta della *matrice PBWT*. Si è quindi dovuto pensare ad un metodo che ricomponga data una riga x_j del pannello X a partire da un elemento, indicizzato con i alla colonna $k+1$, della *matrice PBWT*, muovendosi da destra a sinistra e seguendo in modo inverso il *mapping*.
Per ottenere l'indice alla colonna k-esima da cui "proviene" l'indice i in colonna $k+1$ si inizia analizzando il valore $c[k]$. Infatti, se $i < c[k]$, allora sicuramente, in colonna k, è un indice corrispondente a $\sigma = 0$ quello dal quale proviene, ricordando come la costruzione della colonna $k+1$ nella *matrice PBWT* si abbia grazie ad un *passo di radix sort* con ordinamento stabile. Si sfruttano così ho l'array p_k o le funzioni $rank_{h_k}$ e $select_{h_k}$ per risalire all'indice in colonna k, calcolando prima l'indice di run e l'eventuale offset, per il quale il mapping porta all'indice i' in colonna $k+1$, seguendo "virtualmente" la riga x_j del pannello originale.
Per quanto riguarda la *RLPBWT naive* si ha lo pseudo codice per il mapping inverso consultabile all'algoritmo A.11 mentre per quanto riguarda la *RLPBWT con bitvector* si ha l'algoritmo A.12.
CAPIRE QUANTO APPROFONDIRE I DUE ALGORITMI.
Si procede quindi riadattando l'algoritmo di Durbin all'uso delle *run*, ottenendo, ad ogni step, i medesimi valori per e_k, f_k e g_k. Le uniche differenze sono:

- il calcolo del mapping necessità dell'estrazione dei valori u e v, tenendo conto esplicito degli offset nel caso della *RLPBWT naive*
- non si ha *random access* al pannello quindi bisogna procedere ogni volta con il'inverso del mapping e il calcolo del simbolo a partire dall'indice della run
- non si ha il *prefix array* in memoria quindi non è possibile sapere quali siano le righe che stanno matchando fino alla colonna k ma solo quante, sapendo che sono $g_k - f_k$

Anche in questo caso i due algoritmi sono consultabili, rispettivamente, all'algoritmo A.13 e all'algoritmo A.14.
APPROFONDIRE SPIEGAZIONE ALGORITMI

3.5 RLPBWT con matching statistics

Le precedenti versioni della *RLPBWT*, come anticipato, hanno permesso di poter ideare una variante *run-length encoded* della *PBWT*. In realtà anche in questo caso

c'è stata una fase transitoria di sviluppo, avendo due varianti della struttura, che verranno introdotte a breve.
Il fine era quello di ottenere quanto visto per la **RLBWT** anche per la variante posizionale, ovvero i concetti di:

- *matching statistics*
- *threshold*
- *LCE query*

A tal fine, come per la *RLBWT*, si necessita di *random access* al pannello. A causa di questo si sono avute due varianti in fase di sviluppo:

- una prima, ancora in ottica di "studio introduttivo", dove il pannello viene memorizzato come *vettore di bitvector classici*
- una seconda, definitiva, dove il pannello è memorizzato come *SLP*, nelle modalità introdotte nella sottosezione 4.1.2

Queste versioni, a loro volta, hanno permesso, in primis, l'ideazione di un algoritmo che sfruttasse l'idea delle threshold, come visto per la *BWT* classica con *MONI* [19], e poi, per quella basata sull'*SLP*, di uno basato sulle *LCE query*, come per *PHONI* [8]. Quest'ultima, con l'aggiunta della *struttura per la funzione* φ, sarà l'implementazione definitiva, per questa tesi, della *RLPBWT*.
Avendo in memoria il pannello si può quindi fare a meno dell'array *LCP* della colonna k-esima, avendo quindi che, per ogni colonna k, si ha in memoria:

- un booleano $start^k$ per specificare se la colonna presenta la prima run costruita su simboli $\sigma = 0$
- un bitvector sparso h_k per indicare l'inizio delle run
- il valore $c[k]$ per sapere quanti simboli $\sigma = 0$ si hanno nella colonna
- i valori u_k e v_k per il mapping
- i cosiddetti **prefix array samples**, ovvero i valori del **prefix array** di inizio e fine di ogni run. Si noti quindi che, anche in questo caso, si ha un'informazione in memoria proporzionale al numero di run r

In pratica si hanno in memoria le stesse informazioni della *RLPBWT con bitvector* al più di l_k, ovvero l'*array LCP* della colonna k-esima, e dei *prefix array samples*.

3.5.1 Matching statistics per la RLPBWT

La definizione formale per il concetto di **matching statistics**, nonché il calcolo dell'array stesso, vista per la *RLBWT* deve essere ovviamente riadattata allo studio di match tra un aplotipo e un pannello di aplotipi.

Definizione 19. *Dato un pannello X, di dimensioni $M \times N$, con M individui e N siti, e un aplotipo esterno/pattern z, tale che $|z| = N$, si definisce matching statistics di z su X un array MS di coppie (row, len), di lunghezza N, tale che (avendo che x_i indica l'i-esima riga del pannello X):*

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

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

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

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

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

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

Si vedrà in sezione 3.6 come calcolare, a partire da tali MEM*, tutte le righe del panello per le quali si ha lo stesso* MEM.
Il calcolo dell'array MS di z rispetto al pannello X si basa su due fasi:

1. la fase di **extension**
2. la fase di **bootstrap**

Si assuma di avere due indici i e j, $0 \leq i \leq j \leq N$, tali per cui $z[i, j]$ è un suffisso di uno tra $x_1[1, j]$, ..., $x_M[1, j]$.
La **fase di extension** estende il match di $z[i, j]$ a $z[i, j + 1]$ sse:

- $j < M$
- $z[i, j + 1]$ è un suffisso di uno tra $x_1[1, j + 1]$, ..., $x_M[1, j + 1]$

D'altro canto la **fase di bootstrap** cerca il più piccolo indice i', avendo $i \leq i' \leq j$, tale per cui $z[i', j]$ è un suffisso di uno tra $x_1[1, j+1]$, ..., $x_M[1, j+1]$.
Si ha quindi il computo di ogni valore $MS[i]$, $\forall i \in [0, N)$, dell'array delle *matching statistics*:

- si assume inizialmente che $MS[0].len = 0$
- si applica la *fase di bootstrap* per cercare il minimo indice i', avendo $i \leq i'$, tale che $z[i', i' + MS[i].len]$ è un suffisso di uno tra $x_1[1, i' + MS[i].len]$, ..., $x_M[1, i' + MS[i].len]$. Inoltre, per minimalità di i' si ha che, $\forall i < j < i'$, $MS[j].len = MS[j-1].len + 1$
- a questo punto si itera la *fase di estensione* per trovare il più lungo prefisso $z[i', k]$ che è anche un suffisso di uno tra $x_1[1, k]$, ..., $x_M[1, k]$, avendo che $MS[i'].len = k - i' + 1$
- avendo che $i' > i$ si può procedere induttivamente al calcolo dell'array MS

PARTE PRESA DAL PAPER: RIVEDERE PROFONDAMENTE.
In altri termini, più "pratici", il calcolo dell'array MS avviene nel seguente modo:

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

Si noti che non si è parlato di come calcolare i vari $MS[i].len$, questo in quanto si hanno due soluzioni (che verranno poi approfondite), che riprendono appunto *MONI* e *PHONI* per la *RLBWT*;

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

Prima di procedere con i dettagli dei due metodi è bene proporre un veloce esempio di array MS.

Esempio 21. *Si riprenda nuovamente l'esempio 15, con un pannello e i match con la query z:*

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 sarebbe, avendo scelto come riga iniziale la 19:

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

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

3.5.2 Threshold per la RLPBWT

Come anticipato una prima strategia per la scelta di una nuova riga j, qualora la precedente riga i comporti un mismatch in colonna k con l'aplotipo query, avendo $x_i[k] \neq z[k]$ è l'utilizzo delle **threshold**.

Definizione 20. *Data la colonna k-esima della* ***matrice PBWT****, y^k, memorizzata tramite compressione* ***run-length*** *e data la run j-esima, indicizzata da i a i′, si definisce* ***threshold*** *come l'indice del minimo valore LCP, che ricordiamo essere calcolato sull'ordinamento inverso, compreso negli indici della run, compreso l'eventuale $LCP_k[i' + 1]$, qualora $i' \neq M - 1$. Si noti che quest'ultimo valore, se esistente, deve essere considerato in quanto per il suo calcolo, come specificato nei preliminari alla sezione 2.6, si prende in considerazione $y^k_{i'}$ e $y^k_{i'+1}$.*

Con tale informazione, unita ai *prefix array sample*, si può quindi ottenere un comportamento analogo a quanto si ottiene con l'**R-index** per la *RLBWT*.
Sia infatti data t la posizione della *threshold* nella run corrente, in colonna k, e si supponga che tale run, con testa all'indice h, non sia associata al simbolo desiderato, ovvero $z[k]$. Si supponga che, con il mapping, si sia arrivati all'indice i della colonna k. Si supponga inoltre che la run successiva abbia testa in indice e. Si hanno due casi possibili, denotando con $LCS(x, y)$ il *longest common suffix* tra le stringhe X e Y e con a_k il *prefix array* in colonna K:

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

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

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

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

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

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

Qualora una colonna presenti solo simboli $\sigma \neq z[k]$, per convenzione, si imposta che $MS[k].row = M$ e si ricomincia, in colonna $k+1$, dall'ultima posizione, indicizzata nel pannello originale dal valore finale del *prefix array sample* dell'ultima run.
In termini implementativi anche le posizioni delle *threshold* vengono memorizzate tramite un *bitvector sparso* per ogni colonna k, avendo che, qualora il minimo LCP si ritrovi nell'indice della testa della run successiva, la posizione della threshold verrà comunque memorizzata all'indice della coda della run corrente. Purtroppo

questa è una situazione di ambiguità, avendo che, seguendo la definizione sopra, avendo la *threshold* a fine run, bisognerebbe scegliere la testa della run successiva, qualora l'indice i si trovi esattamente a fine run. Invece, qualora la *threshold* sia a fine run a causa del fatto che il minimo LCP si trovi nella testa della run successiva, bisogna scegliere la coda della run precedente. L'unico modo per disambiguare è quindi effettuare *random access* al pannello per vedere quale sia la soluzione migliore, ovvero quale tra la coda della run precedente e la testa della run successiva siano relative alla riga del pannello originale con un suffisso comune alla query più lungo.
Purtroppo non è possibile salvare la threshold direttamente nella testa della run successiva in quanto questa potrebbe essere anche la posizione della threshold della run successiva e avere due threshold sovrapposte impedirebbe di capire a quale run appartiene una certa threshold, tramite la funzione *rank*.
Tale bitvector deve essere quindi aggiunto alle informazioni memorizzate per ogni singola colonna. Lo pseudocodice per la costruzione di una colonna con anche il bitvector per le *threshold* è consultabile all'algoritmo A.4.

Esempio 22. *Si vede quindi un esempio di funzionamento delle threshold.*
Si prenda pannello visto all'esempio 12 e si effettui la permutazione secondo a_2*:*

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

Si prenda la seconda run, di simboli $\sigma = 1$*, indicizzata tra 17 e 18.*
Si supponga che, tramite il mapping, si sia arrivati alla riga 17 ma che si abbia $z[2] = 0$*. la scelta è quindi tra la coda della run precedente, avendo che* $a_2[16] = 16$

o la testa della run successiva, avendo che $a_2[19] = 17$*. Si può notare come il minimo LCP si trovi, per la run, all'indice 18 (a causa del fatto che il minimo LCP è all'indice 19, quello della testa della run successiva). Si può quindi proseguire o con la riga. Questo significa che il suffisso comune più lungo con la query si ha con la riga 16 del pannello, per definizione di threshold, avendo che questa sarà memorizzata nell'array* MS*:*

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

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

SISTEMARE ESEMPIO

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

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

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

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

Come si può verificare nell'esempio 21.
L'algoritmo per il match tramite *threshold* è visualizzabile all'algoritmo A.15. **CAPIRE SE DESCRIVERE ALGORITMO**

3.5.3 LCE query per la RLPBWT

La memorizzazione di un *bitvector sparso* atto a rappresentare le posizioni delle threshold ha però un costo in memoria. Inoltre, per ora, si è parlato di tenere in memoria il pannello sotto forma di vettore di *bitvector* e anche questo ha un costo non indifferente in termini di spazio necessario.
Per arrivare all'implementazione conclusiva della *RLPBWT* si è quindi optato per la memorizzazione del pannello sotto forma di *SLP*, struttura che, come anticipato, permette anche di eseguire efficientemente le *LCE query*.

Definizione 21. *Dato un pannello X, $M \times N$, e due righe x_i e x_j tali che $0 \leq i < m$ e $0 \leq j < M$, con $i \neq j$, si definisce* ***LCE query*** *il suffisso comune più lungo tra le due stringhe. Per comodità nella sezione si definisce la funzione:*

$$LCE_k(x_i, x_j) = l$$

dove l è la lunghezza del suffisso comune più lungo tra le due stringhe terminante in colonna $k-1$.

Compressione del panel

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

Esempio 23. *Si vede quindi un breve esempio.*
Si assuma di avere il seguente pannello nel file in input.

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

Dove però come detto le righe sono i siti e le colonne i sample. Per ottenere l'SLP biosgna quindi, in primis, trasporre la matrice:

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

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

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

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

0010 0110 0111 1100 0110

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

ESEMPIO MAGARI DA SCRIVERE MEGLIO

LCE query

Grazie all'uso delle *LCE query* è quindi possibile calcolare l'array delle *matching statistics* in un solo scorrimento da sinistra a destra. Infatti è possibile usare tali query per calcolare non solo quale nuova sequenza scegliere in caso di mismatch con l'aplotipo query in colonna i, come si faceva con l'uso delle *threshold*, ma anche di computare la lunghezza del suffisso comune tra essa e l'aplotipo query, calcolando nello stesso momento sia $MS[i].row$ che $MS[i].len$.
Anche in questo caso, per convenzione, si inizia la computazione dell'ultima riga della prima colonna.
Si illustra ora come computare l'array delle *matching statistics*. Si assuma di avere calcolato l'array MS di una query z rispetto al pannello X. le cui righe si

identificano tramite $x_i, \forall i \in \{0, M\}$, fino alla colonna $k-1$. Sia i l'indice di riga sulla *matrice PBWT* al quale si è arrivati mediante il mapping, avendo che tale riga è quella che ha il più lungo suffisso comune con $z[1, k-1]$. Si assuma che l'indice i appartenga alla run r, di simboli σ, testa di indice h e coda di indice $e-1$. Si hanno diversi casi:

1. $z[k] = \sigma$, quindi la riga i può essere usata per estendere il match, avendo che $MS[k].row = MS[k-1].row$ e $MS[k].len = MS[k-1].len + 1$, e per proseguire col mapping in colonna $k+1$

2. $z[k] \neq \sigma$ e si una sola run in colonna k, avendo quindi che non si possono avere match. Per convenzione, si imposta che $MS[k].row = M$ e $MS[k].len = 0$. Infine si ricomincia, in colonna $k+1$, dall'ultima posizione, indicizzata nel pannello originale dal valore finale del *prefix array sample* dell'ultima run

3. $z[k] \neq \sigma$ ma si hanno anche altre run, dovendo quindi scegliere la nuova riga da seguire. Si ha che il più lungo suffisso di $z[1, k]$ che è anche suffisso di $x_1[1, k], \ldots, x_m[1, k]$ è uno tra:

   - $x_{a_k[h-1]}$, se $h \neq 0$, ovvero la riga del pannello corrispondente alla fine della run precedente a r nella *matrice PBWT*, se esistente
   - $x_{a_k[e+1]}$, se $e \neq M-1$, ovvero la riga del pannello corrispondente all'inizio della run successiva a r nella *matrice PBWT*, se esistente

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

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

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

SISTEMARE

Esempio 24. *Riprendiamo l'esempio 22, visto per il calcolo tramite threshold. Senza usare le threshold, nella medesima situazione si dovrebbe calcolare, avendo che* $MS[1].row = 19$ *e* $MS[1].len = 2$*:*

$$LCE_1(x_{19}, x_{16}) = 2$$

$$LCE_1(x_{19}, x_{17}) = 1$$

Come verificabile dal pannello presente all'esempio 12.
Si ha quindi che $MS[2].row = 16$*. Inoltre, sempre per quanto detto sopra:*

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

Con questa variante quindi:

- non si necessita di tenere in memoria le informazioni per le *threshold*
- si tiene in memoria il pannello sotto forma di *SLP*, soluzione vantaggiosa dal punto di vista della memoria anche se svantaggiosa da quello temporale (come descritto nella sezione 2.3)
- si permette il calcolo dell'array MS in una singola scansione del pattern

Essendo lo scopo principale della tesi la riduzione dello spazio occupato dalla struttura dati questa è la soluzione migliore, avendo in memoria una struttura run-length encoded per la PBWT in grado di permettere pattern matching con un aplotipo esterno.

3.6 Funzione Phi

L'ottenimento dell'array *matching statistics* permette di sapere solo l'indice di una della righe del pannello per le quali si ha un match con l'aplotipo query. Analogamente a quanto discusso in *PHONI* [8], anche per la *RLPBWT* si è pensato a due funzioni, $\boldsymbol{\varphi}$ e $\boldsymbol{\varphi^{-1}}$, per il riconoscimento di tutte le righe del pannello per cui si ha il match.
L'intuizione alla base del ragionamento è molto semplice. Nell'ordinamento alla colonna k-esima, dato da a_k, tutte le righe per le quali si ha un match sono poste consecutivamente, questo a causa del fatto che l'ordinamento è lessicografico.

Definizione 22. *Dati:*

- *un pannello* X*, di dimensioni* $N \times M$

- *una colonna k, il prefix array a_k e la sua permutazione inversa α_k*

Si definiscono formalmente:

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

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

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

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

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

VERIFICARE DEFINIZIONE IN QUANTO "NUOVA"

Esempio 25. *Per praticità si riporta un breve esempio.*
Si ipotizzi di avere, come per l'esempio 12:

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

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

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

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

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

Avendo quindi $MS[i].row = p$ e $MS[i].len = l$ basta iterare le righe a partire a p in a_i, che denotiamo con l'indice q, fino a che si ha si ha $LCE_k(x_p, x_q) \geq l$. Ovviamente bisogna iterare in entrambe le direzioni. Tutte le righe x_q che soddisfano un match di lunghezza l con l'aplotipo query. L'algoritmo 3.1 rappresenta esattamente quanto detto, avendo che con la funzione *lce_bounded* si limita il calcolo della LCE alla lunghezza l.

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

Algoritmo 3.1 Algoritmo per estendere un match in colonna k usando φ, φ^{-1}.

```
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
```

3.6.1 Costruzione della struttura di supporto

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

- un vettore di *sparse bitvector* 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 *sparse bitvector* 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 interi a supporto, denotato Φ_{supp}, del vettore di *sparse bitvector* per φ che memorizza, per ogni 1 di tale vettore, il *prefix array sample* della coda della run precedente o l'altezza del pannello, M, qualora non si abbia alcuna run precedente

- un vettore di interi a supporto, denotato Φ^{-1}_{supp}, del vettore di *sparse bitvector* per φ^{-1} che memorizza, per ogni 1 di tale vettore, il *prefix array sample* della testa della run successiva o l'altezza del pannello, M, qualora non si abbia alcuna run successiva.

Si ha quindi che la lunghezza della riga i-esima di Φ_{supp} è uguale al numero di uni presenti nella riga i-esima di Φ. Analogamente si ha per Φ^{-1}_{supp}. In entrambi i casi, inoltre, si hanno M righe.
Al fine della costruzione bisogna, inoltre, sfruttare a_{N-1} per poter identificare quelle coppie di valori consecutivi non presenti nei vari *prefix array samples*, in modo che sia possibile effettuare le query per qualsiasi valore di *prefix array* in input.
L'algoritmo 3.2 riporta quindi la costruzione della struttura, iterando in primis i vari *prefix array samples* e completando i risultati con a_{N-1}.
CAPIRE SE COMMENTARE ULTERIORMENTE LA COSTRUZIONE Dal punto di vista delle query, data una colonna k e un valore di *prefix array* p, si procede quindi nel seguente modo:

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

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

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

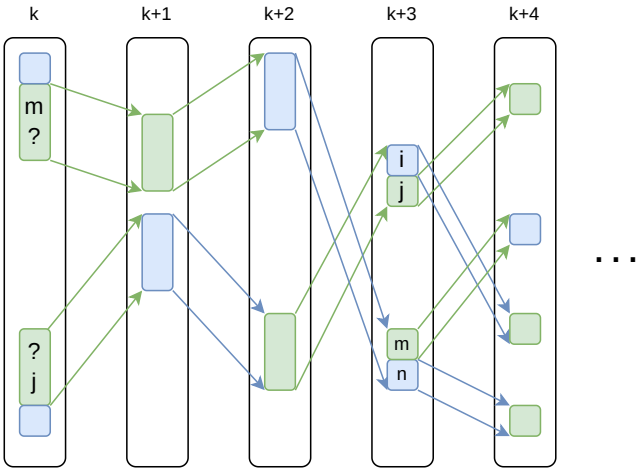
$$\varphi^{-1}_k(p) = \begin{cases} null & \text{se } \Phi^{-1\ p}_{supp}[rank^{\varphi^{-1}}_p(k)] = M \\ \Phi^{-1\ p}_{supp}[rank^{\varphi^{-1}}_p(k)] & \text{altrimenti} \end{cases}$$

Si riporta un esempio chiarificatore.

Esempio 26. *Si ha la seguente situazione nella matrice PBWT:*

Algoritmo 3.2 Algoritmo per la costruzione della struttura per φ e φ^{-1}

function BUILD_PHI($cols, panel, prefix$) ▷ $prefix$ is the last prefix array
  $\Phi \leftarrow [[0..0]..[0..0]],\ \Phi^{-1} \leftarrow [[0..0]..[0..0]]$ ▷ sparse bit vector panels for φ and φ^{-1}
  $\Phi_{supp} = [],\ \Phi^{-1}_{supp} = []$ ▷ vectors for φ and φ^{-1} row values
  for *every* $k \in [0, |cols|)$ **do**
    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**
    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
        $push(\Phi^{-1}_{supp}[prefix[k]], prefix^k[i+1])$
  build rank for every sparse bitvector in Φ and Φ^{-1}



Dove si noti che, a parità di colore, si ha lo stesso simbolo tra due indici consecutivi.
In colonna k, che per praticità assumiamo essere $k = 0$, si vorrebbe avere informazione in merito a $\varphi_k(j)$ e $\varphi_k^{-1}(m)$.
Si nota che, per definizione della struttura dati, si ha che (limitandoci alle colonne della figura):

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

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

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

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

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

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

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

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

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

SISTEMARE

Capitolo 4

Risultati

Verranno ora riportati alcuni risultati sperimentali, ottenuti su pannelli simulati, relativi all'implementazione, in C++11, della **RLPBWT**.

4.1 Introduzione agli strumenti usati

Prima di addentrarci nella discussione dei risultati è bene trattare gli strumenti computazionali usati durante la fase di sviluppo.
Dal punto di vista dei linguaggi di programmazione si sono usati:

- **C++**, per l'implementazione delle strutture dati e degli algoritmi
- **Python**, per la creazione della struttura a partire dal pannello in input e per gestire l'intera pipeline di sperimentazione, partendo dal *preprocessing* dell'input fino alla produzione dei grafici al termine della computazione

Nel dettaglio la costruzione della **RLPBWT**, i cui singoli step verranno approfonditi nel corso del capitolo, si articola nel seguente modo:

1. ***input:*** pannello binario generato tramite **MaCS**
2. **opzionale:** produzione dell'**SLP** del pannello
3. **step intermedio:** estrazione dal pannello in input di un pannello di query di grandezza selezionata dall'utente e costruzione della struttura dati
4. **opzionale:** serializzazione della struttura dati, tramite **SDSL**
5. ***output:*** file contenente i risultati dei match

Si specifica che per il file di output si è mantenuto lo stesso formato utilizzato da Durbin nella sua implementazione della **PBWT** [27]. Tale formato prevede, per facilitarne il parsing, un `tsv` (*tab-separated values*) con le seguenti colonne:

1. colonna semplicemente indicante che si ha un *MATCH*
2. l'indice della query di cui si annota il match
3. l'indice dell'aplotipo per cui si ha il match
4. l'indice della colonna da cui parte il match
5. l'indice della colonna in cui termina il match
6. la lunghezza del match

4.1.1 MaCS

RIVEDERE!
MaCS [28], sviluppato da Gary K. Chen, è un simulatore di *processi coalescenti*, basati sulla **teoria della coalescenza**. Tale teoria è un modello di come gli alleli campionati da una popolazione possano essere originati da un antenato comune. Il tool simula genealogie spaziali tra i cromosomi sfruttando processi Markoviani. Nel dettaglio il lavoro è fortemente ispirato dai risultati di Wiuf e Hein [29], che per primi proposero un algoritmo basato sulla costruzione e sulla memorizzazione di un **ancestral recombination graph (*ARG*)**.
Chen stesso segnala le seguenti differenze con l'algoritmo di Wiuf e Hein:

- gli eventi di ricombinazione si verificano solo sulla geneologia locale nella posizione attuale sulla sequenza invece che in qualsiasi altro punto dell'*ARG*, ma possono unirsi a qualsiasi lignaggio sull'ARG compresi quelli non sulla geneologia locale (ad esempio un arco non ancestrale)
- i tempi di attesa (ovvero la distanza tra le ricombinazioni sulla sequenza) sono calcolati in modo esponenziali con intensità basata sulla lunghezza dell'arco della geneologia locale invece della lunghezza ARG
- l"algoritmo è detto dell'n-esimo ordine Markoviano, dove n è basato sui parametri inserito dall'utente

L'autore ricorda che queste modifiche rendono l'algoritmo sostanzialmente più efficiente del Wiuf e Hein con poca perdita di precisione.
Dal punto di vista pratico l'esecuzione di *MaCS* produce i pannelli binari, da intendersi come pannelli di aplotipi, che verranno poi studiati tramite la *PBWT* e la *RLPBWT*. Tali pannelli presentano:

- un header, con informazioni in merito al comando usato e al seed
- una riga per ogni sito, con prima alcune informazioni in merito a come è stato prodotto il dato e poi la sequenza di valori binari, uno per ogni sample
- un footer, con ulteriori informazioni, tra cui le dimensioni del pannello

Quindi, trascurando le varie informazioni aggiuntive, il pannello è **trasposto** rispetto a quanto studiato dalla *PBWT* e dalla *RLPBWT*. Questa però risulta essere una comodità in quanto, leggendo iterativamente il file, si legge di volta in volta la i-esima colonna, ovvero quanto serve per la costruzione della struttura dati.
Per capire meglio come venga prodotto un pannello tramite questo strumento, analizziamo un semplice esempio:

```
./macs 5 3000 -t 0.001 -r 0.001
```

Dove:

- 5 è il numero di sample richiesto, ovvero il numero di sequenze che il software simulerà
- 3000 è la lunghezza in paia-basi della regione genomica su cui verranno simulate le 5 sequenze
- `-t` 0.001 segnala il *mutation rate* per ogni sito, ovvero la frequenza di *nuove mutazioni* per un sito nel tempo
- `-r` 0.001 segnala il *recombination rate* per ogni sito, ovvero la frequenza di *ricombinazioni geniche*, che sono i processi per i quali si ottengono nuove combinazioni di alleli a partire da un *genotipo*, per un sito nel tempo

Il risultato, dove si noti vengono selezionati 4 siti dopo la simulazione, del comando appena descritto è visualizzabile alla listing 1.

Listing 1 Esempio di output di **MaCS**.

```
COMMAND:        ./macs 5 3000 -t 0.001 -r 0.001
SEED:   1656169933
SITE:   0          0.0364063206     0.103883682 00100
SITE:   1          0.0808017426     0.668414883 11101
SITE:   2           0.413947166      0.25054948 01111
SITE:   3           0.714643416     0.175509499 00010
TOTAL_SAMPLES:  5
TOTAL_SITES:    4
BEGIN_SELECTED_SITES
0       1       2       3
END_SELECTED_SITES
```

4.1.2 BigRePair e ShapedSlp

Come introdotto alla sezione 2.3, una delle varianti della **RLPBWT**, richiede l'uso, estremamente vantaggioso dal punto di vista della memoria occupata, degli **SLP**.
Da un punto di vista implementativo, l'oggetto contenente l'*SLP* del pannello viene costruito ed interrogato mediante l'uso della libreria **ShapedSlp** [30], implementazione dei risultati ottenuti da Gagie et al. [3]. Inoltre, tale libreria basa il suo funzionamento sull'uso di un'altra libreria, detta **BigRePair** [31], che implementa i quanto studiato da Gagie et al. [32] in merito alla compressione, via uso di grammatiche, di file con frequenti ripetizioni (come possono essere, nel nostro caso, pannelli binari di aplotipi).
In termini di pipeline si procede quindi:

1. generando la *grammatica* tramite *BigRePair*, che accetta come file di input un file `txt` "raw" ma anche un file in formati più standard come i *FASTA*

2. generando l'*SLP* tramite *ShapedSlp* specificatamente a partire dai risultati di *BigRePair* (si segnala che la libreria accetta anche input prodotti tramite altri tool che non verranno qui approfonditi)

4.1.3 Snakemake

L'intera pipeline è stata gestita tramite **Snakemake** [33], un *workflow management system*, uno strumento molto usato in *bioinformatica* per creare analisi dati scalabili e riproducibili. Nel dettaglio la pipeline comprende, avendo in input un pannello ed una lista di quantità di aplotipi di query:

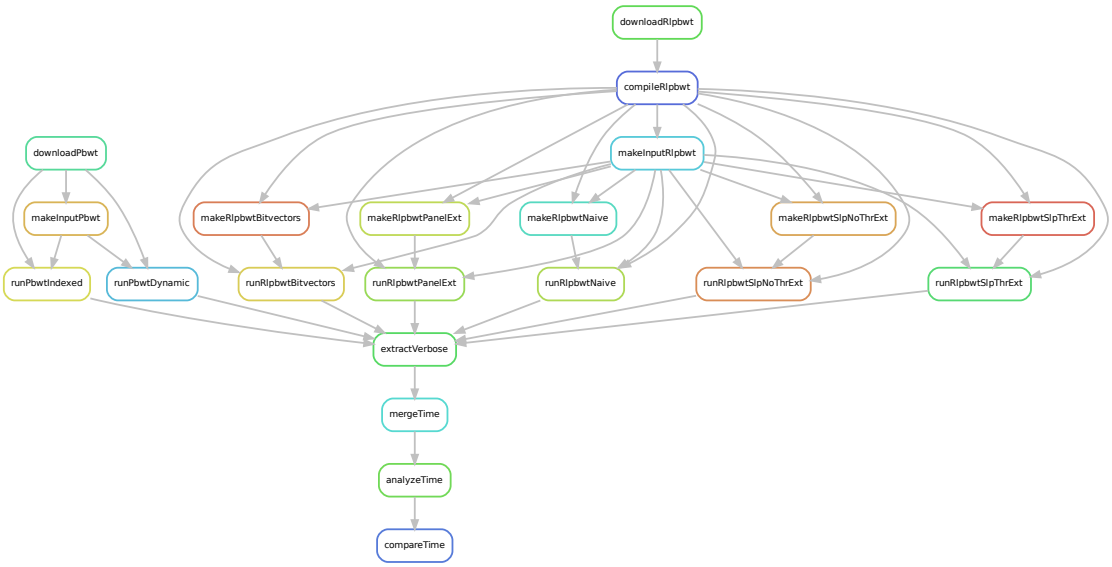


Figura 4.1: Regole usate in Snakemake per la sperimentazione.

- scaricamento dei tool e delle loro dipendenze per la *PBWT* di Durbin e la *RLPBWT* proposta in questa tesi
- produzione dell'input per la *PBWT* e della *RLPBWT* per ogni quantità di query richiesta
- produzione delle strutture dati
- esecuzione degli algoritmi per il pattern matching
- produzione di vari grafici relativi sia ai tempi di esecuzione che alla memoria richiesta

Al fine di ottenere risultati non banali infatti l'idea è quella di partire da un pannello iniziale fisso ed estrarre un numero di righe pari al numero di query richieste, righe che, a loro volta, andranno a formare il pannello di query.
Uno schema delle regole usate è visualizzabile in figura 4.1.

4.2 Confronto tra PBWT e RLPBWT

Al fine di analizzare i risultati ottenuti si sono confrontate 5 varianti della *RLPBWT*:

- *RLPBWT naive*

- *RLPBWT con bitvector*
- *RLPBWT con pannello completo e threshold*
- *RLPBWT con pannello compresso (SLP) e threshold*
- *RLPBWT con pannello compresso (SLP) e LCE query*

Confrontandole con l'implementazione originale dell'algoritmo 5 di Durbin, nominato *MatchIndexed*. Studiando la repository di Durbin inoltre si è scoperto l'esistenza di un ulteriore algoritmo, non descritto formalmente nel paper del 2014 [20] ma solo citato in una tabella, che considera in un unico panello sia il pannello che l'insieme delle query ed effettua il matching interno al pannello stesso, calcolando in modo dinamico l'indicizzazione ad ogni colonna. Nonostante l'algoritmo presenti limiti dal punto di vista dell'estendibilità ad altre problematiche, avendo che le varianti della *PBWT* citate in sezione 2.6 si basano, nel caso di match con aplotipi esterni, sulle idee dell'algoritmo 5, esso risulta essere davvero molto performante sia dal punto di vista del tempo macchina che della memoria occupata. A causa di ciò, per completezza, tale algoritmo, chiamato *MatchDynamic*, è stato incluso nei risultati sperimentali, pur mancandone una trattazione teorica approfondita.

4.2.1 Analisi spaziale

Lo scopo principale di questa tesi era la riduzione delle informazioni in memoria necessarie a permette il mapping, quindi in primis si sono valutati i vari risultati dal punto di vista della memoria.

Dimensioni dell'SLP

Prima ancora di affrontare i requisiti in memoria dell'intera struttura è interessante analizzare le capacità di compressione che si ha con l'uso degli *SLP*, grazie ai due tool sopra citati. In figura 4.2 si può iniziare ad apprezzare l'efficacia di tale grammatica. Si nota infatti come, per quanto i pannelli siano di dimensione modesta, hanno un peso che varia in un range di un centinaio di megabytes mentre gli *SLP* relativi nel centinaio di kilobytes. Si ha infatti:

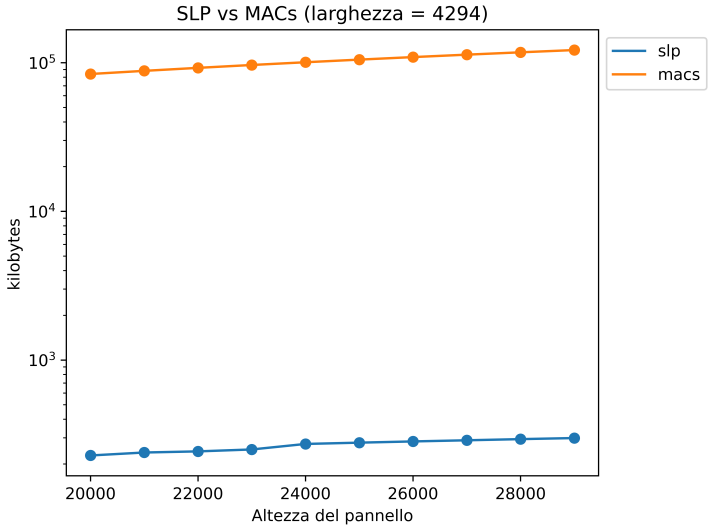


Figura 4.2: Confronto delle dimensioni, espresse in kilobytes, dei pannelli in formato `macs` e dei rispettivi *SLP*. Il grafico è in scala logaritmica.

altezza	**larghezza**	**SLP (*kb*)**	**MACs (*kb*)**	%
20000	4294	228.13	84050.48	0.2714
21000	4294	238.83	88243.83	0.2707
22000	4294	243.04	92437.18	0.2629
23000	4294	250.37	96630.53	0.2591
24000	4294	272.72	100823.87	0.2705
25000	4294	278.22	105017.22	0.2649
26000	4294	283.57	109210.57	0.2597
27000	4294	288.62	113403.92	0.2545
28000	4294	293.85	117597.27	0.2499
29000	4294	298.76	121790.62	0.2453

Notando come, per pannelli di grandezza simile, pare si abbia una compressione proporzionale alla dimensione del pannello.

Andando a vedere pannelli molto più grossi si nota come il rateo di compressione continui essere proporzionale alla dimensione del pannello e, nonostante il esso cresca di dimensione, la grandezza dell'*SLP* resta molto piccola:

altezza	**larghezza**	**SLP (*kb*)**	**MACs (*kb*)**	%
100000	358653	14771.0	35042963.54	0.0422
100000	100000	9077.88	9075120.49	0.1
100000	46538	8017.09	4448994.19	0.1802

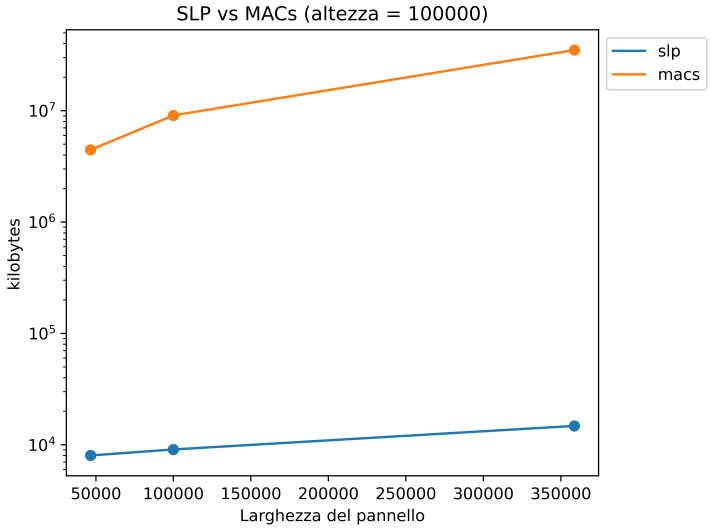


Figura 4.3: Confronto delle dimensioni, espresse in kilobytes, dei pannelli in formato `macs` e dei rispettivi *SLP*. Il grafico è in scala logaritmica.

Tale risultato è anche apprezzabile in figura 4.3. Il caso estremo, un pannello 100000×358653, occupante in memoria circa 35gb in formato `.macs`, viene compresso in circa 15mb. Questo accade soprattutto in quanto un pannello di soli simboli $\Sigma = \{0, 1\}$ contiene molte ripetizioni, permettendo la costruzione di una grammatica, tramite l'*SLP*, particolarmente "compatta".

Strutture dati

Si analizzano ora le due strutture dati, confrontando lo spazio richiesto dalle varie sotto-strutture per effettuare il match con una query esterna, descritte alle sezioni 2.6, 3.2, 3.3 e 3.5.
Si precisa che i dati ora descritti sono stati calcolati nel seguente modo:

- per quanto riguarda la **PBWT**, sfruttando le stime fatte da Durbin stesso
- per quanto riguarda la **RLPBWT**, sfruttando le serializzazioni ottenute tramite *SDSL*

Con un studio al leggero variare del pannello si nota, graficamente in figura 4.4, come quanto descritto precedentemente venga confermato. Le informazioni richieste dall'algoritmo 5 di Durbin sono quelle che richiedono maggior memoria mentre la variante della *RLPBWT* basata su *SLP* e *LCE query* risulta essere la soluzione

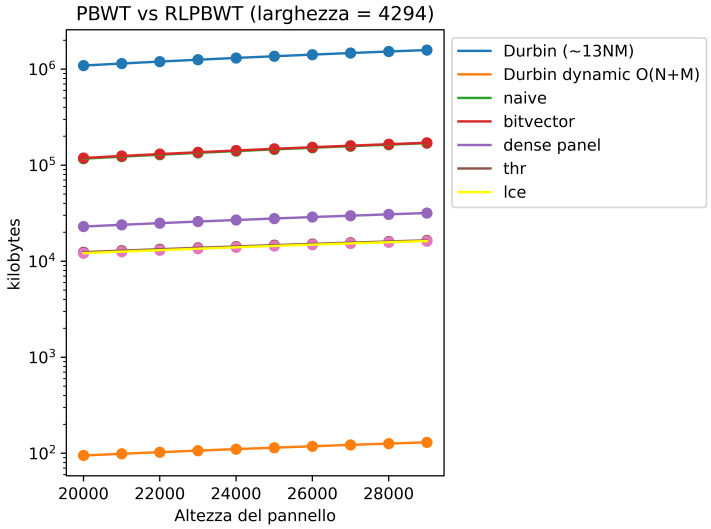


Figura 4.4: Confronto dello spazio in memoria, in kilobytes, richiesto dalle varie strutture dati.

migliore tra le varianti della *RLPBWT*. Bisogna però notare come la soluzione *matchDynamic* ritrovabile nella repository della *PBWT* risulti essere incredibilmente più efficace, avendo, secondo Durbin stesso, una richiesta in spazio proporzionale a $\mathcal{O}(M + N)$.
Limitiamo però ora il confronto all'algoritmo 5 di Durbin, in quanto obbiettivo della tesi. Da un punto di vista di guadagno percentuale in memoria i risultati sembrano essere interessanti, confrontando tale soluzione con la migliore per la *RLPBWT*:

altezza	**larghezza**	**RLPBWT SLP-LCE (*kb*)**	**PBWT Indexed (*kb*)**	%
20000	4294	12118.62	1090270.65	1.1115
21000	4294	12583.13	1144784.18	1.0992
22000	4294	13033.78	1199297.71	1.0868
23000	4294	13487.57	1253811.24	1.0757
24000	4294	13954.44	1308324.78	1.0666
25000	4294	14419.27	1362838.31	1.058
26000	4294	14867.82	1417351.84	1.049
27000	4294	15316.41	1471865.37	1.0406
28000	4294	15765.41	1526378.9	1.0329
29000	4294	16214.09	1580892.44	1.0256

Provando in modo quantitativo l'efficacia in memoria della soluzione ultima proposta in questa tesi.

4.2.2 Analisi temporale

Bisogna infine considerare i tempi di esecuzione per il pattern matching con un pannello di query. Dal punto di vista della *RLPBWT* bisogna considerare in primis due aspetti:

- avere meno informazione in memoria comporta molto probabilmente, a parità di risultati, tempi maggiori
- l'uso di strutture dati succinte ed eventualmente dell'*SLP* comporta costi dal punto di vista temporale. Come anticipato in sezione 2.2, le operazioni sugli sparse bitvector non sono tutte i tempo costante e, come invece anticipato in sezione 2.3, gli *SLP* non garantiscono *random access* in tempo costante e questo, per quanto poi l'algoritmo di estensione sia efficiente, si ripercuote anche sul calcolo delle *LCE query*

Questa premessa fa capire come ci si aspettasse che i tempi fossero maggiori con la *RLPBWT*, in ogni sua variante, rispetto all'algoritmo 5 di Durbin. Parlando invece dell'algoritmo *matchDynamic* si ha che, per quanto asintoticamente presenti la stessa complessità dell'algoritmo 5, ovvero $\mathcal{O}(N(M+Q))$, con Q numero di query, esso risulta incredibilmente più performante.
Alcuni risultati sono visualizzabili in figura 4.5 e 4.6, dove si possono osservare sia i tempi che lo spazio richiesto. Anche la completa esecuzione quindi conferma come l'algoritmo 5 sia incredibilmente esoso dal punto di vista dello spazio richiesto, pur avendo ottime performance temporali. Dal punto di vista invece delle varianti della *RLPBWT* si nota come:

- la *RLPBWT naive*, priva dell'uso dei bitvector e dell'*SLP*, risulti essere la più performante, anche se, si ricordi, non permette di identificare quali righe stiano effettivamente matchando ma solo quante
- la *RLPBWT con bitvector*, avente lo stesso limite della variante naive, presenta anche maggiori costi in termini di memoria di quest'ultima, avendo anch'essa ancora l'*LCP array* completo ma anche tutte le informazioni memorizzate in bitvector, che aumentano, come anticipato, i tempi di calcolo. La chiave delle varianti che sfruttano le *matching statistics* è infatti quella di non avere l'*array LCP* in memoria, una delle cause principali dell'aumento di spazio richiesto
- le tre varianti basate sulle *matching statistics* hanno spazio occupato pressoché uguale, anche se si può percepire, nei due casi studiati, come il tenere l'intero pannello in forma di bitvector, all'aumentare della grandezza dello stesso, comporti molta più memoria degli *SLP*. Dal

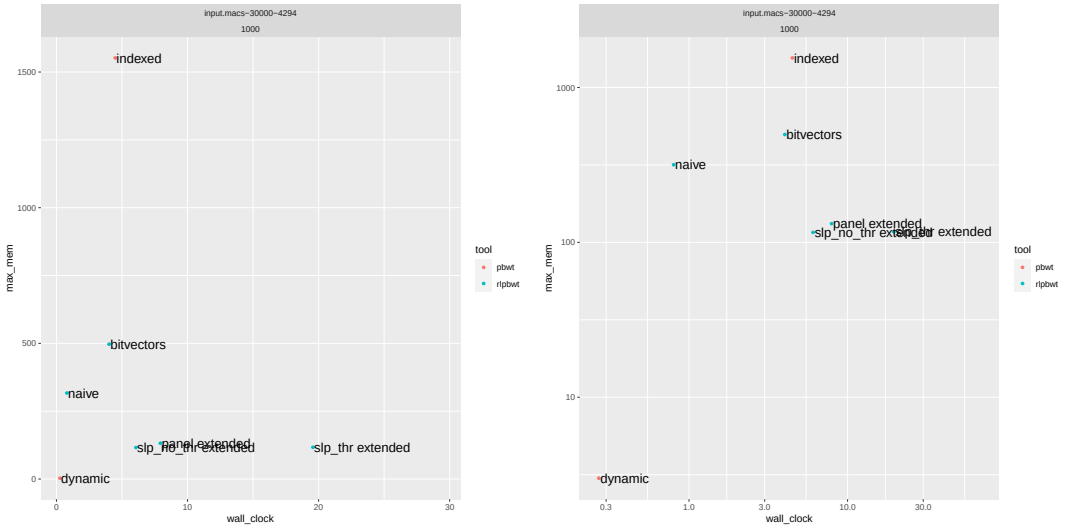


Figura 4.5: Esecuzione dei vari algoritmi di match su un pannello 29000 × 4294 e 1000 query. Il grafico a destra è in scala logaritmica.

punto di vista temporale, inoltre, anche se si ha *random access* in tempo costante, all'aumentare del pannello, il numero di accessi allo stesso comporta forti costi in termini di tempo macchina. Questi ultimi, infatti, come già visto occupano pochissima memoria anche con pannelli molto estesi. Dal punto di vista temporale si rileva come la variante basata su *SLP* e *threshold* richieda molto più tempo. Si nota che ciò accade a causa di due fattori:

- il continuo accesso all'SLP per calcolare $MS[i].len$
- l'eventuale accesso all'SLP per disambiguare le threshold a fine run

Tra le tre quindi la variante con *SLP* e *LCE query*, all'aumentare della grandezza del pannello, risulta essere la soluzione migliore

Per completezza, in figura 4.7, si riportano anche i risultati in tempo e spazio di una sperimentazione su un pannello di grandi dimensioni: 70000 × 46538 con 30000 query. Sono riportati anche i risultati delle tre varianti basate su *matching statistics* senza l'estensione dei match tramite le **funzioni ϕ** e ϕ^{-1}. Si può notare come la struttura dati aggiuntiva non comporti praticamente alcuna differenza sostanziale sia in termini di memoria che di tempo di calcolo. In merito agli altri risultati si ha che seguono tutti il trend già descritto negli esempi precedenti. In particolare si nota che:

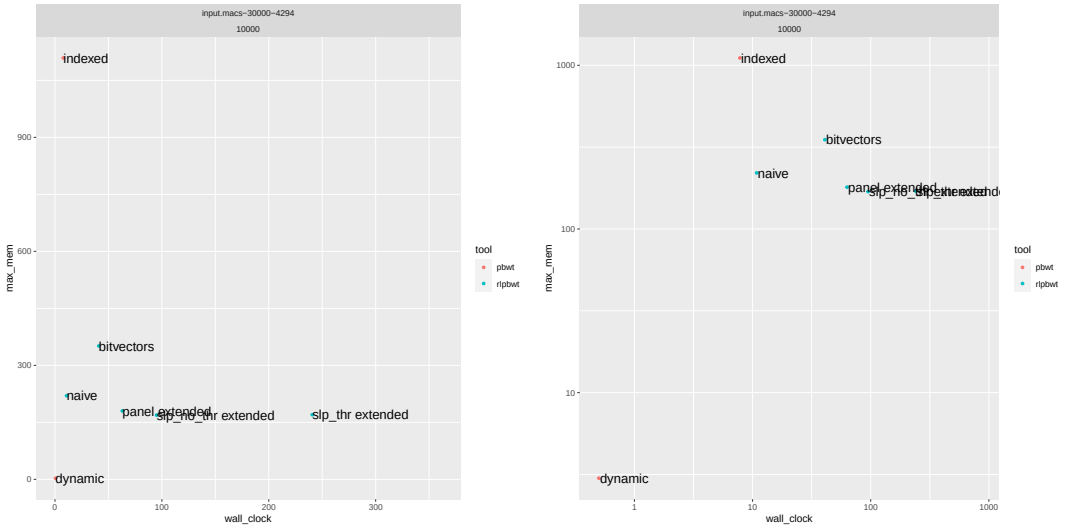


Figura 4.6: Esecuzione dei vari algoritmi di match su un pannello 20000 × 4294 e 10000 query. Il grafico a destra è in scala logaritmica.

- l'algoritmo 5 di Durbin ha una richiesta di memoria davvero molto grande, parlando di circa 40.75 gb di memoria richiesta
- l'algoritmo *matchDynamic* di Durbin risulta essere migliore sia dal punto di vista dello spazio richiesto che del tempo d'esecuzione
- la variante *RLPBWT* con *SLP* e *threshold*, per le problematiche già descritte richiede un tempo d'esecuzione importante, parlando di circa 2 ore di esecuzione

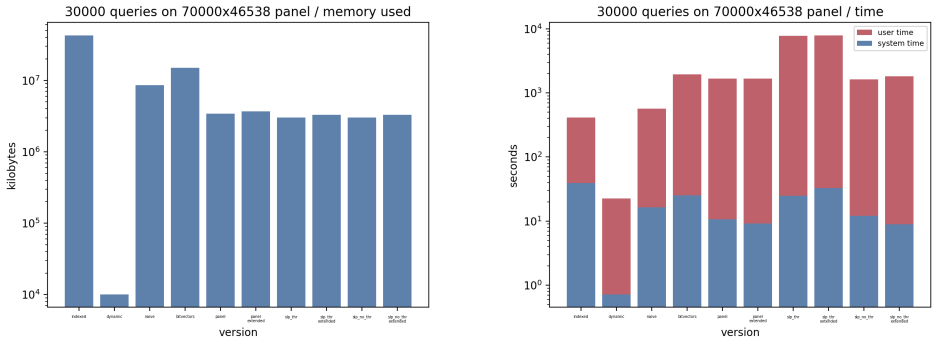


Figura 4.7: Risultati, in scala logaritmica, dell'esecuzione dei vari algoritmi su un pannello di grosse dimensioni. Si noti che, quantitativamente, la variante *matchIndexed* richieda 42736132 kb di memoria mentre la *RLPBWT* con *SLP* e *LCE query* solo 3286424 kb, richiedendo quindi appena il 7% di memoria richiesta dall'algoritmo 5 di Durbin.

Capitolo 5

Conclusioni

Fissato l'iniziale obbiettivo di risolvere le problematiche relative alla memoria richiesta dall'algoritmo 5 di Durbin, l'implementazione della **RLPBWT**, nel dettaglio basata sull'uso dei *SLP* e *LCE query*, ha riportato risultati molto incoraggianti. Come descritto nel capitolo 4, la quantità di memoria richiesta risulta essere incredibilmente inferiore. D'altro canto l'algoritmo *matchDynamic* di Durbin, per quanto non approfondito nell'articolo del 2014 [20], risulta essere ancor meno esoso di risorse, nonché incredibilmente più veloce dal punto di vista dei tempi di calcolo. Lo svantaggio di questo algoritmo è che i risultati sono prodotti in "ordine sparso" e, giudicando la letteratura degli ultimi anni le cui trattazioni si basano sempre sull'algoritmo 5, che non sembra essere facilmente riadattabile per la risoluzione di altri task.
Si possono comunque rilevare alcune possibili migliorie in merito all'implementazione attuale della *RLPBWT*:

- si potrebbe pensare ad un metodo per gestire in modo efficiente lo studio di più query contemporaneamente, migliorando i tempi di calcolo complessivi
- studiare eventuali ottimizzazioni per l'algoritmo di mapping e per le strutture dati richieste, studiando, ad esempio, se sia possibile tenere in memoria un solo bitvector uv_k che funzioni in modo similare a quanto si era inizialmente pensato per la *RLPBWT naive*
- migliorare il sistema di serializzazione. Allo stato attuale l'intera struttura viene serializzata e caricata in modo completo. Studiare una strategia efficiente per caricare, di volta in volta, in memoria solo la colonna necessaria ad un dato passo di computazione o comunque un sottoinsieme di colonne

Nonostante queste possibili migliorie la qualità dei risultati è sufficiente per stabilire che una variante *run-length encoded* della *PBWT*, alla stregua di quanto analizzato negli ultimi anni sulla *RLBWT* con *MONI* [19] 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 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*, della *medicina personalizzata* etc...

5.1 Sviluppi futuri

Ovviamente questa prima implementazione completa della *RLPBWT* 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:

- **pannelli multi-allelici**, ovvero costruiti su un alfabeto Σ arbitrario e non limitato ai simboli $\sigma = 0$ e $\sigma = 1$
- **pannelli con dati mancanti**, ovvero pannelli costruiti direttamente da *dati reali* che possono contenere siti, per certi individui, per i quali non si ha certezza in merito all'allele

Inoltre, allo stato attuale, la struttura dati è stata sviluppata per permettere unicamente il calcolo dei match massimali con un aplotipo esterno. Anche in questo caso, quindi, si potrebbe avere lo sviluppo di nuovi algoritmi che rispondano a task diversi, come il calcolo dei match interni al panel, i cosiddetti *blocchi*, o anche il calcolo di tutti i match con un aplotipo esterno di lunghezza maggiore ad una fissata o che includano un numero stabilito di sequenze di aplotipi nel panello.

RLPBWT multi-allelica Per quanto i pannelli di aplotipi prodotti dal sequencing del genoma umano raramente presentino siti multi-allelici si ha una presenza stimata, al momento, di circa il 2% di siti tri-allelici [34]. Inoltre, all'aumentare della disponibilità di dati genomici, si ha in letteratura la propensione a credere che tale percentuale di siti sia non solo sottostimata (stimando che sia stimato circa un terzo dei reali siti tri-allelici) ma anche destinata a cresce in modo non lineare rispetto al numero di individui sequenziati [35]. Inoltre, molte specie, soprattutto vegetali, sono già riconosciute essere poliploidi, quindi una struttura dati efficiente in memoria in grado di gestire pannelli costruiti su un alfabeto arbitrario risulterà

necessaria nel breve futuro.
Ipotizzando un possibile funzionamento della **RLPBWT multi-allelica (*m-RLPBWT*)** si può pensare ad una soluzione molto simile a quanto visto per la *RLPBWT*. Infatti, per ogni colonna, si potrebbero memorizzare:

- una stringa che memorizzi quale simbolo corrisponda ad una certa run, non potendo sfruttare l'alternanza di simboli vista nel caso binario
- una rivisitazione delle strutture necessarie al mapping, tenendo in memoria vettori di *bitvector sparsi*
- riadattamento del calcolo dell'array delle *matching statistics*

In merito allo spazio richiesto e ai tempi di calcolo bisognerà considerare la grandezza dell'alfabeto su cui è costruito il pannello, che ci si aspetta comune inferiore a 10 nella maggioranza dei casi di studio biologico.
Nonostante, allo stato dell'arte, ci siano pochissimi studi in merito si ritiene possibile generalizzare, in modo computazionalmente efficiente, la *RLPBWT* anche a questa casistica.

RLPBWT con dati mancanti La maggior parte delle soluzioni attualmente sviluppate sono basate su una forte assunzione: i dati in input sono corretti e senza dati mancanti. Ovviamente, limitandosi a studiare pannelli simulati o comunque "riempiti" in una fase di preprocessing, si rischia di non poter comprendere a fondo l'efficacia dei metodi su dati reali, oltre che a limitare l'inferenza dai pannelli stessi.
Come anticipato alla sezione 2.6, 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 *m-RLPBWT*, per la gestione di dati mancanti nel pannello. Inoltre si potrebbero sviluppare algoritmi in grado di gestire le wildcard anche all'interno delle query stesse.
Sempre in via ipotetica, l'uso di *algoritmi parametrici* (manche anche di *algoritmi approssimati*) adattati al funzionamento della *RLPBWT* potrebbero portare a soluzioni interessanti per la gestione di pannelli reali.

K-MEM 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 [36] ha proposto un articolo in cui dimostra come la struttura implementata in *MONI* [19] sia già predisposta al calcolo dei **k-MEM**, ovvero

match massimali tra sotto-stringhe di un pattern e un testo che occorrono esattamente k volte nel testo stesso.
In merito alla *RLPBWT* si potrebbe adattare l'idea di Gagie al calcolo di match massimali tra sotto-stringhe dell'aplotipo query e il pannello che comportino il match con esattamente k righe del pannello stesso. L'ormai empiricamente dimostrata correlazione tra la *RLBWT* e la *RLPBWT* porta a pensare che tale problema sia risolvibile anche con la nuova definizione di *matching statistics* per la *RLPBWT*.
Ovviamente 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 si sotto-sequenze di un aplotipo esterno all'interno di un pannello di aplotipi.
SISTEMARE

Bibliografia e sitografia

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

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

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

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

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

[6] 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.

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

[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] 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.

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

[11] 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.

[12] Alberto Policriti and Nicola Prezza. Lz77 computation based on the run-length encoded bwt. *Algorithmica*, 80(7):1986–2011, 2018.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] Richard Durbin. Efficient haplotype matching and storage using the positional burrows–wheeler transform (pbwt). *Bioinformatics*, 30(9):1266–1272, 01 2014.

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

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

[23] 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.

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

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

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

[27] Richard Durbin. PBWT. `https://github.com/richarddurbin/pbwt`, 2014.

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

[29] Carsten Wiuf and Jotun Hein. Recombination as a point process along sequences. *Theoretical population biology*, 55(3):248–259, 1999.

[30] I. Tomohiro. ShapedSlp. `https://github.com/itomomoti/ShapedSlp`, 2020.

[31] Giovanni Manzini. BigRePair. `https://gitlab.com/manzai/bigrepair`, 2019.

[32] 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.

[33] 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.

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

[35] 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.

[36] Travis Gagie. Moni can find k-mems. *arXiv preprint arXiv:2202.05085*, 2022.

Appendice A

Algoritmi

Algoritmo A.1 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)
```

Algoritmo A.2 Algoritmo per l'LF-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]
```

Algoritmo A.3 Algoritmo per la costruzione di una colonna della *RLPBWT* naive

```
1:  function BUILD_NAIVE(col, pref, div)
2:      c ← 0, u ← 0, v ← 0, u' ← 0, v' ← 0, run ← 0
3:      start ← ⊤, beg_run ← ⊤, push_zero ← ⊥, push_one ← ⊥
4:      rows ← []
5:      for every k ∈ [0, height) do
6:          if k = 0 ∧ col[pref[k]] = 1 then
7:              start ← ⊥
8:          if col[k] = 0 then
9:              c ← c + 1
10:     if start then
11:         push_one ← ⊤
12:     else
13:         push_zero ← ⊤
14:     for every k ∈ [0, height) do
15:         if beg_run then
16:             u ← u', v ← v'
17:             beg_run ← ⊥
18:         if col[pref[k]] = 1 then
19:             v' ← v' + 1
20:         else
21:             u' ← u' + 1
22:         if k = 0 ∨ col[pref[k]] ≠ col[pref[k − 1]] then
23:             run ← k
24:         if k = height − 1 ∨ col[pref[k]] ≠ col[pref[k + 1]] then
25:             if push_one then
26:                 push(rows, (run, v))
27:                 swap(push_one, push_zero)
28:             else
29:                 push(rows, (run, u))
30:                 swap(push_one, push_zero)
31:             beg_run ← ⊤
32:     return (start, c, rows, div)
```

Algoritmo A.4 Algoritmo per la costruzione di una colonna della *RLPBWT* con bitvectors

```
1: function BUILD_BV(col, pref, div)
2:     c ← 0, u ← 0, v ← 0, u' ← 0, v' ← 0, curr_lcs ← 0, tmp_thr ← 0, tmp_beg ← 0
3:     start ← ⊤, beg_run ← ⊤, push_zero ← ⊥, push_one ← ⊥
4:     for every k ∈ [0, height) do
5:         if k = 0 ∧ col[pref[k]] = 1 then
6:             start ← ⊥
7:         if col[k] = 0 then
8:             c ← c + 1
9:     runs ← [0..0]                                  ▷ sparse bitvector for runs of length height + 1
10:    thrs ← [0..0]                                  ▷ sparse bitvector for thresholds of length height
11:    zeros ← [0..0]                                 ▷ sparse bitvector for zeros of length c
12:    ones ← [0..0]                                  ▷ sparse bitvector for ones of length height − c
13:    samples_beg ← [], samples_end ← []             ▷ couple of vectors for samples of length r
14:    if start then
15:        push_one ← ⊤
16:    else
17:        push_zero ← ⊤
18:    for every k ∈ [0, height) do
19:        if beg_run then
20:            u ← u', v ← v', tmp_beg ← pref[k]
21:            beg_run ← ⊥
22:        if col[pref[k]] = 1 then
23:            v' ← v' + 1
24:        else
25:            u' ← u' + 1
26:        if k = 0 ∨ col[pref[k]] ≠ col[pref[k − 1]] then
27:            curr_lcs ← div[k], tmp_thr ← k
28:        if div[k] < curr_lcs then
29:            curr_lcs ← div[k], tmp_thr ← k
30:        if k = height − 1 ∨ col[pref[k]] ≠ col[pref[k + 1]] then
31:            runs[k] ← 1
32:            if k ≠ height − 1 ∧ div[k + 1] < div[tmp_thr] then
33:                thrs[k] ← 1
34:            else
35:                thrs[tmp_thr] ← 1
36:            push(samples_beg, tmp_beg)
37:            push(samples_end, pref[k])
38:            if push_one then
39:                if v ≠ 0 then
40:                    ones[k − 1] = 1
41:                swap(push_zero, push_one)
42:            else
43:                if u ≠ 0 then
44:                    zeros[k − 1] = 1
45:                swap(push_zero, push_one)
46:            beg_run ← ⊤
47:    if |zeros| ≠ 0 then
48:        zeros[|zeros| − 1] ← 1
49:    if |ones| ≠ 0 then
50:        ones[|ones| − 1] ← 1
51:    build rank/select for the four bitvectors
52:    return (start, c, runs, zeros, ones, samples_beg, samples_end, div)
```

Algoritmo A.5 Algoritmo per estrazione simbolo da una run in una colonna

```
1: function GET_SYMBOL(s, r)                ▷ s = ⊤ iff column start with 0, r run index
2:     if s 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
```

Algoritmo A.6 Algoritmo per uvtrick naive

```
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
7:             return (u, v)
8:         else
9:             return (v, u)
10:    else
11:        u ← uv_k[i], v ← uv_k[i − 1]
12:        if start^k then
13:            return (u, v)
14:        else
15:            return (v, u)
```

Algoritmo A.7 Algoritmo per uvtrick con bitvector

```
1: function UVTRICK(k, i)                          ▷ k is column index, i row index
2:     if i = 0 then
3:         return (0, 0)
4:     run ← rank_h^k(i)
5:     if run = 0 then
6:         if start^k then
7:             return (i, 0)
8:         else
9:             return (0, i)
10:    else if run = 1 then
11:        if start^k then
12:            return (select_h^k(run) + 1, i − (select_h^k(run) + 1))
13:        else
14:            return (i − (select_h^k(run) + 1), select_h^k(run) + 1)
15:    else
16:        if run mod 2 = 0 then
17:            pre_u ← select_u^k(run/2) + 1
18:            pre_v ← select_v^k(run/2) + 1
19:            offset ← i − (select_h^k(run) + 1)
20:            if start^k then
21:                return (pre_u + offset, pre_v)
22:            else
23:                return (pre_u, pre_v + offset)
24:        else
25:            run_u ← (run/2) + 1
26:            run_v ← run/2
27:            if ¬start^k then
28:                swap(run_u, run_v)
29:            pre_u ← select_u^k(run_u) + 1
30:            pre_v ← select_v^k(run_v) + 1
31:            offset ← i − (select_h^k(run) + 1)
32:            if start^k then
33:                return (pre_u, pre_v + offset)
34:            else
35:                return (pre_u + offset, pre_v)
```

Algoritmo A.8 Algoritmo per convertire un indice in indice di run

```
1: function INDEX_TO_RUN(k, i)                ▷ k indice di colonna, i indice di run
2:     run ← 0, found ← ⊥
3:     if i ≥ p_k[|p_k| − 1] then
4:         return |p_k| − 1
5:     for every r ∈ [0, |p_k| − 1] do
6:         if p_k[r] ≥ i < p_k[r + 1] then
7:             run ← r, found ← ⊤
8:             break
9:     if ¬found then
10:        run = |p_k| − 1
11:    return run
```

Algoritmo A.9 Algoritmo per lf-mapping

```
1: function LF(k, i, s)                       ▷ k is column index, i row index, s symbol
2:     c ← c[k]
3:     (u, v) ← uvtrick(k, i)
4:     if s = 0 then
5:         return u
6:     else
7:         return c + v
```

Algoritmo A.10 Algoritmo per lf-mapping con offset

```
1: function LF_OFF(k, i, s, o)   ▷ k is column index, i row index, s symbol, o
   offset
2:     (u, v) ← uvtrick(k, i)
3:     if p_k[i] + 0 = height then
4:         if get_symbol(start^k, i) = 0 then
5:             v ← v − 1
6:         else
7:             u ← u − 1
8:     if s = 0 then
9:         return u + o
10:    else
11:        return c[k] + v + o
```

Algoritmo A.11 Algoritmo per lf-mapping inverso naive

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

Algoritmo A.12 Algoritmo per lf-mapping inverso con bitvector

```
1:  function REVERSE_LF(k, i)                    ▷ k is column index, i row index
2:      if k = 0 then                                                  ▷ by design
3:          return 0
4:      k ← k − 1
5:      c ← rlpbwt[k].c
6:      if i < c then
7:          if start^k then
8:              run ← rank_u^k(i) · 2
9:          else
10:             run ← rank_u^k(i) · 2 + 1
11:         i_run ← 0
12:         if run ≠ 0 then
13:             i_run ← select_h^k(run) + 1
14:         (prev_0, _) ← uvtrick(k, i_run)
15:         return i_run + (i − prev_0)
16:     else
17:         if start^k then
18:             run ← rank_v^k(i) · 2 + 1
19:         else
20:             run ← rank_v^k(i) · 2
21:         i_run ← 0
22:         if run ≠ 0 then
23:             i_run ← select_h^k(run) + 1
24:         (_, prev_1) ← uvtrick(k, i_run)
25:         return i_run + (i − (c + prev_1))
```

Algoritmo A.13 Algoritmo per match con aplotipo esterno con panel *width* × *height* naive

```
1: function EXTERNAL_MATCHES(z)                          ▷ assuming |z| = rlpbwt.width
2:     f ← 0, f_run ← 0, f' ← 0, e ← 0, l ← 0
3:     g ← 0, g_run ← 0, g' ← 0, f_off ← 0, g_off ← 0
4:     for every k ∈ [0, |z|) do
5:         f_run ← index_to_run(f, k), g_run ← index_to_run(g, k)
6:         f_off ← f − p_k[f_run], f_off ← g − p_k[g_run]
7:         if z[k] = 0 then
8:             if get_symbol(start^k, f_run) = 1 then f_off ← 0
9:             if get_symbol(start^k, g_run) = 1 then g_off ← 0
10:        else
11:            if get_symbol(start^k, f_run) = 0 then f_off ← 0
12:            if get_symbol(start^k, g_run) = 0 then g_off ← 0
13:        f' ← lf_off(k, f, z[k], f_off), g' ← lf(k, g, z[k], g_off), l ← g − f
14:        if f' > height then f' ← f' − f_off
15:        if g' > height then g' ← g' − g_off
16:        if f' < g' then
17:            f ← f', g ← g'
18:        else
19:            if k ≠ 0 then
20:                report matches in [e, k − 1] with l haplotypes
21:            if f' = |lcp^{k+1}| then e ← k + 1 else e ← k − lcp^{k+1}[f']
22:            if (z[e] = 0 ∧ f' > 0) ∨ f' = height then
23:                f' ← g' − 1
24:                if e ≥ 1 then
25:                    f_rev ← f', k' ← k + 1
26:                    while k' ≠ e − 1 do f_rev ← reverse_lf(k', f_rev), k' ← k' − 1
27:                    run ← index_to_run(f_rev, k'), symb ← get_symbol(start^{k'}, run)
28:                    while e > 0 ∧ z[e − 1] = symb do
29:                        f_rev ← reverse_lf(e, f_rev)
30:                        run ← index_to_run(f_rev, e − 1), symb ← get_symbol(start^{e−1}, run)
31:                while f' > 0 ∧ (k + 1) − lcp^{k+1}[f] ≤ e do e ← e − 1
32:                f ← f', g ← g'
33:            else
34:                g' ← f' − 1
35:                if e ≥ 1 then
36:                    f_rev ← f', k' ← k + 1
37:                    while k' ≠ e − 1 do f_rev ← reverse_lf(k', f_rev), k' ← k' − 1
38:                    run ← index_to_run(f_rev, k'), symb ← get_symbol(start^{k'}, run)
39:                    while e > 0 ∧ z[e − 1] = symb do
40:                        f_rev ← reverse_lf(e, f_rev)
41:                        run ← index_to_run(f_rev, e − 1), symb ← get_symbol(start^{e−1}, run)
42:                while e < height ∧ (k + 1) − lcp^{k+1}[e] ≤ e do e ← e + 1
43:                f ← f', g ← g'
44:    if f < g then
45:        report matches in [e, |z| − 1] with l ← g − f haplotypes
```

Algoritmo A.14 Algoritmo per match con aplotipo esterno con panel *width* × *height* con bitvectors

```
1: function EXTERNAL_MATCHES(z)                          ▷ assuming |z| = rlpbwt.width
2:     f ← 0, f_run ← 0, f' ← 0
3:     g ← 0, g_run ← 0, g' ← 0
4:     e ← 0, l ← 0
5:     for every k ∈ [0, |z|) do
6:         f_run ← rank_h^k(f), g_run ← rank_h^k(g)
7:         f' ← lf(k, f, z[k]), g' ← lf(k, g, z[k])
8:         l ← g − f
9:         if f' < g' then
10:            f ← f', g ← g'
11:        else
12:            if k ≠ 0 then
13:                report matches in [e, k − 1] with l haplotypes
14:            if f' = |lcp^{k+1}| then
15:                e ← k + 1
16:            else
17:                e ← k − lcp^{k+1}[f']
18:            if (z[e] = 0 ∧ f' > 0) ∨ f' = height then
19:                f' ← g' − 1
20:                if e ≥ 1 then
21:                    f_rev ← f', k' ← k + 1
22:                    while k' ≠ e − 1 do
23:                        f_rev ← reverse_lf(k', f_rev), k' ← k' − 1
24:                    run ← rank_h^{k'}(f_rev), symb ← get_symbol(start^{k'}, run)
25:                    while e > 0 ∧ z[e − 1] = symb do
26:                        f_rev ← reverse_lf(e, f_rev)
27:                        run ← rank_h^{e−1}(f_rev)
28:                        symb ← get_symbol(start^{e−1}, run)
29:                while f' > 0 ∧ (k + 1) − lcp^{k+1}[f] ≤ e do e ← e − 1
30:                f ← f', g ← g'
31:            else
32:                g' ← f' − 1
33:                if e ≥ 1 then
34:                    f_rev ← f', k' ← k + 1
35:                    while k' ≠ e − 1 do
36:                        f_rev ← reverse_lf(k', f_rev), k' ← k' − 1
37:                    run ← rank_h^{k'}(f_rev), symb ← get_symbol(start^{k'}, run)
38:                    while e > 0 ∧ z[e − 1] = symb do
39:                        f_rev ← reverse_lf(e, f_rev)
40:                        run ← rank_h^{e−1}(f_rev)
41:                        symb ← get_symbol(start^{e−1}, run)
42:                while e < height ∧ (k + 1) − lcp^{k+1}[e] ≤ e do e ← e + 1
43:                f ← f', g ← g'
44:    if f < g then
45:        report matches in [e, |z| − 1] with l ← g − f haplotypes
```

Algoritmo A.15 Algoritmo per match con matching-statistics (MS) e thresholds

function MATCHES_MS(z)
  $ms_{row} \leftarrow [0..0],\ ms_{len} \leftarrow [0..0]$ ▷ ms vectors with row and len of length $|z|$
  $curr_{row} \leftarrow rlpbwt[0].samples_{end}[|rlpbwt[0].samples_{end}| - 1]$
  $curr_{index} \leftarrow curr_{row}$
  $curr_{run} \leftarrow rank_h^0(curr_{index})$
  $symb \leftarrow get\_symbol(start^0, curr_{run})$ ▷ **build matching statistics row**
  for *every* $k \in [0, |z|)$ **do**
    if $z[i] = symb$ **then**
      $ms_{row}[k] \leftarrow curr_{row}$
      if $k \neq |z| - 1$ **then**
        $(curr_{index},\ curr_{run},\ symb) \leftarrow UPDATE(k, curr_{index}, z)$
    else
      $curr_{thr} \leftarrow rank_t^k(curr_{index})$
      $force_{down} \leftarrow \top$ *iff we are over a threshold not at the end of a run*
      $force_{down} \leftarrow \top$ *iff we are over a threshold at the end of a run and DOWN function is* $\top$
      if $|samples_{beg}^k| = 1$ **then**
        $ms_{row}[k] \leftarrow height$
        if $k \neq |z| - 1$ **then**
          $curr_{row} \leftarrow rlpbwt[k+1].samples_{end}[|rlpbwt[k+1].samples_{end}| - 1]$
          $curr_{index} \leftarrow height - 1$
          $curr_{run} \leftarrow rank_h^{k+1}(curr_{index})$
          $symb \leftarrow get\_symbol(start^{k+1}, curr_{run})$
      else if $(curr_{run} \neq 0 \wedge curr_{run} = curr_{thr} \wedge \neg down) \vee curr_{run} = |samples_{beg}^k| - 1$ **then**
        $curr_{index} \leftarrow select_h^k(curr_{run})$
        $curr_{row} \leftarrow samples_{end}^k[curr_{run} - 1]$
        $ms_{row}[k] \leftarrow curr_{row}$
        if $k \neq |z| - 1$ **then**
          $(curr_{index},\ curr_{run},\ symb) \leftarrow UPDATE(k, curr_{index}, z)$
      else
        $curr_{index} \leftarrow select_h^k(curr_{run} + 1) + 1$
        $curr_{row} \leftarrow samples_{beg}^k[curr_{run} + 1]$
        $ms_{row}[k] \leftarrow curr_{row}$
        if $k \neq |z| - 1$ **then**
          $(curr_{index},\ curr_{run},\ symb) \leftarrow UPDATE(k, curr_{index}, z)$
  ▷ **build matching statistics len**
  for *every* $k \in [0, |ms_{row}|)$ **do**
    if $ms_{row}[k] = height$ **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 is a data structure for random access over the orginale panel
      $tmp_{index} \leftarrow i,\ tmp_{len} \leftarrow 0$
      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* $k \in [0, |ms_{row}|)$ **do** ▷ **build matching statistics matches**
    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 match ending in k, with length $ms_{len}[k]$*, with at least row* $ms_{row}[k]$
  in case extend the matches

function DOWN($pos, prev, next$)
  using LCE queries or random access check the longest common prefix between pos and prev and between pos and next
  if the latter is greater or equal return $\top$*, else* $\bot$

Algoritmo A.16 Algoritmo per match con matching-statistics (MS) e LCE

function MATCHES_MS_LCE(z)
  $ms_{row} \leftarrow [0..0],\ ms_{len} \leftarrow [0..0]$ ▷ ms vectors with row and len of length $|z|$
  $curr_{row} \leftarrow rlpbwt[0].samples_{end}[|rlpbwt[0].samples_{end}| - 1]$
  $curr_{index} \leftarrow curr_{row},\ curr_{run} \leftarrow rank_h^0(curr_{index})$
  $symb \leftarrow get\_symbol(start^0, curr_{run})$ ▷ **build matching statistics row**
  for *every* $k \in [0, |z|)$ **do**
    if $z[i] = symb$ **then**
      $ms_{row}[k] \leftarrow curr_{row}$
      if $k = 0$ **then**
        $ms_{len}[k] \leftarrow 1$
      else
        $ms_{len}[k] \leftarrow ms_{len}[k-1] + 1$
      if $k \neq |z| - 1$ **then**
        $(curr_{index},\ curr_{run},\ symb) \leftarrow UPDATE(k, curr_{index}, z)$
    else
      if $|samples_{beg}^k| = 1$ **then**
        $ms_{row}[k] \leftarrow height$
        $ms_{len}[k] \leftarrow 0$
        if $k \neq |z| - 1$ **then**
          $curr_{row} \leftarrow rlpbwt[k+1].samples_{end}[|rlpbwt[k+1].samples_{end}| - 1]$
          $curr_{index} \leftarrow height - 1$
          $curr_{run} \leftarrow rank_h^{k+1}(curr_{index})$
          $symb \leftarrow get\_symbol(start^{k+1}, curr_{run})$
      else
        if $curr_{run} = |samples_{beg}^k| - 1$ **then**
          $curr_{index} \leftarrow select_h^k(curr_{run}),\ prev_{row} \leftarrow samples_{end}^k[curr_{run} - 1]$
          $lce \leftarrow LCE(k, curr_{row}, prev_{row})$
          $ms_{row}[k] \leftarrow prev_{row},\ curr_{row} \leftarrow prev_{row}$
          if $k = 0$ **then**
            $ms_{len}[k] \leftarrow 1$
          else
            $ms_{len}[k] \leftarrow min(ms_{len}[k-1], lce_{len}) + 1$
          if $k \neq |z| - 1$ **then**
            $(curr_{index},\ curr_{run},\ symb) \leftarrow UPDATE(k, curr_{index}, z)$
        else if $curr_{run} = 0$ **then**
          $curr_{index} \leftarrow select_h^k(curr_{run} + 1) + 1,\ next_{row} \leftarrow samples_{beg}^k[curr_{run} + 1]$
          $lce \leftarrow LCE(k, curr_{row}, next_{row})$
          $ms_{row}[k] \leftarrow next_{row},\ curr_{row} \leftarrow next_{row}$
          if $k = 0$ **then**
            $ms_{len}[k] \leftarrow 1$
          else
            $ms_{len}[k] \leftarrow min(ms_{len}[k-1], lce_{len}) + 1$
          if $k \neq |z| - 1$ **then**
            $(curr_{index},\ curr_{run},\ symb) \leftarrow UPDATE(k, curr_{index}, z)$
        else
          $prev_{row} \leftarrow samples_{end}^k[curr_{run} - 1],\ next_{row} \leftarrow samples_{beg}^k[curr_{run} + 1]$
          $lce \leftarrow \max_{len}(LCE(k, curr_{row}, prev_{row}), LCE(k, curr_{row}, next_{row}))$
          $curr_{row} \leftarrow lce_{row}$
          $ms_{row}[k] \leftarrow curr_{row}$
          if $k = 0$ **then**
            $ms_{len}[k] \leftarrow 1$
          else
            $ms_{len}[k] \leftarrow min(ms_{len}[k-1], lce_{len}) + 1$
          if $k \neq |z| - 1$ **then**
            $(curr_{index},\ curr_{run},\ symb) \leftarrow UPDATE(k, curr_{index}, z)$
  for *every* $k \in [0, |ms_{row}|)$ **do** ▷ **build matching statistics matches**
    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 match ending in k, with length $ms_{len}[k]$*, with at least row* $ms_{row}[k]$
  in case extend the matches

Algoritmo A.17 Algoritmo per l'update usando le matching statistics

1: **function** UPDATE($k, curr_{index}, z$)
2: $curr_{index} \leftarrow lf(k, curr_{index}, z[k])$
3: $curr_{run} \leftarrow rank_h^{k+1}(curr_{index})$
4: $symb \leftarrow get\_symbol(start^{k+1}, curr_{run})$
5: **return** $(curr_{index},\ curr_{run},\ symb)$