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**, 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*, introdotta qui insieme alla versione classica della trasformata, che per la mia variante relativa alla *Position Burrows-Wheeler Transform*, che verrà anch'essa trattata nel capitolo.

2.1 Motivazioni Biologiche

2.2 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 in *SDSL* [1], delle principali varianti è visualizzabile in tabella A.1. Il secondo vantaggio consiste nel fatto che i *bit vector* permettono l'implementazione efficiente di due funzioni:

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

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

2.2.1 Funzione rank

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

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

Come detto, da un punto di vista teorico, al costo di $\mathcal{O}(n)$ bit aggiuntivi in memoria tale funzione sarebbe supportata in tempo $\mathcal{O}(1)$. Questo però non risulta vero nelle principali implementazioni. La complessità temporale varia infatti a seconda dell'implementazione, anche in conseguenza al fatto che si ha una quantità diversa di bit aggiuntivi salvati in memoria. La tabella con le complessità temporali stimate della *funzione rank*, per le varianti di *bit vector* implementate in *SDSL*, è visualizzabile in tabella A.2.

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 A.3.
Si può quindi vedere un semplice esempio esplicativo.

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

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

Si ha che, per esempio:

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

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

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

2.3 Straight-Line Program

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

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

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

dove:

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

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

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

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

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

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

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

Esempio 2. *Si prenda la seguente stringa:*

$$s = GATTAGATACAT\$GATTACATAGAT$$

Si potrebbe produrre il seguente SLP*:*

$$S \to ZWAY\$ZYAW$$

$$Z \to WX$$

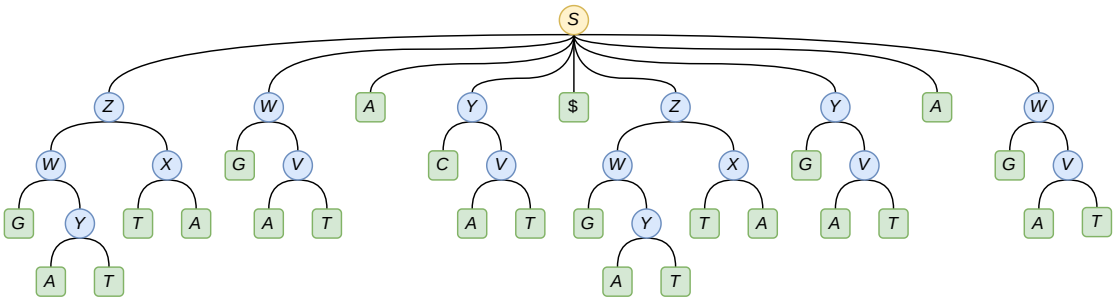
$$Y \to CV$$

$$X \to TA$$

$$W \to GV$$

$$V \to AT$$

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



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

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

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

2.3.1 Longest Common Extension

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

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

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

lunghezza l è effettuabile in tempo:

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

VERIFICARE BENE COSA SIA n.

2.4 Suffix Array

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

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

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

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

Si vede quindi esempio chiarificatore.

Esempio 3. *Si prenda la stringa:*

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

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:

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

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].

2.5 Trasformata di Burrows-Wheeler

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

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

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

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

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

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

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

Esempio 5. *Si prenda la stringa:*

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

Si produce la seguente matrice delle rotazioni riordinate:

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

Avendo quindi:

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

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 8. *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 6. *Si riprende l'esempio precedente, avendo:*

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

2.5.1 Trasformata di Burrows-Wheeler run-length

2.5.2 Matching Statistics

2.5.3 R-index

2.5.4 PHONI

2.6 Trasformata di Burrows-Wheeler posizionale

Presentata nel 2014 da Richard Durbin la **Positional Burrows-Wheeler Transform (*PBWT*)**, traducibile con *trasformata di Burrows-Wheeler posizionale*, è

una struttura efficiente per la memorizzazione e l'interrogazione di pannelli di aplotipi.
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 9. *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 10. *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 11. *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 B.2.
Ai fini della trattazione dell'algoritmo di match con un'aplotipo esterno ricordiamo un'ulteriore definizione.

Definizione 12. *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 7. *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 B.2 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 B.1:

$$w_k : \{0, \dots, N\} \times \Sigma \to \{0, \dots, N\}$$

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 8. *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 9. *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:

$$y_{f_{k+1}-1}^{k+1} < z < y_{f_{k+1}}^{k+1}$$

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

METTERE ESEMPIO
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 [7], 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 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

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'
```

- 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 M

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

PBWT multi-allelica

PBWT con struttura LEAP

PBWT dinamica

PBWT bidirezionale

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

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, dopo una breve introduzione agli strumenti computazionali usati, si approfondiranno tutte le varianti della **RLPBWT** ottenute durante lo studio, evidenziandone pro e contro

3.1 Introduzione agli strumenti usati

Prima di addentrarci negli aspetti più teorici 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
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** [8]. 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

Un esempio è visualizzabile alla listing C.1.

3.1.1 MaCS

RIVEDERE!
MaCS [9], 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 [10], 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 C.2.
MANCA SIGNIFICATO DEGLI "HEADER" DI OGNI SITE NEL RISULTATO.

3.1.2 SDSL

La libreria più utilizzata nel progetto, come anche in diverse sue dipendenze, è **Succinct Data Structure Library (*SDSL*)** [1]. Questa libreria, scritta in C++11, fornisce diverse implementazioni riguardanti strutture dati succinte, come i già citati, nella sezione 2.2, **bitvectors**.
Nel dettaglio, in questo progetto, *SDSL* è stata usata per:

- gli **sparse bitvectors**, il cui uso specifico verrà specificato più avanti nel capitolo
- i cosiddetti **int vectors**, ovvero vettori di interi memorizzati in modo efficiente
- le funzioni di atte a gestire le **serializzazioni** delle strutture dati implementate
- stimare, in certi casi, lo **spazio in memoria** richiesto per le varie strutture

3.1.3 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** [11], implementazione dei risultati ottenuti da Gagie et al. [3]. Inoltre, tale libreria basa il suo funzionamento sull'uso di un'altra libreria, detta **BigRePair** [12], che implementa i quanto studiato da Gagie et al. [13] 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)

Compressione del panel

I tool appena citati assumono un input "monodimensionale", ovvero una singola sequenze lineare. Nel nostro caso l'input era invece un file `.macs` con rappresentato il pannello trasposto. Nel dettaglio, assumendo di avere il pannello come nella listing C.2 (pannello al quale sono giù state estratte le query), si avrebbe, isolando:

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

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:

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

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). Ad esempio, prendendo la seconda e la terza riga, avremmo una *LCE query* lunga 3, terminante nella prima colonna esclusa:

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

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 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".

3.1.4 Snakemake

PARTE DA FARE UNA VOLTA STABILITA LA PIPELINE FINALE

3.2 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.

3.2.1 Perché un'implementazione run-length

Prima di proseguire con la spiegazione dettagliata delle varianti 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 [14], 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 ottengono 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.

3.2.2 Una prima proposta

3.3 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.
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 10. *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 B.5.
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 11. *Sia data la seguente colonna:*

$$y^5 = 00101111000000000000$$

Per la RLPBWT naive si hanno in memoria:

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

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

$$c[5] = 15$$

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

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.*

3.4 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 12. *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 B.3 (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 13. *Si prendano i dati e i risultati ottenuti all'esempio 12. 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 B.6.

3.5 Mapping nella RLPBWT

3.6 Algoritmo per match massimali

3.7 RLPBWT con pannello denso

3.7.1 Algoritmo con matching statistics

3.8 RLPBWT con SLP

3.8.1 Algoritmo con matching statistics

3.9 Funzione Phi

3.9.1 Costruzione della struttura di supporto

3.9.2 Estensione dei match

Capitolo 4

Risultati

4.1 Ambiente di benchmark

4.1.1 Descrizione input

4.2 Analisi della complessità

4.2.1 Analisi temporale

4.2.2 Analisi spaziale

Capitolo 5

Conclusioni

5.1 Sviluppi futuri

5.1.1 K-mems

5.1.2 RLPBWT multi-allelica

5.1.3 RLPBWT con dati mancanti

Bibliografia e sitografia

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

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

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

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

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

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

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

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

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

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

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

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

[13] 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.

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

Appendice A

Tabelle

Tabella A.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	$64\left\lceil \frac{n}{64} + 1 \right\rceil$
Interleaved bitvector	$\approx n\left(1 + \frac{64}{K}\right)$
H_0-compressed bitvector	$\approx \left\lceil \log \binom{n}{m} \right\rceil$
Sparse bitvector	$\approx m\left(2 + \log \frac{n}{m}\right)$

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

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

Tabella A.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	$\mathcal{O}(1)$

Tabella A.4: Tabella stima complessità temporale delle principali operazioni effettuate nelle varianti della RLPBWT.

Operazione	Complessità temporale
Rank per sparse bitvector	$\mathcal{O}(\frac{height}{\lvert run \rvert})$
Select per sparse bitvector	$\mathcal{O}(1)$
Random Access su slp	$\mathcal{O}(\log(height \times width))$
LCE lunga l su slp	$\mathcal{O}\left(1 + \frac{l}{\log(height \times width)}\right)$

Appendice B

Pseudocodici

Algoritmo B.1 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 B.2 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 B.3 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 B.4 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 B.5 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 B.6 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 B.7 Algoritmo per lf-mapping

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

Algoritmo B.8 Algoritmo per lf-mapping inverso

```
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 B.9 Algoritmo per match con aplotipo esterno con panel *width* × *height*

```
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 ← 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:        l ← g − f
46:        report matches in [e, |z| − 1] with l haplotypes
```

Algoritmo B.10 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 B.11 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 B.12 Algoritmo per l'update usando le matching statistics

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

Algoritmo B.13 Algoritmo per la costruzione della struttura per φ e φ^{-1}

```
1: function BUILD_PHI(cols, panel, prefix)        ▷ prefix is the last prefix array
2:     φ ← [[0..0]..[0..0]], φ^{-1} ← [[0..0]..[0..0]]   ▷ sparse bit vector panels for φ and φ^{-1}
3:     φ_supp = [], φ^{-1}_supp = []                ▷ vectors for φ and φ^{-1} row values
4:     for every k ∈ [0, |cols|) do
5:         for every i ∈ [0, |samples_beg|) do
6:             φ[sample^k_beg[i]][k] ← 1
7:             if i = 0 then
8:                 push(φ_supp[sample^k_beg[i]], panel_height)
9:             else
10:                push(φ_supp[sample^k_beg[i]], sample^k_end[i − 1])
11:            φ^{-1}[sample^k_end[i]][k] ← 1
12:            if i = |sample^k_beg| − 1 then
13:                push(φ^{-1}_supp[sample^k_end[i]], panel_height)
14:            else
15:                push(φ^{-1}_supp[sample^k_end[i]], sample^k_beg[i + 1])
16:    for every k ∈ [0, |prefix|) do
17:        if φ[k][|φ[k]| − 1] = 0 then
18:            φ[k][|φ[k]| − 1] ← 1
19:            if k = 0 then
20:                push(φ_supp[prefix^k], panel_height)
21:            else
22:                push(φ_supp[prefix^k], prefix^k[i − 1])
23:        if φ^{-1}[k][|φ[k]| − 1] = 0 then
24:            φ^{-1}[k][|φ[k]| − 1] ← 1
25:            if k = |prefix| − 1 then
26:                push(φ^{-1}_supp[prefix^k], panel_height)
27:            else
28:                push(φ^{-1}_supp[prefix^k], prefix^k[i + 1])
29:    build rank/select for every sparse bitvector in φ and φ^{-1}
```

Algoritmo B.14 Algoritmi per le query a φ e φ^{-1}

function $\varphi(prefix_{value}, col)$
    $res \leftarrow \varphi_{supp}^{prefix_{value}}[rank_{\varphi}^{prefix_{value}}(col)]$
    if $res = panel_{height}$ **then**
        return *null*
    else
        return res

function $\varphi^{-1}(prefix_{value}, col)$
    $res \leftarrow \varphi_{supp}^{-1\ prefix_{value}}[rank_{\varphi^{-1}}^{prefix_{value}}(col)]$
    if $res = panel_{height}$ **then**
        return *null*
    else
        return res

Algoritmo B.15 Algoritmo per estendere un match in *col* usando φ, φ^{-1} e MS

function EXTEND_MATCHES(col, row, len)
    $check_{down} \leftarrow \top,\ check_{up} \leftarrow \top$
    while $check_{down}$ **do**
        $down_{row} \leftarrow \varphi^{-1}(row, col)$
        if $lce\_bounded(col, row, down_{row}, len)$ **then**
            $push(haplos, down_{row})$
            $row \leftarrow down_{row}$
        else
            $check_{down} \leftarrow \bot$
    while up_{down} **do**
        $up_{row} \leftarrow \varphi(row, col)$
        if $lce\_bounded(col, row, up_{row}, len)$ **then**
            $push(haplos, up_{row})$
            $row \leftarrow up_{row}$
        else
            $check_{up} \leftarrow \bot$
    return *haplos*

Appendice C

Esempi di File

Listing C.1 Esempio del formato file in output dopo il calcolo dei match con l'implementazione della **PBWT** di Durbin.

```
MATCH   0       64435   0       16138   16138
MATCH   0       68611   0       16138   16138
MATCH   0       12520   16136   16351   215
MATCH   0       3578    16136   16351   215
MATCH   0       4042    16136   16351   215
MATCH   0       5446    16136   16351   215
MATCH   0       29111   16136   16351   215
MATCH   0       56327   16136   16351   215
MATCH   0       42859   16136   16351   215
MATCH   0       38750   16136   16351   215
MATCH   0       42872   16136   16351   215
MATCH   0       33743   16136   16351   215
MATCH   0       46913   16136   16351   215
MATCH   0       497     16136   16351   215
MATCH   0       49708   16138   46537   30399
MATCH   1       26103   0       12800   12800
MATCH   1       14003   1671    16731   15060
MATCH   1       66121   7873    29338   21465
MATCH   1       48305   15585   31210   15625
MATCH   1       60588   15585   31210   15625
```

Listing C.2 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
```