RLPBWT

Davide Cozzi

Dipartimento di Informatica, Sistemistica e Comunicazione (DISCo)
Università degli Studi di Milano Bicocca

Outline

Outline

Some definitions

The permutation, panel M, $n \times m$

In *RLPBWT* we have a permutation π_j, $\forall\ 1 \leq j \leq m$ that stably sorts the bits of the j-th column of the PBWT.

This permutation can be stored in space proportional to the number of runs in the j-th column of the PBWT

Some definitions

The permutation, panel M, $n \times m$

In *RLPBWT* we have a permutation π_j, $\forall\ 1 \leq j \leq m$ that stably sorts the bits of the j-th column of the PBWT.
This permutation can be stored in space proportional to the number of runs in the j-th column of the PBWT

The positions in the columns of the PBWT of the bits in the i-th row of M are:

Some definitions

The permutation, panel M, $n \times m$

In *RLPBWT* we have a permutation π_j, $\forall\ 1 \leq j \leq m$ that stably sorts the bits of the j-th column of the PBWT.
This permutation can be stored in space proportional to the number of runs in the j-th column of the PBWT

The positions in the columns of the PBWT of the bits in the i-th row of M are:

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

Extracting the bits of the i-th row of M reduces to iteratively applying the π_{m-1} permutations, corresponding to iteratively apply LF in a standard BWT

The permutation

Computing the permutation

$$\pi_j(p) = \begin{cases} p - count_1 & \text{if } column[pref[p]] = 0 \\ count_0 + count_1 - 1 & \text{if } column[pref[p]] = 1 \end{cases}$$

- $count_0$: total number of zeros in the PBWT column
- $count_1$: number of ones in the PBWT column as far as index p

"LF-mapping" in Durbin's algorithm

$$w(i, \sigma) = \begin{cases} u[i] & \text{if } \sigma = 0 \\ c + v[i] & \text{if } \sigma = 1 \end{cases}$$

- c: total number of zeros in the column
- $u[i]$: number of zeros in the column as far as index i
- $v[i]$: number of ones in the column as far as index i

Travis's example

	1	2	3	4	5	6	7	8	9	10	11	12
0	0 1	0 1	0 0	0 0	0 0	0 1	0 0	0 0	0 1	0 1	0 1	0 1
1	0 1	0 1	0 0	0 0	0 0	0 1	0 0	0 0	0 1	0 1	0 **1**	0 1
2	0 1	0 1	0 1	0 0	0 0	0 0	0 1	0 1	0 1	0 0	0 1	0 1
3	0 1	0 1	0 0	0 **0**	0 **0**	0 **1**	0 0	0 0	0 1	0 1	0 0	0 1
4	0 1	0 0	0 1	0 0	0 0	0 1	0 0	0 0	0 1	0 1	0 0	0 1
5	0 1	0 0	0 1	0 0	0 0	0 0	0 0	0 0	0 1	0 **0**	0 0	0 1
6	0 1	0 0	0 1	0 0	0 0	0 0	0 0	0 0	0 1	0 0	0 0	0 0
7	0 1	0 1	0 1	0 0	0 0	0 0	0 0	0 0	0 0	1 1	0 0	0 1
8	0 0	0 1	0 **0**	0 0	0 0	0 1	0 0	0 0	0 0	1 1	0 0	0 1
9	0 **1**	1 0	0 0	0 0	0 0	0 1	0 0	0 0	0 0	1 0	0 0	0 1
10	0 1	1 1	0 0	0 0	0 0	0 0	0 0	0 0	0 1	1 1	0 0	0 1
11	0 0	1 1	1 0	0 1	0 1	0 0	0 0	0 0	0 1	1 0	0 0	0 1
12	0 0	1 1	1 0	0 0	0 1	0 0	0 0	0 0	0 0	1 0	0 0	0 1
13	0 0	1 0	1 0	0 0	0 1	0 0	0 0	0 0	0 0	1 0	1 0	0 1
14	0 0	1 0	1 0	0 0	0 0	0 0	1 0	0 0	0 **0**	1 0	1 0	0 1
15	0 0	1 0	1 0	1 0	0 0	0 0	1 0	0 **0**	0 0	1 0	1 0	0 1
16	0 1	1 0	1 0	1 0	0 0	0 0	1 **0**	0 0	1 1	1 0	1 0	0 1
17	0 0	1 **0**	1 0	1 0	0 0	1 0	1 0	0 1	1 1	1 0	1 0	1 1
18	1 0	1 0	1 0	1 0	0 0	1 0	1 0	0 1	1 1	1 0	1 0	1 **1**
19	1 0	1 0	1 0	1 0	1 0	1 0	1 0	1 1	1 1	1 0	1 0	1 1

The compressed data structure

The tables

- a set of m tables in which the m-th table stores only the positions of the run-heads in the m-th column and a bool to check the first symbol: 0 or 1
- the i-th row of the j-th table stores a quadruple

The compressed data structure

The tables

- a set of m tables in which the m-th table stores only the positions of the run-heads in the m-th column and a bool to check the first symbol: 0 or 1
- the i-th row of the j-th table stores a quadruple

The quadruple

1. the position p of the i-th run-head in the j-th column of the PBWT
2. the permutation $\pi_j(p)$
3. the index of the run containing bit $\pi_j(p)$ in the $(j+1)$-st column of the PBWT
4. the threshold, that's the index of the minimum *LCP value* (current column minus divergence array value) in the run

Row extraction

First step

We start by finding the row of the first table that starts with the position p of the head of the run containing bit i in first column of the PBWT, computing:

Row extraction

First step

We start by finding the row of the first table that starts with the position p of the head of the run containing bit i in first column of the PBWT, computing:

$$\pi_1(i) = \pi_1(p) + i - p$$

Row extraction

First step

We start by finding the row of the first table that starts with the position p of the head of the run containing bit i in first column of the PBWT, computing:

$$\pi_1(i) = \pi_1(p) + i - p$$

looking up the row for the run containing bit $\pi_1(p)$ in the the second table and scanning down the table until we find the row for the run containing bit $\pi_1(i)$

Next step

We continue repeating this procedure for each column

Travis's example I

	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				

Travis's example II

Extraction of row 9, $\pi_j(i) = \pi_j(p) + i - p$

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

Outline

The matrixes

Panel and query

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

PBWT Matrix

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

Prefix and Divergence Arrays

Prefix Arrays

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

LCP Arrays: current k minus the original Durbin's divergence arrays

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

Run-Length PBWT I, *p*, *perm*, *next perm*, *threshold*

$[0, 1, 2, 3, 4]$

$$\begin{matrix} 0 & 4 & 4 & 0 \\ 1 & 0 & 0 & 1 \\ 3 & 5 & 5 & 3 \\ 4 & 2 & 2 & 4 \\ 5 & 6 & 6 & 5 \\ 6 & 3 & 3 & 6 \end{matrix} \Longrightarrow \begin{matrix} 0 & 3 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 2 & 4 & 1 & 2 \\ 3 & 1 & 0 & 3 \\ 4 & 5 & 2 & 4 \\ 5 & 2 & 0 & 5 \\ 6 & 6 & 2 & 6 \end{matrix} \Longrightarrow \begin{matrix} 0 & 0 & 0 & 0 \\ 4 & 6 & 3 & 4 \\ 5 & 4 & 2 & 5 \end{matrix} \Longrightarrow \begin{matrix} 0 & 3 & 2 & 0 \\ 3 & 0 & 0 & 3 \\ 4 & 6 & 4 & 4 \\ 5 & 1 & 1 & 6 \end{matrix} \Longrightarrow \begin{matrix} 0 & 0 & 0 & 0 \\ 1 & 4 & 2 & 2 \\ 3 & 1 & 0 & 3 \\ 5 & 6 & 4 & 5 \\ 6 & 3 & 2 & 6 \end{matrix}$$

$[5, 6, 7, 8, 9]$

$$\begin{matrix} 0 & 0 & 0 & 0 \\ 2 & 5 & 2 & 2 \\ 3 & 2 & 2 & 4 \\ 5 & 6 & 2 & 5 \\ 6 & 4 & 2 & 6 \end{matrix} \Longrightarrow \begin{matrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 2 & 2 & 1 & 5 \end{matrix} \Longrightarrow \begin{matrix} 0 & 0 & 0 & 0 \\ 1 & 3 & 1 & 1 \\ 3 & 1 & 1 & 3 \\ 5 & 5 & 1 & 5 \end{matrix} \Longrightarrow \begin{matrix} 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 3 \end{matrix} \Longrightarrow \begin{matrix} 0 & 3 & 2 & 0 \\ 1 & 0 & 0 & 1 \\ 3 & 4 & 2 & 3 \\ 6 & 2 & 1 & 6 \end{matrix}$$

Run-Length PBWT II

[10, 11, 12, 13, 14]

$$\begin{array}{cccc} 0 & 2 & 2 & 0 \\ 1 & 0 & 0 & 2 \\ 3 & 3 & 3 & 3 \end{array} \Longrightarrow \begin{array}{cccc} 0 & 0 & 0 & 0 \\ 1 & 4 & 1 & 1 \\ 2 & 1 & 0 & 2 \\ 3 & 5 & 2 & 3 \\ 4 & 2 & 1 & 4 \\ 6 & 6 & 3 & 6 \end{array} \Longrightarrow \begin{array}{cccc} 0 & 4 & 2 & 0 \\ 2 & 0 & 0 & 4 \\ 5 & 6 & 3 & 5 \\ 6 & 3 & 1 & 6 \end{array} \Longrightarrow \begin{array}{cccc} 0 & 4 & 2 & 0 \\ 2 & 0 & 0 & 2 \\ 4 & 6 & 4 & 4 \\ 5 & 2 & 1 & 6 \end{array} \Longrightarrow \begin{array}{cccc} 0 & 3 & 1 & 0 \\ 2 & 0 & 0 & 2 \\ 4 & 5 & 3 & 4 \\ 5 & 2 & 0 & 5 \\ 6 & 6 & 4 & 6 \end{array}$$

[15, 16, 17, 18, 19]

$$\begin{array}{cccc} 0 & 0 & 0 & 0 \\ 3 & 5 & 2 & 3 \\ 4 & 3 & 1 & 4 \\ 5 & 6 & 3 & 5 \\ 6 & 4 & 1 & 6 \end{array} \Longrightarrow \begin{array}{cccc} 0 & 3 & 1 & 0 \\ 3 & 0 & 0 & 3 \\ 5 & 6 & 3 & 5 \\ 6 & 2 & 0 & 6 \end{array} \Longrightarrow \begin{array}{cccc} 0 & 0 & 0 & 0 \\ 3 & 5 & 2 & 3 \\ 4 & 3 & 0 & 5 \\ 6 & 6 & 3 & 6 \end{array} \Longrightarrow \begin{array}{cccc} 0 & 2 & 2 & 0 \\ 4 & 0 & 0 & 4 \\ 5 & 6 & 4 & 5 \\ 6 & 1 & 1 & 6 \end{array} \Longrightarrow \begin{array}{cccc} 0 & 4 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 2 & 5 & 0 & 2 \\ 3 & 1 & 0 & 5 \\ 6 & 6 & 0 & 6 \end{array}$$

Match with external haplotype I

First case, bits matches at column j-th

- we are looking at d-th bit of the k-th run, that come from the i-th row of the panel
- if this bit match the next bit of the pattern we can go to column $j+1$ and we figure out which bit to look at in that column
- the next bit we look at is still from row i-th

Match with external haplotype II

Second case, bits doesn't matches at column j-th

- we are looking at d-th bit of the k-th run and that bit doesn't match the next bit in the pattern
- we look at the threshold for the k-th run:
  - if d is at most the threshold (check this "at most") than we move to the last bit of the $(k-1)$-st run in the j-th column and then we proceed as in *case 1*
  - if d is greater than the threshold than we move to the first bit of the $(k-1)$-st run in the j-th column and then we proceed as in *case 1*

Travis's New Version

A column C's representation consists of a bitvector $B[0..m-1]$ and a sequence of thresholds T. It supports the query $CANDIDATE\_STEP$, which takes a single integer i and a bit b and returns a boolean flag f and a single integer i'. If $B[i] = b$, then $f = TRUE$ and i' is the position of $B[i]$ after B is stably sorted. If $B[i] \neq b$ but there is some copy of b in B, the $f = FALSE$ and i' is the position after B is stably sorted of either of the last copy of b before B[i] or of the first copy of b after $B[i]$, depending on whether $B[i]$ is before or after the threshold for the run containing $B[i]$. If there is no copy of b in B, then $f = FALSE$ and $i' = -1$.
We store B and T run-length compressed, so they take $O(r_c)$ words of space and $CANDIDATE\_STEP$ takes $O(\log \log m)$ time. We start a search with $i = 0$; we go from one column to the next setting $i = i'$ when $i' \geq 0$, with f telling us whether we've jumped or not (**but not telling us whether we've hit the end of a MEM**); when $i' = -1$, we were unable to match a column and we start over at the next column with $i = 0$.

Outline

Durbin's Algorithm Example

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

Durbin's Algorithm Example

e = 0 e = 3 e = 7 e = 11

X	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
00	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
01	0	0	0	0	0	0	0	0	0	2	2	2	7	9	6	15
02	0	0	0	0	0	1	4	4	4	5	5	5	8	0	0	6
03	0	0	0	0	0	4	2	2	2	0	3	6	9	12	0	0
04	0	0	2	2	0	2	0	0	0	3	6	4	0	6	8	0
05	0	0	0	0	0	0	0	0	0	6	4	0	4	0	0	8
06	0	0	0	0	1	0	5	5	5	4	0	0	0	0	11	0
07	0	0	0	0	3	0	0	0	0	0	0	0	12	8	9	11
08	0	0	0	0	0	0	0	0	0	0	0	7	2	0	0	0
09	0	0	0	0	4	0	0	0	0	0	7	8	5	11	10	10
10	0	0	0	0	0	0	3	3	3	7	8	0	6	9	13	13
11	0	0	0	0	0	5	0	4	6	8	0	0	4	0	7	7
12	0	0	0	0	0	0	4	0	4	0	0	9	0	10	9	9
13	0	0	0	0	2	0	0	0	0	0	9	0	0	13	0	0
14	0	0	0	0	0	0	0	6	0	9	0	4	8	7	12	12
15	0	1	0	0	0	3	6	4	0	0	4	0	0	9	2	2
16	0	0	0	0	0	0	4	0	7	4	0	11	11	0	6	6
17	0	0	0	1	0	4	0	0	8	0	5	9	9	12	14	14
18	0	0	0	3	0	0	0	0	0	5	0	0	0	2	9	9
19	0	0	1	0	0	0	0	7	0	0	10	10	10	6	0	0

14

Durbin's Algorithm Example

X	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
00	0	4	0	0	8	14	14	14	14	0	0	0	7	1	18	11
01	1	5	1	1	11	15	15	15	15	16	16	16	19	9	4	18
02	2	6	2	2	12	17	0	0	0	8	11	18	1	10	5	4
03	3	7	3	3	13	0	9	9	9	11	18	17	14	18	6	5
04	4	8	4	4	14	4	10	10	10	18	17	4	15	4	2	6
05	5	9	5	5	15	5	16	16	16	17	4	5	9	5	3	2
06	6	10	6	6	17	6	8	8	8	4	5	6	10	6	11	3
07	7	11	7	7	18	7	11	11	11	5	6	7	0	2	12	12
08	8	12	8	8	19	9	12	12	12	6	7	19	16	3	13	13
09	9	13	9	9	0	10	13	13	13	7	19	1	18	11	8	8
10	10	14	10	10	1	16	18	18	18	19	1	2	17	12	7	7
11	11	15	11	11	2	8	19	1	17	1	2	3	4	13	19	19
12	12	16	12	12	3	11	1	2	4	2	3	14	5	8	14	14
13	13	18	13	13	4	12	2	3	5	3	14	15	6	7	15	15
14	14	19	14	14	5	13	3	17	6	14	15	9	2	19	0	0
15	15	0	15	15	6	18	17	4	7	15	9	10	3	14	16	16
16	16	1	16	16	7	19	4	5	19	9	10	11	11	15	17	17
17	17	2	18	17	9	1	5	6	1	10	12	12	12	0	1	1
18	18	3	19	18	10	2	6	7	2	12	13	13	13	16	9	9
19	19	17	17	19	16	3	7	19	3	13	8	8	8	17	10	10

Durbin's Algorithm Example

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

$pre_e = 6$ at 7

$pre_e = 8$ at 11

$pre_e = 13$ at 13

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

New Idea I

Let's take a step back and get closer to Durbin's original idea.

At the moment we save:

- the position of every head of a run
- a boolean to mark if the first run is composed by zeros or ones
- the c value of the column
- a single value for u and v (that are as in Durbin)
- the whole *divergence array*, actually the *LCP array*, (**WIP**)

New Idea I

```
column: 5
start with 0? yes, c: 15
0   0
2   2
3   1
4   3
8   5
0 5 4 1 3 5 5 5 5 5 5 0 5 5 5 2 5 1 5 5
```

Figura: Example, column 5: 0010111000000000000

New Idea II

uv values trick

Values u and v increase alternately in the biallelic case so we save every time the only value that increase at the head of a run.
The, with a simple *If/Else* selection based on the first element of the column and the index of the run and on being even or odd of the index we can extract both u and v values. Infact the two values are, alternatively, saved in the current index and in the previous one.

$w(i, \sigma)$ function

We can use the same *LF-mapping* as in Durbin but we have to consider every time an *offset* between the position of the head of the run, that's i, which contains the "virtual" index, and the index itself.

$$w(i, \sigma) = \begin{cases} u[i] + \mathit{offset} & \text{if } \sigma = 0 \\ c + v[i] + \mathit{offset} & \text{if } \sigma = 1 \end{cases}$$

New Idea III

External haplotype matches

Than we proceed as in Durbin, updating f and g using $w(i, \sigma)$.
Every time we "virtually" use indexes over the whole column but actually run heads plus offsets are used.
In case we update e using f and the *divergence/LCP array* (**WIP**).
In order to update e we should in theory follow the line indicated in $i+1$ by f in the original panel which we have not memorized. So, at most at a cost of $O(r)$ for every column, we proceed to reverse te use of u and v to move backwards between the columns virtually following a row of the original panel.
Than we use the *divergence/LCP array* (**WIP**) to update f and g depending on the case.
After detect a match, we can know the cardinality of the lines that match but not what they are,

New Idea IV

WIP

At the moment *divergence array* is saved as an `sdsl::int_vector<>` on which it's used `sdsl::util::bit_compress()` in order to save space.
The original idea of thresholds seems to me absolutely not applicable but maybe we can think of storing only a subset of the *divergence/LCP array* and I'm thinking how to do it.

Two Pass on RLPBWT

In order to not save the *divergence/LCP array* we could make two *RLPBWT*, one for the normal order and one for the reverse order.

Instead of use the *divergence/LCP array* to retrieve matches that are overlapped in the panel we look for non overlapped matches only.
To do this when we find a match we continue the search updating f and g based only on c and h, the total number of haplotypes in the panel:

$$(f, g) = \begin{cases} (0, c) & \text{if } \sigma = 0 \\ (c, h) & \text{if } \sigma = 1 \end{cases}$$

We query in this way the first one with the haplotype and the second one with the reverse of the haplotype. Than we intersect the results.

Two Pass on RLPBWT

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

Two Pass on RLPBWT

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

Two Pass on RLPBWT

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

Match Forward Backward