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

$[10, 11, 12, 13, 14]$

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

$[15, 16, 17, 18, 19]$

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

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 $i = 0$.

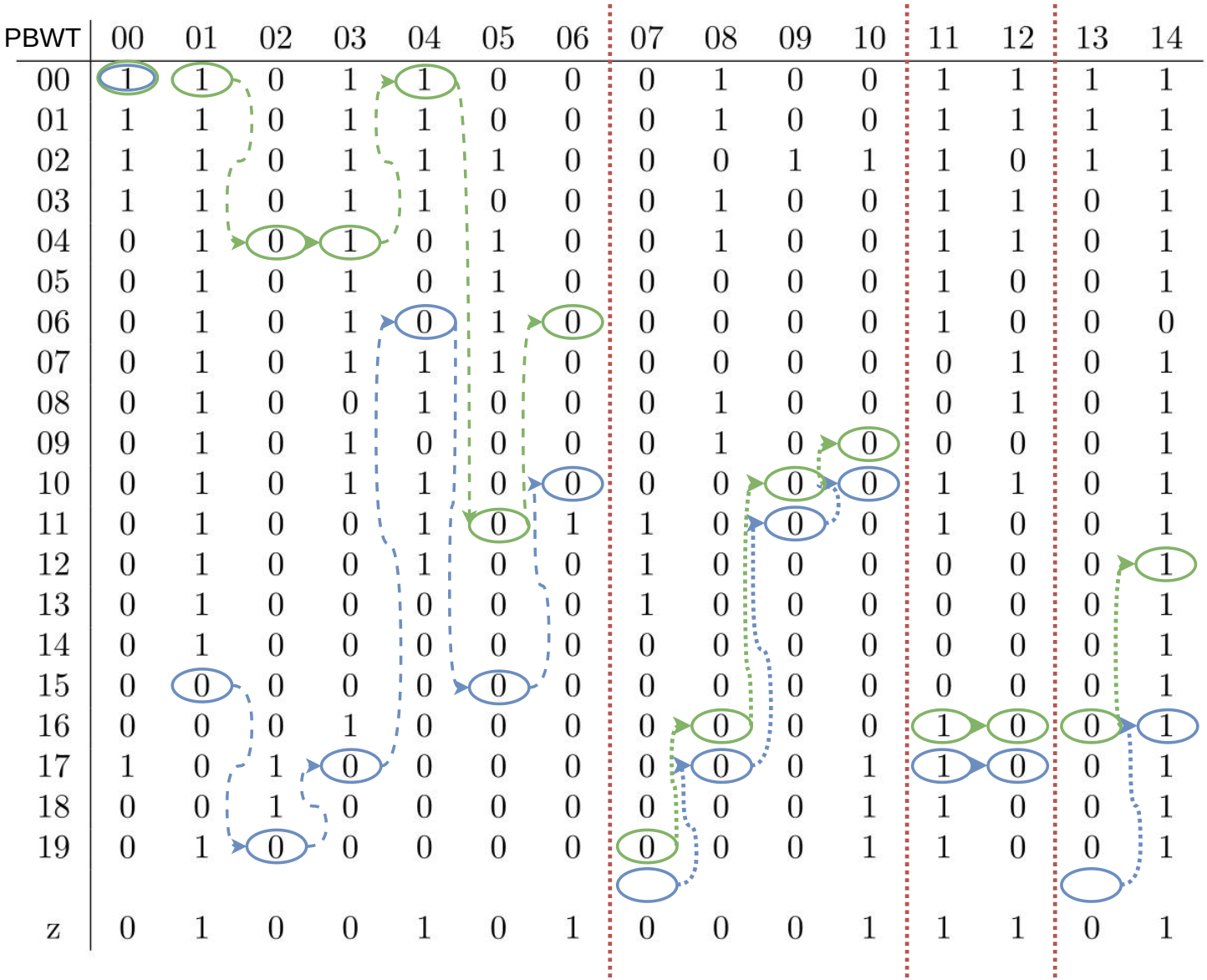
Outline

Durbin's Algorithm

Algorithm 1 Algorithm 5 from Durbin's paper

function FIND_SET_MAXIMAL_MATCHES_FROM_Z(z)
  for $k \leftarrow 0$ **to** N **do**
    $e, f, g \leftarrow Update\_Z\_Matches(k, z, e, f, g)$
function UPDATE_Z_MATCHES(k, z, e, f, g)
  $f' \leftarrow w(k, f, z[k])$ ▷ a_k, d_k and y_i^k as in Durbin's paper
  $g' \leftarrow w(k, g, z[k])$
  if $f' < g'$ **then** ▷ *if k is $N-1$ report matches from e_k to $N-1$*
    $e' \leftarrow e_k$
  else ▷ *report matches from e_k to k*
    $e' \leftarrow d_{k+1}[f'] - 1$
    if $z[e'] = 0$ **and** $f' > 0$ **then**
      $f' \leftarrow g' - 1$
      while $z[e' - 1] = y_{f'}^{k+1}[e' - 1]$ **do** $e' \leftarrow e' - 1$
      while $d_{k+1}[f'] \leq e'$ **do** $f' \leftarrow f' - 1$
    else
      $g' \leftarrow f' + 1$
      while $z[e' - 1] = y_{f'}^{k+1}[e' - 1]$ **do** $e' \leftarrow e' - 1$
      while $g' < M$ **and** $d_{k+1}[g'] \leq e'$ **do** $g' \leftarrow g' + 1$
  return e', f', g'

Durbin's Algorithm Example



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

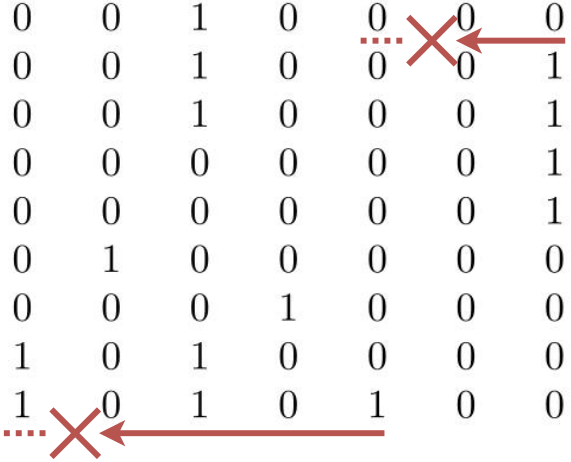
14

Durbin's Algorithm Example

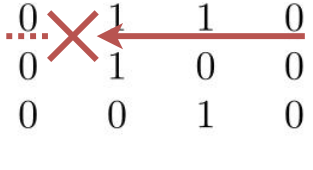
X	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
00	0	4	0	0	8	14	14	14	14	0	0	0	7	1	18	11
01	1	5	1	1	11	15	15	15	15	16	16	16	19	9	4	18
02	2	6	2	2	12	17	0	0	0	8	11	18	1	10	5	4
03	3	7	3	3	13	0	9	9	9	11	18	17	14	18	6	5
04	4	8	4	4	14	4	10	10	10	18	17	4	15	4	2	6
05	5	9	5	5	15	5	16	16	16	17	4	5	9	5	3	2
06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

Outline

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: 00101111000000000000

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 sometimes an *offset* between the position of the head of the run, that's i, which contains the "virtual" index, and the index itself.

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

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.

Testing

Benchmark

At the moment I'm testing the implementation using the sample data (VCF files) at:
https://github.com/ZhiGroup/Syllable-PBWT/tree/master/sample_data
(the panel is 900×500 with 100 queries).

Outline

The column data I

We save, for every column:

- a bit vector of the same length of the dense PBWT column, with 1 in every position before a head of a run
- 2 bit vector that can be queried to obtain u and v value. So we have a bitvector for zeros that contains 1 in position i iff we have i zeros at point in the column when we change value anche similar for ones(**to be exaplained better**)
- the c value and a bool to indicate how a column start

The column data II

example

If we have a column:

$$c = 00001110001111110\mathbf{0}001111111$$

We save:

$$h = 00010010010000010010000000(1)$$

$u = 00010010001$ (to rappresent 4,3 and 4 zeros)

$v = 0010000010000001$ (to rappresent 3,6 and 7 ones)

The column data III

example

If we want to obtain, for example, the number of zeros before and index i, for example $i = 18$:

- we *rank* h to obtain the run that contain i, $rank_h(18) = 4$
- we know that the column start with 0 so we are in a run of zeros and we have $\lfloor \frac{4}{2} \rfloor = 2$ run's of zeros before
- we know that the number of zeros in the previous complete runs is $select_u(2) + 1 = 7$
- remain zeros in the run to compute are given by $i - (select_h(4) + 1)$

Outline

Matching Statistics and RLPBWT

Some definitions

- for every run in a column of the PBTW with define the **threshold** as the index of the minimum *lcp value*, including the lcp of the first value of the next run and we save it as a sparse bitvector of the same length of a column with 1 in these positions. In case the thresholds is the first value of the next run we set as threshold the last index of the current run
- for every run in a column of the PBTW with define the **prefix array samples** as the prefix array value at the begin of the run and at the end
- we define matching statistics as a two components (row, len) vector MS of the same length of the query such that $\forall k \in [0, |z|)$:
  - $panel_{row}[k-(len+)1..k] = z[k-(len+1)..k]$
  - $z[k-(len+1)..k+1]$ doesn't match with any row in the panel
- we compute *row* and *len* in two different scans
- we need random access to the panel. At this time we are using a bitvector for every column of the panel but we can use a data structure, called *SLP*, with a very low memory cost but with high access times

Example

PBWT	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14
00	1	1	0	1	1	0	•0	0	1	0	0	1	1	1	1
01	1	1	0	1	1	0	0	0	1	0	0	1	1	1	1
02	1	1	0	1	1	1	0	0	0	1	1	1	0	1	1
03	1	1	0	1	1	0	0	0	1	0	0	1	1	0	1
04	0	1	0	1	0	1	0	0	1	0	0	1	1	0	1
05	0	1	0	1	0	1	0	0	0	0	0	1	0	0	1
06	0	1	0	1	•0	1	0	0	0	0	0	1	0	0	0
07	0	1	0	1	1	1	0	0	0	0	0	0	1	0	1
08	0	1	0	0	1	0	0	0	1	0	0	0	1	0	1
09	0	1	0	1	0	0	0	0	1	0	0	0	0	0	1
10	0	1	0	1	1	0	0	0	0	0	0	1	1	0	1
11	0	1	0	0	1	0	1	1	0	0	0	1	0	0	1
12	0	1	0	0	1	0	0	1	0	0	0	0	0	0	1
13	0	1	0	0	0	0	0	1	0	0	•0	0	0	0	1
14	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
16	0	0	0	1	0	0	0	0	0	0	0	1	•0	0	1
17	1	0	1	0	0	0	0	0	0	0	1	1	0	0	1
18	0	0	•1	0	0	0	0	0	0	0	1	1	0	0	1
19	0	1	0	0	0	0	0	0	0	0	1	1	0	0	1
z	0	1	0	0	1	0	1	0	0	0	1	1	1	0	1
POS	19	19	16	15	13	13	19	19	19	19	11	11	17	17	17
LEN	1	2	3	4	5	6	4	5	6	7	4	5	2	3	4

Example

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

Matching Statistics using LCE queries

- using an SLP we can make **Longest Common Extensions (*LCE*) queries**. Given two row of the panel and a column we can compute the longest common suffix as far as the given column.
- as for the case of the thresholds study we starts from the bottom-left symbol in the panel and we move left to right using LF-mapping
- if we have a match we continue adding 1 every time to the *len* of *MS*
- every time we have a mismatch we look for the previous and the next good symbol in the column. Using the prefix array samples we have the original-row index of the end of the previous run and the original-row index of the begin of the next run (if they exists)
- if we are in a column with only the wrong symbol we restart the computation from the last row of the original panel
- we compute the two *LCE* and we take the longest
- next *row* in the *MS* will be the index of the row with the maximum *LCE* and the *len* of the *MS* will be the length of the *LCE* plus 1

Example

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

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

Example

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

Extract the matching rows I

The idea on the RLBWT

On the *RLBWT* we can list every starting position of a pattern in a text, of length n, using two functions. Given an SA value p:

- $\varphi(p) = SA[ISA[p] - 1]$ or $NULL$ if $ISA[p] = 0$
- $\varphi^{-1}(p) = SA[ISA[p] + 1]$ or $NULL$ if $ISA[p] = n - 1$

In other words the two functions gives pre previous and the next values of the SA.

Extract the matching rows II

The idea on the RLPBWT

We can use something similar on the *RLPBWT* to understand, when we have a match ending in column k, which rows are matching, having knowledge of one of this rows, thanks to the *matching statistics row* value.
In order to obtain this result we need an additional data structure:

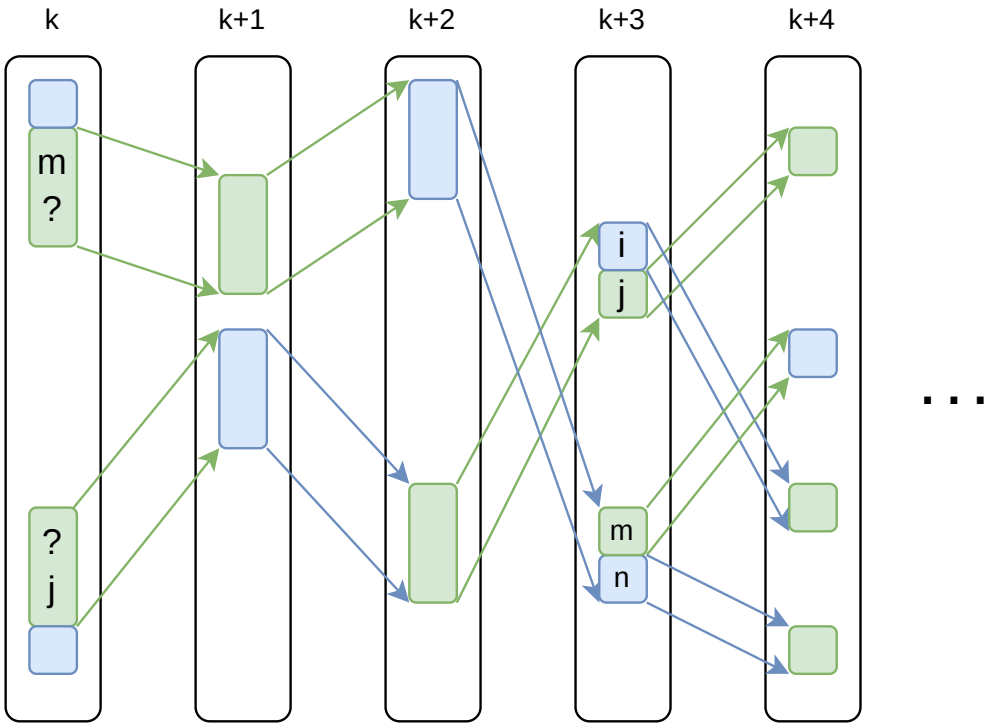
- a sparse bit vector matrix for the φ. Given a row and a column we have a 1 in this position iff that row is a run head in that column
- a sparse bit vector matrix for the φ^{-1}. Given a row and a column we have a 1 in this position iff that row is a run end in that column
- two matrix of int_vector, for the two panel, with a value for every 1 in the sparse bit vector matrices

In addition we use the last prefix array to complete the informations of the last column, for examples for equal rows in panel.
Infact the key idea is to record every time two consecutive rows in a certain order as far as column $k-1$ change their relative order in column k.

This data structure could be computed only at request and take less space than a complete prefix array for every column.

We use φ and φ^{-1} to make queries from a given row and a given column. To make this type of query we simply *rank* the given column in the bitvector at the given row and we give the result based on the support vector.

So every time we have a match we could start from *row* value and go up/down as far as the row obtained from the two queries is "good". Infact we can use *LCE* function or simply the random access to check the matches with the near rows in the permutation at column k.



PhiVec_j	0	0	0	1	0	. . .
PhiSup_j	i	. . .				
PhiInvVec_m	0	0	0	1	0	. . .
PhiInvSup_m	n	. . .				

Figura: In this example we have $\varphi(j,k) = i$ and $\varphi^{-1}(m,k) = n$: Notice that k could also be $k+1$, $k+2$ or $k+3$.