RLPBWT

Davide Cozzi

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

Outline

1. The Old Idea
2. The New Idea
3. BitVector Version

Outline

1 The Old Idea

2 The New Idea

3 BitVector Version

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**
        $e, f, g \leftarrow \mathit{Update\_Z\_Matches}(k, z, e, f, g)$
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

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

X	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
	e = 0						*e = 3*				*e = 7*		*e = 11*			
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

1 The Old Idea

2 The New Idea

3 BitVector Version

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.

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

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

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).

Next Work

Trying to implement also non maximal matches with a query, as, for example, in Naseri's paper.

Outline

1 The Old Idea

2 The New Idea

3 BitVector Version

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 of 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:

$$c = 00001110001111110001111111$$

We save:

$$h = 00001001001000000100001000000$$

$$u = 0010010001 \text{ (to rappresent 4,3 and 4 zeros)}$$

$$v = 0010000010000001 \text{ (to rappresent 3,6 and 7 ones)}$$

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 $select_u(2) = 7$
- remain zeros in the run to compute are given by $i - select_h(4)$

TODO

Re-implement the algorithm using bitvectors, *rank* and *select* instead of run head indices and *uv* values.