RLPBWT con bitvectors

Algorithm 1 Algoritmo per la costruzione di una colonna della RLPBWT con bitvectors

```
1: function BUILD(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)
```

Algorithm 2 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
```

Algorithm 3 Algoritmo per uvtrick

```
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 (index, 0)
8:         else
9:             return (0, index)
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)
```

Algorithm 4 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
```

Algorithm 5 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))
```

Algorithm 6 Algoritmo per match con aplotipo esterno con panel $width \times height$

```
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
```

Algorithm 7 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 ▷ 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**
    report match at $ms_{row}[k]$ *with* $ms_{len}[k]$ *iff* $ms_{len}[k] > ms_{len}[k+1]$
    or $ms_{len}[k] = ms_{len}[k+j]$, $j \geq 1$ and $\nexists j$ *such that* $ms_{len}[k] < ms_{len}[k+j]$

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* $\bot$

Algorithm 8 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**
    report match at $ms_{row}[k]$ *with* $ms_{len}[k]$ *iff* $ms_{len}[k] > ms_{len}[k+1]$
    or $ms_{len}[k] = ms_{len}[k+j]$, $j \geq 1$ and $\nexists j$ *such that* $ms_{len}[k] < ms_{len}[k+j]$

Algorithm 9 Algoritmo per l'update usando le matching statistics

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

Algorithm 10 Algoritmo per la costruzione della struttura per φ e φ^{-1}

function BUILD_PHI($cols, panel, prefix$) ▷ $prefix$ is the last prefix array
    $\varphi \leftarrow [[0..0]..[0..0]],\ \varphi^{-1} \leftarrow [[0..0]..[0..0]]$ ▷ sparse bit vector panels for φ and φ^{-1}
    $\varphi_{supp} = [],\ \varphi^{-1}_{supp} = []$ ▷ vectors for φ and φ^{-1} row values
    for *every* $k \in [0, |cols|)$ **do**
        for *every* $i \in [0, |samples_{beg}|)$ **do**
            $\varphi[sample^k_{beg}[i]][k] \leftarrow 1$
            if $i = 0$ **then**
                $push(\varphi_{supp}[sample^k_{beg}[i]], panel_{height})$
            else
                $push(\varphi_{supp}[sample^k_{beg}[i]], sample^k_{end}[i-1])$
            $\varphi^{-1}[sample^k_{end}[i]][k] \leftarrow 1$
            if $i = |sample^k_{beg}| - 1$ **then**
                $push(\varphi^{-1}_{supp}[sample^k_{end}[i]], panel_{height})$
            else
                $push(\varphi^{-1}_{supp}[sample^k_{end}[i]], sample^k_{beg}[i+1])$
    for *every* $k \in [0, |prefix|)$ **do**
        if $\varphi[k][|\varphi[k]| - 1] = 0$ **then**
            $\varphi[k][|\varphi[k]| - 1] \leftarrow 1$
            if $k = 0$ **then**
                $push(\varphi_{supp}[prefix^k], panel_{height})$
            else
                $push(\varphi_{supp}[prefix^k], prefix^k[i-1])$
        if $\varphi^{-1}[k][|\varphi[k]| - 1] = 0$ **then**
            $\varphi^{-1}[k][|\varphi[k]| - 1] \leftarrow 1$
            if $k = |prefix| - 1$ **then**
                $push(\varphi^{-1}_{supp}[prefix^k], panel_{height})$
            else
                $push(\varphi^{-1}_{supp}[prefix^k], prefix^k[i+1])$
    build rank/select for every sparse bitvector in φ and φ^{-1}

Algorithm 11 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

Algorithm 12 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*