Algorithm 1: `fill_runs_slice`$(\mathtt{T}, \mathtt{t}, \mathtt{from}, \mathtt{to}, \mathtt{v}, \mathtt{runs})$

Input: Cst, `T`, of index string `s` supporting `WeinerLink` and `Parent`; Query string `t`; Interval over `t`, [`from`, `to`); Starting node `v`; Vector `runs`.

Output: Tripple (`first`, `last`, `v`): `first` index of the first failing `WeinerLink` call; `last` index last failing `WeinerLink` call; `v` the node corresponding to `last`;

```
1  v ← T.WeinerLink(T.root(), t[to − 1]);
2  idx_set ← FALSE;
3  k ← to − 2;
4  repeat
5  |  c ← t[k − 1];
6  |  if T.IsRoot(T.WeinerLink(v, t[to − 1])) then
7  |  |  if idx_set = FALSE then
8  |  |  |  first ← k;
9  |  |  |  idx_set ← TRUE;
10 |  |  end
11 |  |  runs[k] ← 0;
12 |  |  repeat
13 |  |  |  v ← T.parent(v);
14 |  |  until T.IsRoot(T.WeinerLink(v, c));
15 |  |  v ← T.WeinerLink(v, c);
16 |  |  last ← k;
17 |  else
18 |  |  runs[k] ← 1
19 |  end
20 |  v ← T.WeinerLink(v, c);
21 |  k ← k − 1;
22 until k > from;
```

Algorithm 2: Build `runs` in parallel.

Input: Cst, `T`, of index string `s` supporting `WeinerLink` and `Parent`; Query string `t`; Number of threads `nthreads` dividing |`t`|.

```
1 S ← |t|/nthreads;
2 R ← [1..nthreads];
3 runs ← [1..t];
4 for i ← 1 to nthreads do // run asynchronously
5 |  R_i ← fill_runs_slice(T, t, (i − 1) × S, i × S);
6 end
7 for i ← 1 to nthreads − 1 do // run asynchronously
8 |  fill_runs_slice(T, t, R_{i−1}[0], R_i[1], R_i[2]);
9 end
```

Algorithm 3: `fill_ms_slice`(`T`, `t`, `ms`, `runs`, `from`, `to`)

Input: Cst of index string $\bar{\mathtt{s}}$, `T`, supporting `WeinerLink` and `Parent`; Query string `t`; Vector `ms` initialized at 0; Vector `runs`; Interval over `t`, [`from`, `to`)

Output: Last index `j` of `ms`

```
1  k ← from, j ← 0, h* ← k + 1;
2  c ← t[k];
3  v ← T.root();
4  repeat //extend current match
5  |  h ← h*;
6  |  u ← v;
7  |  repeat
8  |  |  c ← t[h*];
9  |  |  u ← T.WeinerLink(v, c);
10 |  |  if !T.IsRoot(u) then
11 |  |  |  v ← u;
12 |  |  |  h* ← h* + 1;
13 |  |  end
14 |  until !T.IsRoot(u) and h* < |t|;
15 |  // resize ms if needed;
16 |  j ← j + h* − h + 1 // set ms[j..j + h* − h + 1] to 0;
17 |  ms[j + +] ← 1;
18 |  if h* < |t| then
19 |  |  repeat// clip prefixes of t[k..h*] until can extend by c
20 |  |  |  v ← T.Parent(v);
21 |  |  |  u ← T.WeinerLink(v, c);
22 |  |  until !T.IsRoot(u);
23 |  |  h* ← h* + 1;
24 |  end
25 |  k′ ← k + 1 // index of first 1 in runs[k + 1..];
26 |  repeat
27 |  |  k′ ← k′ + 1;
28 |  until k′ < |t| and runs[k′] ≠ 0;
29 |  // resize ms if needed;
30 |  ms[k + 1..k′ − 1] ← 1;
31 |  j ← max(k′ − 1, to) // set ms[j.. max(k′ − 1, to)] to 0;
32 until k < to;
```

Algorithm 4: Build `ms` in parallel.

Input: Cst of index string $\bar{\mathtt{s}}$, `T`, supporting `WeinerLink` and `Parent`; Query string `t`; Number of threads `nthreads` dividing $|\mathtt{t}|$.

```
1  S ← |t|/nthreads;
2  for i ← 1 to nthreads do
3  |  ms_i ← [1..2|t|/nthreads];
4  end
5  for i ← 1 to nthreads do // run asynchronously
6  |  J_i ← fill_ms_slice(T, t, ms_i, runs, (i − 1) × S, i × S);
7  end
8  for i ← 1 to nthreads − 1 do
9  |  ms_i ← resize_to(J_i);
10 end
```