LCP_k Implementation notes

srirampc

Notations

- I use 0 based indexing in this document.
- X and Y are the strings
- For the string $R = X + \# + Y + \$$, SA, ISA is the suffix array and inverse suffix arrays respectively.
- Length of SA, ISA is $|R|$
- LCP is constructed for R. $LCP[i]$ is least common prefix of suffixes corresponding to $SA[i]$ and $SA[i-1]$. $LCP[0]$ is set as 0.
- RMQ is the range minimum query table constructed for LCP. $RMQ(i,j)$, $i \leq j$ returns the minimum index k of LCP $(i \leq k \leq j)$ for which the value is minimum in the range $LCP[i], \ldots, LCP[j]$.

Top-level View

- Algorithm does a depth-first search of suffixes with errors. When we allow one error, the search goes down a level deep.
- At search depth 0, we start with all suffixes of R, as we go down the levels we consider a subset of suffixes of R, constructed from the suffixes at the previous level.
- Search continues until we reach a depth of level k, and at which point we update LCP_k using the suffixes collected at the current search frontier. We continue the depth-first search until we run out of suffixes.
- Output is a 3-dimensional array $MLCP$, where $LCP_k[0][1][i]$ is $\max_{Y_i} LCP_k(X_i, Y_i)$ and $LCP_k[0][0][i]$ is $\arg\max_{Y_i} LCP_k(X_i, Y_i)$. Similarly $LCP_k[1]$ records the corresponding values for Y with respect to X.

Suffix Representation

- We consider the suffixes under the context of a specific internal node u. u is present in from a suffix tree (at search depth 0) or a suffix trie (at search depths $1, \ldots, k-1$).
- We represent the suffixes of the trie using the following tuple

  $$(c, c', src)$$

  where
  - c is the position in the string R,
  - c' is the position in SA of suffix c after chopping $1 + \textsc{String-Depth}(u)$
  - src is the source string either 0 or 1, 0 corresponding to X and 1 corresponding to Y
- In this document, I use access of a specific element in the tuple as function. For example, to access field c of the suffix r, I write $c(r)$.

Internal Node Representation

- We construct suffixes at search level j by chopping prefixes of the suffixes under at search level $j-1$.
- At level j, we represent an interal node u (both for the suffix tree and the suffix tries implicitly generated) with the following tuple

$$(bp, ep, d, \delta)$$

  where
  - bp and ep are the start and end indices in the sorted array of suffixes. At level 0, this is the suffix array of R, and this bp and ep are indices of suffix array. At level $1, \ldots, k-1$, they the some subset of suffixes selected from R
  - d string depth of the internal node
  - δ the sum of chopped of lengths at the levels below $0, \ldots, j-1$.

Suffix Array of R

- Since $\# < \$ < \{A, \ldots, Z\}$, in $SA[0]$ and $S[1]$ are $\#Y$ and $\$$ – the end of first and second strings – respectively.
- As we chop off suffixes and reach towards the end of the strings, we end up in either at the first or second entry of the suffix array. When we proceed to the next search level, we ignore these suffixes.

Compute LCP_k

- Main function of computing LCP_k, each function being called is written in the later slides.

COMPUTE-LCPK(k)

U = SA-INTERNAL-NODES()
for $i = 0$ **to** $|U| - 1$
    S_0 = SA-CHOP-PREFIX($U[i]$)
    RECURSIVE-COMPUTE-LCPK($U[i], S_0, k-1$)

RECURSIVE-COMPUTE-LCPK(u, S_{j-1}, k)

if $k = 0$
    UPDATE-MLCP(u, S_{j-1})
    return
U_j = TRIE-INTERNAL-NODES(u, S_{j-1})
for $i = 0$ **to** $|U_j| - 1$
    S_j = TRIE-CHOP-PREFIX($U_j[i], S_{j-1}$)
    RECURSIVE-COMPUTE-LCPK($U_j[i], S_j, k-1$)

Selecting Internal Nodes from SA or Trie

- Representation of internal node is given in slide 5

SA-INTERNAL-NODES()

```
Initialize U_0 to be of an array of Internal nodes of size |SA|
for leaf = 0 to |SA| - 1
        U_0[leaf] = SA-SubTree(leaf)
Sort and remove duplicates in U_0
return U_0
```

TRIE-INTERNAL-NODES(u, S_{j-1})

```
Initialize U_j to be an array of Internal nodes of size |S_{j-1}|
for leaf = 0 to |S_{j-1}| - 1
        U_j[leaf] = Trie-SubTree(u, leaf, S_{j-1})
Sort and remove duplicates in U_j
return U_j
```

Internal Nodes of Leaf in Suffix Tree

```
SA-SubTree(leaf)
LCP[leaf + 1] corresponds to (leaf, leaf + 1)
sp = ep = leaf
while sp > 2
        if LCP[sp + 1] >= LCP[leaf + 1]
                break
        sp = sp − 1
while ep < |SA| − 1
        if LCP[ep + 1] >= LCP[leaf + 1]
                break
        ep = ep + 1
return (sp, ep, LCP[leaf + 1], 0)
```

Internal Nodes of a Leaf in Suffix Trie

- The input u, internal node at a level $j-1$ (slide 5)
- The input S_j, sorted suffixes at level $j-1$ (slide 4)

TRIE-SUBTREE$(u, leaf, S_{j-1})$

$l_x = RMQ(c'(S_{j-1}[leaf]) + 1, c'(S_{j-1}[leaf+1]))$
$sp = ep = leaf$
while $sp > 0$
    if $RMQ(c'(S_{j-1}[sp]) + 1, c'(S_{j-1}[sp+1])) >= lx$
        break
    Decrement sp
while $ep < |S_{j-1}|$
    if $RMQ(c'(S_{j-1}[ep]) + 1, c'(S_{j-1}[ep+1])) >= lx$
        break
    Increment ep
return $(sp, ep, l_x, d(u) + \delta(u) + 1)$

Chop Suffixes w.r.t Internal Node u - Suffix Array

SA-CHOP-PREFIX(u)

Initialize S_0 array of size $ep(u) - bp(u) + 1$
for $i = bp(u)$ **to** $ep(u)$
    $c(S_0[i]) = SA[i]$
    if $SA[i] < |X|$
        $src(S_0[i]) = 0$
    else
        $src(S_0[i]) = 1$
    $c'(S_0[i]) = ISA[c + d_u + 1]$
Remove invalid suffixes in S_0 (indices beyond $|X| - 1$ or $|Y| - 1$)
Sort S_0 based on c'
return S_0

Chop Suffixes w.r.t Internal Node u - Trie

TRIE-CHOP-PREFIX(u, S_{j-1})

Initialize S_j array of size $ep(u) - bp(u) + 1$
for $i = bp(u)$ **to** $ep(u)$
    $src(S_j[i]) = src(S_{j-1}[i])$
    $epx = c'(S_{j-1}[i])$
    $c(S_j[i]) = SA[epx]$
    $c'(S_j[i]) = ISA[SA[epx] + d_u + 1]$
Remove invalid suffixes in S_j (indices beyond $|X| - 1$ or $|Y| - 1$)
Sort S_j based on c'
return S_j

Update $MLCP$ Pass

UPDATE-MLCP-LTOR(u, S_j)

$p = 0$; $q = 1$; $L_H[0] = 0$; $L_H[1] = 1 + |X|$
while $q < |SA|$ **and** $src(S_j[p])! = src(S_j[q])$
    $p = q$; $q = q + 1$
while $q < |SA|$
    $i = c(S_j[q]) - \delta(u) - L_H[src(S_j[q])]$
    $j = c(S_j[p]) - \delta(u) - L_H[src(S_j[p])]$
    $rmin = RMQ(c'(S_j[p]), c'(S_j[q]))$
    $score = d(u) + \delta(u) + rmin + 1$
    if $score > MLCP[src(S_j[q])][1][i]$
        $MLCP[src(S_j[q])][1][i] = score$;
        $MLCP[src(S_j[q])][0][i] = j$
    $q_x = q$; $q = q + 1$
    if $src(S_j[p]) == src(S_j[q])$
        $p = q_x$