LCP_k Implementation and Experiments

srirampc

1 Implementation

1.1 Outline

Our algorithm computes LCP_k by a depth-first search of sorted suffixes allowing upto k-mismatches. Output of the algorithm is a 3-dimensional array $KLCP$. $KLCP[1, 1, i]$ records $\max_j LCP_k(X_i, Y_j)$, while $KLCP[1, 2, i]$ records $\arg\max_j LCP_k(X_i, Y_j)$. Similarly, $KLCP[2, \cdot, \cdot]$ records the corresponding values for Y with respect to X. $ACS_k(X, Y)$ can be computed using the $KLCP$ arrays using the forumla given in section (TODO: ref above).

Our algorithm uses Suffix array, ISA, LCP arrays and RMQ tables as the underlying data-structures. We construct suffix array SA_R for the string R = X#Y\$ using the libdivsufsort library (TODO: cite). We also construct ISA and LCP arrays (LCP_R), and the RMQ tables (RMQ_R) based on the implementations from the SDSL library (TODO: cite). We use the implementations of Kasai Algorithm (TODO: cite) and Bender-Farach's algorithm (TODO: cite) for construction of LCP arrays and RMQ tables respectively. Though SDSL library uses bit compression techniques to reduce the size of the tables and arrays in exchange for relatively longer time to answer queries, we don't compress these data structures. We use 32-bit integers for indices and prefix lengths since they are sufficient for our purposes.

The following psuedo-code shows the outline of our implementation of the depth-first search algorithm.

```
COMPUTE-LCPK()
1  $U$ = ST-INTERNAL-NODES()
2  for $i = 1$ to $|U|$
3        $S_1$ = ST-CHOP-PREFIX($U[i]$)
4        RECURSIVE-COMPUTE-LCPK($U[i], S_1, 1$)
```

The procedure COMPUTE-LCPK first calls the function ST-INTERNAL-NODES to select all the internal nodes of the suffix tree of R. Then, for each internal node u, a call to ST-CHOP-PREFIX(u) chops the prefixes of length STRING-DEPTH(u) $+ 1$ of all the suffixes corresponding to u. After chopping the prefix, a recursive procedure RECURSIVE-COMPUTE-LCPK is called to explore the search space further in a depth-first manner.

```
RECURSIVE-COMPUTE-LCPK($u, S_j, j$)
1  **if** $j = k$
2      UPDATE-KLCP($u, S_j$)
3      **return**
4  $U_j$ = TRIE-INTERNAL-NODES($u, S_j$)
5  **for** $i = 1$ **to** $|U_j|$
6      $S_{j+1}$ = TRIE-CHOP-PREFIX($U_j[i], S_j$)
7      RECURSIVE-COMPUTE-LCPK($U_j[i], S_{j+1}, j + 1$)
```

The recursive procedure RECURSIVE-COMPUTE-LCPK takes as input an internal node (u) of a suffix trie, the corresponding set of sorted suffixes (S_j) and the search depth (j). When the search depth reaches k, recursion ends and $KLCP$ arrays are updated. Otherwise, the procedure selects the internal nodes of the suffix trie induced by S_j, chops the common prefix of the suffixes corresponding to each internal node in the trie, and recursively calls itself while incrementing the search depth.

1.2 Representation

The procedures shown above represents sets of internal nodes and suffixes as arrays of tuples. We represent an interal node u (both for the suffix tree and the suffix tries) as a tuple (bp, ep, d, δ). Here, bp and ep are the start and end indices pointing to a sorted array of suffixes. At search level 0, this array is SA_R, and bp and ep are indices to SA_R. At search levels $j = 1, \ldots, k-1$, they are a subset of suffixes selected from R. d is the string depth of the internal node in this trie, and δ the sum of the chopped lengths at the previous levels $1, \ldots, j-1$. At level 0, $\delta = 0$.

Suffixes are considered under the context of an internal node u. At search level 0, suffixes are just indices of R. At search levels $j = 1, \ldots, k-1$, we represent the suffixes of the trie with the tuple (c, c', str), where c is the position in the string R, c' is the position in SA_R of suffix c after chopping the first STRING-DEPTH$(u) + 1$ characters. str field indicates whether the suffix is from either X or Y. In this paper, we use function notation to access a specific element of the tuple. For example, to access field c of the suffix x represented by tuple $(i, j, 2)$, we write $c(r) = i$.

Since $\# < \$ < \{A, \ldots, Z\}$, $SA_R[1]$ and $SA_R[2]$ 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 might 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.

1.3 Procedures

As noted above, ST-INTERNAL-NODES selects the internal nodes of the suffix tree of R. In this procedure , there is no explicit construction of the suffix tree. For our purposes, we require only the left and right bounds of the sub-tree corresponding to all internal nodes, which can be computed using only SA_R and LCP_R as follows. $LCP_R[i+1]$ records the string depth of the least common ancestor u of two consecutive pair of suffixes $(i, i+1)$ in SA_R. The left and right bounds of u can be computed by enclosing the pair $(i, i+1)$ with all the suffixes having the same least common ancestor u. By collecting suffixes of the least common ancestor of every pair $(i, i+1), i = 3, \ldots, |SA_R|$, the procedure ST-INTERNAL-NODES computes the left and right bounds for all the internal nodes in the suffix tree of R.

ST-CHOP-PREFIX chops the common prefixes of the suffixes and construct the set of chopped suffix tuples. It also sorts the chopped suffix tuples with c' as the key.

TRIE-INTERNAL-NODES selects left and right bounds of the suffixes corresponding to the internal nodes of the trie. Similar to the suffix tree right and left bounds can be computed by enclosing a pair of suffixes, but instead of LCP_R arrays, we use the RMQ_R tables to retrieve the length of least common prefixes of two suffixes in a trie. Similar to the suffix tree counterpart, TRIE-CHOP-PREFIX chops common prefix of a set of suffixes allowing one mismatch and sorts the suffixes by the c'.

The UPDATE-KLCP procedure, as shown below, makes two passes one from left to right and another from right to left.

```
UPDATE-KLCP($u, S_j$)
1  UPDATE-KLCP-LEFT-TO-RIGHT($u, S_j$)
2  UPDATE-KLCP-RIGHT-TO-LEFT($u, S_j$)
```

Except for the direction of scan, both left-to-right and right-to-left passes are similiar in that they scan through the array of suffixes to identify the pairs of suffixes $(p, q), p < q$ such that one each from X and Y to update the corresponding $KLCP$ entries. UPDATE-KLCP-LEFT-TO-RIGHT(u, S_j) is as shown below.

```
UPDATE-KLCP-LEFT-TO-RIGHT($u, S_j$)
 1  $p = 1; q = 2; L_H[1] = 0; L_H[2] = 1 + |X|$
 2  while $q < |S_j|$ and $src(S_j[p]) = src(S_j[q])$
 3      $p = q; q = q + 1$
 4  while $q < |S_j|$
 5      $r = c(S_j[p]) - L_H[src(S_j[p])] - \delta(u)$
 6      $t = c(S_j[q]) - L_H[src(S_j[q])] - \delta(u)$
 7      $l_{p,q} = \delta(u) + d(u) + RMQ_R(c'(S_j[p]), c'(S_j[q])) + 1$
 8      if $l_{p,q} > KLCP[src(S_j[q])][1][t]$
 9          $KLCP[src(S_j[q])][1][t] = l_{p,q}$
10          $KLCP[src(S_j[q])][2][t] = r$
11      if $src(S_j[p]) == src(S_j[q+1])$
12          $p = q$
13      $q = q + 1$
```

Lines 1 – 3 initializes pointers p and q to the first consecutive suffixes in S_j such that $S_j[p]$ and $S_j[q]$ originate from two different strings. Lines 5 – 10 evaluate the length of longest common prefix with k-mismatches between suffixes $S_j[p]$ and $S_j[q]$. If the evaluted prefix length is longer than current length recored by $KLCP$ for $src(S_j[q])$, it updates the corresponding $KLCP$ entries. Lines – updates the pointers p and q to the next pair of suffixes that from originate from two different strings.