

Common Intervals of Strings

Literature:

Didier, G., Schmidt, T., Stoye, J., & Tsur, D. (2006). *Character sets of strings*. Journal of Discrete Algorithms, 5(2), 330–340.

Definition 1. *The character set $C(S)$ of a string S is the (unordered) set of characters that constitute S :*

$$C(S) := \{S[i] \mid i = 1..|S|\}$$

Example 1. $S = \mathit{biadkfb lhdba}$, $C(S) = \{a, b, d, f, h, i, k, l\}$

An interval of a string is represented by $[i, j] := [i, i + 1, \dots, j]$ and corresponds to the indices of its bounds.

Example 2. *Interval $[5, 8]$ of $S = \mathit{biadkfb lhdba}$ is associated with substring $S[5, 8] = \mathit{kfbl}$.*

Of particular interest to us are intervals that are *maximal*, which means that the interval cannot be extended to its left or right without changing its character set. Maximality can be sub-divided into *left-* and *right-maximality*.

Definition 2. *An interval $[i, j]$ in string S is left-maximal iff $i = 1$ or $C(S[i, j]) \neq C(S[i - 1, j])$, it is right-maximal iff $j = |S|$ or $C(S[i, j]) \neq C(S[i, j + 1])$. It is maximal, if its both left- and right-maximal.*

Example 3. *Interval $[1, 6]$ of $S = \mathit{biadkfb lhdba}$ is left-maximal, but not right-maximal. Interval $[1, 7]$ is maximal (and so are many other intervals of S).*

Definition 3. *A pair of intervals $[i, j]$ and $[i', j']$ of two strings S and T , respectively, is common, iff their character sets are identical, i.e., $C(S[i, j]) = C(T[i', j'])$.*

Example 4. $S = \mathit{biadkfb lhdba}$, $T = \mathit{iecdbl fhkbbhea}$, intervals $[4, 11]$ of S and $[4, 11]$ of T are maximal common intervals.

This leads to the following problem statement:

Problem 1. *Given two strings S and T , find all their maximal common intervals.*

Didier's Algorithm

The algorithm implements the following strategy to identify maximal common intervals of two strings S and T :

1. Iterate through each position i of S , fixing i as left bound of all intervals of S that are examined in the current iteration.
2. Identify positions in T that correspond to the character $S[i]$. Mark these positions as (trivial) intervals.
3. Successively move the right bound j of S from i to $|S|$. In each such move, do as follows:
 - a) To ensure right-maximality, move j to the rightmost position such that character set $C(S[i, j])$ has exactly one more character than the character set $C(S[i, j'])$ corresponding to the substring of its preceding position $j' < j$. For further reference, let c be this new character.
 - b) If $S[i - 1] = c$, i.e., $S[i, j]$ is not left-maximal, go to Step 1 and continue with next position $i' = i + 1$.
 - c) Extend all intervals in T to the right and left if c is neighboring their current bounds; merge intervals; or create new intervals only consisting of j .
 - d) Let $T[k, l]$ be such an (updated) interval. Report an interval pair $[i, j], [k, l]$ iff $C(S[i, j]) = C(T[k, l])$.

Note that marked intervals in T are left- and right-maximal by construction.

The new right bound j for Step 3a can be identified in constant time using a table that tracks the leftmost occurrences of characters in string $S[i, |S|]$:

Definition 4. *The rank of a character c in string S is the number of different characters that occur up to and including the leftmost occurrence of c in S .*

The ranks of all characters are stored in a *rank table* called RANK. POS is a table that maps each rank to the leftmost position of the corresponding character in S . If a character does not occur, then its rank is infinite and its position undefined.

Example 5. $S = biadkfbldhba$, the RANK and POS table of S are:

character	a	b	c	d	e	f	g	h	i	j	k	l
RANK	3	1	∞	4	∞	6	∞	8	2	∞	5	7

rank	1	2	3	4	5	6	7	8
POS	1	2	3	4	5	6	8	9

We can now use RANK and POS in Step 3a to determine the next right bound j of S . Let j' be the bound before the move, then $S[j' + 1]$ is the new character c . Then $j = \text{POS}[\text{RANK}[c] + 1] - 1$ is the next right-maximal bound in S .

We now construct a data structure very similar to the set of intervals associated to vector R of a generator of common intervals in permutations.

Definition 5. *Given table RANK, the rank interval of a position p in T is the largest interval $[k, l]$, $k \leq p \leq l$, such that $\max(\{\text{RANK}[c] \mid c \text{ in } C(T[k, l])\}) = \text{RANK}[T[p]]$.*

Didier's Algorithm pre-computes all rank intervals and stores them in a table called INT. Rank intervals enable constant time extension of each marked interval in Step 3c of the strategy.

One last challenge remains unsolved: How can the equality of character sets $C(T[k, l])$ and $C(S[i, j])$ in Step 3d be tested efficiently? For this, Didier uses another concept, called the *rank-nearest successor*, which is determined by computing *rank distances* between positions of string T :

Definition 6. *Given a table RANK, the rank distance $d(p, p')$ of two positions p, p' of string T is $d(p, p') = \max(\{\text{RANK}[c] \mid c \in C(T[p, p'])\})$.*

Definition 7. *Let p be a position in T and P the set of positions of rank $\text{RANK}[T[p]] + 1$. If P is empty, p does not have a successor. Otherwise, let p_{left} be the greatest position in P smaller than p , if it exists, and p_{right} be the smallest position in P greater than p , if it exists. If only one of both exists, the rank-nearest successor is p_{left} or p_{right} , respectively. If both exist, the rank-nearest successor is*

- (i) p_{left} if $d(p, p_{\text{left}}) \leq d(p, p_{\text{right}})$, or
- (ii) p_{right} otherwise.

The path of rank-nearest successors spans a forest in T , which Didier's Algorithm follows from the leaves to the roots. Following this path is ingenious, because it is clear that all characters the corresponding interval $[i, j]$ in S will be visited one by one. Assume we followed a path of rank-nearest successors and arrived at position p_r . To test whether the rank interval $[p, p']$ of position p_r is a common interval, it suffices to check whether all positions previously encountered on this path are within the bounds of $[p, p']$.

More formally, let p_r be the r -th position of a path that follows rank-nearest successors from a position p_1 of string T with $\text{RANK}[T[p_1]] = 1$ to its rank-nearest successor p_2 with $\text{RANK}[T[p_2]] = 2$, etc, to position p_r with $\text{RANK}[T[p_r]] = r$. Further, let p_k and p_l the outermost positions observed on this path, i.e., $p_l = \min(\{p_1, \dots, p_r\})$ and $p_k = \max(\{p_1, \dots, p_r\})$. The rank interval $[p, p']$ of position p_r has the same character set as $C(S[i, j])$, where $j := \text{POS}[r + 1] - 1$, iff $[p_k, p_l] \subseteq [p, p']$.

Didier precomputes rank-nearest successors and stores them in a table called SUCC. The full algorithm is described in pseudo-code in Algorithm 1.

Algorithm 1 Didier's Algorithm

Input: Two strings S and T **Output:** List of all maximal common intervals between S and T

```
1: for  $i = 1, \dots, |S|$  do
2:   Compute tables POS and RANK w.r.t.  $S[i, |S|]$ 
3:   Compute table of intervals INT for  $T$ 
4:   Compute table SUCC for  $T$ 
5:   Initialize list LIST with positions in  $T$  of rank 1 and their corresponding rank intervals
6:   while LIST is not empty do
7:      $(y, [p_k, p_l]) \leftarrow$  the first element of LIST verifying  $[p_k, p_l] \subseteq \text{INT}[y]$ 
8:      $r_y \leftarrow \text{RANK}[T[y]]$ 
9:      $j \leftarrow \text{POS}[r_y + 1] - 1$ 
10:    // test if there are locations to output
11:    if  $i = 1$  or  $\text{RANK}[S[i - 1]] > r_y$  then
12:      report  $([i, j], \text{INT}[y])$ 
13:       $\text{previous} \leftarrow \text{INT}[y]$ 
14:      for each following element  $(y, [p_k, p_l])$  in LIST do
15:        if  $[p_k, p_l] \subseteq \text{INT}[y]$  and  $\text{INT}[y] \neq \text{previous}$  then
16:          report  $([i, j], \text{INT}[y])$ 
17:           $\text{previous} \leftarrow \text{INT}[y]$ 
18:        end if
19:      end for
20:    // compute the next level
21:    for each element  $(y, [p_k, p_l])$  in LIST do
22:      remove  $(y, [p_k, p_l])$  from LIST
23:      if  $\text{SUCC}[y]$  exists and  $y$  is the leftmost index with minimal rank distance to  $\text{SUCC}[y]$  among
24:      positions of the list having the same successor then
25:         $y \leftarrow \text{SUCC}[y]$ 
26:        // update path bounds
27:        if  $y < p_k$  then
28:           $p_k \leftarrow y$ 
29:        end if
30:        if  $y > p_l$  then
31:           $p_l \leftarrow y$ 
32:        end if
33:        append  $(y, [p_k, p_l])$  to LIST
34:      end if
35:    end for
36:  end while
37: end for
```

Runtime. Each position i in string S is used once as left bound, hence the outer for loop contributes $O(|S|)$ time. For each fixed i , table RANK can be computed in $O(|S|)$ time, whereas tables INT and SUCC can be computed in $O(|T|)$ time (no proof). The number of positions visited when following all paths of rank-nearest successors in T cannot exceed $|T|$. In each step, tracking and extending the path's bounds and testing whether the path's bounds are within the current rank interval can be performed in constant time. Thus, the algorithm overall takes $O(n^2)$ time, where $n := \max(|S|, |T|)$.