Journal of Discrete Algorithms 34 (2015) 49–53

Contents lists available at ScienceDirect

Journal of Discrete Algorithms

www.elsevier.com/locate/jda

A note on the longest common compatible prefix problem for partial words

M. Crochemore [a,b], C.S. Iliopoulos [a,c], T. Kociumaka [d], M. Kubica [d], A. Langiu [e], J. Radoszewski [d,*], W. Rytter [d], B. Szreder [d], T. Waleń [d]

[a] King's College London, London WC2R 2LS, UK
[b] Université Paris-Est, France
[c] Digital Ecosystems & Business Intelligence Institute, Curtin University of Technology, Perth WA 6845, Australia
[d] Faculty of Mathematics, Informatics and Mechanics, University of Warsaw, ul. Banacha 2, 02-097 Warsaw, Poland
[e] Institute for Coastal Marine Environment of the National Research Council (IAMC-CNR), Unit of Capo Granitola, Via del Faro no. 3, 91021 Granitola, TP, Italy

ARTICLE INFO

Article history:
Available online 22 May 2015

Keywords:
Partial word
Longest common compatible prefix
Longest common prefix
Dynamic programming

ABSTRACT

For a partial word w the longest common compatible prefix of two positions i, j, denoted $lccp(i, j)$, is the largest k such that $w[i, i+k-1]$ and $w[j, j+k-1]$ are compatible. The LCCP problem is to preprocess a partial word in such a way that any query $lccp(i, j)$ about this word can be answered in $O(1)$ time. We present a simple solution to this problem that works for any linearly-sortable alphabet. Our preprocessing is in time $O(n\mu + n)$, where μ is the number of blocks of holes in w.

1. Introduction

A regular word (a string) is a finite sequence of symbols from an alphabet Σ. The notion of partial word is a generalization of the notion of regular word. It may contain occurrences of a special symbol $\diamond$ (a "hole", a don't care symbol), which may represent any symbol of the alphabet. Motivation on partial words and their applications can be found in the book [1].

The longest common compatible prefix (LCCP) problem is a natural generalization into partial words of the longest common prefix (LCP) problem for regular words. For the LCP problem an $O(n)$-preprocessing-time and $O(1)$-query-time solution exists. Recently an efficient algorithm for the LCCP problem has been given by F. Blanchet-Sadri and J. Lazarow [2]. The preprocessing time is $O(nh + n)$, where h is the number of holes in w, and the query time is constant. Their data structure is rather complex. It is based on suffix dags which are a modification of suffix trees and requires Σ to be a fixed alphabet (i.e. $|\Sigma| = O(1)$).

We show a much simpler data structure that requires only $O(n\mu + n)$ construction time and space and also allows constant-time LCCP-queries. Our algorithm is based on alignment techniques and suffix arrays for regular words and works for any integer alphabet (that is, the letters can be treated as integers in a range of size $n^{O(1)}$).

* Corresponding author at: University of Warsaw, ul. Banacha 2, 02-097 Warsaw, Poland.
E-mail addresses: maxime.crochemore@kcl.ac.uk (M. Crochemore), c.iliopoulos@kcl.ac.uk (C.S. Iliopoulos), kociumaka@mimuw.edu.pl (T. Kociumaka), kubica@mimuw.edu.pl (M. Kubica), alessio.langiu@iamc.cnr.it (A. Langiu), jrad@mimuw.edu.pl (J. Radoszewski), rytter@mimuw.edu.pl (W. Rytter), szreder@mimuw.edu.pl (B. Szreder), walen@mimuw.edu.pl (T. Waleń).

http://dx.doi.org/10.1016/j.jda.2015.05.003
1570-8667/

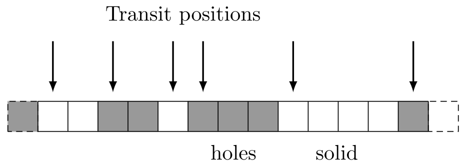

Fig. 1. Illustration of transit positions; $\mu = 3$, $\kappa = 6$. The first and the last symbols are sentinels.

Let w be a partial word of length n. That is, $w = w_1 \dots w_n$, with $w_i \in \Sigma \cup \{\diamond\}$, where Σ is called the alphabet (the set of letters) and $\diamond \notin \Sigma$ denotes a hole. A non-hole position in w is called solid. By h we denote the number of holes in w and by μ we denote the number of blocks of consecutive holes in w.

By $\uparrow$ we denote the compatibility relation: $a \uparrow \diamond$ for any $a \in \Sigma$ and moreover $\uparrow$ is reflexive. The relation $\uparrow$ is extended in a natural letter-by-letter manner to partial words of the same length. Note that $\uparrow$ is not transitive: $a \uparrow \diamond$ and $\diamond \uparrow b$ whereas $a \not\uparrow b$ for any letters $a \neq b$.

Example 1. Let $w = a\,b \diamond \diamond a \diamond \diamond \diamond b\,c\,a\,b \diamond$. There are 7 solid positions in w, $h = 6$ and $\mu = 3$.

By $w[i, j]$ we denote the subword $w_i \dots w_j$. If $j < i$ then $w[i, j] = \varepsilon$, the empty word. The longest common compatible prefix of two positions i, j, denoted $lccp(i, j)$, is the largest $k \geq 0$ such that $i + k - 1, j + k - 1 \leq n$ and $w[i, i + k - 1] \uparrow w[j, j + k - 1]$.

Example 2. For the word w from Example 1, we have $lccp(2, 9) = 3$, $lccp(1, 2) = 0$, $lccp(3, 6) = 8$.

We tackle the following problem.

LCCP Problem
Input: A partial word w of length n over an integer alphabet.
Queries: $lccp(i, j)$ for $1 \leq i, j \leq n$.

2. Data structure

We denote the set of all positions in w by $[n] = \{1, \dots, n\}$. By *type*(i) we mean *hole* or *solid* depending on the type of w_i. We add two sentinel symbols, w_0 and w_{n+1}. We set $w_0 = \diamond$ if w_1 is solid or $w_0 = a \in \Sigma$ if w_1 is a hole. Similarly, we set $w_{n+1} = \diamond$ if w_n is solid or $w_{n+1} = a \in \Sigma$ if w_n is a hole.

A position $i \in [n]$ in w is called *transit* if it is a hole directly preceded by a solid position or a solid position directly preceded by a hole. Let all transit positions in w be

$$TRANSIT = \{i_1, i_2, \dots, i_\kappa\}.$$

Note that $i_1 = 1$ and that $\kappa \leq 2\mu + 1$.

Example 3. Let $w = ab \diamond \diamond a \diamond \diamond \diamond bcab \diamond$. Then $TRANSIT = \{1, 3, 5, 6, 9, 13\}$; see also Fig. 1.

Our data structure consists of two parts:

(1) a data structure of size $O(n)$ allowing to compute in $O(1)$ time the length of the *longest common prefix*, denoted $lcp(i, j)$, between any two positions in the regular word $\hat{w}$, which results from w by treating holes as solid symbols.
(2) a $n \times \kappa$ table

$$LCCP[i, j] = lccp(i, j) \quad \text{for } i \in [n],\ j \in TRANSIT.$$

For convenience we extend this table with $LCCP[i, n + 1] = LCCP[n + 1, i] = 0$ for $i \in \{1, \dots, n + 1\}$.

The data structure (1) consists of the suffix array for $\hat{w}$ and Range Minimum Query data structure. A suffix array is composed of three tables: *SUF*, *RANK* and *LCP*. The *SUF* table stores the list of positions in $\hat{w}$ sorted according to the increasing lexicographic order of suffixes starting at these positions. The *LCP* array stores the lengths of the longest common prefixes of consecutive suffixes in *SUF*. We have $LCP[1] = -1$ and, for $1 < i \leq n$, we have:

$$LCP[i] = lcp(SUF[i-1], SUF[i]).$$

Finally, the *RANK* table is an inverse of the *SUF* table:

$$SUF[RANK[i]] = i \quad \text{for } i \in [n].$$

All tables comprising the suffix array for a word over a linearly-sortable alphabet can be constructed in $O(n)$ time [3,5,6].

The Range Minimum Query data structure (RMQ, in short) is constructed for an array $A[1..n]$ of integers. This array is preprocessed to answer the following form of queries: for an interval $[i, j]$ (where $1 \le i \le j \le n$), find the minimum value $A[k]$ for $i \le k \le j$. The best known RMQ data structures have $O(n)$ preprocessing time and $O(1)$ query time [4].

To compute $lcp(i, j)$ for $i \neq j$ we use a classic combination of the two data structures, see also [3]. Let x be $\min(RANK[i], RANK[j])$ and y be $\max(RANK[i], RANK[j])$. Then:

$$lcp(i, j) = \min\{LCP[x+1], LCP[x+2], \dots, LCP[y]\}.$$

This value can be computed in $O(1)$ time provided that RMQ data structure for the table *LCP* is given.

3. The algorithms

We present two algorithms: one for fast LCCP queries and one for preprocessing of *LCCP* table (part (2) of the data structure).

For $i \in [n]$ define

$$NextChange[i] = \min\{k > 0 : type(i+k) \neq type(i)\}.$$

Clearly the *NextChange* table can be computed in $O(n)$ time. We denote

$$next(i, j) = \min(NextChange[i], NextChange[j]).$$

Observation 1. Let $i, j \in [n]$ and $d = next(i, j)$. Then $i + d \in TRANSIT \cup \{n+1\}$ or $j + d \in TRANSIT \cup \{n+1\}$.

Lemma 2. *Assume we have the data structures from points (1)–(2) above. Then lccp(i, j) for any $i, j \in [n]$ can computed in $O(1)$ time.*

Proof. The algorithm is shown on the following pseudocode.

```
Algorithm LCCP-QUERY(w, i, j)
    d := next(i, j);
    k := lcp(i, j);   { Upper bound on the result }
    if type(w_i) ≠ solid or type(w_j) ≠ solid or k ≥ d then
        { We have: w[i, i+d−1] ↑ w[j, j+d−1] }
        if j + d ∈ TRANSIT ∪ {n+1} then
            return d + LCCP[i+d, j+d];
        else
            return d + LCCP[j+d, i+d];
    else
        return k;
```

Let us explain the algorithm. If any of the positions i, j is a hole then certainly $w[i, i+d-1] \uparrow w[j, j+d-1]$. In this case computation of the result is based on either $LCCP[i+d, j+d]$ or $LCCP[j+d, i+d]$. Observation 1 guarantees that one of those two values is present in the table.

Otherwise, both i, j are solid. If $k = lcp(i, j) < d$ then $w[i, i+k-1] = w[j, j+k-1]$, $w[i+k] \neq w[j+k]$ and both $w[i+k]$ and $w[j+k]$ are solid. Consequently the result is k. Otherwise $w[i, i+d-1] \uparrow w[j, j+d-1]$ and again the result is based on either $LCCP[i+d, j+d]$ or $LCCP[j+d, i+d]$. $\square$

Theorem 3. *Let w be a partial word of length n over an integer alphabet. We can preprocess w in $O(n\mu + n)$ time to enable lccp-queries in constant time.*

Proof. The data structure (1) for *lcp*-queries is constructed in $O(n)$ time from the suffix array for $\hat{w}$ and the RMQ data structure for the *LCP* table of $\hat{w}$.

The construction of part (2) is shown in the following LCCP-Preprocess algorithm. This algorithm is based on the dynamic programming technique. It traverses all $(i, j) \in ([n] \times \mathit{TRANSIT}) \cup (\mathit{TRANSIT} \times [n])$ in decreasing lexicographical order and for each such pair it computes either $\mathit{LCCP}[i, j]$ or $\mathit{LCCP}[j, i]$ using the LCCP-Query algorithm from Lemma 2.

The query algorithm computes the result using the value of $\mathit{LCCP}[i+d, j+d]$ (or $\mathit{LCCP}[j+d, i+d]$) for $d = \mathit{next}(i, j) > 0$. By Observation 1,

$$i + d \in \mathit{TRANSIT} \cup \{n+1\} \quad \text{or} \quad j + d \in \mathit{TRANSIT} \cup \{n+1\}.$$

If any of these is $n+1$, the corresponding *LCCP* value is set in the initialization stage. Otherwise, the pair $(i+d, j+d)$ precedes the pair (i, j) in the considered order and therefore the value $\mathit{LCCP}[i+d, j+d]$ (or $\mathit{LCCP}[j+d, i+d]$) has already been computed.

```
Algorithm LCCP-PREPROCESS(w)
  { Initialization }
  for i := 1 to n + 1 do  LCCP[i, n + 1] := LCCP[n + 1, i] := 0;

  { Main loop with 2nκ iterations }
  foreach (i, j) ∈ [n]^2, i ∈ TRANSIT or j ∈ TRANSIT
          in decreasing lexicographical order do
      if j ∈ TRANSIT then
          LCCP[i, j] := LCCP-QUERY(w, i, j);
      if i ∈ TRANSIT then
          LCCP[j, i] := LCCP-QUERY(w, i, j);
```

Each LCCP-Query works in $O(1)$ time, therefore the whole algorithm works in $O(n\kappa + n) = O(n\mu + n)$ time. Note that, due to the specific order of pairs of positions considered in the algorithm, a single element of the table can be computed more than once. However, this does not influence the asymptotic order of time complexity.

Using the two data structures, by Lemma 2 we can answer $\mathit{lccp}(i, j)$ queries in $O(1)$ time. □

Example 4. Let $w = ab \diamond \diamond a \diamond \diamond \diamond bcab \diamond$. Then $[n] = \{1, \dots, 13\}$, $\mathit{TRANSIT} = \{1, 3, 5, 6, 9, 13\}$ and the pairs of positions (i, j) considered in the algorithm LCCP-Preprocess(w) are:

$$(13,13), (13,12), \dots, (13,1), (12,13), (12,9), (12,6), (12,5), (12,3), (12,1),$$
$$(11,13), (11,9), (11,6), (11,5), (11,3), (11,1), (10,13), (10,9), (10,6), (10,5),$$
$$(10,3), (10,1), (9,13), (9,12), \dots, (9,1), \dots$$

The resulting *LCCP* table is as follows.

j	w_i												
	a	b	$\diamond$	$\diamond$	a	$\diamond$	$\diamond$	$\diamond$	b	c	a	b	$\diamond$
1	13	0	8	1	4	4	7	4	0	0	3	0	1
3	8	7	11	6	6	8	2	2	5	2	3	2	1
5	4	0	6	5	9	4	4	6	0	0	3	0	1
6	4	3	8	5	4	8	3	3	5	4	3	2	1
9	0	3	5	1	0	5	2	1	5	0	0	2	1
13	1	1	1	1	1	1	1	1	1	1	1	1	1

4. Conclusions

We have presented a simple data structure for constant-time answering of longest common compatible prefix queries in a partial word. The data structure uses small space provided that μ, that is, the number of *blocks* of holes in the partial word, is small. An open problem is to try to design an efficient data structure for the case when μ is large, e.g., $\mu = \Omega(n)$, where n is the length of the partial word.

Acknowledgements

Tomasz Kociumaka is supported by Polish budget funds for science in 2013–2017 as a research project under the 'Diamond Grant' program (Ministry of Science and Higher Education, Republic of Poland, grant number DI2012 01794). Jakub Radoszewski receives financial support of Foundation For Polish Science and is supported by the Polish Ministry of Science and Higher Education under the 'Iuventus Plus' program in 2015–2016 grant number 0392/IP3/2015/73. Wojciech Rytter is supported by grant number NCN2014/13/B/ST6/00770 of the National Science Centre.

References

[1] F. Blanchet-Sadri, Algorithmic Combinatorics on Partial Words, Chapman & Hall/CRC Press, Boca Raton, FL, 2008.
[2] F. Blanchet-Sadri, J. Lazarow, Suffix trees for partial words and the longest common compatible prefix problem, in: A.H. Dediu, C. Martín-Vide, B. Truthe (Eds.), LATA, in: Lecture Notes in Computer Science, vol. 7810, Springer, 2013, pp. 165–176.
[3] M. Crochemore, C. Hancart, T. Lecroq, Algorithms on Strings, Cambridge University Press, 2007.
[4] D. Harel, R.E. Tarjan, Fast algorithms for finding nearest common ancestors, SIAM J. Comput. 13 (2) (1984) 338–355.
[5] J. Kärkkäinen, P. Sanders, Simple linear work suffix array construction, in: J.C.M. Baeten, J.K. Lenstra, J. Parrow, G.J. Woeginger (Eds.), ICALP, in: Lecture Notes in Computer Science, vol. 2719, Springer, 2003, pp. 943–955.
[6] T. Kasai, G. Lee, H. Arimura, S. Arikawa, K. Park, Linear-time longest-common-prefix computation in suffix arrays and its applications, in: A. Amir, G.M. Landau (Eds.), CPM, in: Lecture Notes in Computer Science, vol. 2089, Springer, 2001, pp. 181–192.