A Note on the Longest Common Compatible Prefix Problem for Partial Words

M. Crochemore\textsuperscript{a,b}, C. S. Iliopoulos\textsuperscript{a,c}, T. Kociumaka\textsuperscript{d}, M. Kubica\textsuperscript{d}, A. Langiu\textsuperscript{a}, J. Radoszewski\textsuperscript{d}, W. Rytter\textsuperscript{d,e}, B. Szreder\textsuperscript{d}, T. Waleń\textsuperscript{d,f}

\textsuperscript{a}King’s College London, London WC2R 2LS, UK
\textsuperscript{b}Université Paris-Est, France
\textsuperscript{c}Digital Ecosystems & Business Intelligence Institute, Curtin University of Technology, Perth WA 6845, Australia
\textsuperscript{d}Faculty of Mathematics, Informatics and Mechanics, University of Warsaw, ul. Banacha 2, 02-097 Warsaw, Poland
\textsuperscript{e}Faculty of Mathematics and Computer Science, Copernicus University, ul. Chopina 12/18, 87-100 Toruń, Poland
\textsuperscript{f}Laboratory of Bioinformatics and Protein Engineering, International Institute of Molecular and Cell Biology in Warsaw, Poland

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 $w[i, i + k - 1] \uparrow w[j, j + k - 1]$, where $\uparrow$ is the compatibility relation of partial words (it is not an equivalence relation).

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. It is a natural generalization of the longest common prefix (LCP) problem for regular words, for which an $O(n)$ preprocessing time and $O(1)$ query time solution exists.

Recently an efficient algorithm for this problem has been given by F. Blanchet-Sadri and J. Lazarow (LATA 2013). The preprocessing time was $O(nh + n)$, where $h$ is the number of “holes” in $w$. The algorithm was designed for partial words over a constant alphabet and was quite involved.

We present a simple solution to this problem with slightly better runtime that works for any linearly-sortable alphabet. Our preprocessing is in time $O(n\mu + n)$, where $\mu$ is the number of blocks of holes in $w$. Our algorithm uses ideas from alignment algorithms and dynamic programming.

Keywords: partial word, longest common compatible prefix, longest common prefix, dynamic programming
Let \( w \) be a partial word of length \( n \). That is, \( w = w_1 \ldots w_n \), with \( w_i \in \Sigma \cup \{\diamond\} \), where \( \Sigma \) 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 \( \mu \) 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 \( a \nleftrightarrow b \) for any letters \( a \neq b \).

Motivation on partial words and their applications can be found in the book [1].

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

By \( w[i, j] \) we denote the subword \( w_i \ldots w_j \). The longest common compatible prefix of two positions \( i, j \), denoted \( lccp(i, j) \), is the largest \( k \) such that \( w[i, i+k-1] \uparrow w[j, j+k-1] \).

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

In [2] F. Blanchet-Sadri and J. Lazarow provide a data structure that is constructed in \( O(nh + n) \) time and space and allows computing LCCP for any two positions in \( O(1) \) time. Their data structure is based on suffix dags which are a modification of suffix trees and requires \( \Sigma \) 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 full (regular) words and works for any integer alphabet (that is, the letters can be treated as integers in a range of size \( n^{O(1)} \)).

By \( \text{type}(i) \) we mean \( \text{hole} \) or \( \text{solid} \) depending on the type of \( w_i \). We add a sentinel position: \( w_0 = \diamond \) if \( w_1 \) is solid or \( w_0 = a \in \Sigma \) if \( w_1 \) is a hole. A position 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, see Fig. 1. Enumerate all transit positions \( T = \{i_1, i_2, \ldots, i_\kappa\} \). Note that \( \kappa \leq 2\mu \).

![Transit positions](image)

**Figure 1:** Illustration of transit positions, \( \mu = 3 \), \( \kappa = 6 \).
Example 3. Let \( w = a \circ b c a b \circ a \circ b c a b \circ \). Then \( T = \{1, 3, 5, 6, 9, 13\} \), see also Fig. 1.

Our data structure consists of two parts:

1. a data structure of size \( O(n) \) allowing to answer in \( O(1) \) time the longest common prefix, denoted \( \text{lcp}(i, j) \), between any two positions in the full word \( \hat{w} \), which results from \( w \) by treating holes as solid symbols

2. a \( n \times \mu \) table

\[
LCCP[i, j] = \text{lcp}(i, j) \quad \text{for } i \in \{1, \ldots, n\}, j \in T.
\]

For simplicity we assume \( LCCP[j, i] = LCCP[i, j] \) if \( i \in T, j \in \{1, \ldots, n\} \).

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: \textit{SUF}, \textit{RANK} and \textit{LCP}. The \textit{SUF} table stores the list of positions in \( \hat{w} \) sorted according to the increasing lexicographic order of suffixes starting at these positions. The \textit{LCP} array stores the lengths of the longest common prefixes of consecutive suffixes in \textit{SUF}. We have \( LCP[1] = -1 \) and, for \( 1 < i \leq n \), we have:

\[
LCP[i] = \text{lcp}(\text{SUF}[i - 1], \text{SUF}[i]).
\]

Finally, the \textit{RANK} table is an inverse of the \textit{SUF} table:

\[
\text{SUF}[\text{RANK}[i]] = i \quad \text{for } i = 1, 2, \ldots, n.
\]

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

To compute \( \text{lcp}(i, j) \) for \( i \neq j \) we use a classic combination of the two data structures, see also \[5\]. Let \( x \) be \( \min(\text{RANK}[i], \text{RANK}[j]) \) and \( y \) be \( \max(\text{RANK}[i], \text{RANK}[j]) \). Then:

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

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

For \( i \in \{1, \ldots, n\} \) define

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

If no such \( k \) exists then \( \text{NextChange}[i] = n + 1 - i \). Clearly the \textit{NextChange} table can be computed in \( O(n) \) time. We denote

\[
\text{next}(i, j) = \min(\text{NextChange}[i], \text{NextChange}[j]).
\]
Lemma 1. Assume we have the data structures from points (1)-(2) above. Then \( \text{lccp}(i, j) \) for any \( 1 \leq i, j \leq n \) can be computed in \( O(1) \) time.

Proof. If any of the positions \( i, j \) belongs to \( \mathcal{T} \) then we simply use the \( \text{LCCP} \) table. Otherwise we have two cases.

If any of the positions \( i, j \) is a hole then the result is \( d + \text{lccp}(i + d, j + d) \), where \( d = \text{next}(i, j) \).

Otherwise, both \( i, j \) are solid. Let \( k = \text{lcp}(i, j) \). The result is \( d + \text{lccp}(i + d, j + d) \) if \( k \geq d \) or \( k \) otherwise.

\[ \square \]

\textbf{Algorithm LCCP-Query}(w, i, j)
\begin{align*}
d &:= \text{next}(i, j); k := \text{lcp}(i, j);
\text{if } \text{type}(w_i) \neq \text{solid} \text{ or } \text{type}(w_j) \neq \text{solid} \text{ or } k \geq d \text{ then}
\text{return } d + \text{lccp}(i + d, j + d);
\text{else return } k
\end{align*}

Theorem 2. 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 \( \text{lccp} \)-queries in constant time.

Proof. The data structure (1) for \( \text{lcp} \)-queries is constructed in \( O(n) \) time from the suffix array for \( \hat{w} \) and the RMQ data structure for the \( \text{LCP} \) table. The construction of the data structure (2) is shown in the following \text{LCCP-Preprocess} algorithm. This algorithm is based on the dynamic programming technique and works in \( O(n\mu + n) \) time. Using the two data structures, by Lemma 1 we can answer \( \text{lccp}(i, j) \) queries in \( O(1) \) time. \[ \square \]

\textbf{Algorithm LCCP-Preprocess}(w)
\begin{align*}
\text{for } i := 1 \text{ to } n + 1 \text{ do } & \quad \text{LCCP}[i, n + 1] := 0;
\text{foreach } (i, j) : i \in \{1, \ldots, n\}, j \in \mathcal{T} \text{ in decreasing lex. order do } & \quad \text{LCCP}[i, j] := \text{LCCP-Query}(w, i, j);
\end{align*}

Example 4. Let \( w = a b \diamond a \diamond \diamond b c a b \diamond \). The \( \text{LCCP} \) table computed by the algorithm \text{LCCP-Preprocess}(w) is as follows.

\begin{tabular}{c|cccccccccccc}
\hline
j & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\
\hline
w_j & a & b & \diamond & \diamond & a & \diamond & \diamond & b & c & a & b & \diamond \\
\hline
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 \\
\hline
\end{tabular}
References

[1] F. Blanchet-Sadri. *Algorithmic Combinatorics on Partial Words.* Chapman & Hall/CRC Press, Boca Raton, FL, 2008.

[2] F. Blanchet-Sadri and J. Lazarow. Suffix trees for partial words and the longest common compatible prefix problem. In A. H. Dediu, C. Martín-Vide, and B. Truthe, editors, *LATA,* volume 7810 of *Lecture Notes in Computer Science,* pages 165–176. Springer, 2013.

[3] M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on Strings.* Cambridge University Press, 2007.

[4] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.

[5] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In J. C. M. Baeten, J. K. Lenstra, J. Parrow, and G. J. Woeginger, editors, *ICALP,* volume 2719 of *Lecture Notes in Computer Science,* pages 943–955. Springer, 2003.

[6] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In A. Amir and G. M. Landau, editors, *CPM,* volume 2089 of *Lecture Notes in Computer Science,* pages 181–192. Springer, 2001.