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/

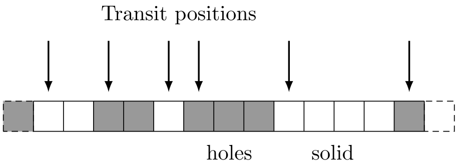

Fig. 1. Illustration of transit positions; $\mu = 3$, $\kappa = 6$. The first and the last symbols are sentinels.

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: