Fast practical computation of the longest common Cartesian substrings of two strings

Simone Faro[1], Thierry Lecroq[2], and Kunsoo Park[3]

[1] University of Catania, Department of Mathematics and Computer Science, Italy
faro@dmi.unict.it
[2] Normandie Univ, UNIROUEN, LITIS, 76000 Rouen, France
thierry.lecroq@univ-rouen.fr
[3] Seoul National University, Seoul, Korea
kpark@theory.snu.ac.kr

Abstract. Cartesian trees have been introduced 40 years ago. They are associated to strings of numbers. They are structured as heap and original strings can be recovered by symmetrical traversal of the trees. The Cartesian tree matching problem appeared very recently. It consists of finding all substrings of a given text which have the same Cartesian tree as that of a given pattern. Here we present two methods for computing the longest common Cartesian substrings of two strings. The first method is a classical linear suffix tree based method. The alternative method runs in quadratic worst case time but is more space economical. Experiments show that the alternative solution runs faster for short strings.

1 Introduction

Cartesian trees have been introduced by Vuillemin [11]. They are associated to strings of numbers. They are structured as heap and original strings can be recovered by symmetrical traversal of the trees. They have many applications including finding patterns in time series data such as share prices in stock markets. It has been shown that they are connected to Lyndon trees [7,3], to Range Minimum Queries [4] or to parallel suffix tree construction [9]. Recently new results on Cartesian pattern matching appear [8,10,6]. It consists of finding substrings of a text that have the same Cartesian tree as a pattern. Recent studies concern finding periods in Cartesian tree matching [1].

In this article we are interested in computing the longest common Cartesian substrings of two strings which means common substrings of maximal length and that have the same Cartesian tree. A usual linear time method for computing longest common substrings for classical strings consists of building the generalized suffix tree of the two strings and the deepest internal nodes (in terms of string depth) having leaves for suffixes of both strings identify longest common substrings. This method can be applied for computing longest common Cartesian substrings of two strings. However the suffix tree has to be built on the parent-distance representation of the two strings and classical suffix tree construction algorithms cannot be used. We propose a quadratic time algorithm for computing the longest Cartesian substrings of two strings that uses only constant extra space in addition to the two strings and their parent-distance representation. Experimental results show that for short strings and in most practical settings our alternative solution is faster than the suffix tree based method.

This article is organized as follows. Section 2 presents the notations and definitions used throughout the rest of the article. Section 3 presents the method for constructing

Simone Faro, Thierry Lecroq, Kunsoo Park: Fast practical computation of the longest common Cartesian substrings of two strings, pp. 48–60.
Proceedings of PSC 2020, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06749-9 © Czech Technical University in Prague, Czech Republic

the Cartesian tree of a string. Section 4 briefly presents the suffix tree based method for computing longest common Cartesian substrings. Section 5 describes our new alternative solution. Section 6 presents experimental results. Section 7 concludes the article.

2 Notations and Definitions

A string is a sequence of symbol drawn from an alphabet Σ, which is a set of integers. We assume that a comparison between any two symbols of the alphabet can be done in constant time. For a string x, $x[i]$ represents the i-th symbol of x, and $x[i\,..\,j]$ represents the substring of x starting from position i and ending at position j. We denote by $x_i = x[1\,..\,i]$ the prefix of x of length i.

Let x be a string of numbers of length m. The Cartesian tree $\mathcal{CT}(x)$ of x is the binary tree where:

- the root corresponds to the index i of the minimal element of x (if there are several occurrences of the minimal element, the leftmost one is chosen);
- the left subtree of the root corresponds to the Cartesian tree of $x[1\,..\,i-1]$;
- the right subtree of the root corresponds to the Cartesian tree of $x[i+1\,..\,m]$.

For simplicity in what follows we will use the symbol $x[i]$ to refer to both the i-th character of x and the node of $\mathcal{CT}(x)$ whose key is i, depending on the context.

The following definition of the *right path* of a binary tree is particularly relevant to this paper.

Definition 1 (Right path). *The right path, $rp(T)$, of a binary tree T is the sequence of nodes encountered starting from the root of the tree and always going right.*

3 Construction of the Cartesian Tree

The construction of the Cartesian tree of a string x of length m can be done by means of an iterative procedure which iterates over the elements of x, proceeding from left to right, and computes the Cartesian tree of x_{i+1} from the Cartesian tree of x_i, for $1 \leq i < m$.

To better describe such approach we observe that the Cartesian tree of $x[1]$ consists of a single node, while $\mathcal{CT}(x_{i+1})$ can be computed from $\mathcal{CT}(x_i)$ by identifying the number of nodes that are on the right path of $\mathcal{CT}(x_i)$ but not on the right path of $\mathcal{CT}(x_{i+1})$.

Going deeper into the details, let $rp(\mathcal{CT}(x_i)) = \langle x[j_1], x[j_2], \ldots, x[j_k] \rangle$ be the right path of $\mathcal{CT}(x_i)$, with $1 \leq k \leq i$ and where $x[j_1]$ is the root of the Cartesian tree and $j_k = i$. Assume that $x[j_u] < x[i+1] < x[j_{u+1}]$, for some $0 \leq u \leq k$. We can distinguish three cases, as depicted in Figure 1:

1. if $u = 0$ then $x[i+1]$ is less than $x[j_1]$, the current root of the tree, and $x[i+1]$ becomes the new root of $\mathcal{CT}(x_{i+1})$;
2. if $0 < u < k$ then $x[i+1]$ is the smallest value on the right of $x[j_u]$ and all elements in the substring $x[j_u+1\,..\,i]$ are greater than $x[i+1]$. Then $x[i+1]$ is inserted as the right child of $x[j_u]$ and the subtree rooted at $x[j_u+1]$ is moved on the left of $x[i+1]$.

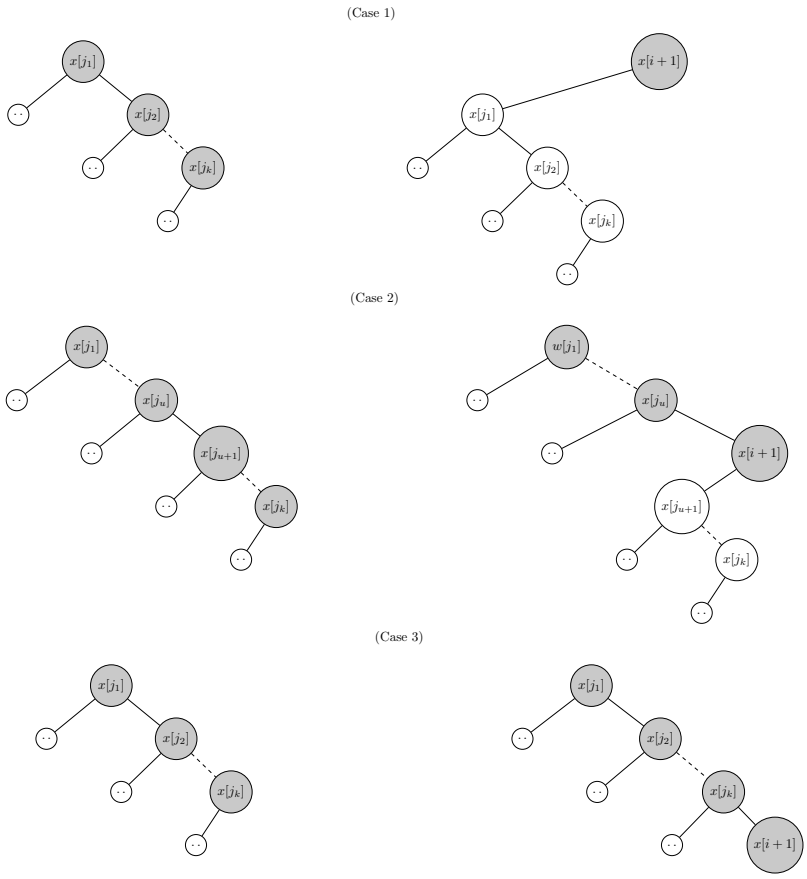

Figure 1. The three cases occurring when computing $\mathcal{CT}(x_{i+1})$ from $\mathcal{CT}(x_i)$. (Case 1) $x[i+1]$ is less than the current root of the tree and it is added as the new root; (Case 2): we have $x[j_u] < x[i+1] < x[j_{u+1}]$; (Case 3): we have $x[i+1]$ is greater than $x[i]$. In all cases $x[i+1]$ becomes the last node of the right path of the tree. Nodes belonging to the right path are filled in gray.

3. if $u = k$ then $x[i+1]$ is greater than $x[j_k]$, the rightmost character in the right path of x_i, so that $x[i+1]$ is added as the right child of $x[j_k]$ in $\mathcal{CT}(x_{i+1})$.

Example 2. Let $x = \langle 4, 22, 3, 5, 2, 23, 15, 1\rangle$ be a numeric sequence of length 8. Figure 3 shows the Cartesian trees computed by such incremental procedure.

Lemma 3 ([5]). *Given a numeric string x, of length m, the Cartesian tree of x can be computed in $O(m)$-time.*

Proof. In order to analyse the time complexity for the computation of the Cartesian tree of a string we refer to the algorithm whose pseudocode, presented in Figure 2, was described in [5].

The for cycle of line 4 is executed $m-1$ times. The while loop of line 7 consists of scanning upward the right path of the tree. Each iteration of this loop decreases the current length of the right path by one and the scanned node will not be scanned again thus the overall number of iterations of the while loop over all the iterations of the for loop is bounded by m. All the other operations can be done in constant time. Therefore the time complexity of the algorithm for building the Cartesian tree of a string of length m is $O(m)$.

```
BUILD-CARTESIAN-TREE(x, m)
 1  root ← NEW-NODE()
 2  ELEMENT(root) ← x[1]
 3  q ← root
 4  for i ← 2 to m do
        ▷ q is the last node of the right path
 5      p ← NEW-NODE()
 6      ELEMENT(p) ← x[i]
 7      while q ≠ nil and x[i] < ELEMENT(q) do
 8          q ← PARENT(q)
 9      if q = nil then
            ▷ Case 1
10          LEFT(p) ← root
11          PARENT(root) ← p
12          root ← p
13      else ▷ Cases 2 and 3
14          if RIGHT(q) ≠ nil then
15              PARENT(RIGHT(q)) ← p
16          LEFT(p) ← RIGHT(q)
17          RIGHT(q) ← p
18          PARENT(p) ← q
19      q ← p
20  return root
```

Figure 2. The iterative procedure BUILD-CARTESIAN-TREE for building the Cartesian tree of a string x of length m. A node of the Cartesian tree has 4 components: PARENT, ELEMENT, LEFT and RIGHT. The function NEW-NODE() creates a new node and initializes its 4 components to NIL.

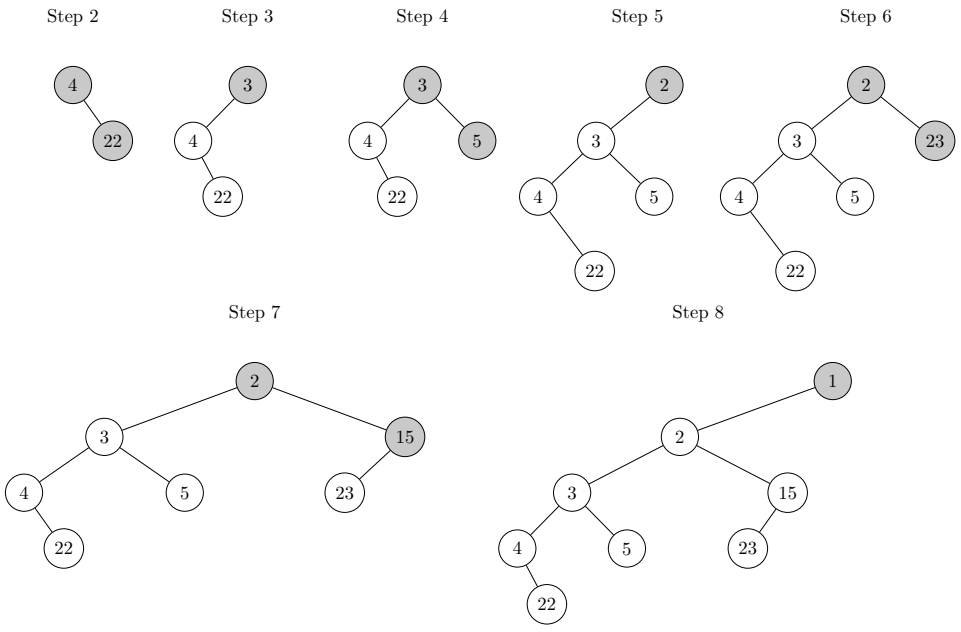

Figure 3. Different steps of the construction of $\mathcal{CT}(x)$ when $x = \langle 4, 22, 3, 5, 2, 23, 15, 1 \rangle$.

For the sake of completeness we point out that the Cartesian tree of a string x can be also computed by iterating over the elements of x and proceeding from right to left (instead of from left to right) by means of a symmetrical procedure.

For realizing the algorithm given in Figure 2 the right path can be implemented as a stack so that there is no need to have a link to its parent for each node of the tree.

Instead of building the Cartesian tree for every position in the text to solve Cartesian tree matching, Park et al. [8] introduced the following representation for a Cartesian tree.

Definition 4 (Parent-distance representation). *The parent-distance representation of a string $x[1\,..\,m]$ is a function PD_x, which is defined as follows:*

$$PD_x(i) = \begin{cases} i - \max_{1 \le j < i}\{j \mid x[j] \le x[i]\} & \text{if such } j \text{ exists} \\ 0 & \text{otherwise.} \end{cases}$$

Example 5. The following table gives the parent-distance representation for $x = \langle 4, 22, 3, 5, 2, 23, 15, 1\rangle$.

i	1	2	3	4	5	6	7	8
$x[i]$	4	22	3	5	2	23	15	1
$PD_x(i)$	0	1	0	1	0	1	2	0

Since the parent-distance representation has a one-to-one mapping to the Cartesian tree [8], it can replace the Cartesian tree without any loss of information. It can be computed and stored in a table in linear time and space using the algorithm given in Figure 4 (see [8]).

```
COMPUTE-PARENT-DISTANCE(x, m)
 1  ST ← empty stack
 2  for i ← 1 to m do
 3     while ST is not empty do
 4        (value, index) ← ST.top
 5        if value ≤ x[i] then
 6           break
 7        ST.pop
 8     if ST is empty then
 9        PD_x[i] ← 0
10     else PD_x[i] ← i − index
11     ST.push((x[i], i))
12  return PD_x
```

Figure 4. Computation of the parent-distance representation for a string x of length m.

4 Longest common Cartesian substrings: suffix tree based solution

The Cartesian suffix tree of a string has to be built on the parent-distance representation of the string. The parent-distance representation of a substring of x can be easily computed as follows (see [8]):

$$PD_{x[i..j]}[k] = \begin{cases} 0 & \text{if } PD_x[i+k-1] \ge k \\ PD_x[i+k-1] & \text{otherwise.} \end{cases}$$

This can be used for getting all the suffixes of the parent-distance representation for building its suffix tree. However classical linear time suffix tree construction algorithms cannot be used because the distinct right context property should hold in order to apply these algorithms, which means that the suffix link of every internal node should point to an explicit node. In other words if $lcp(x[i\,..\,m], x[j\,..\,m]) = \ell$ then $lcp(x[i{+}1\,..\,m], x[j{+}1\,..\,m]) = \ell{-}1$ for $1 \leq i, j \leq m$ where $lcp(u, v)$ is the length of the longest prefix common to two strings u and v. The Cartesian suffix tree does not have the distinct right context property meaning that if $lcp(PD_x[i\,..\,m], PD_x[j\,..\,m]) = \ell$ then $lcp(PD_x[i+1\,..\,m], PD_x[j+1\,..\,m])$ can be greater than $\ell - 1$.

Example 6. With $x = \langle 4, 22, 3, 5, 2, 23, 15, 1\rangle$,

$$lcp(PD_{x[5..8]}, PD_{x[6..8]}) = lcp(\langle 0, 1, 2, 0\rangle, \langle 0, 0, 0\rangle) = 1$$

and

$$lcp(PD_{x[6..8]}, PD_{x[7..8]}) = lcp(\langle 0, 0, 0\rangle, \langle 0, 0\rangle) = 2.$$

A linear time construction algorithm for suffix tree with missing suffix links was first given in [2]. It can be used for building Cartesian suffix trees. These Cartesian suffix trees can be used to compute longest common Cartesian substrings of two strings x and y: for instance, by building the generalized Cartesian suffix tree of PD_x and PD_y. Then the internal nodes with the largest string depth having leaves corresponding to both PD_x and PD_y identify longest common Cartesian substrings of x and y. This can be done during a traversal of the tree. Thus longest common Cartesian substrings of two strings can be computed in linear time and in linear space. The space overhead, in addition to the two strings and their parent-distance representation, is constituted by the generalized suffix tree.

5 Longest common Cartesian substrings: alternative solution

Let x and y be two strings of numbers of length m and n respectively. We are interested in finding the longest substrings of x and y having the same Cartesian tree. We will describe a solution based on dynamic programming. This solution also uses the parent-distance representation. We will show that if $x[i'\,..\,i-1]$ and $y[j'\,..\,j-1]$ are the longest suffixes of $x[1\,..\,i-1]$ and $y[1\,..\,j-1]$ having the same Cartesian tree then the longest suffixes of $x[1\,..\,i]$ and $y[1\,..\,j]$ having the same Cartesian tree can easily be computed. Let us first state that if two substrings have the same parent-distance so have their suffixes.

Fact 7 *If* $PD_{x[i'..i]} = PD_{y[j'..j]}$ *then* $PD_{x[i-\ell..i]} = PD_{y[j-\ell..j]}$ *for* $0 \leq \ell < i - i'$.

Proof. Let $0 \leq \ell < i - i'$ and $i - \ell - i' + 1 < k < i - i'$, then only two cases have to be considered:

1. $PD_{x[i'..i]}[k] = PD_{y[j'..j]}[k] < k - i + \ell$ then $PD_{x[i-\ell..i]}[k - i + \ell + i'] = PD_{x[i'..i]}[k] = PD_{y[j'..j]}[k] = PD_{y[j-\ell..j]}[k - i + \ell + i']$ (see Figure 5) or
2. $PD_{x[i'..i]}[k] = PD_{y[j'..j]}[k] \geq k{-}i{+}\ell$ then $PD_{x[i-\ell..i]}[k{-}i{+}\ell{+}i'] = 0 = PD_{y[j-\ell..j]}[k{-}i + \ell + i']$ (see Figure 6).

In both cases $PD_{x[i-\ell..i]}[p] = PD_{y[j-\ell..j]}[p]$ for $1 \leq p \leq \ell + 1$ thus $PD_{x[i-\ell..i]} = PD_{y[j-\ell..j]}$.

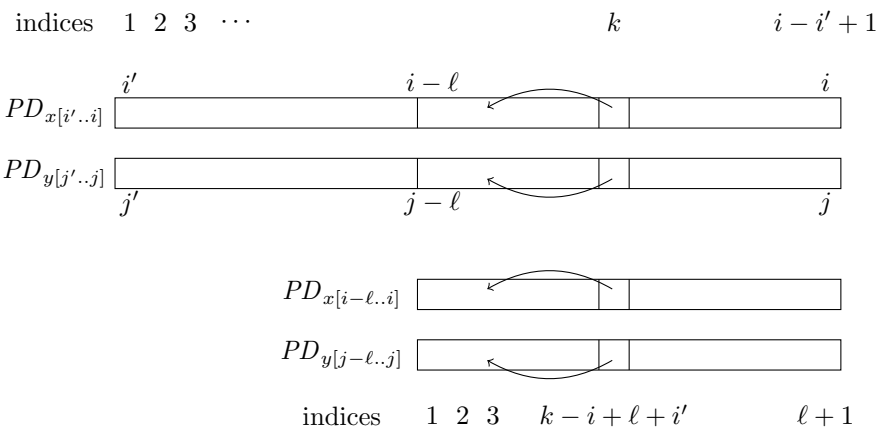

Figure 5. $PD_{x[i'..i]}[k] = PD_{y[j'..j]}[k] < k - i + \ell$

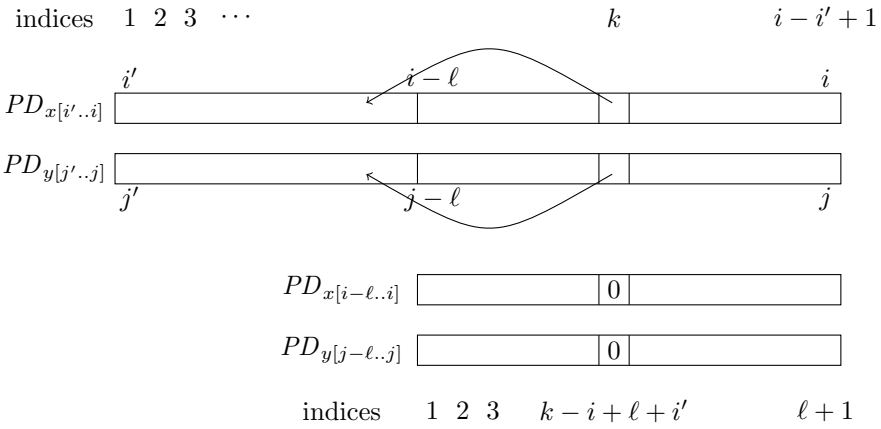

Figure 6. $PD_{x[i'..i]}[k] = PD_{y[j'..j]}[k] \geq k - i + \ell$

We can now state the next lemma.

Lemma 8. *Let $x[i'..i-1]$ and $y[j'..j-1]$ be the longest suffixes of $x[1..i-1]$ and $y[1..j-1]$ having the same Cartesian tree. Let $\ell_x = PD_{x[i'..i]}[i - i' + 1]$, $\ell_y = PD_{y[j'..j]}[j - j' + 1]$ and $\ell = \min\{\ell_x, \ell_y\}$.*

The longest suffixes of $x[1..i]$ and $y[1..j]$ having the same Cartesian tree are:

1. *$x[i'..i]$ and $y[j'..j]$ if $\ell_x = \ell_y$;*
2. *$x[i - \ell_y + 1..i]$ and $y[j - \ell_y + 1..j]$ if $\ell_x \neq \ell_y$ and $\ell_x = 0$;*
3. *$x[i - \ell_x + 1..i]$ and $y[j - \ell_x + 1..j]$ if $\ell_x \neq \ell_y$ and $\ell_y = 0$;*
4. *$x[i - \ell + 1..i]$ and $y[j - \ell + 1..j]$ if $\ell_x \neq \ell_y$ and $\ell_x \neq 0$ and $\ell_y \neq 0$.*

Proof. If $x[i'..i-1]$ and $y[j'..j-1]$ have the same Cartesian tree then $PD_{x[i'..i-1]} = PD_{y[j'..j-1]}$. We will detail the 4 cases:

1. If $\ell_x = \ell_y$ then $PD_{x[i'..i]} = PD_{y[j'..j]}$ and thus $x[i'..i]$ and $y[j'..j]$ have the same Cartesian tree. Thus $x[i'..i]$ and $y[j'..j]$ are the longest suffixes of $x[1..i]$ and $y[1..j]$ having the same Cartesian tree. Longer suffixes with the same Cartesian tree would contradict the maximality of the length of $x[i'..i-1]$ and $y[j'..j-1]$.

2. If $\ell_x \neq \ell_y$ and $\ell_x = 0$ then $PD_{y[k..j]}[j-k+1] = \ell_y \neq \ell_x$ for $j' \leq k \leq j - \ell_y$ and $PD_{y[j-\ell_y+1..j]}[\ell_y] = 0 = \ell_x$. Thus by Fact 7, $x[i-\ell_y+1\,..\,i]$ and $y[j-\ell_y+1\,..\,j]$ are the longest suffixes of $x[1\,..\,i]$ and $y[1\,..\,j]$ having the same Cartesian tree.
3. If $\ell_x \neq \ell_y$ and $\ell_y = 0$ then $PD_{x[k..i]}[i-k+1] = \ell_x \neq \ell_y$ for $i' \leq k \leq i - \ell_x$ and $PD_{x[i-\ell_x+1..i]}[\ell_x] = 0 = \ell_y$. Thus by Fact 7, $x[i-\ell_x+1\,..\,i]$ and $y[j-\ell_x+1\,..\,j]$ are the longest suffixes of $x[1\,..\,i]$ and $y[1\,..\,j]$ having the same Cartesian tree.
4. If $\ell_x \neq \ell_y$ and $\ell_x \neq 0$ and $\ell_y \neq 0$ then $PD_{x[k..i]}[i-k+1] \neq PD_{y[k'..j]}[j-k'+1]$ for $i' \leq k \leq i - \ell$ and for $j' \leq k' \leq j - \ell$ and $PD_{x[i-\ell+1..i]}[\ell] = 0 = PD_{y[j-\ell+1..j]}[\ell]$. Thus by Fact 7, $x[i-\ell+1\,..\,i]$ and $y[j-\ell+1\,..\,j]$ are the longest suffixes of $x[1\,..\,i]$ and $y[1\,..\,j]$ having the same Cartesian tree.

A diagonal d corresponds to a pair of factors $x[i\,..\,s]$ and $y[j\,..\,t]$ such $1 \leq i \leq s \leq m$, $1 \leq j \leq t \leq n$, $s - i = t - j$ and $d = j - i$. Since factors of length 1 always constitute common Cartesian substrings between two strings, the algorithm Compute-Longest-Cartesian-Substring given in Figure 7 processes diagonals from $-m+2$ to $n-2$. For each diagonals it uses 4 indices i, i', j and j' to compare $x[i'\,..\,i]$ and $y[j'\,..\,j]$ starting with the first factors of length 2 of the current diagonal. It updates indices i, i', j and j' according to Lemma 8. It only computes the length ℓ of the longest common Cartesian substrings of x and y but could easily be computed to report two positions i in x and j in y such that $x[i\,..\,i+\ell-1]$ and $y[j\,..\,j+\ell-1]$ have the same Cartesian tree with the same complexities.

Theorem 9. *Given two strings x and y of numbers of length m and n respectively, the longest substrings of x and y having the same Cartesian tree can be computed in time $O(mn)$ and in space $O(m+n)$.*

Proof. Given x and y, the parent-distance representations PD_x and PD_y can be computed in space and time $O(m)$ and $O(n)$ respectively. Then the results of Lemma 8 can be applied on any pair of ending positions of substrings of x and y. There are $O(mn)$ such pairs and the computation for one pair takes constant time if the result for the correct previous pair is available. By performing the computation diagonal-wise the result follows.

Example 10. $x = \langle 70, 84, 63, 74, 86, 97\rangle$ and $y = \langle 50, 83, 76, 39, 90, 67, 1, 6\rangle$. Then $PD_x = \langle 0, 1, 0, 1, 1, 1\rangle$ and $PD_y = \langle 0, 1, 2, 0, 1, 2, 0, 1\rangle$. Let us look at starting positions $i' = 3$ in x and $j' = 4$ in y:

- $PD_{x[3..3]}[1] = 0$ and $PD_{y[4..4]}[1] = 0$
- $PD_{x[3..4]}[2] = 1$ and $PD_{y[4..5]}[2] = 1$
- $PD_{x[3..5]}[3] = 1$ and $PD_{y[4..6]}[3] = 2$ then i' becomes 5 and j' becomes 6
- $PD_{x[5..6]}[2] = 1$ and $PD_{y[6..7]}[2] = 0$ then i' becomes 6 and j' becomes 7

thus the longest Cartesian substring of $x[3\,..\,6]$ and $y[4\,..\,7]$ has length 2.

6 Experiments

The two methods have been implemented in C programming language. The experiments were performed on a computer running MacOS 10.12.6 with an Intel Core i5 1.3 GHz processor and 4GB RAM. We want to compute the length of the longest common Cartesian substring of x of length m and of y of length n. Assume w.l.o.g. that $m < n$.

```
COMPUTE-LONGEST-CARTESIAN-SUBSTRING(x, m, y, n)
 1  PD_x ← COMPUTE-PARENT-DISTANCE(x, m)
 2  PD_y ← COMPUTE-PARENT-DISTANCE(y, n)
 3  maxlength ← 1
 4  for d ← −m + 2 to n − 2 do
      ▷ Initialization of i, j, i′, j′
 5    (i, j) ← (2, 2)
 6    if d < 0 then
 7      i ← −d + 2
 8    else if d > 0 then
 9          j ← d + 2
10    (i′, j′) ← (i − 1, j − 1)
      ▷ Processing diagonal d
11    while i ≤ m and j ≤ n do
12      (ℓ_x, ℓ_y) ← (PD_{x[i′..i]}[i − i′ + 1], PD_{y[j′..j]}[j − j′ + 1])
        ▷ Application of Lemma 8
13      if ℓ_x ≠ ℓ_y then
14        if ℓ_x = 0 then
15          ℓ ← ℓ_y
16        else if ℓ_y = 0 then
17              ℓ ← ℓ_x
18          else ℓ ← min{ℓ_x, ℓ_y}
19        (i′, j′) ← (i − ℓ + 1, j − ℓ + 1)
20      else ▷ maxlength can only increase when ℓ_x = ℓ_y
21        maxlength ← max{maxlength, i − i′ + 1}
22      (i, j) ← (i + 1, j + 1)
23  return maxlength
```

Figure 7. Computation of the length of the longest Cartesian substrings of x of length m and y of length n.

The suffix tree with missing suffix links from [2] has been implemented naively without back propagation and with a simple hashing function. Our implementation is not linear in the worst case but we argue that for short strings it is faster than the linear method which is quite intricate and will lead to an increase in runtime for such short strings. The method for finding the longest Cartesian substring between two strings is then the following: the parent distance representation PD_x and PD_y are computed. Then we construct $PD_{yx} = PD_y\$_1 PD_x\$_2$ where $\$_1$ and $\$_2$ are terminators that do not occur in PD_x and PD_y. Then the suffix tree of PD_{yx} is build for the part corresponding of the longest string y. For the remaining part the suffix tree is scanned to determine the fork where to insert the tail of the current suffix. The string depth of this fork is used to compute the length of the longest common Cartesian substring. The tail is then not inserted since it is not necessary for our purposes and will only lead to a loss of time.

For our alternative solution, main long diagonals are processed first. Shortest diagonals corresponding to prefixes and suffixes of y are processed in a second time. During the computation of a diagonal, when the length of the remaining part of the diagonal is too short and will not possibly contribute to a longest common Cartesian substring, we processed to the next diagonals.

6.1 Random data

We built random strings of integers with 4 different alphabets $[0; 10[$, $[0; 100[$, $[0; 1000[$ and $[0; 10000[$. Then we consider 3 different values for n: 50, 125 and 200. For each value of n we consider 4 values of m: $n/10$, $n/5$, $n/2.5$ and $n/1.25$.

Figure 8 to 10 show the running times of the two solutions: σ denotes the alphabet size, alt the times for the alternative solution and ST the times for the suffix tree based solution. Times are expressed in μseconds.

σ	m	5	10	20	40
10	alt	4283	6189	13621	24711
	ST	36120	33230	46744	56796
100	alt	4963	7210	13702	25644
	ST	40437	39994	46495	58973
1000	alt	4779	7618	14708	25621
	ST	39886	42869	50787	58650
10000	alt	4280	7083	14636	25825
	ST	36074	40522	51166	58184

Figure 8. Experiments with random data for $n = 50$

σ	m	12	24	48	96
10	alt	16589	34500	71790	150029
	ST	98865	95663	108624	132481
100	alt	15577	37178	77217	139539
	ST	96253	106864	121404	128341
1000	alt	14420	35633	72197	141706
	ST	85163	101248	111795	126697
10000	alt	15112	34439	74415	140266
	ST	94438	95248	114660	128906

Figure 9. Experiments with random data for $n = 125$

σ	m	20	40	80	160
10	alt	38677	91698	203764	436527
	ST	127756	150781	181238	261578
100	alt	50687	114319	204344	444743
	ST	197489	209726	196585	280190
1000	alt	46502	104542	190889	393760
	ST	172624	183040	174690	216348
10000	alt	43269	97887	198550	414818
	ST	150034	160674	181007	236281

Figure 10. Experiments with random data for $n = 200$

For $n = 50$ our alternative solution is always fastest than the suffix tree solution. For $n = 125$ our alternative solution is fastest than the suffix tree solution for values of m up to $n/2$. For $n = 200$ our alternative solution is fastest than the suffix tree solution for values of m up to $n/4$.

6.2 Real data

We use data from the COVID-19 pandemic.[1] We extracted the numbers of cases and numbers of deaths for the 15 most infected countries at that time. Data were given in reverse chronological order. We trimmed the data for the tailing runs of 0s and 1s at the end.

Table 1 gives the country numbers and the number of days for cases and deaths for each country.

♯	Country	Cases	Deaths
1	China	102	98
2	France	62	53
3	Germany	62	49
4	Iran	68	68
5	Italy	66	65
6	Korea	72	67
7	Spain	63	54
8	Turkey	45	40
9	UK	105	98
10	USA	67	58
11	Russia	47	32
12	Brazil	54	41
13	Canada	63	42
14	Belgium	57	47
15	Netherlands	60	50

Table 1. Country number, number of cases and deaths considered.

Then we computed the length of the longest common Cartesian substrings for each pair of countries.

	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	11	8	11	11	10	10	13	9	13	14	9	9	10	11
2		9	12	11	8	9	16	11	11	10	9	11	7	12
3			8	11	8	11	9	11	9	8	12	8	8	11
4				11	10	9	12	8	14	12	8	9	8	10
5					9	10	10	13	11	11	10	12	9	10
6						9	7	9	8	8	8	9	7	8
7							9	10	9	9	9	9	11	11
8								9	12	15	8	9	7	12
9									10	10	9	14	12	11
10										14	9	10	9	10
11											9	9	8	10
12												11	10	8
13													9	8
14														9

Figure 11. Length of the longest common Cartesian substring between countries for the number of cases of the COVID-19.

Figure 11 shows the results for the number of cases. These results have been computed in 9381 μs with the Suffix Tree method and in 5904 μs with our alternative method.

[1] We downloaded data from `https://www.ecdc.europa.eu`. on 27th April 2020

	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	9	8	14	7	10	10	10	10	8	10	8	9	13	10
2		10	10	9	7	7	10	7	9	9	9	9	12	14
3			9	9	8	11	9	11	10	8	7	8	10	7
4				8	9	10	13	11	9	9	8	9	10	8
5					8	8	7	10	14	10	11	10	8	11
6						8	7	9	7	9	8	7	8	8
7							10	8	9	9	10	9	9	8
8								7	9	10	7	7	11	8
9									8	9	10	10	9	9
10										8	11	8	11	9
11											9	8	10	9
12												7	7	9
13													9	10
14														8

Figure 12. Length of the longest common Cartesian substring between countries for the number of deaths of the COVID-19.

Figure 12 shows the results for the number of deaths. These results have been computed in 8727 μs with the Suffix Tree method and in 4758 μs with our alternative method.

Again, these results show that for such short strings our alternative solution is faster than the suffix tree based method.

In our experiments, our alternative solution is faster than the suffix tree solution in 38 cases out of 50 cases (on both random and real data); when it is faster, our alternative solution is 3.81 times faster on average (up to 8.43 times for the maximum), and when it is slower, it is 1.3 times slower on average than the suffix tree solution (up to 1.82 for the maximum).

7 Conclusion

In this article we presented a classical suffix tree based solution for computing the longest Cartesian substrings between two strings. This solution is based on the parent-distance representation of the two strings and runs in linear time and linear extra-space in addition to the two strings and their parent-distance representation. Then we proposed an alternative solution based on dynamic programming that runs in quadratic time in the worst case and in constant extra-space in addition to the two strings and their parent-distance representation. We presented experiments that show that our alternative solution runs faster for short strings than the suffix tree based solution. Further works would include the search for the longest approximate common Cartesian substrings between two strings but the notion of approximation in this context has to be defined.

References

1. M. Bataa, S. G. Park, A. Amir, G. M. Landau, and K. Park: *Finding periods in Cartesian tree matching*, in Combinatorial Algorithms - 30th International Workshop, IWOCA 2019, Pisa, Italy, July 23-25, 2019, Proceedings, C. J. Colbourn, R. Grossi, and N. Pisanti, eds., vol. 11638 of Lecture Notes in Computer Science, Springer, 2019, pp. 70–84.
2. R. Cole and R. Hariharan: *Faster suffix tree construction with missing suffix links.* SIAM J. Comput., 33(1) 2003, pp. 26–42.

3. M. Crochemore and L. M. S. Russo: *Cartesian and Lyndon trees.* Theor. Comput. Sci., 806 2020, pp. 1–9.
4. E. D. Demaine, G. M. Landau, and O. Weimann: *On Cartesian trees and Range Minimum Queries.* Algorithmica, 68(3) 2014, pp. 610–625.
5. H. N. Gabow, J. L. Bentley, and R. E. Tarjan: *Scaling and related techniques for geometry problems*, in Proceedings of the 16th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1984, Washington, DC, USA, R. A. DeMillo, ed., ACM, 1984, pp. 135–143.
6. G. Gu, S. Song, S. Faro, T. Lecroq, and K. Park: *Fast multiple pattern Cartesian tree matching*, in WALCOM: Algorithms and Computation - 14th International Conference, WALCOM 2020, Singapore, March 31 - April 2, 2020, Proceedings, M. S. Rahman, K. Sadakane, and W. Sung, eds., vol. 12049 of Lecture Notes in Computer Science, Springer, 2020, pp. 107–119.
7. C. Hohlweg and C. Reutenauer: *Lyndon words, permutations and trees.* Theor. Comput. Sci., 307(1) 2003, pp. 173–178.
8. S. G. Park, A. Amir, G. M. Landau, and K. Park: *Cartesian tree matching and indexing*, in 30th Annual Symposium on Combinatorial Pattern Matching, CPM 2019, June 18-20, 2019, Pisa, Italy., N. Pisanti and S. P. Pissis, eds., vol. 128 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019, pp. 16:1–16:14.
9. J. Shun and G. E. Blelloch: *A simple parallel Cartesian tree algorithm and its application to parallel suffix tree construction.* ACM Trans. Parallel Comput., 1(1) 2014, pp. 8:1–8:20.
10. S. Song, C. Ryu, S. Faro, T. Lecroq, and K. Park: *Fast Cartesian tree matching*, in String Processing and Information Retrieval - 26th International Symposium, SPIRE 2019, Segovia, Spain, October 7-9, 2019, Proceedings, N. R. Brisaboa and S. J. Puglisi, eds., vol. 11811 of Lecture Notes in Computer Science, Springer, 2019, pp. 124–137.
11. J. Vuillemin: *A unifying look at data structures.* Commun. ACM, 23(4) 1980, pp. 229–239.