Journal of Discrete Algorithms 37 (2016) 34–43

Contents lists available at ScienceDirect

Journal of Discrete Algorithms

www.elsevier.com/locate/jda

An improved algorithm for the all-pairs suffix–prefix problem

William H.A. Tustumi [a], Simon Gog [b], Guilherme P. Telles [a], Felipe A. Louza [a,*]

[a] Institute of Computing, University of Campinas, Campinas, SP, Brazil
[b] Institute of Theoretical Informatics, Karlsruhe Institute of Technology, Germany

ARTICLE INFO

Article history:
Available online 28 April 2016

Keywords:
Suffix–prefix matching
Suffix array
LCP array

ABSTRACT

Finding all longest suffix–prefix matches for a collection of strings is known as the all pairs suffix–prefix match problem and its main application is de novo genome assembly. This problem is well studied in stringology and has been solved optimally in 1992 by Gusfield et al. [8] using suffix trees. In 2010, Ohlebusch and Gog [13] proposed an alternative solution based on enhanced suffix arrays which has also optimal time complexity but is faster in practice. In this article, we present another optimal algorithm based on enhanced suffix arrays which further improves the practical running time. Our new solution solves the problem locally for each string, scanning the enhanced suffix array backwards to avoid the processing of suffixes that are no suffix–prefix matching candidates. In an empirical evaluation we observed that the new algorithm is over two times faster and more space-efficient than the method proposed by Ohlebusch and Gog. When compared to Readjoiner [5], a good practical solution, our algorithm is faster for small overlap lengths, in pace with theoretical optimality.

1. Introduction

The all-pairs suffix–prefix matching problem (APSP) is well known in stringology [7,12]. Given a collection of m strings $S^1, S^2, \ldots, S^m$, the APSP is the problem of finding, for all pairs S^i and S^j, the longest suffix of S^i that is a prefix of S^j.

The APSP has an important application in the context of DNA sequencing, where the collection of strings represents fragments coming from the sequencing process, and the reconstruction of the original biological sequence is based on overlaps between these fragments [2].

In 1992, Gusfield et al. [8] presented the first optimal $O(N + m^2)$ time algorithm for the APSP using (generalized) suffix trees [21], where N is the sum of string lengths. This solution is optimal because the input size is N and the output size is m^2. Almost 20 years later, Ohlebusch and Gog [13] proposed algorithm OG, an – in practice better – optimal solution using (generalized) enhanced suffix arrays [11,1], which uses less memory than suffix trees used by Gusfield et al. [8] and has a better locality of memory reference. Their experimental results have shown that the solution is about three times faster than the one that uses suffix trees. To the best of our knowledge, that is the best practical solution so far which is also theoretically optimal.

* Corresponding author.
E-mail addresses: william.tustumi@students.ic.unicamp.br (W.H.A. Tustumi), gog@kit.edu (S. Gog), gpt@ic.unicamp.br (G.P. Telles), louza@ic.unicamp.br (F.A. Louza).

http://dx.doi.org/10.1016/j.jda.2016.04.002
1570-8667/

i	SA	LCP	STR	SA′	$S^{\$}_{\mathsf{SA}[i]}$
1	4	0	1	4	$\$_1$
2	8	0	2	4	$\$_2$
3	11	0	3	3	$\$_3$
4	15	0	4	4	$\$_4$
5	7	0	2	3	a$\$_2$
6	10	1	3	2	a$\$_3$
7	14	1	4	3	a$\$_4$
8	9	1	3	1	aa$\$_3$
9	13	2	4	2	aa$\$_4$
10	1	2	1	1	aac$\$_1$
11	2	1	1	2	ac$\$_1$
12	5	2	2	1	aca$\$_2$
13	3	0	1	3	c$\$_1$
14	6	1	2	2	ca$\$_2$
15	12	2	4	1	caa$\$_4$

Fig. 1. The generalized enhanced suffix array of $\mathcal{S} = \{aac, aca, aa, caa\}$.

Practical solutions which are non-optimal have been proposed in the past few years. A significant reduction of memory consumption was achieved by Rachid et al. [17,16] that presented new space-efficient algorithms using compressed data structures [19,18]. However, their experimental results have shown that their solutions are about 100 times slower in practice than the previous ones [8,13]. More recently, Rachid and Malluhi [15] solved the problem more efficiently using a compact prefix tree. Their algorithm was named SOF.

Finding suffix–prefix matches is a fundamental stage performed by sequence assembler algorithms based on overlaps [3]. Although there are many assemblers, Readjoiner [5] is very fast in practice and has a matching stage separated from the other assembling stages. Readjoiner outperforms SOF in time and space.

In this article we propose an optimal algorithm that is faster and more space-efficient in practice using also (generalized) enhanced suffix arrays. Our algorithm reduces memory consumption by about 15% and is 2.6 times faster (on the average) than algorithm OG. Using a different auxiliary data structure and scanning the enhanced suffix array in another fashion enabled these improvements. Experiments have also shown that this algorithm may be a good practical solution when searching for suffix–prefix overlaps of small length.

The rest of the article is organized as follows. Section 2 introduces concepts and notation, Section 3 presents algorithm OG, Section 4 describes the proposed algorithm and its theoretical analysis, Section 5 presents experimental results and Section 6 concludes the article.

2. Definitions and notation

Let S be a string of length $|S| = n$ over an ordered alphabet of symbols Σ. $S[i]$ denotes the i-th symbol of S, where $1 \leq i \leq n$. $S[i, j] = S[i] \dots S[j]$ denotes a substring of S, for $i \leq j$. In particular, $S[1, j]$ is the prefix of S that ends at position j, and $S[i, n]$ is the suffix of S that starts at position i and is denoted by S_i. We use the symbol $<$ for the lexicographic order relation between strings.

The suffix array SA of a string S is an array of integers in the range 1 to n that gives the lexicographic order of all suffixes of S, such that $S_{\mathsf{SA}[1]} < S_{\mathsf{SA}[2]} < \dots < S_{\mathsf{SA}[n]}$ [11,6]. We denote the position of suffix S_i in SA as $\mathsf{pos}(S_i)$.

The LCP-array is an array of integers that stores the length of the longest common prefix (lcp) of two consecutive suffixes in SA, such that $\mathsf{LCP}[1] = 0$ and $\mathsf{LCP}[i] = \mathsf{lcp}(S_{\mathsf{SA}[i]}, S_{\mathsf{SA}[i-1]})$ for $1 < i \leq n$, where $\mathsf{lcp}(u, v)$ denotes the lcp of strings u and v. Both SA and the LCP-array can be constructed in linear time (see [14,9] for reviews).

The range minimum query (RMQ) with respect to the LCP is the smallest lcp value in an interval of SA. We define $\mathsf{RMQ}(i, j) = \min_{i<k\leq j}\{\mathsf{LCP}[k]\}$. Given a string S of length n and its LCP-array, it is easy to see that $\mathsf{lcp}(S_{\mathsf{SA}[i]}, S_{\mathsf{SA}[j]}) = \mathsf{RMQ}(i, j)$.

Let $\mathcal{S} = \{S^1, S^2, \dots, S^m\}$ be a collection of m strings. The generalized suffix array of $\mathcal{S}$ is the suffix array SA of the concatenated string $S = S^1\$_1 S^2\$_2 \dots S^m\$_m$, where each symbol $\$_i$ is a distinct separator that does not occur in Σ and precedes every symbol in Σ, and $\$_i < \$_j$ if $i < j$. For a suffix $S_{\mathsf{SA}[i]}$ of S, we denote the prefix of $S_{\mathsf{SA}[i]}$ that ends at the first separator $\$_j$ by $S^{\$}_{\mathsf{SA}[i]}$. The total length of the generalized suffix array is $N = m + \Sigma_{l=1}^{m}|S^l|$.

To simplify the notation, we introduce the arrays STR and SA′. STR indicates the string in $\mathcal{S}$ which a suffix came from, formally $\mathsf{STR}[i] = j$ if the suffix $S^{\$}_{\mathsf{SA}[i]}$ ends with symbol $\$_j$. SA′ holds the position of a suffix with respect to the string it came from (up to the separator), defined as $\mathsf{SA}'[i] = k$ if $S^{\$}_{\mathsf{SA}[i]} = S^j_k\$_j$. Taken together, STR and SA′ specify the order of all suffixes in $\mathcal{S}$. We will denote the generalized suffix array enhanced with the arrays STR, SA′ and LCP-array as GESA. Fig. 1 illustrates the GESA of $\mathcal{S} = \{aac, aca, aa, caa\}$.

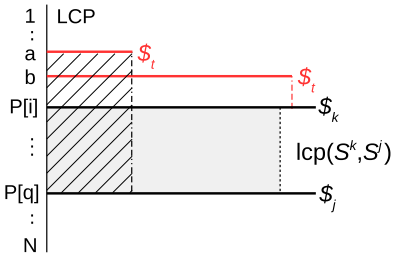

Fig. 2. Illustration of Lemma 2.

3. Related work

The algorithm OG, proposed by Ohlebusch and Gog in 2010 [13], solves the APSP in $O(N + m^2)$ based on the following lemma.

Lemma 1. *Let $i = \text{pos}(S^k)$ be the position of string S^k in* GESA *such that* $\text{SA}'[i] = 1$ *and* $\text{STR}[i] = k$, *i.e.,* $S^{\$}_{\text{SA}[i]} = S^k\$_k$. *If the suffix S^j_p is equal to a prefix of S^k, for $j \neq k$, then either:*

- $\text{pos}(S^j_p) < i$, *or*
- $\text{pos}(S^j_p) = q$, *with* $i < q \leq i + j - k$.

In other words, all suffixes that are a prefix of S^k are either in positions prior to $\text{pos}(S^k)$ or directly succeed $\text{pos}(S^k)$ in the GESA (if they are identical to S^k). The proof of Lemma 1 is quite simple and can be found in [13,12].

OG scans GESA from position $m + 1$ to N (the first m suffixes are skipped because they are separator symbols $\$_i$), keeping track of all suffixes seen so far that are a prefix of the current suffix $S^{\text{STR}[i]}_{\text{SA}'[i]}$. Suffixes that are prefixes are stored in a set of m stacks, one for each string in $\mathcal{S}$. If the current suffix $S^{\text{STR}[i]}_{\text{SA}'[i]}$ is a prefix of the next suffix $S^{\text{STR}[i+1]}_{\text{SA}'[i+1]}$, then the value of $\text{LCP}[i + 1]$ is pushed onto $stack[\text{STR}[i]]$.

Whenever the current suffix is a complete string, that is $\text{SA}'[i] = 1$, the top element of each stack k, $k \neq \text{STR}[i]$, is the length of the longest suffix of S^k that is a prefix of string $S^{\text{STR}[i]}$. These values are used to fill an "overlap" squared matrix Ov, where $\text{Ov}[i, j]$ represents the longest suffix of S^j that is a prefix of S^i. When there are suffixes identical to $S^{\text{STR}[i]}$ that directly succeed position i in GESA, the algorithm scans the next positions verifying while $S^{\text{STR}[i+q]}_{\text{SA}'[i+q]}$ is a prefix of $S^{\text{STR}[i]}$, for increasing values of q.

During the scan, whenever the lcp value of the current suffix decreases, i.e. $\text{LCP}[i + 1] < \text{LCP}[i]$, the algorithm removes all suffixes S^j_p of length ℓ, such that $\text{LCP}[i] \geq \ell \geq \text{LCP}[i + 1] + 1$, from stack j because they are not prefixes of suffix $S^{\text{STR}[i+1]}_{\text{SA}'[i+1]}$. The algorithm uses one list for each possible value of ℓ to find all stacks that have a suffix of length ℓ on top to be removed.

4. Algorithm

At a glance, our algorithm partitions GESA into m blocks, one for each string $S^j \in \mathcal{S}$. Then, it finds all suffixes that are a prefix of S^j scanning its block, and reuses solutions obtained so far (in the previous blocks) to compose the complete solution of S^j.

Let P be an array of length m, such that $\text{P}[i]$ stores the position of the i-th (lexicographically) smallest complete string S^k of $\mathcal{S}$ in GESA, i.e. if $\text{P}[i] = \text{pos}(S^k)$ and $\text{P}[i + 1] = \text{pos}(S^j)$ then $\text{pos}(S^k) < \text{pos}(S^j)$, for $1 \leq i < m$.

Let the interval $B^i = (\text{P}[i - 1], \text{P}[i]]$ be a block of GESA corresponding to the i-th smallest complete string S^k of $\mathcal{S}$, such that B^i starts at the position just after the position of the $(i - 1)$-th smallest complete string of $\mathcal{S}$, and ends in $\text{P}[i] = \text{pos}(S^k)$. We define $\text{P}[i] = m$ if $i < 1$, since the first m suffixes in GESA are separators.

We know that all suffixes that are a prefix of S^k are either in positions prior to $\text{P}[i] = pos(S^k)$ in GESA or are identical to S^k (Lemma 1). Let us concentrate on the former. As a consequence, these suffixes may be in blocks $B^1, B^2, \ldots, B^i$. Furthermore, if two different suffixes of S^j are a prefix of S^k, the longest suffix is the suffix positioned closest to $\text{pos}(S^k)$ in GESA.

Lemma 2. *Let $\text{P}[i] = \text{pos}(S^k)$ and $\text{P}[q] = \text{pos}(S^j)$, with $0 < i < q \leq m$. All suffixes of length $l \leq \text{lcp}(S^k, S^j)$ that are a prefix of S^k are also a prefix of S^j.*

Proof. Let S_p^t be a suffix of length $l = |S^t| - p$, that is a prefix of S^k, such that $\mathsf{pos}(S_p^t) = a$ (see Fig. 2). We know that $\mathsf{lcp}(S_p^t, S^j) = \mathsf{RMQ}(a, \mathsf{P}[q])$ and it is easy to see that $\mathsf{RMQ}(a, \mathsf{P}[q]) = min(\mathsf{RMQ}(a, \mathsf{P}[i]), \mathsf{RMQ}(\mathsf{P}[i], \mathsf{P}[q])) = min(l, \mathsf{lcp}(S^k, S^j))$. Then if $min(l, \mathsf{lcp}(S^k, S^j)) = l$, S_p^t must also be a prefix of S^j. □

As a consequence of Lemma 2 if a suffix of length l is a prefix of S^k and $l > \mathsf{lcp}(S^k, S^j)$, then such suffix is not a prefix of S^j. In Fig. 2 *b* illustrates this case. Note that identical suffixes are also covered by Lemma 2.

We introduce Algorithm 1, which is based on Lemmas 1 and 2 and works as follows. For each block $B^j \in \{B^1, B^2, \ldots, B^m\}$, it scans B^j backwards from position $\mathsf{P}[j]$ to position $\mathsf{P}[j-1]$ (lines 4 to 9). Consider that $\mathsf{pos}(S^k) = \mathsf{P}[j]$ and $\mathsf{pos}(S^t) = \mathsf{P}[j-1]$. To track all suffixes seen so far that are a prefix of another complete string, it uses m local lists L_{local}, and m global lists L_{global}, one for each string $S^i \in \mathcal{S}$. P, L_{global} and L_{local} are used implicitly in the algorithm.

Algorithm 1:

```
   Data: GESA of the collection S = {S^1, S^2 ..., S^m}
   Result: result matrix Ov
 1 for j ← 1 to m do
 2 |   k ← STR[P[j]];
 3 |   ℓ ← ∞;
 4 |   for i ← P[j] to P[j − 1] do
 5 |   |   ℓ ← min(ℓ, LCP[i + 1])
 6 |   |   if |S^{STR[i]}_{SA'[i]}| = ℓ then
 7 |   |   |   insert_at_end(L_local[STR[i]], ℓ)
 8 |   |   end
 9 |   end
10 |   for i ← 1 to m do
11 |   |   while first(L_global[i]) > ℓ do
12 |   |   |   remove_first(L_global[t])
13 |   |   end
14 |   |   L_global[i] ← insert_at_front(L_local[i]);
15 |   |   Ov[k, i] ← first(L_global[i])
16 |   end
17 |   q ← P[j] + 1;
18 |   while |S^{STR[q]}_{SA'[q]}| = LCP[q] and q < N do
19 |   |   Ov[k, STR[q]] ← LCP[q]
20 |   |   q ← q + 1
21 |   end
22 end
```

Whenever the current suffix $S^{\mathsf{STR}[i]}_{\mathsf{SA}'[i]}$ is a prefix of S^k (line 6), i.e. its length is equal to the $\mathsf{lcp}(S^{\mathsf{STR}[i]}_{\mathsf{SA}'[i]}, S^k) = \mathsf{RMQ}(i, \mathsf{P}[j])$, the value of $\mathsf{RMQ}(i, \mathsf{P}[j])$ is inserted at the end of $L_{local}[\mathsf{STR}[i]]$ (line 7), which is computed in ℓ as the minimum lcp value between $\mathsf{LCP}[i+1]$ and $\mathsf{LCP}[\mathsf{P}[j]]$ during the scanning of B^j (line 5). At the end of the first for-loop (line 9), the value in ℓ

i	SA	LCP	STR	SA′	$L_{local}[1]$	$L_{local}[2]$	$L_{local}[3]$	$L_{local}[4]$
8	9	1	3	1	[ ]	[ ]	[ ]	[ ]
7	14	1	4	3	[ ]	[ ]	[ ]	[1]
6	10	1	3	2	[ ]	[ ]	[1]	[1]
5	7	0	2	3	[ ]	[1]	[1]	[1]
10	1	2	1	1	[ ]	[ ]	[ ]	[ ]
9	13	2	4	2	[ ]	[ ]	[ ]	[2]
8	9	1	3	1	[ ]	[ ]	[2]	[2]
12	5	2	2	1	[ ]	[ ]	[ ]	[ ]
11	2	1	1	2	[2]	[ ]	[ ]	[ ]
10	1	2	1	1	[2]	[ ]	[ ]	[ ]
15	12	2	4	1	[ ]	[ ]	[ ]	[ ]
14	6	1	2	2	[ ]	[2]	[ ]	[ ]
13	3	0	1	3	[1]	[2]	[ ]	[ ]
12	5	2	2	1	[1]	[2]	[ ]	[ ]

Fig. 3. GESA and local lists as our algorithm is executed for $\mathcal{S} = \{aac, aca, aa, caa\}$. Blocks are shown separately.

$P[j-1]$	$L_{global}[1]$	$L_{global}[2]$	$L_{global}[3]$	$L_{global}[4]$
5	[ ]	[1]	[1]	[1]
8	[ ]	[1]	[2,1]	[2,1]
10	[2]	[1]	[1]	[1]
12	[1]	[2]	[ ]	[ ]

Fig. 4. Global lists at the end of each block in Fig. 3.

	1	2	3	4
1		1	2	2
2	2		1	1
3		1		2
4	1	2		

Fig. 5. The output matrix Ov for $\mathcal{S} = \{aac, aca, aa, caa\}$.

is equal to RMQ(P[$j-1$], P[j]), which corresponds to lcp(S^t, S^k). Moreover, the values in each local list $L_{local}[r]$ are ordered decreasingly by the length of the suffixes of $S^r \in \mathcal{S}$ that are a prefix of S^k.

However, the longest suffix of a string $S^r \in \mathcal{S}$ that is prefix of S^k may not be positioned in block B^j. In this case, we know that this suffix is in one of the previous blocks $B^{j-1}, B^{j-2}, \ldots, B^1$ seen so far. These suffixes are stored in the global list $L_{global}[r]$, which is composed by the elements inserted in $L_{local}[r]$ and has to be updated in each iteration of the algorithm (lines 10 to 16).

In order to update the global lists, the algorithm removes the suffixes that have length larger than lcp(S^t, S^k) from all local lists (lines 11 to 13). Then the local list $L_{local}[r]$ is added to the beginning of the global list $L_{global}[r]$ (line 14). The first element of each global list $L_{global}[r]$ corresponds to the length of the longest suffix of S^r that is a prefix of S^k and such value is inserted in the "overlap" matrix Ov (line 15). If $L_{global}[r]$ is empty, no suffix of S^r is a prefix of S^k.

Finally, the algorithm deals with the identical suffixes to S^k that directly succeed pos(S^k) in the GESA (lines 17 to 21). It is easy to see that the algorithm scans all suffix–prefix pairs in GESA according to Lemmas 1 and 2, then the "overlap" matrix Ov is completely filled.

As an example, consider the structure in Fig. 3, that represents the GESA and the local lists for $\mathcal{S} = \{aac, aca, aa, caa\}$. The GESA for the same string collection appears in Fig. 1. Extra columns show the contents of each local list at every iteration. Blocks are shown separately in the figure, and lines are disposed in the order that the positions are scanned by the algorithm.

Blocks B^1, B^2, B^3 and B^4 are scanned one by one. For each block $B^j = (P[j-1], P[j])$, positions from $P[j]$ to $P[j-1]+1$ are scanned. Whenever a position i is scanned, $\text{lcp}(S^{\text{STR}[i]}_{\text{SA}'[i]}, S^{\text{STR}[P[j]]}_{\text{SA}'[P[j]]})$ is computed and inserted in list $L_{local}[\text{STR}[i]]$. Note that the evaluation of lcp values may be performed easily as the GESA is scanned.

At the end of each block B^j, the global lists are updated and the contents of local lists are added to the beginning of the global lists. Fig. 4 shows the contents of global lists at the end of each block. Each snapshot is labeled $P[j-1]$, which corresponds to the end of B^j. The first lcp value in each list gives the length of the suffix that matches the complete string that is being processed. The first elements of each global lists are inserted in Ov.

The output matrix Ov is shown in Fig. 5. The case of suffix $S^4_2 = aa$, which is contained in $S^3 = aa\$_3$, is treated after processing the block B^1, when GESA is scanned forward while the succeeding suffixes overlap the current complete string S^3.

Theoretical analysis

An aggregate analysis of the loop at line 4 gives $O(N)$ since each of the N suffixes can be inserted at most once into the local lists. For the loop at line 10, line 12 is executed at most N times overall, since the number of removal operations from the global lists is bounded by the insertions into local lists, while lines 14 and 15 will be executed m^2 times. The loop at line 18 will scan the identical suffixes that follow the current prefix, and because there can be only m such suffixes in $\mathcal{S}$, this loop will be executed in $O(m^2)$ time overall. Thus the algorithm runs in $O(N + m^2)$ time, which is optimal.

The space-complexity is bounded by the memory used by the GESA structure, the m auxiliary global and local lists, and by the output matrix Ov. The GESA uses $O(N)$ space and there are at most N elements added to the m lists. The output matrix Ov uses $O(m^2)$ space. Thus the space-complexity of the algorithm is $O(N + m^2)$.

5. Experimental results

We have compared our algorithm with OG [13] and with Readjoiner [5]. We used real DNA sequences of the EST database from *C. elegans*[1]. The number of strings used in the experiments varies from 10.000 to 300.000 ESTs. We used

[1] Downloaded from http://www.uni-ulm.de/in/theo/research/seqana.html.

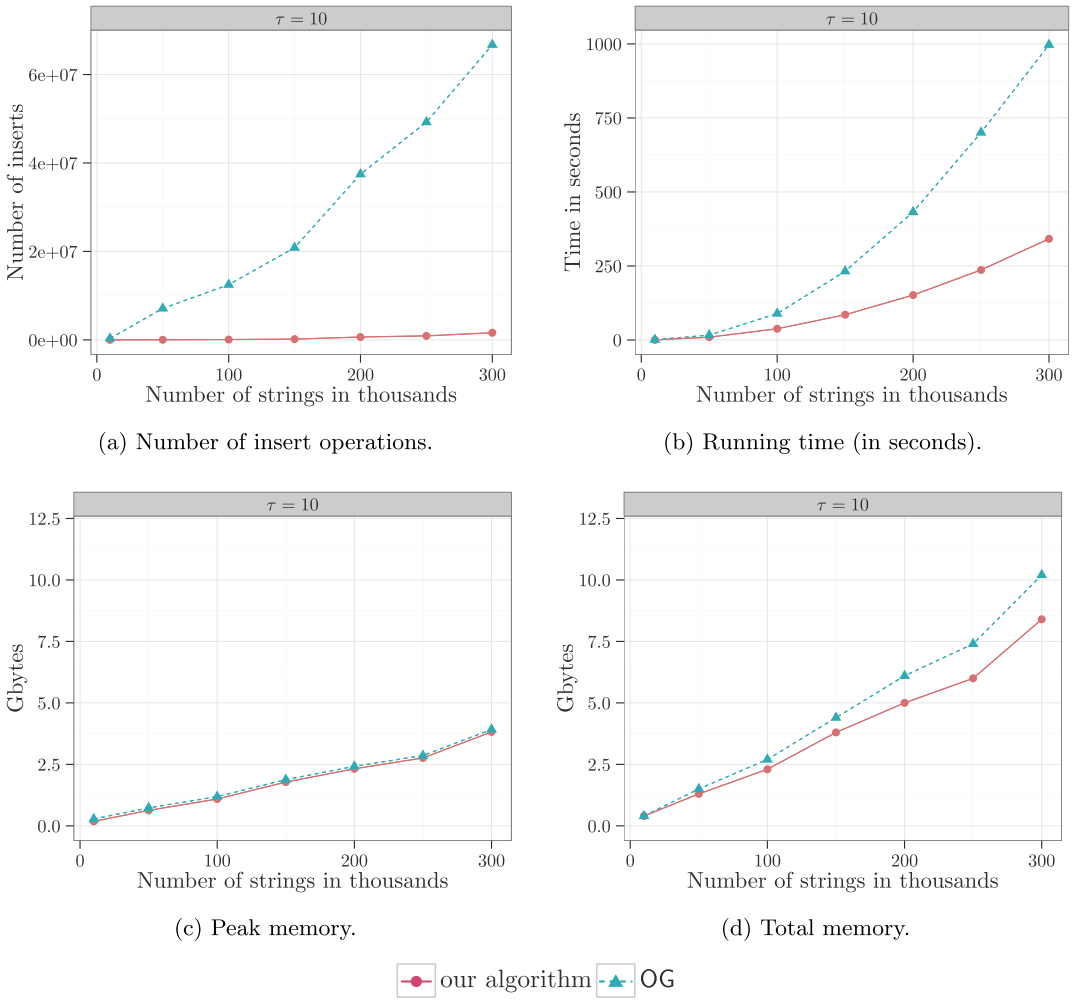

Fig. 6. Number of insert operations, running time and memory usage of our algorithm and OG for $\tau = 10$.

different threshold values $\tau = \{5, 10, 15, 20\}$ to limit the output of the algorithms to suffix–prefix matches of length larger than or equal to τ.

Our algorithm was implemented in C++ using the SDSL library [4] version 2.0[2] to construct the GESA. We also used the malloc_count library[3] to measure the memory usage. To make a fair comparison we updated the SDSL components of the OG implementation from version 1 to version 2. The source code of both algorithms is available at `https://github.com/felipelouza/apsp`.

Both our algorithm and OG were compiled by g++ v 4.7.2, with the flag std c++11 and the optimization flags O3, ffast-math, funroll-loops, m64, fomit-frame-pointer and D_FILE_OFFSET_BITS=64. Readjoiner was compiled in the same environment with the flags defined in its own Makefile. The experiments were conducted in the Linux Debian 7.0 kernel 3.2.60+1deb7u3/64 bits operating system, running on an Intel Core i7-3770 3.4 GHz processor 8 MB cache, 32 GB of internal memory and a 1 TB SATA hard disk with 7200 RPM and 64 MB cache.

We present the results in two sections, the first containing the comparison of our algorithm and OG, which are both theoretically optimal. The second contains the comparison between our algorithm and Readjoiner.

5.1. Comparison with OG

Fig. 6a shows the number of insert operations performed by our algorithm and OG on their respective auxiliary data structures. We show the results for $\tau = 10$, for other values the behavior is similar. The percentage of insert operations performed by our algorithm is about 1% of the push operations performed by OG. In OG it is possible for a suffix S_p^j to

[2] *sdsl-lite* library is available at `https://github.com/simongog/sdsl-lite`.
[3] *malloc_count* library is available at `http://panthema.net/2013/malloc_count`.

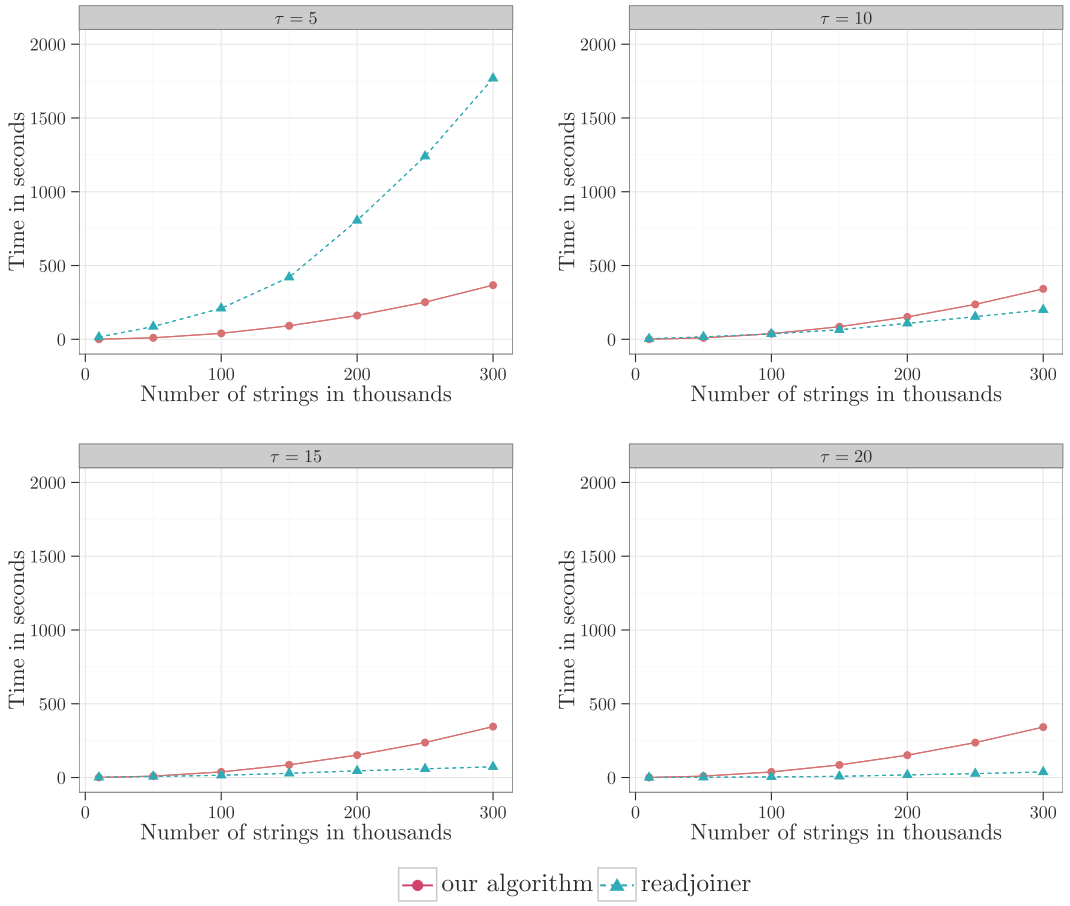

Fig. 7. Running time of our algorithm and Readjoiner for varying values of τ.

be inserted in *stack*[j] because it is a prefix of another suffix that is not a complete string. Then, the lcp value decreases before the algorithm is able to find a complete string, and we may say that this situation causes an unnecessary insertion (and removal) of S_p^j.

Fig. 6b shows the total running time (in seconds) of each algorithm, not accounting for the time to build the GESA. Our algorithm has outperformed OG by a factor of 2.6 on the average. We can see that the improvement on the running time was not proportional to the reduction of number of insert operations shown in Fig. 3. We believe that it happened because the operations on the stacks used by OG are more efficient than the operations on the lists used by our algorithm. It may also be the case that OG has a better cache locality, because operations are always performed on the top of stacks.

Figs. 6c and 6d show the amount of memory used by each algorithm. We evaluated the total and peak memory usage. The peak memory used by each algorithm is quite the same, since both algorithms use the same GESA structure, that dominates the memory usage. However, the total memory used by our algorithm was 15% less on average, which is related to the insert operations (which trigger memory allocations) performed by each algorithm, as shown in Fig. 6a.

5.2. Comparison with Readjoiner

Readjoiner includes a series of filtering steps that reduces significantly the number of strings to be compared. All contained strings and non-relevant suffixes are removed, such as those smaller than τ or without a prefix equal to the first τ symbols of another complete string. It also encodes the strings using *GtEncseq software library* [20], which improves practical performance for DNA, but prevents theoretical optimality for larger alphabets. Readjoiner process the strings in blocks, as ours and many other algorithms do.

We can see that when τ is small, for instance $\tau = 5$, Readjoiner is significantly slower than our algorithm, as shown in Fig. 7, and uses much more memory with a higher peak, as shown in Figs. 8 and 9. Such performance loss as a function of τ is pointed out in the Readjoiner manual[4]. In this particular experiment, it was necessary to limit the memory allowance for

[4] http://www.zbh.uni-hamburg.de/?id=349.

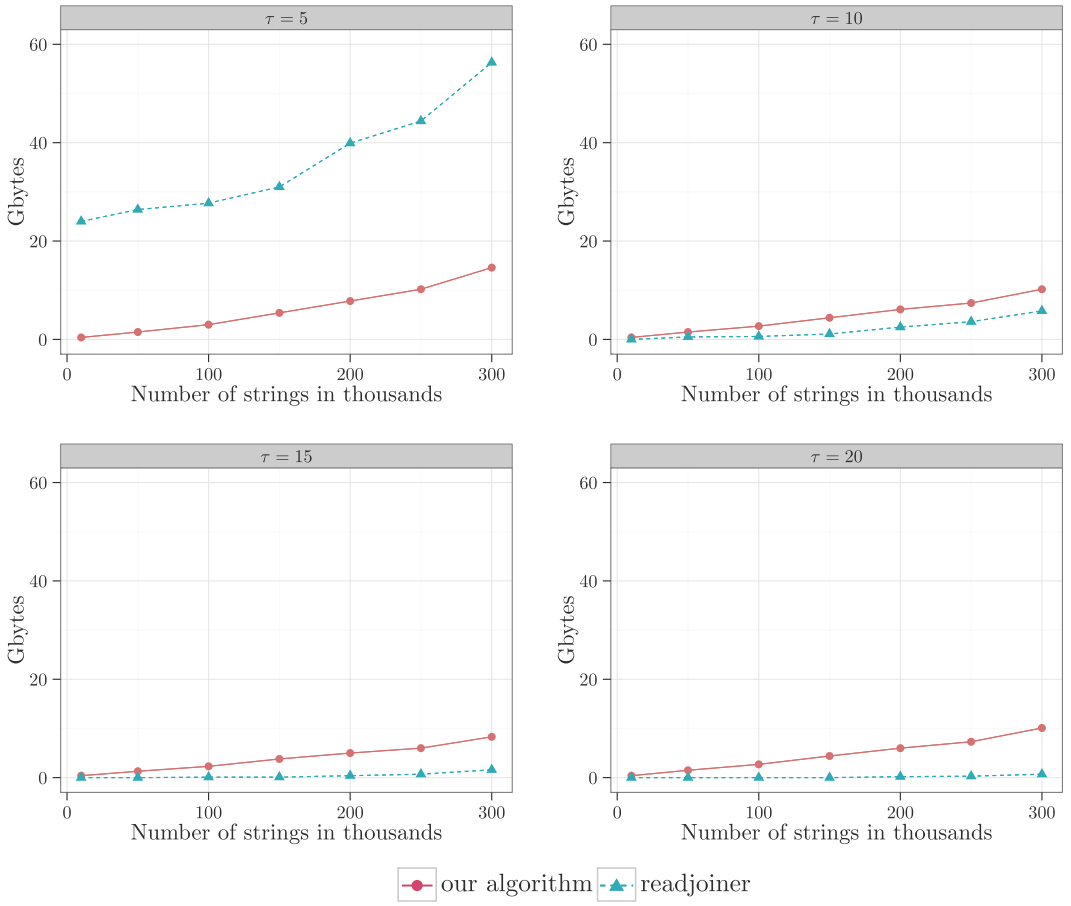

Fig. 8. Total memory usage of our algorithm and Readjoiner for varying values of τ.

Readjoiner to 12 GB as shown in the plot, otherwise it would be killed by the system. The number of suffix–prefix matches found by Readjoiner for 100 000 strings using $\tau = 5$ was 87 times larger than using $\tau = 10$, which by its turn was 3 times larger than using $\tau = 15$. Therefore, as τ increases, Readjoiner is more efficient both in time and in memory, as the filtering strategy pays off.

6. Conclusion

In this article we presented a faster and more space-efficient optimal algorithm to solve the all-pairs suffix–prefix matching problem. The new algorithm differs from the best known solution, proposed by Ohlebusch and Gog in 2010 [13] that also uses enhanced suffix arrays, in the way that the enhanced suffix array is scanned, avoiding processing suffixes that are not a suffix–prefix matching. Performance tests with real data showed that the new algorithm is 2.6 times faster (on average) using about 15% less memory. While compared to Readjoiner, a very optimized practical algorithm, we saw that when we require an exact solution for small overlap lengths, our algorithm is also very good. Although Readjoiner has outperformed our algorithm for larger overlap lengths, we may conclude that our algorithm is the best among theoretically optimal algorithms.

Our algorithm can be easily parallelized to improve its performance, since each string can be processed independently, and then all local solutions can be merged at once. Another important improvement could be modifying the algorithm to work in semi-external memory fashion, reducing the peak memory, since the GESA can be constructed externally (e.g. [10]) and can be buffered from the disk as necessary.

Acknowledgements

We thank the anonymous reviewers for comments that improved the presentation of the manuscript. WHAT acknowledges the financial support of CNPq (grant No 118372/2014-9). GPT acknowledges the support of CNPq. FAL acknowledges the financial support of CAPES and CNPq (grant No 162338/2015-5).

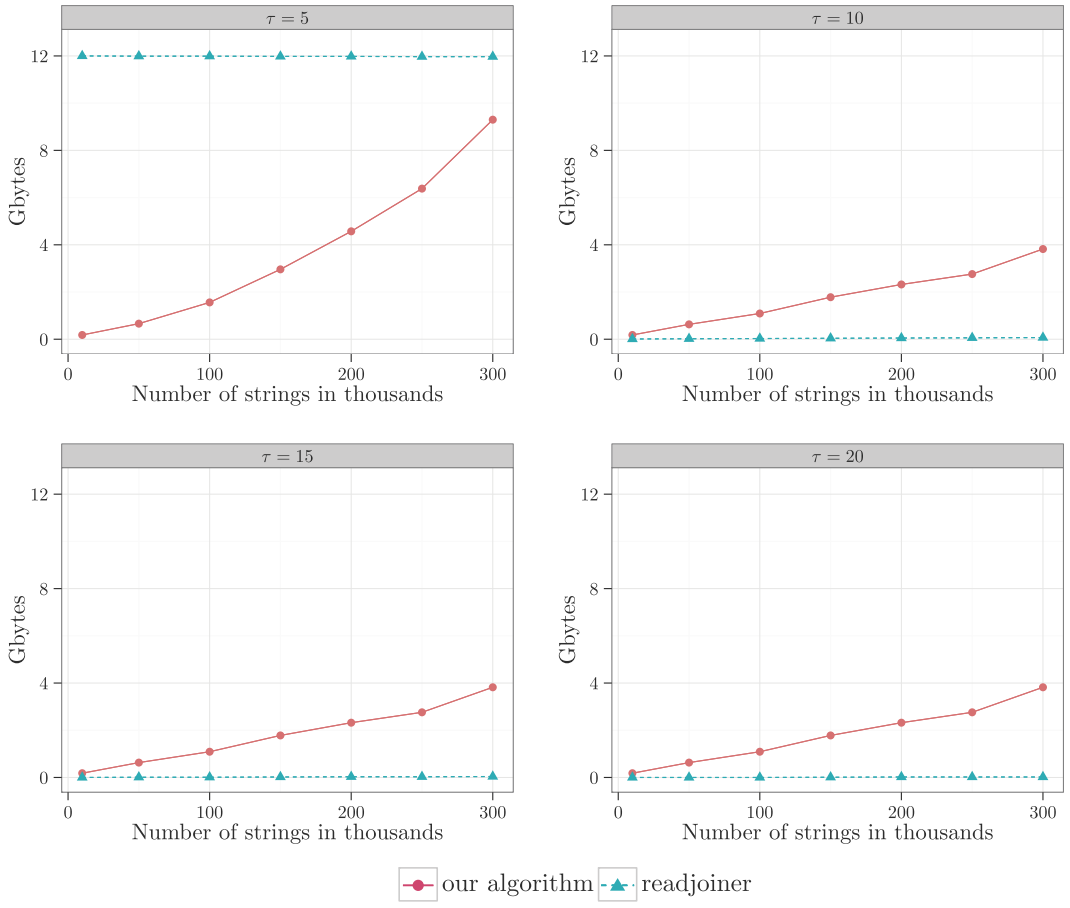

Fig. 9. Peak memory usage of our algorithm and Readjoiner for varying values of τ.

References

[1] M.I. Abouelhoda, S. Kurtz, E. Ohlebusch, Replacing suffix trees with enhanced suffix arrays, J. Discret. Algorithms 2 (2004) 53–86.
[2] M. Baker, De novo genome assembly: what every biologist should know, Nat. Methods 9 (2012) 333–337.
[3] S. El-Metwally, T. Hamza, M. Zakaria, M. Helmy, Next-generation sequence assembly: four stages of data processing and computational challenges, PLoS Comput. Biol. 9 (2013) e1003345.
[4] S. Gog, T. Beller, A. Moffat, M. Petri, From theory to practice: plug and play with succinct data structures, in: Proceedings Experimental Algorithms – 13th International Symposium, SEA 2014, June 29–July 1, 2014, in: LNCS, vol. 8504, Springer, Copenhagen, Denmark, 2014, pp. 326–337.
[5] G. Gonnella, S. Kurtz, Readjoiner: a fast and memory efficient string graph-based sequence assembler, BMC Bioinform. 13 (2012) 82.
[6] G.H. Gonnet, R. Baeza-yates, T. Snider, New Indices for Text: PAT Trees and PAT Arrays, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992, pp. 66–82.
[7] D. Gusfield, Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology, Cambridge University Press, New York, NY, USA, 1997.
[8] D. Gusfield, G.M. Landau, B. Schieber, An efficient algorithm for the all pairs suffix–prefix problem, Inf. Process. Lett. 41 (1992) 181–185.
[9] T. Kasai, G. Lee, H. Arimura, S. Arikawa, K. Park, Linear-time longest-common-prefix computation in suffix arrays and its applications, in: Proceedings Combinatorial Pattern Matching, 12th Annual Symposium, CPM 2001, July 1–4, 2001, in: LNCS, vol. 2089, Springer, Jerusalem, Israel, 2001, pp. 181–192.
[10] F.A. Louza, G.P. Telles, C.D.A. Ciferri, External memory generalized suffix and LCP arrays construction, in: Proceedings Combinatorial Pattern Matching, 24th Annual Symposium, CPM 2013, June 17–19, 2013, in: LNCS, vol. 7922, Springer, Bad Herrenalb, Germany, 2013, pp. 201–210.
[11] U. Manber, E.W. Myers, Suffix arrays: a new method for on-line string searches, SIAM J. Comput. 22 (1993) 935–948.
[12] E. Ohlebusch, Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction, Oldenbusch Verlag, 2013.
[13] E. Ohlebusch, S. Gog, Efficient algorithms for the all-pairs suffix–prefix problem and the all-pairs substring-prefix problem, Inf. Process. Lett. 110 (2010) 123–128.
[14] S.J. Puglisi, W.F. Smyth, A.H. Turpin, A taxonomy of suffix array construction algorithms, ACM Comput. Surv. 39 (2007) 1–31.
[15] M.H. Rachid, Q. Malluhi, A practical and scalable tool to find overlaps between sequences, BioMed Res. Int. 2015 (2015) 1–12.
[16] M.H. Rachid, Q.M. Malluhi, M.I. Abouelhoda, A space-efficient solution to find the maximum overlap using a compressed suffix array, in: 2014 Middle East Conference on Biomedical Engineering (MECBME), 2014, pp. 7–11.
[17] M.H. Rachid, Q.M. Malluhi, M.I. Abouelhoda, Using the Sadakane compressed suffix tree to solve the all-pairs suffix prefix problem, BioMed Res. Int. 2014 (2014) 1–18.
[18] K. Sadakane, Compressed suffix trees with full functionality, Theory Comput. Syst. 41 (2007) 589–607.

[19] J. Sirén, V. Mäkinen, N. Välimäki, G. Navarro, Run-length compressed indexes for repetitive sequence collections, in: Proceedings String Processing and Information Retrieval, 15th International Symposium, SPIRE 2008, November 10–12, 2008, in: LNCS, vol. 5280, Springer, Melbourne, Australia, 2008, pp. 164–175.
[20] S. Steinbiss, S. Kurtz, A new efficient data structure for storage and retrieval of multiple biosequences, IEEE/ACM Trans. Comput. Biol. Bioinform. 9 (2012) 345–357.
[21] P. Weiner, Linear pattern matching algorithms, in: 14th Annual Symposium on Switching and Automata Theory, October 15–17, IEEE Comp. Society, Iowa City, USA, 1973, pp. 1–11.