Theoretical Computer Science ••• (••••) •••–•••

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Faster algorithms for 1-mappability of a sequence ☆

Mai Alzamel [a,b,*], Panagiotis Charalampopoulos [a], Costas S. Iliopoulos [a], Solon P. Pissis [c], Jakub Radoszewski [d], Wing-Kin Sung [e]

[a] Department of Informatics, King's College London, London, UK
[b] Department of Computer Science, King Saud University, Riyadh, Saudi Arabia
[c] CWI, Amsterdam, the Netherlands
[d] Faculty of Mathematics, Informatics and Mechanics, University of Warsaw, Warsaw, Poland
[e] Department of Computer Science, National University of Singapore, Singapore, Singapore

ARTICLE INFO

Article history:
Received 2 December 2018
Received in revised form 12 April 2019
Accepted 24 April 2019
Available online xxxx

Keywords:
Algorithms on strings
Sequence mappability
Hamming distance

ABSTRACT

In the k-mappability problem, we are given a string x of length n and integers m and k, and we are asked to count, for each length-m factor y of x, the number of other factors of length m of x that are at Hamming distance at most k from y. We focus here on the version of the problem where $k = 1$. There exists an algorithm to solve this problem for $k = 1$ requiring time $\mathcal{O}(mn \log n / \log\log n)$ using space $\mathcal{O}(n)$. Here we present two new algorithms that require worst-case time $\mathcal{O}(mn)$ and $\mathcal{O}(n \log n \log\log n)$, respectively, and space $\mathcal{O}(n)$, thus greatly improving the previous result. Moreover, we present another algorithm that requires average-case time and space $\mathcal{O}(n)$ for integer alphabets of size σ if $m = \Omega(\log_\sigma n)$. Notably, we show that this algorithm is generalizable for arbitrary k, requiring average-case time $\mathcal{O}(kn)$ and space $\mathcal{O}(n)$ if $m = \Omega(k \log_\sigma n)$, assuming that the letters are independent and uniformly distributed random variables. Finally, we provide an experimental evaluation of our average-case algorithm demonstrating its competitiveness to the state-of-the-art implementation.

1. Introduction

The focus of this work is directly motivated by the well-known and challenging application of *genome re-sequencing*—the assembly of a genome directed by a reference sequence. New developments in sequencing technologies [2] allow whole-genome sequencing to be turned into a routine procedure, creating sequencing data in massive amounts. Short sequences, known as *reads*, are produced in huge amounts (tens of gigabytes); and in order to determine the part of the genome from which a read was derived, it must be mapped (aligned) back to some reference sequence that consists of a few gigabases. A wide variety of short-read alignment techniques and tools have been published in the past years to address the challenge of efficiently mapping tens of millions of reads to a genome, focusing on different aspects of the procedure: speed, sensitivity, and accuracy [3]. These tools allow for a small number of errors in the alignment.

The *k-mappability* problem was first introduced in the context of genome analysis in [4] (and in some sense earlier in [5]), where a heuristic algorithm was proposed to approximate the solution. The aim from a biological perspective is to

☆ A preliminary version of this article has appeared in [1].

* Corresponding author.
E-mail addresses: mai.alzamel@kcl.ac.uk (M. Alzamel), panagiotis.charalampopoulos@kcl.ac.uk (P. Charalampopoulos), csi@kcl.ac.uk (C.S. Iliopoulos), solon.pissis@cwi.nl (S.P. Pissis), jrad@mimuw.edu.pl (J. Radoszewski), ksung@comp.nus.edu.sg (W.-K. Sung).

https://doi.org/10.1016/j.tcs.2019.04.026
0304-3975/

compute the mappability of each region of a genome sequence; i.e. for every factor of a given length of the sequence, we are asked to count how many other times it occurs in the genome with up to a given number of errors. This is particularly useful in the application of genome re-sequencing. By computing the mappability of the reference genome, we can then assemble the genome of an individual with greater confidence by first mapping the segments of the DNA that correspond to regions with low mappability. Interestingly, it has been shown that genome mappability varies greatly between species and gene classes [4].

Formally, we are given a string x of length n and integers $m < n$ and $k < m$, and we are asked to count, for each length-m factor y of x, the number of other length-m factors of x that are at Hamming distance at most k from y.

Example 1. Consider the string $x = \texttt{aabaaabbbb}$ and $m = 3$. The following table shows the k-mappability counts for $k = 0$ and $k = 1$.

position	0	1	2	3	4	5	6	7
factor occurrence	aab	aba	baa	aaa	aab	abb	bbb	bbb
0-mappability	1	0	0	0	1	0	1	1
1-mappability	3	2	1	4	3	5	2	2

For instance, consider the position 0. The 0-mappability is 1, as the factor aab occurs also at position 4. The 1-mappability at this position is 3 due to the occurrence of aab at position 4 and occurrences of two factors at Hamming distance 1 from aab: aaa at position 3 and abb at position 5.

For $k = 0$, the k-mappability problem can be solved in $\mathcal{O}(n)$ time with the well-known LCP data structure [6]. For $k = \mathcal{O}(1)$ and constant-sized alphabets, there is an algorithm requiring $\mathcal{O}(\min\{nm^k, n\log^{k+1} n\})$ time and $\mathcal{O}(n)$ space [7]. In [8] the authors introduced an efficient construction of a *genome mappability array* B_k in which $B_k[\mu]$ is the smallest length m such that at least μ of the length-m factors of x do not occur elsewhere in x with at most k mismatches. The construction algorithm was later improved in [9].

For $k = 1$, the first algorithm for the k-mappability problem was published by Manzini in [10]. This solution runs in $\mathcal{O}(mn\log n/\log\log n)$ time and $\mathcal{O}(n)$ space and works only for strings over a constant-sized alphabet. Since the problem for $k = 0$ can be solved in $\mathcal{O}(n)$ time, we focus on counting, for each length-m factor y of x, the number of other factors of x that are at Hamming distance *exactly* 1 — instead of at most 1 — from y.

Our contributions. Here we make the following fourfold contribution:

(a) We present an algorithm that, given a string x of length n over an integer alphabet of size $\sigma > 1$ and a positive integer $m = \Omega(\log_\sigma n)$, solves the 1-mappability problem for x in average-case time $\mathcal{O}(n)$ and space $\mathcal{O}(n)$. Notably, we show that this algorithm is generalizable for arbitrary k requiring average-case time $\mathcal{O}(kn)$ and space $\mathcal{O}(n)$ if $m = \Omega(k\log_\sigma n)$. Here we assume that the letters are independent and uniformly distributed random variables.
(b) We present an algorithm that, given a string of length n over an integer alphabet and a positive integer m, solves the 1-mappability problem in $\mathcal{O}(mn)$ time and $\mathcal{O}(n)$ space.
(c) We present an algorithm that, given a string of length n over a constant-sized alphabet and a positive integer m, solves the 1-mappability problem in $\mathcal{O}(\min\{mn, n\log n\log\log n\})$ time and $\mathcal{O}(n)$ space, thus improving on the algorithm of [10] that requires $\mathcal{O}(mn\log n/\log\log n)$ time and $\mathcal{O}(n)$ space.
(d) We provide an open-source implementation of our average-case algorithm for arbitrary k and also experimental results demonstrating its competitiveness to the state-of-the-art implementation for the same problem [4].

2. Preliminaries

Let $x = x[0]x[1]\ldots x[n-1]$ be a *string* of length $|x| = n$ over a finite ordered alphabet Σ of size $|\Sigma| = \sigma = \mathcal{O}(1)$. We also consider the case of strings over an *integer alphabet*, where each letter is replaced by its rank in such a way that the resulting string consists of integers in the range $\{1, \ldots, n\}$.

For two positions i and j on x, we denote by $x[i..j] = x[i]\ldots x[j]$ the *factor* (sometimes called *substring*) of x that starts at position i and ends at position j (it is of length 0 if $j < i$). By ε we denote the *empty string* of length 0. We recall that a *prefix* of x is a factor that starts at position 0 ($x[0..j]$) and a *suffix* of x is a factor that ends at position $n-1$ ($x[i..n-1]$). We denote the *reverse* string of x by $\mathsf{rev}(x)$, i.e. $\mathsf{rev}(x) = x[n-1]x[n-2]\ldots x[1]x[0]$.

Let y be a string of length m with $0 < m \leq n$. We say that there exists an *occurrence* of y in x, or, more simply, that y *occurs in* x, when y is a factor of x. Every occurrence of y can be characterized by a starting position in x. Thus we say that y occurs at the *starting position* i in x when $y = x[i..i+m-1]$.

The *Hamming distance* between two strings x and y, $|x| = |y|$, is defined as $d_H(x, y) = |\{i : x[i] \neq y[i],\ i = 0, 1, \ldots, |x|-1\}|$. If $|x| \neq |y|$, we set $d_H(x, y) = \infty$. If two strings x and y are at Hamming distance k, we write $x \approx_k y$.

The computational problem in scope can be formally stated as follows.

Please cite this article in press as: M. Alzamel et al., Faster algorithms for 1-mappability of a sequence, Theoret. Comput. Sci. (2019), https://doi.org/10.1016/j.tcs.2019.04.026

1-MAPPABILITY
Input: A string x of length n and an integer m, where $1 \leq m < n$
Output: An integer array C of size $n - m + 1$ such that $C[i]$ stores the number of factors of x that are at Hamming distance 1 from $x[i..i + m - 1]$

2.1. Suffix array and suffix tree

Let x be a string of length $n > 0$. We denote by SA the *suffix array* of x. SA is an integer array of size n storing the starting positions of all (lexicographically) sorted non-empty suffixes of x, i.e. for all $1 \leq r < n$ we have $x[\mathsf{SA}[r-1]..n-1] < x[\mathsf{SA}[r]..n-1]$ [11]. Let $\mathsf{lcp}(r, s)$ denote the length of the longest common prefix between $x[\mathsf{SA}[r]..n-1]$ and $x[\mathsf{SA}[s]..n-1]$ for positions r, s on x. We denote by LCP the *longest common prefix* array of x defined by $\mathsf{LCP}[r] = \mathsf{lcp}(r-1, r)$ for all $1 \leq r < n$, and $\mathsf{LCP}[0] = 0$. The inverse iSA of the array SA is defined by $\mathsf{iSA}[\mathsf{SA}[r]] = r$, for all $0 \leq r < n$. It is known that SA, iSA, and LCP of a string of length n, over an integer alphabet, can be computed in time and space $\mathcal{O}(n)$ [12,6]. It is then known that a range minimum query (RMQ) data structure over the LCP array, that can be constructed in $\mathcal{O}(n)$ time and $\mathcal{O}(n)$ space [13], can answer lcp-queries in $\mathcal{O}(1)$ time per query [11]. A symmetric construction on $\mathsf{rev}(x)$ can answer the so-called *longest common suffix* (lcs) queries in the same complexity. The lcp and lcs queries are also known as *longest common extension* (LCE) queries.

The *suffix tree* $\mathcal{T}(x)$ of string x is a compact trie representing all suffixes of x. The nodes of the trie which become nodes of the suffix tree are called *explicit* nodes, while the other nodes are called *implicit*. Each edge of the suffix tree can be viewed as an upward maximal path of implicit nodes starting with an explicit node. Moreover, each node belongs to a unique path of that kind. Thus, each node of the trie can be represented in the suffix tree by the edge it belongs to and an index within the corresponding path. The label of an edge is its first letter. We let $\mathcal{L}(v)$ denote the *path-label* of a node v, i.e., the concatenation of the edge labels along the path from the root to v. We say that v is path-labeled $\mathcal{L}(v)$. Additionally, $\mathcal{D}(v) = |\mathcal{L}(v)|$ is used to denote the *string-depth* of node v. Node v is a *terminal* node if its path-label is a suffix of x, that is, $\mathcal{L}(v) = x[i..n-1]$ for some $0 \leq i < n$; here v is also labeled with index i. It should be clear that each factor of x is uniquely represented by either an explicit or an implicit node of $\mathcal{T}(x)$. In standard suffix tree implementations, we assume that each node of the suffix tree is able to access its parent. Once $\mathcal{T}(x)$ is constructed, it can be traversed in a depth-first manner to compute $\mathcal{D}(v)$ for each node v.

It is known that the suffix tree of a string of length n, over an integer alphabet, can be computed in time and space $\mathcal{O}(n)$ [14]. For integer alphabets, in order to access the children of an explicit node by the first letter of their edge label, perfect hashing [15] can be used.

3. Efficient average-case algorithm

In this section we assume that x is a string over an integer alphabet Σ. For clarity of presentation, we first describe the algorithm for $k = 1$ and then show how it can be generalized for arbitrary k. Recall that if two strings y and z are at Hamming distance 1, we write $y \approx_1 z$.

Fact 1 *(Folklore). Given two strings y and z of length m, we have that if $y \approx_1 z$, then y and z share at least one factor of length $\lfloor m/2 \rfloor$.*

Fact 2. *Given a string x and any two positions i, j on x, we have that if $x[i..i+m-1] \approx_1 x[j..j+m-1]$, then $x[i..i+m-1]$ and $x[j..j+m-1]$ have at least one common factor of length $L = \lfloor m/3 \rfloor$ starting at positions $i' \in \{i, \ldots, i+m-L\}$ and $j' \in \{j, \ldots, j+m-L\}$ of x, such that $i' - i = j' - j$ and $i' = 0 \pmod{L}$.*

Proof. It should be clear that every factor of x of length m fully contains at least two factors of length L starting at positions equal to 0 mod L. Then, if $x[i..i+m-1]$ and $x[j..j+m-1]$ are at Hamming distance 1, analogously to Fact 1, at least one of the two factors of length L that are fully contained in $x[i..i+m-1]$ occurs at a corresponding position in $x[j..j+m-1]$; otherwise we would have a Hamming distance greater than 1. □

We first initialize an array C of size $n - m + 1$, with 0 in all positions; for all i, $C[i]$ will eventually store the number of factors of x that are at Hamming distance 1 from $x[i..i+m-1]$. We apply Fact 2 by implicitly splitting the string x into $B = \lfloor \frac{n}{\lfloor m/3 \rfloor} \rfloor$ *blocks* of length $L = \lfloor m/3 \rfloor$—the suffix of length $n \bmod \lfloor m/3 \rfloor$ is not taken as a block—starting at the positions of x that are equal to 0 mod L. In order to find all pairs of length-m factors that are at Hamming distance 1 from each other, we can find all the exact matches of every block and try to extend each of them to the left and to the right, allowing at most one mismatch. However, we need to tackle some technical details to correctly update our counters and avoid double counting.

We start by constructing the SA and LCP arrays for x and $\mathsf{rev}(x)$ in $\mathcal{O}(n)$ time. We also construct RMQ data structures over the LCP arrays for answering LCE queries in constant time per query. By exploiting the LCP array information, we can

Please cite this article in press as: M. Alzamel et al., Faster algorithms for 1-mappability of a sequence, Theoret. Comput. Sci. (2019), https://doi.org/10.1016/j.tcs.2019.04.026

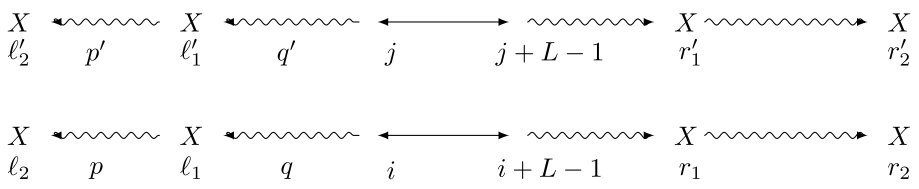


Fig. 1. Performing two LCE queries in each direction.

then find in $\mathcal{O}(n)$ time all maximal sets of indices such that the longest common prefix between any two of the suffixes starting at these indices is at least L and at least one of them is the starting position of some block.

Then for each such set, denoted by P, we have to do the following procedure for each index $i \in P$ such that $i = 0 \pmod L$.

For every other $j \in P$, we try to extend the match by asking two LCE queries in each direction. I.e., we ask an $\mathsf{lcs}(i-1, j-1)$ query to find the first mismatch positions ℓ_1 and ℓ'_1, respectively, and then $\mathsf{lcs}(\ell_1 - 1, \ell'_1 - 1)$ to find the second mismatch (ℓ_2 and ℓ'_2, respectively). A symmetric procedure computes the mismatches r_1, r'_1 and r_2, r'_2 to the right, as shown in Fig. 1. We omit here some technical details with regards to reaching the start or end of x.

Now we are interested in positions p such that $\ell_2 < p \leq \ell_1$ and $i + L - 1 \leq p + m - 1 < r_1$ and positions q such that $\ell_1 < q \leq i$ and $r_1 \leq q + m - 1 < r_2$. Each such position p (resp. q) implies that $x[p..p+m-1] \approx_1 x[p'..p'+m-1]$, where $p' = j - (i - p)$. Henceforth, we only consider positions of the type p, p'.

Note that if $x[p..p+m-1] \approx_1 x[p'..p'+m-1]$, we will identify the unordered pair $\{p, p'\}$ based on the described approach $t_{p,p'}$ times, where $t_{p,p'}$ is the total number of full blocks contained in $x[p..p+m-1]$ and in $x[p'..p'+m-1]$ after the mismatch position. It is not hard to compute the number $t_{p,p'}$ in $\mathcal{O}(1)$ time based on the starting positions p and p' as well as ℓ_1 and r_1 each time we identify $x[p..p+m-1] \approx_1 x[p'..p'+m-1]$. To avoid double counting, we then increment the $C[p]$ and $C[p']$ counters by $1/t_{p,p'}$.

By $\mathsf{EXT}_{i,j}$ we denote the time required to process a pair of elements i, j of a set P such that at least one of them, i or j, equals 0 mod L.

Lemma 3. *The time $\mathsf{EXT}_{i,j}$ is $\mathcal{O}(m)$.*

Proof. Given $i, j \in P$, with at least one of them equal to 0 mod L, we can find the pairs (p, p') of positions that satisfy the inequalities discussed above in $\mathcal{O}(m)$ time. They are a subset of $\{(i-m+L, j-m+L), \ldots, (i-1, j-1)\}$. For each such pair (p, p') we can compute $t_{p,p'}$ and increment $C[p]$ and $C[p']$ accordingly in $\mathcal{O}(1)$ time. The total time to process all pairs (p, p') for given i, j is thus $\mathcal{O}(m)$. □

It should be clear that the aforementioned algorithm is generalizable for arbitrary k. We proceed with proving the following theorem.

Theorem 2. *Given a string x of length n over an integer alphabet Σ of size $\sigma > 1$ with the letters of x being independent and identically distributed random variables, uniformly distributed over Σ, the k-mappability problem can be solved in average-case time $\mathcal{O}(kn)$ and space $\mathcal{O}(n)$ if $m \geq (k+2) \cdot (\log_\sigma n + 1)$.*

Proof. The time and space required for constructing the SA and LCP array for x and $\mathsf{rev}(x)$ and the RMQ data structures over the LCP arrays is $\mathcal{O}(n)$.

Let B denote the number of blocks over x and L be the block length. We set

$$L = \lfloor \tfrac{m}{k+2} \rfloor, \quad B = \lfloor \tfrac{n}{L} \rfloor$$

to apply the pigeon-hole principle: at least one block must be an exact match (generalization of Fact 2). Recall that by P we denote a maximal set of indices of the LCP array such that the length of the longest common prefix between any two suffixes starting at these indices is at least L and at least one of them is the starting position of some block. Processing all such sets P requires time

$$\mathsf{EXT}_{i,j} \cdot Occ$$

where $\mathsf{EXT}_{i,j}$ is the time required to process a pair i, j of elements of a set P; and Occ is the sum of the multiples of the cardinality of each set P times the number of the elements of set P that are equal to 0 mod L. We generalize Lemma 3 for arbitrary k, showing that $\mathsf{EXT}_{i,j} = \mathcal{O}(m)$ as follows. We perform at most $2k+2$ longest common extension queries (to the left and to the right); list all $\mathcal{O}(k)$ blocks that do not contain a mismatch within these extensions; and then consider $\mathcal{O}(m)$ positions to be updated. Additionally, by the stated assumption on the string x, the expected value for Occ is no more than $\frac{Bn}{\sigma^L}$. Hence, the algorithm on average requires time

$$\mathcal{O}(n + m \cdot \frac{B \cdot n}{\sigma^L}).$$

Let $m = (k+2)q + r$, for $0 \le r \le k+1$, $q \ge 1$; note that here we assume that $m \ge k+2$; further note that $\lfloor m/(k+2) \rfloor = q$. If q satisfies $n \le \sigma^q$ we have

$$m \cdot \frac{B}{\sigma^L} = \frac{m \cdot \lfloor \frac{n}{\lfloor m/(k+2) \rfloor} \rfloor}{\sigma^{\lfloor \frac{m}{k+2} \rfloor}} = \frac{m \cdot \lfloor \frac{n}{q} \rfloor}{\sigma^q} \le \frac{m \cdot \frac{n}{q}}{\sigma^q} \le \frac{m}{q} = \frac{(k+2)q + r}{q}$$
$$= k + 2 + \frac{r}{q} \le 2k + 3.$$

Consequently, in the case when

$$m \ge (k+2) \cdot (\log_\sigma n + 1)$$

we have that

$$m \frac{B \cdot n}{\sigma^L} \le (2k+3)n$$

and hence the algorithm requires $\mathcal{O}(kn)$ time on average. The extra space usage is $\mathcal{O}(n)$. □

We thus obtain the following corollary with respect to the 1-MAPPABILITY problem; namely, for $k = 1$.

Corollary 4. *Given a string x of length n over an integer alphabet Σ of size $\sigma > 1$ with the letters of x being independent and identically distributed random variables, uniformly distributed over Σ, the* 1-MAPPABILITY *problem can be solved in average-case time $\mathcal{O}(n)$ and space $\mathcal{O}(n)$ if $m \ge 3 \cdot \log_\sigma n + 3$.*

4. Efficient worst-case algorithms

4.1. $\mathcal{O}(mn)$-time and $\mathcal{O}(n)$-space algorithm

In this section we assume that x is a string over an integer alphabet Σ. The main idea is that we want to first find all pairs $x[i_1..i_1 + m - 1] \approx_1 x[i_2..i_2 + m - 1]$ that have a mismatch in the first position, then in the second, and so on.

Let us fix $0 \le j < m$. In order to identify the pairs $x[i_1..i_1 + m - 1] \approx_1 x[i_2..i_2 + m - 1]$ with $x[i_1 + j] \ne x[i_2 + j]$ (i.e. with the mismatch in the jth position), we do the following. For every $i = 0, 1, \ldots, n - m$, we find the explicit or implicit node $u_{i,j}$ in $\mathcal{T}(x)$ that represents $x[i..i + j - 1]$ and the node $v_{i,j}$ in $\mathcal{T}(\mathsf{rev}(x))$ that represents $\mathsf{rev}(x[i + j + 1..i + m - 1]) = \mathsf{rev}(x)[n - i - m..n - i - j - 2]$. In each such node $v_{i,j}$, we create a set $V(v_{i,j})$—if it has not already been created—and insert the triple $(u_{i,j}, x[i + j], i)$.

When we have done this for all possible starting positions of x, we group the triples in each set $V(v)$ by the node variable (i.e., the first component in the triples). For each such group in $V(v)$ we count the number of triples that have each letter of the alphabet and increment array C accordingly. More precisely, if $V(v)$ contains q triples that correspond to the same node u, among which r correspond to the letter $c \in \Sigma$, then for each such triple $(u, c, i) \in V(v)$ we increment $C[i]$ by $q - r$; we subtract r to avoid counting equal factors in C. Before we proceed with the computations for the next index j, we delete all the sets $V(v)$. We formalize this algorithm, denoted by 1-MAP, in the pseudocode presented below and provide an example.

```
1-Map(x, n, m)
 1  T(x) ← SuffixTree(x)
 2  T(rev(x)) ← SuffixTree(rev(x))
 3  for string-depth j = 0 to m − 1 do
 4      for i = 0 to n − m do
 5          u_{i,j} ← Node_{T(x)}(x[i..i + j − 1])
 6          v_{i,j} ← Node_{T(rev(x))}(rev(x)[n − i − m..n − i − j − 2])
 7          Insert (u_{i,j}, x[i + j], i) to V(v_{i,j})
 8      for every node v of string-depth m − j − 2 in T(rev(x)) do
 9          Group triples in V(v) by the node variable
10          for a group corresponding to the node u in V(v) do
11              Count number of triples with each letter c ∈ Σ
12              Update C[i] accordingly for each triple (u, c, i)
13          Delete V(v)
```

Example 3. Suppose we have $V(v) = \{(u, \mathtt{A}, i_1), (u, \mathtt{A}, i_2), (u, \mathtt{A}, i_3), (u, \mathtt{C}, i_4), (u, \mathtt{C}, i_5), (u, \mathtt{C}, i_6), (u, \mathtt{G}, i_7), (u, \mathtt{G}, i_8), (u, \mathtt{T}, i_9)\}$, for some distinct positions $i_1, i_2, \ldots, i_9$. We then increment $C[i_1]$, $C[i_2]$, $C[i_3]$, $C[i_4]$, $C[i_5]$, and $C[i_6]$ by 6; $C[i_7]$ and $C[i_8]$ by 7; and $C[i_9]$ by 8.

Please cite this article in press as: M. Alzamel et al., Faster algorithms for 1-mappability of a sequence, Theoret. Comput. Sci. (2019), https://doi.org/10.1016/j.tcs.2019.04.026

We now analyze the time complexity of this algorithm. The algorithm iterates j from 0 to $m-1$. In the jth iteration, we need to compute $\{u_{i,j}, v_{i,j} \mid i = 0, \ldots, n-m\}$. When $j = 0$, $u_{i,0}$ for every i is the root of $\mathcal{T}(x)$ and we can find $v_{i,0}$ for all i naïvely in $\mathcal{O}(mn)$ time. For $j > 0$, $v_{i,j}$ can be found in $\mathcal{O}(1)$ time from $v_{i,j-1}$ by moving one letter up in $\mathcal{T}(\mathtt{rev}(x))$ for all i, while $u_{i,j}$ can be obtained from $u_{i,j-1}$ by going down in $\mathcal{T}(x)$ based on letter $x[i+j]$. We then include $(u_{i,j}, x[i+j], i)$ in $V(v_{i,j})$.

This requires in total $\mathcal{O}(mn)$ randomized time due to perfect hashing [15] which allows to go down from a node in $\mathcal{T}(x)$ (or in $\mathcal{T}(\mathrm{rev}(x))$) based on a letter in $\mathcal{O}(1)$ randomized time. We can actually avoid this randomization, as queries for a particular child of a node are asked in our solution in a somewhat off-line fashion: we use them only to compute $v_{i,0}$ (m times) and $u_{i,j}$ (from $u_{i,j-1}$).

Observation 5. *For an integer alphabet $\Sigma = \{1, \ldots, n\}$, one can answer off-line $\mathcal{O}(n)$ queries in $\mathcal{T}(x)$ asking for a child of an explicit or implicit node u labeled with the letter $c \in \Sigma$ in (deterministic) $\mathcal{O}(n)$ time.*

Proof. A query for an implicit node u is answered in $\mathcal{O}(1)$ time, as there is only one outgoing edge to check. All the remaining queries can be sorted lexicographically as pairs (u, c) using radix sort. We can also assume that the children of every explicit node of $\mathcal{T}(x)$ are ordered by the letter (otherwise we also radix sort them). Finally, all the queries related to a node u can be answered in one go by iterating through the children list of u once. □

Lastly, we use bucket sort to group the triples for each $V(v)$ according to the node variable (recall that the nodes are represented by the edge and the index within the edge) and update the counters in $\mathcal{O}(n)$ time in total (using a global array indexed by the letters from Σ, which is zeroed in $\mathcal{O}(|V(v)|)$ time after each $V(v)$ has been processed). Overall the algorithm requires $\mathcal{O}(mn)$ time. The suffix trees require $\mathcal{O}(n)$ space and we delete the sets $V(v_{i,j})$ after the jth iteration; the space complexity of the algorithm is thus $\mathcal{O}(n)$. We obtain the following result.

Theorem 4. *Given a string of length n over an integer alphabet and an integer m, where $1 \leq m < n$, the* 1-MAPPABILITY *problem can be solved in $\mathcal{O}(mn)$ time and $\mathcal{O}(n)$ space.*

Corollary 4 and Theorem 4 imply the following result.

Theorem 5. *Given a string x of length n over an integer alphabet Σ of size $\sigma > 1$ with the letters of x being independent and identically distributed random variables, uniformly distributed over Σ, the* 1-MAPPABILITY *problem can be solved in average-case time $\mathcal{O}(n \log n)$ and space $\mathcal{O}(n)$.*

Proof. If $m \geq 3 \cdot \log_\sigma n + 3$, apply Corollary 4. Otherwise, apply Theorem 4. □

Remark 6. Theorem 4 can also be obtained via utilizing the gapped suffix array data structure (see [16] for an efficient construction algorithm).

4.2. $\mathcal{O}(n \log n \log\log n)$-time and $\mathcal{O}(n)$-space algorithm

In this section we assume that x is a length-n string over an ordered alphabet Σ, where $|\Sigma| = \sigma = \mathcal{O}(1)$. Consider two factors of x represented by nodes u and v in $\mathcal{T}(x)$; we observe that the first mismatch between the two factors is the first letter of the labels of the distinct outgoing edges from the lowest common ancestor of u and v that lie on the paths from the root to u and v. For 1-mappability we require that what follows this mismatch is an exact match.

Definition 1. Let T be a rooted tree. For each non-leaf node u of T, the *heavy edge* (u, v) is an edge for which the subtree rooted at v has the maximal number of leaves (in case of several such subtrees, we fix one of them). The *heavy path of a node v* is a maximal path of heavy edges that passes through v (it may contain 0 edges). The *heavy path of T* is the heavy path of the root of T.

Consider the suffix tree $\mathcal{T}(x)$ and its node u. We say that an (explicit or implicit) node v is a *level ancestor of u at string-depth ℓ* if $\mathcal{D}(v) = \ell$ and $\mathcal{L}(v)$ is a prefix of $\mathcal{L}(u)$. The heavy paths of $\mathcal{T}(x)$ can be used to compute level ancestors of nodes in $\mathcal{O}(\log n)$ time. However, a more efficient data structure is known.

Lemma 7 *([17]). After $\mathcal{O}(n)$-time preprocessing on $\mathcal{T}(x)$, level ancestor queries of nodes of $\mathcal{T}(x)$ can be answered in $\mathcal{O}(\log\log n)$ time per query.*

Definition 2. Given a string x and a factor y of x, we denote by $range(x, y)$ the range in the SA of x that represents the suffixes of x that have y as a prefix.

Please cite this article in press as: M. Alzamel et al., Faster algorithms for 1-mappability of a sequence, Theoret. Comput. Sci. (2019), https://doi.org/10.1016/j.tcs.2019.04.026

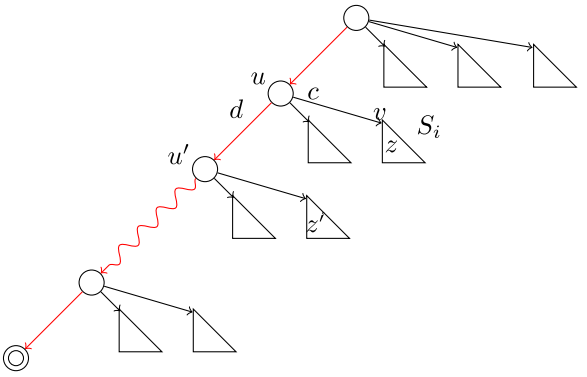


Fig. 2. Illustration; the heavy path of $\mathcal{T}(x)$ is shown in red. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

Every node u in $\mathcal{T}(x)$ corresponds to an SA range $I_u = \mathit{range}(x, \mathcal{L}(u)) = (u_{\min}, u_{\max})$. We can precompute I_u for all explicit nodes u in $\mathcal{T}(x)$ in $\mathcal{O}(n)$ time while performing a depth-first traversal of the tree as follows. For a non-terminal node v with children $u^1, \ldots, u^q$, we set $v_{\min} = \min_i\{u^i_{\min}\}$ and $v_{\max} = \max_i\{u^i_{\max}\}$. If v is a terminal node (with children $u^1, \ldots, u^q$), representing the suffix $x[j..n-1]$, we set $v_{\min} = \mathsf{iSA}[j]$ and $v_{\max} = \max\{\mathsf{iSA}[j], \max_i\{u^i_{\max}\}\}$. When a considered node v is implicit, say along an edge (p, q), then $I_v = I_q$.

Our algorithm relies heavily on the following auxiliary lemmas.

Lemma 8. *Consider a node u in $\mathcal{T}(x)$ with $p = \mathcal{L}(u)$. Let $\mathit{suf}(u, \ell)$ be the node v such that $\mathcal{L}(v) = p[\ell..|p|-1]$. Given the SA and the iSA of x, v can be computed in $\mathcal{O}(\log\log n)$ time after $\mathcal{O}(n)$-time preprocessing.*

Proof. The SA range of the node u is $I_u = (u_{\min}, u_{\max})$; $u_{\min}$ corresponds to the suffix $x[\mathsf{SA}[u_{\min}]..n-1]$. By removing the first ℓ letters, the suffix becomes $x[\mathsf{SA}[u_{\min}]+\ell..n-1]$. The corresponding SA value is $v_{\min} = \mathsf{iSA}[\mathsf{SA}[u_{\min}]+\ell]$.

Let v_1 be the node of $\mathcal{T}(x)$ such that $\mathcal{L}(v_1) = x[\mathsf{SA}[v_{\min}]..n-1]$. The sought node v is the ancestor of v_1 located at string-depth $|p|-\ell$. It can be computed in $\mathcal{O}(\log\log n)$ time using the level ancestor data structure of Lemma 7. □

Lemma 9. *Let u and v be two nodes in $\mathcal{T}(x)$. We denote $\mathcal{L}(u)$ by p_1 and $\mathcal{L}(v)$ by p_2. We further denote by $\mathit{concat}(u, v)$ the node w such that $\mathcal{L}(w) = p_1p_2$. Given the SA and the iSA of x, as well as $\mathit{range}(x, p_1)$ and $\mathit{range}(x, p_2)$, w can be located in $\mathcal{O}(\log\log n)$ time after $\mathcal{O}(n\log\log n)$-time and $\mathcal{O}(n)$-space preprocessing.*

Proof. We can compute $\mathit{range}(x, p_1p_2) = (w_{\min}, w_{\max})$ in $\mathcal{O}(\log\log n)$ time after $\mathcal{O}(n\log\log n)$-time and $\mathcal{O}(n)$-space preprocessing [18]; we can then locate w in $\mathcal{O}(\log\log n)$ time using the level ancestor data structure of Lemma 7. □

We are now ready to present an algorithm for 1-mappability that requires $\mathcal{O}(n\log n\log\log n)$ time and $\mathcal{O}(n)$ space. The first step is to build $\mathcal{T}(x)$. We then make every node u of string-depth m explicit in $\mathcal{T}(x)$ and initialize a counter $\mathit{Count}(u)$ for it. For each explicit node u in $\mathcal{T}(x)$, the SA range $I_u = \mathit{range}(x, \mathcal{L}(u))$ is also stored. We also identify the node v_c with path-label c for each $c \in \Sigma$ in $\mathcal{O}(\sigma) = \mathcal{O}(1)$ time.

```
PerformCount(T, m)
 1  HP ← HeavyPath(T)
 2  for each side-tree S_i attached to a node u on HP with D(u) < m do
 3      Let (u, v) be the edge that connects S_i to HP
 4      c ← the edge label of (u, v)
 5      d ← the edge label of the heavy edge (u, u′)
 6      for each node z in S_i with D(z) = m do
 7          w ← suf(z, D(u) + 1)
 8          for each c′ ≠ c, label of an outgoing edge from u do
 9              t ← concat(u, concat(v_c′, w))
10              Count(z) ← Count(z) + |I_t|
11          z′ ← concat(u, concat(v_d, w))
12          Count(z′) ← Count(z′) + |I_z|
13      PerformCount(S_i, m − D(u))
```

We then call PerformCount($\mathcal{T}(x), m$), which does the following (inspect also the pseudocode above and Fig. 2). At first, a heavy path HP of $\mathcal{T}(x)$ is computed. Initially, we want to identify the pairs of factors of x of length m at Hamming distance

1 that have a mismatch in the labels of the edges outgoing from a node in *HP*. Given a node u in *HP*, with $\mathcal{L}(u) = p_1$, for every side tree S_i attached to it (say by an edge with label $c \in \Sigma$), we find all nodes of S_i with string-depth m. For every such node z, with path-label p_1cp_2, we use Lemma 8 to obtain the node $w = \text{suf}(z, |p_1| + 1)$; that is, $\mathcal{L}(w) = p_2$. We then use Lemma 9 to compute $range(x, p_1c'p_2)$ for all $c' \neq c$ such that there is an outgoing edge from u with label c' and increment $Count(z)$ by $|range(p_1c'p_2)|$. Let the heavy edge from u have label d; we also increment $Count(z')$, where $z' = \text{concat}(u, \text{concat}(v_d, w))$ is the node with path-label p_1dp_2, by $|I_z|$ while processing node z.

This procedure then recurs on each of the side trees; i.e. for side tree S_i, attached to node u, it calls PerformCount$(S_i, m - \mathcal{D}(u))$. Finally, we construct array C from array $Count$ while performing one more depth-first traversal.

On the recursive calls of PerformCount in each of the side trees (e.g. S_i) attached to *HP*, we first compute the heavy paths (in $\mathcal{O}(|S_i|)$ time for S_i) and then consider each node of string-depth m of $\mathcal{T}(x)$ at most once; as above, we process each node in $\mathcal{O}(\log\log n)$ time due to Lemmas 8 and 9. As there are at most n nodes of string-depth m, we do $\mathcal{O}(n \log\log n)$ work in total. This is also the case as we go deeper in the tree. Since the number of leaves of the trees we are dealing with at least halves in each iteration, there at most $\mathcal{O}(\log n)$ steps. Hence, each node of string-depth m will be considered $\mathcal{O}(\log n)$ times and every time we will do $\mathcal{O}(\log\log n)$ work for it. The overall time complexity of the algorithm is thus $\mathcal{O}(n \log n \log\log n)$. The space complexity is $\mathcal{O}(n)$. By applying Theorem 4 we obtain the following result.

Theorem 6. *Given a string of length n over a constant-sized alphabet and an integer m, where $1 \leq m < n$, the* 1-MAPPABILITY *problem can be solved in $\mathcal{O}(\min\{mn, n \log n \log\log n\})$ time and $\mathcal{O}(n)$ space.*

Remark 10. The data structure presented by Cole et al. [19] for pattern matching with up to k mismatches can be used. For $k = 1$, this data structure is of size $\mathcal{O}(n \log n)$ and can be built in time $\mathcal{O}(n \log n)$. We can then find all occ occurrences of a given factor of x with at most 1 mismatch in time $\mathcal{O}(\log n \log\log n + occ)$. However, the $\omega(n)$ space required for this data structure is prohibitive for genome-scale analyses—in Theorem 6 we use $\mathcal{O}(n)$ space.

5. Experimental results

We have implemented the average-case algorithm described in Section 3 as a program to compute the mappability values. The program has been implemented in the `C++` programming language and developed under the GNU/Linux operating system. Our open-source implementation is made available at https://github.com/maialzamel/k-map under the GNU General Public License.

Our task in this section is to evaluate the performance of our implementation with respect to the performance of the implementation provided in [4]; we call our implementation `k-map` and the one of [4] `Gemtool`. Let us stress, however, that `Gemtool` is a heuristic algorithm as opposed to `k-map`, which is an exact algorithm: it always returns the correct solution.

As input we used sequences extracted from a real DNA corpus ranging in length from 1 MB to 512 MB. This DNA corpus is available at http://pizzachili.dcc.uchile.cl/texts/dna/. For each input sequence we used different values for m and k. All experiments have been conducted on a Desktop PC using one core of Intel Core CPU i5-4690 at 3.50 GHz. Both implementations were compiled with `g++` version 6.2.0 at optimization level 3 (-O3).

The experimental results (recorded elapsed times and memory usage) are depicted in Figs. 3 and 4:

1. For fixed values of k and m, our implementation requires time linear in n up until a certain value of n (see Theorem 2—notice that the restriction is not exactly the one stated as the input is not uniformly random). After that n value, the performance of `k-map` starts approaching the performance of `Gemtool`, which eventually becomes faster.
2. For fixed values of n, our implementation becomes considerably faster with increasing values of m (see Theorem 2).
3. The memory usage of our implementation grows linearly with n (see Theorem 2). The memory usage of `Gemtool` grows also linearly with n but with a lower constant factor.

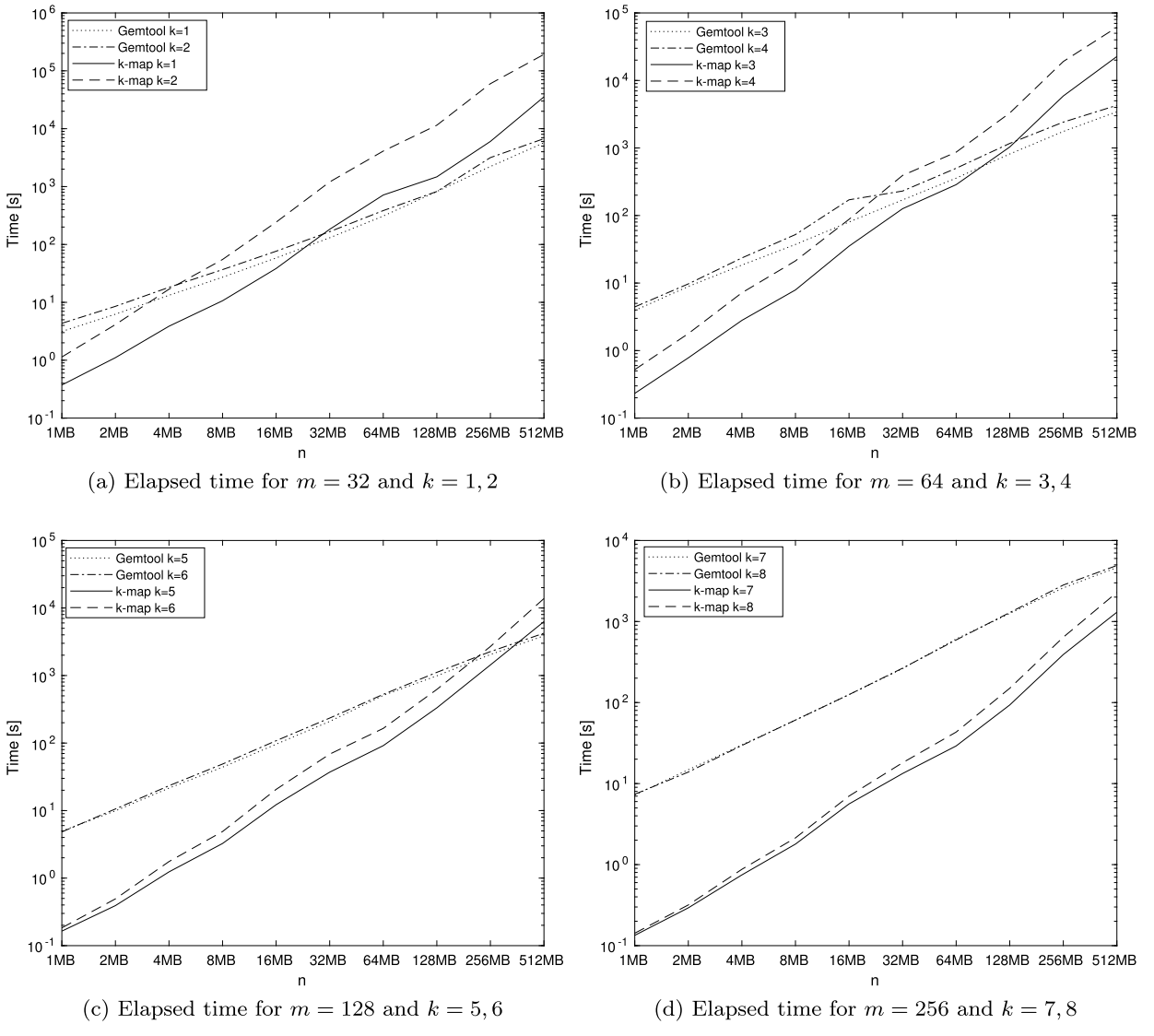
6. Final remarks

The k-mappability problem can be solved in $\mathcal{O}(\min\{nm^k, n \log^{k+1} n\})$ time and $\mathcal{O}(n)$ space for $k = \mathcal{O}(1)$ and constant-sized alphabets [7]. In this paper, we investigated the special case of $k = 1$. We presented an algorithm that requires $\mathcal{O}(\min\{nm, n \log n \log\log n\})$ time and $\mathcal{O}(n)$ space for this special case.

We also presented another algorithm that requires average-case time and space $\mathcal{O}(n)$ for integer alphabets of size σ if $m = \Omega(\log_\sigma n)$, and showed that this algorithm is generalizable for arbitrary k, requiring average-case time $\mathcal{O}(kn)$ and space $\mathcal{O}(n)$ if $m = \Omega(k \log_\sigma n)$. We have provided an open-source implementation of this algorithm and also experimental results demonstrating its competitiveness to the state-of-the-art implementation [4].

Let us note that it seems possible to apply the technique of Thankachan et al. [20] to obtain $\mathcal{O}(\min\{nm, n \log n\})$ time and $\mathcal{O}(n)$ space for when $k = 1$ (for a preliminary exposition of the ideas, see [21]). We leave as an open question whether there exists an $o(n \log n)$-time algorithm for the 1-mappability problem. To this end, the technique of Charalampopoulos et al. [22] may prove useful.

Please cite this article in press as: M. Alzamel et al., Faster algorithms for 1-mappability of a sequence, Theoret. Comput. Sci. (2019), https://doi.org/10.1016/j.tcs.2019.04.026



(a) Elapsed time for $m = 32$ and $k = 1, 2$

(b) Elapsed time for $m = 64$ and $k = 3, 4$

(c) Elapsed time for $m = 128$ and $k = 5, 6$

(d) Elapsed time for $m = 256$ and $k = 7, 8$

Fig. 3. Elapsed-time comparison between `k-map` and `Gemtool`.

Another direction is to consider the k-mappability problem under the edit distance model. In this model, a decision needs to be made whether sufficiently similar factors only of length exactly m or of all lengths between $m - k$ and $m + k$ should be counted. The techniques presented recently in [23,9] may prove useful for counting. We also leave this problem for future investigation.

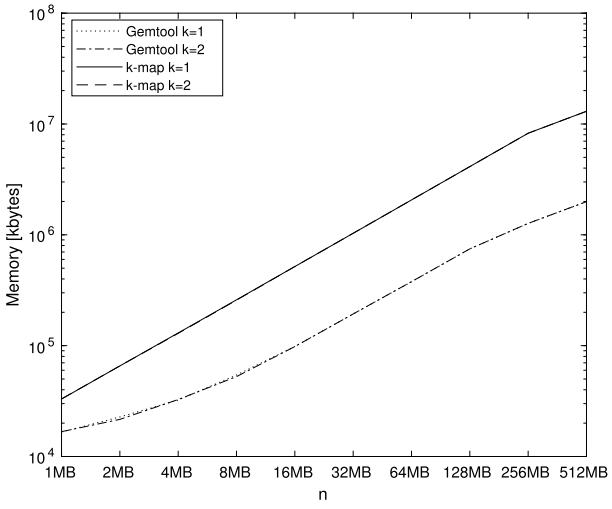
Declaration of Competing Interest

None declared.

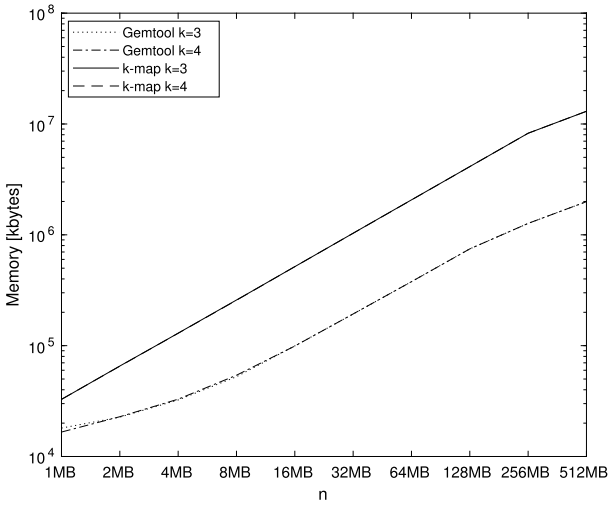
Acknowledgements

We warmly thank Szymon Grabowski who drew our attention via personal communication to Remark 6 and reference [18]; the latter reduced the complexity of the algorithm described in Section 4.2 from $\mathcal{O}(n \log^2 n)$ to $\mathcal{O}(n \log n \log \log n)$. Mai Alzamel was fully supported by the Ministry of Education – Kingdom of Saudi Arabi. Panagiotis Charalampopoulos was partially supported by the Graduate Teaching Scholarship scheme of the Department of Informatics at King's College London and an A.G. Leventis Foundation Educational Grant. Jakub Radoszewski was supported by the "Algorithms for text processing with errors and uncertainties" project carried out within the HOMING programme of the Foundation for Polish Science co-financed by the European Union under the European Regional Development Fund.

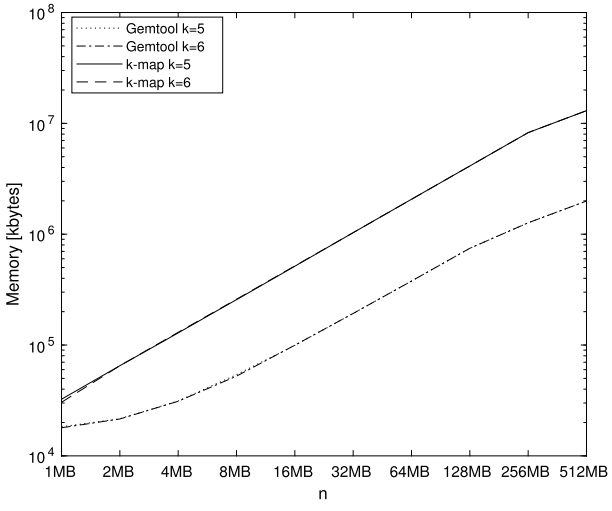
Please cite this article in press as: M. Alzamel et al., Faster algorithms for 1-mappability of a sequence, Theoret. Comput. Sci. (2019), https://doi.org/10.1016/j.tcs.2019.04.026



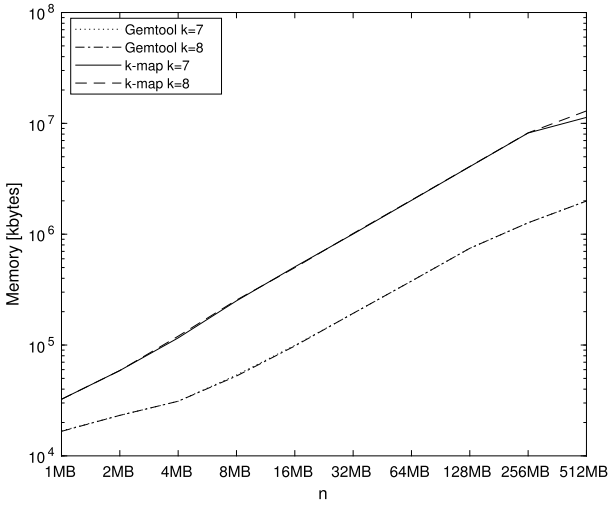
(a) Memory usage for $m = 32$ and $k = 1, 2$



(b) Memory usage for $m = 64$ and $k = 3, 4$



(c) Memory usage for $m = 128$ and $k = 5, 6$



(d) Memory usage for $m = 256$ and $k = 7, 8$

Fig. 4. Memory-usage comparison between `k-map` and `Gemtool`.

References

[1] M. Alzamel, P. Charalampopoulos, C.S. Iliopoulos, S.P. Pissis, J. Radoszewski, W. Sung, Faster algorithms for 1-mappability of a sequence, in: X. Gao, H. Du, M. Han (Eds.), Combinatorial Optimization and Applications - Proceedings of the 11th International Conference, Part II, COCOA 2017, Shanghai, China, December 16-18, 2017, in: Lecture Notes in Computer Science, vol. 10628, Springer, 2017, pp. 109–121.

[2] M.L. Metzker, Sequencing technologies – the next generation, Nat. Rev. Genet. 11 (1) (2010) 31–46, https://doi.org/10.1038/nrg2626.

[3] N.A. Fonseca, J. Rung, A. Brazma, J.C. Marioni, Tools for mapping high-throughput sequencing data, Bioinformatics 28 (24) (2012) 3169–3177, https://doi.org/10.1093/bioinformatics/bts605.

[4] T. Derrien, J. Estellé, S. Marco Sola, D. Knowles, E. Raineri, R. Guigó, P. Ribeca, Fast computation and applications of genome mappability, PLoS ONE 7 (1) (2012) 1–16, https://doi.org/10.1371/journal.pone.0030377.

[5] P. Antoniou, J.W. Daykin, C.S. Iliopoulos, D. Kourie, L. Mouchard, S.P. Pissis, Mapping uniquely occurring short sequences derived from high throughput technologies to a reference genome, in: 2009 9th International Conference on Information Technology and Applications in Biomedicine, IEEE Computer Society, 2009, pp. 1–4.

[6] J. Fischer, Inducing the LCP-array, in: F. Dehne, J. Iacono, J. Sack (Eds.), Algorithms and Data Structures - Proceedings of the 12th International Symposium, WADS 2011, in: Lecture Notes in Computer Science, vol. 6844, Springer, 2011, pp. 374–385.

[7] M. Alzamel, P. Charalampopoulos, C.S. Iliopoulos, T. Kociumaka, S.P. Pissis, J. Radoszewski, J. Straszynski, Efficient computation of sequence mappability, in: T. Gagie, A. Moffat, G. Navarro, E. Cuadros-Vargas (Eds.), String Processing and Information Retrieval - Proceedings of the 25th International Symposium, SPIRE 2018, Lima, Peru, October 9-11, 2018, in: Lecture Notes in Computer Science, vol. 11147, Springer, 2018, pp. 12–26.

[8] H. Alamro, L.A.K. Ayad, P. Charalampopoulos, C.S. Iliopoulos, S.P. Pissis, Longest common prefixes with k-mismatches and applications, in: A.M. Tjoa, L. Bellatreche, S. Biffl, J. van Leeuwen, J. Wiedermann (Eds.), SOFSEM 2018: Theory and Practice of Computer Science - Proceedings of the 44th International Conference on Current Trends in Theory and Practice of Computer Science, Krems, Austria, January 29-February 2, 2018, in: Lecture Notes in Computer Science, vol. 10706, Springer, 2018, pp. 636–649.

Please cite this article in press as: M. Alzamel et al., Faster algorithms for 1-mappability of a sequence, Theoret. Comput. Sci. (2019), https://doi.org/10.1016/j.tcs.2019.04.026

[9] L.A.K. Ayad, C. Barton, P. Charalampopoulos, C.S. Iliopoulos, S.P. Pissis, Longest common prefixes with k-errors and applications, in: T. Gagie, A. Moffat, G. Navarro, E. Cuadros-Vargas (Eds.), String Processing and Information Retrieval - Proceedings of the 25th International Symposium, SPIRE 2018, Lima, Peru, October 9-11, 2018, in: Lecture Notes in Computer Science, vol. 11147, Springer, 2018, pp. 27–41.

[10] G. Manzini, Longest common prefix with mismatches, in: C.S. Iliopoulos, S.J. Puglisi, E. Yilmaz (Eds.), String Processing and Information Retrieval - Proceedings of the 22nd International Symposium, SPIRE 2015, in: Lecture Notes in Computer Science, vol. 9309, Springer, 2015, pp. 299–310.

[11] U. Manber, E.W. Myers, Suffix arrays: a new method for on-line string searches, SIAM J. Comput. 22 (5) (1993) 935–948, https://doi.org/10.1137/0222058.

[12] G. Nong, S. Zhang, W.H. Chan, Linear suffix array construction by almost pure induced-sorting, in: J.A. Storer, M.W. Marcellin (Eds.), Data Compression Conference, DCC 2009, IEEE Computer Society, 2009, pp. 193–202.

[13] M.A. Bender, M. Farach-Colton, The LCA problem revisited, in: G.H. Gonnet, D. Panario, A. Viola (Eds.), LATIN 2000: Theoretical Informatics, Proceedings of the 4th Latin American Symposium, 2000, in: Lecture Notes in Computer Science, vol. 1776, Springer, 2000, pp. 88–94.

[14] M. Farach, Optimal suffix tree construction with large alphabets, in: 38th Annual Symposium on Foundations of Computer Science, FOCS '97, IEEE Computer Society, 1997, pp. 137–143.

[15] M.L. Fredman, J. Komlós, E. Szemerédi, Storing a sparse table with O(1) worst case access time, J. ACM 31 (3) (1984) 538–544, https://doi.org/10.1145/828.1884.

[16] M. Crochemore, G. Tischler, The gapped suffix array: a new index structure for fast approximate matching, in: E. Chávez, S. Lonardi (Eds.), String Processing and Information Retrieval - Proceedings of the 17th International Symposium, SPIRE 2010, in: Lecture Notes in Computer Science, vol. 6393, Springer, 2010, pp. 359–364.

[17] A. Amir, G.M. Landau, M. Lewenstein, D. Sokol, Dynamic text and static pattern matching, ACM Trans. Algorithms 3 (2) (2007) 19, https://doi.org/10.1145/1240233.1240242.

[18] J. Fischer, D. Köppl, F. Kurpicz, On the benefit of merging suffix array intervals for parallel pattern matching, in: R. Grossi, M. Lewenstein (Eds.), 27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016, in: LIPIcs, vol. 54, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016, pp. 26:1–26:11.

[19] R. Cole, L. Gottlieb, M. Lewenstein, Dictionary matching and indexing with errors and don't cares, in: L. Babai (Ed.), Proceedings of the 36th Annual ACM Symposium on Theory of Computing, 2004, ACM, 2004, pp. 91–100.

[20] S.V. Thankachan, A. Apostolico, S. Aluru, A provably efficient algorithm for the k-mismatch average common substring problem, J. Comput. Biol. 23 (6) (2016) 472–482, https://doi.org/10.1089/cmb.2015.0235.

[21] S. Hooshmand, P. Abedin, D. Gibney, S. Aluru, S.V. Thankachan, Faster computation of genome mappability, in: A. Shehu, C.H. Wu, C. Boucher, J. Li, H. Liu, M. Pop (Eds.), Proceedings of the 2018 ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics, BCB 2018, Washington, DC, USA, August 29-September 01, 2018, ACM, 2018, p. 537.

[22] P. Charalampopoulos, M. Crochemore, C.S. Iliopoulos, T. Kociumaka, S.P. Pissis, J. Radoszewski, W. Rytter, T. Walen, Linear-time algorithm for long LCF with k mismatches, in: G. Navarro, D. Sankoff, B. Zhu (Eds.), Annual Symposium on Combinatorial Pattern Matching, CPM 2018, July 2-4, 2018 - Qingdao, China, in: LIPIcs, vol. 105, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018, pp. 23:1–23:16.

[23] S.V. Thankachan, C. Aluru, S.P. Chockalingam, S. Aluru, Algorithmic framework for approximate matching under bounded edits with applications to sequence analysis, in: B.J. Raphael (Ed.), Research in Computational Molecular Biology - Proceedings of the 22nd Annual International Conference, RECOMB 2018, Paris, France, April 21-24, 2018, in: Lecture Notes in Computer Science, vol. 10812, Springer, 2018, pp. 211–224.

Please cite this article in press as: M. Alzamel et al., Faster algorithms for 1-mappability of a sequence, Theoret. Comput. Sci. (2019), https://doi.org/10.1016/j.tcs.2019.04.026