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

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

The exact multiple pattern matching problem solved by a reference tree approach

Yi-Kung Shieh [a], Shyong Jian Shyu [b,*,1], Chin Lung Lu [a], Richard Chia-Tung Lee [a]

[a] Department of Computer Science, National Tsing Hua University, Hsinchu, 30013, Taiwan, ROC
[b] Department of Computer Science and Information Engineering, Ming Chuan University, Taoyuan, 33348, Taiwan, ROC

ARTICLE INFO

Article history:
Received 5 August 2019
Received in revised form 25 January 2021
Accepted 1 June 2021
Available online xxxx

Keywords:
Exact multiple pattern matching
Reference tree
Reference string
DNA sequence
Suffix array
Suffix tree

ABSTRACT

Given a text T and a set of r patterns $P_1, P_2, \ldots, P_r$, the exact multiple pattern matching problem reports the ending positions of all occurrences of P_i in T for $1 \leq i \leq r$. By transforming all substrings with a fixed length of T into a reference tree such that each internal node stores a reference string, the exact multiple pattern matching problem can be efficiently solved by searching patterns in the tree via the guidance of the reference strings. We design elegant algorithms to construct the reference tree (the preprocessing phase) and to search patterns in the tree (the searching phase) using bitwise operations. The experiments involving problem instances from the DNA sequence and the English language are conducted to compare the performance of our approach against those of the suffix tree and suffix array algorithms. The computational results demonstrate the advantage of our approach over these algorithms. In spite of the simplicity, our approach is quite efficient, flexible and robust.

1. Introduction

The *exact multiple pattern matching* (EMPM) is one of the classical and significant research topics in computer science. Its applications cover diverse areas such as information security, image processing, information retrieval, operating system and computational molecular biology [7,16,24].

Given a text string $T = t_1 t_2 \ldots t_n$ and r pattern strings $P_1, P_2, \ldots, P_r$ over alphabet Σ, EMPM reports all occurrences of P_i in T for $1 \leq i \leq r$. In this paper, we are interested in the situation that there would be a very large number of pattern strings. A typical example arises in the DNA sequencing problem. At first, a target unknown DNA sequence is delivered million short reads by Next Generation Sequencing (NGS) technology [5,19] and the short reads are reassembled to continuous segments (*contigs*) by the overlapping regions [4,20,37]. Then, contigs are scaffolded by aligning these contigs with a reference DNA [3,10,30] and the gap regions are closed by remapping the paired-end reads back to the scaffolds [22,27,28]. In the process of the alignment, we need to determine the positions of all occurrences of contigs, usually more than thousands, on the reference DNA. As other examples, we may try to find where certain phrases appear in the Bible (classic novels/speeches), or to ensure that our computer files are not infected with viruses, worms, Trojan horses, and so on. These manifest the practical applications of the EMPM problem. To cope with the problem, it is worthwhile to conduct a preprocessing on the reference DNA (Bible, files, etc.) such that the later searching for contigs (phrases, viruses, etc.) will be very fast.

* Corresponding author.
E-mail addresses: d9762814@oz.nthu.edu.tw (Y.-K. Shieh), sjshyu@mail.mcu.edu.tw (S.J. Shyu), cllu@cs.nthu.edu.tw (C.L. Lu), rctlee@rctlee.cyberhood.net.tw (R.C.-T. Lee).

[1] This research was supported in part by the Ministry of Science and Technology, Taiwan, under Grants MOST 109-2221-E-130-010.

https://doi.org/10.1016/j.tcs.2021.06.003
0304-3975/

The most popular methods based on preprocessing (text T) and searching to solve the EMPM problem are the *suffix tree* or *suffix array* algorithms. The suffix tree, proposed by Weiner [36], is a universal data structure to deal with many problems in stringology. Later, Ukkonen provided the first online-construction of the suffix tree [35]. The time and space complexities of constructing the suffix tree are both $O(n)$. To find the exact matches of pattern P_i for $1 \le i \le r$ in the suffix tree can be accomplished in $O(|P_i| + occ)$ time where $|P_i|$ is the length of P_i and occ is the number of occurrences. Manber and Myer introduced the suffix array [21], which is an alternative to the suffix tree. The time and space complexities of building the suffix array are also $O(n)$ [18]. Utilizing the auxiliary array (*longest common prefix* (LCP) array), the searching algorithm in the suffix array reports the positions of all occurrences of pattern P_i for $1 \le i \le r$ in $O(|P_i| + occ + \log n)$ time [17].

Many researchers focused on refining the suffix array and suffix tree algorithms. The most successful results include *compressed suffix array* and *compressed suffix tree*, which reduce the space requirements of the suffix array and suffix tree, respectively. It is noticed that Abouelhoda et al. incorporated suffix array, LCP array and *lcp-interval tree* to simulate all kinds of suffix tree traversals very efficiently [1]. In the compressed suffix array algorithms proposed by Sadakane [32], Grossi et al. [13] and Ferragina et al. [9], the space requirements are $(1/\epsilon)nH_0$+$O(n \log H_0)$, $nH_k \log\log_\sigma n$+$O(n)$ and nH_k+$O(n \log\sigma / \log^\epsilon n)$ bits, respectively, where $\epsilon > 0$, σ is the size of alphabet Σ and H_k denotes the kth-order entropy of text T, and the access times are $O(occ \log^\epsilon n)$, $O(occ(\log\log_\sigma n + \log\sigma))$ and $O(occ \log^{1+\epsilon} n)$, respectively. Regarding the compressed suffix tree, the algorithms devised by Sadakane [33] and Russo et al. [31] need $nH_k + 6n$+$O(n \log\sigma)$ and nH_k+$O(n \log\sigma)$ spaces, respectively, and both the locate times are in $O((\log_\sigma \log n) \log n)$.

In this paper, we propose a novel *reference tree* approach to deal with the EMPM problem. We first construct the reference tree with respect to text T and then determine the exact matches by searching all patterns in the tree. Each internal node of the tree consists of a *reference string* so that the search for the patterns starting from the root node could be guided by the reference strings either to some leaf node where the exact matches could be decided; or to some internal node where the occurrence of the pattern P in T could be entirely denied.

The rest of the paper is organized as follows. In Section 2, we define the reference tree and present our designs of the preprocessing (for constructing the reference tree) and searching (for determining the exact matches of the given patterns via the tree) algorithms. Experimental results for real data of the DNA sequence and the English language are summarized and discussed in Section 3. We conclude this research in Section 4. The expected time and space complexities are analyzed in Appendix A. Experimental results for random text and patterns simulating the DNA sequence and the English language are presented and discussed in Appendix B.

2. Reference tree approach

Let the text $T = t_1 t_2 \dots t_n$ and r patterns $P_1, P_2, \dots, P_r$ of lengths $m_1, m_2, \dots, m_r$, respectively, be strings over alphabet Σ. Let $|T|$ be the length of T (i.e., $n = |T|$) and σ be the size of Σ (i.e., $\sigma = |\Sigma|$). The EMPM problem is to find the ending positions of all occurrences of P_i in T for $1 \le i \le r$. That is, we have to report all locations j's such that $T[j - m_i + 1, j]$ is equal to P_i where $T[j - m_i + 1, j]$ denotes the substring $t_{j-m_i+1} t_{j-m_i+2} \dots t_j$ for $m_i \le j \le n$ and $1 \le i \le r$.

We observe that determining whether P_i appears in T is less efficient than determining whether a prefix of P_i of length, say ℓ ($< m_i$), appears in T. Let such a prefix be referred to as an *ℓ-prefix*. Thus, the exact matches of P_i in T may be decided in two stages: (1) resolving whether or not the ℓ-prefix of P_i appears in T; (2) verifying whether the rest part of P_i also occurs in T, if (1) holds. The first is called the *candidate-finding* stage, while the second the *verifying* stage. Apparently, if the former does not obtain any candidate, the latter can be ignored.

Let a substring in T of length ℓ be referred to as an *ℓ-substring*. Suppose that we have a proper structure to keep all ℓ-substrings of T and an elegant algorithm for answering whether the ℓ-prefixes of the patterns occur in the structure. It would be of great benefit to accomplish the candidate-finding stage. Such ideas inspire us to construct a reference tree in order to keep all ℓ-substrings of T based on which our searching algorithm could resolve the existence of the ℓ-prefixes of the patterns efficiently. The verifying stage, only if any candidate has been found, requires merely a special pattern matching algorithm to verify whether exact matches exist for the rest parts of the patterns. Note that our idea demands that ℓ should be no greater than the length of any pattern. Otherwise, our approach is not working. Suppose that we know some small-length patterns exist in advance, the choice of ℓ should be accordingly adjusted before the construction of the reference tree.

2.1. Keeping ℓ-substrings of text in reference tree

Let s_i denote the ℓ-substring $T[i, i + \ell - 1]$ for $1 \le i \le n - \ell + 1$. We collect all ℓ-substrings in T into a set S, i.e., $S = \{s_1, s_2, \dots, s_{n-\ell+1}\}$. In the physical realization, S can be easily represented by the starting positions (i.e., $1, 2, \dots, n - \ell + 1$) of the ℓ-substrings in T. Our reference tree aims at storing all elements in S. Conceptually, each internal node simply keeps one ℓ-substring (physically, a starting position in T), called the *reference string*, based on which at most $\ell + 1$ subtrees would be expanded where the related ℓ-substrings are stored accordingly. Another parameter k, designated the maximum number of ℓ-substrings stored in a leaf, is introduced to adjust the flexibility of the expansion of the tree. Given the two pre-defined parameters ℓ and k, the reference tree of T, denoted as $R_{\ell,k}$, is characterized as follows:

(1) $R_{\ell,k}$ is an $(\ell + 1)$-nary tree rooted at node N.

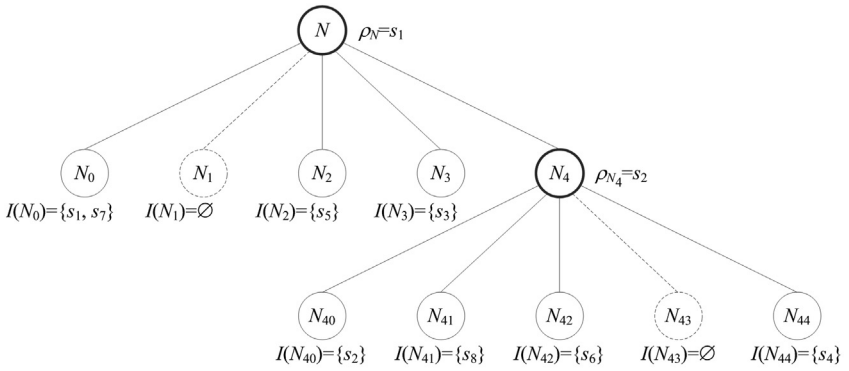

Fig. 1. Reference tree $R_{4,3}$ of T = GAGTCAGAGTA.

(2) Each node X in $R_{\ell,k}$ is associated with a constructing data set, denoted as $I(X)$ ($\subseteq S$). For root node N, $I(N) = S$.
(3) If $|I(X)| \leq k$ or all elements in $I(X)$ are identical, X becomes a leaf node where all strings in $I(X)$ are stored. Otherwise, it is an internal node.
(4) Each internal node X, whose children are named as $X_0, X_1, \ldots, X_\ell$, contains a reference string, denoted as ρ_X, chosen from $I(X)$. Those strings in $I(X)$ whose distances from ρ_X are d would be dispatched to the subtree rooted at X_d, which is also a reference tree and its constructing data set is $I(X_d)$ for $0 \leq d \leq \ell$.

The measurement of the distance between two strings in this paper is the Hamming distance [14]. Since the Hamming distance between two ℓ-strings ranges from 0 to ℓ, each internal node X dealing with data $I(X)$ (where $|I(X)| > k$) with respect to ρ_X has at most $\ell + 1$ subtrees rooted at $X_0, X_1, \ldots, X_\ell$ (as characterized by (1) and (4)) where constructing data $I(X_0), I(X_1), \ldots, I(X_\ell)$ are respectively dispatched. X_0 is definitely a leaf node (as characterized by (3)) so that it stores $I(X_0)$, whose members are exactly identical to ρ_X. Those X_i's with $|I(X_i)| \leq k$ are also leaf nodes for $1 \leq i \leq \ell$ (by (3), too). It is probable for some X_i's to be empty nodes as long as $|I(X_i)| = 0$.

Initially, $I(N) = S$ for root node N. For each internal node X, we merely choose the substring with the smallest staring position in $I(X)$ as ρ_X. It is not hard to see that parameter ℓ (k) affects the width (height) of the reference tree. In general, for a fixed ℓ (k), a smaller k (ℓ) results in a higher (narrower) tree. The combinations of these two parameters provide a broad range of probable reference trees.

Let us resort to an example. Consider T = GAGTCAGAGTA, $\ell = 4$ and $k = 3$. The reference tree $R_{4,3}$ of T is shown in Fig. 1. The 4-substrings of T are s_1 = GAGT, s_2 = AGTC, s_3 = GTCA, s_4 = TCAG, s_5 = CAGA, s_6 = AGAG, s_7 = GAGT and s_8 = AGTA, and thus $I(N) = \{s_1, s_2, \ldots, s_8\}$ ($\{1, 2, \ldots, 8\}$ for a physical realization). In N, s_1 is chosen to be the reference string (i.e., $\rho_N = s_1$). The distances between $s_1, s_2, \ldots, s_8$ and s_1 are 0, 4, 3, 4, 2, 4, 0 and 4, respectively. Hence, we obtain that $I(N_0) = \{s_1, s_7\}$, $I(N_1) = \emptyset$, $I(N_2) = \{s_5\}$, $I(N_3) = \{s_3\}$ and $I(N_4) = \{s_2, s_4, s_6, s_8\}$. Apparently, N_1 is an empty node, while N_0, N_2 and N_3 are leaf nodes (see character (3) of the reference tree) where the corresponding data sets are stored. Owing to $|I(N_4)| = 4$ (which is greater than k), N_4 becomes an internal node which needs to be expanded. The chosen reference string is $\rho_{N_4} = s_2$. The distances between the members in $I(N_4)$ (i.e., s_2, s_4, s_6 and s_8) and ρ_{N_4} are 0, 4, 2, and 1, respectively. We have $I(N_{40}) = \{s_2\}$, $I(N_{41}) = \{s_8\}$, $I(N_{42}) = \{s_6\}$, $I(N_{43}) = \emptyset$ and $I(N_{44}) = \{s_4\}$. Now, all child nodes of N_4 are leaves. The reference tree of T is obtained.

One critical part in applying the reference tree is to compute the Hamming distances between all strings in the constructing data and the reference string in an internal node. It would be very time-consuming if the distances are computed by the simply linear scan method. For improving the performance, we transfer the strings over alphabet Σ into bit strings and apply bitwise operations to accelerate the distance computations. Consider symbol $a_i \in \Sigma$ for $1 \leq i \leq \sigma$. We denote a_i by bit string α_i of $1 + \lceil \log_2 \sigma \rceil$ bits where the $\lceil \log_2 \sigma \rceil$ bits distinguish the σ symbols and the extra leading bit, called the *witness bit*, aims at detecting whether a carry exists after a bitwise operation. For example, assume that alphabet $\Sigma = \{a_1, a_2, a_3, a_4\}$ = {A, C, G, T}. The corresponding bit strings of symbols A, C, G and T are $\alpha_1 = 000$, $\alpha_2 = 001$, $\alpha_3 = 010$ and $\alpha_4 = 011$, respectively, where all leading witness bits are 0's initially. Hence, T can be transferred to bit string B with $n\lambda$ bits or $\lceil n\lambda/\omega \rceil$ words where $\lambda = 1 + \lceil \log_2 \sigma \rceil$ and ω is the size of a computer word (i.e., ω = 32 or 64 (bits)). For each ℓ-substring $s_i = T[i, i + \ell - 1]$ where $1 \leq i \leq n - \ell + 1$, its bit string b_i consists of $\ell\lambda$ bits starting from position $(i - 1)\lambda + 1$ and ending to position $(i + \ell - 1)\lambda$ in B. Fig. 2 depicts the relationship between s_i in T and the corresponding b_i in B.

Let β_X be the bit string of reference string ρ_X in internal node X. Let $b_{witness} = (1(0)^{\lceil \log_2 \sigma \rceil})^\ell$ be the *witness string* and $b_{inv\_witness} = (0(1)^{\lceil \log_2 \sigma \rceil})^\ell$ be the *inverse witness string*. The Hamming distance between s_i and ρ_X can be computed by the corresponding bit strings b_i and β_X as follows:

$$popcount(((b_i \oplus \beta_X) + b_{inv\_witness}) \wedge b_{witness}) \tag{1}$$

where $\oplus$, $+$ and $\wedge$ are the "exclusive or" (XOR), "ADD" and "AND" bitwise operations, respectively, and $popcount(b)$ is the method to count the number of 1's of bit string b [15]. Consider the case of $\sigma = 4$ and $\ell = 3$. We have $b_{witness}$ = 100100100

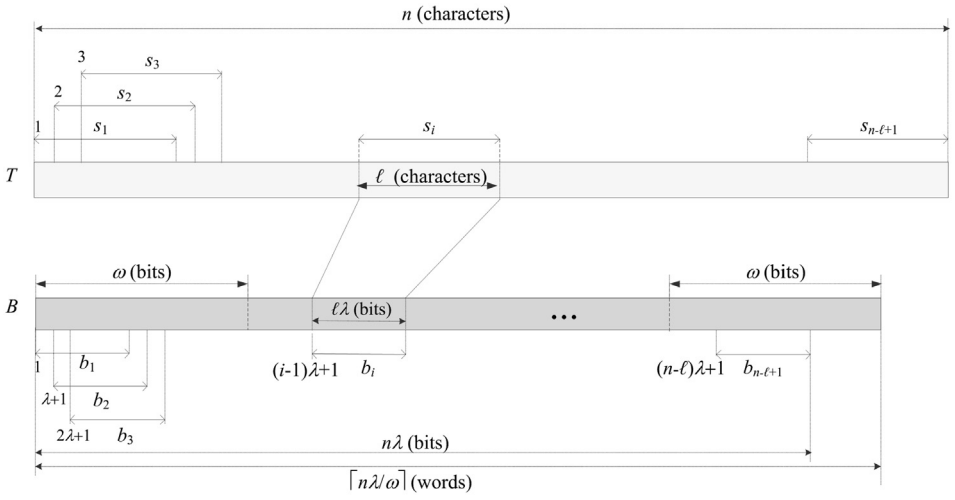

Fig. 2. Relationship between s_i in T and the corresponding b_i in B.

and $b_{inv\_witness} = 011011011$. The locations of 1's in $b_i \oplus \beta_X$ indicate the positions of b_i and β_X in which the corresponding bits are different. $(b_i \oplus \beta_X) + b_{inv\_witness}$ makes the 1's of $(b_i \oplus \beta_X)$ be summed up to the corresponding carries (witnesses). Next, the bits which are not witnesses will be cleared by $((b_i \oplus \beta_X) + b_{inv\_witness}) \wedge b_{witness}$. Finally, the number of 1's in the resultant bit string is exactly the Hamming distance between s_i and ρ_X, which is counted by the *popucount* method. When the size of each b_i (and β_X) where $1 \le i \le n - \ell + 1$ is smaller than or equal to a computer word, Equation (1) counts the number of 1's in $O(1)$.

The algorithm for constructing the reference tree is presented in Algorithm 1. The tree is constructed in a breadth-first-search manner so that a queue Q is adopted.

2.2. Finding occurrences of patterns via reference tree

All ℓ-substrings of T are recoded in reference tree $R_{\ell,k}$ of T by Algorithm 1. The following property signifies the effectiveness of searching patterns in the reference tree.

Property 1. *Let a, b and c be strings with the same length ℓ. Let $\delta(a, b)$ denote the Hamming distance between a and b where $0 \le \delta(a, b) \le \ell$. We have*

(1) $a \ne c$ if $\delta(a, b) \ne \delta(c, b)$, and
(2) a may be equal to c if $\delta(a, b) = \delta(c, b)$.

Consider pattern P and its ℓ-prefix p as well as internal node X and its reference string ρ_X in $R_{\ell,k}$. In the candidate-finding stage, by Property 1(1), if $\delta(p, \rho_X) = d$ where $0 \le d \le \ell$, it is impossible for p to be in the subtree rooted at $X_{d'}$ where $d' \ne d$. In fact, all the ℓ-substrings in the subtree rooted at X_d have the same distance from ρ_X. Hence, the search for the matches of p (and subsequently, P) just needs to be considered in X_d by Property 1(2). In other words, the ℓ-substrings in subtree X_d are the candidates of p (and subsequently, P). With the guidance of the reference strings along the internal nodes from the root, the candidates could be percolated to some leaf and lots of branches containing non-candidates can be pruned away. Further verification for the exact matches of P only takes in a particular leaf. Surely, the computation of $\delta(p, \rho_X)$ is also accomplished by Equation (1) (i.e., $popcount(((b_p \oplus \beta_X) + b_{inv\_witness}) \wedge b_{witness})$ where b_p and β_X are the bit representations of p and ρ_X, respectively).

On condition that X_d does not exist, we merely stop searching for P. If X_d is not empty and $d = 0$ (an extreme case that all strings in $I(X_0)$ are identical to ρ_X, and consequently equal to the ℓ-prefix of P), the verifying phase goes on: For each $i \in I(X_0)$, we examine whether $T[i + \ell, i + m - 1] = P[\ell + 1, m]$ by linear comparison (character by character) until a mismatch occurs or they exactly match where m is the length of P. Whenever an exact match occurs, the ending position $i + m - 1$ would be collected as one of the solutions with respect to P for all i's in $I(X_0)$.

Suppose that X_d is again an internal node where $I(X_d) \ne \emptyset$ and $d \ne 0$. The search goes on until we reach a leaf node Y. The verifying phase is applied in Y by examining whether $T[i, i + m - 1]$ is equal to P or not for $i \in I(Y)$ character by character. Whenever $T[i, i + m - 1]$ exactly matches with P, the ending position $i + m - 1$ would be reported for all i's in $I(Y)$.

The searching for pattern P in the reference tree is shown in Algorithm 2. By utilizing Algorithm 2 for all patterns, the EMPM problem can be solved.

Algorithm 1 Constructing the reference tree of T.

Input: Text string T over alphabet $\Sigma = \{a_1, a_2, \ldots, a_\sigma\}$, and parameters ℓ and k
Output: Bit string B with respect to T and reference tree $R_{\ell,k}$ of T

for (each symbol $a_i \in \Sigma$ for $1 \le i \le \sigma$) **do**
  $\alpha_i \leftarrow 0w$ where w is the binary representation of $i-1$ with $\lceil \log_2 \sigma \rceil$ bits
end for
Obtain bit string B with respect to T
$b_{inv\_witness} \leftarrow (0(1)^{\lceil \log_2 \sigma \rceil})^\ell$ and $b_{witness} \leftarrow (1(0)^{\lceil \log_2 \sigma \rceil})^\ell$
$S \leftarrow \{i \mid 1 \le i \le n - \ell + 1\}$
Create node N to be the root node of $R_{\ell,k}$
Create queue Q which records the nodes to be expanded
Enqueue the pair (N, S) into Q where S is the constructing data of N (i.e., $I(N) \leftarrow S$)
while ($Q \neq \emptyset$) **do**
  Dequeue a pair $(X, I(X))$ from Q
  if ($|I(X)| > k$) **then**
  // X is an internal node contains ρ_X and child nodes $(X_0, X_1, \ldots, X_\ell)$
    Mark X as an internal node
    Choose s_i to be ρ_X where i is the smallest integer in $I(X)$
    Obtain bit string β_X with respect to ρ_X
    for (each j, $0 \le j \le \ell$) **do** $S_j \leftarrow \emptyset$
    for (each element i, $i \in I(X)$) **do**
      $d \leftarrow popcount(((b_i \oplus \beta_X) + b_{inv\_witness}) \wedge b_{witness})$
      $S_d \leftarrow S_d \cup \{i\}$
    end for
    for (each j, $0 \le j \le \ell$) **do**
      if ($S_j \neq \emptyset$) **then**
        Create node X_j to be the child node of X
        if ($j = 0$) **then**
          Mark X_0 as a leaf node storing all members in S_0
        else
          Enqueue the pair (X_j, S_j) into Q
        end if
      end if
    end for
  else
  // X is a leaf node since $|I(X)| \le k$
    Mark X as a leaf node storing all members in $I(X)$
  end if
end while
return bit string B and reference tree $R_{\ell,k}$ of T rooted at N

Algorithm 2 Searching a pattern in the reference tree.

Input: Pattern string $P = p_1 p_2 \ldots p_m$, text string T, parameters ℓ and k, reference tree $R_{\ell,k}$ of T rooted at N and bit string B with respect to T
Output: Ending positions of all occurrences of P in T

Obtain the bit string b_p of the ℓ-prefix p of P
$search\_flag \leftarrow$**true**
$X \leftarrow N$, $b_{inv\_witness} \leftarrow (0(1)^{\lceil \log_2 \sigma \rceil})^\ell$ and $b_{witness} \leftarrow (1(0)^{\lceil \log_2 \sigma \rceil})^\ell$
while ($search\_flag$ is **true**) **do**
  if (X has child nodes) **then**
  // X is an internal node (the candidate-finding stage)
    $d \leftarrow popcount(((b_p \oplus \beta_X) + b_{inv\_witness}) \wedge b_{witness})$
    if ($d = 0$) **then** // X_0 is a leaf node (the verifying stage)
      for (each $i \in I(X_0)$) **do**
        if ($T[i + \ell, i + m - 1] = P[\ell + 1, m]$) **then output** position $i + m - 1$
      end for
      $search\_flag \leftarrow$**false**
    else if ($I(X_d) = \emptyset$) **then**
      $search\_flag \leftarrow$**false**
    else
      $X \leftarrow X_d$ // X is updated by the child node X_d
    end if
  else
  // X is a leaf node (the verifying stage)
    for (each $i \in I(X)$) **do**
      if ($T[i, i + m - 1] = P[1, m]$) **then output** position $i + m - 1$
    end for
    $search\_flag \leftarrow$**false**
  end if
end while

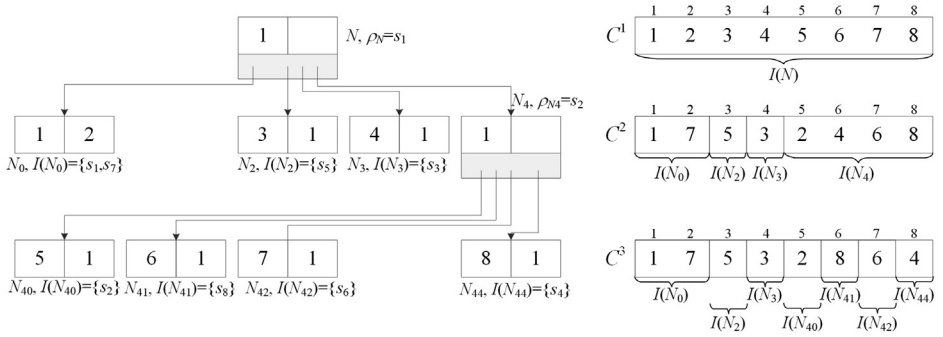

Fig. 3. The reference tree and its corresponding consecutive memory C.

Note that each internal node contains a reference string represented by its starting position of T and at most $\ell+1$ child nodes. In the physical implementation, constructing set S, also represented by the starting positions of all its elements in T, can be stored in a consecutive memory space C. When each internal node X classifies these positions (i.e., the strings in $I(X)$) into $\ell+1$ groups according to their distances from ρ_X, we simply move these positions in C to new locations accordingly such that each of the $\ell+1$ groups still occupies the consecutive space in C. Consequently, in leaf node Y, all positions (strings in $I(Y)$) definitely locate in C in a consecutive segment. We may use an integer to keep the starting location of the segment and another integer to memorize the length of the segment. As a result, the node in our reference tree only requires 2 integers and $\ell+1$ pointers (pointing to the $\ell+1$ subtrees). Following the aforementioned example (see Fig. 1), Fig. 3 illustrates our data structures where C^i shows the contents of the consecutive memory for the constructing data from level 1 to level i of the tree. Initially, C^1 records the starting positions of all ℓ-substrings of T.

The number of leaf nodes in the reference tree is smaller than n (even when $k=1$), and the number of internal nodes would be less than that of leaf nodes. Since each node (except the root node) is connected by a link from its parent node, the number of links is at most n for leaf nodes and is at most n for internal nodes. Thus, the space complexity for these links is $O(n)$. Each internal node records the starting position of the reference string and the necessary links to its children. Thus, the space complexity of all internal nodes is $O(n)$. Since all ℓ-substrings, represented by their starting positions in T, would be dispatched into the leaf nodes, the space complexity for the leaves is also $O(n)$. Consequently, the space complexity of the reference tree is $O(n)$ in the worst case.

In our reference tree, ℓ and k act as the parameters for adjusting the size and the shape of the tree so that the tree could meet various searching requirements for different data instances. The expected height of the reference tree, denoted as h and analyzed in Appendix A, is $1+\log_x \frac{k}{(n-\ell+1)}$ where $x=\max\binom{\ell}{\ell-d}\frac{(\sigma-1)^d}{\sigma^\ell}$. The expected number of comparisons in constructing tree is thus $O\left((n-\ell+1)\times\sigma^\ell\times\left(1-\left(1-\left(\frac{1}{\sigma}\right)^\ell\right)^h\right)\right)$ and the expected space complexity is $O\left(\left(\frac{k}{n-\ell+1}\right)^{\log_x \ell}\right)$. The search for a pattern P needs $O(h)$ steps on the tree, and verifying the positions in a *special* leaf takes $O((|P|-\ell)\times n) = O(|P|\times n)$ time in the worst case (i.e., all ℓ-substrings of T are in a leaf node) and that in an *ordinal* leaf needs $O(|P|\times k)$ time where the special leaf is the one whose input substrings are equal to the reference string in its parent and the ordinal leaf contains no greater than k substrings that are different from its parent's reference string.

3. Experimental results and discussions

The parameters in our reference tree approach include n (the length of text T), σ (the size of alphabet Σ), r (the number of patterns), ℓ (the length of the substrings of T) and k (the maximum number of the ℓ-substrings stored in the ordinal leaves of the tree). As we mentioned before, in computational biology the researchers often need to know where contigs of the target DNA sequence occur in its reference sequences. Or, we would like to find inspiring phrases in the Bible or classic novels/speeches. Therefore, our experiments focused on these two alphabets: DNA sequence and English language with $\sigma=4$ and $\sigma=63$, respectively. The number of patterns was roughly set to be $r=10^3$, which is a common amount in the DNA sequencing problem. Concerning different matching requirements (e.g., finding short/long contigs, searching simple/compound phrases, etc), the test patterns were classified into 3 categories according to their lengths:

(a) $7\pm20\%, 8\pm20\%, \ldots, 20\pm20\%$;
(b) $30\pm20\%, 40\pm20\%, \ldots, 90\pm20\%$; and
(c) $100\pm20\%, 200\pm20\%, \ldots, 1000\pm20\%$

where length $u\pm v$ means that the lengths are in the range $[u-\lfloor uv\rfloor, u+\lfloor uv\rfloor]$ (e.g., $[800, 1200]$ for $1000\pm20\%$). The test patterns were randomly selected from the text (that is, each pattern occurs in the text at least once) in each pattern group $u\pm v$. This setting not only eases the verification to the correctness of our approach, but also meets our interest in the applications of DNA sequencing where the occurrences of contigs (particularly, long patterns like those in Category (c))

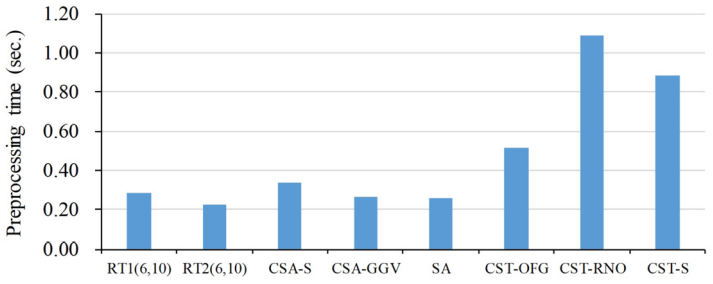

Fig. 4. Preprocessing times for RT1$(6, 10)$, RT2$(6, 10)$ and the suffix tree/array algorithms on Drosophila DNA sequence with length 10^6.

in the DNA sequence (text) are informative. For each group consisting of r $(= 10^3)$ patterns, 1000 independent pattern sets were prepared and tested. The computational results for each group reported in the following are the average outcomes of these 1000 independent runs. The experimental results concerning randomly generated texts and patterns (where each pattern may or may not occur in the text) are summarized in Appendix B.

The experiments were run on a personal computer with Intel Core2 Quad Q6600 with 2.4 GHz, 4 MB of cache and 4 GB of RAM. The operating system was CentOS 6.10 64-bit. Our programs[2] were coded in C and complied with gcc (GCC) version 4.8.5 with optimization option -O3. Either Algorithm 1 (for the preprocessing stage) or Algorithm 2 (for the searching stage) of our approach with parameters ℓ and k, is referred to as RT1(ℓ, k) for short.

In the physical implementation, to compute $\delta(b_i, \beta_X)$ (the Hamming distance between s_i and ρ_X in node X where $i \in I(X)$, see Equation (1)), b_i and β_X need to be extracted from the bit string B of T (see Fig. 2) where the starting positions of b_i (i.e., $(i-1)\lambda + 1$) and β_X in B should be calculated. Note that bit string b_i (or β_X) may be located in two or more consecutive computer words. The extractions of bit strings could be avoided by utilizing an extra table, in which the bit string of each ℓ-substring is pre-stored. That is, the ith element in the table stores bit string b_i of s_i $(= T[i, i + \ell - 1])$ for $1 \le i \le n - \ell + 1$. By accessing b_i from this table directly, we omit the calculation for locating and extracting b_i from B for every distance computation. The performance of our approach can thus be improved to some degree. We refer to this accelerated version as RT2(ℓ, k).

To verify the efficiency of our approach, we also implemented the suffix tree and suffix array algorithms including

(1) SA: the suffix array without compression,
(2) CSA-S: the compressed suffix array proposed by Sadakane [32],
(3) CSA-GGV: the compressed suffix array by Grossi et al. [13],
(4) CST-OFG: the compressed suffix tree by Ohlebusch et al. [25],
(5) CST-RNO: the compressed suffix tree by Russo et al. [31] and
(6) CST-S: the compressed suffix tree by Sadakane [33].

We called the corresponding functions in the SDSL [12] version 2.0,[3] which is a popular and quality tool, and has been applied by many computational works [2,6,8,11,23,26,29,34]. The library was complied with their default options (with optimization -O3).

We designated two experiments to summarize our abundant experimental results. The first concentrated on a real DNA sequence of $\sigma = 4$; whereas, the second addressed on the Bible text of $\sigma = 63$. The computational results, including the preprocessing and searching times, of our approaches as well as aforementioned suffix array and suffix tree algorithms are presented accordingly in the following.

3.1. Experiment 1: EMPM on DNA sequence

To obtain the real DNA sequence, we downloaded the sequence of *Drosophila yakuba* from the web-site of National Center for Biotechnology Information (NCBI). The GenBank assembly accession is GCA_000005975.1 and RefSeq assembly accession is GCF_000005975.2. After eliminating the symbols N's, the length of the whole sequence is about 168 M. We simply chose the prefix with length 1 M (i.e., $n = 10^6$) as our test text.

Based on our preliminary and extensive tests, we set $\ell = 6$ and $k = 10$ under $\sigma = 4$ to construct the reference tree for the DNA text. The preprocessing times for all the eight algorithms compared in this experiment are reported in Fig. 4. It is easily seen that the performances of RT1$(6, 10)$, RT2$(6, 10)$, CSA-S, CSA-GGV and SA are more efficient than those of the other three. Among the leading and comparable five, our RT2$(6, 10)$ delivers the best execution time; while RT1$(6, 10)$ is slightly slower than CSA-GGV and SA, yet faster than CSA-S.

[2] The codes of our reference tree approaches are available at https://github.com/AlgLab-NTHU/Reference-Tree-Approach.
[3] SDSL-lite library is available at https://github.com/simongog/sdsl-lite.

Table 1
Spaces usage for the eight algorithms on Drosophila DNA sequence with length 10^6.

	RT1(6, 10)	RT2(6, 10)	CSA-S	CSA-GGV	SA	CST-OFG	CST-RNO	CST-S
Space usage (MB)	9.34	334	5.18	5.18	7.98	25.52	25.52	39.91

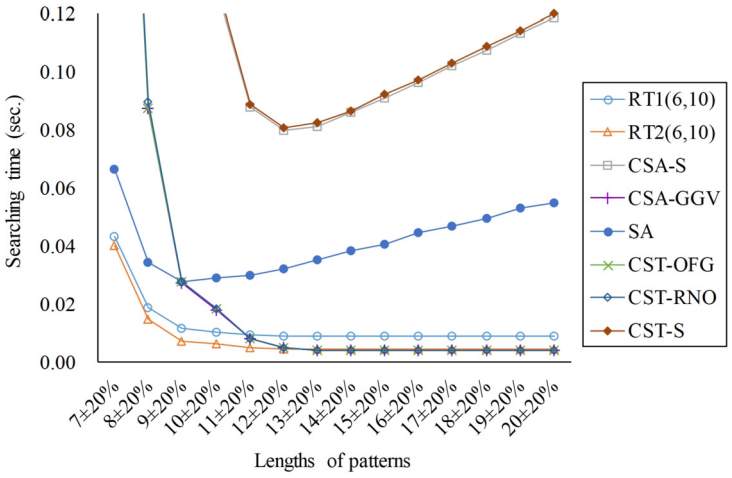

Fig. 5. Searching times for patterns in Category (a) on Drosophila DNA sequence with length 10^6.

Table 2
Average numbers of the occurrences of a pattern with regard to lengths in Category (a) on Drosophila DNA sequence with length 10^6.

Lengths of patterns	$7 \pm 20\%$	$8 \pm 20\%$	$9 \pm 20\%$	$10 \pm 20\%$	$11 \pm 20\%$	$12 \pm 20\%$	$13 \pm 20\%$	$14 \pm 20\%$	$15 \pm 20\%$
Average number of occurrences	146.16	41.15	12.49	8.11	3.20	1.79	1.35	1.21	1.18

Table 1 lists the memory spaces usage for these eight algorithms. It is seen that compressed suffix array algorithms (e.g., CSA-S and CSA-GGV) require the least memory space (e.g., 5.18 MB); SA takes the second least (7.98 MB) and RT1(6, 10) the third (9.34 MB), which are followed by compressed suffix tree algorithms (e.g., CST-OFG, CST-RNO and CST-S); whereas, RT2(6, 10) requires the greatest space. Note that because all texts for the three pattern-categories are the same DNA sequence, the preprocessing times and spaces of these three categories would be equal.

Let us first consider the patterns in Category (a), whose lengths range from $7 \pm 20\%$ to $20 \pm 20\%$. Fig. 5 shows the searching times of our approaches and the other six algorithms. Table 2 presents the average number of the occurrences of a pattern in the text for the pattern groups with lengths no larger than $15 \pm 20\%$. For the pattern-lengths ranging from $12 \pm 20\%$ to $20 \pm 20\%$, CSA-GGV, CST-OFG and CST-RNO outperform all the others. RT1(6, 10) and RT2(6, 10) are slightly slower, but faster than the other three algorithms, CSA-S, SA and CST-S.

For the very short pattern-lengths ranging from $7 \pm 20\%$ to $10 \pm 20\%$, our RT1(6, 10) and RT2(6, 10) have the best performances among all. In such cases, all the 6-substrings of the text have been constructed in the reference tree (with at most $k = 10$ substrings in the ordinal leaves). This structure leads a short searching time from the root to a certain leaf (with height at most 17, which is also the depth of our tree) and a rapid verification time since the rest part needed to be verified (in a special leaf) is very short. In particular, for lengths $7 \pm 20\%$ ([6, 8]), only at most 2 characters (the rest part) need to be verified for a pattern. There are in average 227 occurrences for a 6-substring stored in a certain special leaf, which incur at most 227×2 character comparisons. The average number of the occurrences of a pattern is about $occ = 146$ (see Table 2). Thus we need about $(227 \times 2)/146 = 3.11$ character comparisons (or $O(3.11 \times occ)$) in average to verify a pattern in a special leaf, which is very efficient. It is noticed that Table 2 also reveals that as the length of the patterns grows, the average number of the occurrences of a pattern in the text decreases.

In short, from Fig. 5, Tables 1 and 2, we realize that for pattern lengths ranging from $12 \pm 20\%$ to $20 \pm 20\%$ with less pattern-occurrences CSA-GGV, CST-OFG and CST-RNO offer the best performance using a fair memory space. For those raging from $7 \pm 20\%$ to $11 \pm 20\%$ with more pattern-occurrences RT2(6, 10) runs the fastest under the support of a relatively large space, while RT1(6, 10) follows to a lesser extent using also a fair space. The construction of all ℓ-substrings in the reference tree receives a quality performance here.

Fig. 6 presents the searching times of CSA-GGV, CST-OFG, CST-RNO, RT1(6, 10) and RT2(6, 10) for the patterns in Category (b). Note that those of CSA-S, SA and CST-S, which are relatively slower, are not included. RT2(6, 10) runs the fastest among all with almost a constant performance. The tradeoff for gaining acceleration at the expense of more space in RT2(6, 10) is now manifest. The execution times of CSA-GGV, CST-OFG and CST-RNO grow as the pattern-lengths increase linearly.

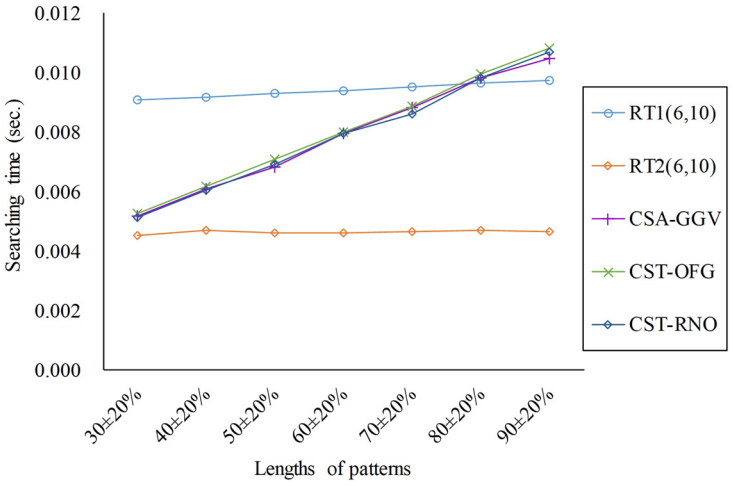

Fig. 6. Searching times of the best five algorithms for patterns in Category (b) on Drosophila DNA sequence with length 10^6.

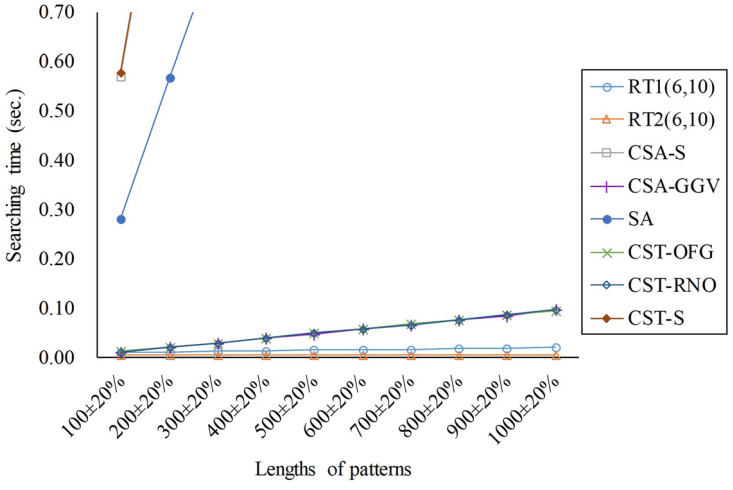

Fig. 7. Searching times for patterns in Category (c) on Drosophila DNA sequence with length 10^6.

They run faster than RT1$(6, 10)$ when pattern lengths are smaller than $80 \pm 20\%$, but slower when those are no less than $80 \pm 20\%$. It is noticed that our reference tree approaches behave quite stable (where the searching times vary insignificantly as the pattern-length grows) for coping with patterns in Category (b).

Regarding the patterns in the third Category (c), the searching times for all eight algorithms are illustrated in Fig. 7. As can be seen, RT1$(6, 10)$ and RT2$(6, 10)$ report better performances than CSA-GGV, CSA-OFG and CST-RNO (which spent nearly the same time), and behave much better than the other three algorithms. For the lengths ranging in $[80, 120]$, RT2$(6, 10)$ is faster than RT1$(6, 10)$ about 2.11 times and faster than other six algorithms at least 2.46 times. Regarding the longest pattern group with lengths ranging in $[800, 1200]$, RT2$(6, 10)$ is faster than RT1$(6, 10)$ about 3.51 times and faster than the others at least 16.33 times. The performances of our approaches are indeed appealing when dealing with the EMPM on the DNA sequence for the patterns in Category (c).

3.2. Experiment 2: EMPM on the Bible

We downloaded the King James Version of the Bible (Old Testament) from the website of String Matching Researching Tool (SMART).[4] The length of the strings in this version is about 4.04 M. The whole strings were included as our test text (i.e., $n = 4.04 \times 10^6$) with $\sigma = 63$.

Based on our pilot tests, we set $\ell = 9$ and $k = 100$ to construct the reference tree for the Bible text. The preprocessing times of the eight algorithms are presented in Fig. 8. Similar as the discussions about Fig. 4, RT1$(9, 100)$, RT2$(9, 100)$, CSA-S,

[4] SMART is available at https://smart-tool.github.io/smart/.

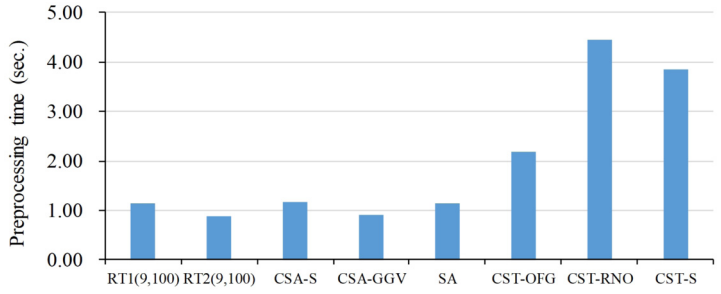

Fig. 8. Preprocessing times on the Bible text with length 4.04 M.

Table 3
Spaces usage on Bible with length 4.04 M and $\sigma = 63$.

	RT1(9, 100)	RT2(9, 100)	CSA-S	CSA-GGV	SA	CST-OFG	CST-RNO	CST-S
Spaces usage (MB)	46.39	622	19.71	19.71	24.44	78.87	78.87	157

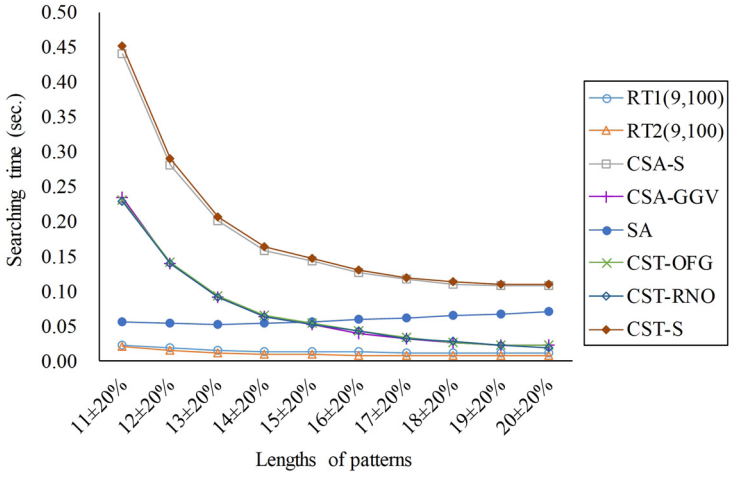

Fig. 9. Searching times for patterns in Category (a) on the Bible text with length 4.04 M.

Table 4
The average numbers of the occurrences of a pattern in the Bible text with length 4.04 M.

Lengths of patterns	$11 \pm 20\%$	$12 \pm 20\%$	$13 \pm 20\%$	$14 \pm 20\%$	$15 \pm 20\%$	$16 \pm 20\%$	$17 \pm 20\%$	$18 \pm 20\%$	$19 \pm 20\%$	$20 \pm 20\%$
Average number of occurrences	54.6	32.54	20.91	14.37	11.68	8.68	6.68	5.15	4.13	3.63

CSA-GGV and SA dominate the other three. Among all, RT2(9, 100) gives the best performance, while CSA-GGV lags behind slightly. RT2(9, 100) is about 1.02 times faster than CSA-GGV and at least 1.27 times faster than the all other suffix tree/array algorithms.

Table 3 summarizes the spaces usage for these algorithms. The space of RT1(9, 100) is larger than those of CSA-S and CSA-GGV about 2.35 times and that of SA about 1.90 times but smaller than those of CST-OFG and CST-RNO about 1.7 times and that of CST-S about 3.38 times. RT2(9, 100) demands a large memory space, which is about 13.41 times larger than that of RT1(9, 100), to accelerate the searching speed.

For the patterns in Category (a), Fig. 9 depicts the searching times for the eight algorithms. Note that the lengths here range in $11 \pm 20\%, 12 \pm 20\%, \ldots, 20 \pm 20\%$, which are not shorter than ℓ. Table 4 exhibits the average number of the occurrences of a pattern in the text for each pattern group. Our approaches are more efficient than the other six algorithms. For the shortest patterns ($11 \pm 20\%$), RT1(9, 100) and RT2(9, 100) are about 2.46 and 2.81 times, respectively, faster than SA, which reports the best performance among the other six algorithms. For the patterns with lengths $20 \pm 20\%$, RT1(9, 100) and RT2(9, 100) run about 1.68 and 2.78 times faster than CST-RNO, which is the fastest among the other six.

The searching times for the patterns in Category (b) are presented in Fig. 10. It is seen that our approaches are more efficient than the suffix tree/array algorithms and the superiority tends to grow as the pattern-length increases. For the patterns with lengths ranging in $30 \pm 20\%$, RT1(9, 100) and RT2(9, 100) operate about 1.08 and 1.92 times speedier than

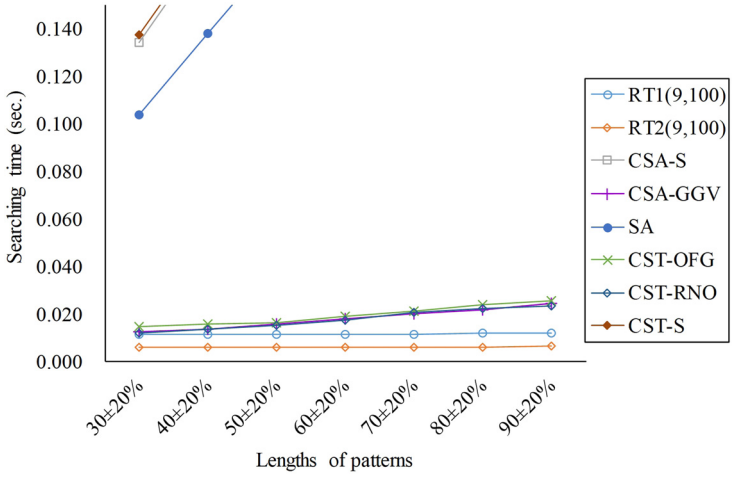

Fig. 10. Searching times for patterns in Category (b) on the Bible text with length 4.04 M.

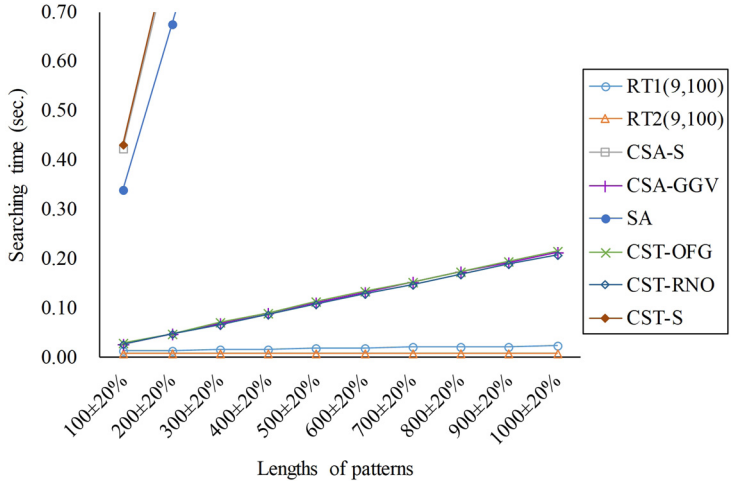

Fig. 11. Searching times for patterns in Category (c) on the Bible text with length 4.04 M.

CST-RNO which is the best among the six suffix tree/array algorithms. For those ranging in $90 \pm 20\%$, the corresponding ratios advance to 1.94 and 3.64. It is also seen that the running times of the suffix tree/array algorithms grow as the pattern-length increases; on the other hand, those of our approaches barely change. We may say that the behavior of our approaches is quite stable here.

Concerning the patterns in Category (c), Fig. 11 explores the searching times for the eight algorithms. It is easy to see from Fig. 11 that both RT1$(9, 100)$ and RT2$(9, 100)$ deliver the shortest running times, CSA-GGV, CST-OFG and CST-RNO, which run close to each other, come next, while SA, CSA-S and CST-S are a long way behind. The growing superiority of our approaches over the others as pattern-length increases, observed in Category (b), persists by more evidential supports. Specifically, for the patterns with lengths $100 \pm 20\%$, RT1$(9, 100)$ and RT2$(9, 100)$ execute about 2.09 and 3.99 times, respectively, faster than CST-RNO which dominates the others. With regard to the patterns with lengths $1000 \pm 20\%$, their corresponding ratios over CST-RNO become 8.96 and 28.53, respectively. In addition, RT2$(9, 100)$ runs about 3.18 times faster than RT1$(9, 100)$ and at least 29.12 times than the others.

Based on the computational outcomes from our experiments, we realize that our approaches are capable of dealing with the EMPM problem with satisfactory performances for those applications that patterns occur in the text. For DNA sequence and the patterns in Categories (a) with lengths less than $11 \pm 20\%$ and (c) all, RT1$(6, 10)$ and RT2$(6, 10)$ deliver the best performances among all tested algorithms. Moreover, for Bible and the patterns in Categories (a), (b) and (c), RT1$(9, 100)$ and RT2$(9, 100)$ outperform the others as well. In particular, it is very promising to see that the exact matches of 10^3 patterns with lengths in $[800, 1200]$ can be determined in 0.0204 and 0.0058 (0.0232 and 0.0073) seconds for the DNA sequence (the Bible text) by our RT1 and RT2, respectively. The searching efficiency of our reference tree approaches for EMPM is very encouraging.

Concerning the preprocessing stage, RT2(6, 10) performs the best using merely 0.23 seconds for the DNA sequence with $n = 10^6$; while RT2(9, 100) acts the second best taking 0.89 (with a tiny disparity from 0.90 spent by the leading CSA-GGV) seconds for the Bible text with $n = 4.04 \times 10^6$. Nevertheless, the preprocessing in RT1(6, 10) using about 0.28 (or RT1(9, 100) using 1.14) seconds is still efficient and competitive. Generally speaking, the preprocessing time is quite acceptable.

Although our experiments were designed to cope with the problems in computational biology and text searching, such ideas can be easily applied to problems in information security (like virus detection), or pattern matching (such as object recognition), and so on.

4. Conclusions

To deal with the exact multiple pattern matching problem, we design simple and elegant algorithms to construct the ℓ-substrings of text T as a reference tree and search the patterns in the tree. The reference strings in the internal nodes act as filters that percolate the ℓ-substrings of T into corresponding subtrees until reaching the leaves with sizes no more than k. With the guidance of these reference strings in the tree, the exact match of a pattern could be efficiently determined in $O(h)$ from the root to some leaf (for locating the candidates of matches), $O(n \times |P|)$ in a special leaf and $O(k \times |P|)$ in an ordinal leaf (for verifying exact matches) where h is the height of the tree. For those patterns that do not possess any candidate, the search in the tree terminates immediately. All Hamming distance computations are based on bitwise operations in order to enhance the performances of our tree construction and search algorithms.

The execution behavior of our approach (RT1) could be further accelerated by adopting more memory for storing bit strings separately (RT2). The construction of the reference tree and the subsequence searches of patterns are easy and efficient via bitwise operations. In our experiments conducting the DNA sequence and the Bible text, our reference tree approaches deliver pleasing computational results as compared to the popular methods using suffix tree and suffix array. Even though our ideas are simple, the experimental results demonstrate the efficiency and applicability of our reference tree approaches in coping with the EMPM problem. The expected height of the reference tree is analyzed in Appendix A, based on which we obtain the expected space requirement and the expected numbers of comparison.

The parameters involved in the experiments, i.e., $(n, r, \sigma, \ell, k) = (10^6, 10^3, 4, 6, 10)$ and $(4.04 \times 10^6, 10^3, 63, 9, 100)$ with different pattern lengths, are quality and robust settings among our tentative trails. More exploration and explanation on the computational outcomes with respect to various parameter settings form an interesting research topic worthy of further investigation. We are also interested in resolving the approximate multiple pattern matching problem by extending our reference tree approach. Our ideas may be applied to a broader range of applications with deliberate parameter settings as long as they could be modeled as the EMPM problem.

CRediT authorship contribution statement

All authors have participated in (a) conception and design, or analysis and interpretation of the data; (b) drafting the article or revising it critically for important intellectual content; and (c) approval of the final version.

Declaration of competing interest

The authors have no affiliation with any organization with a direct or indirect financial interest in the subject matter discussed in the manuscript.

Appendix A

Owing to $|\Sigma| = \sigma$, the probability for two characters, say z_1 and z_2, to be equal is $\frac{1}{\sigma}$ (where the distance between z_1 and z_2 is 0), while, $\frac{(\sigma-1)}{\sigma}$ for them to be different (where the distance between z_1 and z_2 is 1). Let ℓ-string denote the string with length ℓ. Consider two ℓ-strings. The probability for them to be with distance d is:

$$\binom{\ell}{\ell-d}\left(\frac{1}{\sigma}\right)^{\ell-d}\binom{d}{d}\left(\frac{\sigma-1}{\sigma}\right)^{d} = \binom{\ell}{\ell-d}\left(\frac{1}{\sigma}\right)^{\ell-d}\left(\frac{\sigma-1}{\sigma}\right)^{d} \tag{A.1}$$

where $0 \le d \le \ell$. Concerning m ℓ-strings and one ℓ-string α, the expected number of these m ℓ-strings that have distance d with α becomes

$$\binom{\ell}{\ell-d}\left(\frac{1}{\sigma}\right)^{\ell-d}\left(\frac{\sigma-1}{\sigma}\right)^{d} \times m.$$

In root node N, the expected number of the strings whose distances with the reference string ρ_N are d should be

$$\binom{\ell}{\ell-d}\left(\frac{1}{\sigma}\right)^{\ell-d}\left(\frac{\sigma-1}{\sigma}\right)^{d}(n-\ell+1) \tag{A.2}$$

where $0 \le d \le \ell$.

Probabilistically, the leaf having the greatest depth is in the subtree rooted at the node which has been dispatched the most input strings. We shall track such nodes level by level to explore the leaf having the greatest depth, and subsequently to obtain the height of the reference tree. Let N_t^* be the node containing the most strings in level t where $t > 1$. By applying Equation (A.2), the number of the strings in N_t^* is

$$\max\binom{\ell}{\ell-d}\left(\frac{1}{\sigma}\right)^{\ell-d}\left(\frac{\sigma-1}{\sigma}\right)^{d}|I\left(N_{t-1}^*\right)| = \max\binom{\ell}{\ell-d}\frac{(\sigma-1)^d}{\sigma^\ell} \times |I\left(N_{t-1}^*\right)|$$

where $|I\left(N_{t-1}^*\right)|$ is the number of the strings in node N_{t-1}^*. In level 2, the number of the strings in N_2^* is

$$\max\binom{\ell}{\ell-d}\frac{(\sigma-1)^d}{\sigma^\ell} \times (n-\ell+1)\,.$$

In general, the expected number of the strings for node $N_{h'}^*$ in level h' becomes

$$\left(\max\binom{\ell}{\ell-d}\frac{(\sigma-1)^d}{\sigma^\ell}\right)^{h'-1}(n-\ell+1)\,.$$

When the number of the strings in a node is smaller than or equal to k, this node becomes a leaf. Therefore, the expected height h of the reference tree can be computed as follows.

$$\begin{aligned}
&\left(\max\binom{\ell}{\ell-d}\frac{(\sigma-1)^d}{\sigma^\ell}\right)^{h-1}(n-\ell+1) \le k\\
&\Longrightarrow \left(\max\binom{\ell}{\ell-d}\frac{(\sigma-1)^d}{\sigma^\ell}\right)^{h-1} \le \frac{k}{n-\ell+1}\\
&\Longrightarrow h-1 \ge \log_{\left(\max\binom{\ell}{\ell-d}\frac{(\sigma-1)^d}{\sigma^\ell}\right)}\frac{k}{n-\ell+1}\\
&\Longrightarrow h \ge 1+\log_{\left(\max\binom{\ell}{\ell-d}\frac{(\sigma-1)^d}{\sigma^\ell}\right)}\frac{k}{n-\ell+1}
\end{aligned} \tag{A.3}$$

Now, we would like to analyze the space needed in our approach (including leaves and internal nodes). Recall that there are two types of leaves in our tree: (a) the special leaves, containing the strings that are equal to the reference strings of their parents, and (b) the ordinal leaves, containing no greater than k strings that are not equal to the corresponding reference strings. Consider the reference tree with height h. All nodes at level h are leaves and the maximal number of the internal nodes in level $h-1$ is ℓ^{h-2}. In total, the maximum number of the internal nodes would be

$$\sum_{i=0}^{h-2}\ell^i = \frac{\ell^{h-1}-1}{\ell-1}. \tag{A.4}$$

Surely, (A.4) gives an upper bound for the number of the internal nodes. Owing to the fact that the number of the internal nodes is equal to that of the special leaves in our reference tree, we obtain that the maximum number of the special leaves is also $\frac{\ell^{h-1}-1}{\ell-1}$. Because the maximum number of the internal nodes in level $h-1$ is ℓ^{h-2}, the maximum number of the ordinal leaves in level h would be ℓ^{h-1}. Then, the space complexity of the reference tree with parameters ℓ and k is:

$$\begin{aligned}
&O\left(\left(\frac{\ell^{h-1}-1}{\ell-1}\right)+\left(\ell^{h-1}+\frac{\ell^{h-1}-1}{\ell-1}\right)\right)\\
&= O\left(\ell^{h-1}\right),
\end{aligned} \tag{A.5}$$

where $h = 1+\log_{\left(\max\binom{\ell}{\ell-d}\frac{(\sigma-1)^d}{\sigma^\ell}\right)}\frac{k}{n-\ell+1}$. Substituting h into (A.5), we have

$$O\left(\ell^{\left(1+\log_{\left(\max\binom{\ell}{\ell-d}\frac{(\sigma-1)^d}{\sigma^\ell}\right)}\frac{k}{n-\ell+1}\right)-1}\right)$$

$$= O\left(\ell^{\log\left(\max\left(\binom{\ell}{\ell-d}\frac{(\sigma-1)^d}{\sigma^\ell}\right)\frac{k}{n-\ell+1}\right)}\right). \tag{A.6}$$

Because $a^{\log_b c} = a^{\log_a c/\log_a b} = c^{1/\log_a b} = c^{\log_b a}$, the expected space complexity of the reference tree is:

$$O\left(\left(\frac{k}{n-\ell+1}\right)^{\log\left(\max\left(\binom{\ell}{\ell-d}\frac{(\sigma-1)^d}{\sigma^\ell}\right)\right)^\ell}\right) \tag{A.7}$$

where $0 \le d \le \ell$ and σ is the size of alphabet.

We know that for each internal node X, the number of comparisons (i.e., *popcount*'s via bitwise operations to compute the distances between all strings of the constructing data and ρ_X) is $O(|I(X)|)$. Naturally, the number of input strings in the root node N is $n-\ell+1$. Next, let us consider the input strings of the internal nodes in level 2. According to Equation (A.2), the expected number of input strings of all internal nodes (i.e., the expected number of comparisons) in level 2 with parameters ℓ and k is:

$$\begin{aligned}
&\binom{\ell}{\ell-1}\left(\frac{1}{\sigma}\right)^{\ell-1}\left(\frac{\sigma-1}{\sigma}\right)^1 \times (n-\ell+1) + \\
&\binom{\ell}{\ell-2}\left(\frac{1}{\sigma}\right)^{\ell-2}\left(\frac{\sigma-1}{\sigma}\right)^2 \times (n-\ell+1) + \cdots + \\
&\binom{\ell}{\ell-\ell}\left(\frac{1}{\sigma}\right)^{\ell-\ell}\left(\frac{\sigma-1}{\sigma}\right)^\ell \times (n-\ell+1) \\
&= \left(\binom{\ell}{\ell-1}\left(\frac{1}{\sigma}\right)^{\ell-1}\left(\frac{\sigma-1}{\sigma}\right) + \binom{\ell}{\ell-2}\left(\frac{1}{\sigma}\right)^{\ell-2}\left(\frac{\sigma-1}{\sigma}\right)^2 + \cdots + \right. \\
&\quad \left.\binom{\ell}{\ell-\ell}\left(\frac{1}{\sigma}\right)^{\ell-\ell}\left(\frac{\sigma-1}{\sigma}\right)^\ell\right) \times (n-\ell+1) \\
&= \left(1-\left(\frac{1}{\sigma}\right)^\ell\right) \times (n-\ell+1).
\end{aligned} \tag{A.8}$$

Note that in Equation (A.8), the expected number of the strings equal to ρ_N in N is $\left(\frac{1}{\sigma}\right)^\ell (n-\ell+1)$ and that different from ρ_N is $\left(1-\left(\frac{1}{\sigma}\right)^\ell\right)(n-\ell+1)$.

Likewise, the expected number of the strings in internal nodes of level 3 would be

$$\left(1-\left(\frac{1}{\sigma}\right)^\ell\right)^2 \times (n-\ell+1).$$

With the same reasoning, with respect to h (the height of the reference tree) the expected number of comparisons for constructing the tree is

$$\begin{aligned}
&O\left(\sum_{i=0}^{h-1}\left(1-\left(\frac{1}{\sigma}\right)^\ell\right)^i \times (n-\ell+1)\right) \\
&= O\left((n-\ell+1)\sum_{i=0}^{h-1}\left(1-\left(\frac{1}{\sigma}\right)^\ell\right)^i\right) \\
&= O\left((n-\ell+1) \times \left(1+\left(1-\left(\frac{1}{\sigma}\right)^\ell\right)^1 + \cdots + \left(1-\left(\frac{1}{\sigma}\right)^\ell\right)^{h-1}\right)\right) \\
&= O\left((n-\ell+1) \times \frac{1-\left(1-\left(\frac{1}{\sigma}\right)^\ell\right)^h}{1-\left(1-\left(\frac{1}{\sigma}\right)^\ell\right)}\right) \\
&= O\left((n-\ell+1) \times \sigma^\ell \times \left(1-\left(1-\left(\frac{1}{\sigma}\right)^\ell\right)^h\right)\right).
\end{aligned} \tag{A.9}$$

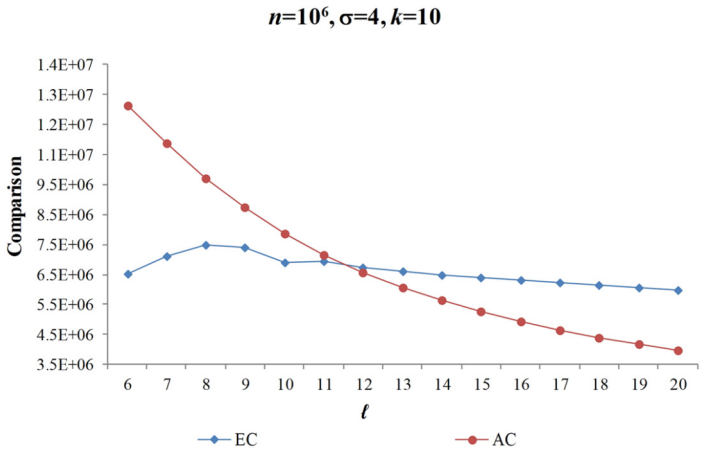

Fig. A.1. Results of AC and EC (numbers of comparisons) with respect to ℓ for DNA sequence.

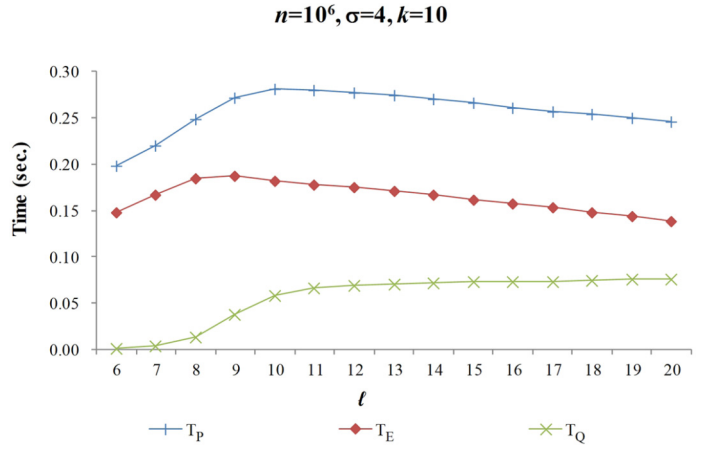

Fig. A.2. Results of T_Q, T_E and T_P with respect to ℓ for DNA sequence.

To realize the applicability of (A.9) for constructing the reference tree, we performed some experiments to compare the analytic results with the realistic computational outcomes. The text is the prefix with length $n = 10^6$ of *Drosophila yakuba* (which was also adopted in Experiment 1 of Section 3). We tested the different settings of (ℓ, k)'s where $6 \le \ell \le 20$ and $k = 10$. Let the number of comparisons based on (A.9) under the expected height h (A.3) be referred to as AC, and that from our physical computational outcomes be as EC. Fig. A.1 illustrates the values of AC and EC with respect to various (ℓ, k)'s. It is seen that both AC and EC have the similar decreasing tendency as ℓ grows for $\ell \ge 8$. We say that the numbers of comparisons are with the same order for a larger ℓ. These computational outcomes, to some degree, demonstrate that our analysis is quite applicable.

It is also observed from Fig. A.1 that a larger ℓ tends to produce small amount of comparisons. Still, when constructing the tree, we need a queue (in Algorithm 1) to store the nodes waiting to be expanded. Let T_Q denote the time for maintaining the queue, including all operations for enqueue and dequeue. Let T_E be the time for all comparisons (i.e., *popcount*'s). Let T_P be the time for the whole preprocessing stage. Fig. A.2 presents the time information of T_Q, T_E and T_P as ℓ grows in the above experiment. It is seen that $T_Q + T_E$ approximates T_P (which involves other operations such as converting characters into bit-strings, grouping strings in the internal nodes, and so on). Furthermore, when ℓ goes large, the change of T_E is slight; whereas, the increment of T_Q is relatively crucial. Consequently, setting $\ell = 6$ in T_P, which consists of a small T_E and the smallest T_Q, delivers the best execution time among the other settings. This is the reason that we chose $\ell = 6$ (instead of a larger one) in our experiments.

Appendix B

B.1. EMPM on text and patterns randomly generated from $\sigma = 4$

The text was randomly generated with length 1 M and $\sigma = 4$ and the patterns were likewise generated with Categories (a), (b) and (c). In general, the occurrences of short random patterns in the text would be more than those of long ones.

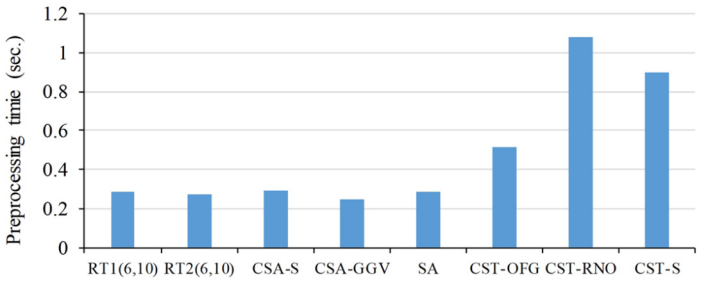

Fig. B.1. Preprocessing times for the eight algorithms on randomly generated text with length 10^6 and $\sigma = 4$.

Table B.1
Spaces usage for the eight algorithms on random text with length 10^6 and $\sigma = 4$.

	RT1(6, 10)	RT2(6, 10)	CSA-S	CSA-GGV	SA	CST-OFG	CST-RNO	CST-S
Spaces usage (MB)	9.25	334	5.09	5.08	7.89	25.43	25.43	39.82

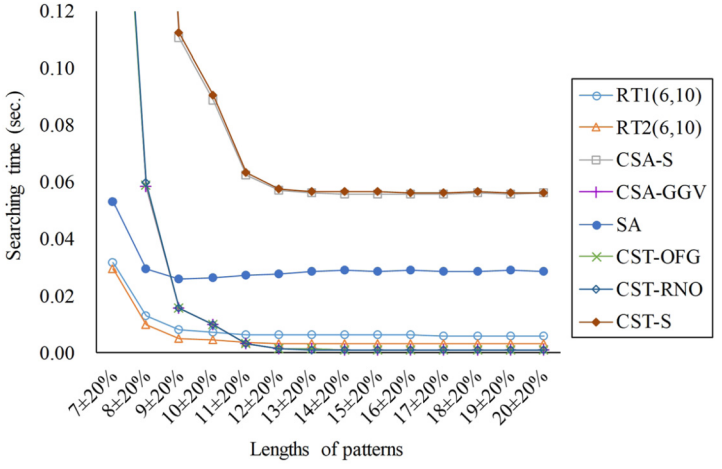

Fig. B.2. Searching times of the eight algorithms for random patterns in Category (a) on random text with length 10^6 and $\sigma = 4$.

When random patterns are lengthy enough, there may be hardly any occurrence. It can thus be expected that random patterns in Category (a) would appear many times in the text, while those in Category (c) might hardly appear.

The preprocessing times for the eight algorithms are displayed in Fig. B.1. As similar to the results in Experiment 1, RT1(6, 10), RT2(6, 10), CSA-S, CSA-GGV and SA are more efficient than other three. Among the leading five algorithms, the efficiency of CSA-GGV is the best, and those for the other four are similar.

The spaces usage for these eight algorithms is listed in Table B.1. Compressed suffix array algorithms (e.g., CSA-S and CSA-GGV) utilize the least memory spaces (e.g., 5.09 and 5.08 MB, respectively). The spaces of SA and RT1(6, 10) are about 1.55 and 1.82 times more than CSA-GGV. CST-OFG (and CST-RNO) needs about 5.01 times more than CSA-GGV. The CST-S and RT2(6, 10) utilize the greatest spaces, which are about 7.84 and 65.75 times larger than CSA-GGV, respectively.

The searching times for the patterns in Category (a), are presented in Fig. B.2. For the pattern-lengths ranging from $7 \pm 20\%$ to $10 \pm 20\%$, the performances of RT1(6, 10) and RT2(6, 10) are the best among all. When patterns are longer than $11 \pm 20\%$, CSA-GGV, CST-OFG and CST-RNO turn to be the fastest. This is similar to the findings in Experiment 1. As shown in Table B.2, for the cases of the lengths ranging in $[7 \pm 20\%, 10 \pm 20\%]$, such short random patterns incur quite many occurrences (106.8 (4.07) in average for one pattern with length $7 \pm 20\%$ ($10 \pm 20\%$)) in the text, which consumes suffix tree/array algorithms more searching times. For the patterns with lengths $7 \pm 20\%$, RT1(6, 10) outperforms about 1.67 times than SA (with a comparable memory space), which reported the best among the suffix tree/array algorithms. On the other hand, when the lengths grow up to $12 \pm 20\%$ or more, it is hard to see that a random pattern appears in the text (see Table B.2). Such a "few occurrences" situation is benefit to CSA-GGV, CST-OFG and CST-RNO so that they obtain the best performances.

The corresponding results for patterns in Categories (b) and (c) are exhibited in Figs. B.3 and B.4, respectively. Observing Figs. B.3 and B.4, we find that the growth of the curves of the eight algorithms seems to be steady. Indeed, when the lengths of patterns are greater than $20 \pm 20\%$, there is no occurrence of patterns in the text and all the curves remain almost the same.

Table B.2
Average numbers of the occurrences of a pattern with lengths from $7 \pm 20\%$ to $15 \pm 20\%$ in the random text with length 10^6 and $\sigma = 4$.

Lengths of patterns	$7 \pm 20\%$	$8 \pm 20\%$	$9 \pm 20\%$	$10 \pm 20\%$	$11 \pm 20\%$	$12 \pm 20\%$	$13 \pm 20\%$	$14 \pm 20\%$	$15 \pm 20\%$
Average number of occurrences	106.79	26.70	6.68	4.07	1.02	0.25	0.06	0.02	0.01

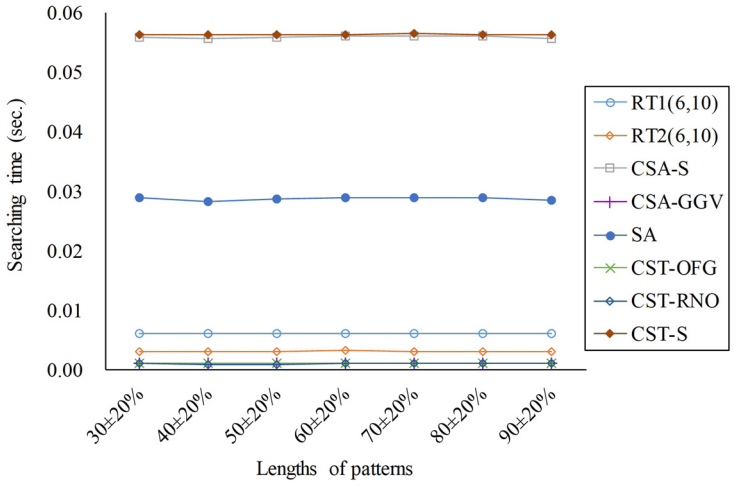

Fig. B.3. Searching times for random patterns in Category (b) on random text with length 10^6 and $\sigma = 4$.

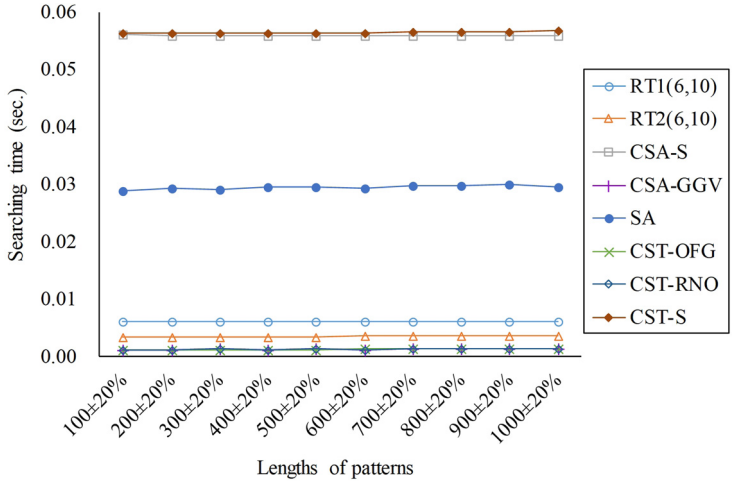

Fig. B.4. Searching times for random patterns in Category (c) on random text with length 10^6 and $\sigma = 4$.

Table B.3
Spaces usage for the eight algorithms on random text with length 4.04 M and $\sigma = 63$.

	RT1(9, 100)	RT2(9, 100)	CSA-S	CSA-GGV	SA	CST-OFG	CST-RNO	CST-S
Spaces usage (MB)	51.57	629	19.63	19.63	24.36	82.79	82.79	160

B.2. EMPM on English text and patterns randomly generated from $\sigma = 63$

The text with length 4.04 M (which is equal to that of the Bible) and patterns were randomly generated from $\sigma = 63$. As the setting in Experiment 2, we set $\ell = 9$ and $k = 100$ to construct the reference tree. The preprocessing times of the eight algorithms are shown in Fig. B.5. Suffix array algorithms (e.g., CSA-S, CSA-GGV and SA) outperform the other five algorithms. RT2(9, 100), CST-OFG and RT1(9, 100) follow in sequence and CST-RNO and CST-S are the slowest. The best CSA-GGV is about 2.83 and 1.83 times faster than RT1(9, 100) and RT2(9, 100), respectively.

The memory spaces usage for these algorithms is illustrated in Table B.3. The results are similar to those of the Bible text in Table 3.

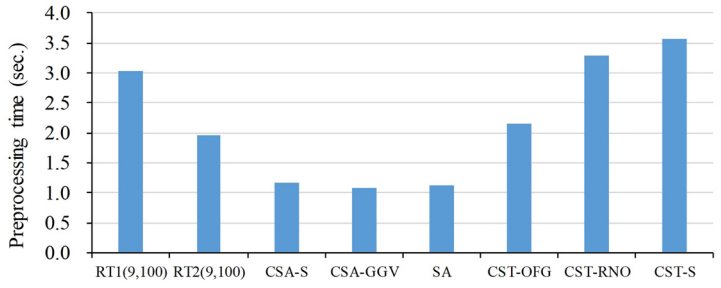

Fig. B.5. Preprocessing times on random text with length 4.04 M and $\sigma = 63$.

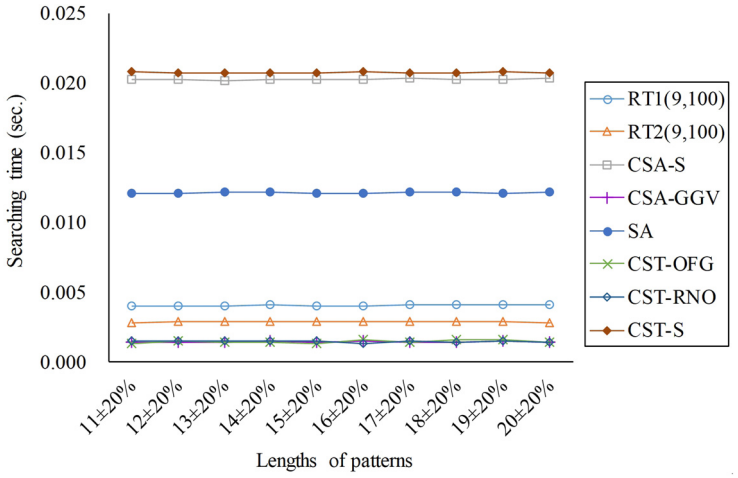

Fig. B.6. Searching times for random patterns in Category (a) on random text with length 4.04 M and $\sigma = 63$.

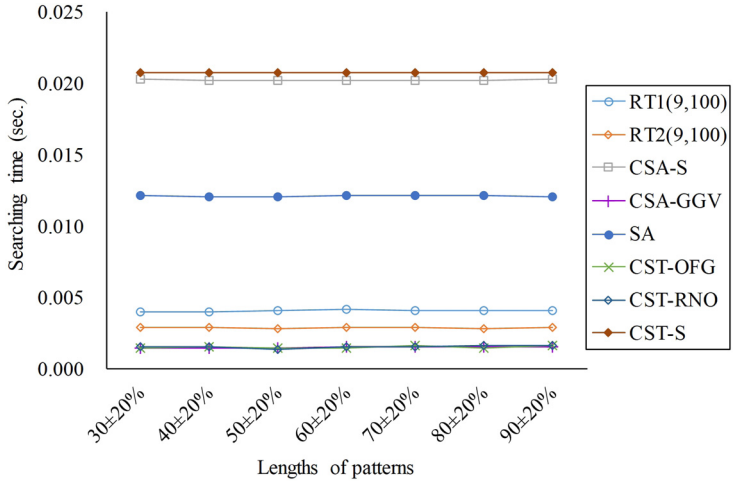

Fig. B.7. Searching times for random patterns in Category (b) on random text with length 4.04 M and $\sigma = 63$.

The probability for finding an occurrence of one random pattern with length $[9, 13]$ for $11 \pm 20\%$ is at most $(1/63)^9$, which is rather small. Actually, there is no occurrence in this experiment. Such a "no occurrence" situation results in the similar effect as the "few occurrences" situation in Appendix B.1. The searching times of the eight algorithms are reported in Figs. B.6, B.7 and B.8 for the patterns in Categories (a), (b) and (c), respectively. All the curves of these algorithms are almost horizontal. The searching time of the fastest algorithm CST-RNO runs about 2.77 and 1.95 times faster than RT1$(6, 10)$ and RT2$(6, 10)$, respectively.

Based on the outcomes of the above results, we realize that CSA-GGV, CST-OFG and CST-RNO are superior than our approaches. We note that the occurrences of the patterns in these experiments are few.

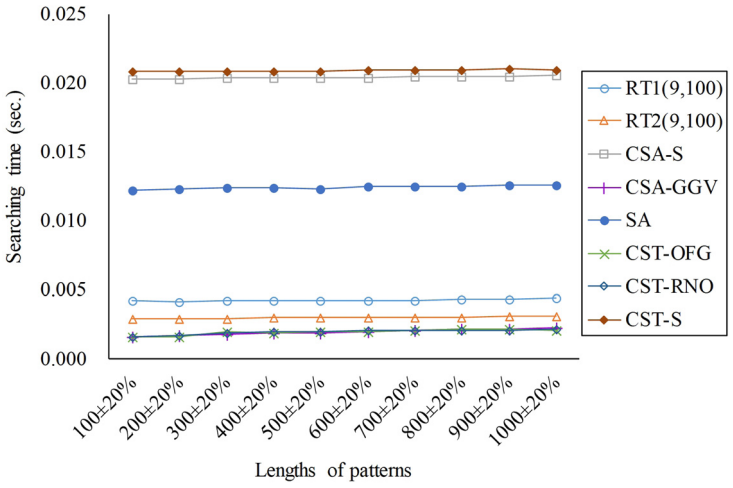

Fig. B.8. Searching times for random patterns in Category (c) on random text with length 4.04 M and $\sigma = 63$.

References

[1] Mohamed I. Abouelhoda, Stefan Kurtz, Enno Ohlebusch, Replacing suffix trees with enhanced suffix arrays, J. Discret. Algorithms 2 (1) (2004) 53–86, https://doi.org/10.1016/S1570-8667(03)00065-0.

[2] Rachit Agarwal, Anurag Khandelwal, Ion Stoica, Succinct: enabling queries on compressed data, in: 12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4–6, 2015, 2015, pp. 337–350, https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/agarwal.

[3] Samuel A. Assefa, Thomas M. Keane, Thomas D. Otto, Chris Newbold, Matthew Berriman, ABACAS: algorithm-based automatic contiguation of assembled sequences, Bioinformatics 25 (15) (2009) 1968–1969, https://doi.org/10.1093/bioinformatics/btp347.

[4] Anton Bankevich, Sergey Nurk, Dmitry Antipov, Alexey A. Gurevich, Mikhail Dvorkin, Alexander S. Kulikov, Valery M. Lesin, Sergey I. Nikolenko, Son K. Pham, Andrey D. Prjibelski, Alex Pyshkin, Alexander Sirotkin, Nikolay Vyahhi, Glenn Tesler, Max A. Alekseyev, Pavel A. Pevzner, Spades: a new genome assembly algorithm and its applications to single-cell sequencing, J. Comput. Biol. 19 (5) (2012) 455–477, https://doi.org/10.1089/cmb.2012.0021.

[5] P.S.G. Chain, D.V. Grafham, R.S. Fulton, M.G. FitzGerald, J. Hostetler, D. Muzny, J. Ali, B. Birren, D.C. Bruce, C. Buhay, J.R. Cole, Y. Ding, S. Dugan, D. Field, G.M. Garrity, R. Gibbs, T. Graves, C.S. Han, S.H. Harrison, S. Highlander, P. Hugenholtz, H.M. Khouri, C.D. Kodira, E. Kolker, N.C. Kyrpides, D. Lang, A. Lapidus, S.A. Malfatti, V. Markowitz, T. Metha, K.E. Nelson, J. Parkhill, S. Pitluck, X. Qin, T.D. Read, J. Schmutz, S. Sozhamannan, P. Sterk, R.L. Strausberg, G. Sutton, N.R. Thomson, J.M. Tiedje, G. Weinstock, A. Wollam, J.C. Detter, Genome project standards in a new era of sequencing, Science (ISSN 0036-8075) 326 (5950) (2009) 236–237, https://doi.org/10.1126/science.1180614.

[6] Francisco Claude, Roberto Konow, Gonzalo Navarro, Efficient indexing and representation of web access logs, in: String Processing and Information Retrieval - Proceedings of the 21st International Symposium, SPIRE 2014, Ouro Preto, Brazil, October 20–22, 2014, 2014, pp. 65–76.

[7] Maxime Crochemore, Christophe Hancart, Thierry Lecroq, Algorithms on Strings, Cambridge University Press, ISBN 978-0-521-84899-2, 2007.

[8] Felipe Alves da Louza, Simon Gog, Leandro Zanotto, Guido Araujo, Guilherme P. Telles, Parallel computation for the all-pairs suffix-prefix problem, in: String Processing and Information Retrieval - Proceedings of the 23rd International Symposium, SPIRE 2016, Beppu, Japan, October 18–20, 2016, 2016, pp. 122–132.

[9] Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, Gonzalo Navarro, Compressed representations of sequences and full-text indexes, ACM Trans. Algorithms 3 (2) (2007) 20, https://doi.org/10.1145/1240233.1240243.

[10] Marco Galardini, Emanuele G. Biondi, Marco Bazzicalupo, Alessio Mengoni, Contiguator: a bacterial genomes finishing tool for structural insights on draft genomes, Source Code Biol. Med. 6 (2011) 11, https://doi.org/10.1186/1751-0473-6-11.

[11] Simon Gog, Enno Ohlebusch, Fast and lightweight lcp-array construction algorithms, in: Proceedings of the Thirteenth Workshop on Algorithm Engineering and Experiments, ALENEX 2011, Holiday Inn San Francisco Golden Gateway, San Francisco, California, USA, January 22, 2011, 2011, pp. 25–34.

[12] Simon Gog, Timo Beller, Alistair Moffat, Matthias Petri, From theory to practice: plug and play with succinct data structures, in: Experimental Algorithms - Proceedings of the 13th International Symposium, SEA 2014, Copenhagen, Denmark, June 29 - July 1, 2014, 2014, pp. 326–337.

[13] Roberto Grossi, Ankur Gupta, Jeffrey Scott Vitter, High-order entropy-compressed text indexes, in: Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12–14, 2003, Baltimore, Maryland, USA, 2003, pp. 841–850, http://dl.acm.org/citation.cfm?id=644108.644250.

[14] Richard W. Hamming, Error detecting and error correcting codes, Bell Syst. Tech. J. (ISSN 0005-8580) 29 (2) (April 1950) 147–160, https://doi.org/10.1002/j.1538-7305.1950.tb00463.x.

[15] Henry S. Warren Jr., Hacker's Delight, second edition, Pearson Education, ISBN 0-321-84268-5, 2013, http://www.hackersdelight.org/.

[16] Ming-Yang Kao (Ed.), Encyclopedia of Algorithms - 2016 Edition, Springer, ISBN 978-1-4939-2863-7, 2016.

[17] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, Kunsoo Park, Linear-time longest-common-prefix computation in suffix arrays and its applications, in: Combinatorial Pattern Matching, Proceedings of the 12th Annual Symposium, CPM 2001, Jerusalem, Israel, July 1–4, 2001, 2001, pp. 181–192.

[18] Zhize Li, Jian Li, Hongwei Huo, Optimal in-place suffix sorting, in: 2018 Data Compression Conference, DCC 2018, Snowbird, UT, USA, March 27–30, 2018, 2018, p. 422.

[19] Linda Liu, Yinhu Li, Shiliang Li, Ni Hu, Yimin He, Ray Pong, Danni Lin, Lihua Lu, Maggie Law, Comparison of next-generation sequencing systems, BioMed Res. Int. 2012 (2012) 251364, https://doi.org/10.1155/2012/251364.

[20] Ruibang Luo, Binghang Liu, Yinlong Xie, Zhenyu Li, Weihua Huang, Jianying Yuan, Guangzhu He, Yanxiang Chen, Qi Pan, Yunjie Liu, Jingbo Tang, Gengxiong Wu, Hao Zhang, Yujian Shi, Yong Liu, Chang Yu, Bo Wang, Yao Lu, Changlei Han, David W. Cheung, Siu-Ming Yiu, Shaoliang Peng, Zhu Xiaoqian, Guangming Liu, Xiangke Liao, Yingrui Li, Huanming Yang, Jian Wang, Tak-Wah Lam, Jun Wang, SOAPdenovo2: an empirically improved memory-efficient short-read de novo assembler, GigaScience (ISSN 2047-217X) 1 (1) (2012), https://doi.org/10.1186/2047-217X-1-18.

[21] Udi Manber, Eugene 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.

[22] Diego C. Mariano, Felipe L. Pereira, Preetam Ghosh, Debmalya Barh, Henrique C. Figueiredo, Artur Silva, Rommel T. Ramos, Vasco A. Azevedo, Maprepeat: an approach for effective assembly of repetitive regions in prokaryotic genomes, Bioinformation 11 (6) (2015) 276–279, https://doi.org/10.6026/97320630011276.

[23] Martin D. Muggli, Simon J. Puglisi, Christina Boucher, Efficient indexed alignment of contigs to optical maps, in: Algorithms in Bioinformatics - Proceedings of the 14th International Workshop, WABI 2014, Wroclaw, Poland, September 8–10, 2014, 2014, pp. 68–81.

[24] Gonzalo Navarro, Mathieu Raffinot, Flexible Pattern Matching in Strings - Practical On-line Search Algorithms for Texts and Biological Sequences, Cambridge University Press, ISBN 978-0-521-81307-5, 2002, http://www.dcc.uchile.cl/~Egnavarro/FPMbook/.

[25] Enno Ohlebusch, Johannes Fischer, Simon Gog, CST++, in: String Processing and Information Retrieval - Proceedings of the 17th International Symposium, SPIRE 2010, Los Cabos, Mexico, October 11–13, 2010, 2010, pp. 322–333.

[26] Enno Ohlebusch, Stefan Stauß, Uwe Baier, Trickier XBWT tricks, in: String Processing and Information Retrieval - Proceedings of the 25th International Symposium, SPIRE 2018, Lima, Peru, October 9–11, 2018, 2018, pp. 325–333.

[27] Daniel Paulino, René L. Warren, Benjamin P. Vandervalk, Anthony Raymond, Shaun D. Jackman, Inanç Birol, Sealer: a scalable gap-closing application for finishing draft genomes, BMC Bioinform. 16 (2015) 230, https://doi.org/10.1186/s12859-015-0663-4.

[28] Vitor Piro, Helisson Faoro, Vinícius A. Weiss, Maria Steffens, Fabio Pedrosa, Emanuel Souza, Roberto Raittz, Fgap: an automated gap closing tool, BMC Res. Notes 7 (2014) 371, https://doi.org/10.1186/1756-0500-7-371.

[29] Andreas Poyias, Rajeev Raman, Improved practical compact dynamic tries, in: String Processing and Information Retrieval - Proceedings of the 22nd International Symposium, SPIRE 2015, London, UK, September 1–4, 2015, 2015, pp. 324–336.

[30] Anna I. Rissman, Bob Mau, Bryan S. Biehl, Aaron E. Darling, Jeremy D. Glasner, Nicole T. Perna, Reordering contigs of draft genomes using the mauve aligner, Bioinformatics 25 (16) (2009) 2071–2073, https://doi.org/10.1093/bioinformatics/btp356.

[31] Luís M.S. Russo, Gonzalo Navarro, Arlindo L. Oliveira, Fully compressed suffix trees, ACM Trans. Algorithms 7 (4) (2011) 53, https://doi.org/10.1145/2000807.2000821.

[32] Kunihiko Sadakane, New text indexing functionalities of the compressed suffix arrays, J. Algorithms 48 (2) (2003) 294–313, https://doi.org/10.1016/S0196-6774(03)00087-7.

[33] Kunihiko Sadakane, Compressed suffix trees with full functionality, Theory Comput. Syst. 41 (4) (2007) 589–607, https://doi.org/10.1007/s00224-006-1198-x.

[34] José Fuentes Sepúlveda, Erick Elejalde, Leo Ferres, Diego Seco, Efficient wavelet tree construction and querying for multicore architectures, in: Experimental Algorithms - Proceedings of the 13th International Symposium, SEA 2014, Copenhagen, Denmark, June 29 - July 1, 2014, 2014, pp. 150–161.

[35] Esko Ukkonen, On-line construction of suffix trees, Algorithmica 14 (3) (1995) 249–260, https://doi.org/10.1007/BF01206331.

[36] Peter Weiner, Linear pattern matching algorithms, in: 14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15–17, 1973, 1973, pp. 1–11.

[37] Daniel R. Zerbino, Ewan Birney, Velvet: algorithms for de novo short read assembly using de Bruijn graphs, Genome Res. 18 (5) (2008) 821–829, https://doi.org/10.1101/gr.074492.107.