SN Computer Science (2023) 4:260
https://doi.org/10.1007/s42979-023-01690-8

ORIGINAL RESEARCH

On the Multiple Pattern String Matching in DNA Databases

Yangjun Chen[1] · Bobin Chen[2] · Yujia Wu[1]

Received: 3 September 2021 / Accepted: 12 January 2023

Abstract

In this paper, we discuss an indexing method for solving the multiple string pattern matching problem, by which we are given a set of short pattern strings $\boldsymbol{R} = \{r_1,..., r_l\}$ and required to locate all those substrings of a long target string s such that each of them matches an r_j in $\boldsymbol{R}$. The main idea is to construct a pattern matching machine $\boldsymbol{A}$ over $\boldsymbol{R}$ and transform the reverse $\bar{s}$ of s to a Burrow-Wheeler-Transformation array as an index, denoted as $L = BWT(\bar{s})$, and search $\boldsymbol{A}$ against it. During the process, the *failure function* of $\boldsymbol{A}$ is utilized to decrease the number of subranges in L to be searched at each step. In addition, the *Wavelet* tree is used to reduce the searching cost of L, by which its single-character checking is changed to a multi-character checking. In this way, multiple searches of a Wavelet tree are reduced to a single search, and high efficiency can be achieved. Extensive experiments have been conducted, which shows that our method works better than almost all the existing methods for this problem.

Keywords String matching · DNA sequences · Multiple pattern machine · Automation · BWT-transformation

Introduction

By the multiple string pattern matching problem, we will be given a set $\boldsymbol{R} = \{r_1,..., r_l\}$, where each r_i $(1 \leq i \leq l)$ is a (short) string (or say, a finite sequence of symbols) called a pattern, and a (long) target string s. We are required to locate and identify all substrings of s which are patterns in $\boldsymbol{R}$. This problem becomes very important as the next-generation sequencing technique [7] comes into use, which needs to align a huge number of reads (short DNA sequences) against a very long sequence, known as a genome, which is previously well studied and often billions of characters long, for earlier diagnosis of cancers, or some other purposes. Normally, the number of reads is multiple millions and the length of a read is about 100 characters (bps).

Other applications of this problem also include network intrusion detection [25], digital forensics (file carving) [26, 64], business analytics, and natural language processing, just to name a few.

This problem was studied as early as the mid-1970′s. In [1], Aho and Corasick proposed the first efficient algorithm, by which a pattern matching machine (*PMM* for short) or an automaton $\boldsymbol{A}$ is constructed over $\boldsymbol{R}$ and then searched against s by successively reading the characters in s, making state transition and occasionally reporting output. The running time of this process is bounded by $O(\sum_{i=1}^{l}|r_i| + |s|)$.

This algorithm has been extensively used in practise, such as bioinformatics [18, 19], multiple key-word searching [31], and two-dimensional pattern searching [4]; and also improved or modified by different researchers, as reported in [16, 17, 55, 63]. However, the worst case time complexity remains unchanged.

On the other hand, different indexes have been developed for the single string pattern searching in the past several decades, such as suffix trees [47, 62], suffix arrays [44], hashing [30], and BWT-arrays [9, 35, 53, 58]. However, no effort has been directed to building indexes over s

The article is a modification and extension of a conference paper: Y. Chen and Y. Wu, Searching BWT against Pattern Matching Machine to Find Multiple String Matches, in: Proc. Cyber2017, pp. 167–176, Oct. 2017, Nanjing, China.

✉ Yangjun Chen
y.chen@uwinnipeg.ca

Bobin Chen
chen.bobin@outlook.com

Yujia Wu
wyj1128@yahoo.com

1 Department of Applied Computer Science, University of Winnipeg, Winnipeg, Canada

2 Department of Computer Science, University of Toronto, Toronto, Canada

Published online: 11 March 2023

to expedite the multiple string pattern matching described above.

In this paper, we address this issue. We will show that a kind of indexes over s, the so-called BWT-array (Burrow-Wheeler-Transformation array), can be established quickly, which we can use to speed up scanning of s when we search $\boldsymbol{A}$ in some way to bring down the searching time to O(|$\boldsymbol{A}$|). (This time complexity does not include the time for loading an index into main memory from hard disk. However, in practice, this part of time can be completely ignored. For example, for reading the BWT-array of a genome of 1,464,443,456 bytes, only 3 milliseconds are used.)

Specifically, the following techniques will be utilized to achieve high efficiency:

1. *Folding target string*. The positions with the same character in s will be clustered together by the BWT transformation. Then, we are able to search s in 'parallel′. That means, at each step, we will access a collection of positions in s with a same character, instead of a single one. In this sense, s is folded in some way, and becomes shorter. Searching such a "folded" and shorter string, we can save much time for doing the task.
2. *Subrange search reduction*. During a search of $\boldsymbol{A}$ against the BWT-array L for $\bar{s}$ (the reverse of s), a series of subranges within L will be checked. By using the failure function of $\boldsymbol{A}$, the number of such subranges can be greatly reduced.
3. *Speeding up search of BWT*($\bar{s}$). In our method, $L = BWT(\bar{s})$ is stored as a Wavelet tree T_L [22] and the search of a certain value in L corresponds to a search along a path of length $|\Sigma|$ in T_L. By changing a single-value searching to a multi-value searching, the total cost of searching L can be minimized.

In this way, our method can improve the running time of both the on-line algorithm (like the Aho-Corasick's algorithm), and the index-based algorithm (like the suffix tree) by 5–10 times.

The remainder of the paper is organized as follows. First, in "Notations", we summarize all the symbols and notations used throughout the paper. Then, in "Related Work", we review the related work. In "Basic Techniques: PMM and BWT", we briefly describe the PMM and the BWT transformation, based on which our method is established. "Algorithm Description" is devoted to the discussion of our algorithm for finding all occurrences of a set of pattern strings in a target string. In "Integrating Wavelet Tree Searching into PMM Searching", we discuss how to store an L as a Wavelet tree to reduce the searching costs. "Experiments" reports the test results. Finally, we conclude with a short summary and a brief discussion on the future work in "Conclusion".

Notations

In this section, we summarize all the symbols and notations used throughout the paper in the following Table 1.

Related Work

By a single pattern string matching problem, we will find all the occurrences of a pattern string r in a target string s. By a multi-pattern string matching problem, we are asked to identify all the occurrences of patterns each coming from a set $\boldsymbol{R}$ in a target string s. A huge number of algorithms have been proposed to solve these two kinds of problems. But in the following, we only review some of the most noteworthy strategies.

– *Single pattern string matching*

The first interesting algorithm for this problem is the famous *Knuth-Morris-Pratt′*s algorithm [33], which scans both r and s from left to right and uses an auxiliary next-table (for r) containing the so-called *shift* information (or say, *failure function* values) to indicate how far to shift the pattern from right to left when the current character in r fails to

Table 1 Symbols and notations

$\boldsymbol{R}$	$\boldsymbol{R} = \{r_1 ... r_l\}$, a set of patterns
T	$T = trie(\boldsymbol{R})$ constructed over $\boldsymbol{R}$
s	A target string
$trie(\boldsymbol{R})$	A trie built over $\boldsymbol{R}$
$BWT(s)$	BWT-array of s
T_L	A wavelet tree over $L = BWT(s)$
$<e, [\alpha, \beta]>$	A range (segment) from rank α to β in F_e
π	$\pi = <e, [\alpha, \beta]>$
L_π	A range in $L = BWT(s)$, corresponding to $\pi = <e, [\alpha, \beta]>$ in F_e
L_π^1	Start position in L_π
L_π^2	End position in L_π
$rk_F(e)$	Rank of $e \in$ array F
$rk_L(e)$	Rank of $e \in$ array L
$l(v)$	A character labeling a node $v \in T$
$P(v)$	A path from *root* to v in T
$f(v)$	A failure function associated with $v \in T$
$I(v)$	An interval associated with $v \in T$
$\boldsymbol{A}$	A pattern matching machine $\boldsymbol{A} = T \cup \{f(v) \mid v \in T\}$
Σ	Alphabet
Σ_v	Set of characters associated with v in T_L
Σ_l^v	First half in Σ_v
Σ_r^v	Second half in Σ_v

match the current character in s. Its time complexity is bounded by O($m + n$), where $m = |r|$ and $n = |s|$. (By the shift information, we mean a largest integer j associated with a position i in r such that $r[1..\ j] = r[i - j + 1..\ i]$. Thus, if the current character from the target does not match $r[i + 1]$, we will compare $r[j + 1]$ with the character next to the current one at a next step.) The *Boyer-Moore*′s approach [8, 20, 21] works a little bit better than the *Knuth-Morris-Pratt* ′s. In addition to the next-table, a skip-table *skip* of size $|\Sigma|$ (also for r) is kept, in which each entry *skip*[w] is a smallest integer j such that $r[m - j] = w$. Here, Σ is the alphabet, from which we take characters for both s and r_j's. For a large alphabet and small pattern, the expected number of character comparisons is about n/m, and is O($m + n$) in the worst case. By the hash-table-based algorithms [30], short substrings called }*seed*′ will be first extracted from a pattern r and a *signature* (a bit string) for each of them will be created. The search of a target string s is similar to that of the Brute Force searching, but rather than directly comparing the pattern at successive positions in s, their respective signatures are compared. Then, stick each matching seed together to form a complete alignment. Its expected time is O($m + n$), but in the worst case, which is extremely unlikely, it takes O($m \cdot n$) time. The hash technique has also been extensively used in the DNA sequence research [23, 39, 40, 59]. However, almost all experiments show that they are generally inferior to the suffix tree and the *BWT* index in both running time and space requirements. The bit-parallelism introduced by Baeza-Yates and Gonnet [5] takes advantage of the intrinsic parallelism of the bit operations inside a computer word, allowing to cut down the number of operations that an algorithm performs by a factor up to w, where w is the number of bits in the computer word.

In situations where a fixed string s is to be searched repeatedly, it is worthwhile constructing an index over s, such as suffix trees [47, 62], suffix arrays [44], and more recently the Burrows-Wheeler transformation (or say, *BWT* transformation) [9, 11, 41, 41]. A suffix tree is in fact a trie structure [32] over all the suffixes of s; and by using the Weiner′s algorithm [62] it can be built in O(n) time. However, in comparison with the BWT transformation, a suffix tree needs much more space. Especially, for DNA sequences, the BWT transformation works highly efficiently due to the small alphabet Σ of DNA strings. By the BWT transformation, the smaller Σ is, the less space will be occupied by the corresponding indexes. According to a survey done by Li and Homer [38] on sequence alignment algorithms for next-generation sequencing, the average space required for each character is 12–17 bytes for suffix trees while only 0.5–2 bytes for the BWT transformation. Our experiments also confirm this distinction [11, 14]. For example, the file size of chromosome 1 of human is 270 Mb. But its suffix tree is of 26 Gb in size while its BWT transformation needs only 390 Mb–1 Gb for different compression rates of auxiliary arrays, completely handleable on PC or laptop machines.

– *Multi-pattern string matching*

The first efficient algorithm for multi-pattern string matching was proposed by Aho and Corasick in 1975 [1], by which s is searched for occurrences of all patterns from a set: $\{r_1, r_2, ..., r_l\}$. Their algorithm needs only O($\sum_{i=1}^{l}|r_i| + |s|$) time. Later on, some variants of this algorithm have been suggested. In [16], Commentz-Walter combines the Boyer-Moore′s technique into the Aho-Corasick′s algorithm. In [63], Wu and Manber extend the Boyer-Moore′s algorithm to concurrently search multiple pattern strings. Instead of using bad-character heuristics to compute shift values, they utilize a character block containing 2 or 3 characters. In addition, hash tables are created to link the blocks and the related patterns. In [55], a concept of superalphabets is introduced, in which each (super) character corresponds to a set of q-grams (each being a substring from a certain pattern and represented as a bit string, called a signature, generated by using a hash function.) In this way, a super automaton can be created, in which each transition is labeled with a super character. s will also be handled as a sequence of q-grams and searched in the same way as the *Aho-Corasick* ′s algorithm. The main problem of this method is the false positive and entails a very time-consuming verification process. In [12], Crochemore et al. combine the directed acyclic word graphs into the*Aho-Corasick* ′s algorithm. If the total length of all patterns is polynomial with respect to the shortest length m of a pattern, the average number of comparisons is O((n/m) log m).

However, all the improved algorithms have the same worst-case time complexity as the Aho-Corasick′s.

In addition, several researchers have attempted to improve the performance of multi-pattern string matching applications via the use of parallelism, such as those discussed in [27, 28, 46, 50, 56, 57, 60, 64]. They either port the *Aho-Corasick* ′ algorithm to different parallel machines, such as the IBM Cell Broadband Engine (CBE) [56, 57, 64], or GPUs [28, 60]; or simply execute any efficient on-line string matching algorithm, such as the *Knuth-Morris-Pratt* ′s [33], the *Boyer-Moore* ′s [8], or the *Wu-Manber* ′s [63], over distributed patterns. But all of them are only suitable for the cases when the number of patterns is limited, not scaling well to massive sets of patterns. The hardware-based method also suffers from the same problem [2]. However, by our method, even for many millions of patterns, the high performance can be achieved. In addition, our algorithm itself can also be executed in parallel, by porting the BWT-arrays to CBEs.

In Table 2, we compare our method with all the representative on-line, as well as index-based strategies.

Finally, we point out that there is a bunch of work on the inexact string matching, such as [12, 13, 36] for the string matching with k differences, and [10, 15, 37, 52] for the string matching with k differences, as well as [45] for the string matching with *wild-card* symbols. Due to the mismatches, differences, and wild-cards, the match relation is no longer transitive and therefore the techniques established to solve these problems cannot be employed for the multiple pattern string matching.

An interested reader is referred to [3, 49] for a brief, but complete survey on the string matching problem.

Basic Techniques: PMM and BWT

In this section, we briefly describe two basic techniques utilized in our method. They are the pattern matching machine and the BWT transformation.

Pattern Matching Machine

Similar to the *Aho-Corasick*'s algorithm, we need first to construct a pattern matching machine (*PMM*) $\boldsymbol{A}$ over $\boldsymbol{R} = \{r_1,..., r_l\}$. Different from it, however, we will not search $\boldsymbol{A}$ against s, but against $BWT(\bar{s})$.

Intuitively, the pattern matching machine $\boldsymbol{A}$ over $\boldsymbol{R}$ is considered as a directed graph composed of two parts: a *trie* T, denoted as $trie(\boldsymbol{R})$, and a *failure* function $f(v)$ ($v \in T$).

First of all, for each r_j ($j = 1,..., l$), we will attach a special character \$, which does not appear in any r_j, to its end and construct $trie(\boldsymbol{R})$ as described below.

If $|\boldsymbol{R}| = 0$, $trie(\boldsymbol{R})$ is, of course, empty. For $|\boldsymbol{R}| = 1$, $trie(\boldsymbol{R})$ is a single node. If $|\boldsymbol{R}| > 1$, $\boldsymbol{R}$ is split into $|\Sigma| = k$ (possibly empty) subsets $\boldsymbol{R}_1, \boldsymbol{R}_2,..., \boldsymbol{R}_k$ so that each $\boldsymbol{R}_i$ ($i \in \{1,..., k\}$) contains all those strings with the same first character. The tries: $trie(\boldsymbol{R}_1)$, $trie(\boldsymbol{R}_2,)$,..., $trie(\boldsymbol{R}_k)$ are constructed in the same way except that at the ith step, the splitting of sets is based on the ith characters in the sequences. They are then connected from their respective roots to a single node to create $trie(\boldsymbol{R})$.

Example 1 As an example, consider a set of four pattern strings:

r_1: acaga
r_2: ag
r_3: acagc
r_4: ca

For these pattern strings, a trie can be constructed as shown in Fig. 1a.

In this trie, v_0 is a *virtual* root, representing an empty string while any other node v stands for a string equal to the concatenation of all characters labelling the nodes on the path from v_0 to v, denoted as $P(v)$. Especially, if v is a leaf, $P(v)$ must be a string in $\boldsymbol{R}$. For instance, the path from v_0 to v_8 spells out the third pattern r_3 = acagc\$ in Fig. 1. In addition, however, we may associate some nodes v with an *output*

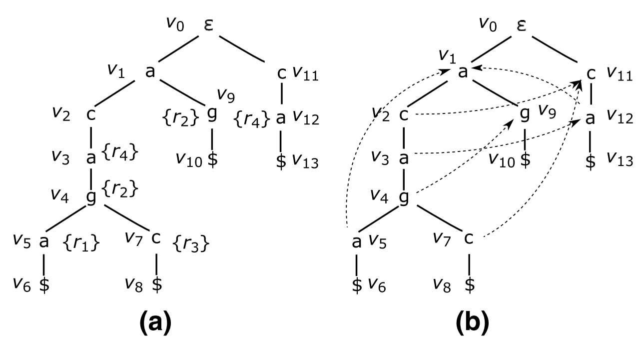

Fig. 1 A trie and a pattern matching machine

Table 2 Comparison of strategies

	Indexing time	Preprocessing time	Matching time
Suffix trees [62]	O(n)	0	O($\sum_{i=1}^{l}|r_i|$)
Suffix arrays [44]	O(n)	0	O($\sum_{i=1}^{l}|r_i|$)
BWT transformation [9]	O(n)	0	O($\sum_{i=1}^{l}|r_i|$)
Hash-based [30]	0	0	$\sum_{i=1}^{i}|r_i| + n$)
Bit-parallel [5]	0	O($\sum_{i=1}^{l}|r_i|$)	O(n)
Aho-Corasick's [1]	0	O($\sum_{i=1}^{l}|r_i|$)	O(n)
Commentz-Water's [16]	0	O($\sum_{i=1}^{l}|r_i|$)	O(n)
Chrochemore's [17]	0	O($\sum_{i=1}^{l}|r_i|$)	O((n/m)log m)*
Wu-Manber's [63]	0	O($\sum_{i=1}^{l}|r_i|$)	O(n)
Ours	O(n)	O($\sum_{i=1}^{l}|r_i|$)	O($|\boldsymbol{A}|$)

*m—the shortest length of a pattern

such that each $r \in output(v)$ is a string in $\boldsymbol{R}$ and also a suffix of $P(v)$. For example, for v_3 shown in Fig. 1a, we have $output(v_3) = \{r_4\}$. It is because $r_4 = \mathsf{ca} \in \boldsymbol{R}$ is a suffix of $P(v_3) = \mathsf{aca}$.

Besides, for a node v, we will use $l(v)$ to represent its character.

In terms of Aho and Corasick [1], $f(v) = u$ if and only if there exists a maximum suffix of the string spelt out along $P(v)$, which is equal to the string spelt out along $P(u)$.

Thus, by $f(v)$, similar to *Knuth-Morris-Pratt*′s, we give the node to be entered at a mismatch of $P(v)$, as illustrated by the dashed arrows in Fig. 1b. For example, $f(v_4) = v_9$ is represented by the dashed arrow from v_4 to v_9. We have this connection since ag, which is represented by $P(v_9)$, is a maximum suffix of $P(v_4) = \mathsf{acag}$ within $\boldsymbol{R}$.

Formally, we have

$$\boldsymbol{A} = T \cup \{f(v) | v \in T \setminus \{v_0\}\}. \tag{1}$$

We will also simply use $f(v)$ to represent a link from v to $f(v)$.

BWT and String Searching

Now, we describe the BWT transformation in some detail. We will use s to denote a string that we would like to transform. Again, assume that s terminates with \$, which does not appear elsewhere in s and is alphabetically prior to all other characters. In the case of DNA sequences, we have $\$ < \mathsf{a} < \mathsf{c} < \mathsf{g} < \mathsf{t}$. As an example, consider $s = \mathsf{ccagaca\$}$.

First, we will rotate s consecutively to create eight different strings, stacked vertically to form a matrix as illustrated in Fig. 2a.

Next, we sort the rows of the matrix alphabetically, and get another matrix, as demonstrated in Fig. 2b, which is called the *Burrow-Wheeler* Matrix [9] and denoted as $BWM(s)$. Especially, the last column L of $BWM(s)$, read from top to bottom, is called the BWT transformation (or the BWT-array) and denoted as $BWT(s)$. So for $s = \mathsf{ccagaca\$}$, we have $BWT(s) = \mathsf{acgcac\$a}$ (see Fig. 2c). The first column in $BWM(s)$ is referred to as F.

(a)	(b)	F	L
c c a g a c a \$	\$ c_1 c_2 a_1 g_1 a_2 c_3 a_3	\$	a_3
c a g a c a \$ c	a_3 \$ c_1 c_2 a_1 g_1 a_2 c_3	a_3	c_3
a g a c a \$ c c	a_2 c_3 a_3 \$ c_1 c_2 a_1 g_1	a_2	g_1
g a c a \$ c c a	a_1 g_1 a_2 c_3 a_3 \$ c_1 c_2	a_1	c_2
a c a \$ c c a g	c_3 a_3 \$ c_1 c_2 a_1 g_1 a_2	c_3	a_2
c a \$ c c a g a	c_2 a_1 g_1 a_2 c_3 a_3 \$ c_1	c_2	c_1
a \$ c c a g a c	c_1 c_2 a_1 g_1 a_2 c_3 a_3\$	c_1	\$
\$ c c a g a c a	g_1 a_2 c_3 a_3 \$ c_1 c_2 a_1	g_1	a_1

(a) **(b)** **(c)**

Fig. 2 Illustration for construction of BWT arrays

Special attention should be paid to Fig. 2b, c. In both of them, for ease of explanation, the position of a character in s is represented by its subscript. (That is, we rewrite s as $c_1c_2a_1g_1a_2c_3a_3\$$.) For example, a_2 represents the second appearance of a in s; and c_1 the first appearance of c in s. In the same way, we can check all the other appearances of different characters.

Additionally, when ranking an elements e in both F and L in such a way that if e is the ith appearance of a certain character it will be assigned i, the same element will get the same number in the two columns. For instance, in F the rank of a_3, denoted as $rk_F(a_3)$, is 1 (showing that a_3 is the first appearance of a in F). Its rank in L, $rk_L(a_3)$ is also 1. We can check all the other elements and find that this property, called the rank correspondence, holds for all the elements. That is, for any element e in s, we always have

$$rk_F(e) = rk_L(e) \tag{2}$$

According to this property, a string searching can be very efficiently conducted, as described below.

Firstly, notice that we can store F as $|\Sigma| + 1$ intervals, such as $F_\$ = F[1... 1]$, $F_a = F[2... 4]$, $F_c = F[5... 7]$, $F_g = F[8... 8]$, and $F_t = \phi$ for the above example (see Fig. 2c). We can also represent a segment within an F_e (with $e \in \Sigma$) as a pair of the form $<e, [\alpha, \beta]>$, where $\alpha \leq \beta$ are two ranks of e. Thus, we have $F_a = F[2... 4] = <\mathsf{a}, [1, 3]>$, $F_c = F[5... 7] = <\mathsf{c}, [1, 3]>$, and $F_g = F[8... 8] = <\mathsf{g}, [1, 1]>$.

Secondly, we will use L_π to represent a range in L corresponding to a pair $\pi = <e, [\alpha, \beta]>$. For example, in Fig. 2c, $L_{<\mathsf{a},[1,3]>} = L[2... 4]$, $L_{<\mathsf{c},[1,2]>} = L[5... 6]$. $L_{<\mathsf{a},[2,3]>} = L[3... 4]$, and so on.

In addition, for $L_\pi = L[a... b]$, we use L_π^1 to refer to a, and L_π^2 to b.

Finally, we implement a procedure $search(z, \pi)$ to search L_π to find the first and the last rank of z (denoted as α' and β', respectively) within L_π, and return $<z, [\alpha', \beta']>$ as the result:

$$search(z, \pi) = \begin{cases} < z, [\alpha', \beta'] >, & if\ z \in L_\pi; \\ \phi, & otherwise. \end{cases} \tag{3}$$

To locate r in s, we work on the characters in r in the reverse order (referred to as a backward search). That is, we will search $\bar{r}$ (reverse of r) against $BWT(s)$. It is because figuring out L_π in terms of π corresponds to a step of scanning in s, but in the reverse direction.

To see how it works, we consider $r = \mathsf{aca}$ and trace how r is identified in $s = \mathsf{ccagaca\$}$ step by step by using $BWT(s)$.

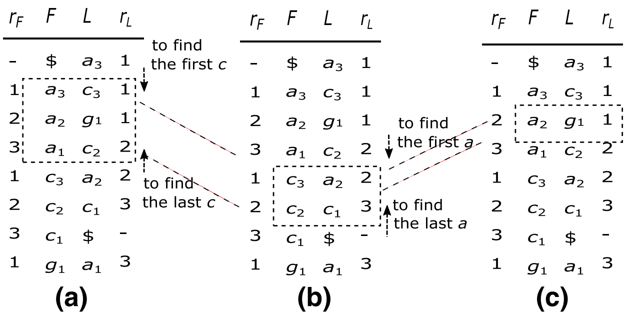

Fig. 3 Illustration of backward search

– Step. 1 (*checking the last character in r*): Check $r[3]$ = a in r, and then figure out $F_a = F[2... 4]$ = <a, [1, 3]>. (See Fig. 3a for illustration.)
– Step 2 (*checking the second character from last*): Check $r[2]$ = c. Call *search*(c , <a, [1, 3]>). By its execution, $L_{<a,[1,3]>} = L[2... 4]$ will be searched to find a range bounded by the first and last rank of c. Concretely, we will find $rk_L(c_3) = 1$ and $rk_L(c_2) = 2$. So, *search*(c, <a, [1, 3]>) returns <c, [1, 2]>. It is $F[5... 6]$. (See Fig. 3b.)
– Step 3 (*check the first character*): Check $r[1]$ = a. Call *search*(a, <c, [1, 2]>). Since $L_{<c,[1,2]>} = L[5... 6]$, *search*(a, <c, [1, 2]>) will return <*a*, [2, 2]>. It is $F[2... 2]$. At this step, we have exhausted all the characters in r and $F[2... 2]$ contains only one element, showing that one occurrence of r in s is found. It is represented by a_2 in s. See Fig. 3c.

Assume that the segment (in F) found at the last step contains i entries. Then, there are i occurrences of r in s with each indicated by an entry in that segment.

The above working process can be represented as a sequence of three pairs:

<a, [1, 3]>, <c, [1, 2]>, <a, [2, 2]>.

In general, for $\bar{r} = z_1... z_m$, its search against $BWT(s)$ can always be represented as a sequence of pairs (with each representing a segment in F_z for some $z \in \Sigma$):

$<z_1, [\alpha_1, \beta_1]>,..., <z_m, [\alpha_m, \beta_m]>$,

where $<z_1, [\alpha_1, \beta_1]> = F_{z_1}$, and $<z_i, [\alpha_i, \beta_i]> = search(z_i, z_{i-1}, [\alpha_{i-1}. \beta_{i-1}])$ for $1 < i \leq m$. We call such a sequence as a *search sequence*. Thus, the time used for this process is bounded by $O(\sum_{i=1}^{m} \tau_i)$, where τ_i is the time for an execution of $search(z_i, z_{i-1}, [\alpha_{i-1}, \beta_{i-1}])$. However, this time complexity can be reduced to $O(m)$ by using the so-called *rankAll* method, but with high space requirements [9]. Concretely, $O(n|\Sigma|\log n)$-bits space is required to store all the *rankAll* arrays. Another way to reduce time for searching L is to store L as a Wavelet tree [22], by which the usage of space can be decremented to $O(n\log|\Sigma|)$-bits, but with the searching time increased to $O(\log|\Sigma|)$. We modify the Wavelet tree method and integrate it into our strategy to reach an average searching time less than $O(\log|\Sigma|)$, but keep the space requirement $O(n\log|\Sigma|)$-bits not increased. This time complexity even beats the quantum string matching algorithm [43].

From the above discussion, we can observe a very important property of the BWT transformation, by which we check, at each step, a subset of characters (represented by a subsegment of F) from a target string s while by any on-line strategy only one character from s is checked at one step. In this sense, s is somehow folded by the BWT transformation.

Finally, we point out that $BWT(s)$ (or $BWT(\bar{s})$) can be constructed in $O(|s|)$ time via its relationship to the suffix array of s, as described below.

Let $s = x_0x_1...x_{n-1}$, ended with \$ (i.e., $x_i \in \Sigma$ for $i = 0,..., n-2$, and $x_{n-1} = \$$). Let $s[i] = x_i$ ($i = 0, 1,..., n-1$) be the ith character of s, $s[i...j] = x_i... x_j$ a substring and $s[i...n-1]$ a suffix of s. Suffix array H of s is a permutation of the integers $0,..., n-1$ such that $H[i]$ is the start position of the ith smallest suffix, as illustrated in Fig. 4, in which we show all the suffixes of ccagaca\$ (Fig. 4a), sorted suffixes (Fig. 4b), and the corresponding array H (Fig. 4c), which contains the positions of all the sorted suffixes′ first character in s.

The relationship between H and the BWT array L can be determined by the following formulas [9]:

$$\begin{cases} L[i] = \$, & if\ H[i] = 0; \\ L[i] = s[H[i]-1], & otherwise. \end{cases} \tag{4}$$

Since a suffix array can be generated in $O(n)$ time (cf. for instance [24, 34, 51]), L can then also be created in linear time. However, most algorithms for constructing a suffix array require at least $O(n\log n)$ bits of working space, which is prohibitively high and amounts to 12 GB for the human genome. Recently, Hon *et al.* [24] proposed a space-economical algorithm that uses n bits of working space and requires only < 1 GB memory at peak time for constructing L of the human genome. This algorithm is further improved

suffixes:	*sorted suffixes*:	*suffix array*:
0 c c a g a c a \$	\$	7
1 c a g a c a \$	a \$	6
2 a g a c a \$	a c a \$	4
3 g a c a \$	a g a c a \$	2
4 a c a \$	c a \$	5
5 c a \$	c a g a c a \$	1
6 a \$	c c a g a c a \$	0
7 \$	g a c a \$	3
(a)	**(b)**	**(c)**

Fig. 4 Suffixes, sorted suffixes, and suffix array

by Nong [51], whose approach needs only O($|\Sigma| \log n$)-bits space. Either of the methods can be used for our purpose.

Algorithm Description

In this section, we present our main algorithm. First, we show a breadth-first search of *trie*($\boldsymbol{R}$) against $BWT(\bar{s})$ in "Searching Tries Over Pattern Strings". Then, in "Searching PMMs Over Pattern Strings", we discuss how the failure function in an PMM can be employed to speed up the working process. "Correctness and Time Complexity" is devoted to the correctness proof and the time complexity analysis.

Searching Tries Over Pattern Strings

It is easy to see that exploring a path in a trie T over $\boldsymbol{R}$ corresponds to scanning a pattern $r \in \boldsymbol{R}$. If we explore, at the same time, the L array (= $BWT(\bar{s})$), we will find all the occurrences of r (without \$ involved) in s. Obviously, by a depth-first search of T, this can be done very efficiently. However, to be able to utilize the failure function to reduce the number of subranges (within L) to be searched at each step, we need to explore T in the breadth-first manner. For this purpose, we use a queue Q to control the searching process.

In Q, each entry is a pair $<v, [a, b]>$ with v being a node in T and $a \leq b$, used to indicate a subsegment within $F_{l(v)}$. For example, when searching the trie shown in Fig. 1a against the L array shown in Fig. 2c, we may have an entry like $<v_1, [1, 3]>$ in Q to represent a subsegment $F_a[1... 3]$ (the first to the third entry in F_a) since $l(v_1)$ = 'a'. In addition, for technical convenience, we use F_ϵ to represent the whole F. Then, $F_\epsilon[a... b]$ represents the segment from the ath to the bth entry in F.

Algorithm 1: $trieSearch(T,\ LF)$

Input : T - trie over a set of patterns; LF - arrays L and F over a target
Output: $\mathcal{R}$ - all occurrences of patterns in target

```
1  v ← root(T); R ← φ;
2  enqueue(Q, <v, [1, |s|]>);
3  while Q is not empty do
4      (v, a, b) ← dequeue(Q);
5      if output(v) ≠ φ then
6          R ← R ∪ {<output(v), l(v), a, b>}
7      let v_1, ..., v_k be the children of v;
8      denote F_{l(v)}[a .. b] by π;
9      for i = 1 to k do
10         let x = l(v_i);
11         if search(x, π) ≠ φ then
12             let search(x, π) = <x, [α, β]>;
13             enqueue(Q, <v_i, [α, β]>);
14 return R;
```

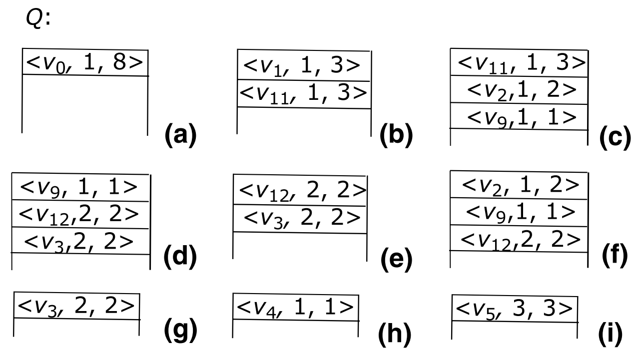

Fig. 5 Illustration for Step 1–9

In addition, each time we encounter a node v with $output(v) \neq \phi$, we will create a quadruple $<output(v), l(v), a, b>$ to record the occurrences of all those pattern strings represented by $output(v)$ in s. Thus, the result of this process is a set $\mathcal{R}$ containing all such quadruples.

In the algorithm, we first enqueue $<root(T), [1, |s|]>$ into queue Q (append at the end of Q) (see lines 1 - 2). Then, we go into the main **while**-loop (lines 3 - 13), in which we will first dequeue the first element from Q (taken out from the front of Q), stored as a pair $<v, [a, b]>$ (line 4). Then, we will check whether $output(v)$ is empty. If it is not the case, a quadruple $<output(v), l(v), a, b>$ will be added to the result $\mathcal{R}$ (see line 5). (In practice, we can store compressed suffix arrays [44, 62] and use their relationship with BWT to calculate positions of pattern occurrences in s.) For each child v_i of v, we will determine a segment in L by executing $search(x, \pi)$, where $x = l(v_i)$ and $\pi = l(v)$, $<[a... b]]>$ (= $F_{l(v)}[a.. b]$). Assume that $search(l(v_i), \pi) = <l(v_i), [\alpha_i, \beta_i]>$. We will then enqueue each $<v_i, [\alpha, \beta]>$ into Q. (see lines 12–13.)

The following example helps for illustration.

Example 2 Consider all pattern strings given in Example 1 again. The trie T over these short strings are shown in Fig. 1a. In order to find all the occurrences of them in s = ccagaca\$, we will run *trieSearch*( ) on T and the LF shown in Fig. 2c. (By LF, we mean the L and F arrays together.)

In the execution of *trieSearch*( ), the following steps will be carried out.

- Step 1: Enqueue $<v_0, [1, 8]>$ into Q, as illustrated in Fig. 5a.
- Step 2: Dequeue the first element $<v_0, [1, 8]>$ from Q. Figure out the two children of v_0: v_1 and v_{11}. First, for v_1, we have $l(v_1)$ = a. By executing $search$(a, $F_\epsilon[1.. 8]$), we get <a, [1, 3]> and then enqueue $<v_1, 1, 3>$ into Q. For v_{11}, we have $l(v_{11})$ = c and get <c, [1, 3]> by execut-

ing *search*(c, F_ϵ[1.. 8]). So, $<v_{11}$, [1, 3]$>$ will also be enqueued into Q. See Fig. 5b for illustration.

– Step 3: Dequeue the first element $<v_1$, [1, 3]$>$ from Q. v_1 also has two children: v_2 and v_9. For v_2, we have $l(v_2)$ = c. By executing *search*(c, F_a[1.. 3]), we get $<$c, [1, 2]$>$. For v_9, we have $l(v_9)$ = g and get $<$g, [1, 1]$>$ by executing *search*(g, F_a[1.. 3]). Similarly, we will consecutively *enqueue* $<v_2$, [1, 2]$>$ and $<v_9$, [1, 1]$>$ into Q. See Fig. 5c.

The remaining steps 4, 5, 6, 7, 8, 9 will be done in the same way as above and Q will be accordingly changed as shown in Fig. 5d–i, respectively. Here, special attention should be paid to Step 5 when $<v_9$, [1, 1]$>$ is dequeued from Q. Since $output(v_9) = r_2$, we will store $<r_2$, g, [1, 1]$>$ in $\mathcal{R}$ as part of the result (see line 5 in *trieSearch*( )), which shows that r_2 appears at g_1-position in $s = c_1c_2a_1g_1a_2c_3a_3\$$.

Searching PMMs Over Pattern Strings

In the algorithm discussed in the previous section, the failure function f is totally ignored. Indeed, due to the difference between the scanning of s and the searching of $BWT(\bar{s})$, the failure function f cannot be used in a way as *Aho-Corasick* $'$s does. It is because when searching $BWT(\bar{s})$ along a path in T, what we will produce is a search sequence: a sequence of pairs; and for any two such sequences (along two different paths in T), which spell out a same sequence of characters, they may have different sequences of intervals. For instance, along the path $v_2 \to v_3$ shown in Fig. 6, we will create a sequence of pairs: $<$c, [1, 2]$>$, $<$a, [2, 2]$>$ while along the path $v_{11} \to v_{12}$ the sequence generated is $<$c, [1, 3]$>$, $<$a, [2, 2]$>$. Although they have the same sequence of characters: ca, their sequences of intervals are different: one is [1, 2][2, 2] and the other is [1, 3][2, 2].

In addition, since a pair sequence cannot be created in a reverse order (by searching a PMM bottom-up), it is completely impossible for us to use the skip-table utilized in the *Boyler-Moore* $'$s algorithm [8] (by which substrings of s need to be scanned backwards), or the DAWG structure (directed acyclic word graph) in the *Crochemore* $'$s algorithm [17] (by which a DAWG also needs to be searched bottom-up.) However, the failure function can be really employed to reduce the number of subranges of L to be searched during an execution of *search*( ).

To this end, we will associate each node v in T with an extra item - the corresponding interval [α_v, β_v], referred to as $I(v)$, which is found for $l(v)$ by running *search*( ). That is, along an edge $w \to v$ in T, we will have *search*($l(v)$, $l(w)$, $<[\alpha_w, \beta_w]>$) = $<l(v)$, [α_v, β_v]$>$. (See Fig. 6 for illustration.) With the help of such intervals, the failure function f can be utilized as follows.

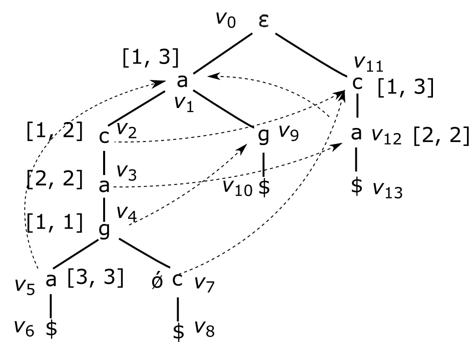

Fig. 6 Illustration for searching PMM

Lemma 1 *Let u, v be two nodes in* **A** *such that* $f(v) = u$. *Then,* $I(v) \subseteq I(u)$.

Proof According to the definition of $f(v) = u$, $P(u)$ is a suffix of $P(v)$. Without loss of generality, assume that $P(v) = z_1 \ldots z_i\, z_{i+1} \ldots z_{i+j}$ and $P(u) = z_{i+1} \ldots z_{i+j}$ with $i, j \geq 0$. Then, by the execution of *trieSearch*(T, LF), along $P(v)$ and $P(u)$, two sequences of pairs will be generated:

along $P(v)$: $\pi_1, \ldots, \pi_i, \pi_{i+1}, \ldots, \pi_{i+j}$,

along $P(u)$: $\pi'_1, \ldots, \pi'_j$,

where $\pi_1 = <z_1, F_{z_1}>$, $\pi'_1 = <z_{i+1}, F_{z_{i+1}}>$, $\pi_{l+1} = search(z_l, \pi_l)$ for $l = 1, \ldots, i + j$ - 1, and $\pi'_{k+1} = search(z_k, \pi'_k)$ for $k = 1, \ldots, j'$ -1.

Let I_l be the interval in pair π_l ($l = 1, \ldots, i + j$). Let I'_k be the interval in pair π'_k ($k = i + 1, \ldots, i + j$). We must have $I_k \subseteq I'_{k-i}$ ($k = i + 1, \ldots, i + j$). Thus, $I(v) = I_{i+j} \subseteq I'_j = I(u)$. □

This lemma enables us to design an efficient procedure to replace *search*( ) for creating $I(v)'$s, as described below.

Let $w \to v$ be an edge in T. Assume that $f(v) = u$. Let $l(u) = z$. Since we explore T in the breadth-first manner, u must be visited before v. Therefore, its interval $I(u) = [\alpha_u, \beta_u]$ must have been created when we meet v. Then, in terms of Lemma 1 (which shows $I(v) \subseteq I(u)$), a simple inference can be made:

If $I(u) \subseteq I(w)$, we must have $I(v) = I(u)$.

It is because in this case, when we search $I(w)$ to find the first and last appearance of v, we will definitely first meet α_u (from top) and β_u (from bottom).

As an example, consider the search process of $\boldsymbol{A}$ shown in Fig. 6 against the LF of $\bar{s}$ shown in Fig. 2c, where s = acagacc\$. By the breadth-first search of T, v_{12} will be visited before v_3 and $f(v_3) = v_{12}$. As shown in Fig. 6, we have $v_2 \to v_3$ and $I(v_{12}) = [2, 2] \subset I(v_2) = [1, 3]$. Then, we definitely have $I(v_3) = I(v_{12}) = [2, 2]$.

Therefore, we will change *search*( ) to combine the above inference mechanism into the process, and refer to the changed procedure as *searchI*(v, w, f(v), LF) to indicate its difference from *search*( ). But its output is the same as *search*( ).

According to the above discussion, we give the following algorithm, which works almost in the same way as *trie-Search*( ). The only difference consists in the use of *searchI*( ). That is, we will still explore T breadth-first. However, each time we encounter a node v, we will call *searchI*(v, w, f(v), LF) (instead of *search*( )) to determine the interval for v, where w represents the parent of v.

Algorithm 2: $pmmSearch(\boldsymbol{A}, LF)$

```
Input  : T - trie over a set of patterns; LF - arrays
         l and F over a target
Output: R - all occurrences of patterns in target
v ← root(T); R ← φ;
enqueue(Q, <v, [1, |s|]>);
while Q is not empty do
    v ← dequeue(Q);
    if output(v) ≠ φ then
        R ← R ∪ {<output(v), l(v), I(v)>}
    let v_1, ..., v_k be the children of v;
    for i = 1 to k do
        <x, [α, β]> ← searchI(v_i, v, f(v_i), LF);
        if [α, β] ≠ φ then
            enqueue(Q, <v_i, [α, β]>);
return R;
```

Correctness and Time Complexity

In this section, we prove the correctness of *pmmSearch* (T, LF) and analyze its time complexity.

First, we have the following lemma.

Lemma 2 *Let* u, v *be two nodes in* $\mathbf{A}$ *such that* $f(v) = u$. *Let* w *be the parent of* v *in* T. *The interval returned by searchI*(v, w, f(v), LF) *is correct.*

Proof This lemma can be directly derived from Lemma 1. □

Proposition 1 *Let* $\mathbf{A}$ *be a PMM constructed over a collections of pattern strings*: $r_1,\ldots, r_m$, *and* LF *a BWT-mapping established for a reversed target string* $\bar{s}$. *Let* $\mathcal{R}$ *be the result of pmmSearch* (T, LF). *Then, for each* r_j, *if it occurs in* s, *there is a quadruple* $<output(v), l(v), [\alpha, \beta]> \in \mathcal{R}$ *such that* $r_j \in output(v)$, $l(v)$ *is equal to the last character of* r_j, *and* $F_{l(v)}[\alpha], F_{l(v)}[\alpha + 1],\ldots, F_{l(v)}[\beta]$ *show all the occurrences of* r_j *in* s.

Proof We prove the proposition by induction on the height h of $\boldsymbol{A}$, which is defined to be the number of edges on the longest downward path from the root to a leaf node.

Basic step. When $h = 1$. The proposition trivially holds.

Induction hypothesis. Suppose that when the height of $\boldsymbol{A}$ is l, the proposition holds. We consider the case that the height of $\boldsymbol{A}$ is $l + 1$. Let $\boldsymbol{A}'$ be a PMM obtained by removing all the leaf nodes in $\boldsymbol{A}$. Then, the height of $\boldsymbol{A}'$ is at most l. According to the induction hypothesis, the interval generated by applying *pmmSearch*( ) to $\boldsymbol{A}'$ must be correct. Now, we consider a leaf node v in $\boldsymbol{A}'$. Let $v_1,\ldots, v_k$ be the children of v in $\boldsymbol{A}$. Then, in terms of Lemma 2, $I(v_i)$ produced by executing *searchI*(v_i, v, $f(v_i)$, LF) for $i = 1,\ldots, k$ must also be correct. Considering that all the nodes in $\boldsymbol{A}$ are visited in the breadth-first manner, the claim in the proposition is correct. □

Concerning the time complexity, we check the main **while**-loop, in which each node v in T is accessed only once. Therefore, the running time of *pmmSearch*(T, LF) is bounded by $O(\sum_{v \in T} \delta_v)$, where δ_v represents the cost for an execution of *searchI*( ) to find $I(v)$.

By using the *rankAll* technique [9], δ_v can be reduced to O(1), but the space overhead will be greatly increased. In the next section, our focus will be on how to further reduce this cost by integrating the Wavelet tree searching [22] into our algorithm to achieve this purpose.

Integrating Wavelet Tree Searching into PMM Searching

As mentioned in the previous section, the cost of searching L can be reduced to O(1) by using the *rankAll* technique [9], but with $O(n|\Sigma|\log n)$ extra space being required. It is prohibitively high when target strings are very long, and Σ is large, such as protein sequences and English documents. To mitigate the problem to some extent, we store L as a *Wavelet* tree [22] and integrate a modified Wavelet tree search into our algorithm. In general, the space requirement of a Wavelet tree is bounded by $O(n\log|\Sigma|)$, but with a higher searching time $O(\log|\Sigma|)$. However, in our implementation, the average searching time can be decreased to less than $O(\log|\Sigma|)$ while the space requirement remains not incremented.

In the following, we first give a detailed description of the Wavelet tree in "Wavelet Trees". Then, in "Modified Wavelet Trees", we slightly change the Wavelet tree searching method. Finally, in "Integration of Wavelet Tree Search into PMM Breadth-First Search", we discuss the integration of the Wavelet tree search into our strategy.

Wavelet Trees

The key idea behind the Wavelet tree is to store $L = BWT(\bar{s})$ as a balanced binary tree T_L of height $|\Sigma|$, as illustrated in Fig. 7, where we show how $BWT(\bar{s})$ = acgcac$a is recursively decomposed and stored in a binary tree structure.

Its purpose is to reduce the space for storing *rankAll* arrays [9], but more time is needed to figure out the rank $z[i]$ for $z \in \Sigma$, i.e., the number of z's appearances up to $L[i]$. Such a number needs to be computed to evaluate *search*( ) or *searchI*( ).

Formally, a Wavelet tree can be constructed as follows.

1. In a Wavelet tree, each node u represents a (sub)string L and contains a Boolean string B representing a partition of L.
2. Let $L = a_1a_2 \ldots a_k$. Let Σ be the set of all characters appearing in L. We divide Σ evenly into two subsets Σ_l and Σ_r. Then, a Boolean array B is constructed as follows.
   - For each $i \in \{1, \ldots, k\}$, if $L[i] \in \Sigma_l$, then $B[i] = 0$;
   - otherwise, $B[i] = 1$.
3. The left child u_l of u represents a substring L_l of L made up of all those characters of $L \in \Sigma_l$, which appear (not necessarily contiguously) in the same order as in L. Further, Σ_l will be divided into subsets Σ_{ll} and Σ_{lr}; and in terms of Σ_{ll} and Σ_{lr}, we partition L_l and construct the corresponding Boolean string B_l as above. That is, for each $i \in \{1,\ldots|L_l|\}$, if $L_l[i] \in \Sigma_{ll}$, then $B_l[i] = 0$; otherwise, $B_l[i] = 1$.
4. In the same way, we will construct the right child u_r of u and the substring L_r of L made up of all those characters of $L \in \Sigma_r$. Similar to Σ_l, Σ_r will also be further divided into two parts: Σ_{rl} and Σ_{rr} and the corresponding Boolean string B_r will be created.
5. We repeat steps (2) and (3) until a substring is met, which contains only the same characters.

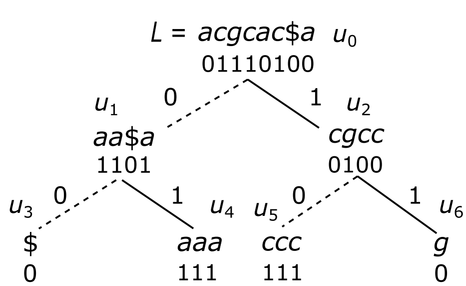

Fig. 7 A wavelet tree

In terms of the above discussion, we can construct a tree as shown in Fig. 7, in which the strings in nodes are not actually stored, but shown here for ease of understanding.

The tree is called a Wavelet tree due to its analogy with the wavelet transform for signals, which recursively decomposes a signal into low-frequency and high-frequency components [48].

In the tree shown in Fig. 7, the root u_0 represents $L_0 =$ acgcac \$ a. We first divide the set of all the different characters appearing in L_0: $\Sigma_0 = \{\$, \mathtt{a}, \mathtt{c}, \mathtt{g}\}$ evenly into two parts: $\Sigma_l^0 = \{\$, \mathtt{a}\}$ and $\Sigma_r^0 = \{\mathtt{c}, \mathtt{g}\}$. Then, the corresponding Boolean string is set to be $B_0 = 01110100$, representing a partition of L_0, by which for any character $z \in \Sigma_l^0$ the corresponding bit in B_0 is set to 0 and z will be sent to the left child while for any character $z' \in \Sigma_r^0$ the corresponding bit in B_0 is set to 1 and z' will be sent to the right child

Now, let us have a close look at the left child u_1 of the root u_0. It represents the substring $L_1 = aca\,\$\,g$, made up of all the characters $\in \Sigma_l^0 = \{\$, \mathtt{a}\}$ and rendered in the same order as in the original string L_0. If we further divide all its characters in Σ_l^0 into $\Sigma_l^1 = \{\$\}$ and $\Sigma_r^1 = \{\mathtt{c}\}$, we will get its Boolean string $B_1 = 0010$. In the same way, we can check all the other nodes in the tree shown in Fig. 7.

Using the Wavelet tree built over $L = BWT(\bar{s})$, $z[i]$ can be evaluated by exploring a root-to-leaf path as below.

Initially, the current node u is set to be the *root* u_0.

1. At node u, count the number x of 0s or 1s in the range $B[1..\, i]$, depending on whether $z \in \Sigma_l$, or Σ_r. If u is a leaf node, it contains only the same characters and x is set to be i. which is returned as the result.
2. If $z \in \Sigma_l$, go to the left child u_l. $B \leftarrow B_l$, $i \leftarrow x$. $\Sigma \leftarrow \Sigma_l$. $u \leftarrow u_l$. Go to (1).
3. If $z \in \Sigma_r$, go to the right child u_r. $B \leftarrow B_r$, $i \leftarrow x$. $\Sigma \leftarrow \Sigma_r$. $u \leftarrow u_r$. Go to (1).

For instance, to evaluate a[6] over acgcac \$ a, the path $u_0 \rightarrow u_1 \rightarrow u_4$ in Fig. 7 will be searched. First, in the root u_0, we will count the number of 0 s in $B_{u_0}[1..\,6] = 011101$ since $a \in \Sigma_l^0 = \{\$, a\}$. It is $x = 2$. Then, we go to the left child u_1 of u_0, in which we count 1s in $B_{u_1}[1..\,x] = B_{u_1}[1..\,2] = 11$ since $a \in \Sigma_r^1 = \{a\}$. This time, we get $x = 2$ again. Next, we go to the right child u_4 of u_1, and get $x = 2$. Since it is a leaf node, we get the answer a[6] = 2.

Modified Wavelet Trees

Since any Wavelet tree is balanced and each level corresponds to an even splitting of a certain (sub)set of characters, the length of any path in it is bounded by O(log $|\Sigma|$). Besides, the counting of 0s or 1s in a Boolean string (stored in any node) can be done in O(1) time by using the so-called *RRR* data structure discussed in [54], or the succinct data structure in [29]. Therefore, the cost of calculating $z[i]$ is bounded by O(log $|\Sigma|$). The space requirement of both *RRR* and the succinct data structure is bounded by O(n). Thus, the size of a Wavelet tree is bounded by O($n \cdot \log|\Sigma|$).

However, we can also store a Boolean string B as an integer array A with $A[i]$ being the number of 0s in $B[1..i]$. Then, the counting of 0s or 1s can also be done in O(1) time as illustrated in Fig. 8.

We pay attention to array A in u_0. To know the number of 0s up to position 5 in B in u_0, simply access $A[5] = 2$. we get $0[5] = 2$. If you want to know the number of 1s up to position 5, simply compute 5 - $A[5] = 3$. Therefore, we have $1[5] = 3$.

But the array A is much simpler than *RRR* [54] or the succinct data structure [29].

The drawback of this method is that we need log n bits to store an integer in A. For this reason, we replace A with a compact array U_A, in which only part of A is stored. For example, we can divide A into a set of buckets of the same size and for each bucket only a value will be stored in the compact array. Obviously, doing so, more searching effort is required to find missing values. In practice, the size of a bucket (referred to as a compact factor ω) can be set to different values. For instance, we can set $\omega = 4$, indicating that for each four contiguous elements in A only one value will be stored. That is, we will not store all the values in A in u_0, but only store $A[4]$ and $A[8]$ in the corresponding compact array (as illustrated by the values marked grey in Fig. 9).

Obviously, each $A[j]$ can be easily derived from U_A by using one of the following formulas:

$$A[j] = U_A[i] + \tau \tag{5}$$

where $i = \lfloor j/\omega \rfloor$ and is the number of 0's appearances within $B[i \cdot \omega + 1..j]$ which have to be searched, or

$$A[j] = U_A[i'] - \tau' \tag{6}$$

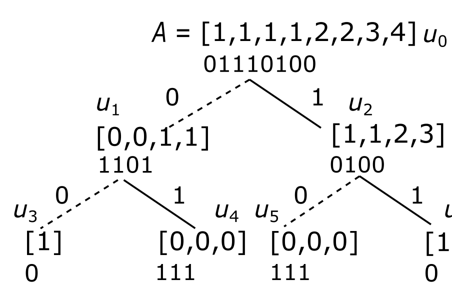

Fig. 8 A modified wavelet tree

where $i' = \lceil j/\omega \rceil$ and τ' is the number of 0′s appearances within $B[j + 1..i' \cdot \omega]$. Also, τ' has to be obtained by searching part of B.

Therefore, we need two procedures: *toRight*(B, j, ω) and *toLeft*(B, j, ω) to find τ and τ', respectively. In terms of whether $j - i \cdot \omega \leq i' \cdot \omega - j$, we will call *toRight*(B, j, ω) or *toLeft*(B, j, ω) so that fewer entries in B will be scanned to find $A[j]$.

In terms of [9], the BWT arrays are normally well clustered and proved to be a good data compression approach for data transmission. Thus, array B should be even better clustered since it contains only two different values: 0 and 1, and so does array A. Thus, we can attach two extra values with each $U_A[i]$. They are the numbers of consecutive values same as $U_A[i]$ just before and after $U_A[i]$, respectively, mainly used to improve the performance of *toRight*( ) and *toLeft*( ). Note that we need only log ω bits to represent such a number.

Integration of Wavelet Tree Search into PMM Breadth-First Search

As an important step to integrate the Wavelet tree searching into our algorithm, we designed a procedure to do a multiple-value searching over a Wavelet tree (for $L = BWT(\bar{s})$), instead of a single-value searching as discussed in "Wavelet Trees".

We first consider a set $X = \{x_1(i_1),..., x_l(i_l)\}$ as a query to find out the number of x_j's appearances prior to $L[i_j]$ (including $L[i_j]$) for each $j \in \{1,...., l\}$. We need to evaluate such a query when we try to determine $<l(v_1), [\alpha_{v_1}, \beta_{v_1}]>$, ..., $<l(v_r), [\alpha_{v_r}, \beta_{v_r}]>$ for all the children v_1, ..., v_r of a certain node v. Assume that the pair associated with v is $\pi = <l(v), [\alpha_v, \beta_v]>$. Then, for each v_j, we need to find two values:

$x_j[L_\pi^1 - 1]$, and

$x_j[L_\pi^2]$,

where $x_j = l(v_j)$.

If $x_j[L_\pi^1 - 1] = x_j[L_\pi^2]$, L_π contains no $l(v_j)$. Otherwise, $[\alpha_j, \beta_j] = [x_j[L_\pi^1 - 1] + 1, x_j[L_\pi^2]]$.

As an example, consider $\pi = <a, [1, 3]>$ (representing a segment of F shown in Fig. 2c). We have $L_\pi = L[2..4]$, $L_\pi^1 = 1$, and $L_\pi^2 = 4$. To find the first and the last appearance

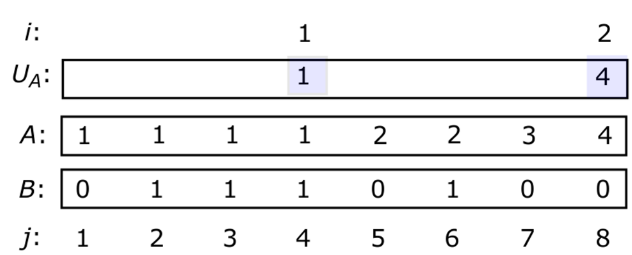

Fig. 9 Illustration for U_A

of c in L[2.. 4], we only need to find c[L_π^1 - 1] = c[2 - 1] = c [1] = 0 and c[L_π^2] = c[4] = 2. So the corresponding range is [c[2 − 1] + 1, c[4]]= [1, 2].

Let T_L be the Wavelet tree over $L = BWT(\bar{s})$. To evaluate query $X = \{x_1(i_1),..., x_l(i_l)\}$, we give an algorithm below, in which the following notations are used:

- Q: a queue to control the search of T_L in the breadth-first manner;
- (v, X): a pair, where $v \in T_L$, and X is query;
- Σ_l^v: the first half of the set of characters appearing in L_v;
- Σ_r^v: the second half of the set of characters appearing in L_v;
- B_v: the Boolean array stored in v;
- v_l: left child of v;
- v_r: right child of v.

The result $\mathcal{R}$ of the algorithm is also of the form $\{x_1(j_1), ..., x_l(j_l)\}$, but showing that for each x_f ($f \in \{1, ..., l\}$) the number of its appearances up to $L[i_f]$ is j_f.

Algorithm 3: $waveletSearch(T_L, X)$

Input : T_L - a Wavelet tree over L
Output: $\mathcal{R}$ - for each $x(i) \in X$, number of x's appearances up to $L[i]$

1 $enqueue(Q, (root(T_L), X))$; $\mathcal{R} \leftarrow \phi$;
2 **while** Q is not empty **do**
3   $(v, X') \leftarrow dequeue(Q)$;
4   **if** v *is leaf* **then**
5     $\mathcal{R} \leftarrow \mathcal{R} \cup X'$; break;
6   let $X' = \{x_1(i_1), ..., x_l(i_l)\}$;
7   $Y_l \leftarrow \phi$; $Y_r \leftarrow \phi$;
8   **for** $j = 1$ to l **do**
9   **if** $x_j \in \Sigma_l^v$ **then**
10   find number k of 0's before $B_v[i_j]$ in B_v;
11       $Y_l \leftarrow Y_l \cup \{x_j(k)\}$;
12   **else**
13   find number k of 1's before $B_v[i_j]$ in B_v;
14       $Y_r \leftarrow Y_r \cup \{x_j(k)\}$;
15   $enqueue(Q, (v_l, Y_i))$; $enqueue(Q, (v_r, Y_r))$;
16 return $\mathcal{R}$;

The above algorithm is in essence a breadth-first searching of T_L. In queue Q, each element is a pair of the form (v, X). At the very beginning, Q contains only one element $(root(T_L), \{x_1(j_1), ..., x_l(j_l)\})$.

Each time we dequeue an element (v, X) out of Q, we will calculate, for each $x_j(i_j) \in X$, the number k of $0'$s up to $B_v[i_j]$ if $x_j \in \Sigma_l^v$, or the number k of $1'$s up to $B_v[i_j]$ if $x_j \in \Sigma_r^v$. In the former case, we append $x_j(i_j)$ to Y_l (see lines 9–11). In the latter case, $x_j(i_j)$ is added to Y_r (see lines 13–14).

After all subqueries ($x_j(i_j)'$s) in X are evaluated, both (v_l, Y_l) and (v_r, Y_r) will be enqueued into Q (see line 15).

Example 3 Consider the trie T shown in Fig. 1a again. The pair for the root v_0 is $\pi =$ <ϵ, [1, 8]>. Then, $L_\pi = L$[1.. 8], $L_\pi^1 = 1$, and $L_\pi^2 = 8$. For its two children v_1 and v_{11}, we will construct a query $X = \{ l(v_1)(L_\pi^1 - 1), l(v_1)(L_\pi^2), l(v_{11})(L_\pi^1 - 1), l(v_{11})(L_\pi^2)\}$ = {a(0), a(8), c(0), c(8)}. By executing $waveletSearch(T_L, X)$ (T_L is shown in Fig. 8), the following steps will be carried out.

- Step 1: Enqueue <u_0, X> into Q.
- Step 2: Dequeue the first element from Q. We have
  $u = u_0$, and X' = {a(0), a(8), c(0), c(8)}.
  Σ_0 = {$ a, c, g }; Σ_l^0 = {$, a }; Σ_r^0 = {c, g };
  Y_l^0 = {a(0), a(4)}; Y_r^0 = {c(0), c(4)};
  $Q = \{(u_1, Y_l^0), (u_2, Y_r^0)\}$.
- Step 3: Dequeue the first element from Q. We have
  $u = u_1$, and X' = {a(0), a(4)}.
  Σ_1 = {$, a };
  Σ_l^1 = {$}; Σ_r^1 = { a };
  $Y_l^1 = \phi$; Y_r^1 = {a(0), a(3)};
  $Q = \{(u_2, Y_r^0)\ (v_4, Y_r^1)\}$.
- Step 4: Dequeue the first element from Q. We have
  $u = u_2$, and X' = {c(0), c(4)}.
  Σ_2 = {c, g }; Σ_l^2 = { c };
  Σ_r^2 = {g};
  Y_l^2 = {c(0), c(3)}; $Y_r^2 = \phi$;
  $Q = \{(v_4, Y_r^1), (v_5, Y_l^2)\}$.
- Step 5: Dequeue the first element from Q. We have
  $u = u_3$, and X' = {a(0), a(3)}.
  u_3 is a leaf node. X' is added to $\mathcal{R}$.
- Step 6: Dequeue the first element from Q. We have
  $u = u_5$, and X' = {c(0), c(3)}.
  u_5 is a leaf node. X' is added to $\mathcal{R}$.

The final result is $\mathcal{R}$ = {a[0], a[3]),c[0], c[3]}.

By using the algorithm $waveletSearch(\ )$, our basic algorithm can be changed as shown in Algorithm 4.

Algorithm 4: *mPmmSearch*(T, LF)

```
   Input  : T - trie over a set of patterns; LF - arrays
            L and F over a target
   Output: R - all occurrences of patterns in target
1  v ← root(T); R ← φ;
2  enqueue(Q, <v, 1, |s|>);
3  while Q is not empty do
4      (v, a, b) ← dequeue(Q);
5      if output(v) ≠ φ then
6          R ← R ∪ {<output(v), l(v), a, b>}
7      let v_1, ..., v_k be the children of v;
8      denote F_{l(v)}[a .. b] by π; Denote l(v_i) by x_i (1 ≤
         i ≤ k);
9      X ← {x_1(L^1_π - 1), x_1(L^2_π), ..., x_k(L^1_π - 1),
         x_k(L^2_π)};
10     R' ← waveletSearch(T_L, X);
11     for j = 0 to k - 1 do
12         if R'[2j + 1] ≠ R'[2j + 2] then
13             enqueue(Q, <v_i, R'[2j + 1] + 1, R'[2j
                 + 2]>);
14 return R;
```

This algorithm is almost the same as Algorithm 2. The only difference consists in the searching of T_L. For each encountered node v in trie T, we will first create a query $X = \{x_1(L^1_\pi - 1), x_1(L^2_\pi), ..., x_1(L^k_\pi - 1), x_k(L^2_\pi)\}$ (see lines 7 - 9), where $v_1, ..., v_k$ are the children of v, $x_1 = l(v_1), ..., x_k = l(v_k)$, and π is a segment of F associated with v. Then, we call *wavwletSearch*(T_L, X) to find $<x_j, [\alpha_j, \beta_j]>$ for each $j \in \{1, ..., k\}$.

For simplicity, the inference based on failure functions is not included in the above algorithm. But it is an easy task to extend the algorithm with the inference mechanism involved.

Proposition 2 *Let* $<output(v), l(v), a, b>$ *be a tuple in* $\mathcal{R}$ *returned by mPmmSearch*($\boldsymbol{A}$, *LF*). *Then, for any* $r \in output(v)$, $l(v)$ *is equal to the last character of* r, *and* $F_{l(v)}[a]$, $F_{l(v)}[a+1]$, ..., $F_{l(v)}[b]$ *show all the occurrences of* r *in* s.

Proof Comparing lines 7–13 of *mPmmSearch*($\boldsymbol{A}$, *LF*)( ) and line 7–11 of *mmSearch*($\boldsymbol{A}$, *LF*)( ), we can see that by *mPmmSearch*($\boldsymbol{A}$, *LF*)( ) the multi-value search of Wavelet trees is used to speed up computation. But the results of both of them are the same. Therefore, the proposition holds. □

According to the analysis of "Wavelet Trees", the cost of searching a Wavelet tree is bounded by $O(\log|\Sigma|)$. Thus, the cost of the multi-value searching of a Wavelet tree should be bounded by $O(\frac{1}{d}\log|\Sigma|)$, where d is the smallest outdegree of an internal node in T. Therefore, the time complexity is bounded by $O(\frac{1}{d}n\log|\Sigma|)$.

Experiments

In our experiments, we have tested altogether seven strategies:

- suffix trees (*ST* for short, [61]),
- BWT transformation (*BWT* for short, [9]),
- hash-based (*hash* for short, [30]),
- Aho-Corasick′s algorithm (*AC* for short, [1]),
- Wu-Manber′s algorithm (*WM* for short, [16]),
- Crochemore′s algorithm (*Cr* for short, [17]), and
- *mPmmSearch* (*mPS* for short), discussed in this paper.

Among them, the codes for the suffix tree based and hash based methods are taken from the gsuffix package [3] while all the others, except ours, are taken from *github* (https://github.com). All of them are able to find all occurrences of every pattern in a target, written in C++, compiled by GNU make utility with optimization of level 2. In addition, all of our experiments are performed on a 64-bit Ubuntu operating system, run on a single core of a 2.40GHz Intel Xeon E5-2630 processor with 32GB RAM.

The test results are categorized in two groups: one is on a set of synthetic data and another is on a set of real data. For both of them, four genome and one protein sequences were downloaded from ensemble.org (ftp://ftp.ensembl.org/pub/release93/) and SMS (https://www.bioinformatics.org/sms2/random_protein.html), respectively. The size of the alphabet for the protein sequences is 20, much larger than that of DNA sequences. The patterns which were tested against different genome and protein sequences were generated by using the *wgsim* tool which is part of the *SAMtools* package [40]. In Table 3, we show all the tested sequences, as well as the time spent for constructing their BWT-arrays.

Tests on Synthetic Data Sets

All the synthetic data are created by simulating reads or protein sequences from the five sequences shown in Table 3,

Table 3 Genome and protein sequences, as well as time for constructing their BWT-arrays

Reference sequences*	Num. characters	Time (s)
Rat chr 1 (Rnor_6.0))	290,094,217	27.06
C. merolae (ASM9120v1)	16,728,967	1.64
Zebra fish (GRCz10)	1,464,443,456	181.35
Rat (Rnor_6.0)	2,909,701,677	317.28
protein	144,000,000	15.67

*The first four are genome sequences while the last one is a protein sequence

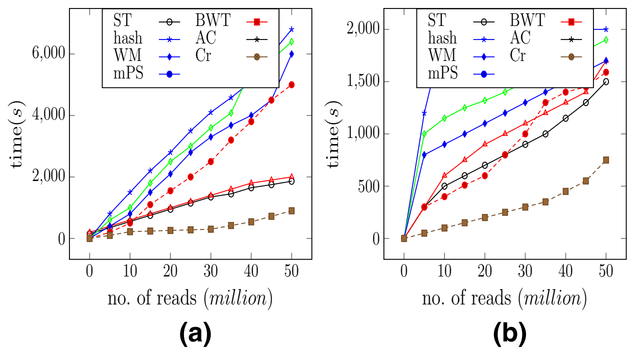

Fig. 10 Test results on varying amount of reads - Rat Chr1

Table 4 Time for false positive checking by hash method

No. of reads	30 M	35 M	40 M	40 M	50 M
Verification time	900 s	1550 s	2900 s	3195 s	4210 s

with varying lengths and amounts. It is done by using the wgsim program included in the SAMtools package [40] with default model for single read simulation.

Over such data, the impact of five factors on the searching time are tested: number l of patterns, length m of patterns, size n of target sequences, compact factors f_1 of *rankAlls* (see "Wavelet Trees") and compression factors f_2 of suffix arrays [44], which are used to find locations of matching reads (in a target sequence) in terms of their relationship with BWT-arrays.

Tests with Varying Amount of Reads

In this experiment, we vary the amount l of reads with $l = 5$, 10, 15,..., 50 millions while the reads are 50 bps or 100 bps in length extracted randomly from Rat chr1 and C. merolae genomes. For this test, the compact factors f_1 of rankAlls is set to be 32 for Rat chr1 to do some compression, but for C. merolae, f_1 is set to be 1, no compression at all. However, for both of them, the compression factor f_2 of suffix arrays are set to 16.

In Fig. 10a, b, we report the test results of searching the Rat chr1 for matching reads of 100 and 50 bps, respectively. From these two figures, it can be clearly seen that the hash based method has the worst performance while ours works best. For long reads (of length 100 bps), the suffix-based is worse than the BWT, but for short reads (of length 50 bps), they are comparable. Both the Crochemore′s and the Wu-Manber′s are worse than the BWT. But the Crochemore′s is better than the Wu-Manber′s. The poor performance of the hash-based is due to its false positive verification process. To see this, in Table 4, we show the time of this process for reads of 100 bps. The poor performance of both the BWT

Table 5 Time for PMM construction over reads of 100 bps

No. of reads	30 M	35 M	40 M	40 M	50 M
Time for PMM con	91 s	123 s	152 s	195 s	210 s

Table 6 Number of BWT-array accesses

No. reads	30 M	3 M	40 M	40 M	50 M
BWT	67954K	76532K	83321K	90732K	98165K
mPS	19105K	22177K	25261K	28227K	31204K

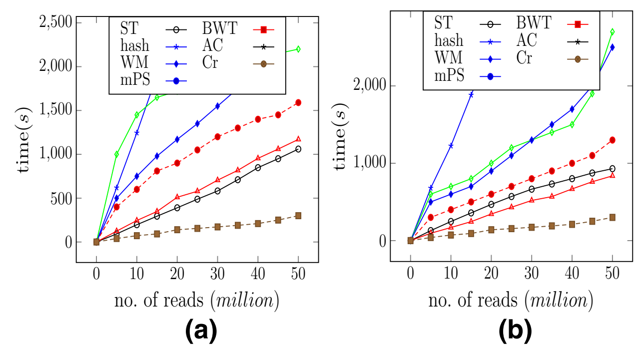

Fig. 11 Test results on varying amount of reads - C. merolae

and the suffix-based is due to the huge amount of reads and each time only one read is checked. In the opposite, for our method, the combination of PMMs and BWT arrays enables us to avoid repeated checking for similar reads. In these two figures, the time for constructing PMMs over reads is included. To see the impact of the construction of PMMs, we show the times for constructing them over different amounts of reads (of length 100 bps), demonstrated in Table 5.

The difference between the BWT and ours is due to the different number of BWT-array accesses as shown in Table 6. By the access of a BWT-array, we will scan a segment in the array to find the first and last appearance of a certain character from a read (by BWT) or a set of characters from more than one read (by ours).

Figure 11a, b show respectively the results for reads of length 50 bps and 100 bps over the C. merolae genome. Again, our method outperforms all the other six methods.

Tests with Varying Length of Reads

In this experiment, we test the impact of the read length on performance. For this, we fix all the other four factors but vary length m of simulated reads with $m = 35$, 50, 75, 100, 125,..., 200. The results in Fig. 12a shows the difference among seven methods, in which each tested set has 20 million reads simulated from the Rat chr1 genome with $f_1 = 32$

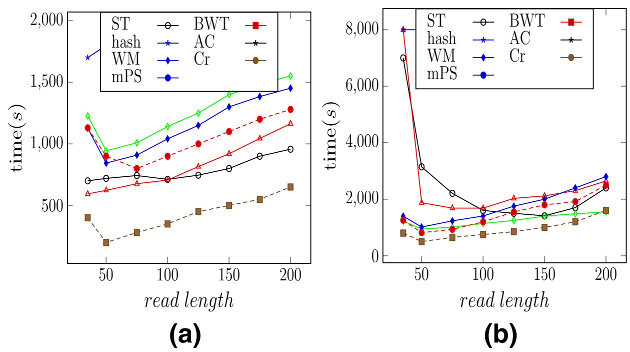

Fig. 12 Test results on varying length of reads - Rat Chr1

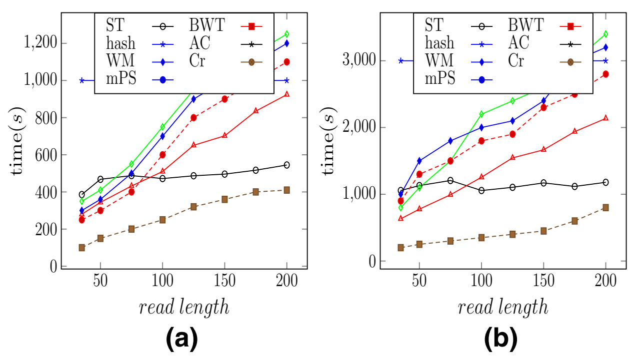

Fig. 13 Test results on varying length of reads - C. merolae

and $f_2 = 16$. In Fig. 12b, the results show the the case that each set has 50 million reads. Figure 13a, b show the results of the same data settings but on C. merolae genome with $f_1 = 1$ and $f_2 = 16$.

Again, in this test, the hash based performs worst while the suffix tree and the BWT method are comparable, and both the Crochemore′s and Wu-Manber′ are worse than them. Our algorithm uniformly outperforms the others when searching on short reads (shorter than 100 bps). It is because shorter reads tend to have multiple occurrences in genomes, which makes the trie used in ours more beneficial. However, for long reads, the suffix tree beats the BWT since on one hand long reads have fewer repeats in a genome, and on the other hand, higher possibility that variations occurred in long reads may result in earlier termination of a searching process. In practice, short reads are more often than long reads.

Tests with Varying Lengths of Target Sequences

To examine the impacts of varying sizes of targets, we have made four tests with each testing a certain set of patterns against different target sequences shown in Table 3. To be consistent with foregoing experiments, factors except sizes of targets remain the same as for the previous tests. In Fig. 15a, b, we show the searching time on each target sequence for 20 million and 50 million patterns of 50 characters, respectively. Figure 14a, b also demonstrate the results of 20 million and 50 million patterns but with each being of 100 bps. These figures show that, in general, as the size of a target sequence increases the time of pattern aligning for all the tested algorithms become longer. We also notice that the larger the size of a target sequence, the bigger the gaps between our method and the other algorithms. The hash-based is always much slower than the others. For the suffix tree, however, we only show the matching time for the two short genomes and the unique protein sequence. It is because the testing computer cannot meet its huge memory requirement for indexing the Zebra fish and Rat genomes (which is the main reason why people use the BWT, instead of the suffix tree, in practice.) Details for the patterns of 50 characters in Fig. 15a, b show that our methods is at least 5 times faster than the BWT and the suffix tree, which happened on the genomes. For the protein sequence, our algorithm is even more than 10 times faster than the others since the multi-character checking by our method is more effective for larger alphabets.

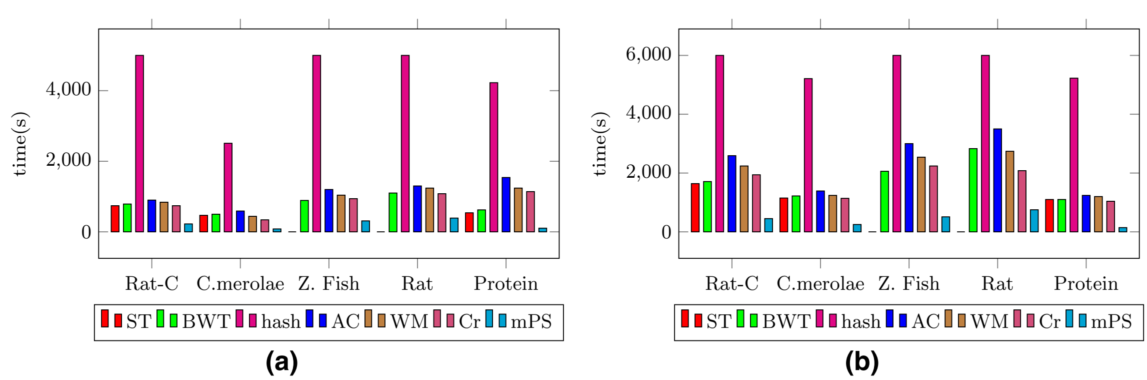

Fig. 14 Test results

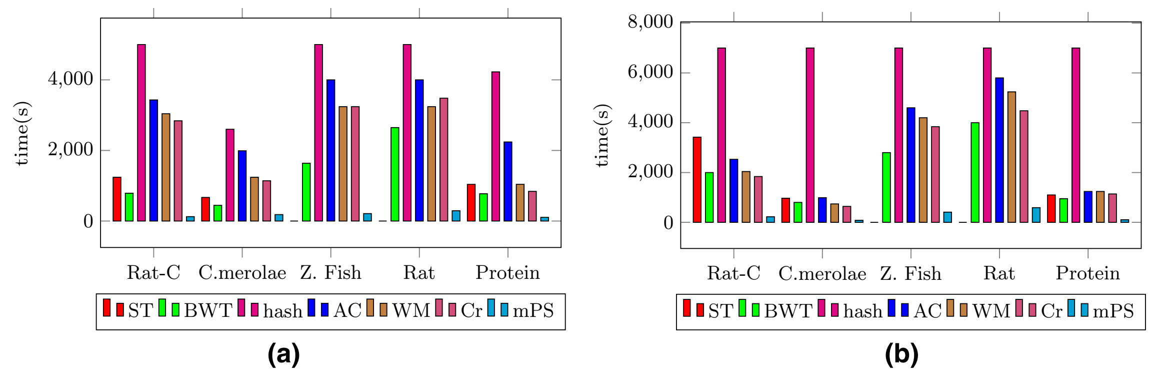

Fig. 15 Test results

Table 7 Sizes of samples

Sample ID	s1	s2	s3	s4	s5	s6	s7	s8	s9
No. of reads (million)	31.3	72.1	69.6	45.7	79.4	56.4	63.6	20.3	34.6

Now, let us have a look at Fig. 14a, b. Although our methods do not perform as good as for the 50 bp reads due to the increment of length of reads, they still gain at least 22% improvement on speed and nearly 50% acceleration in the best case, compared with the BWT.

Tests on Real Data Sets

For the performance assessment on real data, we obtain RNA-sequence data from the project conducted in an RNA laboratory at University of Manitoba (lab website: http://home.cc.umanitoba.ca/~xiej/, retrieved: 2016). This project includes over 500 million single reads produced by Illumina from a rat sample. Length of these reads is between 36 bps and 100 bps after trimming using Trimmomatic [7].

The reads in the project are divided into 9 samples with different amount ranging between 20 millions and 75 millions (see Table 7). Two tests have been conducted. In the first test, we mapped the 9 samples back to rat genome of ENSEMBL release 79 [18]. We were not able to test the suffix tree due to its huge index size. The hash-based method was ignored as well since its running time was too high in comparison with the BWT. In order to balance between searching speed and memory usage of the BWT index, we set $f_1 = 128$, $f_2 = 16$ and repeated the experiment 20 times. Figure 16a shows the average time consumed for each algorithm on the 9 samples.

Since the source of RNA-sequence data is the transcripts, the expressed part of the genome, we did a second test, in

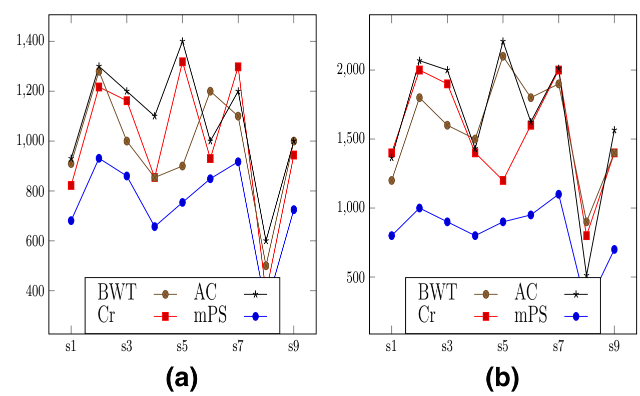

Fig. 16 Test on real data

which we mapped the nine samples again directly to the Rat transcriptome. This is the assembly of all transcripts in the Rat genome. This time more reads, which failed to be aligned in the first test, are able to be exactly matched. This result is showed in Fig. 16b.

From Fig. 16a, b, we can see that the test results for real data set are consistent with the simulated data. Our algorithm is faster than the BWT, the Aho-Corasick′s and the Crochemore′s on all nine samples. Counting all the data sets together, ours is more than 45% faster compared with these methods. Although the performance would be dropped by taking PMMs′ construction time into consideration, we are still able to save 40% time using our method.

Conclusion

In this paper, an efficient algorithm for solving the set matching problem has been discussed, by which we are required to locate and identify all substrings of a long string s which match some short strings from a set $\boldsymbol{R} = \{r_1, \ldots, r_m\}$. The main idea is to construct a *pattern matching machine* $\boldsymbol{A}$ over $\boldsymbol{R}$ and transform the reverse $\bar{s}$ of s to a BWT-array as index, $BWT(\bar{s})$, and search $\boldsymbol{A}$ against it. During the process, the failure function of $\boldsymbol{A}$ is used to reduce the subranges of $BWT(\bar{s})$ to be searched at each step. In addition, we change a single-character checking against $BWT(\bar{s})$ to a multiple-character checking, by which multiple searches of $BWT(\bar{s})$ are reduced to a single scanning of it. In this way, high efficiency can be achieved. Extensive experiments have been conducted, which shows that our method works better than the existing method for this problem.

As a future work, we will use the BWT to solve some other important problems, such as the string matching with *wild-card* symbols. A wild-card matches any characters, and we may have wild-cards in patterns, in targets, or in both of them.

Funding This study was funded by Natural Sciences and Engineering Research Council of Canada (DDG-2019-04100).

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Aho AV, Corasick MJ. Efficient string matching: an aid to bibliographic search. Commun ACM. 1975;23(1):333–40.
2. Aldwairi M. Hardware efficient pattern matching algorithms and architectures for fast intrusion detection. Ph.D dissertation, Graduate Faculty of North Carolina State University, USA. 2006.
3. Al-Khamaiseh K, ALShagarin S. A survey of string matching algorithms. Int J Eng Res Appl. 2014;4(7(Version 2)):144–56.
4. Baeza-Yates RA, Régnier M. Fast algorithms for two-dimensional and multiple pattern matching. In: Proceedings of SWAT '90 the second Scandinavian workshop on algorithm theory. Springer, Bergen; 1990. pp. 332–47.
5. Baeza-Yates RA, Gonnet GH. A new approach to text searching. Commun ACM. 1992;35(10):74–82.
6. Bauer S, Schulz MH, Robinson PN. gsuffix. 2014. http://gsuffix.Sourceforge.net/.
7. Bolger AM, Lohse M, Usadel B. Trimmomatic: bolger: a flexible trimmer for Illumina Sequence Data. Bioinformatics. 2014;30(15):2114–20.
8. Boyer RS, Moore JS. A fast string searching algorithm. Commun ACM. 1977;20(10):762–72.
9. Burrows M, Wheeler DJ. A block-sorting lossless data compression algorithm. Syst Res Center. 1994.
10. Chang WL, Lampe J. Theoretical and empirical comparisons of approximate string matching algorithms. In: Apostolico A, Crocchemore M, Galil Z, Manber U (eds) Combinatorial pattern matching, lecture notes in computer science. Springer, Berlin; 1992. pp. 175–84.
11. Chen Y, Wu Y. On the massive string matching problem. In: Proceedings of ICNC-FSKD, IEEE. China, August: Changsha. 2016. p. 350–355.
12. Chen Y, Wu Y. Mismatching trees and BWT arrays: a new way for string matching with k-mismatches. In: Proceedings of ICDE'17, IEEE, San Diego, USA. 2017. pp. 399–410.
13. Chen Y, Wu Y. On the String matching with k mismatches. Theor Comput Sci. 2018;726:5–29.
14. Chen Y, Wu Y. Searching BWT against pattern matching machine to find multiple string matches. In: Proceedings of 9th international conference on cyber-enabled distributed computing and knowledge discovery, IEEE. 2017. pp. 167–76.
15. Chen Y, Nguyen HH. On the string matching with k differences in DNA databases. PVLDB. 2021;14(6):903–15.
16. Commentz-Walter B. A string matching algorithm fast on the average. In: Proceedings of 6th colloquium on automata, languages and programming. 1979; pp. 118–32.
17. Crochemore M, et al. Fast practical multi-pattern matching. Inf Process Lett. 1999;71:107–33.
18. Cunningham F, et al. Nucleic Acids Res. 2015;43(Database issue):D662–9.
19. Dandass YS, Burgess SC, Lawrence M, Bridges SM. Accelerating string set matching in FPGA hardware for bioinformatics research. BMC Bioinform. 2008;9:197.
20. Galil Z. On improving the worst case running time of the Boyer-Moore string searching algorithm. Commun ACM. 1977;22(9):505–8.
21. Galil Z, Giancarlo R. Improved string matching with k mismatches. ACM SIGACT News. 1986;17(4):52b–54.
22. Grossi R, Gupta A, Vitter J. High-order entropy-compressed text indexes. In: Proceedings of 14th SODA, 2003, pp. 841–50.
23. Harrison MC. Implementation of the substring test by hashing. Commun ACM. 1971;14(12):777–9.
24. Hon W, et al. A space and time efficient algorithm for constructing compressed suffix arrays. Algorithmica. 2007;48:23–36.
25. https://www.sciencedirect.com/topics/computer-science/network-intrusion-detection-system.
26. https://cisomag.com/what-does-a-digital-forensics-investigator-do-in-an-investigation/.
27. Huang N, Hung H, Lai S, et al. A GPU-based multiple-pattern matching algorithm for network intrusion detection systems. In: The 22nd international conference on advanced information networking and applications, 2008.
28. Jacob N, Brodley C. Offloading IDS computation to the GPU. In: The 22nd annual computer security applications conference, 2006.
29. Jacobson G. Space-efficient static trees and graphs. In: 30th IEEE symposium on foundations of computer science; 1989.
30. Karp RL, Rabin MO. Efficient randomized pattern-matching algorithms. IBM J Res Dev. 2010;11(5):473–83. https://doi.org/10.1093/bib/bbq015.
31. Kim JY, Yaylor JS. Fast multiple keyword searching. In: Proceedings of third annual symposium on combinatorial pattern matching. Springer; 1992. pp. 41–51.
32. Knuth DE. The art of computer programming, vol. 3. Massachusetts: Addison-Wesley Publish Com; 1975.

33. Knuth DE, Morris JH, Pratt VR. Fast pattern matching in strings. SIAM J Comput. 1977;6(2):323–50.
34. Ko P, Aluru S. Space efficient linear time construction of suffix arrays. J Discrete Algor. 2005;3:143–56.
35. Kim JY, Yaylor JS. Introduction to the Burrows-Wheeler transform. 2014. http://www.youtube.com/watch?v=4n7NPk5lwbI.
36. Landau GM, Vishikin U. Efficient string matching with k mismatches. Theor Comput Sci. 1988;37(1):63–78.
37. Landau GM, Vishikin U. Fast string matching with k differences. J Comput Syst Sci. 1988;37(1):63–78.
38. Li H, Homer N. A survey of sequence alignment algorithms for next-generation sequencing. Brief Bioinform. 2010;11(5):473–83. https://doi.org/10.1093/bib/bbq015.
39. Li H, et al. Mapping short DNA sequencing reads and calling variants using mapping quality scores. Genome Res. 2008;18:1851–8.
40. Li H, wgsim: a small tool for simulating sequence reads from a reference genome. 1994. https://github.com/lh3/wgsim/.
41. Li H, Durbin R. Fast and accurate short read alignment with Burrows-Wheeler transform. Bioinformatics. 2009;25(14):1754–60.
42. Li H, Durbin R. Fast and accurate long-read alignment with Burrows-Wheeler transform. Bioinformatics. 2010;26(5):589–95.
43. Lin T-S, Lu C-Y, Kuo S-Y. Quantum switching and quantum string matching. In: 10th IEEE international conference on nanotechnology. 2010. https://doi.org/10.1109/NANO.2010.5697866.
44. Manber U, Myers EW. Suffix arrays: a new method for on-line string searches. In: Proceedings of the 1st annual ACM-SIAM symposium on discrete algorithms. SIAM, Philadelphia; 1990. pp. 319–27.
45. Manber U, Baeza-Yates RA. An algorithm for string matching with a sequence of don't cares. Inf Process Lett. 1991:133–36.
46. Marziale L, Richard III G, Roussev V. Massive threading: using GPUs to increase the performance of digit forensics tools. Science Direct; 2007.
47. McCreight EM. A space economical suffix tree construction algorithm. J ACM. 1987;31(2):249–60.
48. Meyer Y. Wavelets and operators. Cambridge University Press, Cambridge. ISBN 0-521-42000-8. 1992.
49. Michailidis P. On-line string matching algorithms: survey and experimental results. Int J Comput Math. 2001;76:411–4.
50. Ni J, Lin C, Chen Z, Ungsunan P. A fast multi-pattern matching algorithm for deep packet inspection on a network processor. In: Proceedings of international conference on parallel processing (ICPP2007) IEEE; 2007.
51. Nong G, Zhang S, Chan WH. Two efficient algorithms for linear time suffix array construction. IEEE Trans Comput. 2011;60(10):1471–84.
52. Petri M, Culpepper JS. Efficient indexing algorithms for approximate pattern matching in text. In: ADCS'12. Dunedin, NZ: Otago. 2012.
53. Ktistakis R, Fournier-Viger P, Puglisi3 SJ, Raman R. Succinct BWT-based sequence prediction, DEXA2019, Otago, Bratislava, Slovakia, 2019. pp. 91–101.
54. Raman R, Raman V, Satti SR. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Trans Algor. 2007;3(4).
55. Salmela L, Tarhio J, Kytöjoki J. Multi-pattern string matching with q-grams. ACM J Exp Algor 11. 2006.
56. Scarpazza D, Villa O, Petrini F. Peak-performance DFA-based string matching on the cell processor. In: Third IEEE/ACM international workshop on system management techniques: processes, and services, within IEEE/ACM Intl. parallel and distributed processing symposium; 2007.
57. Scarpazza D, Villa O, Petrini F. Accelerating real-time string searching with multicore processors. IEEE Computer Society, 2008.
58. Salsona M, Lecroqa T, Leonarda M, Mouchard L. A four-stage algorithm for updating a Burrows-Wheeler transform. Theor Comput Sci. 2019;410(43):4350–9.
59. Schatz M. Cloudburst: highly sensitive read mapping with mapreduce. Bioinformatics. 2009;25:1363–9.
60. Smith R, Goyal N, Ormont J. et al. Evaluating GPUs for network packet signature matching. In: International symposium on performance analysis of systems and software, 2009.
61. Ukkonen E. Algorithms for approximate string matching. Inf Control. 1985;64:100–18.
62. Weiner P. Linear pattern matching algorithm. In: Proceedings of 14th IEEE symposium on switching and automata theory, 1973, pp. 1–11.
63. Wu S, Manber U. A fast algorithm for multi-pattern searching, Technical Report TR-94-17. Department of Computer Science: Chung-Cheng University; 1994.
64. Zha X, Sahni S. Fast in-place file carving for digital forensics, e-Forensics. LNICST: Springer; 2010.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.