Metadata of the chapter that will be visualized in SpringerLink

Book Title	Big Data in Engineering Applications	
Series Title		
Chapter Title	BWT: An Index Structure to Speed-Up Both Exact and Inexact String Matching	
Copyright Year	2019	
Copyright HolderName	Springer Nature Singapore Pte Ltd.	

Corresponding Author	Family Name	**Chen**
	Particle	
	Given Name	**Yangjun**
	Prefix	
	Suffix	
	Role	
	Division	Department of Applied Computer Science
	Organization	University of Winnipeg
	Address	Winnipeg, Canada
	Email	y.chen@uwinnipeg.ca
Author	Family Name	**Wu**
	Particle	
	Given Name	**Yujia**
	Prefix	
	Suffix	
	Role	
	Division	Department of Applied Computer Science
	Organization	University of Winnipeg
	Address	Winnipeg, Canada
	Email	wyj1128@yahoo.com

Abstract

The BWT transformation of a string is originally proposed for string compression, but can also be used to speed up string matchings. In this chapter, we address two issues around this mechanism: (1) how to use BWT to improve the running time of a multiple pattern string matching process; and (2) how to integrate mismatching information into a search of BWT arrays to expedite string matching with k mismatches. For the first problem, we will first construct the BWT array of a target string s, denoted as $BWT(s)$; and then establish a *trie* structure over a set of pattern strings $R = \{r_1, \ldots, r_l\}$, denoted as $T(\boldsymbol{R})$. By scanning $BWT(s)$ against $T(\boldsymbol{R})$, the time spent for finding occurrences of r_i's can be significantly reduced. For the second problem, for a given pattern string r, we will precompute its mismatching information (over some different substrings of it, denoted as $M(r)$) and construct a tree structure, called a *mismatching tree*, to record the mismatches between r and s during a search of $BWT(s)$ against r. In this process, the mismatching tree can be effectively utilized to do some kind of useful mismatching information derivation based on $M(r)$ to avoid any possible redundancy. Extensive experiments have been done to compare our methods with the existing ones, which show that for both the problems described above our methods are promising.

Author Proof

BWT: An Index Structure to Speed-Up Both Exact and Inexact String Matching

Yangjun Chen and Yujia Wu

UNCORRECTED PROOF

Abstract The BWT transformation of a string is originally proposed for string compression, but can also be used to speed up string matchings. In this chapter, we address two issues around this mechanism: (1) how to use BWT to improve the running time of a multiple pattern string matching process; and (2) how to integrate mismatching information into a search of BWT arrays to expedite string matching with k mismatches. For the first problem, we will first construct the BWT array of a target string s, denoted as $BWT(s)$; and then establish a *trie* structure over a set of pattern strings $\boldsymbol{R} = \{r_1, \ldots, r_l\}$, denoted as $T(\boldsymbol{R})$. By scanning $BWT(s)$ against $T(\boldsymbol{R})$, the time spent for finding occurrences of r_i's can be significantly reduced. For the second problem, for a given pattern string r, we will precompute its mismatching information (over some different substrings of it, denoted as $M(r)$) and construct a tree structure, called a *mismatching tree*, to record the mismatches between r and s during a search of $BWT(s)$ against r. In this process, the mismatching tree can be effectively utilized to do some kind of useful mismatching information derivation based on $M(r)$ to avoid any possible redundancy. Extensive experiments have been done to compare our methods with the existing ones, which show that for both the problems described above our methods are promising.

1 Introduction

The recent development of next-generation sequencing has changed the way we carry out the molecular biology and genomic studies. It has allowed us to sequence a *DNA* (Deoxyribonucleic acid) sequence at a significantly increased base coverage, as well as at a much faster rate. This requires us considering all the string patterns as

Y. Chen (✉) · Y. Wu
Department of Applied Computer Science, University of Winnipeg, Winnipeg, Canada
e-mail: y.chen@uwinnipeg.ca

Y. Wu
e-mail: wyj1128@yahoo.com

© Springer Nature Singapore Pte Ltd. 2019

S. S. Roy et al. (eds.), *Big Data in Engineering Applications*,
Studies in Big Data 2017, https://doi.org/10.1007/978-981-10-8476-8_12

Layout: T1 Standard STIX | Book ID: 439206_1_En | Book ISBN: 978-981-10-8475-1
Chapter No.: 12 | Date: 17-3-2018 Time: 8:46 pm | Page: 2/44

Author Proof

a whole, rather than separately check them one by one. Two kinds of string matching need to be handled: *exact matching* and *inexact matching*. By the exact matching, we will find all the occurrences of a pattern string r in a target string s, but by the inexact matching we allow each occurrence having up to k positions different between r and s. The inexact matching is important due to the polymorphisms or mutations among individuals or even sequencing errors, the pattern may disagree in some positions at an occurrence of r in the target s.

The string matching is always an interesting and important research topic in computer science and computer engineering. In the past several decades, a bunch of efficient strategies have been proposed to find all the occurrences of a pattern in a target very fast, such as those discussed in [1–7]. Roughly speaking, all these methods can be classified as illustrated in Fig. 1.

From Fig. 1, we can see that for the exact matching problem we distinguish between two kinds of strategies: the single-pattern oriented and the multi-pattern oriented methods. By the former, each time only one pattern string will be mapped to a target string, and for this we have both on-line methods such as *Knuth-Morris-Pratt* [6], *Boyer-Moore* [5], and *Apostolico-Giancarlo* [8], and off-line (index-based) methods like *suffix* trees [9, 10], *suffix* arrays [11], and *BWT*-transformation (*Burrows-Wheeler* Transformation) [12–14]. However, by the latter, we have only on-line strategies, such as the Aho-Corasick's algorithm proposed in 1975 [15], and its improved versions [16–18], by which an automaton is established over all the patterns and will be searched against a target in one scan.

For the inexact matching problem, we have string matching with k mismatches, k errors, as well as *don't-care* symbols. By the string matching with k mismatches, we will find all the occurrences of a pattern string r in a target string s with each occurrence having up to k positions different between r and s. Different methods for this problem have been proposed, such as the on-line strategies discussed in [1, 3, 19, 20], and the index-based method proposed in [21]. The methods of [3, 19, 20] have the worst-case time complexities bounded by $O(kn + m\log m)$, where $n = |s|$ and $m = |r|$. By these three methods, the *mismatch information* among substrings of r is used to speed up the working process. The method discussed in [1] is with a slightly better time complexity $O\left(n\sqrt{k}\log k\right)$. By this method, the *periodicity*

UNCORRECTED PROOF

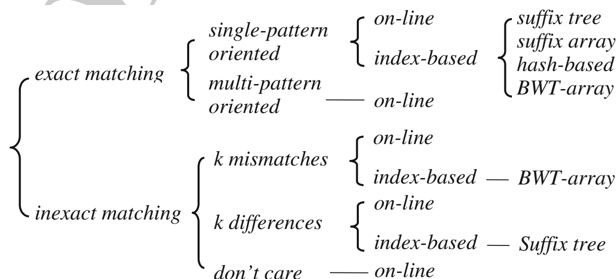

Fig. 1 Classification of methods for string matching

Author Proof

within r is utilized. In [21], a target string s is transformed to a BWT-array (denoted as $BWT(s)$) as an index [12]. In comparison with *suffix trees* [9], $BWT(s)$ uses much less space [12]. However, its time complexity is bounded by $O(mn' + n)$, where n' is the number of leaf nodes of a tree produced during the search of $BWT(s)$. This time requirement can be much worse than the best on-line algorithm for large patterns. The reason for this is that by this method neither mismatch information nor periodicity within r is employed.

The string matching with k errors is quite different from the string matching with k mismatches, by which we will find all the occurrences of a pattern string r in a target string s such that the edit distance between each occurrence and s is $\leq k$. To do such a task, the *dynamic programming paradigm* has to be employed [22], possibly with suffix trees being used as indexes [23, 24]. By the string matching with *don't-care* symbols, we allow *don't-care* to appear in r, in s, or in both of them [25, 26].

In this chapter, we address two issues. One is to construct indexes for the multiple pattern string matching, and another one is to construct indexes for the string matching with k mismatches. As discussed above, up to now no effective indexes have been established for these two problems. Specifically, for the first problem, we will

- Construct a trie T over all the pattern sequences, and check T against the BWT-array of s's reverse, denoted as $BWT(\bar{s})$ created as an index for s. This enables us to avoid repeated search of the same part of different pattern strings.
- Change a single-character checking to a multiple-character checking. (That is, each time a set of characters respectively from more than one pattern strings will be checked against a BWT-array in one scan, instead of checking them separately one by one in multiple scans.)

Our experiment shows that it can be more than 40% faster than single-pattern oriented methods when multi-million pattern strings are checked.

For the second problem, two techniques are introduced, which will be combined with a BWT-array scanning as described below:

- An efficient method to calculate the mismatches between $r[i \ldots m]$ and $r[j \ldots m]$ ($i, j \in \{1, \ldots, m\}$, $i \neq j$), where $r[i \ldots m]$ represents a substring of r starting from position i and ending at position m. The mismatches between them is stored in an array R such that if $R[p] = q$ then we have $r[i + q - 1] \neq r[j + q - 1]$ and it is their pth mismatch.
- A new tree (forest) structure D to store the mismatches between r and different segments of s. In D, each node v stores an integer i, indicating that there are some positions $i_1, i_2, \ldots, i_l$ in s such that $s[i_q + i - 1] \neq r[i]$ ($q = 1, \ldots, l$). If v is at the pth level of D, it also shows that it is the pth mismatch between each $s[i_q \ldots i_q + i - 1]$ and r.

UNCORRECTED PROOF

Layout: T1 Standard STIX Book ID: 439206_1_En Book ISBN: 978-981-10-8475-1
Chapter No.: 12 Date: 17-3-2018 Time: 8:46 pm Page: 4/44

Author Proof

By using these two techniques, the time complexity for solving the string matching with k mismatches can be reduced to $O(kn' + n)$. Our experiment shows that $n' \ll n$.

2 Related Work

The string matching problem has always been one of the main focuses in computer science. A huge number of algorithms have been proposed, which can be generally divided into two categories: *exact matching* and *inexact matching*. By the former, all the occurrences of a pattern string r in a target string s will be searched. By the latter, a best alignment between r and s (i.e., a correspondence with the highest score) is searched in terms of a given distance function or a score matrix, which is established to indicate the relevance between different symbols.

- *Exact matching*

The first interesting algorithm for this problem is the famous *Knuth-Morris-Pratt*'s algorithm [6], 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 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 $r[1 \ldots j] = r[i - j + 1 \ldots i]$. 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 [5] works a little bit better than the *Knuth-Morris-Pratt*'s. In addition to the next-table, a skip-table *skip* (also for r) is kept, in which each entry $skip[w]$ is a smallest integer j such that $r[m - j] = w$. (Here, we notice that the entries in *skip* are indexed by characters w in the alphabet Σ.) 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. These two methods have sparked a series of subsequent research on this problem [15, 24, 27, 28]. Especially, the idea of the ′shift information′ has also been adopted by Aho and Corasick [15] for the *multiple pattern* matching, by which s is searched for an occurrence of any one of a set of l patterns: $\{r_1, r_2, \ldots, r_l\}$. Their algorithm needs only $O\left(\sum_{i=1}^{l} |r_i| + n\right)$ time. This method has been slightly improved in different ways. In [16], Commentz-Walter combines the Boyer-Moore's technique into the Aho-Corasick's algorithm. In [17], Wu and Mamber 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 [29], 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

UNCORRECTED PROOF

Layout: T1 Standard STIX | Book ID: 439206_1_En | Book ISBN: 978-981-10-8475-1
Chapter No.: 12 | Date: 17-3-2018 Time: 8:46 pm | Page: 5/44

Author Proof

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 and Corasick's algorithm. The main problem of this method is the *false positive* and a very time-consuming verification process is needed. In [18], 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 situations where a fixed string s is to be searched repeatedly, it is worthwhile constructing an index over s, such as suffix trees [9, 10], suffix arrays [11], and more recently the BWT-transformation [12, 14, 21, 30]. A suffix tree is in fact a *trie* structure [31] over all the suffixes of s; and by using the Weiner's algorithm [10] 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, the smaller Σ is, the less space will be occupied by the corresponding indexes. According to a survey done by Li and Homer [32] 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. The experiments reported in [19] also confirm this distinction. 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 needs only 390 Mb–1 Gb for different compression rates of auxiliary arrays, completely handleable on PC or laptop machines.

By the hash-table-based algorithms [33], short substrings called 'seeds' 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 $O(mn)$ time. The hash technique has also been extensively used in the DNA sequence research [34–37]. 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.

- *Inexact matching*

By the inexact matching, we will find, for a certain pattern r and an integer k, all the substrings s' of s such that $d(s', r) \leq k$, where d is a distance function. In terms of different distance functions, we distinguish between two kinds of inexact matches: string matching with k mismatches and string matching with k errors. A third kind of inexact matching is that involving Don't Care, or wild-card symbols which match any single symbol, including another Don't Care.

UNCORRECTED PROOF

Author Proof

184 ***k mismatches*** When the distance function is the *Hamming* distance, the problem
185 is known as the string matching with *k mismatches* [1, 20]. By the Hamming
186 distance, the number of differences between *r* and the corresponding substring s' is
187 counted. There are a lot of algorithms proposed for this problem, such as [1, 20, 38–
188 42]. They are all on-line algorithms. Except those discussed in [1, 20], all the other
189 methods have the worst-case time complexity O(*mn*). The method discussed in
190 [20], however, requires only O(*kn* + *m*log*m*) time, by which the mismatch arrays
191 for *r* is precomputed and exploited to speed up the search of *s*. The method dis-
192 cussed in [1] works slightly better, by which the periodicity within *r* is utilized. Its
193 time complexity is bounded by $O\left(n\sqrt{k}\log k\right)$. The algorithm discussed in [21] is
194 index-based, by which *s* is transformed to a BWT-array, used as an index; but its
195 time complexity is bounded by O(*mn*′+ *n*), where *n*′ is the number of leaf nodes of
196 a tree produced during the search of $BWT(\bar{s})$. If *m* is large, it can be worse than all
197 those on-line methods discussed in [1, 20, 40, 41]. Another index-based method is
198 based on a brute-force searching of suffix trees [43]. Its time complexity is bounded
199 by $O\left(m+n+\left(c\log n^k/k!\right)\right)$, where *c* is a very large constant. It can also be worse
200 than an on-line algorithm when *n* is large and *k* is larger than a certain constant.

201 ***k errors*** When the distance function is the *Levenshtein* distance, the problem is
202 known as the string matching with *k errors* [39]. By the Levenshtein distance, we
203 have
204

$$d_{ij}=\min\left\{d_{i-1,j}+w(r_i,\phi),d_{i,j-1}+w\left(\phi,s_j^{'}\right),d_{i-1,j-1}+w\left(r_{i,}s_j^{'}\right)\right\},$$

206
207 where $d_{i,j}$ represents the distance between $r[1 \ldots i]$ and $s'[1 \ldots j]$, r_i (s_j') the *i*th
208 character in *r* (*j*th character in s'), ϕ an empty character, and $w(r_i, s_j')$ the cost to
209 transform r_i into s_j'.

210 Also, many algorithms have been proposed for this problem [3, 22–24]. They are
211 all some kinds of variants of the *dynamic programming* paradigm [22] with the
212 worst-case time complexity bounded by O(*mn*). However, by the algorithm dis-
213 cussed in [23], the expected time can reach O(*kn*).

214 ***don't care*** As a different kind of inexact matching, the string matching with
215 *Don't-Cares* (or *wild-cards*) has been a third active research topic for decades, by
216 which we may have wild-cards in *r*, in *s*, or in both of them. Due to the wild
217 character's property that it can matches any character, the ′match′ relation is no
218 longer transitive, which precludes straightforward adaption of the shift information
219 used by *Knuth-Morris-Pratt* and *Boyer-Moore*. Therefore, all the methods proposed
220 to solve this problem seem not so skillful and need a quadratic time [26]. Using a
221 suffix array as the index, however, the searching time can be reduced to O(log*n*) for
222 some patterns, which contain only a sequence of consecutive Don't Cares [25].

UNCORRECTED PROOF

Layout: T1 Standard STIX | Book ID: 439206_1_En | Book ISBN: 978-981-10-8475-1
Chapter No.: 12 | Date: 17-3-2018 Time: 8:46 pm | Page: 7/44

Author Proof

3 BWT Transformation

In this section, we give a brief description of the BWT transformation to provide a discussion background.

3.1 BWT and String Searching

We use s to denote a string that we would like to transform. Assume that s terminates with a special character \$, which does not appear elsewhere in s and is alphabetically prior to all other characters. In the case of DNA sequences, we have $\$ < a < c < g < t$. As an example, consider $s = acagaca\$$. We can rotate s consecutively to create eight different strings, and put them in a matrix as illustrated in Fig. 2a.

In Fig. 2a, for ease of explanation, the position of a character in s is represented by its subscript. (That is, we rewrite s as $a_1c_1a_2g_1a_3c_2a_4\$$.) For example, a_2 representing 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.

Now we sort the rows of the matrix alphabetically, and get another matrix, as demonstrated in Fig. 2b, which is called the *Burrow-Wheeler Matrix* [12, 13, 30] 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 $s = acagaca\$$, we have $BWT(s) = acg\$caaa$. The first column is referred to as F.

When ranking the elements x in both F and L in such a way that if x 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 example, in F the rank of a_4, denoted as $rk_F(a_4)$, is 1 (showing that a_4 is the first appearance of a in F). Its rank in L, $rk_L(a_4)$ is also 1. We can check all the other elements and find that this property, called the

(a)	(b)	(c) rk_F	F	L	rk_L
$a_1\ c_1\ a_2\ g_1\ a_3\ c_2\ a_4\ \$$	$\$\ a_1\ c_1\ a_2\ g_1\ a_3\ c_2\ a_4$	–	\$	a_4	1
$c_1\ a_2\ g_1\ a_3\ c_2\ a_4\ \$\ a_1$	$a_4\ \$\ a_1\ c_1\ a_2\ g_1\ a_3\ c_2$	1	a_4	c_2	1
$a_2\ g_1\ a_3\ c_2\ a_4\ \$\ a_1\ c_1$	$a_3\ c_2\ a_4\ \$\ a_1\ c_1\ a_2\ g_1$	2	a_3	g_1	1
$g_1\ a_3\ c_2\ a_4\ \$\ a_1\ c_1\ a_2$	$a_1\ c_1\ a_2\ g_1\ a_3\ c_2\ a_4\ \$$	3	a_1	\$	–
$a_3\ c_2\ a_4\ \$\ a_1\ c_1\ a_2\ g_1$	$a_2\ g_1\ a_3\ c_2\ a_4\ \$\ a_1\ c_1$	4	a_2	c_1	2
$c_2\ a_4\ \$\ a_1\ c_1\ a_2\ g_1\ a_3$	$c_2\ a_4\ \$\ a_1\ c_1\ a_2\ g_1\ a_3$	1	c_2	a_3	2
$a_4\ \$\ a_1\ c_1\ a_2\ g_1\ a_3\ c_2$	$c_1\ a_2\ g_1\ a_3\ c_2\ a_4\ \$\ a_1$	2	c_1	a_1	3
$\$\ a_1\ c_1\ a_2\ g_1\ a_3\ c_2\ a_4$	$g_1\ a_3\ c_2\ a_4\ \$\ a_1\ c_1\ a_2$	1	g_1	a_2	4

Fig. 2 Rotation of a string

UNCORRECTED PROOF

Author Proof

rank correspondence, holds for all the elements. That is, for any element a in s, we always have

$$rk_F(a) = rk_L(a) \tag{1}$$

According to this property, a string searching can be very efficiently conducted. To see this, let us consider a pattern string $r = aca$ and try to find all its occurrences in $s = acagaca\$$.

First, we notice that we can store F as $|\Sigma| + 1$ intervals, such as $F_\$ = F[1 \ldots 1]$, $F_A = F[2 \ldots 5]$, $F_C = F[6 \ldots 7]$, $F_G = F[8 \ldots 8]$, and $F_T = \Phi$ for the above example (see Fig. 1c) We can also represent a segment within an F_x with $x \in \Sigma$ as a pair of the form $<x, [\alpha, \beta]>$, where $\alpha \le \beta$ are two ranks of x. Thus, we have $F_A = F[2 \ldots 5] = <a, [1, 4]>$, $F_C = F[6 \ldots 7] = <c, [1, 2]>$, and $F_G = F[8 \ldots 8] = <g, [1, 1]>$. In addition, we can use L_v to represent a range in L corresponding to a pair v. For example, in Fig. 1c, $L_{<a, [1, 4]>} = L[2 \ldots 5]$, $L_{<c, [1, 2]>} = L[6 \ldots 7]$. $L_{<a, [2, 3]>} = L[3 \ldots 4]$, and so on.

We will also use a procedure *search*(z, v) to search L_v to find the first and the last rank of z (denoted as α' and β', respectively) within L_v, and return $<z, [\alpha', \beta']>$ as the result:

$$search(z, v) = \begin{cases} <z, [\alpha', \beta']>, & \text{if } z \text{ appears in } L_v; \\ \phi, & \text{otherwise.} \end{cases} \tag{2}$$

Then, 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 $BWT(s)$, as shown below.

Step 1: Check $r[3] = a$ in the pattern string r, and then figure out $F_A = F[2 \ldots 5] = <a, [1, 4]>$.

Step 2: Check $r[2] = c$. Call *search*(c, $L_{<a, [1, 4]>}$). It will search $L_{<a, [1, 4]>} = L[2 \ldots 5]$ to find a range bounded by the first and last rank of c. Concretely, we will find $rk_L(c_2) = 1$ and $rk_L(c_1) = 2$. So, *search*(c, $L_{<a, [1, 4]>}$) returns $<c, [1, 2]>$. It is $F[6 \ldots 7]$.

Step 3: Check $r[3] = a$. Call *search*(a, $L_{<c, [1, 2]>}$). Notice that $L_{<c, [1, 2]>} = L[6 \ldots 7]$. So, *search*($a$, $L_{<c, [1, 2]>}$) returns $<a, [2, 3]>$. It is $F[3 \ldots 4]$. Since now we have exhausted all the characters in r and $F[3 \ldots 4]$ contains only two elements, two occurrences of r in s are found. They are a_1 and a_3 in s, respectively.

The above working process can be represented as a sequence of three pairs: $<a, [1, 4]>$, $<c, [1, 2]>$, $<a, [2, 3]>$. In general, for $\bar{r} = C_1 \ldots C_m$, its search against $BWT(s)$ can always be represented as a sequence:

$$<x_1, [\alpha_1, \beta_1]>, \ldots, <x_m, [\alpha_m, \beta_m]>$$

UNCORRECTED PROOF

Author Proof

(a)

j	F	L	$	A_a	A_c	A_g	A_t
1	$	a_4	0	1	0	0	0
2	a_4	c_2	0	1	1	0	0
3	a_3	g_1	0	1	1	1	0
4	a_1	$	1	1	1	1	0
5	a_2	c_1	1	1	2	1	0
6	c_2	a_3	1	2	2	1	0
7	c_1	a_1	1	3	2	1	0
8	g_1	a_2	1	4	2	1	0

(b)

	A_a	A_c	A_g	A_t
0	0	0	0	0
1	1	1	1	0
2	4	2	1	0

For each 4 values in *L*, a *rankAll* value is stored.

Fig. 3 Illustration for *rankAlls*

289 where $<x_1, [\alpha_1, \beta_1]> = F_{x_1}$, and $<x_i, [\alpha_i, \beta_i]> =$ $search\left(x_i, L_{<x_{i-1}, [\alpha_{i-1}, \beta_{i-1}]>}\right)$ for
290 $1 < i \leq m$. We call such a sequence as a *search sequence*. Thus, the time used for
291 this process is bounded by $O\left(\sum_{i=1}^{m} \tau_i\right)$, where τ_i is the time for an execution of
292 $search\left(x_i, L_{<x_{i-1}, [\alpha_{i-1}, \beta_{i-1}]>}\right)$. However, this time complexity can be reduced to
293 O(*m*) by using the so-called *rankAll* method [12], by which |Σ| arrays each for a
294 character $x \in \Sigma$ are arranged such that $A_x[k]$ (the *k*th entry in the array for *x*) is the
295 number of appearances of *x* within *L*[1 … *k*] (i.e., the number of *x*-characters
296 appearing before *L*[*k* + 1].) (See Fig. 3a for illustration.)

297 Now, instead of scanning a certain segment *L*[*i* … *j*] ($i \leq j$) to find a subrange
298 for a certain $x \in \Sigma$, we can simply look up the array for *x* to see whether $A_x[i -$
299 $1] = A_x[j]$. If it is the case, then *x* does not occur in *L*[*i* … *j*]. Otherwise, $[A_x[i -$
300 $1] + 1, A_x[j]]$ should be the range to be found.

301 For instance, to find the subrange for *g* within *L*[6 … 7], we will first check
302 whether $A_g[6 - 1] = A_g[7]$. Since $A_g[6 - 1] = A_g[5] = A_g[7] = 1$, we know that
303 *g* does not appear in *L*[6 … 7]. However, since $A_c[2 - 1] \neq A_c[5]$, we immediately
304 get the subrange for *c* within *L*[2 … 5]: $[A_c[2 - 1] + 1, A_c[5]] = [1, 2]$.

305 The problem of this method is its high space requirement, which can be miti-
306 gated by replacing *x*[] with a compact array A_x for each $x \in \Sigma$, in which, rather than
307 for each *L*[*i*] ($i \in \{1, …, n\}$), only for some entries in *L* the number of their
308 appearances will be stored. For example, we can divide *L* into a set of buckets of the
309 same size and only for each bucket a value will be stored in A_x. Obviously, doing
310 so, more search will be required. In practice, the size π of a bucket (referred to as a
311 *compact factor*) can be set to different values. For example, we can set $\pi = 4$,
312 indicating that for each four contiguous elements in *L* a group of |Σ| integers (each
313 in an A_x) will be stored. That is, we will not store all the values in Fig. 3a, but only
314 store $[4], *a*[4], *c*[4], *g*[4], *t*[4], and $[8], *a*[8], *c*[8], *g*[8], *t*[8] in the corresponding
315 compact arrays, as shown in Fig. 4b. However, each *x*[*j*] for $x \in \Sigma$ can be easily
316 derived from A_α by using the following formulas:

UNCORRECTED PROOF

Author Proof

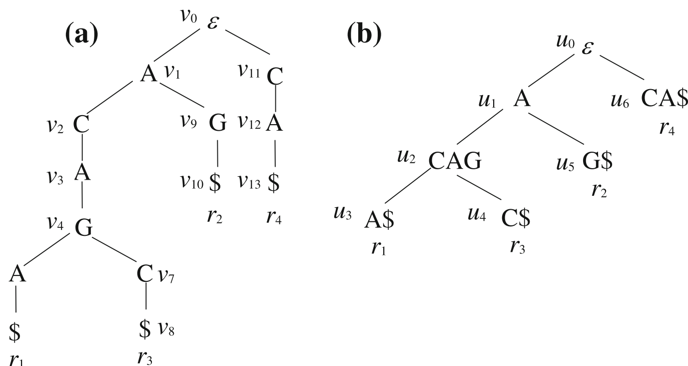

Fig. 4 A trie and its compact version

317
$$x[j] = A_x[i] + \rho, \tag{3}$$
319

320 where $i = \lfloor j/\pi \rfloor$ and ρ is the number of x's appearances within $L[i \cdot \pi + 1 \ldots j]$, and
321
$$x[j] = A_x[i'] + \rho', \tag{4}$$
323

324 where $i' = \lceil j/\pi \rceil$ and ρ′ is the number of α's appearances within $L[j + 1 \ldots i' \cdot \pi]$.
325 Thus, we need two procedures: *sDown*(L, j, π, x) and *sUp*(L, j, π, x) to find ρ and
326 ρ′, respectively. In terms of whether $j - i \cdot \pi \leq i' \cdot \pi - j$, we will call *sDown*($L$, j, π,
327 x) or *sUp*(L, j, π, x) so that fewer entries in L will be scanned to find $x[j]$.
328 We notice that the column for $ needn't be stored since it will never be searched.
329 We can also create *rankAll*s only for part of the elements to reduce the space
330 overhead, but at cost of some more searches. See Fig. 3b for illustration.

331 *3.2 Construction of BWT Arrays*

332 A BWT-array can be constructed in terms of a relationship to the *suffix arrays* [12,
333 13, 30].
334 As mentioned above, a string $s = a_1 \ldots. a_n$ is always ended with $ (i.e., $a_i \in \Sigma$
335 for $i = 1, \ldots, n - 1$, and a_n = $). Let $s[i] = a_i$ (i = 1, 2, …, n) be the ith character
336 of s, $s[i \ldots j] = a_i \ldots a_j$ a substring and $s[i \ldots n]$ a suffix of s. Suffix array H of s is a
337 permutation of the integers 1, …, n such that $H[i]$ is the start position of the ith
338 smallest suffix. The relationship between H and the BWT-array L can be determined
339 by the following formulas:

UNCORRECTED PROOF

Layout: T1 Standard STIX Book ID: 439206_1_En Book ISBN: 978-981-10-8475-1
Chapter No.: 12 Date: 17-3-2018 Time: 8:46 pm Page: 11/44

Author Proof

$$\begin{cases} L[i] = \$, & \text{if } H[i] = 0; \\ L[i] = s[H[i] - 1], & \text{otherwise.} \end{cases} \quad (5)$$

Since a suffix array can be generated in O(n) time [44], L can then be created in a linear time. However, most algorithms for constructing suffix arrays 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. [44] 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. We use this for our purpose.

4 Multiple Pattern Matching

In this section, we present our algorithm to search a bunch of pattern strings against a target s. Its main idea is to organize all the reads into a trie T and search T against L to avoid any possible redundancy. First, we present the concept of tries in Sect. 4.1. Then, in Sect. 4.2, we discuss our basic algorithm for the task. We improve this algorithm in Sect. 4.3.

4.1 Tries over Pattern Strings

Let $\boldsymbol{D} = \{s_1, \ldots, s_n\}$ be a DNA database, where each s_i (i =1, …, n) is a genome, a very long string $\in \Sigma^*$ ($\Sigma = \{A, T, C, G\}$). Let $\boldsymbol{R} = \{r_1, \ldots, r_m\}$ be a set of *reads* with each r_j being a short string $\in \Sigma^*$. The problem is to find, for every r_j's (j =1, …, m), all their occurrences in an s_i (i =1, …, n) in $\boldsymbol{D}$.

A simple way to do this is to check each r_j against s_i one by one, for which different string searching methods can be used, such as suffix trees [9, 10], BW-transformation [12], and so on. Each of them needs only a linear time (in the size of s_i) to find all occurrences of r_j in s_i. However, in the case of very large m, which is typical in the new genomic research, one-by-one search of reads against an s_i is no more acceptable in practice and some efforts should be spent on reducing the running time caused by huge m.

Our general idea is to organize all r_j's into a trie structure T and search T against s_i with the BW-transformation being used to check the string matching. For this purpose, we will first attach \$ to the end of each s_i (i = 1, …, n) and construct $BWT(s_i)$. Then, attach \$ to the end of each r_j (j = 1, …, m) to construct T = *trie* ($\boldsymbol{R}$) over $\boldsymbol{R}$ as 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 $|\Sigma| = 5$ (possibly empty) subsets $\boldsymbol{R}_1, \boldsymbol{R}_2, \ldots, \boldsymbol{R}_5$ so that each $\boldsymbol{R}_i$ ($i \in \{1, \ldots, 5\}$) contains all those sequences with the same first character $\alpha_i \in \{A, T, C, G\} \cup \{\$\}$. The tries: *trie*($\boldsymbol{R}_1$), *trie*($\boldsymbol{R}_2$), …, *trie*($\boldsymbol{R}_5$) are constructed in the

UNCORRECTED PROOF

Author Proof

376 same way except that at the *k*th step, the splitting of sets is based on the *k*th
377 characters in the sequences. They are then connected from their respective roots to a
378 single node to create *trie*(***R***).

379 **Example 4.1** As an example, consider a set of four reads:

380 r_1: ACAGA
381 r_2: AG
382 r_3: ACAGC
383 r_4: CA
384

385 For these reads, a trie can be constructed as shown in Fig. 4a. In this trie, v_0 is a
386 virtual root, labeled with an *empty* character ε while any other node v is labeled with
387 a *real* character, denoted as $l(v)$. Therefore, all the characters on a path from the root
388 to a leaf spell a read. For instance, the path from v_0 to v_8 corresponds to the third
389 read r_3 = ACAGC\$. Note that each leaf node v is labelled with \$ and associated
390 with a *read identifier*, denoted as $\gamma(v)$.
391 The size of a trie can be significantly reduced by replacing each branchless path
392 segment with a single edge. By a branchless path we mean a path P such that each
393 node on P, except the starting and ending nodes, has only one incoming and one
394 outgoing edge. For example, the trie shown in Fig. 4a can be compacted to a
395 reduced one as shown in Fig. 4b.

396 4.2 Integrating BWT Search with Trie Search

397 It is easy to see that exploring a path in a trie T over a set of reads ***R*** corresponds to
398 scanning a read $r \in \boldsymbol{R}$. If we explore, at the same time, the L array established over
399 a *reversed* genome sequence $\bar{s}$, we will find all the occurrences of r (without \$

CORRECTED PROOF

(a)

$\$\ A_4 C_2 A_3 G_1 A_2 C_1 A_1$
$A_1 \$\ A_4 C_2 A_3 G_1 A_2 C_1$
$A_2 C_1 A_1 \$\ A_4 C_2 A_3 G_1$
$A_4 C_2 A_3 G_1 A_2 C_1 A_1 \$$
$A_3 G_1 A_2 C_1 A_1 \$\ A_4 C_2$
$C_1 A_1 \$\ A_4 C_2 A_3 G_1 A_2$
$C_2 A_3 G_1 A_2 C_1 A_1 \$\ A_4$
$G_1 A_2 C_1 A_1 \$\ A_4 C_2 A_3$

(b)

j	F	L
1	\$	A_4
2	A_4	C_2
3	A_3	G_1
4	A_1	\$
5	A_2	C_1
6	C_2	A_3
7	C_1	A_1
8	G_1	A_2

(c)

S:

$\langle v_0, 1, 8\rangle$

Fig. 5 Illustration for Step 1

Layout: T1 Standard STIX | Book ID: 439206_1_En | Book ISBN: 978-981-10-8475-1
Chapter No.: 12 | Date: 17-3-2018 Time: 8:46 pm | Page: 13/44

Author Proof

involved) in s. This idea leads to the following algorithm, which is in essence a depth-first search of T by using a stack S to control the process. However, each entry in S is a triplet $<v, a, b>$ with v being a node in T and $a \leq b$, used to indicate a subsegment in $F_{l(v)}[a \ldots b]$. For example, when searching the trie shown in Fig. 5a against the L array shown in Fig. 2a, we may have an entry like $<v_1, 1, 4>$ in S to represent a subsegment $F_A[1 \ldots 4]$ (the first to the fourth entry in F_A) since $l(v_1) = 'A'$. In addition, for technical convenience, we use F_ε to represent the whole F. Then, $F_\varepsilon[a \ldots b]$ represents the segment from the ath to the bth entry in F.

In the algorithm, we first push $<root(T), 1, |s|>$ into stack S (lines 1–2). Then, we go into the main **while-loop** (lines 3–16), in which we will first pop out the top element from S, stored as a triplet $<v, a, b>$ (line 4). Then, for each child v_i of v, we will check whether it is a leaf node. If it is the case, a quadruple $<\gamma(v_i), l(v), a, b>$ will be added to the result $\Re$ (see line 7), which records all the occurrences of a read represented by $\gamma(v_i)$ in s. (In practice, we store compressed suffix arrays [11, 12] and use formulas (1) and (5) to calculate positions of reads in s.) Otherwise, we will determine a segment in L by calculating α' and β' (see lines 8–9). Then, we will use $sDown(L, \alpha' - 1, \pi, x)$ or $sUp(L, \alpha' - 1, \pi, x)$ to find $x[\alpha' - 1]$ as discussed in the previous section. (See line 10.) Next, we will find $x[\beta']$ in a similar way. (See line 11.) If $x[\beta'] > x[\alpha' - 1]$, there are some occurrences of x in $L[\alpha' \ldots \beta']$ and we will push $<v_i, x[\alpha' - 1] + 1, x[\beta']>$) into S, where $x[\alpha' - 1] + 1$ and $x[\beta']$ are the first and last rank of x's appearances within $L[x' \ldots y']$, respectively. (See lines 12–13.) If $x[\beta'] = x[\alpha' - 1]$, x does not occur in $L[\alpha' \ldots \beta']$ at all and nothing will be done in this case. The following example helps for illustration.

ALGORITHM *readSearch*(T, LF, π)

```
begin
1.  v ← root(T);  ℜ ← Φ;
2.  push(S, <v, 1, |s|>);
3.  while S is not empty do {
4.    <v, a, b> ← pop(S);
5.    let v1, …, vk be the children of v;
6.    for i = k downto 1 do {
7.      if vi is a leaf then ℜ ← ℜ ∪ {<γ(vi), l(v), a, b>};
8.      else{assume that Fl(v) = <l(v); α, β>;
9.            α′ ← α + a - 1; β′ ← α + b - 1; x ← l(vi);
10.           find x[α′ - 1] by sDown(L, α′-1, π, x) or sUp(L, α′-1, π, x);
11.           find x[β′] by sDown(L, β′, π, x) or sUp(L, β′, π, x);
          12.        if x[β′] > x[α′ - 1] then
          13. push(S, <vi, x[α′ - 1] + 1, x[β′]>);
14.       }
15.   }
16. }
end
```

UNCORRECTED PROOF

Layout: T1 Standard STIX | Book ID: 439206_1_En | Book ISBN: 978-981-10-8475-1
Chapter No.: 12 | Date: 17-3-2018 Time: 8:46 pm | Page: 14/44

Author Proof

UNCORRECTED PROOF

Example 4.2 Consider all the reads given in Example 4.1 again. The trie T over these reads are shown in Fig. 4a. In order to find all the occurrences of these reads in s = ACAGACA$, we will run *readSearch*() on T and the *LF* of $\bar{s}$ shown in Fig. 5b. (Note that $s = \bar{s}$ for this special string, but the ordering of the subscripts of characters is reversed. In Fig. 5a, we also show the corresponding *BWM* matrix for ease of understanding.)

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

Step 1: push $<v_0, 1, 8>$ into S, as illustrated in Fig. 5c.

Step 2: pop out the top element $<v_0, 1, 8>$ from S. Figure out the two children of v_0: v_1 and v_{11}. First, for v_{11}, we will use A_c to find the first and last appearances of $l(v_{11})$ = 'C' in $L[1 \ldots 8]$ and their respective ranks: 1 and 2. Assume that $\pi = 4$ (i.e., for each 4 consecutive entries in L a *rankAll* value is stored.) Further assume that for each A_x ($x \in \{a, c, g, t\}$) $A_x[0] = 0$. The ranks are calculated as follows.

- To find the rank of the first appearance of 'C' in $L[1 \ldots 8]$, we will first calculate $C[0]$ by using formula (3) or (4) (i.e., by calling $sDown(L, 0, 4, C)$ or $sUp(L, 0, 4, C)$). Recall that whether (4) or (5) is used depends on whether $j - i \cdot \pi \leq i' \cdot \pi - j$, where $i = \lfloor j/\pi \rfloor$ and $i' = \lceil j/\pi \rceil$. For $C[0]$, $j = 0$. Then, $i = i' = 0$ and (4) will be used:

$$C[0] = A_c[\lfloor 0/4 \rfloor] + \rho$$

Since $A_c[\lfloor 0/4 \rfloor] = A_c[0] = 0$ and the search of $L[i \cdot \pi \ldots j] = L[0 \ldots 0]$ finds $\rho = 0$, $C[0]$ is equal to 0.

- To find the rank of the last appearance of 'C' in $L[1 \ldots 8]$, we will calculate $C[8]$ by using (4) for the same reason as above. For $C[8]$, we have $j = 8$ and $i = 2$. So we have

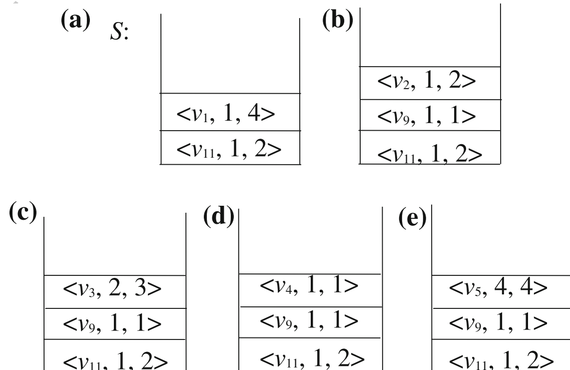

Fig. 6 Illustration for stack changes

Author Proof

$$C[8] = A_c[\lfloor 8/4 \rfloor] + \rho$$

Since $A_c[\lfloor 8/4 \rfloor] = A_c[2] = 2$, and the search of $L[i \cdot \pi \dots j] = L[8 \dots 8]$ finds $\rho = 0$, we have $C[8] = 2$.

So the ranks of the first and the last appearances of 'C' are $C[0] + 1 = 1$, and $C[8] = 2$, respectively. Push <v_{11}, 1, 2> into S.

Next, for v_1, we will do the same work to find the first and last appearances of $l(v_1)$ = 'A' and their respective ranks: 1 and 4; and push <v_1, 1, 4> into S. Now S contains two entries as shown in Fig. 6a after step 2.

Step 3: pop out the top element <v_1, 1, 4> from S. v_1 has two children v_2 and v_9. Again, for v_9 with $l(v_9)$ = 'G', we will use A_g to find the first and last appearances of G in $L[2 \dots 5]$ (corresponding to $F_A[1 \dots 4]$) and their respective ranks: 1 and 1. In the following, we show the whole working process.

- To find the rank of the first appearance of 'G' in $L[2 \dots 5]$, we will first calculate $G[1]$. We have $j = 1, i = \lfloor j/\pi \rfloor = \lfloor 1/4 \rfloor = 0$ and $i' = \lceil 1/4 \rceil = 1$. Since $j - i \cdot \pi = 0 < i' \cdot \pi - j = 3$, formula (4) will be used:

$$G[1] = A_g[\lfloor 1/4 \rfloor] + \rho$$

Since $A_g[\lfloor 0/4 \rfloor] = A_g[0] = 0$ and search of $L[i \cdot \pi \dots j] = L[0 \dots 0]$ finds $\rho = 0$, $G[1]$ is equal to 0.

- To find the rank of the last appearance of 'G' in $L[2 \dots 5]$, we will calculate $G[5]$ by using (4) based on an analysis similar to above. For $G[5]$, we have $j = 5$ and $i = \lfloor j/\pi \rfloor = 1$. So we have

$$G[5] = A_g[5/4] + \rho$$

Since $A_g[\lfloor 5/4 \rfloor] = A_g[1] = 1$, and search of $L[i \cdot \pi \dots j] = L[4 \dots 5]$ finds $\rho = 0$, we have $G[5] = 1$.

We will push <v_9, $G[1] + 1$, $G[5]$> = <v_9, 1, 1> into S.

For v_2 with $l(v_2)$ = 'C', we will find the first and last appearances of C in $L[2 \dots 5]$ and their ranks: 1 and 2. Then, push <v_2, 1, 2> into S. After this step, S will be changed as shown in Fig. 6b.

In the subsequent steps 4, 5, and 6, S will be consecutively changed as shown in Fig. 6c, d, and e, respectively.

In step 7, when we pop the top element <v_5, 4, 4>, we meet a node with a single child v_6 labeled with \$. In this case, we will store <$\gamma(v_6)$, $l(v_5)$, 4, 4> = <r_1, A, 4, 4> in $\Re$ as part of the result (see line 7 in *searchRead*()). From this we can find that $rk_L(A_3) = 4$ (note that the same element in both F and L has the same rank), which shows that in $\bar{s}$ the substring of length $|r_1|$ staring from A_3 is an occurrence of r_1. □

UNCORRECTED PROOF

Author Proof

4.3 Time Complexity and Correctness Proof

In this subsection, we analyze the time complexity of *readSearch*(T, LF, π) and prove its correctness.

4.3.1 Time Complexity

In the main **while**-loop, each node v in T is accessed only once. If the rankAll arrays are fully stored, only a constant time is needed to determine the range for $l(v)$. So the time complexity of the algorithm is bounded by O(|T|). If only the compact arrays (for the rankAll information) are stored, the running time is increased to O(|T| · π), where π is the corresponding compact factor. It is because in this case, for each encountered node in T, $O\left(\frac{1}{2}\ \pi\right)$ entries in L may be checked in the worst case.

4.3.2 Correctness

Proposition 4.1 *Let T be a trie constructed over a collections of reads:* $r_1, \ldots, r_m$, *and LF a BWT-mapping established for a reversed genome* $\bar{s}$. *Let* π *be the compact factor for the allRank arrays, and* $\Re$ *the result of readSearch(T, LF, π). Then, for each* r_j, *if it occurs in s, there is a quadruple* $\{<\gamma(v_i), l(v), a, b>\} \in \Re$ *such that* $\gamma(v_i) = r_j$, *l(v) is equal to the last character of* r_j, *and* $F_{l(v)}[a], F_{l(v)}[a+1], \ldots, F_{l(v)}[b]$ *show all the occurrences of* r_j *in s.*

Proof We prove the proposition by induction on the height h of T.

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

Induction hypothesis. Suppose that when the height of T is h, the proposition holds. We consider the case that the height of T is $h + 1$. Let v_0 be the root with $l(v_0) = \varepsilon$. Let $v_1, \ldots, v_k$ be the children of v_0. Then, $height(T[v_i]) \leq h$ ($i = 1, \ldots, k$), where $T[v_i]$ stands for the subtree rooted at v_i and $height(T[v_i])$ for the height of $T[v_i]$. Let $l(v_i) = x$ and $F_x = <x; a, b>$. Let $v_{i1}, \ldots, v_{il}$ be the children of v_i. Assume that α and β be the ranks of the first and last appearances of x in L. According to the induction hypothesis, searching $T[v_{ij}]$ against $L[a' \ldots b']$, where $a' = a + \alpha - 1$ and $b' = a + \beta - 1$, the algorithm will find all the locations of all those reads with $l(v_i)$ as the first character. This completes the proof. □

UNCORRECTED PROOF

Layout: T1 Standard STIX Book ID: 439206_1_En Book ISBN: 978-981-10-8475-1
Chapter No.: 12 Date: 17-3-2018 Time: 8:46 pm Page: 17/44

Author Proof

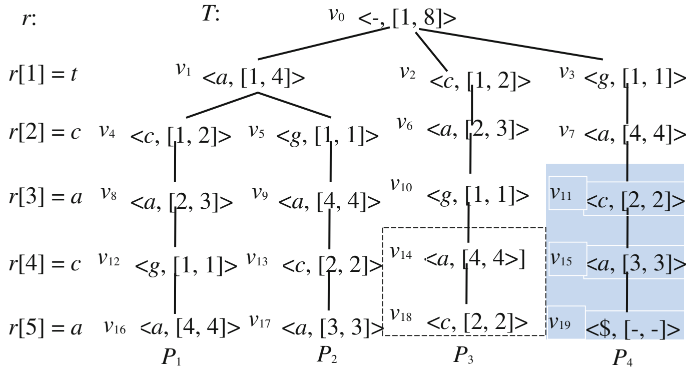

Fig. 7 Search for string matching with 2 mismatches

5 String Matching with *k* Mismatches

5.1 Basic Working Process

By the string matching with *k* mismatches, we allow up to *k* characters in a pattern *r* to match different characters in a target *s*. By using the BWT as an index, for finding all such string matches, a tree structure will be generated, in which each path corresponds to a *search sequence* discussed in the previous section. It is due to the possibility that a position in *r* may be matched to different characters in *s* and we need to call *search*() multiple times to do this task, leading to a tree representation.

Definition 5.1 (*search tree*) Let *r* be a pattern string and *s* be a target string. A search tree *T* (*S*-tree for short) is a tree structure to represent the search of *r* against $BWT(\bar{s})$ (which is equivalent to the search of $\bar{r}$ against *BWT*(*s*)). In *T*, each node is a pair of the form $<x, [\alpha, \beta]>$), and there is an edge from v ($=<x, [\alpha, \beta]>$) to u ($=<x', [\alpha', \beta']>$) if $search(x, L_v) = u$.

As an example, consider the case where $r = tcaca$, $s = acagaca$ and $k = 2$. To find all occurrences of *r* in *s* with up to two mismatches, a search tree *T* shown in Fig. 7 will be created.

In Fig. 7, v_0 is a virtual root, representing the whole *L*, and 'virtually' corresponds to the virtual starting character *r*[0] = '-'. By exploring paths $P_1 = v_1 \rightarrow v_4 \rightarrow v_8 \rightarrow v_{12} \rightarrow v_{16}$ and $P_2 = v_1 \rightarrow v_5 \rightarrow v_9 \rightarrow v_{13} \rightarrow v_{16}$, we will find two occurrences of *r* with 2 mismatches: *s*[1 … 5] ($=a_1c_1a_2g_1a_3$) and *s*[3 … 7] ($=a_2g_1a_3c_2a_4$) while by either $P_3 = v_2 \rightarrow v_6 \rightarrow v_{10} \rightarrow v_{14} \rightarrow v_{18}$ or $P_4 = v_3 \rightarrow v_7 \rightarrow v_{11} \rightarrow v_{15} \rightarrow v_{19}$ no string matching with at most 2 mismatches can be found.

A node $<x, [\alpha, \beta]>$in such a tree is called a *matching node* if it corresponds to a same character in *r*. Otherwise, it is called a *mismatching node*. For example, node $v_4 = <c, [1, 2]>$ is a matching node since it corresponds to *r*[2] = *c* while $v_1 = <a, [1, 4]>$ is a mismatching node since it corresponds to *r*[1] = *t*.

UNCORRECTED PROOF

Author Proof

553 For a path P_l, we can store all its mismatching positions in an array B_l of length
554 $k + 1$ such that $B_l[i] = j$ if $P_l[j] \neq r[j]$ and this is the ith mismatch between P_l and
555 r, where $P_l[j]$ is the jth character appearing on P_l. If the number of mismatches, k',
556 say, between P_l and r is less than $k + 1$, then the default value ∞ onwards, i.e.,
557

$$B_l\left[k^{'}+1\right]=B_l\left[K^{'}+2\right]=\cdots=B_l[k+1]=\infty.$$

559

560 We call B_l a *mismatch array*. For instance, in Fig. 3, for P_1, we have B_1 = [1, 4,
561 ∞], indicating that at position 1, we have the first mismatch $P_1[1] = a \neq r$
562 $[1] = t$ and at position 4 we have the second mismatch $P_1[4] = g \neq r[4] = a$. For
563 the same reason, we have B_2 = [1, 2, ∞], B_3 = [1, 2, 3], and B_4 = [1, 2, 3].
564 These data structures can be easily created by maintaining and manipulating a
565 temporary array B of length $k + 1$ to record the mismatches between the current
566 path P and r. Initially, each entry of B is set to be ∞ and an index variable i pointing
567 to the first entry of B. Each time a mismatch is met, its position is stored in $B[i]$ and
568 then i is increased by 1. Each time r is exhausted or B becomes full (i.e., each entry
569 is set a value not equal to ∞), we will store B as an B_l (and associate it with the leaf
570 node of the corresponding P_l.) Then, 'backtrack' to the lowest ancestor of the
571 current node, which has at least a branch not yet explored, to search a new path. For
572 instance, when we check v_{16}, r is exhausted and the current value of B is [1, 4, ∞].
573 We will store B in B_1 (the array associated with the leaf node v_{16} of P_1) and
574 'backtrack' to v_1 to explore a new path. At the same time, all those values in B,
575 which are set after v_1, will be reset to ∞, i.e., B will be changed to [1, ∞, ∞].
576 Now we consider another path P_3. The search along P_3 will stop at v_{10} since
577 when reaching it B becomes full (B = [1, 2, 3]). Therefore, the search will not be
578 continued, and v_{14}, v_{18} will not be created.
579 It is essentially a brute-force search to check all the possible occurrences of r in
580 s. Denote by n' the number of leaf nodes in T. The time used by this process is
581 bounded by $O(mn')$.
582 In fact, it is the main process discussed in [21]. The only difference is that in [21]
583 a simple heuristics is used, which precomputes, for each position i in r, the number
584 $\sigma(i)$ of consecutive, disjoint substrings in $r[i \ldots m]$, which do not appear in s. For
585 example, in Fig. 3, $\sigma(1) = 2$ since in $r[1 \ldots 5]$ = *tcaca* both $r[1 \ldots 1] = t$ and $r[2$
586 $\ldots 4]$ = *cac* do not occur in s = *acagaca*. But $\sigma(3) = 0$ since any substring in $r[1$
587 $\ldots 3]$ = *aca* does appear in s. Assume that the number of mismatches between $r[1$
588 $\ldots i - 1]$ and $P[1 \ldots i - 1]$ (the current path) is l. Then, if $k - l < \sigma(i)$, we can
589 immediately stop exploring the subtree rooted at $P[i - 1]$ as no satisfactory answers
590 can be found by exploring it.
591 The time required to establish such a heuristics is $O(n)$ by using $BWT(s)$ [21].
592 However, the theoretic time complexity of this method is still $O(mn')$. Even in
593 practice, this heuristics is not quite helpful since $\sigma(i)$ delivers only the information
594 related to $r[i \ldots m]$ and the whole s, rather than the information related to $r[i \ldots$
595 $m]$ and the relevant substrings of s, to which it will be compared. To see this, pay
596 attention to part of the tree marked grey in Fig. 7. Since $\sigma(3) = 0$, the search along
597 P_4 will be continued. But no answer can be found. The heuristics here is in fact

UNCORRECTED PROOF

598 useless since it is not about $r[3 \ldots 5]$ and $s[5 \ldots 7]$, which is to be checked in a next
599 step.

UNCORRECTED PROOF

600 5.2 Mismatch Information

601 Searching *S*-trees in an improvement over scanning strings, but it often happens that
602 there are repetitive traversals of similar subtrees due to the multiple appearances of
603 a same pair. However, such repeated appearance of pairs cannot be simply removed
604 since they may be aligned to different positions in *r*. For example, the first
605 appearance of <*c*, [1, 2]> (v_4 in Fig. 3) is compared to $r[2]$ while its second
606 appearance (v_2) is to $r[1]$. Hence, we cannot use the result computed for v_4
607 (when <*c*, [1, 2]> is first met) as the result for v_2.

608 However, if we have stored the mismatch information *R* between substrings of *r*,
609 like $r[2 \ldots 4]$ and $r[1 \ldots 3]$, in some way, the mismatches along P_3 can be derived
610 from *R* and B_1 (the mismatches recorded for P_1), instead of simply exploring P_3
611 again in a way done for P_1. To do so, for each pair $i, j \in \{1, \ldots, m\}$, we need to
612 maintain a data structure R_{ij} containing the positions of the first $k + 1$ mismatches
613 between $r[i \ldots m - q + i]$ and $r[j \ldots m - q + j]$, where $q = \max\{i, j\}$, such that if
614 $R_{ij}[l] = x$ ($\neq \infty$) then $r[i + x - 1] \neq r[j + x - 1]$ or one of them does not exist,
615 and it is the *l*th mismatch between them.

616 Clearly, this task requires $O(km^2)$ time and space.

617 For this reason, we will precompute only part of *R*, instead of R_{ij} for all $i, j \in \{1,$
618 $\ldots, m\}$. Specifically, $R_{12}, \ldots, R_{1m}$ for *r* will be pre-constructed in a way as described
619 in [20], giving the positions of the mismatches between the pattern and itself at
620 various relative shifts. That is, each R_{1i} ($2 \leq i \leq m$) contains the positions within
621 *r* of the first $2k + 1$ mismatches between the substring $r[1 \ldots m - i]$ and $r[i + 1 \ldots$
622 $m]$, i.e., the overlapping portions of the two copies of pattern *r* for a relative shift of
623 *i*. Thus, if $R_{1i}[j] = x$, then $r[x] \neq r[i + x - 1]$ or one of them does not exist, which
624 is the *j*th mismatch between $r[1 \ldots m - i]$ and $r[i + 1 \ldots m]$. (See Fig. 8a for
625 illustration.)

626 In Fig. 8b, we show a pattern $r_1 = tcacg$ and all the possible right-to-left shifts:
627 $r_2 = r[2 \ldots 5] = cacg$, $r_3 = r[3 \ldots 5] = acg$, and so on. In Fig. 8c, we give R_{12},

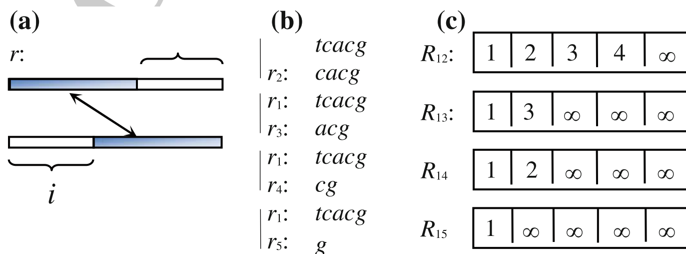

Fig. 8 Illustration for table *R*

Layout: T1 Standard STIX Book ID: 439206_1_En Book ISBN: 978-981-10-8475-1
Chapter No.: 12 Date: 17-3-2018 Time: 8:46 pm Page: 20/44

Author Proof

…, R_{15} for r_1. In an R_{1i}, if the number of mismatches, k', say, between $r[1 \ldots m - i]$ and $r[i + 1 \ldots m]$ is less than $2k + 1$, then the default value ∞ onwards, i.e.,

$$R_{1i}\left[K' + 1\right] = R_{1i}\left[k' + 2\right] = \cdots = R_{1i}[2k + 1] = \infty.$$

We will also use $\delta(R_{1i})$ to represent the number of all those entries in R_{1i}, which are not ∞. Trivially, $R_{11} = [\infty, \ldots, \infty]$.

Using the algorithm of [20], $R_{12}, \ldots, R_{1m}$ can be constructed in O($m$log$m$) time, just before the process for the string matching gets started. In addition, we need to keep $2k + 1$, rather than $k + 1$ mismatches in each R_{1i} ($i = 2, \ldots, m$), since for generating an R_{1j}, up to $2k + 1$ mismatches in some R_{1i} with $i < j$ are needed to get an efficient algorithm (see [20] for detailed discussion.)

Each time we meet a node u (compared to a certain $r[j]$), which is the same as an already encountered one v (compared to an $r[i]$), we need to derive dynamically the relevant mismatches, R_{ij}, between $r[i \ldots m - q + i]$ and $r[j \ldots m - q + j]$ from R_{1i} and R_{1j}, as well as r, to compute mismatch information for some new paths (to avoid exploring them by using *search*()). (A node $<x, [\alpha, \beta]>$ is said to be the same as another node $<x', [\alpha', \beta']>$ if $x = x$, $\alpha = \alpha'$ and $\beta = \beta'$.) For this purpose, we design a general algorithm to create R_{ij} efficiently.

- Let ω, ω_1 and ω_2 be three strings. Let A_1 and A_2 be two arrays containing all the positions of mismatches between ω and ω_1, and ω and ω_2, respectively.
- Create a new array A such that if $A[i] = j$ ($\neq \infty$), then $\omega_1[j] \neq \omega_1[j]$, or one of them does not exists. It is the ith mismatch between them.

The algorithm works in a way similar to the *sort-merge-join*, but with a substantial difference in handling a case when an entry in A_1 is checked against an equal entry in A_2. In the algorithm, two index variables p and q are used to scan A_1 and A_2, respectively. The result is stored in A.

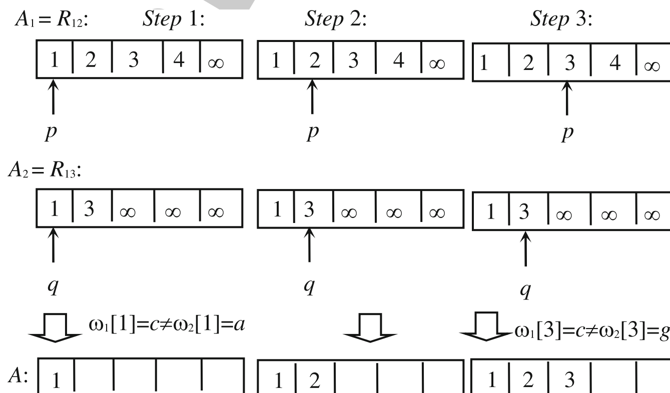

Fig. 9 Illustration for *merge*()

UNCORRECTED PROOF

Author Proof

656 1. $p := 1$; $q := 1$; $l := 1$;
657 2. If $A_2[q] < A_1[p]$, then $\{A[l] := A_2[q]$; $q := q + 1$; $l := l + 1;\}$
658 3. If $A_1[p] < A_2[q]$, then $\{A[l] := A_1[p]$; $p := p + 1$; $l := l + 1;\}$
659 4. If $A_1[p] = A_2[q]$, then {if $\omega_1[p] \neq \omega_2[q]$, then $\{A[l] := q; l := l + 1;\}$ $p := p + 1$;
660 $q := q + 1;\}$
661 5. If $p > |A_1|$, $q > |A_2|$, or both $A_1[p]$ and $A_2[q]$ are ∞, stop (if A_1 (or A_2) has some
662 remaining elements, which are not ∞, first append them to the rear of A, and
663 then stop.)
664 6. Otherwise, go to (2).
665

666 We denote this process as *merge*(A_1, A_2, ω_1, ω_2). As an example, let us consider
667 the case where $A_1 = R_{12} = [1, 2, 3, 4, \infty]$, $A_1 = R_{13} = [1, 3, \infty, \infty, \infty]$, $\omega_1 = r[2$
668 $\ldots 4] = cacg$ and $\omega_1 = r[3 \ldots 5] = acg$, and demonstrate the first three steps of the
669 execution of *merge*(A_1, A_2, ω_1, ω_2) in Fig. 9. The result is $A = [1, 2, 3, 4]$, showing
670 the mismatches between these two substrings.
671 In step 1: $p = 1$, $q = 1$, $l = 1$. We compare $A_1[p] = A_1[1]$ and $A_2[q] = A_2[1]$.
672 Since $A_1[1] = A_2[1] = 1$, we will compare $\omega_1[1]$ and $\omega_2[1]$, and find that
673 $\omega_1[1] = c \neq \omega_2[1] = a$. Thus, $A[1]$ is set to be 1. $p := p + 1 = 2$, $q := q + 1 = 2$,
674 $l := l + 1 = 2$.
675 In step 2: $p = 2$, $q = 2$, $l = 2$. we compare $A_1[2]$ and $A_2[2]$. Since
676 $A_1[2] = 2 < A_2[2] = 3$, $A[2]$ is set to be 2. $p := p + 1 = 3$, $q := 2$, $l := l + 1 = 3$.
677 In step 3: $p = 3$, $q = 2$, $l = 3$. We compare $A_1[3]$ and $A_2[2]$, and find that
678 $A_1[3] = A_2[2] = 3$. So, we need to compare $\omega_1[3]$ and $\omega_2[3]$. Since $\omega_1[3] = c \neq \omega_2$
679 $[3] = g$, $A[3]$ is set to be 3. $p := p + 1 = 4$, $q := 3$, $l := l + 1 = 4$.
680 In a next step, we have $p = 4$, $q = 3$, $l = 4$. We will compare $A_1[4]$ and $A_2[3]$.
681 Since $A_1[4] = 4 < A_2[3] = \infty$, we set $A[4]$ to 4.
682 Obviously, the running time of this process is bounded by O(k).

683 **Proposition 5.1** *Let A be the result of merge(A_1, A_2, ω_1, ω_2) with A_1, A_2, ω_1, ω_2*
684 *defined as above. Let k be the number of mismatches between ω_1 and ω_2. Then, A*
685 *[i] must be the position of the ith mismatch between ω_1 and ω_2, or , depending on*
686 *whether i is $\leq$ k .*

687 *Proof* Consider $\omega_2[j]$. Position j may satisfy either, neither, or both of the following
688 conditions:

689 (i) j corresponds to the lth mismatch between ω and ω_2 for some l, i.e.,
690 $\omega[j] \neq \omega_2[j]$ and $A_2[l] = j$.
691 (ii) j corresponds to the fth mismatch between ω and ω_1 for some f, i.e.,
692 $\omega[j] \neq \omega_1[j]$ and $A_1[f] = j$.
693

694 If (i) holds, but (ii) not, (2) in *merge*(A_1, A_2, ω_1, ω_2) will be executed. Since in
695 this case, we have $\omega[j] \neq \omega_2[j]$ and $\omega[j] = \omega_1[j]$, (2) is correct.
696 If (ii) holds, but (i) not, (3) will be executed. Since in this case, we have
697 $\omega[j] \neq \omega_1[j]$ and $\omega[j] = \omega_2[j]$, (3) is also correct.
698 If both (i) and (ii) hold, no conclusion concerning $\omega_1[j]$ and $\omega_2[j]$ can be drawn
699 and we need to compare them. In this case, (4) is executed. If neither (i) nor (ii) is

UNCORRECTED PROOF

Layout: T1 Standard STIX | Book ID: 439206_1_En | Book ISBN: 978-981-10-8475-1
Chapter No.: 12 | Date: 17-3-2018 Time: 8:46 pm | Page: 22/44

Author Proof

satisfied, we must have $\omega[j] = \omega_2[j]$ and $\omega[j] = \omega_1[j]$. So $\omega_2[j] = \omega_1[j]$, i.e., we have a matching at j. □

5.3 Main Idea: Mismatch Information Derivation

Now we are ready to present the main idea of our algorithm, which is similar to the generation of an S-tree described in Subsection A. However, each time we meet a node u (compared to a position in r, say, $r[j]$), which is the same as a previous one v (compared to a different position in r, say, $r[i]$), we will not explore $T[u]$ (the subtree rooted at u), but do the following operations to derive the relevant mismatching information:

First, we will create R_{ij} by executing $merge(R_{1i}, R_{1j}, r[i \ldots m - q + i], r[j \ldots m - q + j])$, where $q = \max\{i, j\}$. Then, we will created a set of mismatch arrays for all the sub-paths in $T[u]$, which start at u and end at a leaf node, by doing two steps shown below.

- For each path P_i going through v, figure out a sub-array of B_l, denoted as B_l^i, containing only those values in B_l, which are larger than or equal to i. Moreover, each value in it will be decreased by $i - 1$. (For example, for $B_1 = [1, 4, \infty]$, we have $B_l^i = [1, 4, \infty]$, $B_l^2 = [3, \infty]$, $B_l^3 = [2, \infty]$, $B_l^4 = [1, \infty]$, and $B_l^5 = [\infty]$.)
- Create the mismatch arrays for all the paths going through u by executing $merge(B_l^i, R_{ij}, P_l[i \ldots m_l], r[j \ldots m])$ for each P_l, where $m_l = |P_l|$.

We denote this process as $mi\text{-}creation(u, v, j, i)$.

As an example, consider v_2 (in Fig. 7, labeled <c, [1, 2]> and compared to $r[1] = t$), which is the same as v_4 (compared to $r[2] = c$). By executing $mi\text{-}creation(v_2, v_4, 1, 2)$, the following operations will be performed, to avoid repeated access of the corresponding subtree (i.e., part of P_3 shown in Fig. 10a):

1. Create R_{21}:
   $R_{12} = [1, 2, 3, 4, \infty]$, $R_{11} = [\infty, \infty, \infty, \infty, \infty]$,

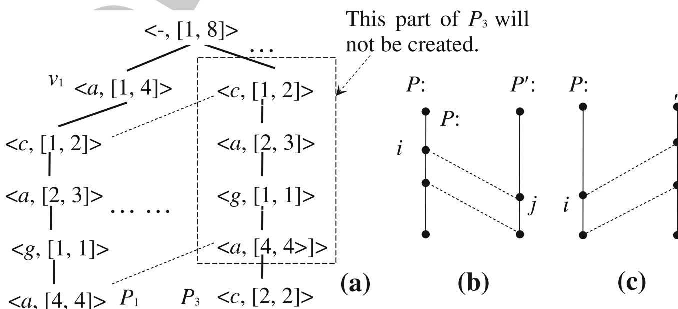

Fig. 10 Illustration for derivation of mismatch information

UNCORRECTED PROOF

Author Proof

729 $R_{21} = merge(R_{12}, R_{11}, r[2 \ldots 5], r[1 \ldots 4]) = [1, 2, 3, 4]$.
730 2. Create part of mismatch information for P_3:
732 $B_1 = [1, 4, \infty]$, $B_l^2 = [3, \infty]$. $P_1[2 \ldots 5] = caga$, $r[1 \ldots 4]) = caca$.
734 $merge(B_l^2, R_{21}, P_1[2 \ldots 5], r[1 \ldots 4]) = [1, 2, 3, 4]$.
735

736 In general, we will distinguish between two cases:

737 (i) $i < j$. This case can be illustrated in Fig. 10b. In this case, the mismatch
738 information for the new paths can be completely derived.
739 (ii) $i > j$. This case can be illustrated in Fig. 10c. In this case, only part of mis-
740 match information for the new paths can be derived. Thus, after the execution
741 of *merge*(), we have to continue to extend the corresponding paths.
742

743 Therefore, among different appearances of a certain node v, we should always
744 use the one compared to $r[i]$ with i being the least to derive as much mismatch
745 information as possible for to be created paths.
746 Finally, we notice that it is not necessary for us to consider the case $i = j$ since
747 the same node will never appear at the same level more than once. The following
748 lemma is easy to prove.

749 **Lemma 5.1** *In an S-tree T, if two nodes are with the same pair, then they must*
750 *appear at two different levels.* □

751 5.4 Algorithm Description

752 The main idea presented in the previous subsection can be dramatically improved.
753 Instead of keeping a B_l for each P_l, we can maintain a general tree structure, called a
754 *mismatch tree*, to store the mismatch information for all the created paths. First, we
755 define two simple concepts related to *S*-trees.

756 **Definition 5.2** (*match path*) A sub-path in an *S*-tree *T* is called a match path if each
757 node on it is a matching node in *T*.

758 **Definition 5.3** (*maximal match sub-path*) A maximal match sub-path (*MM-path* for
759 short) in an *S*-tree *T* is a match sub-path such that the parent of its first node in *T* is a
760 mismatching node and its last node is a leaf node or has only mismatching nodes as
761 its children.
762 For example, edge $v_4 \rightarrow v_8$ in *T* shown in Fig. 7 is a *MM-path*. Path $v_9 \rightarrow v_{13}$
763 $\rightarrow v_{17}$ is another one. The node v_{16} alone is also a *MM-path* in *T*.
764 Based on the above concepts, we define another important concept, the so-called
765 *mismatch trees*.

766 **Definition 5.4** (*mismatch trees*) A mismatch tree *D* (*M-tree* for short) for a given *S*-
767 tree *T*, is a tree, in which for each mismatching node <*x*, [α, β]> (compared to *r*
768 [*i*] for some *i*) in *T* we have a node of the form <*x*, *i*>, and for each *MM*-path a node

UNCORRECTED PROOF

Layout: T1 Standard STIX Book ID: 439206_1_En Book ISBN: 978-981-10-8475-1
Chapter No.: 12 Date: 17-3-2018 Time: 8:46 pm Page: 24/44

Author Proof

of the form $<-, 0>$. There is an edge from u to u' if one of the following two conditions is satisfied:

- u is of the form $<x, i>$ corresponding to a pair $<x, [\alpha, \beta]>$ (compared to $r[i]$), which is the parent of the first node of an *MM*-path (in T) represented by u'; or
- u is of the form $<-, 0>$ and u' corresponds to a mismatching node which is a child of a node on the *MM*-path represented by u.
  Without causing confusion, we will also call $<-, 0>$ in D a *matching node*, and $<x, i>$ a *mismatching node*.

For example, for T shown in Fig. 7, we have its *M*-tree shown in Fig. 11, in which u_0 is a virtual root corresponding to the virtual root of the *S*-tree shown in Fig. 7. Its value is also set to be $<-, 0>$ since it will be handled as a matching node. Then, each path in the *M*-tree corresponds to a B_l. For instance, path $u_0 \rightarrow u_1 \rightarrow u_4 \rightarrow u_8 \rightarrow u_{12}$ corresponds to $B_1 = [1, 4, \infty]$ if all the matching nodes on the path are ignored. For the same reason, $u_0 \rightarrow u_1 \rightarrow u_5 \rightarrow u_{19}$ corresponds to $B_2 = [1, 2, \infty]$.

In addition, we can store all the different nodes v $(=<x, [\alpha, \beta]>)$ in T in a hash table with each entry associated with a pointer to a node in the corresponding *M*-tree D, described as follows.

- If v is a mismatching node compared to $r[i]$ for some $i \in \{1, \ldots, m\}$, a node $u = <x, i>$ will be created in D and a pointer (associated with v, denoted as $p(v)$) to u will be generated.
- If v is a matching node, a node $u = <-, 0>$ will be created in D and $p(v)$ to u will be generated. If the parent u' of u itself is $<-, 0>$, u will be merged into its parent. That is, v will be linked to u' while u itself will not be generated.

For instance, when $<a, [1, 4]>$ (v_1 in T shown in Fig. 7) is created, it is compared to $r[1] = t$. Since $a \neq t$, we have a mismatch and then $u_1 = <a, 1>$ in the *M*-tree D will be generated. At the same time, we will insert $<a, [1, 4]>$ into the hash table and produce a pointer associated with it to u_1 (see Fig. 11 for illustration).

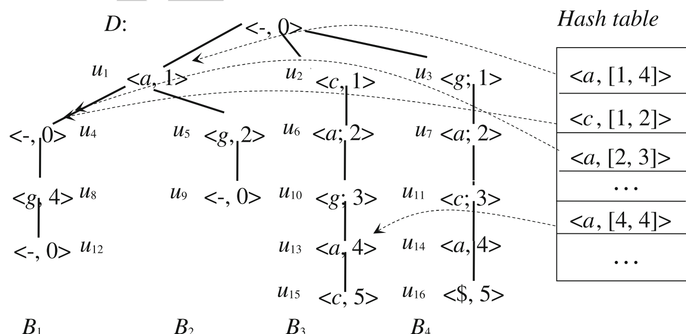

Fig. 11 A mismatch tree

CORRECTED PROOF

Layout: T1 Standard STIX Book ID: 439206_1_En Book ISBN: 978-981-10-8475-1
Chapter No.: 12 Date: 17-3-2018 Time: 8:46 pm Page: 25/44

Author Proof

However, when <*c*, [1, 2]> (v_4 in *T* shown in Fig. 7) is created, it is compared to *r* [2] = *c* and we have a matching. For this, a node <−, 0> (u_4 in Fig. 7) will be generated, and a link from <*c*, [1, 2]> to it will be established. But when <*a*, [2, 3] > (v_8 in *T* shown in Fig. 7, compared to *r* [5] = *a*) is met, no node in *D* will be generated since it is a matching node (in *T*) and the parent (u_4 in Fig. 11) of the node to be created for it is also <−, 0>. We will simply link it to its parent u_4.

In order to generate *D*, we will use a stack *S* to control the process, in which each entry is a quadruple (*v*, *j*, κ, *u*), where

v—a node inserted into the hash table.

j—*j* is an integer to indicate that *v* is the *j*th node on a path in *T* (counted from the root with the root as the 0th node).

κ—the number of mismatches between the path and *r*[0 … *j*] (recall that *r* [0] = '−').

u—the parent of a node in *D* to be created for *v*.

In this way, the *parent/child* link between *u* and the node to be created for *v* can be easily established, as described below.

Each time an entry *e* = (*v*, *j*, κ, *u*) with *v* = <*x*, [α, β]> is popped out from *S*, we will check whether *x* = *r*[*j*].

(i) If *x* = *r*[*j*], we will generate a node *u*′= <*x*, *j*> and link it to *u* as a child.

(ii) If *x* ≠ *r*[*j*], we will check whether *u* is a node of the form <−, 0>. If it is not the case, generate a node *u*′ = <−, 0>.
Otherwise, set *u*′ to be *u*.

(iii) Using *search*() to find all the children of *v*: v_1, …, v_l. Then, push each (v_i, *j* + 1, κ′, *u*′) into *S* with κ′ being κ or κ + 1, depending on whether y_i= *r* [*j* + 1], where v_i = <y_i, [α_i, β_i]> .

Note that in this process it is not necessary to keep *T*, but insert all the nodes (of *T*) in the hash table as discussed above.

CORRECTED PROOF

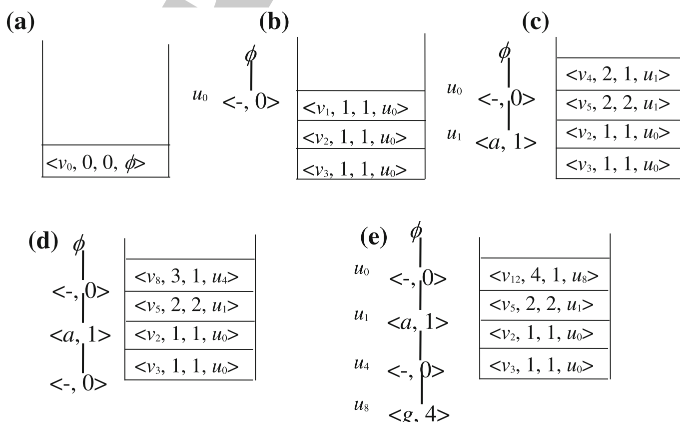

Fig. 12 Illustration for stack changes

Layout: T1 Standard STIX Book ID: 439206_1_En Book ISBN: 978-981-10-8475-1
Chapter No.: 12 Date: 17-3-2018 Time: 8:46 pm Page: 26/44

Author Proof

Example 5.1 In this example, we run the above process on $r = tcaca$ and $L = BWT(\bar{s})$ shown in Fig. 3c with $k = 2$, and show its first 5 steps. The tree created is shown in Fig. 12.

Step 1: Create the *root*, v_0 = <–, [1, 8]>. Push (v_0, 0, 0, ϕ) into S, where ϕ is used to represent the parent of the root D. See Fig. 12a.

Step 2: Pop out the top element (v_0, 0, 0, ϕ) from S. Create the root u_0 of D, which is set to be a child of ϕ. Push <v_3, 1, 1, u_0>, <v_2, 1, 1, u_0>, <v_1, 1, 1, u_0> into S, where v_3, v_2, and v_1 are three children of v_0. See Fig. 12b.

Step 3: Pop out (v_1, 1, 1, u_0) from S. v_1 = <a, [1. 4]>. Since $r[1] = t \neq a$, a mismatching node u_1 = <a, 1> will be created and set to be a child of u_0. Then, push (v_4, 2, 1, u_1) into S, where v_4 is the child of v_1. See Fig. 12c.

Step 4: Pop out (v_4, 2, 1, u_1) from S. v_4 = <c, [1, 2]>. Since $r[2] = c$, we will check whether u_1 is a matching node. It is the case. So, a matching node u_4 = <–, 0> will be created and set to be a child of u_1. Then, push (v_8, 3, 1, u_4) into S, where v_8 is the child of v_4. See Fig. 12d.

Step 5: Pop out (v_8, 3, 1, u_4) from S. v_8 = <a, [2, 3]>. $r[3] = a$. However, no new node is created since u_4 is a matching node. Push (v_{12}, 4, 1, u_4) into S, where v_{12} is the child of v_8. See Fig. 12e. □

From the above sample trace, we can see that D can be easily generated. In the following, we will discuss how to extend this process to a general algorithm for our task.

As with the basic process, each time a node v = <x, [α, β]> (compared to $r[j]$) is encountered, which is the same as a previous one v' = <x', [α′, β′]> (compared to $r[i]$), we will not create a subtree in T in a way as for v', but create a new node u for v in D and then go along $p(v')$ (the link associated with v') to find the corresponding nodes u' in D and search $D[u']$ in the breadth-first manner to generate a subtree rooted at u in D by simulating the merge operation discussed in Subsection B. In other words, $D[u]$ (to be created) corresponds to the mismatch arrays for all the paths going though v in T, which will not be actually produced. See Fig. 13 for illustration.

For this purpose, we introduce a third kind of nodes of the form <–, ∞> into D to represent symbol ∞ in mismatching arrays. Such a node is always the last node of a path in D.

To search $D[u']$ breadth-first, a queue data structure Q is used to control the search of $D[u']$ and at the same time generate $D[u]$. In Q, each entry e is a triplet (w, γ, h) with w being a node in $D[u']$, γ an entry in R_{ij}, and h is the number of

CORRECTED PROOF

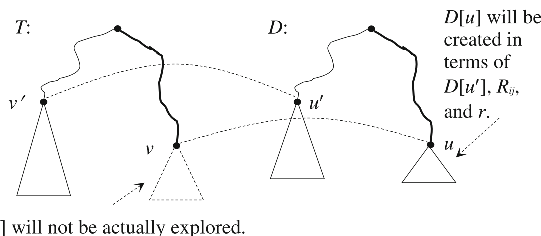

Fig. 13 Illustration for generation of subtrees in T

Author Proof

mismatching nodes on the path from the root to the node to be created in $D[u]$. Initially, put $(u', R_{ij}[1], h')$ into Q, where h' is the number of mismatching nodes on the path from the root to u. In the process, when e is dequeued from Q (taken out from the front), we will make the following operations (simulating the steps in *merge*()):

1. Let $e = (w, R_{ij}[l], h)$. Assume that $w = <z, f>$ and $R_{ij}[l] = val$.
   - If $<z, f>$ is equal to $<-, 0>$, then create a copy of $<-, 0>$ added to $D[u]$. Let u_1, …, u_g be the children of w. We will enqueue (append at the end) $(u_1, R_{ij}[l], h)$, …, $(u_g, R_{ij}[l], h)$ into Q in turn.
   - If is a mismatching node, do (2), (3), or (4).
   - If $<z, f>$ is equal to $<-, \infty >$, do (5).
2. If $f < i + val - 1$, add $<z, j + f - i + 1>$ to $D[u]$. If $h < k + 1$, enqueue $(u_1, R_{ij}[l], h + 1)$, …, $(u_g, R_{ij}[l], h + 1)$ into Q.
3. If $f > i + val - 1$ (and $f \neq \infty$), we will scan R_{ij} starting from $R_{ij}[l]$ until we meet the largest $l' \leq k - h + l$ such that $f > i + R_{ij}[l'] - 1$. For each $R_{ij}[q]$ $(l \leq q \leq l')$, we create a new node $<r[i + R_{ij}[q] - 1], j + R_{ij}[q] - 1>$ added to $D[u]$. If $l' < k - h + l$, add $<-, \infty>$ to $D[u]$, and enqueue $<w, R_{ij}[l' + 1], h + l' - l + 1>$ into Q.
4. If $f = i + val - 1$, we will distinguish between two subcases: $z \neq r[j + val - 1]$ and $z = r[j + val - 1]$. If $z \neq r[j + val - 1]$, we have a mismatch and a copy of w will be generated and added to $D[u]$. If $h < k + 1$, enqueue $(u_1, R_{ij}[l + 1], h + 1)$, …, $(u_g, R_{ij}[l + 1], h + 1)$ into Q. If $z = r[j + val - 1]$, create a node $<-, 0>$ added to $D[u]$. (If its parent is also $<-, 0>$, it will be merged into its parent.) Also enqueue $<u_1, R_{ij}[l + 1], h)$, …, $<u_g, R_{ij}[l + 1], h)$ into Q.
5. If $w = <-, \infty>$, scan R_{ij} starting from $R_{ij}[l]$ until we find the largest $l' \leq k - h + l$ such that $R_{ij}[l] \neq \infty$. For each $R_{ij}[q]$ $(l \leq q \leq l')$, we create a new node $<r[i + R_{ij}[q] - 1], j + R_{ij}[q] - 1>$ added to $D[u]$. If $l' < k - h + l$, add $<-, \infty>$ to $D[u]$, and enqueue $<w, R_{ij}[l' + 1], h + l' - l + 1>$ into Q.

In the above process, (2) corresponds to step 3 in *merge*(), (3) to step 4 in *merge*(), and (4) to step 5 in *merge*().

In (2), we handle the case when $f < i + val - 1$. In this case, we must have $r[f] = r[j + f - i]$. Then, by the following simple inference:

$P[f] \neq r[f], r[f] = r[j + f - i] \Rightarrow P[f] \neq r[j + f - i]$,

we know that a mismatching node should be added to $D[u]$. Here, P stands for a path starting from v' in T corresponding to a path starting from u' in D, and $P[f]$ for the fth node on P. See Fig. 11a for illustration.

In (3), we handle the case that $f > i + val - 1$. In this case, we have, for each $i' \in \{ i + val - 1, \ldots, f\}$ with $R_{ij}[q] = i'$ $(l \leq q \leq l')$,

$$p[i'] = r[i'], r[i'] \neq r[j + i' - i] \Rightarrow P[i'] \neq r[j + i' - i].$$

Thus, for each $R_{ij}[q]$ $(l \leq q \leq l')$, a mismatching node will be created and added to $D[u]$.

UNCORRECTED PROOF

Layout: T1 Standard STIX | Book ID: 439206_1_En | Book ISBN: 978-981-10-8475-1
Chapter No.: 12 | Date: 17-3-2018 Time: 8:46 pm | Page: 28/44

Author Proof

In the above description, we ignored the technical details on how $D[u]$ is constructed for simplicity. However, in the presence of $D[u']$, it is easy to do such a task by manipulating links between nodes and their respetive parents.

Denote the above process by *node-creation*$(w, \gamma, i, j, R_{ij})$. We have the following proposition.

Proposition 5.2 *node-creation*$(w, \gamma, i, j, R_{ij})$ *create nodes in* $D[u]$ *correctly.*

Proof The correctness of *node-creation*$(w, \gamma, i, j, R_{ij})$ can be derived from Proposition 1.□

Again, if $i > j$, $D[u]$ needs to be extended, which can be done in a way similar to the extension of mismatch arrays as discussed in Subsection *C*.

As an example, consider Figs. 7 and 11 once again. When we meet $<g, [1, 1]>$ (v_5 in T, compared to $r[2]$) for a second time, we will not generate $T[v_5]$ in Fig. 3, but $D[u_5]$ in Fig. 11. Comparing T and D, we can clearly see the efficiency of this improvement. In D, an *MM*-path in T is collapsed into a single node of the form $<-, 0>$.

The following is the formal description of the working process.

ALGORITHM ***A***(L, r, k)

```
begin
1. create root of T; push(S, (root, 0, 0, ϕ));
2. while S is not empty do {
3.  (v, j, κ, u) := pop(S); let v = <x, α, β>;
4.  if v is same as an existing v′ (compared to r[i]) then{
5.    q := max{i, j};
6.    R_ij := merge(R_1i, R_1j, r[i .. m – q + i], r[j .. m – q + j]);
7.    enqueuer(Q, (p(v′), R_ij[1]));
8.    while Q is not empty do {
9.      (w, γ) := dequeuer(Q); node-creation(w, γ, i, j, R_ij);}}
10. else {
11.   if x ≠ r[j] then create u′ = <x, j> and make it a child of u;
12.   else if u is <-, 0> then u′ := u
13.         else create u′ = <-, 0> and make it a child of u;
14.   p(v) := u′;     (*associate with v a pointer to u′.*)
15.   if j < |r| and κ ≤ k then {
16.     for each y ∈ ∑ within L_v do {
17.       w := search(y, L_v);
18.       if w ≠ ϕ then {
19.       if y = r[j + 1] then push(S, (w, j + 1, κ, u′));
20.       if y ≠ r[j + 1] and κ < k then {push(S, (w, j + 1, κ + 1, u′));
21.   }}}}
end
```

UNCORRECTED PROOF

Author Proof

If we ignore lines 3–9 in the above algorithm, it is almost a depth-first search of a tree. Each time an entry (v, j, κ, u) is popped out from S (see line 4), it will be checked whether v is the same as a previous one v' (compared to $r[i]$). (See line 4.) If it is not the case, a node u' for v will be created in D (see lines 11–14). Then, all the children of v will be found by using the procedure *search*() (see line 17) and pushed into S (see lines 18, and 19.) Otherwise, we will first create R_{ij} by executing $merge(R_{1i}, R_{1j}, r[i \ldots m - q + i], r[j \ldots m - q + j])$, where $q = max\{i, j\}$. (see lines 5–6.) Then, we create a subtree in D by executing a series of node-creation operations (see lines 8–9.)

Concerning the correctness of the algorithm, we have the following proposition.

Proposition 5.3
*Let L be a BWT-array for the reverse $\bar{s}$ of a target string s, and r a pattern. Algorithm **A**(L, r, k) will generate a mismatching tree D, in which each root-to-leaf path represents an occurrence of r in s having up to k positions different between r and s.*

Proof In the execution of $\boldsymbol{A}(L, r, k)$, two data structures will be generated: a hash table and a mismatching tree D, in which some subtrees in D are derived by using the mismatching information over r. Replacing each matching node in D with the corresponding maximum matching path and each mismatching node $<x, i>$ with the corresponding pair $<x, [\alpha, \beta]>$ (compared to $r[i]$), we will get an S-tree, in which each path corresponds to a *search sequence* discussed in Section III. Thus, in D each root-to-leaf path represents an occurrence of r in s having up to k positions different between r and s. □

The time complexity of the algorithm mainly consists of three parts: the cost for generating the mismatching information over r which is bounded by $O(m\log m)$; the cost for generating the M-tree and maintaining the hash table, which is bounded by $O(kn')$, where n' is the number of the M-tree's leaf nodes; and the cost for checking the characters in s against the characters in r, which is bounded by $O(n)$. So, the total running time is bounded by $O(kn' + n + m\log m)$.

6 Experiments

In this section, we report the test results. For all the experiments on both the multiple pattern string matching and the string matching with k matches, we use the same data sets summarized in Table 1.

Table 1 Characteristics of genomes

Genomes	Genome sizes (bp)
Rat (Rnor_6.0)	2,909,701,677
Zebra fish (GRCz10)	1,464,443,456
Rat chr1 (Rnor_6.0)	290,094,217
C. elegans (WBcel235)	103,022,290
C. merlae (ASM9120v1)	16,728,967

UNCORRECTED PROOF

Author Proof

961 To store *BWT*, $(\bar{s})$ we use 2 bits to represent a character $\in \{a, c, g, t\}$ and store 4
962 *rankAll* values (respectively in A_a, A_c, A_g, and A_t) for every 4 elements (in L) with
963 each taking 32 bits.
964 All the tested methods are implemented in C++, compiled by GNU make utility
965 with optimization of level 2. In addition, all of our experiments are performed on a
966 64-bit Ubuntu operating system, run on a single core of a 2.40 GHz Intel Xeon
967 E5-2630 processor with 32 GB RAM.

968 6.1 *Experiment on Multiple Pattern String Matching*

969 In this experiment, we have tested altogether five different methods:

970 • *Burrows Wheeler Transformation* (BWT for short),
971 • *Suffix tree based* (Suffix for short),
972 • *Hash table based* (Hash for short),
973 • *Trie-BWT* (tBWT for short, discussed in this paper),
974 • *Improved Trie-BWT* (itBWT for short, discussed in this paper).
975

976 Among them, the codes for the suffix tree based and hash based methods are
977 taken from the *gsuffix* package [45] while all the other three algorithms are
978 implemented by ourselves.

979 6.1.1 Tests on Synthetic Data Sets

980 All the synthetic data are created by simulating reads from the five genomes shown
981 in Table 1, with varying lengths and amounts. It is done by using the *wgsim*
982 program included in the *SAMtools* package [36] with default model for single reads
983 simulation.
984 Over such data, the impact of five factors on the searching time are tested:
985 number n of reads, length l of *reads* (pattern strings), size s of genomes, compact
986 factors f_1 of *rankAll*s (see Sect. 3.1) and compression factors f_2 of suffix arrays [11],
987 which are used to find locations of reads (in a reference genome) in terms of
988 formula (5) (see Sect. 3.2).

989 • *Tests with varying amount of reads*
990

991 In this experiment, we vary the amount n of reads with $n = 5, 10, 15, \ldots, 50$
992 millions while the reads are 50 bps or 100 bps in length extracted randomly from
993 *Rat chr1* and *C. merlae* genomes. For this test, the compact factors f_1 of *rankAll*s
994 are set to be 32, 64, 128, 256, and the compression factors f_2 of suffix arrays are set
995 to 8, 16, 32, 64, respectively. These two factors are increasingly set up as the
996 amount of reads gets increased.

UNCORRECTED PROOF

Author Proof

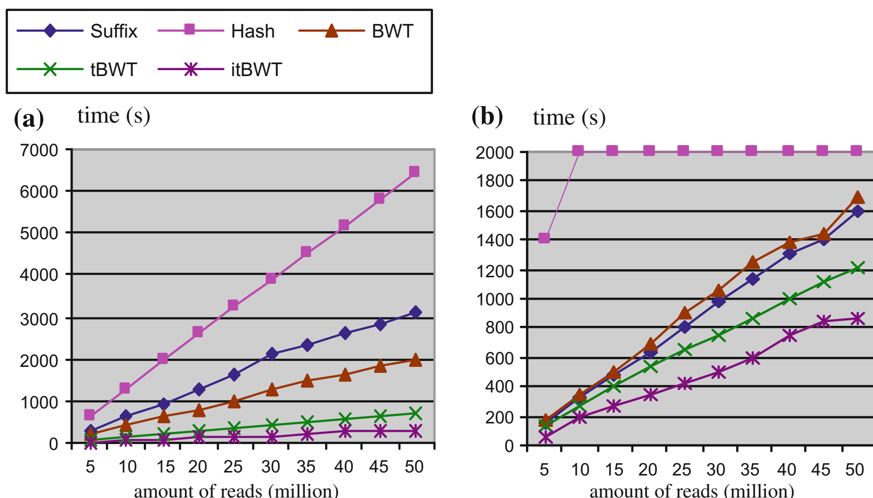

Fig. 14 Test results on varying amount of reads

In Fig. 14a, b, we report the test results of searching the Rat chr1 for matching reads of 50 and 100 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 short reads (of length 50 bps) the suffix-based is better than the BWT, but for long reads (of length 100 bps) they are comparable. The poor performance of the hash-based is due to its inefficient brute-force searching of genomes while for both the BWT and the suffix-based it is due to the huge amount of reads and each time only one read is checked. In the opposite, for both our methods tBWT and itBWT, the use of tries enables us to avoid repeated checkings for similar reads.

In these two figures, the time for constructing tries over reads is not included. It is because in the biological research a trie can be used repeatedly against different genomes, as well as often updated genomes. However, even with the time for constructing tries involved, our methods are still superior since the tries can be established very fast as demonstrated in Table 2, in which we show the times for constructing tries over different amounts of reads.

The difference between tBWT and itBWT is due to the different number of BWT array accesses as shown in Table 3. By an 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 tBWT) or a set of characters from more than one read (by itBWT).

Table 2 Time for trie construction over reads of length 100 BPS

No. of reads	30M	35M	40M	45M	50M
Time for Trie Con. (s)	51	63	82	95	110

UNCORRECTED

Layout: T1 Standard STIX Book ID: 439206_1_En Book ISBN: 978-981-10-8475-1
Chapter No.: 12 Date: 17-3-2018 Time: 8:46 pm Page: 32/44

Author Proof

Table 3 No. of BWT array accesses

No. of reads	30M	35M	40M	45M	50M
tBWT	47856K	55531K	63120K	70631K	78062K
itBWT	19105K	22177K	25261K	28227K	31204K

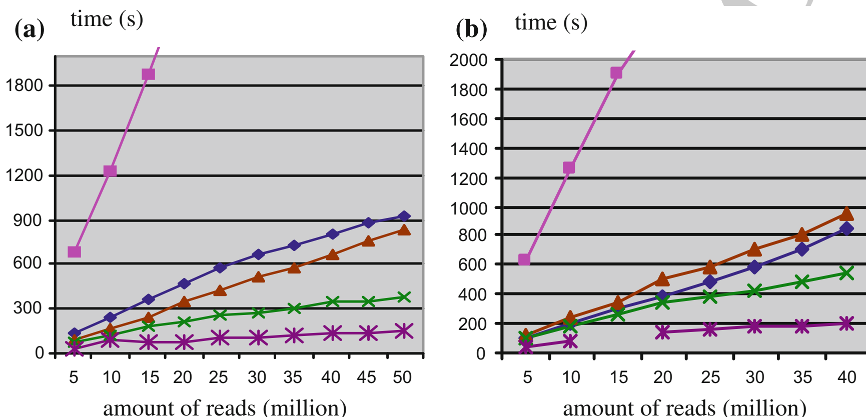

Fig. 15 Test results on varying amount of reads

Figure 15a, b show respectively the results for reads of length 50 bps and 100 bps over the *C. merolae* genome. Again, our methods outperform the other three 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 l of simulated reads with l = 35, 50, 75, 100, 125, …, 200. The results in Fig. 16a shows the difference among five methods, in which each tested set has 20 million reads simulated from the Rat chr1 genome with f_1 = 128 and f_2 = 16. In Fig. 16b, the results show the case that each set has 50 million reads. Figure 17a, b show the results of the same data settings but on *C. merlae* genome.

Again, in this test, the hash based performs worst while the suffix tree and the BWT method are comparable. Both our algorithms uniformly outperform 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 tBWT and itBWT 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.

UNCORRECTED PROOF

Author Proof

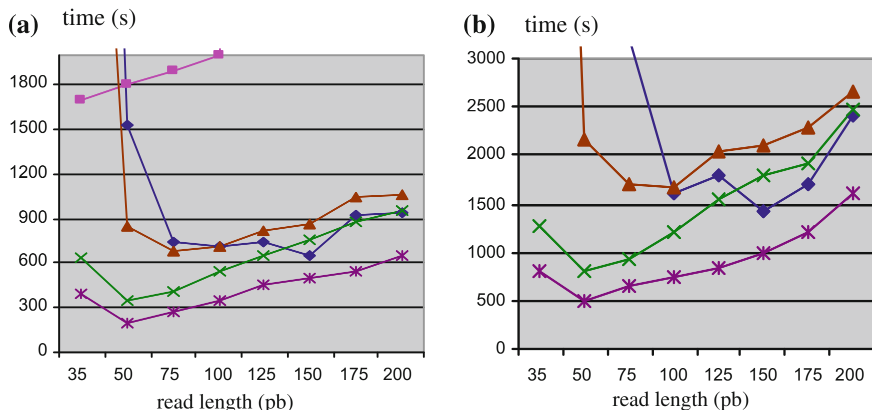

Fig. 16 Test results on varying length of reads

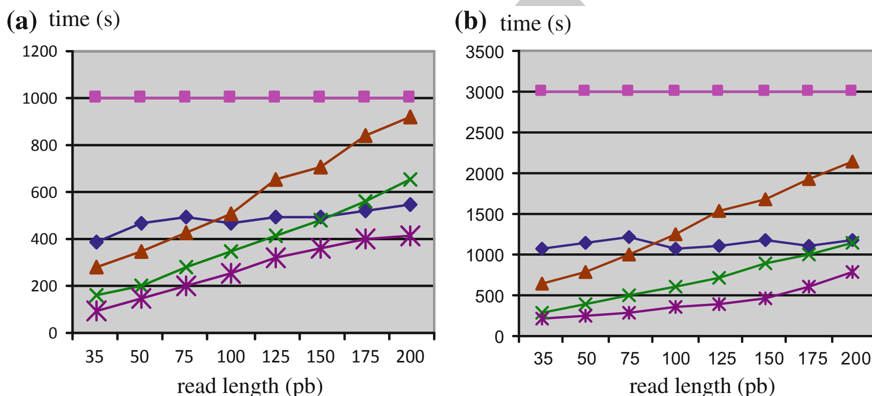

Fig. 17 Test results on varying length of reads

- *Tests with varying sizes of genome*

To examine the impacts of varying sizes of genomes, we have made four tests with each testing a certain set of reads against different genomes shown in Table 1. To be consistent with foregoing experiments, factors except sizes of genomes remain the same for each test with f_1 = 128 and f_2= 16. In Fig. 18a, b, we show the searching time on each genome for 20 million and 50 million reads of 50 bps, respectively. Figures 19a, b demonstrate the results of 20 million and 50 million reads but with each read being of 100 bps.

These figures show that, in general, as the size of a genome increases the time of read aligning for all the tested algorithms become longer. We also notice that the larger the size of a genome, the bigger the gaps between our methods and the other algorithms. The hash-based is always much slower than the others. For the suffix

Layout: T1 Standard STIX Book ID: 439206_1_En Book ISBN: 978-981-10-8475-1
Chapter No.: 12 Date: 17-3-2018 Time: 8:46 pm Page: 34/44

Author Proof

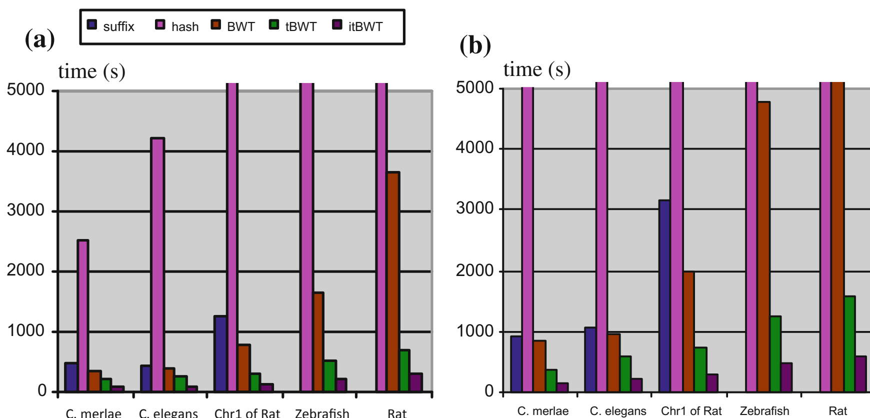

Fig. 18 Test results on varying sizes of genomes

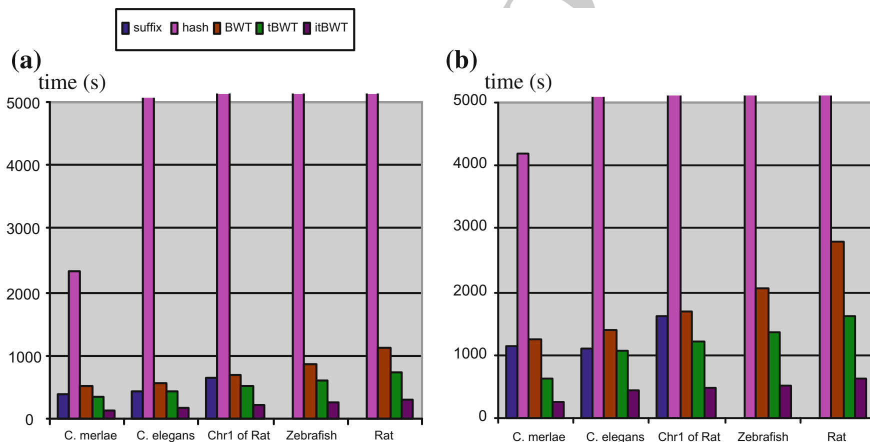

Fig. 19 Test results on varying sizes of genomes

tree, we only show the matching time for the first three genomes. 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 50 bp reads in Figs. 17 and 18 show that the tBWT and the itBWT are at least 30% faster than the BWT and the suffix tree, which happened on the *C. elegans* genome. For the Rat genome, our algorithms are even more than six times faster than the others.

Now let us have a look at Fig. 18a, 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.

Layout: T1 Standard STIX Book ID: 439206_1_En Book ISBN: 978-981-10-8475-1
Chapter No.: 12 Date: 17-3-2018 Time: 8:46 pm Page: 35/44

Author Proof

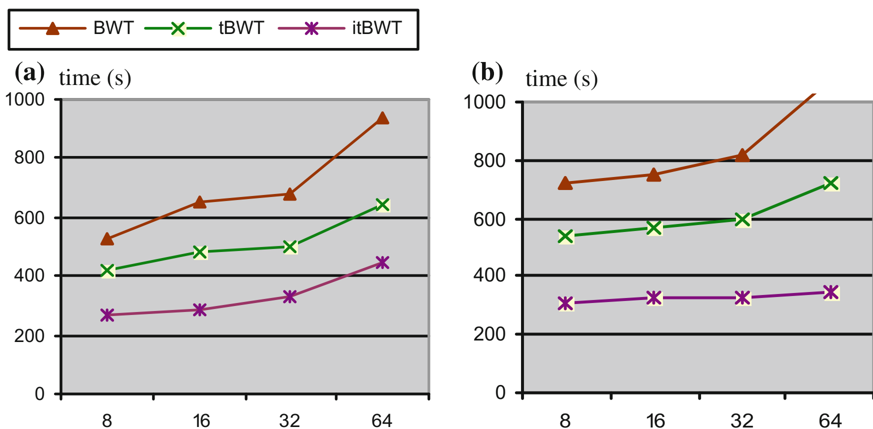

Fig. 20 Test results on varying compact and compression factors

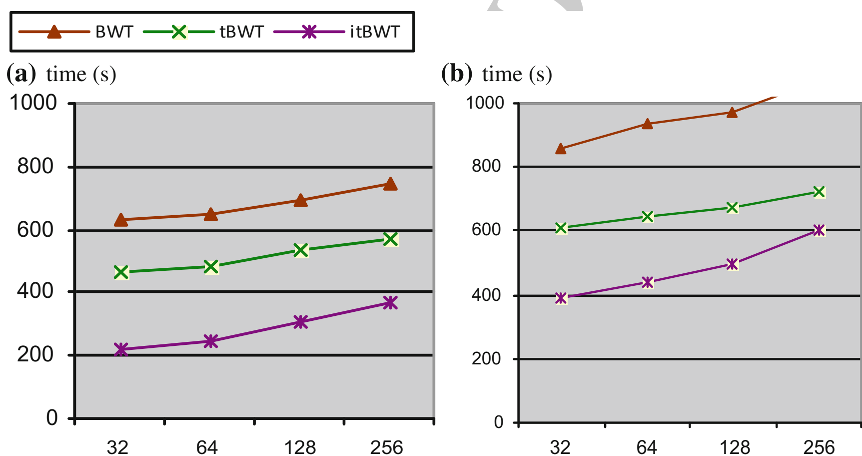

Fig. 21 Test results on varying compact and compression factors

- *Tests with varying compact and compression factors*

In the experiments, we focus only on the BWT method, since there are no compressions in both the suffix tree and the hash-based method. The following test results are all for 20 million reads with 100 bps in length. We first show the impact of f_1 on performance with $f_2 = 16, 64$ in Fig. 20a and b, respectively. Then we show the effect when f_2 is set to 64, 256 in Fig. 21a, b.

From these figures, we can see that the performance of all three methods degrade as f_1 and f_2 increase. Another noticeable point is that both the itBWT and the tBWT are not so sensitive to the high compression rate. Although doubling f_1 or f_2 will slow down their speed, they become faster compared to the BWT. For example, in

Author Proof

1071 Fig. 19, the time used by the BWT grows 80% by increasing f_1 from 8 to 64,
1072 whereas the growth of time used by the tBWT is only 50%. In addition, the factor f_1
1073 has smaller impact on the itBWT than the BWT and the tBWT, since the extra data
1074 structure used in the itBWT effectively reduced the processing time of the trie nodes
1075 by half or more.

1076 6.1.2 Tests on Real Data Sets

1077 For the performance assessment on real data, we obtain RNA-sequence data from
1078 the project conducted in an RNA laboratory at University of Manitoba [46]. This
1079 project includes over 500 million single reads produced by Illumina from a rat
1080 sample. Length of these reads are between 36 bps and 100 bps after trimming using
1081 Trimmomatic [47]. The reads in the project are divided into 9 samples with different
1082 amount ranging between 20 million and 75 million. Two tests have been conducted.
1083 In the first test, we mapped the 9 samples back to rat genome of ENSEMBL release
1084 79 [48]. We were not able to test the suffix tree due to its huge index size. The
1085 hash-based method was ignored as well since its running time was too high in
1086 comparison with the BWT. In order to balance between searching speed and
1087 memory usage of the BWT index, we set $f_1 = 128$, $f_2 = 16$ and repeated the
1088 experiment 20 times. Figure 22a shows the average time consumed for each
1089 algorithm on the 9 samples.

1090 Since the source of RNA-sequence data is the transcripts, the expressed part of
1091 the genome, we did a second test, in which we mapped the 9 samples again directly
1092 to the Rat *transcriptome*. This is the assembly of all transcripts in the Rat genome.
1093 This time more reads, which failed to be aligned in the first test, are able to be
1094 exactly matched. This result is showed in Fig. 22b.

UNCORRECTED PROOF

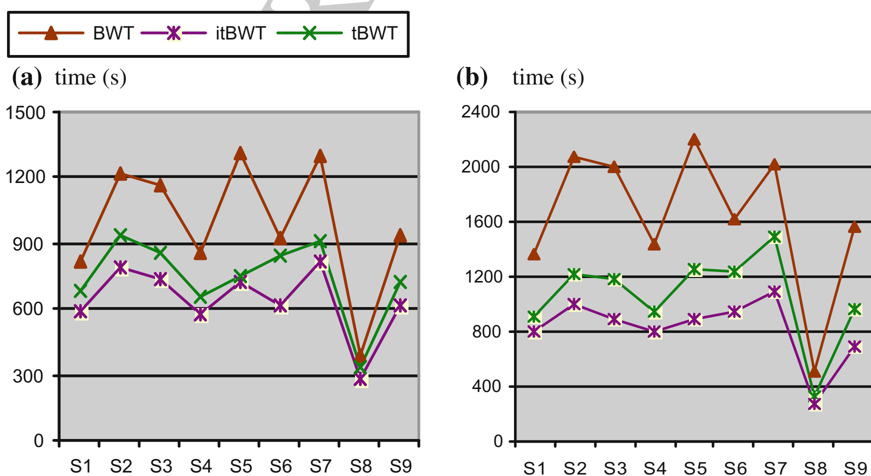

Fig. 22 Test results on real data

Layout: T1 Standard STIX Book ID: 439206_1_En Book ISBN: 978-981-10-8475-1
Chapter No.: 12 Date: 17-3-2018 Time: 8:46 pm Page: 37/44

Author Proof

1095 From Fig. 22a, b, we can see that the test results for real data set are consistent
1096 with the simulated data. Our algorithms are faster than the BWT on all 9 samples.
1097 Counting the whole data set together, itBWT is more than 40% faster compared
1098 with the BWT. Although the performance would be dropped by taking tries'
1099 construction time into consideration, we are still able to save 35% time using
1100 itBWT.

1101 *6.2 Experiment on String Matching with* K *Mismatches*

1102 In this experiment, we have tested altogether four different methods:

1103 • *BWT-based* [12] (BWT for short),
1104 • *Amir's method* [1] (Amir for short),
1105 • *Cole's method* [43] (Cole for short),
1106 • *Algorithm **A** discussed in this paper* ($A()$ for short)
1107

1108 By the BWT-based method, an S-tree will be created as described in Section IV,
1109 but with $\sigma(i)$ being used to cut off branches, where $\sigma(i)$ is the number of consec-
1110 utive, disjoint substrings in $r[i \ldots m]$ not appearing in s. By the Amir's algorithm, a
1111 pattern r is divided into several periodic stretches separated by 2 k aperiodic sub-
1112 strings, called breaks, as illustrated in Fig. 23. Then, for each break b_i, located at a
1113 certain position i, find all those substrings s_j (located at different positions j) in
1114 s such that $b_i = s_j$, and then mark each of them. After that, discard any position that
1115 is marked less than k times. In a next step, verify every surviving position in s.
1116 By the Cole's, a suffix tree for a target is constructed. (The code for constructing
1117 suffix trees is taken from the *gsuffix* package: http://gsuffix.Sourceforge.net/).
1118 For the test, five reference genomes shown in Table 1 are used. Similar to the
1119 first experiment, all the simulating reads are taken from these five genomes, with
1120 varying lengths and amounts. Concretely, we take 5000 reads with length varying
1121 from 100 to 300 bps.
1122 In Fig. 24a, b, we report the average time of testing the Rat (Rnor_6.0) for
1123 matching 100 reads of length 100 to 300 bps. From this figure, we can see that
1124 Algorithm $A()$ outperforms all the other three methods. But the Amir's method is
1125 better than the other two methods. The BWT-based and the Cole's method are
1126 comparable. However, for small k, the Cole's is a little bit better than the
1127 BWT-based method while for large k their performances are reversed.

UNCORRECTED PROOF

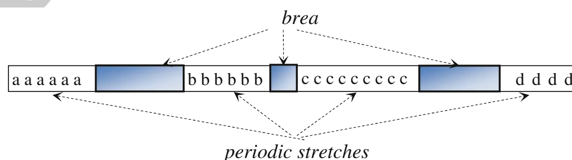

Fig. 23 Illustration for periodic stretches and breaks

Author Proof

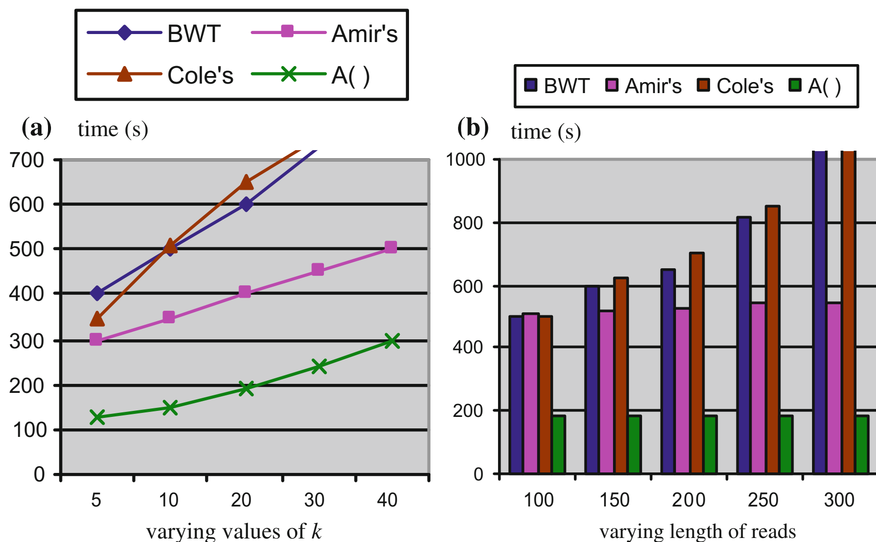

Fig. 24 Test results on varying values of k and read length

Table 4 Number of leaf nodes of S-trees

k/Length-of-read	5/50	10/100	20/150	30/200
No. of leaf nodes	2K	0.7M	16.5M	102M

UNCORRECTED

1128 To show why $A()$ has the best running time, we show the number n' of leaf nodes
1129 in the M-trees created by $A()$ for some tests in Table 4, which demonstrates that n'
1130 can be much smaller than n. Thus, the time complexity $O(kn')$ of $A()$ should be a
1131 significant improvement over $O\left(n\sqrt{k}\log k\right)$—the time complexity of Amir's.
1132 In this test (and also in the subsequent tests), the time for constructing BWT $(\bar{s})$ is
1133 not included as it is completely independent of r. Once it is created, it can be
1134 repeatedly used.
1135 In Fig. 24b, we show the impact of read lengths. For this test, k is set to 25. It
1136 can be seen that only the BWT-based and the Cole's are sensitive to the length of
1137 reads. For the BWT-based, more time is required to construct S-trees for longer
1138 reads while for the Cole's longer paths in a suffix tree will be searched as the
1139 lengths of reads increase. For the other two methods: $A()$ and the Amir's, the
1140 lengths of reads only impact the time for the read pre-processing, but it is com-
1141 pletely overshadowed by the time spent on searching genomes. For the Amir's, the
1142 time for recognizing breaks is linear in $|r|$ [2] while for $A()$ the time for generating
1143 the mismatch information is bounded by $O(|r|\log|r|)$. No significant difference
1144 between them can be measured.
1145 In Fig. 25a, b, we report the test results of searching the Zebra fish (GRCz10).
1146 Again, similar to Fig. 24a, the performance of Algorithm $A()$ is best, and the
1147 Amir's is still better than both the BWT-based and the Cole's.

Layout: T1 Standard STIX Book ID: 439206_1_En Book ISBN: 978-981-10-8475-1
Chapter No.: 12 Date: 17-3-2018 Time: 8:46 pm Page: 39/44

Author Proof

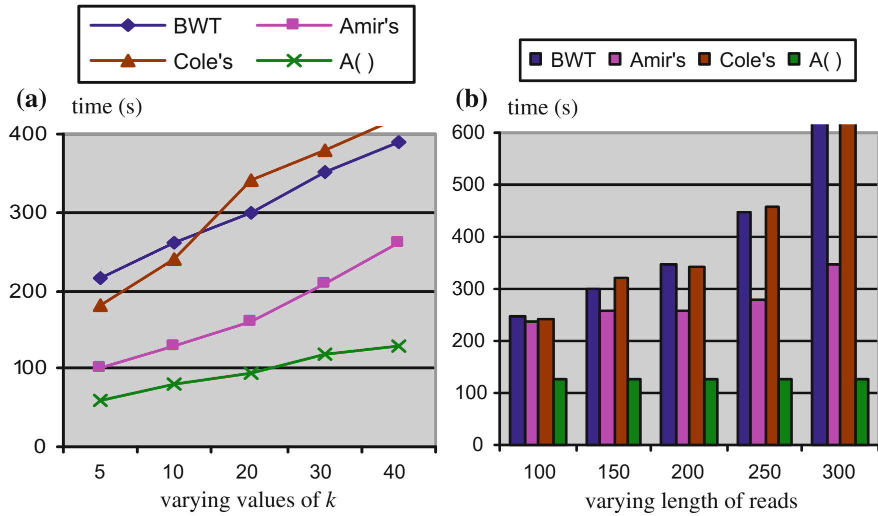

Fig. 25 Test results on varying values of k and read length

Table 5 Number of leaf nodes of S-trees

k/Length-of-read	5/50	10/100	20/150	30/200
No. of leaf nodes	0.7K	0.30M	9.2M	89M

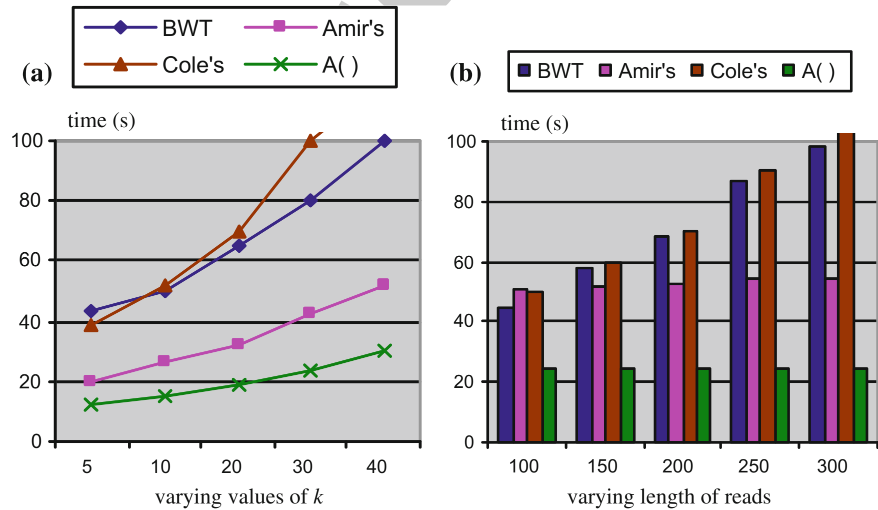

Fig. 26 Test results on varying values of k and read length

Layout: T1 Standard STIX Book ID: 439206_1_En Book ISBN: 978-981-10-8475-1
Chapter No.: 12 Date: 17-3-2018 Time: 8:46 pm Page: 40/44

Author Proof

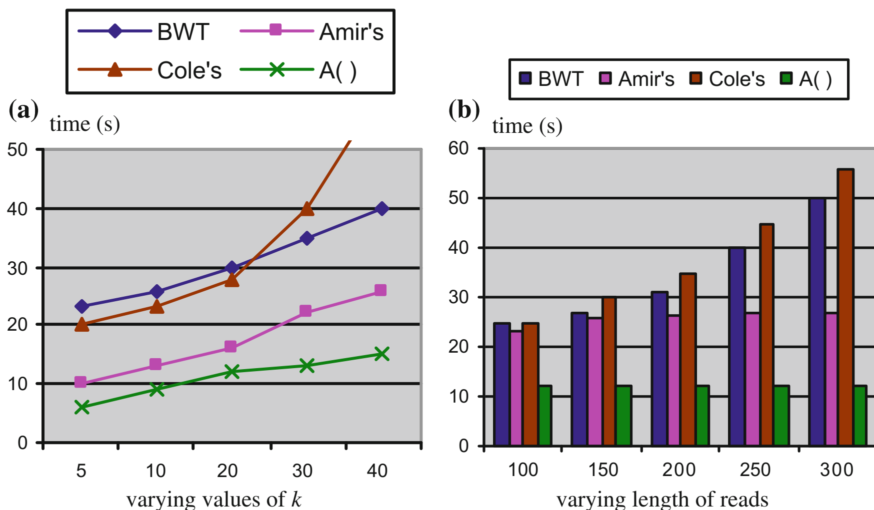

Fig. 27 Test results on varying values of *k* and read length

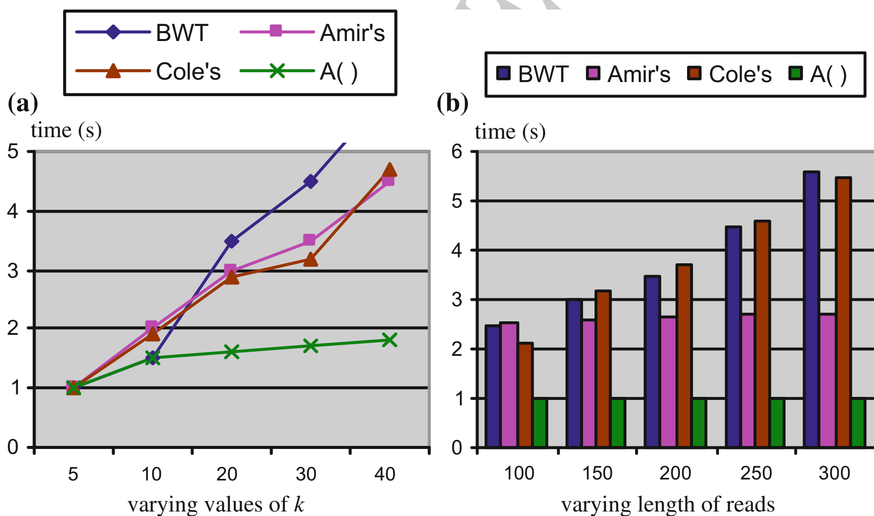

Fig. 28 Test results on varying values of *k* and read length

1148 In Table 5, we show the number n'.

1149 Figure 25b shares the same features as Fig. 24b. It also shows that only the
1150 BWT-based and the Cole's are sensitive to the length of reads.

1151 In Figs. 26, 27, and 28, we show the tests on Rat chr1 (Rnor_6.0), *C. elegans*
1152 (WBcel235), and *C. merlae* (ASM9120v1), respectively.

UNCORRECTED

Author Proof

1153 From these figures, the most important feature we can observe is that as the size 1154 of genomes becomes smaller, the difference between the Amir's and Cole's 1155 diminishes. But the BWT-based and *A*( ) remain the worst and the best, respec- 1156 tively. Although *A*( ) is impacted by the number of leaf nodes of an *S*-tree, the 1157 impact factor is small in comparison with the size of the whole *S*-tree, which 1158 dominates the time complexity of the BWT-based method. Also, the big difference 1159 between *A*( ) and *Amir's* shows that using *M*-trees the cost for creating mismatch 1160 information of *r*'s occurrences in *s* can be significantly reduced.

1161 7 Conclusion and Future Work

1162 In this chapter, two new methods have been discussed. One is to search a large 1163 volume of pattern strings against a single long target string, aiming at efficient 1164 next-generation sequencing in DNA databases. The main idea behind it is to 1165 combine the search of tries constructed over the patterns and the search of the BWT 1166 indexes over the target. Extensive experiments have been conducted, which show 1167 that our method improves the running time of the traditional methods by an order of 1168 magnitude or more.

1169 The second one is to do the string matching with *k* mismatches. Its main idea is 1170 to transform the reverse $\bar{s}$ of target string *s* to *BWT* ($\bar{s}$) and use the mismatch 1171 information over a pattern string *r* to speed up the computation. Its time complexity 1172 is bounded by $O(kn' + n + m\log m)$, where $m = |r|$, $n = |s|$, and n' is the number 1173 of leaf nodes of a tree structure produced during the search of a *BWT*(*s*). Our 1174 experiments show that it has a better running time than any existing on-line and 1175 index-based algorithms.

1176 As a future work, we will use the BWT to solve another important problem, the 1177 string matching with *k* errors. It seems to be more challenging than the *k* mis- 1178 matches since the Levenshtein distance is more difficult to handle than the Ham- 1179 ming distance.

1180 References

1181 1. Amir, A., Lewenstein, M., & Porat, E. (2004). Faster algorithms for string matching with 1182 *k* mismatches. *Journal of Algorithms, 50*(2), 257–275.
1183 2. Aoe, J.-I. (1989). An efficient implementation of static string pattern matching machines. 1184 *IEEE Transactions on Software Engineering, 15*(8), 1010–1016.
1185 3. Baeza-Yates, R. A., Perleberg, C. H. Fast and practical approximate string matching. In A. 1186 Apostolico, M. Crocchemore, Z. Galil, & U. Manber (Eds.), *Combinatorial pattern matching*, 1187 *lecture notes in computer science* (Vol. 644, pp. 185–192). Berlin: Springer.
1188 4. Baeza-Yates, R. A., & Régnier, M. Fast algorithms for two-dimensional and multiple pattern 1189 matching. In *Proceedings of the SWAT '90 the Second Scandinavian Workshop on Algorithm* 1190 *Theory* (pp. 332–347). Bergen, Sweden: Springer.

UNCORRECTED PROOF

Author Proof

1191 5. Boyer, R. S., & Moore, J. S. (1977). A fast string searching algorithm. *Communication of the*
1192 *ACM, 20*(10), 762–772.
1193 6. Knuth, D. E., Morris, J. H., & Pratt, V. R. (1977). Fast pattern matching in strings. *SIAM*
1194 *Journal on Computing, 6*(2), 323–350.
1195 7. Landau, G. M., & Vishkin, U. (1985). Efficient string matching in the presence of errors. In
1196 *Proceedings of the 26th Annual IEEE Symposium on Foundations of Computer Science*
1197 (pp. 126–136).
1198 8. Apostolico, A., & Giancarlo, R. (1986). The Boyer-Moore-Galil string searching strategies
1199 revisited. *SIAM Journal on Computing, 15*(1), 98–105.
1200 9. McCreight, E. M. (1976). A space-economical suffix tree construction algorithm. *Journal of*
1201 *the ACM, 23*(2), 262–272.
1202 10. Weiner, P. (1973). Linear pattern matching algorithm. In *Proceedings of the 14th IEEE*
1203 *Symposium on Switching and Automata Theory* (pp. 1–11).
1204 11. Manber, U., & Myers, E. W. (1990). Suffix arrays: a new method for on-line string searches.
1205 In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms* (pp. 319–
1206 327). Philadelphia, PA: SIAM.
1207 12. Burrows, M., & Wheeler, D. J. (1994). A block-sorting lossless data compression algorithm.
1208 13. Ferragina, P., & Manzini, G. (2000). Opportunistic data structures with applications. In
1209 *Proceedings of the 41st Annual Symposium on Foundations of Computer Science* (pp. 390–
1210 398). IEEE.
1211 14. Langmead, B. (2014, September). Introduction to the Burrows-Wheeler transform. www.
1212 youtube.com/watch?v=4n7NPk5lwbI.
1213 15. Aho, A. V., & Corasick, M. J. (1975). Efficient string matching: An aid to bibliographic
1214 search. *Communication of the ACM, 23*(1), 333–340.
1215 16. Commentz-Walter, B. (1979). A string matching algorithm fast on the average. In
1216 *Proceedings of the 6th Colloquium on Automata, Languages and Programming, 16–20*
1217 *July 1979*, pp. 118–132.
1218 17. Wu, S., & Manber, U. (1994). *A fast algorithm for multi-pattern searching*. Technical Report
1219 TR-94-17, Department of Computer Science, Chung-Cheng University.
1220 18. Crochemore, M., et al. (1999). Fast practical multi-pattern matching. *Information Processing*
1221 *Letters, 71,* 107–113.
1222 19. Colussi, L., Galil, Z., & Giancarlo, R. (1990). On the exact complexity of string matching. In
1223 *Proceedings of the 31st Annual IEEE Symposium of Foundation of Computer Science* (Vol. 1,
1224 pp. 135–144).
1225 20. Landau, G. M., & Vishkin, U. (1986). Efficient string matching with *k* mismatches.
1226 *Theoretical Computer Science, 43,* 239–249.
1227 21. Li, H., & Durbin, R. (2009). Fast and accurate short read alignment with Burrows-Wheeler
1228 transform. *Bioinformatics, 25*(14), 1754–1760.
1229 22. Eddy, S. R. (2004). What is dynamic programming? *Nature Biotechnology, 22,* 909–910.
1230 https://doi.org/10.1038/nbt0704-909.
1231 23. Chang, W. L., & Lampe, J. Theoretical and empirical comparisons of approximate string
1232 matching algorithms. In A. Apostolico, M. Crocchemore, Z. Galil, & U. Manber (Eds.),
1233 *Combinatorial pattern matching*. *Lecture notes in computer science* (Vol. 644, pp. 175–184).
1234 Berlin: Springer.
1235 24. Ukkonen, E. Approximate string-matching with *q*-grams and maximal matches. *Theoretical*
1236 *Computer Science, 92*, 191–211.
1237 25. Manber, U., & Baeza-Yates, R. A. (1991). An algorithm for string matching with a sequence
1238 of don't cares. *Information Processing Letters, 37,* 133–136.
1239 26. Pinter, R. Y. (1985). Efficient string matching with don't' care patterns. In A. Apostolico & Z.
1240 Galil (Eds.), *Combinatorial algorithms on words*. NATO ASI Series (Vol. F12, pp. 11–29).
1241 Berlin: Springer.
1242 27. Lecroq, T. (1992). A variation on the Boyer-Moore algorithm. *Theoretical Computer Science,*
1243 *92*(1), 119–144.

UNCORRECTED PROOF

Layout: T1 Standard STIX Book ID: 439206_1_En Book ISBN: 978-981-10-8475-1
Chapter No.: 12 Date: 17-3-2018 Time: 8:46 pm Page: 43/44

Author Proof

1244 28. Tarhio, J., & Ukkonen, E. Boyer-Moore approach to approximate string matching. In J. R. Gilbert & R. Karlssion (Eds.), *SWAT 90, Proceedings of the 2nd Scandinavian Workshop on Algorithm Theory*, *Lecture Notes in Computer Science* (Vol. 447, pp. 348–359). Berlin: Springer.

1248 29. Salmela, L., Tarhio, J., & Kytojoki, J. (2006). Multi-pattern string matching with q-grams. *ACM Journal of Experimental Algorithmics*, *11*.

1250 30. Li, H., & Durbin, R. (2010). Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinformatics, 26*(5), 589–595.

1252 31. Knuth, D. E. (1975). *The art of computer programming* (Vol. 3). Massachusetts: Addison-Wesley Publish Com.

1254 32. Li, H., & Homer. (2010). A survey of sequence alignment algorithms for next-generation sequencing. *Briefings in Bioinformatics*, *11*(5), 473–483. https://doi.org/10.1093/bib/bbq015.

1256 33. Karp, R. L., & Rabin, M. O. (1987). Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development, 31*(2), 249–260.

1258 34. Harrison, M. C. (1971). Implementation of the substring test by hashing. *Communication of the ACM, 14*(12), 777–779.

1260 35. Li, H., et al. (2008). Mapping short DNA sequencing reads and calling variants using mapping quality scores. *Genome Research, 18,* 1851–1858.

1262 36. Li, H. (2014). wgsim: a small tool for simulating sequence reads from a reference genome. https://github.com/lh3/wgsim/.

1264 37. Schatz, M. (2009). Cloudburst: Highly sensitive read mapping with mapreduce. *Bioinformatics, 25,* 1363–1369.

1266 38. Baeza-Yates, R. A., & Gonnet, G. H. (1989). A new approach to text searching. In N. J. Belkin & C. J. van Rijsbergen (Eds.), *SIGIR 89, Proceedings of the 12th Annual International ACM Conference on Research and Development in Information Retrieval* (pp. 168–175).

1270 39. Baeza-Yates, R. A., & Gonnet, G. H. (1992). A new approach in text searching. *Communication of the ACM, 35*(10), 74–82.

1272 40. Smith, A. D., et al. (2008). Using quality scores and longer reads improves accuracy of Solexa read mapping. *BMC Bioinformatics, 9,* 128.

1274 41. Tarhio, J., & Ukkonen, E. Approximate Boyer-Moore string matching. *SIAM Journal on Computing*, *22*(2), 243–260.

1276 42. Nicolas, M., & Rajasekarian, S. (2013). On string matching with *k* mismatches. https://arxiv.org/pdf/1307.1406.

1278 43. Cole, R., Gottlieb, L., & Lewenstein, M. (2004). Dictionary matching and indexing with errors and don't cares. In *STOC'04* (pp. 91–100).

1280 44. Hon, W., et al. (2007). A space and time efficient algorithm for constructing compressed suffix arrays. *Alrothmica, 48,* 23–36.

1282 45. Bauer, S., Schulz, M. H., & Robinson, P. N. (2014). gsuffix:http:://gsuffixSourceforge.net/.

1283 46. Lab website. (2014). http://home.cc.umanitoba.ca/~xiej/.

1284 47. Bolger, A. M., Lohse, M., & Usadel, B. (2014). Trimmomatic: bolger: A flexible trimmer for Illumina Sequence Data. Bioinformatics, btu170.

1286 48. Cunningham, F., et al. (2015). Nucleic Acids Research 2015, 43, Database issue: D662-D669.

1287 49. Dandass, Y. S., Burgess, S. C., Lawrence, M., & Bridges, S. M. (2008). Accelerating string set matching in FPGA hardware for bioinformatics research. *BMC Bioinformatics, 9,* 197.

1289 50. Ehrenfeucht, A., & Haussler, D. A new distance metric on strings computable in linear time. *Discrete Applied Mathematics*, *20*, 191–203.

1291 51. Galil, Z. (1977). On improving the worst case running time of the Boyer-Moore string searching algorithm. *Communication of the ACM, 22*(9), 505–508.

1293 52. Jiang, H., & Wong, W. H. (2008). SeqMap: Mapping massive amount of oligonucleotides to the genome. *Bioinformatics, 24,* 2395–2396.

1295 53. Kim, J. Y., & Yaylor, J. S. (1992). Fast multiple keyword searching. In *Proceedings of the Third Annual Symposium on Combinatorial Pattern Matching, 29 April–01 May 1992* (pp. 41–51). Springer.
1296
1297

UNCORRECTED PROOF

Layout: T1 Standard STIX Book ID: 439206_1_En Book ISBN: 978-981-10-8475-1
Chapter No.: 12 Date: 17-3-2018 Time: 8:46 pm Page: 44/44

Author Proof

1298 54. Li, R., et al. (2008). SOAP: short oligonucleotide alignment program. *Bioinformatics, 24,*
1299 713–714.
1300 55. Lin, H., et al. (2008). ZOOM! Zillions of oligos mapped. *Bioinformatics*, *24*, 2431–2437.
1301 56. Seward, J. (2007). bzip2 and libbzip2, version 1.0. 5: A program and library for data
1302 compression. http://www.bzip.org.
1303 57. Galil, Z. (1977). On improving the worst case running time of the Boyer-Moore string
1304 searching algorithm. *Communication of the ACM, 22*(9), 505–508.
1305 58. Chen, Y., Wu, Y., & Xie, J. (2016). An efficient algorithm for read matching in DNA
1306 databases. In *Proceedings of the International Conference on DBKDA'2016, Lisbon,*
1307 *Portugal, 26–30 June 2016* (pp. 23–34).
1308 59. Chen, Y., & Wu, Y. (2017). Mismatching trees and BWT arrays: A new way for string
1309 matching with k-mismatches. In *ICDE2017, 19–22 April 2017* (pp. 339–410). San Diego,
1310 USA: IEEE.

AQ1

UNCORRECTED PROOF

Author Proof

Author Query Form

Book ID : **439206_1_En**

Chapter No : **12**

Please ensure you fill out your response to the queries raised below and return this form along with your corrections.

Dear Author,
During the process of typesetting your chapter, the following queries have arisen. Please check your typeset proof carefully against the queries listed below and mark the necessary changes either directly on the proof/online grid or in the 'Author's response' area provided below

Query Refs.	Details Required	Author's Response
AQ1	References [49–59] are given in the list but not cited in the text. Please cite them in text or delete them from the list.	

UNCORRECTED PROOF

MARKED PROOF

Please correct and return this set

Please use the proof correction marks shown below for all alterations and corrections. If you wish to return your proof by fax you should ensure that all amendments are written clearly in dark ink and are made well within the page margins.

Instruction to printer	*Textual mark*	*Marginal mark*
Leave unchanged	• • • under matter to remain	Ⓘ
Insert in text the matter indicated in the margin	⋌	New matter followed by ⋌ or ⋌⊗
Delete	/ through single character, rule or underline or ⊢⊣ through all characters to be deleted	⁄ or ⁄⊗
Substitute character or substitute part of one or more word(s)	/ through letter or ⊢⊣ through characters	new character / or new characters /
Change to italics	— under matter to be changed	⌴
Change to capitals	≡ under matter to be changed	≡
Change to small capitals	= under matter to be changed	=
Change to bold type	~ under matter to be changed	~
Change to bold italic	≂ under matter to be changed	≂
Change to lower case	Encircle matter to be changed	≢
Change italic to upright type	(As above)	⊬
Change bold to non-bold type	(As above)	⊬
Insert 'superior' character	/ through character or ⋌ where required	Y or ⋌ under character e.g. Y or ⋌
Insert 'inferior' character	(As above)	⋌ over character e.g. ⋌
Insert full stop	(As above)	⊙
Insert comma	(As above)	,
Insert single quotation marks	(As above)	Y or ⋌ and/or Y or ⋌
Insert double quotation marks	(As above)	Y or ⋌ and/or Y or ⋌
Insert hyphen	(As above)	⊢⊣
Start new paragraph	⌐	⌐
No new paragraph	⤺	⤺
Transpose	⊔⊓	⊔⊓
Close up	linking ⁐ characters	⁐
Insert or substitute space between characters or words	/ through character or ⋌ where required	Y
Reduce space between characters or words	| between characters or words affected	⊤