STUDIA INFORMATICA 2012
Volume 33 Number 1 (104)

Artur KAJZER, Rafał POKRZYWA
Silesian University of Technology, Institute of Informatics

THE EXACT AND APPROXIMATE STRING MATCHING WITH BURROWS-WHEELER TRANSFORM

Summary. This paper describes how the problem of exact and approximate string matching can be resolved using Burrows-Wheeler Transform. The paper considers how compressed full-text indexes work and allow string matching algorithms to operate in a limited space. Two main data structures presented in the text are Wavelet Tree and Bi-directional Burrows Wheeler Transform.

Keywords: Burrows-Wheeler Transform, string matching

WYSZUKIWANIE DOKŁADNE I PRZYBLIŻONE CIĄGÓW TEKSTOWYCH Z WYKORZYSTANIEM TRANSFORMATY BURROWSA-WHEELERA

Streszczenie. Niniejszy artykuł przedstawia sposoby rozwiązywania problemu wyszukiwania dokładnego i przybliżonego ciągów tekstowych z wykorzystaniem transformaty Burrowsa-Wheelera. Omówione zostało także działanie skompresowanych indeksów pełnotekstowych, dzięki którym algorytmy wyszukiwania tekstu mogą działać w ograniczonej przestrzeni. Głównymi strukturami danych, które zostały opisane w tekście, są Wavelet Tree oraz Bi-directional Burrows-Wheeler Transform.

Słowa kluczowe: transformata Burrowsa-Wheelera, wyszukiwanie ciągów tekstowych

1. Introduction

The size of data used for example in software related with bioinformatics is very large. Very often fast tools for searching some pattern in such data are needed. This problem is defined as the exact string matching. The first algorithm that searches for the location of the first occurrence of the character string in another string was presented in [2]. In bioinformatics more common but also more complicated is an approximate string matching problem. In this problem one need to define some metric for measuring the amount of difference between two string sequences.

A very popular and common use of minimum string distance statistic for measuring error rates was described in [4]. One of the first papers that reviewed the approximate matching of strings with the aim of surveying techniques, suitable for finding an item in a database, when there may be a spelling mistake or another error in the keyword was [3]. Since 1980, when [3] was written, searching techniques have been repeatedly improved. Nowadays fast algorithms that solve this problem are known widely but if good performance is required on large data text preprocessing or some indexing techniques such as compressed full text index must be used. Indexing for approximate text searching is a problem receiving much attention because of its applications in signal processing, computational biology and text retrieval, to name a few. Description and help with essential features of indexing were presented in [17]. Very interesting and still developing group of indexes are the ones based on a block-sorting, lossless data compression algorithm called Burrows-Wheeler Transform. This paper is focused on them. Burrows-Wheeler Transform works by applying a reversible transformation to a block of input text. The transformation does not itself compress the data, but reorders it to make it easy to compress with simple algorithms such as move-to-front coding [7]. Burrows-Wheeler Transform is connected closely with data structure called suffix array. Constructing and querying suffix arrays are reduced to a sort and search paradigm. The main advantage of suffix arrays is that, in practice, they use less space than for example suffix trees. Suffix arrays are described in [1]. The Burrows-Wheeler Transform supports the design of efficient algorithms for solving various problems of the analysis of genomic sequences. For example Burrows-Wheeler Transform can be used to solve exact string matching problem. Some simple algorithms for solving this problem with discussion about time, their complexities are presented in [18]. Burrows-Wheeler Transform is used in popular application Bowtie. Bowtie is an ultrafast, memory-efficient program that aligns short DNA sequences (reads) to the reference genome. Bowtie indexes the genome with a Burrows-Wheeler Transform algorithm to keep its memory footprint small: typically about 2.2 GB for the human genome [16]. The application that efficiently aligns short sequencing reads against a large reference sequence such as the human genome, allowing mismatches and gaps is BWA. BWA is based on backward search with Burrows-Wheeler Transform [13]. Genomes of organisms contain a variety of repeated structures of various length and type, interspersed or tandem. Tandem repeats play important role in molecular biology as they are related to genetic backgrounds of inherited diseases, and also they can serve as markers for DNA mapping and DNA fingerprinting. A very efficient, web-based tool for large scale searching for exact tandem repeats in genomes, based on the use of the Burrows–Wheeler Transform was presented in [19].

2. Definitions

There is a string *X* which is terminated by a unique sentinel character (usually \$) smaller lexicographically than any symbol in *X*. For example *X* = agcagcagact\$. Alphabet of string is build by unique characters from string *X*. Alphabet interval is a range $[1..\sigma]$ of alphabet sorted lexicography (in example above : \$acgt), where $\sigma = |\sum|$

Any substring of string *X* can be denoted by $X[i..j]$, $0 \le i \le j \le n-1$, where $X[i..j]$ is the substring of length $j-i+1$ starting at position *i*.

For any integer $0 \le i \le n-1$, the substring $X[0..i]$ is a prefix of *X*. Similarly, for any integer $0 \le j \le n-1$ the substring $X[j..n-1]$ is a suffix of *X*. For example, *X* = aabc has prefixes a, aa, aab, aabc and suffixes c, bc, abc, aabc.

The suffix array of a string *X*, is an array of integers giving the starting positions of the suffixes of *X* sorted ascending in lexicographical order [1]. For example for string *X* = agcagcagact\$ suffix array *SA* is [11, 8, 6, 3, 0, 5, 2, 9, 7, 4, 1, 10]. Process of creating suffix array is presented in the picture below:

Suffixes :		**Sorted suffixes :**	
0	agcagcagact\$	11	\$
1	gcagcagact\$	8	act\$
2	cagcagact\$	6	agact\$
3	agcagact\$	3	agcagact\$
4	gcagact\$	0	agcagcagact\$
5	cagact\$	5	cagact\$
6	agact\$	2	cagcagact\$
7	gact\$	9	ct\$
8	act\$	7	gact\$
9	ct\$	4	gcagact\$
10	t\$	1	gcagcagact\$
11	\$	10	t\$

Fig. 1. Creating of suffix array
Rys. 1. Tworzenie tablicy sufiksowej

One of the most popular problem in computer science is an exact string matching problem. The exact string matching problem is defined as follows : there is a string *T* and a pattern *P*, where the length of the pattern string is always smaller or equal to the length of the text string. All occurrences of the pattern string within the text string should be found [2]. A more complicated issue is an approximate string matching problem. The approximate string matching problem is to find the text positions that match the pattern with at most *k* errors [3]. The number of positions at which the corresponding letters in pattern and text are different is naming Hamming Distance. In information theory some other metric for measuring the amount of difference between two sequences is often used – the Levenshtein distance. The Levenshtein distance between two strings is defined as the minimum number of characters that is needed to change, insert or delete to transform one string into the other one [4].

An edit distance can be calculated using dynamic programming [20]. Dynamic programming is a method of solving a large problem by regarding the problem as the sum of the solutions to solved subproblems. There are two strings *X* and *Y*, and it is needed to compute the edit distance between *X* and *Y*. A matrix $C_{0..|X|,0..|Y|}$ must be filled. $C_{i,j}$ represents the minimum number of operations needed to match $X_{1..i}$ to $Y_{1..j}$. This is computed as follows :

$$C_{i,0} = i$$
$$C_{0,j} = j$$
$$C_{i,j} = \text{if } (X_i = Y_j) \text{ then } C_{i-1,j-1}$$
$$\text{else } 1 + \min(C_{i-1,i}, C_{i,j-1}, C_{i-1,j-1}) \quad (1)$$

where at the end $C_{|X|,|Y|}$ is the edit distance between strings *X* and *Y*. The dynamic programming algorithm must fill the matrix in such a way that the upper, left, and upper-left neighbors of a cell are computed prior to computing that cell. Figure 2 illustrates the algorithm to compute the edit distance for string "survey" and "surgery" [12].

		s	**u**	**r**	**g**	**e**	**r**	**y**
	0	1	2	3	4	5	6	7
s	1	**0**	1	2	3	4	5	6
u	2	1	**0**	1	2	3	4	5
r	3	2	1	**0**	1	2	3	4
v	4	3	2	1	**1**	2	3	4
e	5	4	3	2	2	**1**	**2**	3
y	6	5	4	3	3	2	2	**2**

Fig. 2. Example of matrix to computer the edit distance between strings „survey" and „surgery"[12]
Rys. 2. Przykład macierzy służącej do wyliczania odległości edycyjnej między wyrazami "survey" i „surgery"[12]

3. Compressed indexes

In order to obtain high-speed search method in large text a variety of full –text indexes is used. Unfortunately, these indexes are characterized by the high space consumption. In recent years a new class of indexes has emerged. Compressed full-text indexes use data compression techniques to produce less space-demanding data structures. A compressed full-text index for a text *T* is a data structure requiring reduced space and able to search for patterns *P* in *T*. It can also reproduce any substring of *T*, thus actually replacing *T* [5]. Some compressed indexes are based directly on Burrows-Wheeler Transform or can be constructed efficiently from the Burrows-Wheeler Transform [6].

3.1. Burrows-Wheeler Transform

The Burrows-Wheeler Transform (BWT) is an computer algorithm that rearranges input text by some sorting operation. The output text contains exactly the same elements (like in input text) but in different order. It has to be known that the transformation is reversible. That means that the original input text can be reconstructed exactly [7]. Fast and easy way to obtain BWT is to calculate it from suffix array. BWT of string X with length n, denoted BWT$_X$, is a permutation of the symbols of X obtained from formula presented below [7]:

$$\forall i : 0 \le i \le n-1, BWT_X[i] = \begin{cases} X[SA_X[i]-1] & if\ SA_X[i] > 0 \\ \$ & if\ SA_X[i] = 0 \end{cases} \tag{2}$$

For example according to the formula (1) BWT of string X = agcagcagact$ with suffix array SA[11, 8, 6, 3, 0, 5, 2, 9, 7, 4, 1, 10] is BWT$_X$ = tgcc$ggaaaac.

The Burrows-Wheeler Transform can be used to search all the occurrences of the given pattern in a text, but in practice the compressed index should be used. One of the most popular compressed index is the Ferragina-Manzini index.

3.2. Ferragina-Manzini Index

Ferragina-Manzini Index (FM-Index) consists of some compressed representation of the transformer string together with some auxiliary information [8]:

- Special „rank" $C_X(a)$ for each unique character in string X which is a number of symbols in X that are lexicographically lower than the symbol a.
- "Checkpoints" taken from BWT for each unique character in string X. Checkpoint $Occ_X(a,i)$ is the number of occurrences of the symbol a in BWT from beginning of the BWT to the position i. (In a popular application Bowtie, these checkpoints are taken every 448 character) [9].
- Fragments of the suffix array. (Popular application Bowtie safes every 32 character from the suffix array) [9].

Since the suffix array is a sorted data structure, the start position of all the instances of a pattern Q in string X will occur in an interval in SA_X called suffix array interval. Suffix array interval is associate with a pair of integers $[l,u]$ denoting the first and last index in SA_X that correspond to a position in X of an instance of Q. According to the algorithms presented in the fig. 3, using Ferragina–Manzini Index of the original string X, the suffix array interval $[l,u]$ can be efficiently found. To search for a string Q, need to first calculate the interval for the last symbol of Q, then iteratively calculate the interval for the remainder of Q. The interval for a single symbol is simply calculated from C_X. If presented algorithm returns an interval where $l>u$, Q is not contained in X. Otherwise $SA_X[i]$ is the position in X of each occurrence of Q for $l \le i \le u$ [10].

```
Algorithm 1 : updateBackward([l,u],a)
    $l \leftarrow C_X(a)+Occ_X(a, l-1)$
    $l \leftarrow C_X(a)+Occ_X(a, u) -1$
    return $[l,u]$
```

```
Algorithm 2 : backwardSearch(Q)
    $i \leftarrow |Q|$
    $l \leftarrow C_X(Q[i])$
    $u \leftarrow C_X(Q[i]+1)-1$
    $i \leftarrow i-1$
    while $l \le u$ & $i \ge 1$ do
        $[l,u] \leftarrow$ updateBackward($[l,u]$, Q[$i$])
        i ← i-1
    end while
    return $[l,u]$
```

Fig. 3. Algorithm to find suffix array interval SA_X for pattern Q [10]
Rys. 3. Algorytm wyznaczający przedział tablicy sufiksowej SA_X dla wzorca Q [10]

Algorithm presented in the fig. 4 give the positions of each occurrence of pattern Q in string X, based on suffix array interval given by backward search algorithm, when only fragment of suffix array is knowing (like in Ferragina-Manzini Index).

```
Algorithm 3 : FindPatternPos(BWT, l, u)
i ← 1
while $i < u$ do
    ActualPos ← i
    ActualPos ← $C_X[BWT[ActualPos]] + Occ_X(BWT[ActualPos], ActualPos)$
    Count ← 0
    while ActualPos > 0 do
        ActualPos ← $C_X[BWT[ActualPos]] + Occ_X(BWT[ActualPos], ActualPos)$
        Count ← Count +1
    end while
    return Count
    $i \leftarrow i+1$
end while
```

Fig. 4. Algorithm to finding position of pattern Q in string X according to suffix array interval
Rys. 4. Algorytm wskazujący pozycje wystąpień wzorca Q w tekście X na podstawie podanego przedziału tablicy sufiksowej

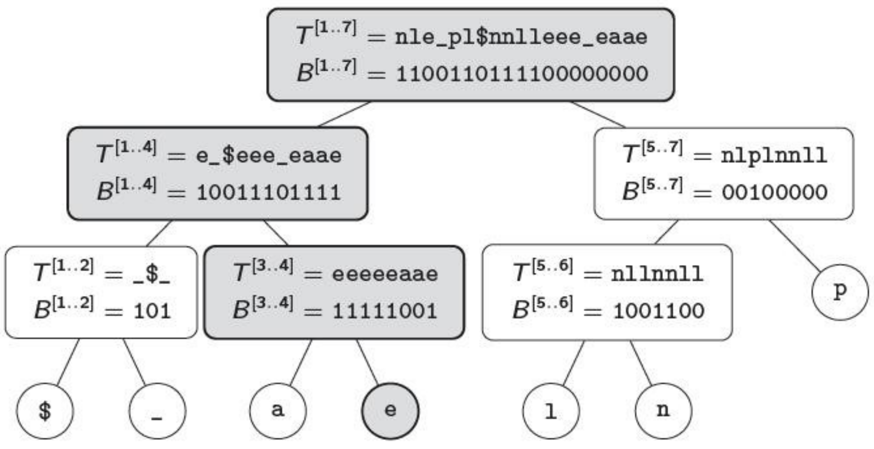
3.3. Wavelet Tree

Another type of compressed index can be built using Burrows-Wheeler Transform and data structure called Wavelet Tree. Wavelet Tree is a balanced binary search tree where :

- The root corresponds to the BWT
- Every node corresponds to some alphabet interval [l ..r]
- If for some node l<>r then node has two sub nodes. First one is corresponding to the BWT$^{[l..m]}$, the second one is corresponding to BWT$^{[m+1..r]}$ where $m = \left\lfloor \frac{l+r}{2} \right\rfloor$.

For the alphabet interval [l..r] BWT$^{[l..r]}$ is giving by deleting from BWT these characters, which aren't in alphabet interval [l..r] [11]. For example for BWT = tgcc$ggaaaac and the alphabet sorted lexicographically [$acgt], BWT$^{[1..3]}$ = cc$aaaac, because characters not corresponding to alphabet interval [1..3] are *g* and *t* and they must be deleted from BWT.

In practice the node isn't built by substrings from BWT but by bit vector $B^{[l..r]}$. In $B^{[l..r]}$ on the position *i* there is 0 when character *v* on this position is corresponding to the alphabet interval [l..m], and there is 1 in a other cases. For the wavelet tree C_X and Occ_X function can be defined in the same way like in Ferragina-Manzini index. C_X gives of course number of occurrences for bit 0 or 1 for some nodes of the tree. Figure 5 shows how Occ$_X$ function for wavelet tree works :



$$Occ(e,16) = Occ'(e,16,[1..7]) = Occ'(e, \underbrace{\mathrm{rank}_0(B^{[1..7]},16)}_{=8}, [1..4]) =$$

$$Occ'(e, \underbrace{\mathrm{rank}_1(B^{[1..4]},8)}_{=5}, [3..4]) = Occ'(e, \underbrace{\mathrm{rank}_1(B^{[3..4]},5)}_{=5}, [4..4]) = 5$$

Fig. 5. Principle of function *Occ* in wavelet tree [11]
Rys. 5. Zasada działania funkcji *Occ* z wykorzystaniem struktury wavelet tree [11]

The picture above shows how the number of occurrences of character e, from the start position, to the position 16, can be obtained for BWT = nle_pl$nnlleee_eaae. Bit vector B$^{[1..7]}$ corresponding to BWT is 1100110111100000000. Lexicographically sorted alphabet in this case is $_aelnp. Character *e* is in range [l..m] in alphabet so the number of occurrences for bit 0 from position 1 to 16 must be counted. It's 8 and calculations are continued in the left subnode. The alphabet for this node is $ael. For vector B[1..4] character e is in range [m+1, r] so then the number of occurrences for bit 1 from position 1 to 8 must be counted. It's 5 and calculations are continued in the right subnode. For vector B[3..4] character *e* is in range [*m*+1, *r*] so this time the number of occurrences for bit 1 from position 1 to 5 must be counted. It's 5 and this is the number of occurrences of character *e* in position from 1 to 16. This example shows that backward search procedure for wavelet tree can be implemented .

If a faster performance in string matching is required, forward search procedure can be implemented . There is some pattern *P* with the suffix array interval. The suffix array interval for pattern *Pc* needs to be calculated, where *c* is any character. For example one wants to calculate the suffix array interval for pattern *el* in string S = el_anele_lepanelen$, and it is known that interval for pattern *e* is equal to [6..11] (note that for any pattern all suffixes with prefixes *Pc* are also suffixes with prefix *P*, so the range for the *le* will be sub-range for the character *e*). Algorithm to calculate this was presented by Thomas Schnattinger in [11].

For actual interval [*i..j*] (in example above [6..1]) for vector $B^{[l..r]}$ (in example above $B^{[1..7]}$) interval $[a_0,b_0]$ and $[a_1,b_1]$ according to the following formulas must be calculated :

$$(a_0, b_0) = \text{rank}_0(B[l..r], i-1), \text{rank}_0(B[l..r], j))$$
$$(a_1, b_1) = \text{rank}_1(B[l..r], i-1), \text{rank}_1(B[l..r], j)) \qquad (3)$$

where $\text{rank}_b(B, i)$ is the number of occurrences of bit *b* in *B*[1..*i*] [11]

Another values are proceeding recursively (until reaching last subnode in wavelet tree), but if considered character (in example above *l*) is corresponding to the alphabet interval [*l..m*] in node then new values are equal $i = a_0 + 1, j = b_0$. In other cases $i = a_1 + 1, j = b_1$. This algorithm gives the interval (a_0,b_0) for suffixes *Pc* (where *c* is any character) and the interval (a_1,b_1) for suffixes *Pd* (where *d* is any character lexicographically greater than *c*).

For example :

STEP 1 :

$(a_0, b_0) = \text{rank}_0 (B[1..7], 6\text{-}1), \text{rank}_0 (B[1..7], 11)) = (2,3)$

$(a_1, b_1) = \text{rank}_1 (B[1..7], 6\text{-}1), \text{rank}_1 (B[1..7], 11)) = (3,8)$

l is corresponding to the alphabet interval [*m*+1, *r*] so $i = 4, j = 8$

STEP 2 :

$(a_0, b_0) = \text{rank}_0 (B[5..7], 4\text{-}1), \text{rank}_0 (B[5..7], 8)) = (2,7)$

$(a_1, b_1) = \text{rank}_1 (B[5..7], 4\text{-}1), \text{rank}_1 (B[5..7], 8)) = (1,1)$

l is corresponding to the alphabet interval [*l*, *m*] so $i = 3, j = 7$

STEP 3 :

$(a_0, b_0) = \text{rank}_0 (B[5..6], 3\text{-}1), \text{rank}_0 (B[5..6], 7)) = (1,4)$

$(a_1, b_1) = \text{rank}_1 (B[5..6], 3\text{-}1), \text{rank}_1 (B[5..6], 7)) = (1,3)$

It has been found that there are $b_0 - a_0 = 3$ occurrences of the character *l* and $b_1 - a_1 = 2$ occurrences of the any character lexicographically greater than *l*, so interval for pattern *le* is (6,9) [11].

4. Approximate pattern matching

Actually two main approaches to approximate string pattern matching are filtering and backtracking. Filtering algorithms filter the text and discard text areas that do not match. These algorithms are based on the fact that it may be much easier to tell that a text position does not match than to tell that it matches. It is important to notice that the filtering algorithm is normally unable to discover matching text positions by itself. Rather, it is used to discard areas of the text that cannot contain a match and must be coupled with a process that verifies all those text positions that could not be discarded by the filter [12].

The idea of filtration algorithm works as follows: if the text positions that match the pattern with at most *k* errors should be found algorithm must partition the pattern into *k*+1 pieces. At least one of the pieces can be found with no errors in any approximate occurrence of the pattern because *k* errors cannot alter all the *k*+1 pieces of the pattern, so at least one of the pieces must appear unaltered. The pattern is split into *k*+1 fragments and searches all them with a multipattern exact string matching algorithm. Each time a fragment of the pattern in the text is found, a neighborhood to determine if the complete pattern appears is verified [15].

A general limitation of filtration methods is that the performance of filtering algorithms is very sensitive to the error level. Most filters work very well on low error levels and very badly otherwise [12]. There is always a maximum error ratio α up to which they are useful, as for larger error levels the text areas verify and cover almost all the text [17].

Another type of algorithms backtrack on the suffix tree or array, simulates the sequential search over all the text suffixes, and take advantage of the factoring of similar substrings achieved by suffix trees or arrays [5]. The idea of backtracking algorithm over suffix tree was presented in [17]. This algorithm compute the edit distance between pattern and every text string that labels a path from tree root to a some tree node *N*. If the calculated edit distance is found to be smaller than maximum number of mismatches *k* then all the leaves of the current subtree are reported as the result of the approximate string matching [17].

In the backtracking algorithm over the suffix array calculating suffix array interval is like in the exact matching. The way through the pattern is from the last character to the first character. If the suffix array interval is empty the pattern at the current position is changed to other character or gap and increment number of mismatches. If the number of mismatches is equal *k* one position back is required in the pattern and another character should be tried. If all possibilities are used without finding a valid match, backtrack must be done. When the number of errors is bigger than *k* searching of course is broken [13].

The idea of searching pattern *Q* = ‘agc’ in string *T* = ‘acaacg” with backtracking algorithm is showed below. In the first step suffix array interval for last letter of pattern is calculated. This letter is ‘g’. In the second step suffix array interval for fragment of pattern equal ‘gc’ should be calculated but substring ‘gc’ does not occur in the text, so it is needed to change character ‘g’ to other character for example ‘a’. The interval for substring ‘ac’ is calculated and in the next step the interval for substring ‘aac’ which cover pattern with one mismatch should be calculated [9].

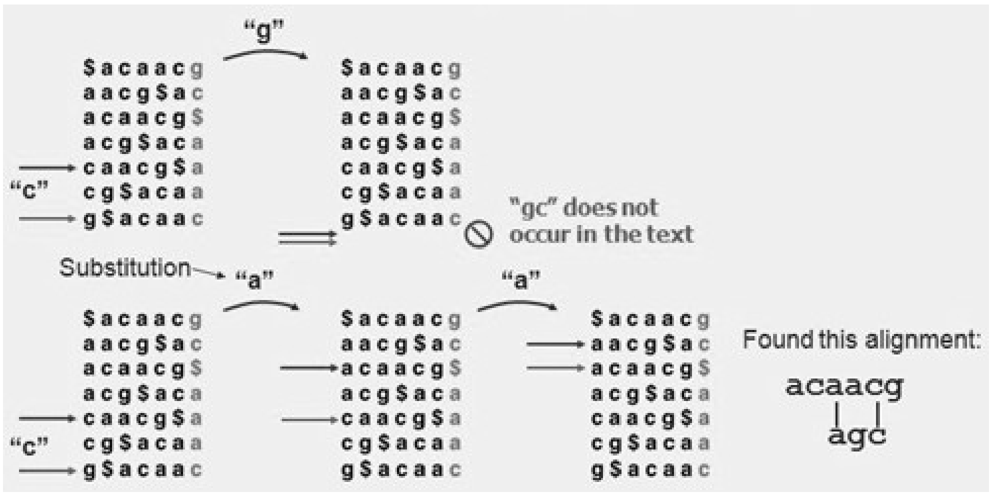


Fig. 6. Backtracking algorithm for approximate pattern matching for pattern Q = ‘agc’ and string T = ‘acaacg’ [9]
Rys. 6. Algorytm z nawrotami wykorzystany do wyszukiwania przybliżonego dla wzorca Q = ‘agc’ oraz tekstu T = ‘acaacg’ [9]

4.1. Simple approximate pattern matching algorithm

The simple approximate pattern matching algorithm using Ferragina-Manzini index was presented by Heng Li and Richard Durbin in [13]. This paper has presented a recursive algorithm to search for the suffix array intervals of substrings of string X that match the query string W with no more than z differences (mismatches or gaps). Essentially, this algorithm uses backward search. The way starts from the end of W and keeps moving towards the beginning of this string. Considering prefix can be each possible character. If match requires a base different from that in the real query, increment mismatch counts. Procedure stops when mismatch count exceeds z. To reduce search space, the backward search is bounded by the $D(.)$ array where $D(i)$ is the lower bound of the number of differences in $W[0,i]$. The better D is estimated, the smaller the search space and the more efficient the algorithm are. A naive bound is achieved by setting $D(i) = 0$ for all i, but the resulting algorithm is clearly exponential in the number of differences and would be less efficient [13].

4.2. Approximate string pattern matching with Bi-directional BWT

An interesting and fast algorithm is the algorithm that uses data structures called Bi-directional BWT described in [14]. Bi-directional BWT supports backward and forward search. To achieve this functionality BWT (denoted *B*) built for the string *T* must be stored, and another BWT (denoted *B*') for reversal of text *T*. The way to obtain interval of suffix array *SA* for pattern *Pc* (where *c* is any character) is similar like in wavelet tree. Exploiting *B*' to conduct backward searching of *Pc* with respect to reverse string *T* gives an interval of suffix array *SA*' for reverse pattern with respect to reverse string *T*. So it could be said that given pattern *P*, its *SA* range and *SA*' range, for any character *c*, returns the *SA* range and *SA*' range of *cP* , as well as the *SA* range and *SA*' range of *Pc* [14].

5. Conclusion

Today a lot of algorithms to solve the exact and approximate string matching problem are known. An interesting group of algorithms are the algorithms discussed in this paper, operating on compressed full text index based on Burrows-Wheeler Transform. These indexes reduce memory space consumption but are also able to solve presented problem in reasonable time. This allows to searching for some pattern on the standard PC even in large text, for example in human genome. Moreover indexes like Ferragina-Manzini index are very simple to implement. The strength of described algorithms is showed by the efficiency of popular application like Bowtie or BWA. The efficiency of the algorithms using backward search can be improved by using data structures like Bi-directional BWT or structures based on Wavelet-Tree that support backward and forward searching. This is probably a good way to improve tools for string matching which use Burrows-Wheeler Transform.

Acknowledgement

This work was supported by the European Union from the European Social Fund.

BIBLIOGRAPHY

1. Manber U., Myers G.: Suffix arrays: a new method for on-line string searches. Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms, 1991, p. 319÷327.
2. Boyer R. S., Moore J. S.: A fast string searching algorithm. Commun. ACM, Vol. 20, No. 10, 1977, p. 762÷772.
3. Hall P., Dowling G. R.: Approximate String Matching. ACM Comput. Surv., Vol. 12, No. 4, 1980, p 381÷402.

4. Levenshtein V.: Binary codes capable of correcting deletions, insertions, and reversals. Soviet Physics Doklady, Vol. 10, 1966, p. 707.
5. Russo L., Navarro G., Oliveira A., Morales P.: Approximate String Matching with Compressed Indexes. Algorithms, Vol. 2, No. 3, 2009, p. 1105÷1136.
6. Kärkkäinen J.: Fast BWT in Small Space by Blockwise Suffix Sorting. Theoretical Computer Science, Vol. 386, Issue 3, 2007, p. 249÷257.
7. Burrow M., Wheeler D.J.: A Block-sorting lossless data compression algorithm. Technical Report 124, Digital SRC Research Report, 1994.
8. Ferragina P., Manzini G.: An experimental study of a compressed index. Information Sciences: special issue on “Dictionary Based Compression”, 135, 2001, p. 13÷28.
9. Langmead B.: Bowtie: A Highly Scalable Tool for Post-Genomic Datasets. National Center for Biotechnology Information Seminar, November 10, 2008.
10. Simpson J., Durbin R.: Efficient construction of an assembly string graph using the FM-index. Bioinformatics, Vol. 26, No. 12, 2010, p. i367÷i373.
11. Schnattinger T, Ohlebusch E., Gog S.: Bidirectional Search in a String with Wavelet Trees. Combinatorial Pattern Matching, Vol. 6129, 2010, p. 40÷50.
12. Navarro G.: A guided tour to approximate string matching. ACM Comput. Surv., Vol. 33, No. 1, 2001, p. 31÷88.
13. Li H., Durbin R.: Fast and accurate short read alignment with Burrows-Wheeler transform. Bioinformatics (Oxford, England), Vol. 25, No. 14., 2009, p. 1754÷1760.
14. Lam T., Li R., Tam A., Wong S., Wu E., Yiu S.: High Throughput Short Read Alignment via Bi-directional BWT. IEEE International Conference on Bioinformatics and Biomedicine, 2009, p. 31÷36.
15. Wu S., U. Manber U.: Fast text searching allowing errors. Commun. ACM, Vol. 35, No. 10, 1992, p.83÷91.
16. Langmead B.: Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. Genome Biology, Vol. 10, No. 3, 10:R25, 2009.
17. Navarro G.: Indexing methods for Approximate String Matching. IEEE Data Engineering Bulletin, Vol. 24, No. 4, 2001, p. 19÷24.
18. Pokrzywa R., Polański A.: Exact string matching with Burrows-Wheeler transform. Krajowa Konferencja Zastosowań Matematyki w Biologii i Medycynie, Koninki, 26-29 September 2006.
19. Pokrzywa R., Polański A.: BWtrs: A tool for searching for tandem repeats in DNA sequences based on the Burrows-Wheeler transform. Genomics, vol. 96(5), 2010, p. 316÷321.

20. Sellers, Peter H.: The Theory and Computation of Evolutionary Distances: Pattern Recognition. Journal of Algorithms 1 (4), 1980, p. 359÷373.

Wpłynęło do Redakcji 8 listopada 2011 r.

Omówienie

W dziedzinach nauki, takich jak bioinformatyka niezbędne jest oprogramowanie wyszukujące pewne podane wzorce tekstowe w bardzo dużych zbiorach danych. Pożądane jest, by wykorzystywane w tym celu algorytmy rozwiązywały przedstawiony problem w bardzo krótkim czasie, najlepiej przy wykorzystaniu rozsądnych ilości pamięci operacyjnej.

W niniejszej pracy zostały przedstawione podstawowe, efektywne metody wyszukiwania ciągów tekstowych, których działanie bazuje na przekształcaniu tekstu źródłowego transformatą Burrowsa-Wheelera. Jako że wyszukiwanie oparte tylko i wyłącznie na wspomnianej transformacie nie jest wystarczająco efektywne, stosuje się różnorakie metody indeksacji tekstu. W artykule omówione zostały najbardziej popularne indeksy, takie jak na przykład Ferragina-Manzini Index.

W bioinformatyce w większości przypadków wykorzystuje się przybliżone wyszukiwanie wzorca. Oznacza to, że jest poszukiwana nie tylko dokładna postać wzorca, ale na przykład postać różniąca się na dwóch czy trzech pozycjach. W artykule przedstawiono podstawowe wiadomości na temat technik wyszukiwania przybliżonego korzystających z transformaty Burrowsa-Wheelera.

Addresses

Artur KAJZER: Silesian University of Technology, Institute of Informatics,
ul. Akademicka 16, 44-100 Gliwice, Polska, arturkajzer@o2.pl.
Rafał POKRZYWA: Silesian University of Technology, Institute of Informatics,
ul. Akademicka 16, 44-100 Gliwice, Polska, rafal.pokrzywa@polsl.pl.