Biocybernetics and Biomedical Engineering
2008, Volume 28, Number 1, pp. 95–104

Searching for Unique DNA Sequences with the Burrows-Wheeler Transform

RAFAŁ POKRZYWA*

Department of Computer Science, Silesian University of Technology, Gliwice, Poland

The objective of this study was to present an efficient algorithm that effectively aids the problem of searching for unique DNA sequences in the set of genes. The presented algorithm is based on the Burrows-Wheeler Transform (BWT), a very fast and effective data compression algorithm. The developed algorithm exploits all the advantages offered by the BWT algorithm and the suffix array data structure. It allows obtaining a structure that is ideal for solving many problems related to the pattern-matching problem. This algorithm is applicable to the identification of yeast species as well as to many other computational molecular biology problems like searching for repetitive structures in genomic sequences, designing of DNA hybridization probes and many more.

K e y w o r d s: Burrows-Wheeler Transform, yeasts species, suffix array, longest common prefix

1. Introduction

The existence of unique DNA sequences allows scientists identifying signature sequences that can be used to detect individual species. Unique DNA sequences are particularly important for diagnosis of the common viruses and diseases. Using these sequences radically simplifies detection and identification of yeast species, as traditional methods can take up to several weeks [1]. Unique DNA sequences are also applied in design primers for polymerase chain reaction (PCR) [2].

The identification of yeast species is conventionally based on morphological, physiological and biochemical characteristics such as their ability to utilize carbon and nitrogen compounds [3]. However, these identification methods are time-consuming and unsuitable for the detection of a mixture of organisms [1]. Hence, there is a need for methods based on the DNA sequence analysis. These methods make use of the

* Correspondence to: Rafał Pokrzywa, Department of Computer Science, Silesian University of Technology, Akademicka 16, 44-100 Gliwice, Poland, e-mail: rafal.pokrzywa@gmail.com

ribosomal DNA (rDNA) genes as a common target for the molecular identification of microorganisms.

It has been shown that most of the yeast species can be identified using a sequence divergence in the 26S rDNA gene [4]. Particularly there are two regions within the 26S rDNA gene (D1 and D2 domains), which are sufficient to deduce the relationships between the species [5]. The D1/D2 variable domain sequences for almost all known yeast species are entirely sequenced.

The D1 and D2 domains are approximately located within the first 650 bases in the 26S rDNA gene. These two regions show high divergence among 26S rDNA genes and therefore are ideal to identify yeasts species [4]. There is another yeast identification method that analyzes the internal transcribed spacer (ITS) region [3]. However, there are more sequence data for the D1/D2 regions of the 26S rDNA in the biological databases than for ITS data. In addition, the analysis of the D1/D2 domains has the advantage that it also facilitates phylogenetic analysis of the yeast [4].

This paper presents the application of the Burrows-Wheeler Transform algorithm to the yeast identification problem based on the D1/D2 domain analysis. Furthermore, the algorithm for solving this problem is presented and its time complexity is discussed. The paper also presents other problems related to text strings, important due to their applications for searching and processing data in bioinformatics databases.

2. Burrows-Wheeler Transform

The Burrows-Wheeler Transform (BWT) is a compression algorithm that rearranges the input text by sorting operation [6]. The output text contains exactly the same elements but in a different order. The transformation does not compress the input text but makes it much amenable to compression with standard techniques. The transformation is reversible, that means, the original input text can be exactly reconstructed.

The BWT is called “block-sorting” algorithm because it processes a block of input text as a single unit instead of processing the input sequentially. The main idea of the algorithm is to perform such a permutation of the input text elements that the same elements are grouped together. Therefore the resulted output text is much suitable to compression with other algorithms such as move-to-front coding [6].

2.1. BWT Algorithm

The algorithm takes an input of a string S of n characters and forms the n rotations of string S. Then the rotations are sorted in alphabetical order. The output string L is formed from the last characters of each of all the rotations. Because of the sorting we lose the information about the original string S. Therefore, the algorithm needs to compute the index I of the original string S in the sorted list of rotations. The output of the Burrows-Wheeler transform is the pair (L, I).

For the purpose of illustration suppose there is a string *S* = *“MISSISSIPPI”*. We form all the rotations of the string *S* (Fig. 1a) and then sort them alphabetically (Fig. 1b). The output string *L* containing the last characters from all the rotations in our example is *L* = *“PSSMIPISSII”*. Because the input string *S* occurs at position 4 in the sorted list the index *I* = 4. The output of the transformation is the pair *(L, I)* = *(“PSSMIPISSII”, 4)*.

a)	b)	
MISSISSIPPI	0	IMISSISSIPP
ISSISSIPPIM	1	IPPIMISSISS
SSISSIPPIMI	2	ISSIPPIMISS
SISSIPPIMIS	3	ISSISSIPPIM
ISSIPPIMISS	4	MISSISSIPPI
SSIPPIMISSI	5	PIMISSISSIP
SIPPIMISSIS	6	PPIMISSISSI
IPPIMISSISS	7	SIPPIMISSIS
PPIMISSISSI	8	SISSIPPIMIS
PIMISSISSIP	9	SSIPPIMISSI
IMISSISSIPP	10	SSISSIPPIMI

Fig. 1. The BWT compression algorithm: a) rotations of the string “MISSISSIPPI”; b) rotations sorted alphabetically

Having only the output of the Burrows-Wheeler Transform, it is possible to reconstruct the original input text. Cyclic rotations applied in the compression algorithm ensure that the last character in each of the rotations immediately precedes the first character in this rotation.

The string *L* containing last characters of all the rotations is the output of the BWT algorithm. The string *F* containing the first characters of all the rotations can be simply computed by the decoder, by sorting the characters of the string *L*. Based on the observation that each character in *L* is followed by the corresponding character in *F*, one can calculate the vector *T* indicating one-to-one correspondence between the column *L* and *F*, such as *F*[*i*] = *L*[[*T*[*i*]] (Fig. 2). Decoding the output of our example gives us the vector *T* = [4, 6, 9, 10, 3, 0, 5, 1, 2, 7, 8].

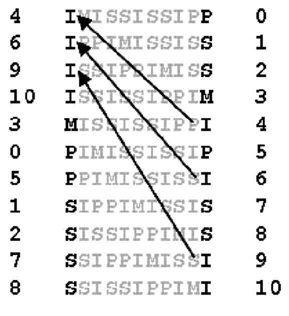

Fig. 2. Example of the Burrows-Wheeler Transformation decompression algorithm for the string “MISSISSIPPI”

Iterating the vector T we calculate the original input text S, in the way that for each $i = 0, ..., n-1$ the next character in the text is given by $L[T[i]]$. A simple algorithm can do this:

$$\forall\, \boldsymbol{i : 0 \leq i \leq n-1, S[i] = L[T^i[x]]} \tag{1}$$

where $T^0[x] = T[I]$, $T^{i+1}[x] = T[T^i[x]]$ and I is the index computed as the output from the BWT algorithm.

Sorting all the rotations of the input text has the greatest effect on the overall BWT algorithm performance. The most popular sorting algorithm – *quicksort* can be applied to most input data. However, its worst-case performance is poor. There is a need to find a better way to calculate the BWT output [7]. The problem of rotations' sorting can be simply reduced to the problem of sorting of all the suffixes of the input text [6].

To calculate the BWT output, firstly, one needs to form new text *S'* by adding "\$" at the end of the input text *S*, where the "\$" is a new character that does not appear in the *S*. Then we can apply compression algorithm to the text *S'* instead of *S* and sort only suffixes. This technique makes computing and keeping the index *I* of the original string *S* in the sorted list redundant. This index can be calculated anytime by simply finding the index of the character "\$" in the output string.

To achieve a better sorting performance Burrows and Wheeler suggested using a suffix tree. Building the suffix tree and then traversing it in the alphabetical order can solve the sorting problem. The suffix tree building time is proportional to the length of the input text, as well as, the time of traversing it [9]. There are also other data structures that can be effectively applied to the problem of computing the Burrows-Wheeler Transform, e.g. suffix array [7].

2.2. Suffix Array

The additional advantage of the Burrows-Wheeler Transform decompression algorithm is that it produces a list of all suffixes S_T of the input text S sorted alphabetically. We can simply refer to such list of sorted suffixes S_T by applying the decompression algorithm to the vector T for all indexes k within the range from 0 to $n-1$ [8].

This facilitates constructing a very simple algorithm that finds all the occurrences of a given pattern P in the input string S. If the pattern P occurs in S then all these occurrences would be grouped consecutively in the vector T. For example, the pattern P = *"ISSI"* occurs in the string S = *"MISSISSIPPI"* at adjacent positions 2 and 3 of the array S_T (Fig. 3).

To search for the occurrences of the pattern P of length m we can perform a binary search over the array S_T. First, we compare the pattern P with the suffix in the middle position of the S_T (suffix $S_T[n/2]$). If the pattern P is lexically less than the suffix, the pattern P must be in the first half of the array S_T. Similarly, if the pattern P is lexically greater than suffix, the pattern *P* must be in the second half of the array S_T. Making such comparisons in a similar manner, one can compute the smallest index *i* and the largest index *j* of the array S_T that the pattern *P* matches first *m* characters of suffix at position *i* and at position *j* respectively [9].

```
 4:   IMISSISSIPP
 6:   IPPIMISSISS
 9:   ISSIPPIMISS
10:   ISSISSIPPIM
 3:   MISSISSIPPI
 0:   PIMISSISSIP
 5:   PPIMISSISSI
 1:   SIPPIMISSIS
 2:   SISSIPPIMIS
 7:   SSIPPIMISSI
 8:   SSISSIPPIMI
```

Fig. 3. Suffix array for the string "MISSISSIPPI"

Using the binary searching over the vector *T* allows to find all of the occurrences of the pattern *P* in the string *S* in O*(m log n)* time. The real algorithm performance depends on how many prefixes of *P* occur in *T*. The O*(m log n)* bound is, in general, very pessimistic since it rarely happens that a specific comparison takes O*(m)* time. In the case of "random" strings, the binary search algorithm runs in O*(m* + log *n)* time on average [8].

3. Searching for Unique Substrings

The fact that all suffixes starting with the same prefix are located consecutively in the array S_T it allows simply checking whether a given pattern *P* is a unique substring within the input string S. If we localized pattern's position *k* in the vector *T* it's enough to compare the pattern with adjoining suffixes $S_T[k–1]$ and $S_T[k+1]$. If the pattern *P* is not the prefix of these suffixes then the pattern is a unique substring of string *S*.

The naive algorithm to compute all unique substrings within the input string *S* is to apply above mechanism to all the suffixes of the string *S* by simply traversing the S_T array from top to bottom. However, this method is simple to understand, its worst-case running time of $O(n^3)$ is unsatisfactory in most cases, especially in application to biological sequences (DNA, RNA, and protein).

First improvement to the naive algorithm is based on the observation that during the top to bottom search the adjacent suffixes are compared twice. In the iteration *k*, the algorithm compares the suffix $S_T[k]$ with the previous suffix $S_T[k–1]$ and the next suffix $S_T[k+1]$. In the next iteration *k*+1, the algorithm compares the suffix $S_T[k+1]$ with the previous suffix $S_T[k]$ and the next suffix $S_T[k+2]$. To avoid redundant comparison of the suffixes $S_T[k–1]$ and $S_T[k]$, one can store the result of the first comparison $S_T[k]$ with $S_T[k+1]$ (mismatch position) in the iteration k and use it in the next iteration $k+1$. Although the number of the comparisons has been radically reduced, the resulted computational complexity of $O(n^2)$ is still disappointing. The algorithm can be further improved and the $O(n^2)$ worst-case bound can be reduced to $O(n)$.

The main goal of the further improvement is to eliminate the redundant character examinations. To achieve such acceleration a new value has to be introduced. *Lcp(i, j)* is the length of the longest common prefix of two suffixes specified at positions i and j of the sorted list of suffixes $S_T[9]$. To speed up the naive algorithm we need a vector of *Lcp(i, i*+1*)* values for each suffix $S_T[i]$, where $0 < i < n–1$.

For the string S = *"MISSISSIPPI"* suffix $S_T[2]$ = *"ISSIPPI"*, suffix $S_T[3]$ = *"ISSISSIPPI"*, so *Lcp(*2, 3*)* is 4. The complete *Lcp* vector for the string S is *Lcp* = [1, 1, 4, 0, 0, 1, 0, 2, 1, 3, 0] (Fig. 4).

```
1   IMISSISSIPP
1   IPPIMISSISS
4   ISSIPPIMISS
0   ISSISSIPPIM
0   MISSISSIPPI
1   PIMISSISSIP
0   PPIMISSISSI
2   SIPPIMISSIS
1   SISSIPPIMIS
3   SSIPPIMISSI
0   SSISSIPPIMI
```

Fig. 4. The longest common prefix vector for the string "MISSISSIPPI"

The *Lcp* values can be effectively used to accelerate the lexical comparisons of the suffixes. Since *Lcp*[i] characters of suffix $S_T[i]$ are the same as the *Lcp*[i] characters of suffix $S_T[i+1]$, the lexical comparison of these two suffixes can begin from position *Lcp*[i] + 1 of the two strings, rather than starting from the first position. This allows to skip all redundant comparisons and run the searching unique substrings algorithm in linear time.

The *Lcp* vector can be obtained in the preprocessing phase, before starting the searching algorithm or alternatively, can be calculated online during character examinations by the searching unique substrings algorithm. In case of online construction of the vector *Lcp* we base on observation that the naive algorithm compare the same suffixes many times. After the comparison of two suffixes i and j is made we can store the *Lcp(i, j)* value in the *Lcp*[i] vector. Because algorithm compares different suffixes with random order (resulting from sorting of suffixes) such calculated *Lcp(i, j)* values may differ from *Lcp(i, i*+1*)* values. However those intermediate values would never be larger than real *Lcp* values because all suffixes are sorted lexicographically.

Therefore intermediate *Lcp* values can be effectively used to accelerate the naive algorithm.

4. Yeast Identification

From the algorithmic point of view the yeast identification problem based of the DNA sequences analysis can be defined in the following way. Given the set *Z* of the DNA sequences represented as strings over the alphabet $\sum = \{A, C, G, T\}$, for every sequence $\alpha \in Z$ find the shortest continuous subsequence α[i..j] that uniquely distinguished α in *Z*. Each of the unique subsequences may have different length.

J.-J. Wesselink et al. has presented an algorithm for identifying yeast species based on the D1/D2 domain analysis using the hashing technique [1]. I have applied my algorithm to the same set of the D1/D2 domain sequences (*http://www2.cmp.uea.ac.uk/~jjw/project/default.htm*) and compared the effectiveness of the two methods. All sequences as used by J.-J. Wesselink et al. were downloaded from the nucleotides database using the Entrez retrieval system (*http://www.ncbi.nlm.nih.gov/entrez*).

In the first step of the algorithm, the input string *S* for the Burrows-Wheeler Transform is created by the concatenation of all the 26S rDNA sequences separated by the character '#' (the new character that does not appear in any sequence). During this concatenation, the auxiliary array *C*, containing starting indices in the string *S* for each of *n* DNA sequences is calculated. This auxiliary array *C* is necessary to discriminate suffixes from the different DNA sequences. Because each of all the 26S rDNA sequences has approximately the same length, the origin of a given suffix can be determined with only one call to the auxiliary array *C*.

The calculation of the *BWT* for the string *S* can be easily accomplished based on the suffix array *SA*. From that, a vector *T* is calculated using the Burrows-Wheeler Transform decompression algorithm. Finally, the *LCP* vector is calculated using linear-time algorithm of T. Kasai et al.

Then, the algorithm searching for unique substrings that compares values from the *LCP* vector is applied. For each *k* from 1 to *n*, the algorithm compares the value *LCP*[*k*] with the value *LCP*[*h*], where *h* is the index of the suffix $S_T[h]$ which is next to the suffix $S_T[k]$ and comes from a different DNA sequence than the suffix $S_T[k]$. In most cases, those are simply successive indices. However, such a condition is necessary to ensure that the algorithm will not fail to find any substring which is unique within all the DNA sequences, not only within the string *S*. When comparing the *LCP* values, the algorithm makes use of the auxiliary array *C* and the suffix array *SA* to determine the origin DNA sequence and verify, if the likely unique substring does not overlap two different DNA sequences (does not contain '#' character).

The implementation of the algorithm is coded in Java programming language and run under Java Runtime Environment version 1.4.2. The hardware environment used consisted of AMD Duron 700 MHz processor, 512MB RAM, and Microsoft Windows XP operation system was used.

5. Results

The result of the top to the bottom comparisons of all suffixes sorted lexicographically is the list of all unique subsequences that unambiguously identifying the yeast species. Algorithm runs in O*(m)* time on average, where *m* is the length of all DNA sequences.

Table 1. Yeast species that can be identified with a unique sequence of up to length 6

Species name	Accession No.	Unique seq.
Ascoidea rubescens	U76195	ttacat
Brettanomyces custersianus	U76199	acgaat
Candida caseinolytica	U70250	tatgaa
Candida friedrichii	CFU45781	gtaaca
Candida gropengiesseri	CGU45721	caaacg
Candida haemulonii	CHU44812	caataa
Candida magnoliae	CMU45722	cacaaa
Candida savonica	CSU62307	ccataa
Candida spandovensis	CSU62309	acgtta
Colacogloea peniophorae	AF189898	cgctag
Metschnikowia	AF017401	taacgc
Pichia capsulata	U70178	attacg
Pichia naganishii	U75724	acatta
Saccharomyces transvaalensis	U68549	accata
Sporobolomyces sasicola	AF177412	atatat
Williopsis californica	U75957	actaat
Yarrowia lipolytica	U40080	tccaca

The result of searching the unique sequences shows that we are able to identify almost all of the examined yeast species (99.6%). The shortest unique sequences founded have length 6 bases (Table 1). With sequences of up to length 10 over 95% of yeast species could be identified. This proves that analysis of 26S rDNA gene is an effective way to identify yeast species.

Histogram presented in Figure 5 shows the number of unique sequences of up to length 8 founded at each position of 26S rDNA gene. We can observe huge diversity in the D1/D2 domains. There are three regions that are much conserved as there are only few unique sequences found. These regions are located in the first 50 positions, between positions 200 and 350 and again over position 570. There are also few variable regions like region around position 173.

J.-J. Wesselink et al. claim that theirs algorithm will outperform alternative techniques because access to hash table is much faster than traversing the suffix tree for example [1]. Performance comparison of the algorithms doesn't confirm that thesis. Although access to hashing table is really fast, the performance of hashing function is not such satisfactory especially for longer sequences. With the algorithm based on the Burrows-Wheeler Transform we are able to compute all unique sequences in linear time. Having *Lcp* values calculated we can determine whether given sequence is unique with just single comparison. We only need to find its position in the suffix array that can be easily done using binary searching for example.

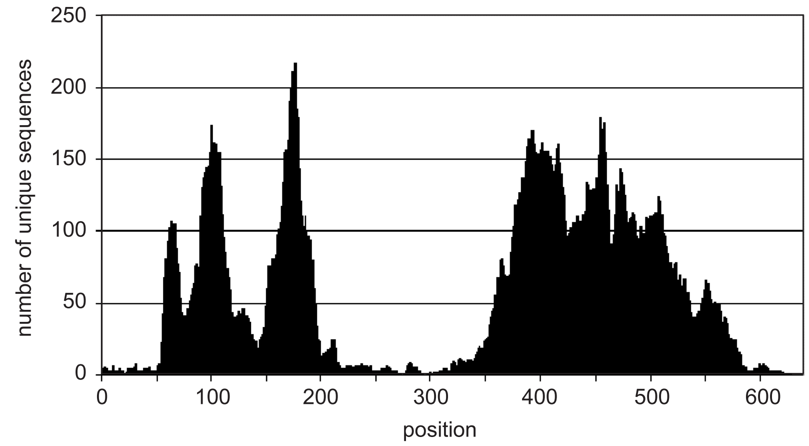

Fig. 5. Histogram presenting the number of unique sequences of up to length 8 founded at each position

6. Conclusion

This paper has presented the approach to the problem of searching unique sequences employing the Burrows-Wheeler Transform. In addition to its good compression advantages the BWT provides structures that are ideal for such searching problems. In particular, all algorithms based on the sorted list of suffixes can derive benefits from the BWT, especially in the area of computational biology.

J.-J. Wesselink et al. presented an algorithm for determining of unique defining sequences based on the method of hashing [1]. I have reanalyzed data and algorithm presented by J.-J. Wesselink et al. and compared the results of using their software to the methods discussed here. I have found algorithm based on the Burrows-Wheeler Transform, superior over the hashing technique proposed by J.-J. Wesselink et al. Additional advantage of my algorithm is that for some reason the top to bottom comparison of suffixes can by easily limited to the selected area. Furthermore, the algorithm can be stopped at any time (e.g. when interesting sequence is founded) and intermediate values can be used.

The sorted list of suffixes can be also effectively applied to solve more complicated problems like computing repetitive structures in the molecular strings (DNA, RNA, and protein). Repetitive structures play incredibly important role in the biological strings. There are many researches concerning repeated structures, theirs function and origin. Study of such structures requires extensive algorithm support. Finding repetitive structures in the molecular strings is, however, out of scope of this paper.

Acknowledgment

This work was partially financed by Committee for Scientific Research University grant BK – 209/Rau-2/06

References

1. Wesselink J.-J. at el.: Determining a unique defining DNA sequence for yeast species using hashing techniques, Bioinformatics, 2002, 18, 1004–1010.
2. Schuler G. D.: Sequence Mapping by Electronic PCR, Genome Research, 1997, 7, 541–550.
3. Sugita T., Nishikawa A.: Fungal Identification Method Based on DNA Sequence Analysis: Reassessment of the Methods of the Pharmaceutical Society of Japan and the Japan Pharmacopoeia, Journal of Health Science, 2003, 531–533.
4. Kurtzman C. P., Robnett C. J.: Identification and phylogeny of ascomycetous yeasts from analysis of nuclear large subunit (26S) ribosomal DBA partial sequence, Antonie Van Leeuwenhoek, 1998, 331–371.
5. Lachance M.A. at el.: The D1/D2 domain of the large-subunit rDNA of the yeast species Clavispora lusitaniae is unusually polymorphic, FEMS Yeast Research, 2003, 253–258.
6. Burrows M., Wheeler D.J.: A Block-sorting Lossless Data Compression Algorithm, Technical Report 124, Digital Equipment Corporation, 1994.
7. Deorowicz S.: Universal lossless data compression algorithms, Doctor of Philosophy Dissertation, Silesian University of Technology, 2003, 31–45.
8. Powell M.: Compressed-Domain Pattern Matching with the Burrows-Wheeler Transform, Honours Project Report, 2001, 7–14, 19–22.
9. Gusfield D.: Algorithms on Strings, Trees, and Sequences, Cambridge University Press, 1997, 151–154.