TVSBS: A fast exact pattern matching algorithm for biological sequences

Rahul Thathoo[1,3], Ashish Virmani[1,3], S. Sai Lakshmi[1,4], N. Balakrishnan[2] and K. Sekar[1,2,*]

[1]Bioinformatics Centre and [2]Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore 560 012, India
[3]Summer Trainees from IIIT, Allahabad 211 002, India
[4]Summer Trainee from University of Madras, Chennai 600 025, India

The post-genomic era is witnessing a remarkable increase in the number of nucleotide and amino acid sequences. The content of biological sequence databases almost doubles frequently. Pattern matching emerges as a powerful tool in locating nucleotide or amino acid sequence patterns in the biological sequence databases. Presently, several pattern-matching algorithms are available in the literature right from the basic Brute Force algorithm to the recent SSABS. The efficiency of the various algorithms depends on faster and exact identification of the pattern in the text. In this article, we propose an exact pattern-matching algorithm for biological sequences. The proposed algorithm, TVSBS, is a combination of Berry–Ravindran and SSABS algorithms. The performance of the new algorithm has been improved using the shift of Berry–Ravindran bad character table, which leads to lesser number of character comparisons. It works consistently well for both nucleotide and amino acid sequences. The proposed algorithm has been compared with the recent algorithm, SSABS. The results show the robustness of the proposed algorithm and thus it can be incorporated in any exact pattern-matching applications involving biological sequences. The best- and worst-case time complexities of the new algorithm are also outlined.

Keywords: Amino acids, character comparisons, exact pattern matching, nucleotides.

THE pattern-matching problem has attracted a lot of interest throughout the history of computer science, particularly in the present-day high performance computing and has been routinely used in various computer applications for several decades. To this end, several pattern-matching algorithms have been reported. These algorithms are applied in most of the operating systems, editors, search engines on the internet, retrieval of information (from text, image or sound) and searching nucleotide or amino acid sequence patterns in genome and protein sequence databases. Theoretical studies of different algorithms suggest various possible means by which these are likely to perform; but in some cases they fail to predict the actual performance. Here we demonstrate that better methods can be devised from theoretical analysis by extensive experimentation and modification of the existing algorithms.

Pattern matching can be defined as finding the occurrence of a particular pattern of characters in a large chunk of text. An exact pattern matching involves identification of all the occurrences of a given pattern of m characters ($x = x_1, x_2, \ldots, x_m$), in a text of n characters ($y = y_1, y_2, \ldots, y_n$), built over a finite alphabet set Σ of size s.

All pattern-matching algorithms scan the text with the help of a window, which is equal to the length of the pattern. The first process is to align the left ends of the window and the text, and then compare the corresponding characters of the window and the pattern. This process is known as an attempt. After a whole match or a mismatch of the pattern, the text window is shifted in the forward direction until the right end of the window reaches the end of the text. The algorithms vary in the order in which character comparisons are made and the distance by which the window is shifted on the text after each attempt.

Many pattern-matching algorithms are available with their own merits and demerits based on the pattern length, periodicity and alphabet set. One of the most viable approaches to this problem is to compare the text and the pattern in an effective pre-defined order. An efficient way is to move the pattern on the text using the best shift value. To this end, several algorithms have been proposed to get a better shift value, for example, Boyer–Moore[1], Quick Search[2] and Berry–Ravindran[3].

The efficiency of an algorithm lies in two phases: the pre-processing phase and the searching phase. The characters in the pattern are pre-processed in the pre-processing phase and this information is used in the searching phase in order to reduce the total number of character comparisons, which in turn minimizes the overall execution time. Effective searching phase can be established by altering the order of comparison of characters in each attempt and by choosing an optimum shift value that allows a maximum skip on the text. The difference between various algorithms is mainly due to the shifting procedure and the speed at which a mismatch is detected. We found that the pre-processing phase provided by the Berry–Ravindran algorithm and the searching phase provided by SSABS algorithm[4] are the best.

*For correspondence. (e-mail: sekar@serc.iisc.ernet.in)

Methods and algorithm

Survey on the existing algorithms

Pattern-matching algorithms can be categorized as single and multiple based on their functionalities. In addition, these algorithms are placed based on their average and worst-case time complexities.

The Boyer–Moore algorithm is widely used in the software industry. This algorithm uses Boyer–Moore bad character (bmBc) and Boyer–Moore good suffix (bmGs) tables to determine the number of shifts required to slide the window on the text, so that no match is left unconsidered. The maximum shift value from both the tables is considered for the shift after each attempt in the searching phase.

In Quick Search algorithm, the Quick Search bad character (qsBc) shift table is used to store the shift value of each character in the pattern. The shift value is given by the corresponding position of that character in the pattern from right to left. A shift value of (m + 1) is assigned to the characters which are not present in the pattern. In the searching phase, character comparisons between the text and the pattern can be done in any order.

Horspool algorithm[5] uses a shift value by finding the bad character shift for the rightmost character of the window. The shift value is computed in the pre-processing stage for all the characters in the alphabet set. Thus, the algorithm is effective in practical situations where the alphabet size is large and the length of the pattern is small.

In Raita algorithm[6], the order of comparison is modified to attain maximum efficiency. Here, the rightmost characters of the pattern and the window are compared, and on an exact match, the leftmost character of the pattern and that of the window are compared. If they match, the algorithm compares the middle characters of both the pattern and the window, and then the characters from the second to the last but one position of the pattern and the window are compared.

The Berry–Ravindran algorithm calculates the shift value by considering the bad character shift for two consecutive text characters (a substring) in the text immediately to the right of the window. The shift values are obtained from a two-dimensional array, computed in the pre-processing stage, based on Berry–Ravindran bad character function. This is the only algorithm which uses two successive characters to calculate the shift value of the pattern on the text. We found that this would reduce the number of character comparisons during the searching phase.

The SSABS algorithm calculates the skip of the window by the Quick Search bad character (qsBc) shift value for the character that follows the window immediately. The searching phase of this algorithm employs a new order of character comparisons, wherein the rightmost character of the window and that of the pattern are compared first and on finding a match, the characters on the leftmost end are compared. On finding a match, the remaining characters are compared from right to left until a complete match or a mismatch occurs. Earlier, we have demonstrated that the SSABS algorithm performs better than the other well-known algorithms described above. Thus, in the rest of the article, we confine ourselves with the SSABS algorithm for comparison.

The proposed algorithm

As pointed out earlier, for a better performance, one needs to implement an efficient way of pre-processing the pattern to get a better shift value. Secondly, good methodology should be employed in the searching phase. The proposed algorithm is a blend of Berry–Ravindran, and SSABS algorithms. The Berry–Ravindran bad character (hereafter, brBc) function is found to be effective during the pre-processing phase and the same has been implemented in the proposed algorithm with suitable modifications. The searching phase of this algorithm is exactly similar to that of the SSABS algorithm. The order of comparisons is carried out by comparing the last character of the window and that of the pattern first and once they match, the algorithm further compares the first character of the window and that of the pattern. This establishes an initial resemblance between the pattern and the window. The remaining characters are then compared from right to left until a complete match or a mismatch occurs. After each attempt, the skip of the window is gained by brBc shift value for the two consecutive characters immediately next to the window. The brBc function has been exploited to obtain the maximal shift and this reduces the number of character comparisons. These factors are collectively responsible for the improved performance of our algorithm.

Pre-processing phase

This is performed using brBc function, for all the characters in the alphabet set. This function provides maximum shift in most cases. Here, the algorithm considers two consecutive characters immediately after the window, whereas qsBc uses only one character immediately after the window. The pre-processing phase of the algorithm consists in computing for each pair of characters (a, b) for all a, b e Σ, the rightmost occurrence of ab in the pattern.

The bad character function can be described as:

$$\mathrm{brBc}[a,b] = \min \begin{cases} 1 & \text{if } x[m-1] = a, \\ m-i+1 & \text{if } x[i]x[i+1] = ab, \\ m+1 & \text{if } x[0] = b, \\ m+2 & \text{otherwise.} \end{cases}$$

For efficiency considerations, the shift values calculated using the brBc function are stored in a one-dimensional array instead of a two-dimensional array, so that they can be used readily and with less access time during the searching phase. The index of the one-dimensional array is computed using simple calculations involving bitwise operators. This considerably reduces the time taken to obtain the shift value after an attempt has been made, as the memory access time overhead has been done away with. The skip of the window is found by obtaining the shift value of the two consecutive characters immediately after the window and the maximum skip value for the window is realized when both these characters are not present in the pattern. The probability of a character occurring in the pattern becomes less when the alphabet size is large, and it helps to get the maximum skip of the window. In the proposed algorithm, we consider brBc over Quick Search bad character and Boyer–Moore bad character for the following reasons:

1. In qsBc, the shift value is assigned for a character immediately next to the window, say *a*, based on the rightmost occurrence of that character. However, brBc calculates the shift value based on the rightmost occurrence of two consecutive characters, say *ab*, where *b* is the character next to *a* in the pattern, outside the window. The probability of the rightmost occurrence of *ab* in the pattern as compared to that of *a*, is less. Therefore, brBc always provides a better shift than qsBc or utmost an equal shift is obtained.
2. brBc value is always defined to be ≥ 1, and hence this could work independently to implement a fast algorithm, while bmBc yields a shift value ≤ 0 in some cases, which requires the use of bmGs (Boyer–Moore good suffix) to calculate the skip of the window.

The pre-processing phase goes hand-in-hand with the searching phase to improve the overall efficiency of the algorithm by calculating larger shift values. The three stages of the searching phase are outlined in the subsequent sections.

Searching phase

Stages 1 and 2 deal with the order of character comparisons between the window and the pattern.

Stage 1: As previously mentioned, the searching phase begins with comparing the last character of the pattern with the last character of the window. If there is a match, the first character of the pattern is compared with that of the window. If both these characters match, the algorithm moves into the next stage; otherwise, it goes to the third stage.

Stage 2: In this stage, a sequential comparison is made, from the last but one character to the second character until a complete match or a mismatch occurs. If the entire characters match, then the corresponding position of the window on the text is displayed and the algorithm enters the third stage. In the case of a mismatch, the algorithm directly moves to the next stage.

Stage 3: This stage involves retrieval of the shift value corresponding to the two characters (placed immediately after the window) from the one-dimensional array generated during the pre-processing phase. The window is shifted from left to right based on this shift value.

All the three stages of the searching phase are repeated until the window is positioned beyond $n - m + 1$.

Implementation

Description

To reduce the time in obtaining the shift value corresponding to the two characters immediately next to the window, the values obtained from the brBc function are stored in a one-dimensional array. A simple method has been devised to get the index of this one-dimensional array. The function used to compute the index of the one-dimensional array is: $F(a, b) = ((a \ll 5) \wedge b)$, where *a* and *b* are the two characters placed immediately after the window. All the alphabets between ‘A’ and ‘Z’ will have ‘10’ at the two most significant bits. For example, the ASCII value of the alphabet ‘A’ is 65. The corresponding binary representation is 1000001. Similarly, the ASCII value of ‘Z’ is 90, with the corresponding binary representation being 1011010. Thus, least significant five bits are distinct for each character. This holds good for all the characters in the alphabet set of biological sequences.

On shifting the bits of *a* by five places to the left, the five least significant bits of the resultant number become zero. For example, shifting the bits of ‘A’ by five places, we get 0100000.

Therefore, the first operand of the bitwise XOR operator has zero at five least significant places and the second operand has two fixed most significant bits, with rest of the five bits being unique for each character. This is represented below:

Operand 1:	*xy*00000
Operand 2:	10*abcde*

where a, b, c, d, e, x, y e $\{0, 1\}$.

The XOR of these two binary numbers will always give a unique number, which can be used as the index of the one-dimensional array without any ambiguity.

Simple bitwise operators [for example, shift (<<) operator instead of multiplication] have been used to calculate the index efficiently, which is unique for a particular substring of exactly two characters.

Working example

The plant genome (*Arabidopsis thaliana*) consists of 27,242 gene sequences distributed over five chromosomes

(CHR_I to CHR_V) (NCBI site, ftp://ftp.ncbi.nih.gov/genomes/Arabidopsis_thaliana/CHR_I). Part of a nucleotide sequence of a gene (only 47 nucleotides) from Chromosome I (CHR_I) has been used (see below for details) to test the proposed algorithm.

Sequence in FASTA format: Part of the sequence considered for the test run:

ATCTAACATCATAACCCTAATTGGCAGAGAGAGA
ATCAATCGAATCA

This sequence has been taken from the gene index 32854 to 32901

>gi|22330780|ref|NC_003070.3| *Arabidopsis thaliana* chromosome 1, complete sequence

y = ATCTAACATCATAACCCTAATTGGCAGAGAGA
GAATCAATCGAATCA
x = GCAGAGAG
$n = 47$, $m = 8$

Pre-processing phase: The brBc function gives the shift values for $s = 4$ as shown in Table 1. The shift values are stored in a one-dimensional array in place of the two-dimensional array calculated using the brBc function. The above-mentioned function $[F(a, b) = ((a << 5) \wedge b)]$ is used to calculate the index of the array.

		AA		AG		GA		GT		TC	
		2145	….	2151	….	2209	….	2228	…	2755	….
…	….	10		2		1		1		10	

Searching phase – First attempt:

```
ATCTAACATCATAACCCTAATTGGCAGAGAGAGAATCAATCGAATCA
       1
GCAGAGAG
```

shift = brBc[T][C] = 10.

In the first attempt, the last characters of the pattern and the window are compared. Since there is a mismatch, the window is moved based on brBc shift value corresponding to (T, C), which is equal to 10.

Table 1. Shift values for $s = 4$ given by the brBc function

brBc	A	C	G	T	*
A	10	10	2	10	10
C	7	10	9	10	10
G	1	1	1	1	1
T	10	10	9	10	10
*	10	10	9	10	10

Second attempt:

```
ATCTAACATCATAACCCTAATTGGCAGAGAGAGAATCAATCGAATCA
                         1
          GCAGAGAG
```

shift = brBc[A][A] = 10.

Once again, comparison of last characters of the pattern and the window leads to a mismatch, so the window is shifted by 10.

Third attempt:

```
ATCTAACATCATAACCCTAATTGGCAGAGAGAGAATCAATCGAATCA
                           1
                    GCAGAGAG
```

shift = brBc[G][A] = 1

In this attempt also, mismatch occurs between the last characters of the pattern and the window. Therefore, the window is shifted by one.

Fourth attempt:

```
ATCTAACATCATAACCCTAATTGGCAGAGAGAGAATCAATCGAATCA
                     2       1
                     GCAGAGAGG
```

shift = brBc[G][T] = 1

Here, comparison of last characters of the pattern and the window is carried out which happens to be a match. Therefore, the first character of the pattern and that of the window are compared and in the event of a mismatch, the window is shifted by one.

Fifth attempt:

```
ATCTAACATCATAACCCTAATTG GCAGAGAG AGAATCAATCGAATCA
                         28765431
                        GCAGAGAG
```

shift = brBc[A][G] = 2

In this case, the given pattern completely matches with the window and the comparison is done as follows: First, the last characters of the pattern and the window are compared, followed by the first, and then in the right to left manner. Then the window is moved based on the shift value of ($y[j + m]$, $y[j + m + 1]$), which is equal to two.

Sixth attempt:

```
ATCTAACATCATAACCCTAATTGGCAGAGAGAGAATCAATCGAATCA
                         2       1
                         GCAGAGAG
```

shift = brBc[A][A] = 10

Once again, comparison of last characters of the pattern and the window leads to a mismatch; so the window is shifted by 10.

Seventh attempt:

```
ATCTAACATCATAACCCTAATTGGCAGAGAGAGAATCAATCGAATCA
                                           1
                                   GCAGAGAGG
```

Total number of attempts: 7
Total number of character comparisons: 16

As shown above, comparison of the last characters of the pattern and the window fails. The shift value is calculated, but the window is not shifted because it goes beyond the right end of the text.

The number of attempts and the corresponding character comparisons during the searching phase for SSABS and the present algorithm are given in Table 2. The values taken by the proposed algorithm (TVSBS) depict the efficiency of the methodology deployed.

Analysis

The pre-processing phase time complexity of the proposed algorithm is $O(s + k*s)$ and space complexity is $O(s + k*s)$. The following section describes the time complexity of the searching phase.

Lemma 4.1

The time complexity is $O([n/(m + 2)])$ in the best case.

Proof: The best case occurs when all the characters in the pattern are completely different from those in the text. Every two characters that neither occur in the first or last position of the pattern nor appear consecutively in the pattern have a shift $(m + 2)$ as defined by brBc. So, in the best case, at each attempt we get a shift of $(m + 2)$, and hence the time complexity is $O([n/(m + 2)])$.

Example 1:
Text: aaaaaaaaaaaaaaaaaaaaaaaa
Pattern: bbbbbb

Table 2. Number of attempts and corresponding character comparisons during the searching phase for the pattern considered in the working example

Algorithm	SSABS	TVSBS[#]
Attempts	9	7
Comparisons	19	16

[#]TVSBS (Thathoo–Virmani–Sai Lakshmi–Balakrishnan–Sekar) algorithm proposed in the present article.

Lemma 4.2
The time complexity is $O(m(n - m + 1))$ in the worst case.

Proof: The worst case occurs when all the characters are matched at each attempt. As every character of the text is matched no more than m times, the total character comparisons for n characters of the text cannot exceed $m(n - m + 1)$ and hence the time complexity is $O(m(n - m + 1))$. The worst case can be realized when all the characters in the pattern are the same as those in the text.

Example 2:
Text: aaaaaaaaaaaaaaaaaaaaaaaa
Pattern: aaaaa

The alphabet size and probability of occurrence of each individual character in the text are the key factors which define the average time complexity. Since both these factors are highly random and in the absence of any reliable prediction mechanism, we admit that the average time complexity cannot be strictly defined.

Results and discussion

As has been stated above, the SSABS algorithm was used for comparison with the proposed algorithm. Two types of data have been analysed for comparisons; one with small alphabet size, i.e. $s = 4$ (nucleotide sequences) and another with big alphabet size, i.e. $s = 20$ (amino acid sequences). For testing and execution purposes, we have used a 3.06 GHz processor with 512 KB of cache memory and 1 GB of RD-RAM. The 'cc' compiler was used to compile the source code and all the programs have been executed on a single user mode to make sure that the results are more reliable and consistent. The source code for all the known algorithms was taken from the literature[7].

Case study with nucleotide sequences

A total of 837 gene sequences (comprising of nucleotides, 826.31 MB size) have been deployed to prove the power of the proposed algorithm. The dataset contains four characters (nucleotides), viz. A – (adenine, 239490165), C – (cytosine, 183940124), G – (guanine, 183818044) and T – (thymine, 239419854) and hence, the alphabet size is equal to four ($s = 4$). In order to avoid bias in the result, the calculation has been carried out for twenty different randomly generated patterns of each pattern length. The same procedure is adopted for different pattern lengths. The number of character comparisons is significantly reduced as shown in Table 3. The average time (10^{-2} s) taken is listed in Table 4, along with standard deviations (within the parentheses). It is evident from Table 4 that the average time taken by the proposed algorithm is low compared to SSABS. Since the alphabet size is four, the maximum number of randomly generated unique patterns for pattern length two is 16 and hence this pattern length is not considered for comparison.

Case study with amino acid sequences

Here the second type of data involving amino acid residues with larger alphabet size (s = 20) is considered. This case study uses 453,861 gene sequences (191.24 MB). In this case, the alphabet set used is $\sum$ = (A (13100890), C (1839722), D (8295604), E (9841468), F (6335049), G (10713539), H (3349835), I (9562897), K (8668206), L (15356872), M (3715491), N (6697619), P (6900621), Q (5838973), R (8414478), S (10200603), T (8319861), V (10559951), W (1837371), Y (4820702)). As stated in the previous case, the computation is carried out for twenty randomly selected patterns for each pattern length. The striking feature in the proposed algorithm is the reduction in the number of character comparisons by a significant amount (Table 5). The average time taken by the algorithms (SSABS and TVSBS) is given in Table 6. It clearly shows that the proposed algorithm is better compared to SSABS. The time taken to search for patterns of different lengths of amino acid sequences (s = 20) is lower than the corresponding lengths in case of nucleotide sequences (s = 4). This is because the amino acid sequence database (191.24 MB) is small compared to that of the nucleotide sequence database (826.31 MB). To conclude, the proposed algorithm performs better irrespective of the alphabet size.

Table 3. Number of character comparisons for all the pattern lengths of nucleotide sequences. Alphabet size is 4 (s = 4)

	No. of comparisons	
Pattern length	SSABS	TVSBS
4	402747713	399203580
6	354503266	332329966
8	241172024	240817934
10	186193732	178419433
12	343804578	231213903
14	217742543	145538214
16	295020591	151305588
18	377874422	211093580
20	313351911	172078077
22	306987726	154879563
24	310491940	142126547
26	253631165	130124659
28	261047436	136954217
30	226471194	123659847

Table 4. Comparison of average time taken by SSABS and TVSBS. Alphabet size is four

	Average time (in 10^{-2} s)	
Pattern length	SSABS	TVSBS
4	1131(56)	1038(61)
6	1050(52)	975(70)
8	1050(83)	978(72)
10	988(57)	948(67)
12	1024(70)	999(84)
14	1000(81)	956(86)
16	990(68)	993(116)
18	1038(212)	961(86)
20	1091(274)	983(93)
22	998(114)	980(85)
24	1010(55)	996(87)
26	1012(80)	981(77)
28	989(66)	979(69)
30	1007(117)	976(90)

Table 5. Number of character comparisons for amino acid sequences. Alphabet size is 20 (s = 20)

	No. of comparisons	
Pattern length	SSABS	TVSBS
2	52158161	47081091
4	34987965	32071044
6	24859972	21446441
8	20564524	16556578
10	17458459	14578492
12	16148712	13658741
14	15011247	13854781
16	13966587	9885574
18	14002115	10114544
20	11122457	8145547
22	12254466	7844548
24	11311364	7122458
26	10233473	6233465
28	9655421	5366984
30	8564471	4984654

Table 6. Comparison of average time taken by the algorithms SSABS and TVSBS. Alphabet size is 20 (s = 20)

	Average time (in 10^{-2} s)	
Pattern length	SSABS	TVSBS
2	141(4)	138(3)
4	126(2)	125(2)
6	121(2)	120(2)
8	120(3)	118(1)
10	118(2)	116(1)
12	115(1)	113(1)
14	114(1)	113(0)
16	113(1)	112(1)
18	112(0)	111(1)
20	112(1)	110(0)
22	112(0)	109(2)
24	111(1)	109(0)
26	111(0)	109(1)
28	110(0)	108(0)
30	110(1)	108(1)

Conclusion

In this article, an algorithm has been proposed for exact pattern matching, wherein the shift value is maximized using the brBc function. The database used to validate the proposed algorithm is sufficiently large and hence, the proposed algorithm is relatively faster. It is noteworthy that the average time taken by the proposed algorithm is less compared to SSABS. Hence, this procedure can possibly be implemented in all applications related to exact pattern matching in biological sequence databases.

1. Boyer, R. S. and Moore, J. S., A fast string searching algorithm. *Commun. ACM*, 1977, **20**, 762–772.
2. Sunday, D. M., A very fast substring search algorithm. *Commun. ACM*, 1990, **33**, 132–142.
3. Berry, T. and Ravindran, S., A fast string matching algorithm and experimental results. In Proceedings of the Prague Stringology Club Workshop '99 (eds Holub, J. and Simánek, M.), Collaborative Report DC-99-05, Czech Technical University, Prague, Czech Republic, 2001, pp. 16–26.
4. Sheik, S. S., Aggarwal, S. K., Poddar, A., Balakrishnan, N. and Sekar, K., A FAST pattern matching algorithm. *J. Chem. Inf. Comput. Sci.*, 2004, **44**, 1251–1256.
5. Horspool, R. N., Practical fast searching in strings. *Software – Practice Experience*, 1980, **10**, 501–506.
6. Raita, T., Tuning the Boyer–Moore–Horspool string-searching algorithm. *Software – Practice Experience*, 1992, **22**, 879–884.
7. Charras, C. and Lecroq, T., *Handbook of Exact String Matching algorithms* (available at the website: http://www-igm.univ-mlv.fr/~lecroq/string/).

ACKNOWLEDGEMENTS. We thank the Bioinformatics Centre (DIC), the Interactive Graphics Based Molecular Modelling Facility (IGBMM) and the Supercomputer Education and Research Centre (SERC), IISc, Bangalore for support. This work is completely supported by the Institute-wide Computational Genomics Project supported by the Department of Biotechnology, New Delhi. R.T., A.V. and S.S. thank the Bioinformatics Centre for providing summer internship under K.S. during the summer of 2004. We also thank Mr M. N. A. Md. Roshan for help.

Received 11 October 2005; revised accepted 18 February 2006