A FAST Pattern Matching Algorithm

S. S. Sheik,[†] Sumit K. Aggarwal,[†] Anindya Poddar,[‡] N. Balakrishnan,[‡] and K. Sekar*[,†,‡]

Bioinformatics Centre and Supercomputer Education and Research Centre, Indian Institute of Science,
Bangalore 560 012, India

Received June 18, 2003

The advent of digital computers has made the routine use of pattern-matching possible in various applications. This has also stimulated the development of many algorithms. In this paper, we propose a new algorithm that offers improved performance compared to those reported in the literature so far. The new algorithm has been evolved after analyzing the well-known algorithms such as Boyer-Moore, Quick-search, Raita, and Horspool. The overall performance of the proposed algorithm has been improved using the shift provided by the Quick-search bad-character and by defining a fixed order of comparison. These result in the reduction of the character comparison effort at each attempt. The best- and the worst- case time complexities are also presented in this paper. Most importantly, the proposed method has been compared with the other widely used algorithms. It is interesting to note that the new algorithm works consistently better for any alphabet size.

1. INTRODUCTION

Over the years, pattern-matching has been routinely used in various computer applications, for example, in editors, retrieval of information (from text, image, or sound), and searching nucleotide or amino acid sequence patterns in genome and protein sequence databases. The present day pattern-matching algorithms match the pattern exactly or approximately within the text. An exact pattern-matching is to find all the occurrences of a particular pattern ($x = x_1\ x_2\ ...\ x_m$) of m-characters in a text ($y = y_1\ y_2\ ...\ y_n$) of n-characters which are built over a finite set of characters of an alphabet set denoted by Σ and the size of this set is equal to σ.

The direct way to this problem is to compare the first m-characters of the text and the pattern in some predefined order and, after a match or a mismatch, slide the entire pattern by one character in the forward direction of the text. This process is repeated until the pattern is positioned at the ($n-m+1$) position of the text. This approach is commonly known as a brute-force method. To facilitate this task, several algorithms have been proposed, and these have their own advantages and limitations based on the pattern length, periodicity, and the type of the text (for e.g., nucleotide or amino acid sequences or language characters, etc.). Most of the well-known algorithms (see below for details) work in two phases: i.e., the preprocessing phase and the search phase. In the preprocessing phase, these algorithms process the pattern and use this information in the search phase to reduce the total number of character comparisons and hence reduce the overall execution time. The efficiency of an algorithm mainly depends on the search phase. The main objective behind the pattern-matching algorithms is to reduce the total number of character comparisons between the pattern and the text to increase the overall efficiency. The improvement in the efficiency of a search can be achieved by altering the order in which the characters are compared at each attempt and by choosing a shift factor that permits the skipping of a predefined number of characters in the text after each attempt.

Pattern-matching algorithms scan the text with the help of a window, whose size is equal to the length of the pattern. The first step 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 procedure is known as attempt. After a match or a mismatch of the pattern, the text window is shifted to the right. The question is how many characters are required to shift the window on the text. This shift value varies based on the methodology used by various algorithms. This procedure is repeated until the right end of the window is within the right end of the text.

2. SURVEY ON THE EXISTING ALGORITHMS

The algorithms reported in the literature are ranked based on their average-case and worst-case time complexities. The Boyer-Moore[1] algorithm and its variants are widely used in the software industry. The algorithm, Quick-search,[2] performs better, when the pattern length is small and the alphabet size is large (which is true in most of the practical situations). In the Raita[3] algorithm, the order of the comparison is modified to attain maximum efficiency. The Horspool[4] algorithm performs the comparison in a simple way, which works for most of the practical cases. The algorithms are further discussed in detail in the subsequent sections.

The Boyer-Moore Algorithm (BM). Theoretically, the Boyer-Moore[1] algorithm is one of the efficient algorithms compared to the other algorithms available in the literature. The algorithm preprocesses the pattern and creates two tables, which are known as Boyer-Moore bad character (bmBc) and Boyer-Moore good-suffix (bmGs) tables. For each character

* Corresponding author phone: +91-080-23601409 and +91-080-22933059; fax: +91-080-23600683 and +91-080-23600551; e-mail: sekar@physics.iisc.ernet.in.

[†] Bioinformatics Centre.

[‡] Supercomputer Education and Research Centre.

10.1021/ci030463z CCC: $27.50 © 2004 American Chemical Society
Published on Web 05/13/2004

in the alphabet set, a bad-character table stores the shift value based on the occurrence of the character in the pattern. On the other hand, a good-suffix table stores the matching shift value for each character in the pattern. The maximum of the shift value between the bmBc (character in the text due to which a mismatch occurred) dependent expression and from the bmGs table for a matching suffix is considered after each attempt, during the searching phase. This algorithm forms the basis for several pattern-matching algorithms.

The Quick-Search Algorithm (QS). The Quick-search[2] algorithm uses the Quick-search bad-character (qsBc) shift table, generated during the preprocessing stage. The shift value for a character in the qsBc table is defined as its corresponding position in the pattern from right to left order. If the character is not present in the pattern, then the shift value is equal to m+1. After an attempt, when the window is positioned on y[j.. j+m-1], the length of the shift is at least equal to one. Therefore, the character y[j+m] is necessarily involved in the next attempt and is used for the bad-character shift of the current attempt. During each attempt of the searching phase, the comparisons between the pattern and the text characters can be performed in any order.

The Horspool Algorithm (HORSPOOL). The Horspool[4] algorithm is a derivative of Boyer-Moore[1] and is easy to implement. When the alphabet size is large and the length of the pattern is small, it is not efficient to use Boyer-Moore's bad-character technique. Instead, it is always enough to find the bad-character shift of the right-most character of the window to compute the value of the shift. These shift values are computed in the preprocessing stage for all the characters in the alphabet set. Hence, the algorithm is more efficient in practical situations where the alphabet size is large and the length of the pattern is small.

The Raita Algorithm (RAITA). Raita[3] designed an algorithm in which the order of character comparisons has been changed to attain maximum efficiency. First, the rightmost character of the pattern and the window are compared, and on a match, the leftmost character of the pattern and the window are compared. If they match, it compares the middle character of both the pattern and the window. Second, if they match, it compares the characters from the second to the penultimate (n-1) position of the pattern and the window. The skip for the window is computed by applying the bmBc (computed during the preprocessing phase) shift of the rightmost character in the window.

3. THE PROPOSED ALGORITHM

The idea behind the proposed algorithm is as follows. The order of comparisons is carried out by comparing the last character of the window and the pattern, and after a match, the algorithm further compares the first character of the window and the pattern. By doing so, an initial resemblance can be established between the pattern and the window, and the remaining characters are compared from right to left until a complete match or a mismatch occurs. After each attempt, the skip of the window is gained by the Quick-Search bad-character (qsBc) shift value for the character that is placed next to the window.

The reason for successively comparing the last character first and the first character second and then continuing the comparison of characters in the right to left order of the pattern and window is mainly due to the fact that the dependency of the neighboring characters is strong compared to the other characters. Hence, it is always better to postpone the comparisons on the neighboring characters. This forms the basis for the new algorithm proposed in this paper. The probability of assessing an exact match between the pattern with the window is increased with a minimum number of comparisons by incorporating the concept of initial resemblance. In addition, the maximization of the skip for the window reduces the number of character-character comparisons and hence increases the performance.

3.1. Preprocessing Phase. This is performed using the Quick-search bad-character function (qsBc) for all the characters in the alphabet set. A table is formed with a size σ, storing the character and its corresponding skip value. The value qsBc for a particular alphabet is defined as the position of that character in the pattern from right to left, and if it does not occur in the pattern, then the value is equal to m+1. The skip value for each character is stored in the qsBc table in such a way that it can be used readily in the searching phase to calculate the skip on the window over the text. In the searching phase, after each attempt, the skip of the window is computed by obtaining the shift value of the character immediately after the window. The maximum skip value for the window is realized when the character (character immediately after the window) is not present in the pattern. The probability of a character occurring in the pattern becomes less when the alphabet size is big, and it helps to get a maximum skip of the window. In the proposed algorithm, we consider the Quick-search bad-character (qsBc) over the Boyer-Moore bad-character (bmBc) for the following reasons:

(1) The qsBc value is always defined to be ≥ 1, and hence this could work independently to implement a fast algorithm. On the other hand, bmBc sometimes yields a shift value ≤ 0 and in such cases it could not be used independently. Hence, it has to work along with bmGs (Boyer-Moore good-suffix) to calculate the skip of the window.

(2) qsBc = bmBc +1, except for the last character in the pattern. Hence, qsBc always gets more shift than bmBc in practice.

(3) qsBc does not depend on the order of comparisons between the pattern and the window. This is because qsBc is defined relative to a character that lies outside the current comparison range of the pattern. However, the Boyer-Moore bmBc strongly depends on the right to left pattern scan order.

The above said reasons clearly demonstrate the advantages of using qsBc over bmBc. The preprocessing phase assists the searching phase to improve the overall efficiency of the proposed algorithm. The three stages of the searching phase of the proposed algorithm are outlined in the following sections.

3.2. Search Phase. Stage 1 and stage 2 deal with the order of character-character comparisons between the window and the pattern.

Stage 1. To find out the initial resemblance between the pattern and the window, first, the last character of the pattern and the window are compared, and in case of a match, the first character of the pattern and the corresponding character in the window are compared. If these characters match, the algorithm enters into the next stage; otherwise, it goes to the last stage.

Stage 2. After establishing an initial resemblance on the window and the pattern, the remaining characters are compared in the right to left order until a mismatch occurs or all the m-2 characters match. If all the characters match, the algorithm displays the corresponding position (j) of the window on the text. Then the algorithm enters into the final stage.

Stage 3. In this stage, the calculation of the distance by which the window has to be shifted is computed using qsBc, generated during the preprocessing phase, for the first character placed immediately after the window.

This procedure is repeated until the window is positioned beyond n−m+1.

3.3. C Language Implementation

```
// This function calculates the shift value for all
// the characters in the alphabet
void preQsBc(char *x, int m, int qsBc[]) {
      int i;

      for (i = 0; i < ASIZE; ++i)
         qsBc[i] = m + 1;
      for (i = 0; i < m; ++i)
         qsBc[x[i]] = m - i;
   }

// Global variables used during the pre-processing
// and searching phases
char firstCh, lastCh;
int qsBc[ASIZE];

///////////////////Pre-processing phase///////////////////

// This phase includes the function call where
// the table qsBc is computed and also stores the first and
// the last characters of the pattern
void PREPROCESS(char *x, int m, char *y, int n) {

      preQsBc(x, m, qsBc);

      firstCh = x[0];
      lastCh = x[m - 1];
}
////////////////////Searching phase////////////////////

void SSABS(char *x, int m, char *y, int n) {

      j = 0;

      while (j <= n - m) {

////////////////////////Stage-1////////////////////////

         if (lastCh == y[j + m - 1] && firstCh == y[j]) {

////////////////////////Stage-2////////////////////////

           for(i = m - 2; i > 0 && x[i] == y[j + i]; i--);
             if (i <= 0) {
                OUTPUT(j);
             }
         }


////////////////////////Stage-3////////////////////////

         j += qsBc[y[j + m]];

       }
}
```

■ The value of ASIZE is dependent on the alphabet size. For efficiency considerations, we have chosen a value accordingly to cover all the ASCII values of the characters defined in the present alphabet set.

■ Void OUTPUT (int) is a function used to print the position (j) of the current window on the text.

3.4. Working Example. The human genome consists of 37 490 gene sequences (NCBI site, U.S.A., ftp://ftp.ncbi.nih.gov/genomes/H_sapiens/protein/). To validate the proposed algorithm, a part of the gene sequence (only 47 residues from a gene sequence consisting of 136 amino acid residues) has been used (see below for details).

Full Sequence in FASTA Format. >gi|4504279|ref|NP_002098.1| H3 histone, family 3A [Homo sapiens] MARTKQTARKSTGGKAPRKQLATKAARKSAPSTGGVKKPHRYRPGTVALREIRRYQKSTELLIRKLPFQRLVREIAQDFKTDLRFQSAAIGALQEASEAYLVGLFEDTNLCAIHAKRVTIMPKDIQLARRIRGERA

Part of the Sequence Considered for the Test Run. MARTKQTARKSTGGKAPRKQLATKAARKSAPSTGGVKKPHRYRPGTV

y = MARTKQTARKSTGGKAPRKQLATKAARKSAPSTGGVKKPHRYRPGTV

x = KAPRKQL

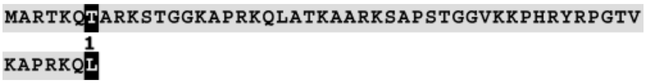
n = 47 m = 7

Preprocessing Phase.

A	A	C	D	E	F	G	H	I	K	L	M	N	P	Q	R	S	T	V	W	Y
qsBc[a]	6	8	8	8	8	8	8	8	3	1	8	8	5	2	4	8	8	8	8	8

A table is formed with a size σ, storing the character and its corresponding skip value.

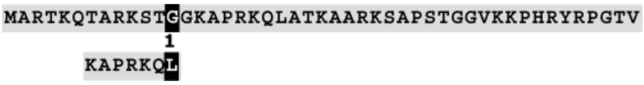
Searching Phase. First attempt:



shift = qsBc[A] = 6

First, the last characters of the pattern and the window are compared. In case of a mismatch, the window moves are based on the Quick-Search bad character shift value of y[j+m].

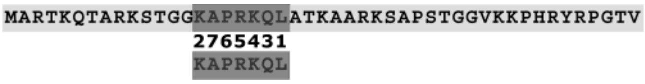
Second attempt:



shift = qsBc[G] = 8

Here again, the comparison of the last characters of the pattern and the text is carried out, and in the event of a mismatch, the window is shifted based on the shift value of y[j+m].

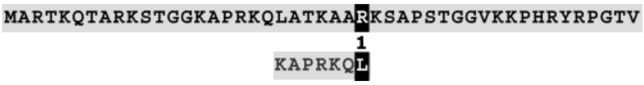
Third attempt:



shift = qsBc[A] = 6

In this case, the given pattern completely matches with the text and the comparison is done as follows: First, the last character 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].

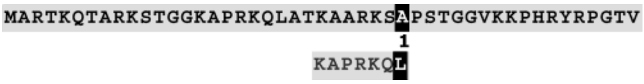
Fourth attempt:



shift = qsBc[K] = 3

In this attempt, the mismatch occurs between the last characters of the pattern and the text. Therefore, the window is shifted based on the shift value.

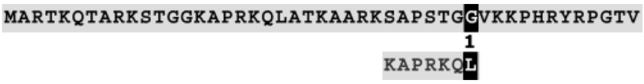
Fifth attempt:



shift = qsBc[P] = 5

In this attempt also, because of the mismatch, the window is moved based on the shift value.

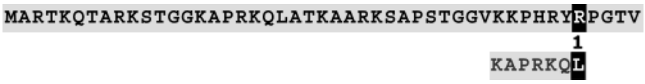
Sixth attempt:



shift = qsBc[V] = 8

Here again, the comparison of the last characters of the pattern and the text fails, so the window is shifted based on the shift value.

Seventh Attempt:



Total number of attempts: 7

Total number of character comparisons: 13

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

4. ANALYSIS OF THE PROPOSED ALGORITHM

The preprocessing phase time complexity of our algorithm is O(m + σ), and the space complexity is O(σ). The subsequent section describes the search phase time complexity.

Lemma 4.1. The time complexity is O ([(n/(m + 1))]) in the best case.

Proof. Every character that does not occur in the pattern has a shift m+1 as defined by the Quick-search bad-character function qsBc, calculated during the preprocessing phase. Considering the best case (all the characters in the pattern is completely different compared to the characters in the text), matching m characters of the pattern in the text (see example 1 below) yields a shift of m+1 at each attempt and hence the time complexity is O ([(n/(m + 1))]).

Example 1:

Text: xxxxxxxxxxxxxxxxxxxxxxxxx

Pattern: yyyyy

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

Proof. By virtue of the fact that every character in the text is matched no more than m times, the total character comparisons for n characters of the text cannot be more than m(n−m+1). In the worst case, the shift is equal to one, and all the characters are matched at each attempt. This can be realized when the characters in the pattern are exactly similar to the characters in the text.

Example 2:

Text: aaaaaaaaaaaaaaaaaaaaaaaaa

Pattern: aaaaa

In the proposed algorithm, the average time complexity cannot be defined strictly because it mainly depends on the alphabet size and the probability of the occurrence of each individual character in the text.

Table 1. Comparison of the Proposed Algorithm with the Well-Known Algorithms Available in the Literature[a]

pattern	algorithm				
length	QS	HORSPOOL	RAITA	BM	SSABS[b]
4	1972 (137)	1259 (76)	1143 (114)	1195 (79)	999 (76)
6	1796 (183)	1140 (100)	1009 (67)	1065 (72)	950 (69)
8	1732 (206)	1109 (109)	980 (79)	1016 (72)	918(57)
10	1774 (255)	1092 (101)	985 (105)	1038 (176)	939 (82)
12	1817 (293)	1125 (135)	991 (96)	1001 (85)	934 (74)
14	1656 (248)	1129 (235)	953 (84)	967 (113)	899 (65)
16	1806 (271)	1126 (122)	984 (88)	1001 (126)	954 (111)
18	1783 (241)	1140 (215)	971 (80)	960 (64)	929 (66)
20	1670 (236)	1098 (103)	967 (83)	930 (53)	900 (68)

[a] The database used is the gene sequences comprised of nucleotides ($\sigma = 4$). The patterns used here are generated randomly. For each pattern length, 50 patterns have been generated and the average time (× 10^{-2}) is calculated. The numbers shown within the parentheses denote the standard deviation. [b] SSABS (Sheik − Sumit − Anindya − Balakrishnan-Sekar) algorithm proposed in the present paper.

5. RESULTS

To assess the performance of our algorithm, we considered all the well-known algorithms for comparison with the proposed algorithm. We have analyzed two types of data, consisting of small ($\sigma = 4$) and big ($\sigma = 20$) alphabet sizes. The first one is the nucleotide sequences available in the genome database and the second is the corresponding amino acid sequences. We have executed and tested all the algorithms under study using a 3.06 GHz processor, 1 GB of RD-RAM with 512 KB of cache memory. The source codes were compiled using the "cc" compiler without any optimization. The reported time (in 10^{-2} seconds) in the tables corresponds to the time taken by the algorithm. The source code for the algorithms used for comparison is taken from the corresponding literature.[5]

6. CASE STUDY WITH THE NUCLEOTIDE SEQUENCES

A total of 837 gene sequences (comprising of nucleotides) (826.31 MB size) have been deployed in the present study. The data set contains four alphabets (nucleotides) in their set viz., A − (Adenine, 239490165), C − (Cytosine, 183940124), G − (Guanine, 183818044), and T − (Thymine, 239419854) and hence, the alphabet size is equal to 4 ($\sigma = 4$). The numbers within the parentheses denote the corresponding occurrences in the entire database. For each pattern length, the calculation has been repeated for 50 different randomly generated patterns to avoid bias in the result. The same procedure is repeated for patterns of different lengths. The average time taken by various algorithms is listed in Table 1. The standard deviation (within the parentheses) is

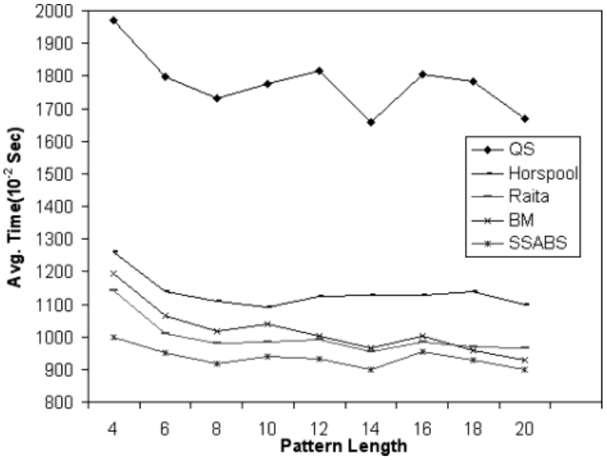


Figure 1. Comparison of the proposed algorithm with the well-known algorithms available in the literature. The database used is the gene sequences comprised of nucleotides ($\sigma = 4$, Table 1). The graph clearly depicts the performance of various algorithms considered in the present study.

also shown. The values in Table 1 are also plotted as a graph (Figure 1), and it is evident that the total time taken by the proposed algorithm is less. It is interesting to note that the standard deviation is low in most of the places, and it shows that the time is uniform irrespective of the pattern and its length.

7. CASE STUDY WITH THE AMINO ACID SEQUENCES

We now consider the situation where the alphabet size is large ($\sigma = 20$) dealing with the amino acid residues. 453 861 gene sequences (191.24 MB size) have been used in the present case study. In this case, the alphabet set is defined as Σ = (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)), and the alphabet size $\sigma = 20$. The numbers within the parentheses denote the corresponding occurrence of a particular amino acid residue in the database. As in the previous case study, the computation has been carried out for 50 randomly selected patterns for each of the pattern lengths considered for study (see Table 2 for details). The average time taken by various algorithms are given in Table 2. The standard deviation is also shown. The values listed in Table 2 clearly show that the proposed algorithm is better compared to other algorithms. The average time and the pattern lengths are also represented graphically (Figure 2). To conclude, the algorithm performs better irrespective of the alphabet size.

8. CONCLUSIONS

In this paper, we propose a new algorithm for exact pattern-matching by defining a new order of character−character comparisons between the pattern and the window at each attempt and by computing the appropriate shift value that maximizes the skip of the window on the text. The new algorithm has been tested, and it is particularly noteworthy that the results obtained are consistently better when compared with the algorithms reported in the open literature. The database used to validate our algorithm is sufficiently large, and at this point, it is enough to conclude that the proposed algorithm is efficient and faster. Hence, it can possibly be implemented in all applications related to exact pattern-matching, in particular, biological sequence database analysis. Our further interests are focused for developing algorithms for multiple pattern matching and patterns that include regular expressions to meet the challenges and demands put forward by the present-day computational genomics research.

Table 2. Comparison of Our Algorithm with the Other Known Algorithms[a]

pattern	algorithm				
length	QS	HORSPOOL	RAITA	BM	SSABS[b]
2	384 (4)	214 (18)	207 (4)	238 (4)	170 (3)
4	293 (4)	175 (10)	174 (3)	189 (3)	158 (2)
6	255 (3)	166 (11)	165 (2)	175 (7)	154 (4)
8	233 (3)	157 (5)	159 (1)	166 (6)	150 (3)
10	220 (4)	154 (3)	156 (2)	163 (8)	148 (2)
12	211 (3)	151 (4)	153 (2)	157 (4)	147 (2)
14	204 (4)	150 (3)	151 (1)	154 (2)	147 (3)
16	199 (4)	149 (2)	151 (1)	153 (2)	146 (2)
18	194 (4)	147 (2)	151 (2)	152 (3)	145 (2)
20	190 (4)	147 (3)	148 (1)	151 (3)	144 (3)

[a] The database used is the gene sequences comprised of amino acid residues ($\sigma = 20$). The patterns used here are generated randomly. For each pattern length, 50 patterns have been generated and the average time ($\times 10^{-2}$) is calculated. The numbers shown within the parentheses denote the standard deviation. [b] SSABS (Sheik − Sumit − Anindya − Balakrishnan − Sekar) algorithm proposed in the present paper.

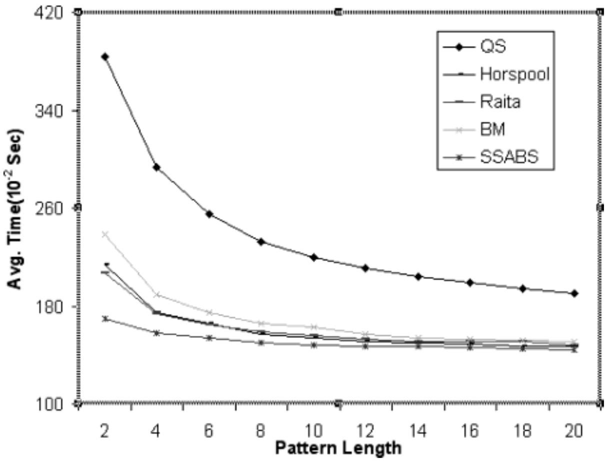


Figure 2. Comparison of the SSABS algorithm with the other known algorithms. The database used is the gene sequences comprised of amino acid residues ($\sigma = 20$, Table 2). The graph clearly shows the smooth behavior of time unlike the prototype observed in Figure 1 and is attributed because of the alphabet size.

ACKNOWLEDGMENT

The authors gratefully thank the Bioinformatics Centre (DIC), the Interactive Graphics Based Molecular Modeling facility (IGBMM) and the Supercomputer Education and Research Centre (SERC) for support. This work is completely supported by the Institute wide Computational Genomics Project supported by the Department of Biotechnology (DBT), Government of India, India. The facilities DIC and IGBMM are supported by DBT. The authors thank the anonymous referees for their useful suggestions. One of the authors (K.S.) thank Ms. B. Sathiyabhama for timely help. Sumit K. Aggarwal is a summer trainee from IIT, Roorkee, India.

REFERENCES AND NOTES

(1) Boyer, R. S.; 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(8)*, 132−142.

(3) Raita, T. Tuning the Boyer-Moore-Horspool string-searching algorithm. *Software − Practice Experience* **1992**, *22(10)*, 879−884.

(4) Horspool, R. N. Practical fast searching in strings. *Software − Practice Experience* **1980**, *10(6)*, 501−506.

(5) Charras, C.; Lecroq, T. Handbook of exact string matching algorithms (Available at the web site − http://www-igm.univ-mlv.fr/∼lecroq/string/).

CI030463Z