A New Substring Searching Algorithm

Xiao ZHAO†a, Sihui LI†, Yun YANG†, Yuyan CHAO††, Nonmembers, and Lifeng HE†††, Member

SUMMARY This paper proposes a new algorithm for substring searching. Our algorithm is an improvement on the famous BM algorithm. When a mismatch happens while searching a substring (pattern), the BM algorithm will use two strategies to calculate shifting distances of the substring respectively and selects the larger one. In comparison, our algorithm uses each of the two strategies for their most suitable cases separately without a selection operation. Experimental results demonstrated that our algorithm is more efficient than the BM algorithm and the Quick Search algorithm, especially for binary strings and DNA strings.

key words: substring searching, pattern matching, binary string, DNA matching

1. Introduction

Searching for a substring, i.e., substring searching, is clearly an essential technique for information retrieval, virus detection, computational biology, etc. in today’s information society [1]. The substring searching problem is to find all the occurrences of a given substring, referred to as pattern $p = p[0] \ldots p[m-1]$, in a larger string, referred to as text $t = t[0] \ldots t[n-1]$. To do that, we should match the substring $p[0] \ldots p[m-1]$ with each substring of text $t[i] \ldots t[i+m-1]$ for all $i$ such that $0 \leq i \leq n - m$. For convenience, a substring of the text being compared with the pattern is called the current substring of the text.

Many algorithms have been proposed for this problem [2]–[6]. Among others, the most famous one is the BM algorithm proposed by Boyer and Moore in 1977 [7].

The BM algorithm matches the pattern and the current substring of the text from right to left. When a mismatch happens, it uses two strategies, one is called bad-character-shift strategy and the other good-suffix strategy, to calculate shifting distances, and then selects the larger one for shifting the pattern to the right. The BM algorithm is quite efficient in practice, and has been improved in two ways: One is by using a different character in the text for bad-character-shift strategy to obtain a larger $skip$ in general, e.g., the Hor-
not occur in the pattern, then, the pattern can be shift over the character, the shift distance is \( j + 1 \) (Fig. 1 (b)); otherwise, let \( k \) be the largest number such that \( t[i + j] = p[k] \), \( 0 \leq k \leq m - 1 \), then, we should shift the pattern to align \( p[k] \) with \( t[i + j] \), the shift distance is \( j - k \) (Fig. 1 (c)).

On the other hand, according to the already-matched suffix, i.e., \( p[j + 1] \ldots p[m - 1] = t[i + j + 1] \ldots t[i + m - 1] \) (Fig. 1 (a)), the good-suffix strategy decides another shift distance, denoted by \( \text{skip}2[j] \), for the pattern as follows: let \( g \) be the largest number such that \( 0 \leq g < j \), \( p[g] \neq p[j] \), and \( p[j + 1] \ldots p[m - 1] = p[g + 1] \ldots p[m - 1 + g - j] \). \( \text{skip}2[j] = j - g \) (Fig. 1 (d)); if there is not such a \( g \), let \( l \) be the smallest number such that \( j < l \leq m - 1 \), \( p[0] \ldots p[m - 1 - l] = p[l] \ldots p[m - 1] \), \( \text{skip}2[j] = l \) (Fig. 1 (e)); if there is also not such an \( l \), \( \text{skip}2[j] = m \) (Fig. 1 (f)).

When a mismatch happens, the BM algorithm will select the maximum one, says \( \text{skip} \), from \( \text{skip}1 \) and \( \text{skip}2 \), i.e., \( \text{skip} = \max(\text{skip}1, \text{skip}2) \), to shift the pattern to the right. Notice that although \( \text{skip}1 \) might be negative, \( \text{skip}2 \) would be always positive, thus, so is \( \text{skip} \). Moreover, both \( \text{skip}1 \) and \( \text{skip}2 \) can be pre-computed by use of the pattern along.

2.2 The QS Algorithm

Because when a mismatch happens, the pattern would be shift to the right at least one character, i.e., the character \( t[i + m] \) in the text will certainly occur in the next current substring of the text, the QS algorithm uses the character \( t[i + m] \) in the text to calculate \( \text{skip}1 \), which is always positive, and in average, larger than that calculated by using \( t[i + j] \). Moreover, observing that most mismatches would happen at the rightmost one or two characters of the pattern (thus, the good-suffix strategy is not very effective), the QS algorithm simplifies the BM algorithm by only using \( \text{skip}1 \) to shift the pattern. A great merit to do this is that the operation for selecting the maximum shift distance, which is made for every mismatch, becomes unnecessary; thus, offsetting the loss in the cases where \( \text{skip}1 < \text{skip} \). Experimental results demonstrated that the QS algorithm is even more efficient than the BM algorithm on English strings in practice [10], [11].

3. Our Proposed Algorithm

For a mismatch between \( p[j] \) and \( t[i + j] \), it is obvious that \( \text{skip}1[t[i + m]] \) is irrelevant to \( j \), but \( \text{skip}2[j] \) is usually inversely proportional to \( j \). Although in the cases where \( j \) is close to \( m - 1 \), \( \text{skip}2[j] \) is usually smaller than \( \text{skip}1[t[i + m]] \), especially for English text, however, in the cases where \( j \) is small, \( \text{skip}2[j] \) might be larger than \( \text{skip}1[t[i + m]] \), especially for binary strings and DNA strings.

Based on the above analysis, in order to archive a shift distance as large as possible in any case, we can decide the shift distance \( \text{skip} \) for shifting the pattern to the right as follows: in the cases where \( j \) is large enough, i.e., larger or equal to \( w \), \( \text{skip} = \text{skip}1[t[i + m]] \); otherwise, \( \text{skip} = \text{skip}2[j] \).

In order to derive a suitable \( w \), we tested the cases for \( w = m - 1, m - 2, m - 3, m - 4 \), we chose different lengths of pattern. The range of lengths is from 2 to 100 in binary strings and DNA strings, and that 2 to 15 in English text. The experimental results showed that different strings had the different result. In English text, the experimental data were similar and not influenced by \( w \). In binary strings and DNA strings, the experimental data obviously were influenced by \( w \) and the suitable \( w \) for them is \( m - 2 \). That is, in any case, if a mismatch happens after less than two characters matched, we should use \( \text{skip}1[t[i + m]] \), otherwise, \( \text{skip}2[j] \), for shifting the pattern. Moreover, notice that our algorithm is only effective for the cases where the length of the pattern is larger than 2. The pseudo codes of our algorithm for testing the match of the pattern \( p[0] \ldots p[m - 1] \) against the current substring of text \( t[i] \ldots t[i + m - 1] \) are shown as follows, where \( m \geq 3 \):

\[
\begin{align*}
\text{if}(p[m - 1] \neq t[i + m - 1] \text{ or } p[m - 2] \neq t[i + m - 2]) \\
\quad l = i + \text{skip}1[t[i + m]]; \\
\text{else} \{ \\
\quad j = m - 3; \\
\quad \text{while}(j \geq 0 \text{ and } p[j] == t[i + j]) j--; \\
\quad \text{if}(j < 0) \\
\quad \quad t[i] \ldots t[i + m - 1] \text{ is an occurrence of the pattern;} \\
\quad \quad i = i + \text{skip}2[j]; \\
\} 
\end{align*}
\]

Obviously, no selection operation is used in our algorithm.

4. Experimental Results

In this section, we compare our algorithm with the BM algorithm and the QS algorithm. All of the three algorithms were implemented in C language on a PC-based workstation (Intel Core i5-3470 CPU@3.20GHz, 4GB Memory, Ubuntu Linux OS), and compiled by the GNU C compiler (version 4.2.3) with the option -O. All experimental results presented in this section were obtained by averaging the execution time for 1000 runs, and executing times are represented in microseconds.

The English text (Bible, 4.0MB), the binary string (5.0MB) and the DNA string (4.5MB) used in our test are downloaded at the site of SMART\(^1\), which is one of basic corpora in the string matching research area.

For comparing efficiencies of the three algorithms on the English text, the lengths of patterns used in test are from 3 to 16. For each length, 100 different patterns randomly selected from the text were tested, and the experimental results are shown in Table 1. On the other hand, for comparing efficiencies of the three algorithms on the binary string, we tested all patterns with lengths from 3 to 7 and 100 different patterns for longer patterns randomly chosen from the binary string, the corresponding experimental results are shown in Fig. 2. Lastly, for comparing efficiencies of the three algorithms on the DNA string, for each length, we

\(^1\)http://www.dmi.unicnt.it/~faro/smart/download.php
Table 1  Experimental results on the English string ($ms$).

| $m$ | 3    | 4    | 5    | 6    | 7    | 8    | 9    |
|-----|------|------|------|------|------|------|------|
| BM  | 16.53| 12.56| 10.25| 8.85 | 7.75 | 6.89 | 6.23 |
| QS  | 9.32 | 7.37 | 6.24 | 5.61 | 4.98 | 4.42 | 4.11 |
| Ours| 9.14 | 7.24 | 6.12 | 5.50 | 4.87 | 4.33 | 4.01 |

Table 1 shows the experimental results on the English string ($ms$).

Fig. 2  Experimental results on the binary string: (a) for lengths of patterns from 3 to 10; (b) for lengths of patterns from 10 to 100.

tested 100 different patterns randomly chosen from the DNA string, the corresponding experimental results are shown in Fig. 3. By the experimental results shown in Table 1, Fig. 2 and Fig. 3, we can find that our algorithm is more efficient than other algorithms, especially on the binary string and the DNA string.

5. Discussion

For binary strings and DNA strings, because the number of alphabets in strings is small, a mismatch often happens at $p[j]$ such that some characters have matched, i.e., where $j$ is relatively small. As indicated above, $skip2[j]$ will become larger and larger as $j$ goes smaller. Therefore, our strategy is quite effective. For example, for the pattern string $\{10011\}$, where $skip1[c]$ for $c = 0, 1$, and $skip2[j]$ for $j = 5, 4, 3, 2, 1, 0$ are shown in Fig. 4, we can find that when $j$ is smaller than 4, $skip2[j]$ is much larger than any of $skip1[c]$. The main advantage of our algorithm is that we can use each of the two strategies in its most suitable cases without using a selection operation.

On the other hand, for English strings, the number of alphabets in strings is large, a mismatch rarely happens at $p[j]$ such that $j$ is smaller than $m - 2$. In other words, most of the time, our algorithm works the same as does the QS algorithm. Therefore, our algorithm is only little effective on English strings.

It is worth to mention that, although we have not discussed, the strategies for reducing the number of matching times for finding a mismatch can be also easily applied in our algorithm. Moreover, for long patterns in binary strings and DNA strings, we can apply the KMP algorithm’s strategy to avoid repeated checking on already-known matched substring, which can guarantee our algorithm to be in a worst-case $O(n)$ search time.

6. Conclusion

In this paper, we presented a new substring searching algorithm. By using different strategies for deciding the shifting distance automatically when a mismatch happens, our algorithm can achieve a larger shifting distance in general,
thus leading to a more efficient search. For future works, we will apply the KMP algorithm’s strategy to avoid repeated checking on already-known matched substring, and apply our algorithm in practice, such as virus detection and DNA detection.

Acknowledgements

We thank the anonymous referee for his/her valuable comments that improved this paper greatly. We are grateful to the associate editor, Prof. Araki, for his kind cooperation.

References

[1] H. Thomas, E. Charles, L. Ronald, and S. Clifford, Introduction to Algorithms, Second ed., MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Chapter 32: String Matching, pp.906–932.

[2] D. Knuth, J. Morris, and V. Pratt, “Fast pattern matching in strings,” SIAM J. Comput., vol.6, no.1, pp.323–350, 1977.

[3] R. Karp and M. Rabin, “Efficient randomized pattern-matching algorithms,” IBM Journal of Research and Development, vol.31, no.2, pp.249–260, 1987.

[4] R. Baeza-Yates and G. Gonnet, “A new approach to text searching,” Commun. ACM, vol.35, no.10, pp.74–82, 1992.

[5] G. Navarro and M. Raffinot, “A bit-parallel approach to suffix automata: Fast extended string matching,” Lect. Notes Comput. Sci., vol.1448, pp.14–31, Springer-Verlag, Berlin, 1998.

[6] C. Allauzen, M. Crochemore, and M. Raffinot, “Factor oracle: A new structure for pattern matching,” Lect. Notes Comput. Sci., vol.1725, pp.291–306, Springer-Verlag, Berlin, 1999.

[7] R. Boyer and J. Moore, “A fast string searching algorithm,” Commun. ACM, vol.20, no.10, pp.762–772, 1977.

[8] R.N. Horspool, “Practical fast searching in strings,” Software - Practice & Experience, vol.10, no.6, pp.501–506, 1980.

[9] M. Sunday, “A very fast substring search algorithm,” Commun. ACM, vol.33, no.8, pp.132–142, 1990.

[10] G. Hen, String Searching Algorithms, World Scientific, 1994.

[11] G. Navarro and M. Raffinot, Flexible Pattern Matching in Strings, Cambridge University Press, 2002.