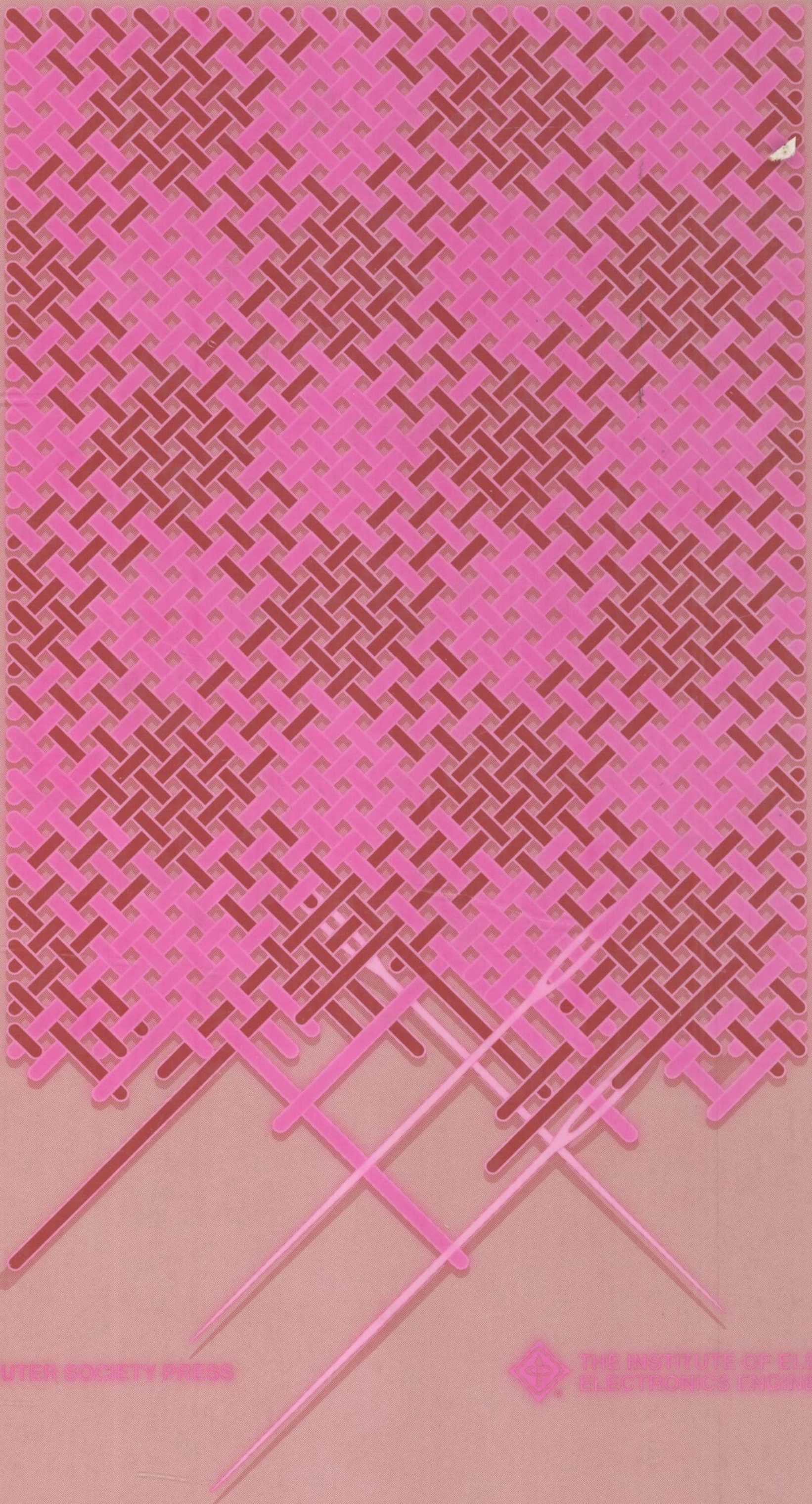
COMPUTER ALGORITHMS

String Pattern Matching Strategies

JUN-ICHI AOE



IEEE COMPUTER SOCIETY PRESS

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

Computer Algorithms

String Pattern Matching Strategies

Jun-ichi Aoe

University of Tokushima, Japan

IEEE Computer Society Press
Los Alamitos, California

Washington • Brussels • Tokyo

Computer Algorithms: string pattern matching strategies / [edited by]
Jun-ichi Aoe.
p. cm.
Includes bibliographical references and index.
ISBN 0-8186-5461-9 (m/f). -- ISBN 0-8186-5462-7 (case). -- ISBN
0-8186-5460-0 (paper)
1. Computer algorithms. I. Aoe, Jun-ichi, 1951-
QA76.9.A43C67 1994
005.1--dc20 93-40355
CIP

IEEE Computer Society Press
10662 Los Vaqueros Circle
P.O. Box 3014
Los Alamitos, CA 90720-1264

IEEE Computer Society Press Order Number 5462-05
IEEE Catalog Number 94EH0389-7
Library of Congress Number 93-40355
ISBN 0-8186-5461-9 (microfiche)
ISBN 0-8186-5462-7 (case)

Additional copies may be ordered from:

IEEE Computer Society Press
Customer Service Center
10662 Los Vaqueros Circle
P.O. Box 3014
Los Alamitos, CA 90720-1264
Tel: +1-714-821-8380
Fax: +1-714-821-4641
Email: cs.books@computer.org

IEEE Service Center
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331
Tel: +1-908-981-1393
Fax: +1-908-981-9667

IEEE Computer Society
13, Avenue de l'Aquilon
B-1200 Brussels
BELGIUM
Tel: +32-2-770-2198
Fax: +32-2-770-8505

IEEE Computer Society
Ooshima Building
2-19-1 Minami-Aoyama
Minato-ku, Tokyo 107
JAPAN
Tel: +81-3-3408-3118
Fax: +81-3-3408-3553

Technical Editor: Ajit Singh
Editorial production by Bob Werner
Cover art design and production by Alexander Torres
Printed in the United States of America by Braun-Brumfield, Inc.

The Institute of Electrical and Electronics Engineers, Inc.

Computer Algorithms

String Pattern Matching Strategies

Preface

String pattern matching is an important component of many areas of science and information processing. It occurs naturally as part of data processing, text editing, symbol manipulation, term-rewriting, lexical analysis, code generation, spelling correction, bibliographic search, text retrieval, and natural language processing. String pattern matching techniques can be also applied to the recognition of patterns such as shapes, pictures, scenes, and so on. In biology, string pattern matching problems arise in the analysis of protein sequences and nucleic acids and in the investigation of molecular phylogeny. String pattern matching is the most time-consuming part of many programs, and the substitution of a poor matching method by a good one often leads to a substantial increase in speed. Therefore, a fast methodology should be selected. The aim of this volume is to introduce the basic concepts and characteristics of string pattern matching strategies and to provide numerous references for further reading.

The pattern matcher is a program that takes as input the text string x and produces as output the locations in x at which patterns, or keywords, appear as substrings. The simplest patterns are single keywords that match themselves. A somewhat broader class of patterns would be sets of keywords. There are two important variants of pattern-matching problems. One is approximate string matching problems, in which one must find all substrings in a text that are close to a pattern according to some measure of closeness. Another is multidimensional matching problems for finding patterns in higher-dimensional structures such as trees and graphs. In recent years some types of pattern matching algorithms have been implemented on hardware based on the finite state automata and signature files in order to improve processing efficiency. In this book, string pattern matching strategies are classified into the following five chapters.

- Single keyword matching
- Matching sets of keywords
- Approximate string matching
- Multidimensional matching
- Hardware matching

As an introduction to each chapter, my survey article describes the basic concepts of classification mentioned above. Fifteen papers have been selected to further illustrate these concepts. Also, I have made considerable efforts to find a large number of corresponding references and to organize them.

Most of the string matching techniques are treated in detail, with mathematical analyses and suggestions for practical applications, in the books and articles cited throughout the book. The references [Aho, 80], [Aho, 90], [Apostolico et al., 85], [Gonnet et al., 85], and [Sankoff et al., 83] are good surveys for general string pattern matching techniques. The six books [Aho et al., 74], [Frakes et al., 92], [Knuth, 73], [Mehlhorn, 84], [Sedgewick, 86], and [Standish, 80] are useful for the corresponding basic data structures and algorithms.

Contents

Chapter 1: Single Keyword Matching

Introduction

Single keyword matching means locating all occurrences of a given pattern in the input text string. It occurs naturally as part of data processing, text editing, text retrieval, and so on. Many text editors and programming languages have facilities for matching strings. The simplest technique is called the brute-force (BF), or naive, algorithm. This approach scans the text from left to right and checks the characters of the pattern character by character against the substring of the text string beneath it. Let m and n be the lengths of the pattern and the text, respectively. In the BF approach, the longest (worst-case) time required for determining that the pattern does not occur in the text is $O(mn)$.

Three major pattern matching algorithms for the improvement of efficiency over the BF technique exist. One of them is the KMP algorithm, developed by Knuth, Morris, and Pratt. The KMP algorithm scans the text from left to right, using knowledge of the previous characters compared to determine the next position of the pattern to use. The algorithm first reads the pattern and in $O(m)$ time constructs a table, called the *next function*, that determines the number of characters to slide the pattern to the right in case of a mismatch during the pattern matching process. The expected theoretical behavior of the KMP algorithm is $O(n+m)$, and the next function takes $O(m)$ space.

The next algorithm, the BM algorithm, was proposed by Boyer and Moore. The BM approach is the fastest pattern matching algorithm for a single keyword in both theory and practice. The BM algorithm compares characters in the pattern from right to left. If a mismatch occurs, the algorithm computes a shift, that is, the amount by which the pattern is moved to the right before a new matching is attempted. It also preprocesses the pattern in order to produce the shift tables. The expected theoretical behavior of the BM algorithm is equal to that of the KMP algorithm, but many experimental results show that the BM algorithm is faster than the KMP algorithm.

The last approach is the KR algorithm, presented by Karp and Rabin. The KR algorithm uses extra memory to advantage by treating each possible m-character section (where m is the pattern length) of the text string as a keyword in a standard hash table, computing the hash function of it, and checking whether it equals the hash function of the pattern. Although the KR algorithm is linear in the number of references to the text string per characters passed, the substantially higher running time of this algorithm makes it unfeasible for pattern matching in strings.

In the rest of the chapter, many improvements, including parallel approaches, and variants of the basic single keyword matching algorithms introduced above are discussed along with the corresponding references.

In order to introduce these typical single keyword matching techniques, I have selected the three papers Knuth, Morris, and Pratt (1977), Boyer and Moore (1977), and Davies and Bowsher (1986). The first two papers are the original papers of the KMP and BM algorithms, respectively. The third paper includes comprehensive descriptions and useful empirical evaluation of the BF, KMP, BM, and KR algorithms. Good surveys of single keyword matching are in [Baeza-Yates, 89a], [Baeza-Yates, 92], and [Pirkldauer, 92].

Brute-force (BF) algorithm

This approach scans the text from left to right and checks the characters of the pattern character by character against the substring of the text string beneath it. When a mismatch occurs, the pattern is shifted to the right one character. Consider the following example.

Pattern:	text
Text:	In $\underline{t}$his example $\underline{t}$he algori$\underline{t}$hm searches in $\underline{t}$he $\underline{t}$ext ...

In this example the algorithm searches in the text for the first character of the pattern (indicated by underline). It continues for every character of the pattern, abandoning the search as soon as a mismatch occurs; this happens if an initial substring of the pattern occurs in the text and is known as a *false start*. It is not difficult to see that the worst-case execution time occurs if, for every possible starting position of the pattern in the text, all but the last character of the pattern matches the corresponding character in the text. For pattern $a^{m-1}b$ and for text a^n with $n>>m$, $O(mn)$ comparisons are needed to determine that the pattern does not occur in the text.

Knuth-Morris-Pratt (KMP) algorithm

The KMP algorithm scans the text from left to right, using knowledge of the previous characters compared, to determine the next position of the pattern to use. The algorithm first reads the pattern and in $O(m)$ time constructs a table, called the next function, that determines how many characters to slide the pattern to the right in case of a mismatch during the pattern matching process. Consider the following example.

Position:	1	2	3	4	5
Pattern:	a	b	a	a	a
next:	0	1	0	2	2

By using this next function, the text scanning is as follows:

	i	1	2	3	4	5					
Pattern:		a	b	a	$\underline{a}$	a					
Text:		a	b	a	$\underline{b}$	a	a	b	a	a	a
	j	1	2	3	4	5	7	8	9	10	11

Let i and j be the current positions for the pattern and the text, respectively. In the position j=4, which is a b in the text, matching becomes unsuccessful in the same position, i=4, which is an a in the pattern. By adjusting i=next[4]=2 to j=4, the pattern is shifted 2 characters to the right as follows:

			i	1	2	3	4	5			
Pattern:				a	b	a	a	$\underline{a}$			
Text:		a	b	a	b	a	a	$\underline{b}$	a	a	a
	j	1	2	3	4	5	7	8	9	10	11

After aa is matched, a mismatch is detected in the comparison of a in the pattern with b in the text. Then, the pattern is shifted 3 characters to the right by adjusting i=next[5]=2 to j=8, and then the algorithm finds a successful match as follows:

```
                              i   1   2   3   4   5
Pattern:                          a   b   a   a   a
  Text:      a   b   a   b   a    a   b   a   a   a
         j   1   2   3   4   5    7   8   9  10  11
```

Summarizing, the expected theoretical behavior of the KMP algorithm is O(n+m), and takes O(m) space for the next function. Note that the running time of the KMP algorithm is independent of the size of the alphabet.

Variants that compute the next function are presented by [Bailey et al., 80], [Barth, 81], and [Takaoka, 86]. Barth ([Barth, 84] and [Barth, 85]) has used Markov-chain theory to derive analytical results on the expected number of character comparisons made by the BF and KMP algorithms on random strings.

Boyer and Moore (BM) algorithm

The BM approach is the fastest pattern matching algorithm for a single keyword in both theory and practice. In the KMP algorithm the pattern is scanned from left to right, but the BM algorithm compares characters in the pattern from right to left. If mismatch occurs, then the algorithm computes a shift; that is, it computes the amount by which the pattern is moved to the right before a new matching attempt is undertaken. The shift can be computed using two heuristics. The *match* heuristic is based on the idea that when the pattern is moved to the right, it has to match over all the characters previously matched and bring a *different* character over the character of the text that caused the mismatch. The *occurrence* heuristic uses the fact that we must bring over the character of the text that caused the mismatch the first character of the pattern that matches it. Consider the following example of [Boyer et al., 77].

```
Pattern:  A T - T H A T
  Text:   W H I C H - F I N A L L Y - H A L T S . - - - A T - T H A T - P O I N T
```

At the start, comparing the seventh character, *F*, of the text with the last character, *T*, fails. Since *F* is known not to appear anywhere in the pattern, the text pointer can be automatically incremented by 7.

```
Pattern:                A T - T H A T
  Text:   W H I C H - F I N A L L Y - H A L T S . - - - A T - T H A T - P O I N T
```

The next comparison is of the hyphen in the text with the rightmost character in the pattern, *T*. They mismatch, and the pattern includes a hyphen, so the pattern can be moved down 4 positions to align the two hyphens.

```
Pattern:                        A T - T H A T
  Text:   W H I C H - F I N A L L Y - H A L T S . - - - A T - T H A T - P O I N T
```

After *T* is matched, comparing *A* in the pattern with *L* in the text fails. The text pointer can be moved to the right by 7 positions, since the character causing the mismatch, *L*, does not occur anywhere in the pattern.

```
Pattern:                                   A T - T H A T
   Text:   WHI CH- F I NAL L Y - HALT S . - - - A T - T HA T - P OI NT
```

After *AT* is matched, a mismatch is detected in the comparison of *H* in the pattern with the hyphen in the text. The text pointer can be moved to the right by 7 places, so as to align the discovered substring *AT* with the beginning of the pattern.

```
Pattern:                                              A T - THAT
   Text:   WHI CH- F I NALL Y - HAL T S . - - - A T - THAT - P OI N T
```

Karp and Rabin (KR) algorithm

An algorithm developed in [Karp et al., 87] is an improvement of the brute-force approach to pattern matching. This algorithm is a probabilistic algorithm that adapts hashing techniques to string searching. It uses extra memory to advantage by treating each possible *m*-character section of the text string (where *m* is the pattern length) as a key in a standard hash table, computing the hash function of it, and checking whether it equals the hash function of the pattern. Similar approaches using signature files will be discussed in chapter 5.

Here the hash function is defined as follows:

$$h(k) = k \textbf{ mod } q, \text{ where } q \text{ is a large prime number.}$$

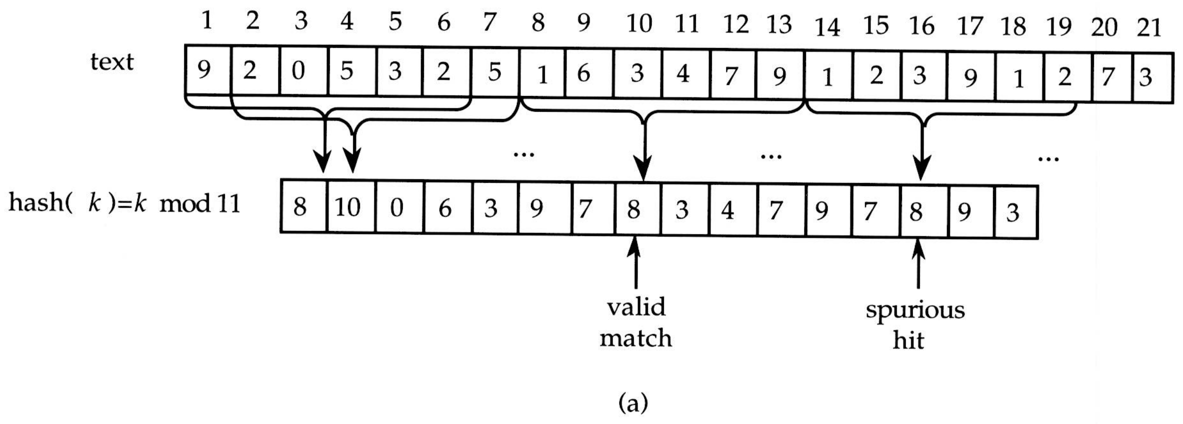
A large value of *q* makes it unlikely that a collision will occur. We translate the *m*-character into numbers by packing them together in a computer word, which we then treat as the integer *k* in the function above. This corresponds to writing the characters as numbers in a radix *d* number system, where *d* is the number of possible characters. The number *k* corresponding to the *m*-character section text[*i*]...text[*i*+*m*-1] is

$$k = \text{text}[i] \times d^{m-1} + \text{text}[i+1] \times d^{m-2} + \cdots + \text{text}[i+m-1]$$

Shifting one position to the right in the text string simply corresponds to replacing *k* by

$$(k - \text{text}[i] \times d^{m-1}) \times d + \text{text}[i+m]$$

Consider the example shown in Figure 1 of the KR algorithm based on [Cormen et al., 90]. Each character is a decimal digit, and the hashed value in computed by modulo 11. In Figure 1a the same text string with values computed modulo 11 for each possible position of a section of length 6. Assuming the pattern k=163479, we look for sections whose value modulo 11 is 8, since h(k)=163479 mod 11=8. Two such sections for 163479 and 123912 are found. The first, beginning at text position 8, is indeed an occurrence of the pattern, and the second, beginning at text position 14, is spurious. In Figure 1b we are computing the value for a section in constant time, given the value for the previous section. The first section has value 163479. Dropping the high-order digit 1 gives us the new value 634791. All computations are performed modulo 11, so the value of the first section is 8 and the value computed of the new section is 3.



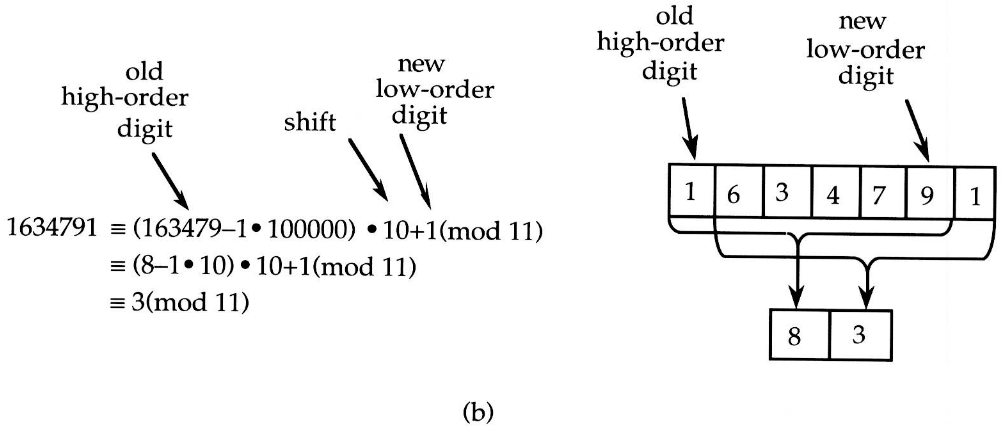


Figure 1. Illustrations of the KR algorithm

Evaluations of single keyword matching algorithms

Rivest ([Rivest, 77]) has shown that any algorithm for finding a keyword in a string must examine at least $n-m+1$ of the characters in the string in the worst case, and Yao ([Yao, 79]) has shown that the minimum average number of characters needed to be examined in looking for a pattern in a random text string is $O(n\lceil \log_A m \rceil / m)$ for $n>2m$, where A is the alphabet size. The upper bound and lower bound time complexities of single pattern matching are discussed in [Galil et al., 91] and [Galil et al., 92].

The minimum number of character comparisons needed to determine all occurrences of a keyword is an interesting theoretical question. It has been considered by [Galil, 79] and [Guibas et al., 80], and they have discussed some improvements to the BM algorithm for its worst-case behavior. Apostolico et al. ([Apostolico et al., 84] and [Apostolico et al., 86]) have presented a variant of the BM algorithm in which the number of character comparisons is at most $2n$, regardless of the number of occurrences of the pattern in the string. Sunday ([Sunday, 90]) has devised string matching methods that are generally faster than the BM algorithm. His faster method uses statistics of the language being scanned to determine the order in which character pairs are to be compared. In the paper [Smith, 91] the peformances of similar, language-independent algorithms are examined. Results comparable with those of language-based algorithms can be achieved with an adaptive technique. In terms of character comparisons, a faster algorithm than Sunday's is constructed by using the larger of two pattern shifts. Evaluating the theoretical time complexity of the BF, KMP, and BM algorithms, based on empirical data presented, Smit ([Smit, 82]) has shown that, in a general text editor operating on lines of text, the best solution is to use the BM algorithm for patterns longer than three characters and the BF algorithm in the other cases. The KMP algorithm may perform significantly better than the BF algorithm when comparing strings from a small alphabet, for example, binary strings. Some experiments in a distributed environment are presented in [Moller et al., 84]. Considering the length of patterns, the number of alphabets, and the uniformity of texts, there is a trade-off between time (on the average) and space in the original BM algorithm. Thus, the original algorithm has been analyzed extensively, and several variants of it have been introduced. An average-case analysis of the KR method is discussed in [Gonnet et al., 90]. Other theoretical and experimental considerations for single keyword matching are in [Arikawa, 81], [Collussi et al., 90], [Li, 84], [Liu, 81], [Miller et al., 88], [Slisenko, 80], [Waterman, 84], and [Zhu et al., 87]. Preprocessing and string matching techniques for a given text and pattern are discussed in [Naor, 91].

For incorrect preprocessing of the pattern based on the KMP algorithm, a corrected version can be found in [Rytter, 80]. For m being similar to n, Iyenger et al. ([Iyenger et al., 80]) have given a variant of the BM algorithm. A combination of the KMP and BM algorithms is presented in [Semba, 85], and the worst cost is proportional to $2n$.

Horspool ([Horspool, 80]) has presented a simplification of the BM algorithm, and based on empirical results has shown that this simpler version is as good as the original one. The simplified version is obtained by using only the match heuristic. The main reason behind this simplification is that, in practice, the occurrence table does not make much contribution to the overall speed. The only purpose of this table is to optimize the handling of repetitive patterns (such as *xabcyyabc*) and so to avoid the worst-case running time, $O(mn)$. Since repetitive patterns are not common, it is not worthwhile to expend the considerable effort needed to set up the table. With this, the space depends only on the size of the alphabet (almost always fixed) and not on the length of the pattern (variable).

Baeza-Yates ([Baeza-Yates, 89b]) has improved the average time of the BM algorithm using extra space. This improvement is accomplished by applying a transformation that practically increases the size of the alphabet in use. The improvement is such that for long patterns an algorithm more than 50 percent faster than the original can be obtained. In this paper different heuristics are discussed that improve the search time based on the probability distribution of the symbols in the alphabet used. Schaback ([Schaback, 88]) has also analyzed the expected performance of some variants of the BM algorithm.

Horspool's implementation performs extremely well when we search for a random pattern in a random text. In practice, however, neither the pattern nor the text is random; there exist strong dependencies between successive symbols. Raita ([Raita, 92]) has suggested that it is not profitable to compare the pattern symbols strictly from right to left; if the last symbol of the pattern matches the corresponding text symbol, we should next try to match the first pattern symbol, because the dependencies are weakest between these two. His resulting code runs 25 percent faster than the best currently known routine.

Davies et al. ([Davies et al., 86]) have described four algorithms (BF, KMP, BM, and KR) of varying complexity used for pattern matching; and have investigated their behavior. Concluding from the empirical evidence, the KMP algorithm should be used with a binary alphabet or with small patterns drawn from any other alphabet. The BM algorithm should be used in all other cases. Use of the BM algorithm may not be advisable, however, if the frequency at which the pattern is expected to be found is small, since the preprocessing time is in that case significant; similarly with the KMP algorithm, so the BF algorithm is better in that situation. Although the KR algorithm is linear in the number of references to the text string per characters passed, its substantially higher running time makes it unfeasible for pattern matching in strings. The advantage of this algorithm over the other three lies in its extension to two-dimensional pattern matching. It can be used for pattern recognition and image processing and thus in the expanding field of computer graphics. The extension will be discussed in Section 4.

Related problems

Cook ([Cook, 71]) has shown that a linear-time pattern matching algorithm exists for any set of strings that can be recognized by a two-way deterministic push-down automaton (2DPDA), even though the 2DPDA may spend more than linear time recognizing the set of strings. The string matching capabilities of other classes of automata, especially k-head finite automata, have been of theoretical interest to [Apostolico et al., 85], [Chrobak et al., 87], [Galil et al., 83], and [Li et al., 86].

Schemes in [Bean et al., 85], [Crochemore et al., 91], [Duval, 83], [Guibas et al., 81a], [Guibas et al., 81b], and [Lyndon et al., 62] have added new vigor to the study of periods and overlaps in strings and to the study of the combinatorics of patterns in strings. [Crochemore et al., 91] presents a new algorithm that can be viewed as an intermediate between the standard algorithms of the KMP and the BM. The algorithm is linear in time and uses constant space like the algorithm of [Galil et al., 83]. The algorithm relies on a previously known result in combinatorics on words, the critical factorization theorem, which relates the global period of a word to its local repetitions of blocks. The following results are presented in [Crochemore et al., 91].

1. It is linear in time O(n+m), as KMP and BM, with a maximum number of letter comparisons bounded by $2n$+$5m$ compared to $2n$+$2m$ for KMP and $2n$+$f(m)$ for BM, where f depends on the version of their algorithm.
2. The minimum number of letter comparisons used during the search phase (executing the preprocessing of the pattern) is $2n/m$ compared to n for KMP and n/m for BM.
3. The memory space used, additional to the locations of the text and the pattern, is constant instead of O(m) for both KMP and BM.

Parallel approaches of string matching are discussed in [Galil, 85], and an O(log log n) time parallel algorithm improving Galil's method is presented in [Breslauer et al., 90]. The paper [Breslauer et al., 92] describes the parallel complexity of the string matching problem using p processors for general alphabets. The other parallel matching algorithms are discussed in [Barkman et al., 89], [Kedam et al., 89], and [Viskin, 85].