Fast String Matching using an *n*-gram Algorithm

jong yong kim and john shawe-taylor

Department of Computer Science, Royal Holloway and Bedford New College, University of London, Egham, Surrey TW20 0EX, U.K. (email: john@dcs.rhbnc.ac.uk)

SUMMARY

Experimental results are given for the application of a new *n*-gram algorithm to substring searching in DNA strings. The results confirm theoretical predictions of expected running times based on the assumption that the data are drawn from a stationary ergodic source. They also confirm that the algorithms tested are the most efficient known for searches involving larger patterns.

key words: String searching Pattern matching Boyer-Moore algorithm

INTRODUCTION

Recently an algorithm was developed for the substring search problem with expected running time provably fast when the text string was drawn from a stationary ergodic source.¹ The theory of ergodic sources was developed by Shannon and others as a theoretical model of natural language. The model is based on the assumption that the probabilities of substrings are invariant to shifts of position together with a regularizing assumption termed metric transitivity which excludes pathological examples. For more information we would refer the reader to Welsh, who gives an excellent introduction to the theory.² In view of this theory it is to be expected that the complexity results of the algorithm should hold when it is applied to natural language texts. Kim and Shawe-Taylor³ give experimental results which confirm this expectation and demonstrate the power of the algorithm in practice. They also compare their results with the modified Baeza-Yates algorithm,⁴ and those presented in a recent paper⁵ describing improvements of the BM⁶ and Sunday algorithm.⁷

An application area of great current interest for string-matching algorithms is that of DNA sequences. It is certainly not immediately clear whether they can reasonably be regarded as ergodic sequences. This paper investigates the performance of the new algorithm on such data and compares it with other known algorithms. The results are very encouraging, suggesting that the expected running time theoretical results for ergodic sequences provide a very good estimate of the algorithm's performance on DNA sequences. In fact the algorithm provides its most striking performance on longer pattern strings and small size of alphabet.

0038–0644/94/010079–10\$10.00 *Rece* © 1994 by John Wiley & Sons, Ltd.

Received 4 December 1991 Revised 25 June 1993

j. y. kim and j. shawe-taylor

ALGORITHMS

Existing algorithms

Let p_i be the *i*th character in the pattern string $P = p_0, \ldots, p_{m-1}$ of length *m* and let t_i be the *j*th character in the text string $T = t_0, \ldots, t_{N-1}$ of length N > m.

The straightforward method (SF) is the simplest algorithm. The pattern is placed over the text at its extreme left and scanned to the right for a mismatch. If a mismatch occurs, the pattern is shifted one position to the right and the scan is restarted at the leftmost position of the pattern. As the N - M + 1 positions in the text are searched, it has O(mN) worst case time complexity.

The main drawback of the SF algorithm is the backtracking which causes the quadratic time complexity. The first algorithm to improve on this bound was the Knuth–Morris–Pratt (KMP) algorithm.⁸ The matching sequence is the same as the SF algorithm but it removes the backtracking by using a precomputed table Δ_2 which indicates how far the pattern should be moved to the right when a mismatch occurs at a particular position. Hence it reduces the worst-case time complexity to O(N + m) where O(m) is the time to build the table Δ_2 .

The KMP algorithm was improved by Boyer and Moore⁶ by changing the scanning direction and adding another shift table Δ_1 . It matches the last character of the pattern first and then proceeds from right to left through the pattern. The additional shift table Δ_1 is used to find the index of the first occurrence from the right end of the pattern for each character in the alphabet. Actual jumps to the right using the Δ_1 table are usually larger than when testing is started at the left. This makes the running time faster than the KMP algorithm in practice, although the worst-case time complexity is O(mN).

The new version of the KMP and BM algorithm was obtained by searching for the last block of *n* characters in the pattern. It was analysed theoretically by KMP⁸ and Schaback,⁹ and implemented practically by Horspool¹⁰ where *n* is 1 and by Baeza-Yates⁴ where n > 1. This idea uses an α^n size of table to store the jump value, where α is the size of alphabet. The entry in the table is calculated using shift instructions. The theoretical average running time is $O(N\log m/m)$ where the size *n* is $O(\log m)$. The size of the table for the Baeza-Yates algorithm is in this case $O(\alpha^{\log m})$. If we require *t* bits to represent a character in the alphabet, the size of the table is $\Omega(m^t)$. Hence the set-up time for the algorithm is $\Omega(m^t + m\log m)$. This gives an overall complexity of $\Omega(m^t + m\log m + N\log m/m)$. For t > 1 this is larger than that reported for the *n*-gram algorithm¹ which has overall complexity of $O((N/m + m)\log m)$. The difference in set-up times becomes significant with larger pattern sizes, as our results will show.

The last previously-published algorithm we consider was described by Sunday.⁷ In this algorithm, any scanning ordering can be considered and the table Δ_2 is adjusted according to the new ordering. The other table Δ_1 is calculated based on the text character *c* beyond the end of the pattern string, and thus each shift value of the Δ_1 table is actually one greater than the BM Δ_1 table. For a mismatch, the current mismatched position and the character *c* determine the entries in the Δ_1 and Δ_2 tables rather than the character on the current position as in the BM algorithm. In practical experiments Sunday suggested two different ways of ordering: one in terms of alphabet frequency and the other by the distance between repeated characters

fast string matching

in the pattern. They are called the optimal shift algorithm (OM) and maximal shift algorithm (MS), respectively. Sunday conjectured the worst-case time complexity to be O(m + N) regardless of the ordering chosen. Later Hume and Sunday⁵ classified a large range of substring-matching algorithms using three components, *skip loop*, *match* and *shift*, and gave performance comparisons for these combinations.

Improved algorithms

Our algorithm using *n*-grams is a natural extension of the BM algorithm. It is also very similar in approach to the Baeza-Yates algorithm.⁴ An *n*-gram is a substring of length *n*. In an ergodic source for most *n*-grams, the probability that they can be found in a sequence falls off exponentially with *n*. By searching for *n*-grams rather than single characters as in the BM algorithm we can significantly reduce the probability of finding a false match in the pattern. The size of *n* is chosen depending on the size of the pattern and the entropy of the background language. The *n*-gram statistics of an ergodic source are treated by Welsh² and experimental results for natural languages are given by Ching.¹¹

From the *n*-gram viewpoint, the Δ_1 table is just a *uni*-gram table. If we increase the size of *n*, we will require a table of size α^n . Clearly for large size of alphabet and *n*, this will be unacceptably large, giving a significant initialization time and a space problem. We overcome this limitation using a new data structure called a *reverse n-gram tree* which is a trie structure. It is well known¹² that the number of nodes in a trie is of the order of the number of entries stored. In our case this will be O(m). Each node has size $O(\alpha) = O(2^t)$. Hence the size of the data structure is $O(m2^t)$, which is asymptotically significantly smaller than the $\Omega(m^t)$ space requirement of the Baeza-Yates algorithm.

Data structure

Let $\mathbf{s} = s_1 s_2 \dots s_m$ be a one-dimensional string over some alphabet A. The reverse of the string \mathbf{s} is the string $\mathbf{s}^{\mathsf{T}} = s_m s_{m-1} \dots s_1$. Let $I = (i_1, i_2, \dots, i_{n-1})$ be a sequence of (n-1) integers where $0 < i_{j+1} < i_j < m$. We define the *n*-gram set $\mathbf{g}(\mathbf{s}, I)$ for the string \mathbf{s} as

$$\{\mathbf{g}_{k} | \mathbf{g}_{k} = s_{k-i_{1}} s_{k-i_{2}} \dots s_{k-i_{n-1}} s_{k}, 1 \leq k \leq m\}$$

Whenever i < 1, s_i is set to the special character $\$ \notin A$.

Let $P = \{(\mathbf{g}_1, k_1), (\mathbf{g}_2, k_2), \dots, (\mathbf{g}_m, k_m)\}$ be a set of *m* pairs of an *n*-gram and its position in the string **s**. The tree T(P) is a trie (digital search tree¹³) generated from the set *P*, as follows:

- 1. Each edge is labelled with a character $x \in A \cup \{\$\}$, and the edges leaving the same node have distinct characters.
- 2. Each internal node including the root is labelled with Δ .
- 3. Each complete path from the root to a leaf corresponds to \mathbf{g}^{r} for some *n*-gram \mathbf{g} of *P*. The leaf corresponding to \mathbf{g}^{r} is labelled with max $\{k_{i}|\mathbf{g}_{i}=\mathbf{g}\}$.

4. For each *n*-gram **g** of *P*, \mathbf{g}^{r} determines a path from the root to a leaf node.

For example, for the string s = cababcbab, n = 3 and I = (2,1), the set P(s,I) is

{(\$c,1), (\$ca,2), ..., (*cba*,8), (*bab*,9)}. Figure 1 shows the reverse *tri*-gram tree $T(P(\mathbf{s},I))$ for the string **s**. Given the tree T(P) for a sequence

$$P = \{ (\mathbf{g}_1, k_1), (\mathbf{g}_2, k_2), \dots, (\mathbf{g}_m, k_m) \}$$

we define an operation $L(T(P),\mathbf{g})$ for an *n*-gram **g**. This operation follows the path determined by \mathbf{g}^r from the root of T(P) as far as possible and then returns the value stored at the last node reached. Thus $L(T(P),\mathbf{g})$ returns Δ or $\max\{k_i|\mathbf{g}_i=\mathbf{g}\}$. Note that $L(T(P),\mathbf{g})$ can be computed in O(n) time. In the above example, $L(T(P),\mathbf{g})$ returns the rightmost position 9 of **s** for $\mathbf{g} = bab$, the position 2 for $\mathbf{g} = xca$ where a suffix of **g** occurs as a prefix of **s**, and Δ for $\mathbf{g} = cbb$.

As the final step to complete the data structure $T(P(\mathbf{p},I))$, we adjust node values of the tree and remove the unnecessary subtrees, searching through the tree using breadth-first search (BFS).¹³

Let the value stored at the node ν after constructing $T(P(\mathbf{p},I))$ be $\mathbf{v}(\nu)$, and the value stored at the present node of ν be $\mathbf{f}(\nu)$. As the operation $L(T(P),\mathbf{g}_i)$ indicates, $\mathbf{v}(\nu)$ is in the set $\{\Delta, k_i\}$. Let $E(\nu)$ be the set of labels directed from the node ν . Note that the BFS runs through the set $E(\nu)$. For each node ν , the shift value $\mathbf{v}(\nu)$ is computed by Algorithm 1 which at the same time removes all the subtrees starting from edges labelled with \$.

Algorithm 1: tree adjustment. Perform BFS of $T(P(\mathbf{p},I))$. For each v encountered do

Case 1. If $\mathbf{v}(\nu)$ is Δ then Case 1.1. If ν is the root then $\mathbf{v}(\nu) := m$. Case 1.2. If ν is not the root then Case 1.2.1. If $\$ \in E(\nu)$ then $\mathbf{v}(\nu) := m$. Case 1.2.2. If $\$ \notin E(\nu)$ then $\mathbf{v}(\nu) := \mathbf{f}(\nu) + 1$. Case 2. If $1 \le \mathbf{v}(\nu) < m$ then Case 2.1. If $1 \le \mathbf{v}(\nu) < m$ then $\mathbf{v}(\nu) := m - \mathbf{v}(\nu) + n$. Case 2.2. If $\mathbf{v}(\nu) = m$ then $\mathbf{v}(\nu) := 0$.

Figure 2 shows the complete reverse tri-gram tree computed from Figure 1 by



Figure 1. Skeleton of a reverse n-gram tree

fast string matching

Algorithm 1 where the dotted rectangle represents the subtrees that were removed. The resulting tree will be denoted by $T_s(P(\mathbf{p},I))$. For example, the root $\mathbf{v}(v_1)$ is set to 9 by Case 1.1 of Algorithm 1, and $\mathbf{v}(v_2)$ to 9 by Case 1.2.1 and then the subtree of v_2 shown in the dotted rectangle is removed. In reality, as the leaf nodes satisfying $\mathbf{v}(v) < n$ will be removed, Algorithm 1 does not consider these nodes in Case 2. The internal node $\mathbf{v}(v_3)$ is set to 10 by Case 1.2.2, and $\mathbf{v}(v_4)$ to 0 by Case 2.2 because the node corresponds to the rightmost *n*-gram of the string \mathbf{p} .

n-gram algorithm

Algorithm 2: text-searching n-gram algorithm. The first stage of the algorithm is to build the data structure $T_s = T_s(P(\mathbf{p},I))$ for the pattern \mathbf{p} where I is taken as the set $(n-1, \ldots, 2, 1)$. The n-gram algorithm searches for the pattern as follows. Initially the text is checked leftward from the position m where m is the pattern size. Suppose now that after a number of iterations, the rightmost character of the pattern is in the text position t (initially t = m as indicated above). For the n-gram \mathbf{g} which begins at the text position t-n+1, $L(T_s, \mathbf{g})$ is computed and used as a shift value. If this shift value is zero, a repeated search loop is entered which compares the remaining leftward characters of the pattern with their corresponding positions in the text, hence detecting a possible match. For a mismatch, the Δ_2 table may be used for a possible maximal shift value, or we can simply move the pattern one position to the right.

We will now show that $L(T_s,\mathbf{g})$ returns the shift value, the sum of the backtracking distance (number of characters tested minus one) and the maximum safe jump value for the pattern. Figure 3 shows the pattern position relative to the text before and after movement, and the shift value as a black rod. We consider cases according to the value l of the backtracking distance or the size of the suffix of the *n*-gram which occurs in the pattern.

Case 1. l = 0, that is, the suffix is an empty string.

In this case the pattern does not include the character in position t and thus the whole pattern can be safely moved beyond position t. The path terminates at



Figure 2. Reverse tri-gram tree



Figure 3. Movement of pattern

the root. The shift value m is found in the root node which is set by Case 1.1 of Algorithm 1.

Case 2. $1 \le l < n$.

A suffix of the n-gram occurs in the pattern. Depending on whether it contains a prefix of the pattern as its own suffix or not, one of the following two cases is examined.

Case 2.1. A prefix (s) of the pattern is a suffix of the n-gram.

The two common substrings must be aligned not to miss a next possible match. Therefore if the shift value is calculated based on the beginning of s, this is the sum of the pattern size m rightward and the backtracking distance l - l' leftward where l' is the size of the prefix s. At first the pattern size m is set in the node corresponding to s^r by Case 1.2.1 of Algorithm 1 and then the backtracking distance l - l' is added by successive evaluations of Case 1.2.2. *Case 2.2. No prefix of the pattern is a suffix of the n-gram.*

 $L(T_s,\mathbf{g})$ returns the shift value m + l for the pattern to be beyond the *n*-gram. For this shift value, Case 1.1 of Algorithm 1 sets *m* in the root node and then this value is increased by 1 by Case 1.2.2 whenever the path goes down to the next node. Hence in all it is the same as the backtracking distance l plus m.

Case 3. n, that is, the suffix is the whole n-gram.

Case 3.1. The n-gram occurs at the end of the pattern.

 $L(T_s,\mathbf{g})$ returns the shift value 0 which is set for \mathbf{g}_m by Case 2.2 of Algorithm 1. This shift value differentiates \mathbf{g}_m from other \mathbf{g}_s , makes the pattern stay at the current text position in the repeated search loop and causes the remaining leftward characters of the pattern to be checked for a possible match.

Case 3.2. The n-gram occurs somewhere other than the end of the pattern.

 $L(T_s,\mathbf{g})$ returns the shift value to align the rightmost occurrence with the *n*-gram taking into account the backtracking distance *n* as given in Case 2.1 of Algorithm 1.

We have verified the correctness of the algorithm, because the shift value for each case is found correctly in the reverse n-gram tree, and the cases considered above deal with all the possible situations that can occur as a result of the tree searching.

If we take the size n the same as the pattern size, the tree becomes a *reverse* suffix tree of the pattern, whose leaves all have a jump value equal to the pattern size plus appropriate backtracking distance (equivalent to the depth of the leaf in the tree). The only exception is the leaf corresponding to the pattern itself which has a shift value 0 differentiating it from the others and indicating a complete match. The internal nodes are set using Algorithm 1. The searching stage is also the same as in the *n*-gram algorithm, except that in Case 3.1 no additional comparisons need to be made leftwards from the *n*-gram as a complete matching has already been detected. In this algorithm the character comparisons have been completely replaced by tree searching. By simply following a path in the tree, we determine whether the pattern occurs in the text at the current position, or we obtain a maximum jump value to continue the substring matching from a new position.

Baeza-Yates algorithm

The Baeza-Yates algorithm⁴ is called a *block* searching algorithm. By preprocessing the pattern, it initializes an $|A|^n$ size array *B* indexed by all possible *n*-grams given by

$$B[\mathbf{g}] = \min\{i | i = m - n + 1 \text{ or } (0 < i < m - n + 1 \text{ and } \mathbf{g}_{m-i-n+1} = \mathbf{g})\}$$

The searching stage for the pattern follows the same order as the BM algorithm, but for a mismatch it only looks up the entry of the array corresponding to the block of the last n characters to find out the number of positions to be moved. The original version was coded inefficiently, since for a mismatch it had to retry the same characters twice, once for comparisons and then for looking up the array (see source code in Reference 4). We have implemented the latter first, and only if the jump value is 0 do the additional character comparisons need to be made. This is achieved by setting 0 as the jump value for the last block of the pattern.

The large space requirement can be modified by masking (bitwise AND operation) and shifting the ASCII code. For example, DNA data consists of (A,C,G,T) and a space or line-feed character. If we mask the ASCII code with the bits (00000111)

the three residual bits still uniquely define the character, and thus we can economize in space and initial time, though it is still $|A|^{\log m}$ or $O(m^3)$ in this example. In general the space and initial time requirement will be $\Omega(m^t)$ where t is the number of bits required to represent each character.

EXPERIMENTS

As shown in the complexity analysis, the larger the pattern size, the faster the *n*-gram algorithm will run. In such a condition the block-searching algorithms are superior to the BM-type algorithms. Therefore, we experimented with the *n*-gram algorithm together with the block-searching algorithm (the modified Baeza-Yates algorithm described in the previous section) for large pattern sizes. The results are compared with a BM-type algorithm *uf.rev.gd2* which is the fastest algorithm for long patterns among those improved by Hume and Sunday.⁵

The block-searching algorithm was implemented very efficiently using a direct array index to calculate the jump value, while the *n*-gram algorithm uses the indirect pointer instruction in searching through the tree. Thus even though the latter searches a smaller fraction of text, it is always slower than the former when comparing user times. To overcome this weakness of the *n*-gram algorithm, each level of the reverse tree is modified to hold an index of size α^b where α is the alphabet size and *b* a block size less than *n*. The source code of the *n*-gram algorithm can easily be modified if the code of the block-searching algorithm is used to calculate the index and care is taken about resetting the shift value based on a block instead of a single character.

We sought 200 random patterns in 1.85 Mb of DNA data on a Sun4 workstation during low system workload. In Table I, the test results are given as the searching time (preprocessing time) averaged in seconds taken in seeking 200 random patterns in the same text data chosen for each size with no overlapping. These patterns also did not overlap with the patterns of different sizes. We searched for each size in the same text 40 times and the total time taken was divided by the number of runs.

In Table I, *n*-gram stands for the modified *n*-gram algorithm, BM for *uf.rev.gd2*, and block_{*i*} for the modified Baeza-Yates algorithm. The block sizes of block₁, block₂ and block₃ are 4, 5 and 6, respectively. We did not consider block sizes over 6

| Pattern size | Algorithm | | | | | | | |
|----------------|----------------|--------------------|--------------------|--------------------|-------------|--|--|--|
| r utterni size | <i>n</i> -gram | block ₁ | block ₂ | block ₃ | BM | | | |
| 300 | 4.12(3.76) | 4.81(2.88) | 4.05(2.79) | 4.80(22.10) | 14.57(0.55) | | | |
| 500 | 2.56(4.46) | 3.60(2.91) | 2.57(2.94) | 2.87(22.15) | 12.76(0.95) | | | |
| 750 | 1.85(5.26) | 3.06(3.04) | 1.88(3.07) | 1.95(22.31) | 11.22(1.25) | | | |
| 1000 | 1.48(6.02) | 2.83(3.10) | 1.57(3.10) | 1.54(22.58) | 10.36(1.71) | | | |
| 1500 | 1.15(7.22) | 2.61(3.24) | 1.30(3.31) | 1.16(22.71) | 9.65(2.54) | | | |
| 2000 | 1.02(8.63) | 2.58(3.40) | 1.20(3.65) | 1.01(22.86) | 9.24(3.54) | | | |
| 2500 | 0.95(9.78) | 2.62(3.71) | 1.18(3.83) | 0.93(23.41) | 8.93(4.14) | | | |
| 3000 | 0.92(11.05) | 2.61(3.79) | 1.18(4.09) | 0.91(23.51) | 8.50(5.05) | | | |
| 3500 | 0.90(12.16) | 2.66(3.99) | 1.18(4.23) | 0.89(23.85) | 8.37(5.79) | | | |
| 4000 | 0.92(13.23) | 2.67(4.15) | 1.20(4.54) | 0.91(23.98) | 8.11(6.68) | | | |

Table I. Searching time (preprocessing time) of DNA in seconds on Sun4 between algorithms

because the size of the table is greater than that of the searched text. However, n of the *n*-gram algorithm can be extended flexibly without too much overhead. We take n to be 7, of which the block size for the first level is 5 and for the next levels is 1. Table II shows the ratio for block/*n*-gram of the number of characters searched for calculating the jump value under column A and the ratio for comparing characters for a complete match under column B.

Tables I and II show that the fast running time of the block-searching algorithm compared with the *n*-gram algorithm is mostly obtained through its efficient use of machine instructions. The optimal algorithm for a given size of pattern depends on the size of *n* and the block size. The preprocessing time of the block-searching algorithm is mostly spent in initializing the table; for example, the table occupies 260 K for block₃. For patterns of size greater than 4000 the searching time is slower than for smaller sizes. This is caused by the fact that the increased ratio of the number of character comparisons for a complete match is more than the decreased ratio of the searched characters to find the jump value. On the whole the *n*-gram algorithm shows fast searching time with the preprocessing time increasing slowly.

CONCLUSIONS

The BM algorithm calculates a shift value by using the information traced back from the substring matched before a mismatch occurs. The block-searching algorithm calculates a shift by using the last fixed-size block, whereas the *n*-gram algorithm obtains its shift value from the position where the *n*-gram first occurs in the pattern from the right or a suffix of the *n*-gram as a prefix of the pattern.

In practice, the competitiveness of all algorithms of BM type reduces as the length of the pattern grows. Hence, although their running time improves with longer patterns, they examine too few characters at each stage to be able to take full advantage of a large pattern size. In contrast the Baeza-Yates algorithm comes into its own when searching for a long pattern, but suffers from a heavy initial time and space requirement of $\Omega(m^t)$ as demonstrated in the experimental results. The *n*-gram algorithm is flexible and the size of *n* can be adapted for a given alphabet and pattern size. In addition its space and time requirements increase gradually with the

| Pattern size | Algorithm | | | | | | | | |
|--------------|--------------------|-------|--------------------|-------|--------------------|-------|--|--|--|
| | block ₁ | | block ₂ | | block ₃ | | | | |
| | А | В | А | В | А | В | | | |
| 300 | 1.215 | 1.244 | 1.074 | 1.050 | 1.159 | 1.015 | | | |
| 500 | 1.496 | 1.121 | 1.127 | 1.030 | 1.142 | 1.014 | | | |
| 750 | 1.824 | 1.080 | 1.193 | 1.021 | 1.131 | 1.005 | | | |
| 1000 | 2.167 | 1.060 | 1.263 | 1.013 | 1.125 | 1.003 | | | |
| 1500 | 2.800 | 1.037 | 1.401 | 1.006 | 1.127 | 1.001 | | | |
| 2000 | 3.507 | 1.029 | 1.551 | 1.006 | 1.138 | 1.004 | | | |
| 2500 | 4.080 | 1.023 | 1.690 | 1.004 | 1.151 | 1.001 | | | |
| 3000 | 4.606 | 1.023 | 1.826 | 1.007 | 1.167 | 1.001 | | | |
| 3500 | 5.246 | 1.016 | 1.970 | 1.003 | 1.186 | 0.997 | | | |
| 4000 | 5.689 | 1.019 | 2.097 | 1.008 | 1.205 | 1.001 | | | |

Table II. Ratio of the searched characters (block/n-gram) of DNA data

pattern size. It can also incorporate the block-search algorithm into the tree by using a block at the first level to cover the most frequently occurring l-grams and extending l to the size of the *n*-grams in the next levels. In this way we can reduce the expensive pointer operation and get fast searching time while sacrificing only a little more space and setup time.

REFERENCES

- 1. J. Shawe-Taylor, 'Fast string matching in a stationary ergodic source', *Technical Report No CSD-TR-633*, RHBNC, University of London, 1990.
- 2. D. Welsh, Codes and Cryptography, Oxford University Press, 1988.
- 3. John Yong Kim and John Shawe-Taylor, 'Fast expected string matching using an *n*-gram algorithm', *Technical Report No CSD-TR-91-16*, RHBNC, University of London, 1991.
- 4. R. Baeza-Yates, 'Improved string matching', Software-Practice and Experience, 19, 257-271 (1989).
- 5. A. Hume and D. M. Sunday, 'Fast string searching', *Software—Practice and Experience*, **21**, 1221–1248 (1991).
- 6. R. S. Boyer and J. S. Moore, 'A fast string searching algorithm', *Commun. ACM*, **20**, (10), 762–772 (1977).
- 7. D. M. Sunday, 'A very fast substring search algorithm', Comm. ACM, 33, (8), 132-142 (1990).
- 8. D. E. Knuth, J. H. Morris and V. R. Pratt, 'Fast pattern matching in strings', SIAM. J. Comput., 6, (2), 323–350 (1977).
- 9. R. Schaback, 'On the expected sublinearity of the Boyer–Moore algorithm', SIAM J. Computing, 17, (4), 648–658 (1988).
- 10. N. Horspool, 'Practical fast searching in strings', Software—Practice and Experience, 10, 501–506 (1980).
- 11. Y. S. Ching, 'n-gram statistics for natural language understanding and text processing', *IEEE Trans.* Pattern Analysis and Machine Intelligence, 1, 164–172 (1979).
- 12. D. E. Knuth, The Art of Computer Programming, Vol. 3, Addison-Wesley, 1973.
- 13. A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.