Acta Cybernetica **14** (1999) 239-250.

A Family of Fast Constant-Space Substring Search Algorithms

Harri Hakonen* and Timo Raita†

Abstract

This paper describes a new strategy for searching a substring in a given text. The method is based on the well-known Boyer–Moore algorithm complementing it with a technique called q-slicing, a form of probabilistic q-gram matching. As a result, we get a family of highly parametric algorithms apt for adaptation to the special properties inherent to the source which generates the input strings. The search procedure is independent of the alphabet size and appropriate for efficient and practical on-line implementations. Simulation results show that they are comparable to the fastest currently known Boyer–Moore variants.

1 Introduction

In the *string searching (pattern matching) problem* our task is to determine all positions in a given text, $text[1..n]$, where the pattern, $pat[1..m]$, occurs. This problem has been studied extensively (see e.g. [1] for a good survey) and several efficient and elegant solutions have been devised. The most efficient implementations, from the practical point of view, are based on the seminal ideas of Boyer and Moore [6]. The original Boyer–Moore algorithm (BM for short) aligns the pattern with a text position j, compares the corresponding symbols of $pat[1..m]$ and $text[j-m+1..j]$ starting from the last symbol of the pattern and advancing to the left. If a mismatch (if any) is found, the pattern is shifted forward with respect to the text and the process is repeated:

*Turku Centre for Computer Science (TUCS), e-mail: `hat@cs.utu.fi`

†Department of Computer Science, University of Turku, Lemminkäisenkatu 14 A, SF-20520 Turku, Finland, e-mail: `raita@cs.utu.fi`. The author acknowledges support by the Academy of Finland under grant No. 865431

```
Boyer–Moore search(pat[1..m], text[1..n])
          Preprocess pattern.
          while all text is not scanned do
(i)           Perform skip loop.
              if witness for a match is found then
(ii)              Perform match loop.
                  if pattern is found then
                      Report match.
                  endif
              endif
(iii)         Shift pattern.
          endwhile
```

The search procedure consists of three distinct phases: (i) fast skipping over non-matching text regions, (ii) match checking when some evidence of a pattern occurrence has been found and (iii) shift to the next position. All these steps have been subject of refinements [7, 8, 10, 13, 15, 16]. A detailed analysis of the various BM substep combinations can be found in [10].

The power of the BM algorithm is largely based on a sophisticated strategy to move the pattern forward relative to the text in steps (i) and (iii). This is accomplished by forming two tables in $O(m)$ time prior to the actual search. The *match heuristic* table determines how much the pattern must be moved in order to realign the matched region of the text, $text[j-k..j]$, with an identical pattern substring also in the new position. For this, we need to determine the rightmost substring $pat[p-k..p]$, $p < m$, which is identical to the matched suffix $pat[m-k..m]$ (not overlooking the special case $0 < p < k$). In fact, when we find that $pat[m-k-1] \neq text[j-k-1]$, we know that in order to succeed at the next probe position, the condition $pat[p-k-1] = text[j-k-1]$ must also hold. Because of the large amount of space and time overhead, however, the match heuristic is usually not made so fine-grained. As a reasonable and quick approximation, it is only required that $pat[p-k-1] \neq pat[m-k-1]$. The *occurrence heuristic* table expresses the position of the rightmost occurrence of each symbol of the input alphabet Σ in the pattern. Thus, the occurrence heuristic determines, how much we can shift the pattern in order to align the mismatched text symbol with an identical pattern symbol. The length of the shift in step (iii) is then given by the maximum of the match and the occurrence heuristic values.

Let us consider the role and importance of the two heuristics. The well-known and widely used BM variant devised by Horspool [8] (BMH) discards the match heuristic due to its small significance with non-periodic patterns. This results in $O(mn)$ worst case complexity. However, on the average, the complexity is only $O(n/m)$, the same as that for the original BM method, implying that the BMH method is very fast in practice. Evidently, BMH makes shorter shifts (on the average) than BM because it uses less information. This behaviour is emphasized when σ, the size of the input alphabet, is small. On the other hand, the role of the match heuristic becomes insignificant when σ becomes large (this is studied in more detail in [14]). As suggested in [13], we should try to compensate the omission of the match heuristic in other ways during the search. One alternative is to extend the occurrence heuristic for bigrams, incorporating thus both heuristics (at least partly) into one. This idea was introduced in [16] and was shown to give improved running times, especially for small input alphabets. Moreover, if we follow the idea of BMH and choose always (independently of the position where the mismatch occurred during the right to left scan) the bigram composed of the text symbols aligning with $pat[m-1]$ and $pat[m]$, we obtain a very close approximation to the match heuristic. In the special case, where the mismatch occurs at $pat[m-2]$, they are used identically. Thus, if we can generalize the approach (as suggested in [2, 3, 12]) and use q-grams ($q > 2$) of arbitrary length, we obtain a heuristic which is a hybrid of the two original ones. However, the disadvantage of the approach is that both the preprocessing time and the space demand increase rapidly, being proportional to σ^q. In Section 2 we show how this can be avoided by retrieving only the most important information scattered around the current text position and storing it into a compact unit. After this, we give an intuitive analysis of the selection strategy which maximizes the length of the shift. Simulation results of the new search method are also given in Section 3. Concluding remarks are presented in Section 4.

2 The q-slicing method

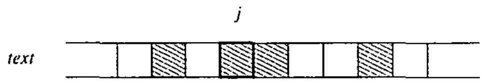
During the search, the pattern $pat[1..m]$ is always aligned with a substring $text[j-m+1..j]$, where j is the *current text position* ($m \leq j \leq n$). Thus, when we say that the current position is increased, it means that the pattern, which is considered to be positioned above the text, is shifted forward relative to the text. The information, on the basis of which the shift is made, is typically gathered from the text region $text[j-m+2..j]$; the symbol $text[j-m+1]$ does not contribute any information, because the length of the shift is always at least one. To increase the average shift length, the symbol $text[j+1]$ can also be used [15]. In the sequel, the text symbols which are used as the basis for the shift are defined by a *template* $\tau = \langle t_1, \ldots, t_q \rangle$ containing a strictly increasing sequence of integers t_k. Each t_k is an offset from the current text position j. Thus, the symbol $text[j+t_k]$ aligns with the pattern symbol $pat[m+t_k]$, iff $-m+1 \leq t_k \leq 0$. Otherwise ($1 \leq t_k \leq n-j$), the text symbol does not reside 'under' the pattern. Clearly, the elements of the template can have a chance to push the pattern forward only, if the condition $t_{k+1} - t_k \leq m$ ($k = 1, \ldots, q-1$) holds. Figure 1(a) presents an example of a template giving four offsets.

To give some insight into the efficiency of some commonly used templates, as well as some alternative choices, the following table summarizes the average shift lengths when a pattern of length 13 was searched for in an English book text (see table on page 247 for more details of this input text).

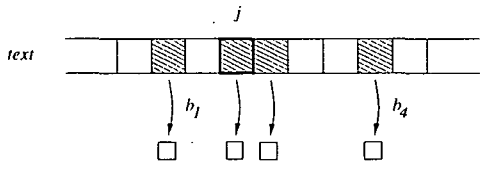
template τ	⟨0⟩	⟨-1,0⟩	⟨0,1⟩	⟨0,2⟩
average shift	9.65	12.53	13.44	13.72

template τ	⟨0,3⟩	⟨-2,-1,0⟩	⟨-1,0,1⟩	⟨0,1,2⟩
average shift	13.97	12.83	13.81	14.54

The text sampling processes defined by templates $\langle 0\rangle$ and $\langle -1,0\rangle$ are used in the Horspool [8] and Zhu–Takaoka [16] algorithms, respectively. Also, $\langle 0,1\rangle$ can be seen as a generalization of Sunday's idea to exploit $text[j+1]$ in shifting [15]. The general tendency is clear: longer templates yield longer shifts. However, a carefully selected sampling strategy compensates short templates, as can be seen e.g. from the figures for $\langle 0,3\rangle$ and $\langle -2,-1,0\rangle$.



(a) template $\tau = \langle -2,0,1,4\rangle$



(b) selective mappings $\mu = \langle b_1(text[j-2]), b_2(text[j]), b_3(text[j+1]), b_4(text[j+4])\rangle$

Figure 1: A structure of a 4-slice defined by $\tau:\mu = \langle -2{:}b_1, 0{:}b_2, 1{:}b_3, 4{:}b_4\rangle$

In order to avoid the excessive time and space requirements during preprocessing and still achieve large shifts, we combine two ideas. First, the template is kept fixed during the search. The symbols indicated by the template are picked from the text at each probe position j. Second, we reduce the size of the alphabet at the cost of losing some accuracy in symbol comparison. The idea is to partition the symbols of the input alphabet Σ into equivalence classes and tag each class uniquely with a symbol of a reduced alphabet Σ' for which $\sigma' < \sigma$. Now, instead of comparing individual symbols, we compare tags of the classes. Thus, the approach is related to the 'shift-or' algorithm of Baeza-Yates and Gonnet [4], which can be used to search for substrings composed of metacharacters representing a set of symbols from the original alphabet. However, in the new scheme the tag is formed by mapping a symbol according to its position in the sampling template. In other words, the tag of $text[j+t_k]$ is the value of the corresponding function $b_k : \Sigma \mapsto \Sigma'$ $(1 \le k \le q)$.

The collection of these *selective mappings* is defined by the vector $\mu = \langle b_1, \ldots, b_q \rangle$. The combination of τ and μ is denoted by $\tau : \mu = \langle t_1 : b_1, \ldots, t_q : b_q \rangle$. Figure 1(b) shows an example of μ for the template $\tau = \langle -2, 0, 1, 4 \rangle$. Concatenation of the (reduced) symbols defined by the current text position j and the pair $\tau : \mu$ is called a *q-slice*:

$$q\text{-slice}_{\tau:\mu}(text, j) = \operatorname*{\texttt{Concat}}_{i=1}^{q} \; b_i(text[j + t_i])$$

The q-slice scheme can be regarded as a hash function defined by τ giving the positions and μ representing the amount of information to be gathered. The equality of two slices is a necessary, but not a sufficient condition for the equality of the corresponding patterns [11]. The q-slice scheme introduces a general information sampling concept, and it has some interesting properties intrinsic to the string searching problem:

- The loss of information caused by the selective mappings is minimized, when the symbols of Σ' are uniformly distributed into the mapped sequences. Interestingly, by taking the three least significant bits from the (ASCII encoded) symbol representation has a nice property for natural languages: the most frequent symbols of the skew distribution fall into different equivalence classes. Because of this, and the fact that support for this type of operation can be found in most machine architectures, we restrict the form of the b_k functions ($k = 1, \ldots, q$) in the sequel as follows:

  $$b_k(\alpha) = \alpha \;\; \texttt{AND} \;\; 0 \ldots 011 \ldots 1, \quad \alpha \in \Sigma.$$

  The number of one-bits in the mask, lsb_k, determines how many least significant bits (LSBs) of the original symbol α we want to select. This specialized μ is called an *LSB-mask*. In what follows, each b_k function is denoted by the corresponding integer lsb_k. Naturally, other kinds of mappings are possible also, but they are not studied in this paper.

- Because the value of the hash function realized by the q-slice scheme is a concatenation of the mapped values, the function does not scramble the bits of the constituent symbols. Since important order information is preserved, it could be taken advantage of in the implementation of the search procedure: for each shift value calculated, we check whether the suffix of the previous hash value overlaps the prefix of the next one and in such a case, leads to a conflict. With a high probability this will happen, and we can increase the length of the shift. This improvement resembles the search strategy of Galil [7] and also the principles used in the construction of the BM match heuristic (cf. the description given in the Introduction). Although the proposed hash function is very simple, false matches occur rarely.

Collision probability of q-slice. Let $B = \log_2 \sigma$ and fix $B' = lsb_k$ for all $k = 1, \ldots, q$. Assuming that the text and the pattern have been generated by a uniform symbol distribution, the probability of a q-slice hash address collision is

$$P(q\text{-slice matches} \mid \text{pattern mismatches}) = \frac{2^{q(B-B')} - 1}{2^{qB} - 1}.$$

Proof. This follows from the probabilities $P(q\text{-slice matches}) = 2^{-qB'}$ and $P(\text{pattern mismatches} \mid q\text{-slice matches}) = 1 - 2^{-q(B-B')}$. ■

A simple but good approximation for this formula is the q-slice matching probability $2^{-qB'}$. For example, if $q = 2$, $B = 8$ and $B' = 3$, then $P(\text{collision}) \approx 0.0156$.

- The q-slice can be used efficiently for searching when its length, $lsb_1 + \ldots + lsb_q$, is conveniently chosen to fill a machine-dependent unit, e.g. a byte or a word, and the sampling strategy is supported by the architecture. Unfortunately, the latter is often true only for the trivial case $q = 1$. Therefore, this parameter is usually chosen to be small and a balanced intertwining of τ and μ becomes crucial. This is discussed in more detail in section 3.

The actual search procedure starts by preprocessing the pattern as follows. In the description below, we use the notation $\bullet_r | \bullet_{r+1} | \ldots | \bullet_s$ $(1 \leq r \leq s \leq q)$, when we want to refer to a part of a q-slice. Also, the set of all possible bit combinations of length lsb_k for the k'th component is denoted by $*_k$. For simplicity, we shall assume that there exists an index i for which $-m + 2 \leq t_i \leq 1$. Without this restriction, the algorithm would contain unnecessary details obscuring the basic idea; these special cases can be easily incorporated into the scheme by analysing them carefully (left as an easy exercise to the reader).

```
Define a proper template $\tau$ (of length $q$) and LSB-mask $\mu$.
shiftLength[$*_1 | *_2 | \ldots | *_q$] := $m + t_q$
        // Each of the $2^{\sum_{i=1}^{q} lsb_i}$ table values is initialized to the maximal shift.
for $c := m + t_q - 1$ downto 1 do
    Find all $t_k$ values in range $-m + 1 + c..m + c$.
        // These are characterized by indices $r$ and $s$ for which $t_r \leq t_k \leq t_s$.
        // Each $t_k$ aligns with $pat[t_k - (m + t_q - c)]$ after a shift of
        // length $c$. Thus, we call these offsets bound (template)
        // positions. The others (i.e. $t_1, \ldots, t_{r-1}$ and $t_{s+1}, \ldots, t_q$)
        // are free positions.
    Determine the unique part of the $q$-slice $\bullet_r | \bullet_{r+1} | \ldots | \bullet_s$ corresponding
        to the bound positions.
    shiftLength[$*_1 | \ldots | *_{r-1} | \bullet_r | \ldots | \bullet_s | *_{s+1} | \ldots | *_q$] := $c$
```

```
        // Make $2^{\sum_{i=1}^{r-1} lsb_i + \sum_{i=s+1}^{q} lsb_i}$ updates.
    endfor
```

After preprocessing, the search is performed by mapping the text symbols to the reduced alphabet on-the-fly using the q-slice$_{\tau:\mu}(text, j)$. The slice contains information which is scattered into a large region near the current context. This gives a basis to increase the average length of a shift using only a small amount of comparisons. A minor drawback of the approach is, that when we encounter a q-slice match, we must confirm that an identical symbol pattern has been found.

Example. Let us assume that the symbols are ASCII encoded and that the pattern is $pat[1..14]$ = *abracadabracab*. The two least significant bits of the symbols '*a*', '*b*', '*c*', '*d*' and '*r*' are '01', '10', '11', '00' and '10', respectively. The template $\langle -1, 0, 1\rangle$ and the LSB-mask $\langle 2, 1, 1\rangle$ generate 16 different q-slices and their corrsponding shift lengths:

q-slice	00|0|0	00|0|1	00|1|0	00|1|1	01|0|0	01|0|1	01|1|0	01|1|1
shift	15	14	6	14	5	7	13	2

q-slice	10|0|0	10|0|1	10|1|0	10|1|1	11|0|0	11|0|1	11|1|0	11|1|1
shift	15	4	13	3	15	14	1	14

For example, the shift value for the q-slice 10|0|1 is 4 because $pat[10..12]$ = *rac* is the rightmost pattern region that matches it. Obviously, the shift values are always in the range $1..m + t_q$. Assuming that the pattern and text are aligned as in (a):

pat	... $a\ c\ a\ b$		*pat*	... $a\ d\ a\ b\ r\ a\ c\ a\ b$
text	... $a\ c\ \underline{a\ d\ b}\ a\ r\ b\ a\ c$ ...		*text*	... $a\ c\ \underline{a\ d\ b}\ a\ r\ \underline{b\ a\ c}$ ...
	(a)			(b)

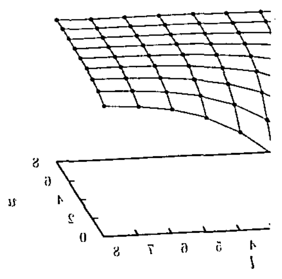
we find that q-slice 01|0|0 obtained from the text symbols a, d, b tells us to shift the pattern 5 positions forward (as shown in (b)) in order to align *abr* with *adb*. No pattern occurrence is found here either, and the template symbols b, a, c generate the q-slice 10|1|1 yielding a new shift of length 3.

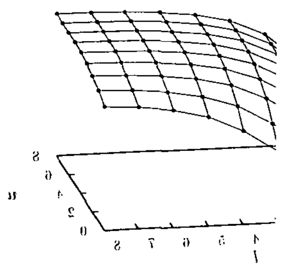
3 Experiments

Expectation of the shift length. The basic structure of the BM algorithm, given in the introduction, shows that before the pattern shift is made in step (iii), we have gathered a lot of information about the symbols near the current text position.

In this experimental analysis, however, we assume that no such information is available. To obtain a realiable comparative analysis and at the same time keep the various parameter combinations practically feasible, we restrict ourselves to a 2-slice of type $\tau : \mu = \langle 0 : \ell, (D + 1) : u \rangle$. In other words, the pattern is shifted according to the ℓ-bit tag of *text*$[j]$ and u-bit tag of *text*$[j + D + 1]$. The ℓ bits are extracted from the text symbol which resides under the ℓast pattern symbol. This information produces an 'initial shift' for the pattern. The $(D + 1) : u$ component gives an 'additional push', since it probes further information from an $u$pcoming text symbol at the distance D from the current text position. This special algorithm family is denoted by $\ell \lhd D \rhd u$, where $\lhd \cdots \rhd$ symbolizes the distance between the two units from which the bits are extracted.

Let us study the expectation of the shift lengths for the $\ell \lhd D \rhd u$ algorithm when $m = 13$; $\ell, u = 0, \ldots, 8$ and $D = 0, 1, m/2$. To analyze the behaviour of the expectation for natural languages, a simulation over an English book text (see table on page 247) was accomplished. The search was performed for 30 randomly selected patterns from the text. Figure 2 shows the expectation of the shift length based on this test arrangement.


(a) $D = 0$


(b) $D = 1$

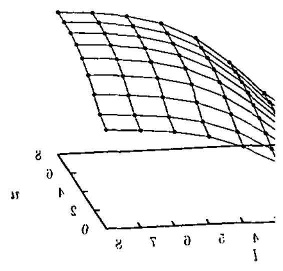

(c) $D = 6$

Figure 2: The expectation of the shift length for $\ell \lhd D \rhd u$ algorithms

The expectation behaves differently for the following two cases.

Case $D = 0$. The average shift values are almost symmetrical wrt the diagonal $\ell = u$ (Fig 2(a)). For the region $\ell + u \geq 6$, the expected shift length is always ≥ 12. This suggests that even a shift table of size 64 gives a good performance for English text.

Case $D \geq 1$. The shape of the expectation function differs from the previous case: whenever $\ell = 0$, we have now no information to use any other shift length except 1. The influence of the parameter ℓ is more significant than that of u and only with parameter values $\ell + u \geq 7$ constrained by $\ell \geq 4$, we reach the average shift of at least 12 positions (Fig 2(b,c)). Referring back to figure 2(a), we can observe that if $u = 0$ and ℓ approaches the length of the original encoding of $text[j]$, we quite quickly reach the situation where the shifts are larger than $m/2$. Comparing this with the results of Fig2(c), we can see how much more the upcoming symbol is able push the pattern forward once the initial push has been given. This also explains why we can have an average shift length which is significantly larger than m, the length of the pattern.

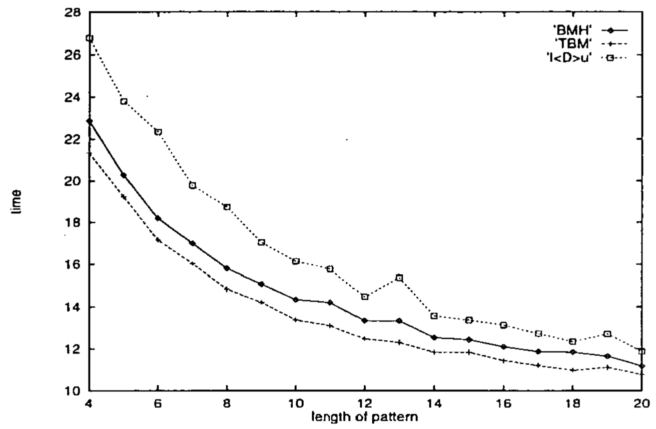
Running times. After extensive test runs, we suggest the schemes $4 \lhd 0 \rhd 2$ and $3 \lhd 0 \rhd 3$ as general purpose substring searching algorithms. Evidently, if the properties of the input strings differ significantly from those of English, some other parameter values may result in better performance. Furthermore, the implementations of the fastest currently known search algorithms are extremely carefully designed and the hardware architecture may have a large effect on their speed.

The $4 \lhd 0 \rhd 2$ algorithm was tested and compared to the basic Horspool variant BMH [8] and to the Hume–Sunday variant TBM [10]. To our knowledge, TBM is one of the fastest, widely known algorithms for natural language text search. Tests were run on a Sparc machine (architecture `sun4m`, kernel `SunOS 5.6`) and the C programs were compiled with gcc (version 2.7.2.3) using the optimization switch `-O3`.

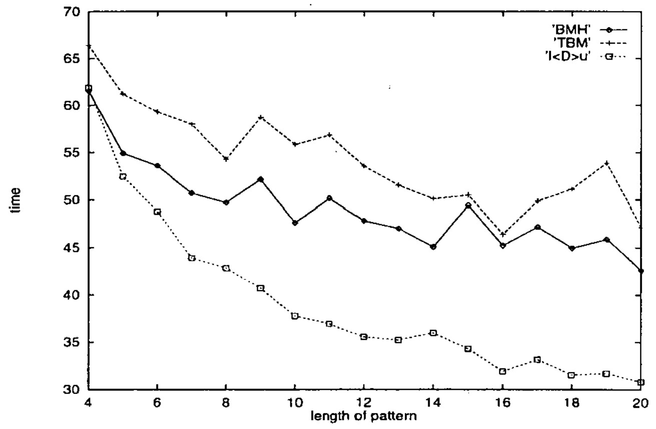
The input data consisted of the English book text and a dna text, having the following properties.

text type	**source**	**used file**	σ	**length**
English book	Calgary Corpus [5]	`book2`	96	611 kB
Dna sequence	[10]	`dna.test`	4	988 kB

The simulation was accomplished by selecting 30 patterns of length m randomly in the text and then searching for all of their occurrences in the text. This was repeated for $m = 4, \ldots, 20$. To make the comparison fair, the running times include both searching and preprocessing phases. Figure 3 shows the results of this test set.



(a) English text



(b) Dna sequence

Figure 3: The running times of BM, BMH, $4 \lhd 0 \rhd 2$ and $2 \lhd 0 \rhd 2$ algorithms

The running times of the $4 \lhd 0 \rhd 2$ algorithm are quite modest for natural languages (see Fig 3(a)). This is due to the hardware architecture, which does not support multicharacter sampling. However, the shape of the curve of the new scheme shows that the information obtained from a q-slice is at least as good as if we used more local, but exact information. When the size of the input alphabet is decreased, the 'traditional' methods begin to lose their power because the machine-level size of a symbol unit is typically kept fixed although the information content of a unit is smaller. This deficiency is handled in the new method, as it gathers and utilizes data of size q extracted from the neighbourhood of the current position. The effect can be seen strikingly in Fig. 3(b): the form of the curve for the q-slice method remains identical to that for large alphabets whereas the smoothness of the performance for the two other methods disappears. Moreover, it is not only the $O(n/m)$ behaviour which is lost but BMH and TBM are also clearly much slower than the new method. Since $\sigma = 4$ for dna sequences, $2 \lhd 0 \rhd 2$ was chosen as the representative of the new approach (Fig 3(b)). As a final remark, recall that the running times of the $\ell \lhd D \rhd u$ algorithms are independent of Σ, unlike the BMH and TBM methods.

4 Summary

A new family of fast substring searching algorithms using q-slices is devised. The concept of a q-slice combines the idea of using q-grams together with the mapping of symbols to a reduced alphabet. This new strategy makes on-line text sampling to skip fast over regions where the pattern cannot occur. In spite of the fact that most machine architectures do not support the core operation of q-slicing on the hardware level, the efficiency of the new method is comparable to the fastest known substring search algorithms. Tests have shown that this approach typically results in average shift lengths which are even larger than the size of the pattern. This algorithm family is highly parametric and can thus easily be adapted to specific application environments when necessary.

References

[1] Baeza-Yates, R.A.: Algorithms for String Searching: A Survey, *SIGIR Forum, Spring/Summer 1989, Vol. 23, No. 3,4, pp. 34-58*

[2] Baeza-Yates, R.A.: Improved String Searching, *Softw. Pract. Exp., Vol. 19, No. 3, March 1989, pp. 257-271*

[3] Baeza-Yates, R., Krogh, F.T., Ziegler, B., Sibbald, P.R. & Sunday, D.M.: Notes on a Very Fast Substring Search Algorithm, *Comm. ACM, Vol. 35, No. 4, April 1992, pp. 132-137*

[4] Baeza-Yates, R.A. & Gonnet, G.H.: A New Approach to Text Searching, *Proc. of the SIGIR Conference 1989, pp. 168-175*

[5] Bell, T.C., Cleary, J.G. & Witten, I.H.: Text Compression, *Prentice-Hall, 1990*

[6] Boyer, R.S. & Moore, J.S.: A Fast String Searching Algorithm, *Comm. ACM, Vol. 20, No. 10, October 1977, pp. 762-772*

[7] Galil, Z.: On Improving the Worst Case Running Time of the Boyer–Moore String Matching Algorithm, *Comm. of the ACM, Vol. 22, No. 9, September 1979, pp. 505-508*

[8] Horspool, R.N.: Practical Fast Searching in Strings, *Softw. Pract. Exp., Vol. 10, 1980, pp. 501-506*

[9] Galil, Z. & Seiferas, J.: Time-Space-Optimal String Matching, *J. Comput. System Sci., Vol. 26, 1983, pp. 280-294*

[10] Hume, A. & Sunday, D.M.: Fast String Searching, *Softw. Pract. Exp., Vol. 21, No. 11, November 1991, pp. 1221-1248*

[11] Karp, R.M. & Rabin, M.O.: Efficient Randomized Pattern-matching Algorithms, *IBM J. Res. Develop., Vol. 31, No. 2, March 1987, pp. 249-260*

[12] Knuth, D.E., Morris, J.H. & Pratt, V.R.: Fast Pattern Matching in Strings, *Siam J. Comput., Vol. 6, No. 2, June 1977, pp. 323-350*

[13] Raita, T.: Tuning the Boyer–Moore–Horspool String Searching Algorithm, *Softw. Pract. Exp., Vol. 22, No. 10, October, 1992, pp. 879-884*

[14] Tarvainen, H.: A Theoretical Framework for the Substring Searching Algorithms, *M.Sc. Thesis (in Finnish), University of Turku, Finland, May 1995*

[15] Sunday, D.M.: A Very Fast Substring Search Algorithm, *Comm. of the ACM, Vol. 33, No. 8, August 1990, pp. 132-142*

[16] Zhu, R.F. & Takaoka, T.: On Improving the Average Case of the Boyer–Moore String Matching Algorithm, *J. of Inf. Proc., Vol. 10, No. 3, 1987, pp. 173-177*