Multipattern String Matching with q-Grams

LEENA SALMELA, JORMA TARHIO, and JARI KYTÖJOKI
Helsinki University of Technology

We present three algorithms for exact string matching of multiple patterns. Our algorithms are filtering methods, which apply q-grams and bit parallelism. We ran extensive experiments with them and compared them with various versions of earlier algorithms, e.g., different trie implementations of the Aho–Corasick algorithm. All of our algorithms appeared to be substantially faster than earlier solutions for sets of 1,000–10,000 patterns and the good performance of two of them continues to 100,000 patterns. The gain is because of the improved filtering efficiency caused by q-grams.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Pattern matching*; H.3.3 [**Information Systems**]: Information Storage and Retrieval—*Search process*

General Terms: Algorithms

Additional Key Words and Phrases: Content scanning, intrusion detection, multiple string matching

1. INTRODUCTION

We consider exact string matching of multiple patterns. Many good solutions have been presented for this problem, e.g., Aho–Corasick [Aho and Corasick 1975], Commentz–Walter [Commentz–Walter 1979; Navarro and Raffinot 2002], and Rabin–Karp [Karp and Rabin 1987; Muth and Manber 1996] algorithms with their variations. However, most of the earlier algorithms have been designed for pattern sets of moderate size, i.e., a few dozens. Unfortunately, they do not scale very well to larger pattern sets. In this paper, we concentrate on practical methods that can efficiently handle several thousand patterns even in a small main memory (e.g., a handheld device). Such algorithms are needed for anti-virus scanning, intrusion detection [Fisk and Varghese 2001; Markatos et al. 2002; Tuck et al. 2004], content scanning

An earlier version of this paper appeared in the *Proceedings of 14th Combinatorial Pattern Matching* (*CPM'03*) (Jun. 25–27, 2003), pp. 211–224.
Work by Jorma Tarhio was supported by Academy of Finland.
Authors' address: Leena Salmela, Forma Tarhio, and Jari Kytöjoki, Helsinki University of Technology, P.O. Box 5400, FI-02015 HUT, Finland; email: leena.salmela@hut.fi.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2006 ACM 1084-6654/2006/0001-ART1.1 $5.00 DOI 10.1145/1187436.1187438 http://doi.acm.org 10.1145/1187436.1187438

and filtering, and specific data-mining problems [Gum and Lipton 2001]. The focus is mainly on finding the occurrences of rare patterns or on checking that unwanted patterns do not occur at all.

The text $T = t_1 t_2 \cdots t_n$ is a string of n characters in an alphabet of size c. There are r patterns $P_1, \ldots, P_r$ of length m in the same alphabet. If the lengths of the patterns are not equal, we select a substring from each pattern according to the length of the shortest pattern. We consider cases where m varies between 4 and 32 and r between 100 and 100,000. All exact occurrences of the patterns should be reported.

As our main contribution, we will present three algorithms HG, SOG, and BG, based on search algorithms for a single pattern: Boyer–Moore–Horspool [Horspool 1980], shift-or [Baeza-Yates and Gonnet 1992], and BNDM [Navarro and Raffinot 2000] algorithms, respectively. Ours are filtering algorithms, which operate in three phases. The patterns are first preprocessed. The second phase reports candidates for matches, which are verified in the third phase. A common feature of our algorithms is matching of q-grams instead of single characters. We search for occurrences of a single generalized pattern of q-grams such that the pattern includes all the original patterns. In addition, SOG and BG apply bit parallelism. Related methods for a single pattern have been suggested by Fredriksson [2002]. It is well known (see e.g., [Baeza-Yates 1989; Boyer and Moore 1977]) that the use of q-grams can increase the average length of shift in the algorithms of Boyer-Moore type. This can also be applied to matching of multiple patterns [Wu and Manber 1994]. We use q-grams in a different way in order to improve filtration efficiency.

In order to show the applicability of our algorithms, we ran extensive tests and compared them with various implementations of earlier algorithms. We used a random text, which ensures the rarity of matches in our setting. In practice, our algorithms showed to be very fast. For example, HG is 16 times faster than the well-known Aho–Corasick algorithm, in the case of random patterns for $r = 10{,}000$, $m = 8$, and $c = 256$. In addition, the filtering phase of our algorithms does not require much memory: 64 kB is sufficient in the specified case. The filtering efficiency of our algorithms will continue beyond 100,000 patterns, if more memory is used.

The following test setting was used in our experiments (if not otherwise stated). We used a 32-MB randomly created text in the alphabet of 256 characters. The patterns were also randomly generated in the same alphabet. The times are averages over 10 runs using the same text and patterns. Both the text and the patterns reside in the main memory in the beginning of each test in order to exclude reading times. The tests were made on a computer with a 1.0 GHz AMD Athlon processor, 512 MB of memory, and 256 kB on-chip cache. The computer was running Linux 2.4.22. The algorithms were written in C and compiled with the gcc compiler.

2. BACKGROUND

This section presents background information necessary to understand our new algorithms. Our algorithms extensively apply q-grams and bit parallelism.

These techniques are first discussed. We then present the Boyer–Moore–Horspool [Horspool 1980], the shift-or [Baeza-Yates and Gonnet 1992] and the BNDM [Navarro and Raffinot 2000] algorithms, which are exact matching algorithms for single patterns.

2.1 q-Grams

Many string matching algorithms rely on a fairly large alphabet for good performance. The idea behind using q-grams is to make the alphabet larger. When using q-grams, we process q characters as a single character. There are two ways of transforming a string of characters into a string of q-grams. We can either use overlapping q-grams or consecutive q-grams. When using overlapping q-grams, a q-gram starts at every position of the original text, while with consecutive q-grams a q-gram start in every qth position. For example transforming the word "pony" into overlapping 2-grams results in the string "po-on-ny" and transforming it into consecutive 2-grams yields the string "po-ny." We tried both overlapping and consecutive q-grams in our algorithms and clearly obtained better results with overlapping q-grams. Thus all new algorithms presented in this paper use overlapping q-grams.

2.2 Bit Parallelism

Bit operations in modern processors are fast. Bit parallelism takes advantage of this fact by packing several variables into a single computer word. These variables can then be updated in a single instruction making use of the intrinsic parallelism of bit operations. For example, if we needed to keep track of $m \leq w$ boolean variables, where w is the length of the computer word, we could store all these variables in a single computer word. Furthermore, we can update all the variables in one instead of m instructions. As the length of the computer word in modern processors is 32 or 64, this technique can give us a significant speedup.

2.3 Boyer–Moore–Horspool

The Boyer–Moore–Horspool (BMH) algorithm [Horspool 1980] is a widely known search algorithm for a single pattern. The preprocessing phase of the algorithm consists of calculating the bad character function $B(x)$. It is defined as the distance from the end of the pattern $p_1 p_2 \cdots p_m$ to the last occurrence of the character x:

$$B(x) = \min\{h \mid p_{m-h} = x, h \geq 1\}$$

if the character x does not appear in the pattern $B(x) = m$.

In the matching phase, the text is processed in windows of length m. First, the last character in the window is compared with the last character of the pattern. If they match, the whole window is compared against the pattern to check for a match. After that or if the last characters did not match, the window is shifted by $B(x)$, where x is the last character of the window.

2.4 Shift–Or

The shift–or algorithm [Baeza-Yates and Gonnet 1992] is a simple bit-parallel algorithm. In the preprocessing phase, a bit vector $B[x]$ is initialized for each character x of the alphabet. The bit in position i is set to 0 in the bit vector if the ith character in the pattern is x. Otherwise the bits are set to one.

In the beginning of the matching phase, the state vector E is initialized to 1^m. The text is then read, one character at a time, and the state vector is updated as follows:

$$E = (E \ll 1)|B[x]$$

where x is the character read, $\ll$ moves the bits to the left, and | is the bitwise or operator. If the mth bit is zero after this update, we have found a match.

2.5 Backward Nondeterministic DAWG Matching

The Backward Nondeterministic DAWG Matching (BNDM) algorithm [Navarro and Raffinot 2000] has been developed from the backward DAWG Matching (BDM) algorithm [Crochemore and Rytter 1994]. In the BDM algorithm, the pattern is preprocessed by forming a DAWG (directed acyclic word graph) of the reversed pattern. The text is processed in windows of size m, where m is the length of the pattern. The window is searched from right to left with the DAWG for the longest prefix of the pattern. When this search ends, we have either found a match (i.e., the longest prefix is of length m) or the longest prefix. If a match was not found, we can shift the start position of the window to the start position of the longest prefix. If a match was found, we can shift on the second longest prefix (the longest one is the match we just found).

The BNDM algorithm [Navarro and Raffinot 2000] is a bit-parallel simulation of the BDM algorithm. It uses a nondeterministic automaton instead of the deterministic one in the BDM algorithm. For each character x, a bit vector $B[x]$ is initialized in the preprocessing phase. The ith bit is 1 in this vector, if x appears in the reversed pattern in position i. Otherwise the ith bit is 0. The state vector D is initialized to 1^m. The same kind of right to left scan in a window of size m is performed, as in the BDM algorithm. The state vector is updated in a similar fashion as in the shift–and algorithm [Abrahamson 1987; Baeza-Yates and Gonnet 1992; Wu and Manber 1992b]. If the mth bit is 1 after this update operation, we have found a prefix starting at position j, where j is the number of updates done in this window. If j is the first position in the window, a match has been found.

3. EARLIER SOLUTIONS

Many of the earlier algorithms for multiple pattern matching build a pattern trie in the preprocessing phase and use it for matching. For example, the Aho–Corasick algorithm [Aho and Corasick 1975], the Commentz–Walter-based algorithms [Commentz–Walter 1979] and the SBOM algorithm [Navarro and Raffinot 2002] take this approach. While this works reasonably well for a small set of patterns, the memory requirements for huge pattern sets are intolerable, because the trie data structure grows quite rapidly.

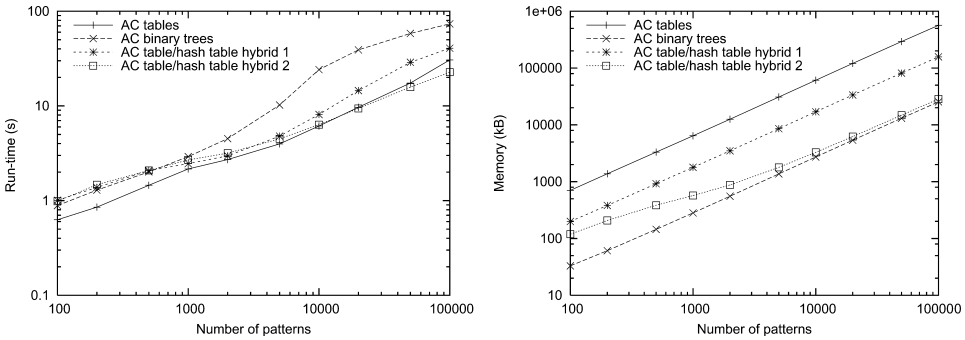


Fig. 1. Performance of different trie implementations of the Aho–Corasick algorithm. The table/hash table hybrid 1 uses tables in the first two levels of the trie and hash tables of size 64 deeper. The second table/hash table hybrid uses tables in the first three levels and hash tables of size four deeper.

Another early solution is to use hashing algorithms. For example, the Rabin–Karp algorithm [Karp and Rabin 1987] can be extended to multiple patterns. This approach leads to modest memory requirements, but the running time is not much better than in the trie-based approaches. Another hashing approach is described in Kim and Kim [1999].

An attempt to combine the best parts of the previous solutions is described in Ping et al. [2005]. In this solution, the pattern set is partitioned based on the length of the patterns and the best possible algorithm for each subset is then used.

3.1 Aho–Corasick

The classical Aho–Corasick algorithm [Aho and Corasick 1975] has been widely used for multiple pattern matching. We used a code based on the implementation by Fisk and Varghese [2001] to test the Aho–Corasick algorithm. We tested three alternative implementations of the goto-function: table, hash table, and binary tree. The hash table version was tested with different table sizes. We also tried a combination of table and hash table implementations. In this approach, the table version was used in the first levels of the trie, while in deeper levels the hash table implementation was utilized. Figure 1 shows the results of these experiments.

Although the speed of the Aho–Corasick algorithm is constant for small pattern sets, the situation is different for large sets even in an alphabet of moderate size. The run-time graph of Figure 1 shows a steady increase. Given the memory graph of Figure 1, the hierarchical memory could explain this behavior. The table implementation of the trie was fastest up to 10,000 patterns. After that the best approach turned out to be using the table implementation in the first three levels of the trie and to use a hash table of size four in further levels.

3.2 Set Horspool

The Commentz–Walter algorithm [Commentz–Walter 1979] for multiple patterns has been derived from the Boyer–Moore algorithm [Boyer and Moore 1977]. A simpler variant of this algorithm is called Set Horspool

[Navarro and Raffinot 2002]. (The same algorithm is called set-wise Boyer–Moore in Fisk and Varghese [2001].) This algorithm is developed from the Boyer–Moore–Horspool algorithm [Horspool 1980] for single patterns by generalizing the bad character function. Which is defined as the minimum of the bad character functions of individual patterns.

The reversed patterns are stored in a trie. The initial endpoint is the length of the shortest pattern. The text is compared from right to left with the trie until no matching entry is found for a character in the text. The bad character function is then applied to the endpoint character and the pattern trie is shifted accordingly.

The Wu–Manber algorithm [Wu and Manber 1994] is a variation of the Set Horspool algorithm. It uses a hash table of the last q-grams of patterns. The agrep tool [Wu and Manber 1992a] is a collection of different algorithms. It uses the Wu–Manber algorithm for exact matching of multiple patterns.

We used the code of Fisk and Varghese [2001] to test the Set Horspool algorithm. The same variations as for the Aho–Corasick algorithm were tried. The results on memory usage were similar to those of the Aho–Corasick algorithm because the trie structure is very similar. The test results on run times also resemble those of the Aho–Corasick algorithm, especially with very large pattern sets. This is probably because of the memory usage. Differences with less than 1000 patterns were not significant between modifications.

3.3 Set Backward Oracle Matching

The third algorithm making use of a trie is the Set Backward Oracle Matching (SBOM) algorithm [Navarro and Raffinot 2002]. In the preprocessing phase of the SBOM algorithm, first a trie of the reversed patterns is built. Some additional transitions are then added to the trie so that at least all factors of the patterns can be recognized with the resulting factor oracle. In the matching phase, the text is scanned backward with the factor oracle. If the oracle fails to recognize a factor at a given position, we can shift the pattern beyond that position.

We also ran tests on the SBOM algorithm. The same variations for the implementation of the trie were tried. The hashing approach proved to be quite slow with SBOM, because the hash tables need to have a more complicated structure. In the trie built by the SBOM algorithm a node can have several incoming links. This means that another structure is needed to implement the chaining of colliding hash table entries, while in the tries built by the AC and Set Horspool algorithms, such a structure is not needed. Thus, the table implementation of the trie turned out to be the fastest.

3.4 Rabin–Karp Approach

A well-known solution [Gum and Lipton 2001; Muth and Manber 1996; Zhu and Takaoka 1989] to cope with large pattern sets with less memory is to combine the Rabin–Karp algorithm [Karp and Rabin 1987] with binary search. During preprocessing, hash values for all patterns are calculated and stored in an ordered table. Matching can then be done by calculating the hash value for

each m-character string of the text and searching the ordered table for this hash value using binary search. If a matching hash value is found, the corresponding pattern is compared with the text. We implemented this method for $m = 8$, 16, and 32. The hash values for patterns of eight characters are calculated as follows. First, a 32-bit integer is formed of the first four bytes of the pattern and another from the last four bytes of the pattern. These are then xor'ed together, resulting in the following hash function where "^" denotes the xor operation:

$$Hash(x_1 \dots x_8) = x_1x_2x_3x_4\hat{}x_5x_6x_7x_8$$

The hash values for $m = 16$ and 32 are calculated in a similar fashion:

$$Hash16(x_1 \dots x_{16}) = (x_1x_2x_3x_4\hat{}x_5x_6x_7x_8)\hat{}(x_9x_{10}x_{11}x_{12}\hat{}x_{13}x_{14}x_{15}x_{16})$$
$$Hash32(x_1 \dots x_{32}) = ((x_1x_2x_3x_4\hat{}x_5x_6x_7x_8)\hat{} \dots \hat{}(x_{25}x_{26}x_{27}x_{28}\hat{}x_{29}x_{30}x_{31}x_{32}))$$

Muth and Manber [1996] use two-level hashing to improve the performance of the Rabin–Karp method. The second hash is calculated from the first one by xor'ing together the lower 16 bits and the upper 16 bits. At preprocessing time, a bitmap of 2^{16} bits is constructed. The ith bit is zero, if no pattern has i as its second hash value, and one, if there is at least one pattern with i as its second hash value. When matching, one can quickly check from the bit table, when the first hash value does not need further inspection, thus avoiding the time-consuming binary search in many cases. In the following, we use the shorthand RKBT for the Rabin–Karp algorithm combined with binary search and two-level hashing.

The Rabin–Karp approach was tested both with and without two-level hashing. The use of the second hash table of 2^{16} bits significantly improves the performance of the algorithm when the number of patterns is less than 100,000. When there are more patterns, a larger hash table should be considered, because this hash table tends to be full of 1's and the gain of two-level hashing disappears.

3.5 Comparison of the Earlier Algorithms

Figure 2 shows a comparison of the earlier algorithms. The times do not include preprocessing. It also contains tests with the agrep tool [Wu and Manber 1992a]. Since agrep is row-oriented, some characters, like newline, were left out of the alphabet. In the agrep tool, lines are limited to 1024 characters so we chopped the text to lines, each containing 1024 characters.

In the experiments of Navarro and Raffinot [2002], agrep was the fastest algorithm for 1000 patterns for $m = 8$. This holds true also for our experiments (excluding the new algorithms). The agrep tool is the fastest up to 1000 patterns, RKBT is fastest between 1000 and 10000 patterns, and the SBOM algorithm is fastest with more than 10,000 patterns.

4. MULTIPATTERN HORSPOOL WITH Q-GRAMS

Besides the Set Horspool approach, the Boyer–Moore–Horspool algorithm [Horspool 1980] can be applied to multiple patterns in another way. We call

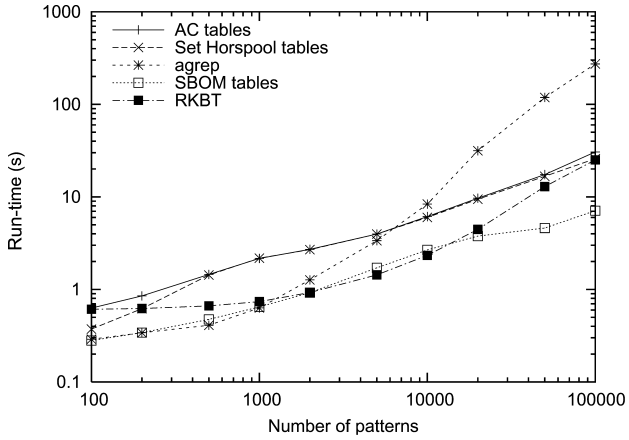


Fig. 2. Run-time comparison of the earlier algorithms.

```
1-gram tables:
1.  2.  3.  4.  5.  6.
q   q   q   q   q   q
    w   w   w   w   w
        e   e   e   e
            r   r   r
                t   t
                    y
```

(a)

```
HGMatcher(T, n)
  i = 0;
  while(i < n-6)
    j = 6;
    while (1)
      if (not 1GramTable[j][T[i+j]])
        i = i+j;
        break
      else if (j = 0)
        Verify-match(i);
        i = i+1;
        break
      else
        j = j-1
```

(b)

Fig. 3. The HG algorithm: (a) the data structures for the pattern “qwerty” and (b) the pseudocode for $m = 6$.

the resulting filtering algorithm HG (short for Horspool with q-Grams). Given patterns of m characters, we construct a bit table for each of the m pattern positions in the preprocessing phase. The first table keeps track of characters appearing in the first position in any pattern, the second table keeps track of characters appearing in the first or second position in any pattern and so on. Figure 3a shows the six tables corresponding to the pattern “qwerty.”

These tables can then be used in the filtering phase as follows. First the mth character is compared with the mth table. If the character does not appear in this table, the character cannot appear in positions $1 \ldots m$ in any pattern and a shift of m characters can be made. If the character is found in this table, the $m-1$th character is compared to the $m-1$th table. A shift of $m-1$ characters can be made if the character does not appear in this table and, therefore, not in any pattern in positions $1, \ldots, m-1$. This process is continued until the algorithm has advanced to the first table and found a match candidate there.

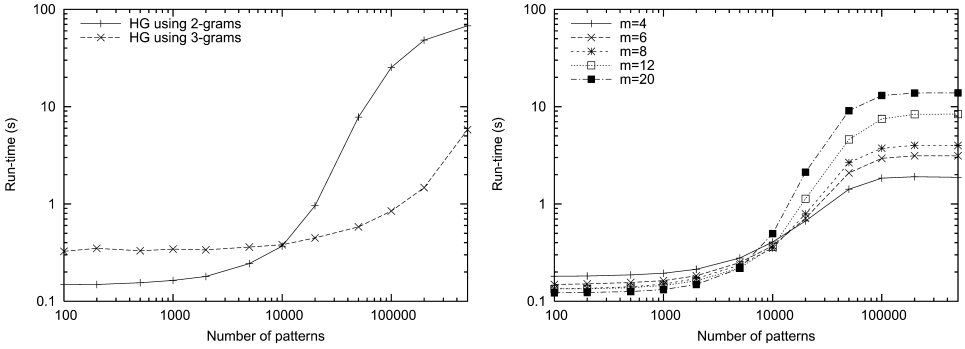


Fig. 4. The HG algorithm: (a) comparison of 2- and 3-gram versions and (b) run-times of the 2-gram version for different pattern lengths.

The pseudocode for $m = 6$ is shown in Figure 3b. Given this procedure, it is clear that all matches are found. However, false matches can also occur, e.g., "qqqqqq" is a false candidate in the example of Figure 3a. In the checking phase, the candidates are verified by using the RKBT method described in Section 3.4.

As the number of patterns grow, the filtering efficiency of the above scheme decreases until almost all the text positions are candidates, because there are only c different characters. A substantial improvement in the filtering efficiency can be achieved by using q-grams, $q \geq 2$, instead of single characters, since there are c^q different q-grams. For an alphabet with 256 characters and for $q = 2$, this means that the alphabet size grows from 256 to 65,536. When using 2-grams, a pattern of m characters is transformed into a sequence of $m - 1$ 2-grams. Thus the pattern "qwerty" would yield the 2-gram string "qw-we-er-rt-ty." The HG algorithm can be applied to these 2-grams just as it was applied to single characters. With even larger pattern sets, 3-grams could be used instead of 2-grams. Because this would require quite a lot of memory, we implemented a 3-gram version of the algorithm with a hashing scheme. Before forming a 3-gram, each character is hashed to a 7-bit value. This diminishes the number of different 3-grams from 2^{24} to 2^{21}.

4.1 Experiments with the HG Algorithm

The HG algorithm was tested both with the 2- and 3-gram versions for $m = 8$ (see Figure 4a). The 3-gram version is faster when the pattern set size is greater than 10,000. This is because of the better filtering efficiency of the 3-gram approach. However, when there are less than 10,000 patterns, the 2-gram version is much faster because of the hashing overhead and memory requirement of the 3-gram approach.

We also tested the HG algorithm with several pattern lengths. The verification of candidates was not carried out in this case, since we implemented the RKBT method only for $m = 8$, 16, and 32. If the verification would be done, the performance of the algorithm would worsen for those set sizes that produce spurious hits. Most of the candidates reported by the HG algorithm are

false matches because the probability of finding a real match is very low in our setting.

Figure 4b shows the results of these tests for the 2-gram version of the algorithm. With 50,000 patterns, the number of matches reported by the HG algorithm is roughly the same regardless of the pattern length. For $c = 256$ there are $2^{16} = 65{,}536$ different 2-grams. Thus, when there are more than 50,000 patterns, nearly all text positions will match. Figure 4b shows that when there are less than 10,000 patterns, HG is faster for longer patterns, because they allow longer shifts. When the number of false matches grows, the algorithm is faster for shorter patterns, because most positions match anyway and the overhead with shorter patterns is smaller.

5. MULTIPATTERN SHIFT–OR WITH *Q*-GRAMS

The shift–or algorithm [Baeza-Yates and Gonnet 1992] can be extended to a filtering algorithm for multiple patterns in a straightforward way. Rather than matching the text against exact patterns, the set of patterns is transformed to a single general pattern containing classes of characters. For example, if we have three patterns, "abcd," "pony," and "abnh," the characters {a, p} are accepted in the first position, characters {b, o} in the second position, characters {c, n} in the third position, and characters {d, h, y} in the fourth position. This approach is similar to extended string matching (see Navarro and Raffinot [2002]) where the pattern is a sequence of classes of characters.

The preprocessing phase now initializes the bit vectors for each character in the alphabet. The ith bit is set to 0 if any of the patterns contains the given character in the ith position. Otherwise the bit is set to 1. The filtering phase then proceeds exactly like the matching phase of the shift–or algorithm. Given this scheme, it is clear that all actual occurrences of the patterns in the text are candidates. However, there are also false candidates. In the previous example, "aocy" would also match. Therefore, each candidate must be verified using the RKBT method in the checking phase.

When the number of patterns grows, this approach is no longer adequate. As in the case of HG, the filtering capability of this approach can be considerably improved by using q-grams instead of single characters. The pattern is then a string of $m - q + 1$ q-gram classes exactly like in the HG algorithm. The bit vectors are initialized for each q-gram and the text is read one q-gram at a time. We call our modification SOG (short for shift–or with q-grams).

The improved efficiency of this approach is achieved at the cost of space. If the alphabet size is 256, storing the 2-gram bit vectors requires 2^{16} bytes for $m = 8$ while the single character vectors only take 2^8 bytes. We implemented SOG for 2- and 3-grams, as in the case of HG.

Baeza-Yates and Gonnet [1992] present a way to extend the Shift–Or algorithm for multiple patterns for small values of r. Patterns $P_1 = p_1^1 \cdots p_m^1, \ldots, P_r = p_1^r \cdots p_m^r$ are concatenated into a single pattern:

$$P = p_1^1 p_1^2 \ldots p_1^r p_2^1 p_2^2 \ldots p_2^r \ldots p_m^1 p_m^2 \ldots p_m^r$$

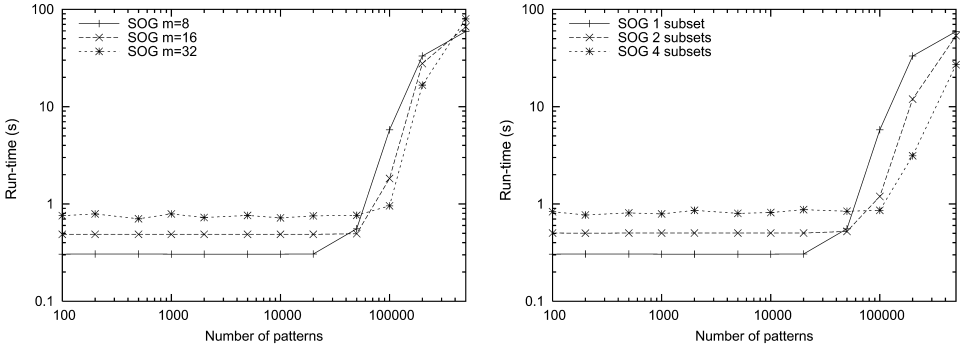


Fig. 5. The SOG algorithm. (a) The effect of pattern length. (b) The effect of one, two, and four subsets.

The patterns can then be searched in the same way as a single pattern, except that the shift of the state vector will be for r bits and a match is found, if any of the r bits corresponding to the highest positions is 0. This method can also be applied to SOG to make the algorithm faster for short patterns. The pattern set is divided into four or two subsets based on the first 2-gram. Each subset is then transformed into a general pattern, like in the plain SOG algorithm. The extension method of Baeza-Yates and Gonnet is then applied to these general patterns.

Fredriksson and Grabowski [2005] have proposed a modification to enhance the performance of the shift–or algorithm. In their scheme, several patterns are formed from the original one by taking every kth character starting at different offsets. For example, for $k = 2$, the pattern "pony" would produce patterns "pn" and "oy." Now we can scan the text reading every kth character and use the shift–or algorithm to find likely matches. These candidates can then be verified. We tried this modification for SOG but the shorter patterns produced more spurious hits and the scanning is a bit more complicated, so this modification did not make SOG faster.

5.1 Experiments with the SOG Algorithm

We tested the SOG algorithm with several pattern lengths and alphabet sizes. The 3-gram variation and the division of patterns to subsets were also tried.

The tests with pattern length were made for $m = 8$, 16, and 32 (see Figure 5a). The performance of the SOG algorithm degrades fast when the number of patterns reaches 100,000. This is the same effect that was found with the HG algorithm; Almost all text positions match because there are only 65,536 different 2-grams. When the pattern set size is less than 20,000, the run-time of the algorithm is constant, because no false matches are found.

Figure 5a also shows that the algorithm is slower for longer patterns. The structures of the SOG algorithm take 64 kB memory for $m = 8$, 128 kB for $m = 16$, and 256 kB for $m = 32$. The increased memory usage seems to slow down the algorithm.

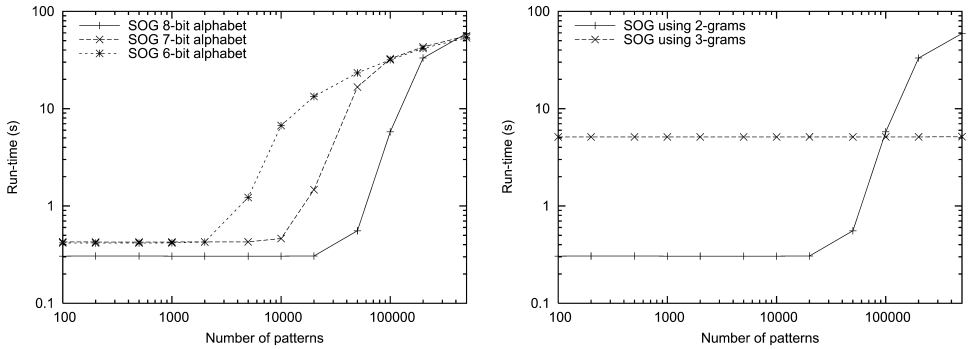


Fig. 6. The SOG algorithm: (a) the effect of alphabet size and (b) a comparison of 2- and 3-gram versions.

The use of subsets with the SOG algorithm was tested for $m = 8$. We tried versions with one, two, and four subsets (see Figure 5b). The versions using two and four subsets have a better filtering efficiency and thus their run-time remains longer constantly when the pattern set size is increased. However, they are again hindered by larger memory requirements. The basic version with one subset needs 64 kB of memory while the version using two subsets needs 128 kB of memory and the four subsets version 256 kB of memory.

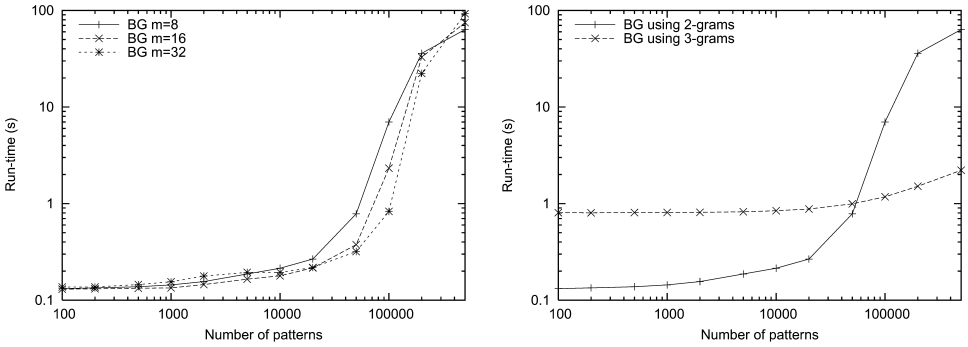
Given r patterns, using four subsets should result in roughly as many false matches as using one subset with $r/4$ patterns, because in the version with four subsets, only one subset can match at a given position. The results of the tests show that there are a little more matches than that. This is because of the more homogeneous sets produced by the division of patterns.

The behavior of SOG with alphabet sizes 64, 128, and 256 is shown in Figure 6a. Given the alphabet size 64, there are 4096 different 2-grams, and so the performance of the SOG algorithm was expected to degrade after 4000 patterns. Using the same reasoning, the performance of the SOG algorithm using the 7-bit alphabet was expected to degrade after 16,000 patterns and the 8-bit alphabet version after 65,000 patterns. The graphs of Figure 6a nicely follow this prediction.

The 3-gram version of the SOG algorithm was tested for $m = 8$. Figure 6b shows a comparison of the 2- and 3-gram versions. With less than 500,000 patterns, the run-time of the 3-gram SOG algorithm is constant and there are only a few false matches, because given our hashing scheme, there are about 2×10^6 different 3-grams. The 3-gram version is, however, much slower than the 2-gram version because of the hashing overhead and the greater memory requirement, which causes cache misses.

6. MULTIPATTERN BNDM WITH *Q*-GRAMS

Our third filtering algorithm is based on the BNDM algorithm by Navarro and Raffinot [2000]. This algorithm can be extended to multiple patterns in the same way as we did with the shift–or algorithm. We call this modification BG



(a) (b)

Fig. 7. The BG algorithm. (a) The effect of pattern length; (b) effect of 2- and 3-grams.

(short for BNDM with q-Grams). This approach corresponds to using the BNDM algorithm to search for a general pattern containing classes of characters. The bit vectors are initialized in the preprocessing phase so that the ith bit is 1 if the corresponding character appears in any of the reversed patterns in position i. In the filtering phase, the matching is then done with these bit vectors. As with HG and SOG, all match candidates reported by this algorithm must be verified. The checking phase of the algorithm uses the RKBT method.

Just like in SOG, 2- and 3-grams can be used to improve the efficiency of the filtering. That is, the pattern is transformed into a string of q-grams, the bit vectors are initialized for each q-gram rather than for a single character, and the text is read one q-gram at a time. The division to subsets, presented for the SOG algorithm, can also be used with the BG algorithm. This scheme works in the same way as with SOG algorithm, except that the subsets are formed based on the last 2-gram of the patterns.

6.1 Experiments with the BG Algorithm

In our experiments, the BG implementation based on the Simple BNDM algorithm presented in Peltola and Tarhio [2003] was fastest. The calculation of shifts is simplified in Simple BNDM. We tested the performance of the BG algorithm for $m = 8$, 16, and 32. Figure 7a shows the results of these tests. The algorithm is almost as fast in all these cases. The greater memory requirement slows the algorithm down with longer patterns, but, on the other hand, longer patterns allow for longer shifts. These two effects seem to balance out each other. The filtering efficiency is also somewhat better with longer patterns.

The 3-gram version of the BG algorithm was also tested and the results are shown in Figure 7b. The result are similar to that of SOG. With less than 50,000 patterns, the 2-gram approach is clearly faster, but, after that, the 3-gram version performs faster. The 3-gram version is slower mainly because of its memory usage. The hashing scheme used also slows it down.

The use of subsets with the BG algorithm was tested for $m = 8$ with one, two, and four subsets, the results are shown in Figure 8. The results of these tests are very similar to the ones of the SOG algorithm.

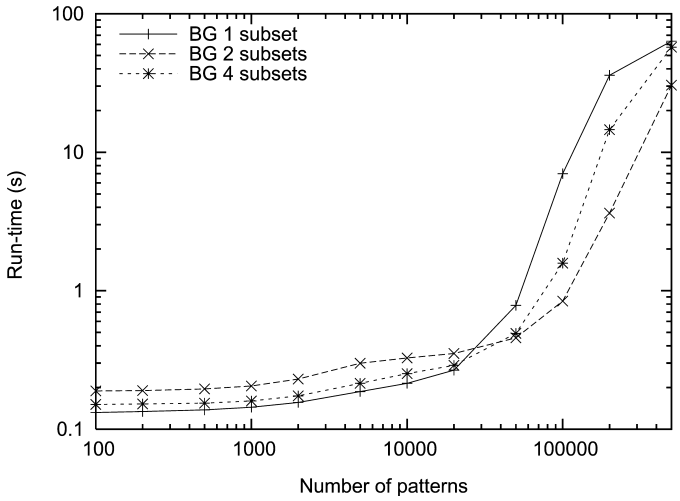


Fig. 8. The effect of subsets on the BG algorithm.

7. ANALYSIS

Let us consider the time complexities of the new algorithms HG, SOG, and BG. The algorithms can be divided in three phases: preprocessing, filtering, and checking. When considering the average case complexity, we assume the standard random string model, where each character of the text and the pattern is selected uniformly and independently.

All of our algorithms use the RKBT method for the checking phase. In the best case, no match candidates are found and then checking needs no time. In the worst case, there are $n' = n - m + 1$ candidates and all the patterns and text positions have the same hash value. In this case, we need to inspect the text pairwisely with each pattern and the worst-case time complexity is thus $O(n'rm) = O(nrm)$. If we assume that all patterns produce different hash values, the worst-case complexity is $O(n'(\log r + m)) = O(n(\log r + m))$, where $O(\log r)$ comes from the binary search and $O(m)$ from pairwise inspection.

The preprocessing phase of the filtering phases of the three algorithms is similar and it works in $O(rm)$. In addition, the initialization of the descriptor bit vectors needs $O(c^q)$. The preprocessing of the checking phase consists of calculating the hash values of the patterns and sorting the patterns according to these values. The sorting of the patterns takes $O(r \log r)$.[1]

Let us now consider the filtering phases of the algorithms. The worst-case complexity of the filtering phase of HG occurs when for each text position we scan the whole window. Thus the worst-case complexity for the filtering phase is $O(nm)$. In the average case, the probability of a text character not appearing in the jth 1-gram table, is

$$d_j = (1 - 1/c)^{rj}$$

[1]Our current implementation utilizes the Quicksort algorithm, which runs in $O(r^2)$ time, in the worst case, and in $O(r \log r)$ time, in the average case.

If the character is not found in the jth 1-gram table and this is the first character not found in the corresponding 1-gram table, a shift of length j will occur. Remember that the mth table is checked first, then the $m-1$th table, and so on. If all characters are found in their corresponding 1-gram tables, a match candidate has been found and after verifying it, a shift of length one will occur. Thus the expected length of shift is

$$S = 1\prod_{i=2}^{m}(1-d_i) + \sum_{j=2}^{m} j d_j \prod_{i=j+1}^{m}(1-d_i)$$

The first term handles the case when we have to check all the 1-gram tables yielding a shift of length one. Whether the character is found in the first table or not does not affect the length of the shift. It merely tells whether we have found a candidate or not. The rest of the formula handles the cases where the checking of the 1-gram tables has terminated at the jth table. The probability for this is $d_j \prod_{i=j+1}^{m}(1-d_i)$ and a shift of length j will occur. With suitable values of r, c, and m this is greater than $m/2$ and thus the filtering phase of HG is sublinear, on average, e.g., it inspects less than n text characters. Switching to q-grams guarantees the sublinearity for larger values of r and smaller values of c.

The probability of finding a candidate in the HG algorithm is the probability that each text character in the window is found in the corresponding 1-gram table. Thus the expected number of candidates is

$$C = n' \prod_{j=1}^{m}(1-d_j)$$

Again for suitable values of m, c, and r, the expected number of candidates is low enough so that the filtering phase dominates. Thus HG is sublinear on average. Again the use of q-grams guarantees the sublinearity for smaller values of c and larger values of r.

Let us next consider the time complexities of SOG and BG. We assume that $m \leq w$ holds, where w is the word length of the computer. Furthermore, we consider the time complexities of SOG and BG without division to subsets.

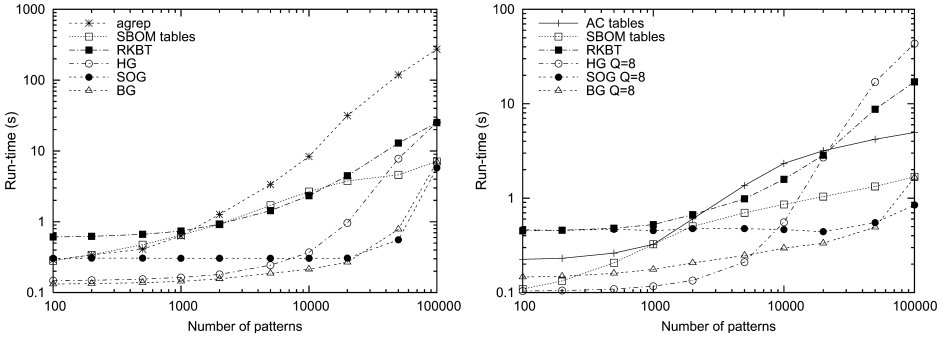
In SOG, the filtering phase is linear with respect to n. The probability that a text character matches a given pattern position in any of the patterns is $1-(1-1/c)^r$. Since the number of possible candidates is n' and there are m positions to match, the number of expected candidates is:

$$C_1 = n'(1-(1-1/c)^r)^m$$

This number can be reduced by utilizing q-grams. With q-grams, we estimate this expression by

$$C_q \leq h(1-(1-1/c^q)^r)^{\lfloor m/q \rfloor}$$

Note that this estimate considers only those q-grams which do not overlap. Thus the real number of candidates is lower. With suitable values of c, q, r, and m the expected number of candidates is low enough so that the filtering time dominates. Thus, SOG is linear, on average.



(a) (b)

Fig. 9. Run-time comparison of the algorithms for (a) random data ($m = 8$, $c = 256$) and (b) DNA data ($m = 32$).

Table I. Run-Times of the Algorithms When r Varies for $m = 8$ and $c = 256$[a]

	100	500	1,000	5,000	10,000	50,000	100,000
AC	0.628	1.453	2.159	3.965	6.132	17.436	30.639
Set Horspool	0.373	1.430	2.177	3.971	5.990	16.672	25.857
agrep	0.290	0.410	0.630	3.370	8.350	118.640	273.600
SBOM	0.280	0.475	0.648	1.710	2.675	4.613	7.086
RKBT	0.611	0.663	0.738	1.434	2.331	12.894	25.074
HG	0.148	0.155	0.164	0.244	0.369	7.812	25.339
SOG	0.305	0.306	0.305	0.305	0.305	0.556	5.773
BG	0.132	0.138	0.144	0.187	0.214	0.783	6.997

[a]AC, Set Horspool, and SBOM algorithms use the table implementation of the trie.

Let us then consider BG. The worst-case complexity of the basic BNDM is $O(nm)$. We did not want to apply any linear modification, because the checking phase of BG is not linear and the linear versions of BNDM are slower, in practice [Navarro and Raffinot 2000]. The average searching time of the BNDM algorithm is $O(n \log_{c'} m/m)$, where c' is the size of the alphabet for the original BNDM. In our approach we need to replace c' by $1/d$ where $d = 1-(1-1/c)^r$ is the probability that a single position of a generalized pattern matches. Clearly $\log_{1/d} m < m$ holds for suitable values of c, r, and m, and BG is then sublinear, on average. Switching to q-grams, $q \geq 2$, guarantees the sublinearity for smaller values of c and larger values of r.

The expected number of candidates for the BG algorithm is the same as for the SOG algorithm. Thus, for suitable parameter values the scanning phase dominates and the BG algorithm is sublinear, on average.

8. COMPARISON OF THE ALGORITHMS

A run-time comparison of the algorithms is shown in Figures 2 and 9a, based on Table I. These times include verification, but exclude preprocessing. The memory usage and the preprocessing times of the algorithms are shown in Table II. These are results from tests with patterns of eight characters, where HG, SOG, and BG use 2-grams. Recall that the size of the text is 32 MB.

Table II. Memory Usage and Preprocessing Times of the Algorithms for $r = 100$ and 100,000[a]

	Memory (kB)		Preprocessing (s)	
Algorithm	100	100,000	10,000	100,000
AC	702	565,600	1.44	48.37
Set Horspool	706	565,607	0.32	3.12
SBOM	719	571,702	0.62	20.87
RKBT	13	1,184	0.02	0.17
HG	69	1,240	0.02	0.21
SOG	77	1,248	0.02	0.18
BG	77	1,248	0.02	0.18

[a]AC, Set Horspool, and SBOM algorithms use the table implementation of the trie.

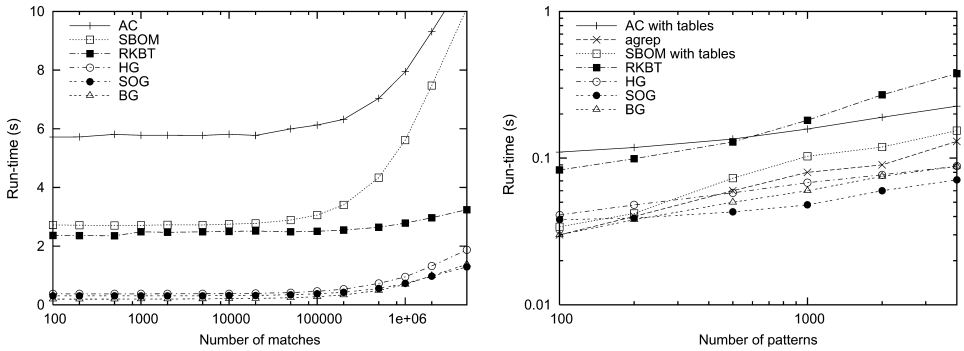
Figure 9a shows that our algorithms are considerably faster than the algorithms presented earlier. The HG and BG algorithms are the fastest, when there are less than 2000 patterns. Between 2000 and 20,000 patterns, the BG algorithm is the fastest and after that the SOG algorithm is the fastest. The BG algorithm has the best overall efficiency. With larger patterns sets, the use of subsets with these algorithms would be advantageous. Our algorithms scale to even larger pattern sets by using larger q-grams, if there is enough memory available.

Table II shows that the preprocessing phase of our algorithms is fast. Table II also shows that the memory usage of our algorithms is fairly small. In fact, the memory usage of our filtering techniques is constant for a fixed q. Because our algorithms use RKBT as a subroutine, their numbers also cover all the structures of RKBT including the second hash table. The space increase in Table II is because of the need to store the patterns for the verification phase. The space for the patterns could be reduced by using clever hash values. For example, for $m = 8$, we could store only four characters of each pattern and use a 32-bit hash value such that the other four characters can be obtained from these characters and the hash value.

We also run tests on DNA data. Our text was a chromosome from the fruit fly genome (20 MB). We used random patterns of 32 characters. We tried the values 4 through 10 of q in our filtering algorithms yielding the best results with 8-grams. The results using 8-grams are shown in Figure 9b. The algorithms HG and BG worked very well for sets of less than 10,000 patterns.

Our algorithms are filtering methods so they are not designed for searching texts that contain a lot of matches. Nevertheless, we also tested the algorithms in a setting where the text contained matches. Results of these tests show that our algorithms also perform surprisingly well in this setting (see Figure 10a).

To further test our algorithms with a text-containing matches, we ran several tests on English text getting somewhat controversial results. We used the King James version of the Bible as a text. First, we used patterns that were formed from at least 8 character long words from the text. Because our algorithms require the patterns to be of equal length, we used 8-character long prefixes of the words. There were 4216 distinct prefixes. Figure 10b shows the results of this experiment. The Set Horspool algorithm, which is not shown in the



(a) (b)

Fig. 10. (a) Run-times of the algorithms when using 10,000 patterns and a text containing a variable amount of matches. (b) Run-times of the algorithms when searching English words from the King James version of the Bible.

figure, was a bit slower than the SBOM algorithm. As the figure shows, agrep is the fastest of the earlier methods, which also holds true in the experiments of Navarro and Raffinot [2002]. The agrep tool skips a line when it has found an occurrence of a pattern, while all the other algorithms scan the whole line even if a match has already been found. This gives agrep a slight advantage in this experiment. Our algorithms, however, are faster with pattern sets containing more than 200 patterns.

In the other experiments with English text we used 8-character long strings randomly chosen from the text. In these tests, the traditional algorithms performed faster than our new ones. The good performance of our algorithms in the first test is probably because of the patterns not containing any space characters, which are very frequent in the text. This allows our algorithms to filter out most of the text positions.

9. CONCLUDING REMARKS

We have presented efficient solutions for multiple string matching using q-grams and bit-parallelism. Our algorithms work in three phases: preprocessing, filtering, and verifying. Our methods use smaller amounts of memory than previous methods and they scale well to very large pattern sets by using larger q-grams. We have demonstrated that our algorithms work well for random test data and DNA data. They are also applicable for searching English text, in some cases.

REFERENCES

ABRAHAMSON, K. 1987. Generalized string matching. *SIAM Journal on Computing 16*, 6, 1039–1051.

AHO, A. AND CORASICK, M. 1975. Efficient string matching: An aid to bibliographic search. *Commun. ACM 18*, 6, 333–340.

BAEZA-YATES, R. 1989. Improved string searching. *Softw. Pract. Exper. 19*, 3, 257–271.

BAEZA-YATES, R. AND GONNET, G. 1992. A new approach to text searching. *Commun. ACM 35*, 10, 74–82.

BOYER, R. AND MOORE, J. 1977. A fast string searching algorithm. *Commun. ACM 20*, 10, 762–772.
COMMENTZ-WALTER, B. 1979. A string matching algorithm fast on the average. In *Proceedings of the 6th International Colloquium on Automata, Languages and Programming*. Lecture Notes in Computer Science, vol. 71. Springer-Verlag, Berlin. 118–132.
CROCHEMORE, M. AND RYTTER, W. 1994. *Text Algorithms*. Oxford University Press, New York.
FISK, M. AND VARGHESE, G. 2001. Fast content-based packet handling for intrusion detection. Tech. Rep. CS2001-0670, University of California, San Diego, CA.
FREDRIKSSON, K. 2002. Faster string matching with super-alphabet. In *Proceedings of 9th Symposium on String Processing and Information Retrieval (SPIRE'02)*. Lecture Notes in Computer Science, vol. 2476. Springer-Verlag, Berlin. 44–57.
FREDRIKSSON, K. AND GRABOWSKI, S. 2005. Practical and optimal string matching. In *Proceedings of 12th Symposium on String Processing and Information Retrieval (SPIRE'05)*. Lecture Notes in Computer Science, vol. 3772. Springer-Verlag, Berlin.
GUM, B. AND LIPTON, R. 2001. Cheaper by the dozen: Batched algorithms. In *Proceedings of the 1st SIAM International Conference on Data Mining*.
HORSPOOL, N. 1980. Practical fast searching in strings. *Softw. Pract. Exper. 10*, 501–506.
KARP, R. AND RABIN, M. 1987. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development 31*, 249–260.
KIM, S. AND KIM, Y. 1999. A fast multiple string-pattern matching algorithm. In *Proceedings of 17th AoM/IAoM Conference on Computer Science*.
MARKATOS, E. P., ANTONATOS, S., POLYCHRONAKIS, M., AND ANAGNOSTAKIS, K. G. 2002. Exclusion-based signature matching for intrusion detection. In *Proceedings of the IASTED International Conference on Communications and Computer Networks (CCN)*. ACTA Press. Calgary, AB, Canada, 146–152.
MUTH, R. AND MANBER, U. 1996. Approximate multiple strings search. In *Proceedings of 7th Combinatorial Pattern Matching (CPM'96)*. Lecture Notes in Computer Science, vol. 1075. Springer-Verlag, Berlin. 75–86.
NAVARRO, G. AND RAFFINOT, M. 2000. Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM Journal of Experimental Algorithmics (JEA) 5*, 4, 1–36.
NAVARRO, G. AND RAFFINOT, M. 2002. *Flexible pattern matching in strings*. Cambridge University Press, Cambrige, UK.
PELTOLA, H. AND TARHIO, J. 2003. Alternative algorithms for bit-parallel string matching. In *Proceedings of 10th Symposium on String Processing and Information Retrieval (SPIRE'03)*. Lecture Notes in Computer Science, vol. 2857. Springer-Verlag, Berlin. 80–94.
PING, L., JIAN-LONG, T., AND YAN-BING, L. 2005. A partition-based efficient algorithm for large scale multiple-strings matching. In *Proceedings of 12th Symposium on String Processing and Information Retrieval (SPIRE'05)*. Lecture Notes in Computer Science, vol. 3772. Springer-Verlag, Berlin.
TUCK, N., SHERWOOD, T., CALDER, B., AND VARGHESE, G. 2004. Deterministic memory-efficient string matching algorithms for intrusion detection. In *Proceedings of the IEEE Infocom Conference*.
WU, S. AND MANBER, U. 1992a. Agrep—a fast approximate pattern-matching tool. In *Proceedings of the Usenix Winter 1992 Technical Conference*. 153–162.
WU, S. AND MANBER, U. 1992b. Fast text searching allowing errors. *Commun. ACM 35*, 10, 83–91.
WU, S. AND MANBER, U. 1994. A fast algorithm for multi-pattern searching. Tech. Rep. TR-94-17, Department of Computer Science. University of Arizona, Tucson, Arizona.
ZHU, R. AND TAKAOKA, T. 1989. A technique for two-dimensional pattern matching. *Commun. ACM 32*, 1110–1120.

Received September 2005; revised February 2006; accepted February 2006