Tuning String Matching for Huge Pattern Sets

Jari Kytöjoki, Leena Salmela, and Jorma Tarhio[⋆]

Department of Computer Science and Engineering
Helsinki University of Technology
P.O. Box 5400, FIN-02015 HUT, Finland

Abstract 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. Our algorithms showed to be substantially faster than earlier solutions for sets of 1,000–100,000 patterns. The gain is due to the improved filtering efficiency caused by q-grams.

1 Introduction

We consider exact string matching of multiple patterns. Many good solutions have been presented for this problem, e.g. Aho-Corasick [1], Commentz-Walter [5,14], and Rabin-Karp [11,12] with their variations. However, most of the earlier algorithms have been designed for pattern sets of moderate size, i.e. a few dozens, and they do not unfortunately 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. in a handheld device). Such algorithms are needed in intrusion detection [8], in content scanning, and in specific data mining problems [9]. The focus is 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 mostly for c=256. All exact occurrences of the patterns should be reported.

As our main contribution we will present three algorithms HG, SOG, and BG based on the Boyer-Moore-Horspool [10], shift-or [3], and BNDM [13] algorithms, respectively. Our algorithms 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

[⋆] Corresponding author: jorma.tarhio@hut.fi.

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 [7].

It is well known (see e.g. [2,4]) 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 [15]. We use q-grams in a different way in order to improve filtration efficiency by changing the alphabet.

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 rareness of matches in our setting. Our algorithms showed to be very fast in practice. For example, HG is 15 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 enough in the specified case. The filtering efficiency of our algorithms will continue beyond 100,000 patterns if more memory is used.

2 Earlier Solutions

The classical Aho-Corasick algorithm [1] has been widely used for multiple pattern matching. Although it works rather well for small pattern sets, it is not suitable for huge pattern sets because of intolerable memory requirements. And the algorithm gets slower when the number of patterns increases.

2.1 Rabin-Karp Approach

A well-known solution [9,12,17] to cope with large pattern sets with less memory is to combine the Rabin-Karp algorithm [11] 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 [12] 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 i'th 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, and 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.

2.2 Set Horspool

The Commentz-Walter algorithm [5] for multiple patterns has been derived from the Boyer-Moore algorithm [4]. A simpler variant of this algorithm is called set Horspool [14]. (The same algorithm is called set-wise Boyer-Moore in [8].) This algorithm is based on the Boyer-Moore-Horspool algorithm [10] for single patterns. In the Boyer-Moore-Horspool algorithm, the bad character function $B(a)$ is defined as the distance from the end of the pattern $p_1p_2\cdots p_m$ to the last occurrence of the character a: $B(a) = min\{h \mid p_{m-h} = a\}$. This function can be generalized for multiple patterns. The bad character function for the set of patterns 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. Then the bad character function is applied to the endpoint character and the pattern trie is shifted accordingly.

The Wu-Manber algorithm [15] is a variation of the set Horspool algorithm. It uses a hash table of the last q-grams of patterns. The famous agrep tool [16] includes an implementation of the Wu-Manber algorithm.

3 Multi-Pattern Horspool with q-Grams

The Boyer-Moore-Horspool algorithm [10] can be applied to multiple patterns also in another way. We call 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 as follows. The first table keeps track of characters which appear in the first position in any pattern, the second table keeps track of characters which appear in the first or second position in any pattern and so on. Figure 1a shows the six tables corresponding to the pattern 'qwerty'.

These tables can then be used for pattern matching as follows. First the m'th character is compared with the m'th table. If the character does not appear in this table, the character cannot appear in positions $1 \ldots m$ in any pattern and

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)

Figure 1. The HG algorithm: (a) the data structures for the pattern ‘qwerty’ and (b) the pseudo-code for m=6.

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. The pseudo-code for m=6 is shown in Figure 1b. Given this procedure, it is clear that all matches are found. However, also false matches can occur. E.g. ‘qqqqqq’ is a false candidate in our example. The candidates are verified by using the RKBT method described in Section 2.1.

As the number of patterns grows, the filtering efficiency of the above scheme decreases until almost all the text positions are candidates because there 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 a 3-gram is used to address the tables, each character is hashed to a 7-bit value. This diminishes the number of different 3-grams from 2^{24} to 2^{21}.

4 Multi-Pattern Shift-Or with q-Grams

The shift-or algorithm [3] 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 has been used for extended string matching (see e.g. [14]). 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 our example 'aocy' would also match. Therefore, each candidate must be verified.

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. Then the pattern is a string of $m - q + 1$ q-gram classes. We call our modification SOG (short for Shift-Or with q-Grams). Again, the RKBT method is used for verification.

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-grams and 3-grams as in the case of HG.

Baeza-Yates and Gonnet [3] 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:

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

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 in a different way to make the SOG 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 treated like a single pattern in the extension method of Baeza-Yates and Gonnet.

5 Multi-Pattern BNDM with q-Grams

Our third filtering algorithm is based on the backward nondeterministic DAWG matching (BNDM) algorithm by Navarro and Raffinot [13]. The BNDM algorithm itself has been developed from the backward DAWG matching (BDM) algorithm [6].

In the BDM algorithm [6], 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 for the longest prefix of the pattern from right to left with the DAWG. 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.

The BNDM algorithm [13] 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. The i'th bit is 1 in this vector if x appears in the reversed pattern in position i. Otherwise the i'th 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. If the m'th bit is 1 after this update operation, we have found a prefix starting at position j. If j is the first position in the window, a match has been found.

The BNDM algorithm can be extended to multiple patterns in the same way as we did with the shift-or algorithm. We call this modification BG (short for Bndm with q-Grams). The matching is done with a general pattern containing classes of characters. The bit vectors are initialized so that the i'th bit is 1 if the corresponding character appears in any of the reversed patterns in position i. As with HG and SOG, all match candidates reported by this algorithm must be verified. Just like in SOG, 2- and 3-grams can be used to improve the efficiency of the filtering. Also the division to subsets, presented for the SOG algorithm, can 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 Analysis

Let us consider the time complexities of the SOG and BG algorithms without division to subsets. The algorithms can be divided in three phases: preprocessing, scanning, and checking. Let us assume that $m \leq w$ holds, where w is the word length of the computer. 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.

In the best case, no match candidates are found and then checking needs no time. In the worst case there are $h = n - m + 1$ candidates, and then the checking time $O(nh \log r) = O(nm \log r)$ dominates. Here $O(\log r)$ comes from binary search and $O(m)$ from pairwise inspection.

The preprocessing phase of the both algorithms is similar and it works in $O(rm)$. In addition, the initialization of the descriptor bit vectors needs $O(c^q)$.

In SOG the scanning phase is linear. The expected number of candidates C_1 depends on r, c, and m:

$$C_1 = h \cdot (1 - (1 - 1/c)^r)^m.$$

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

$$C_q = h \cdot (1 - (1 - 1/c^q)^r)^{m-q+1}.$$

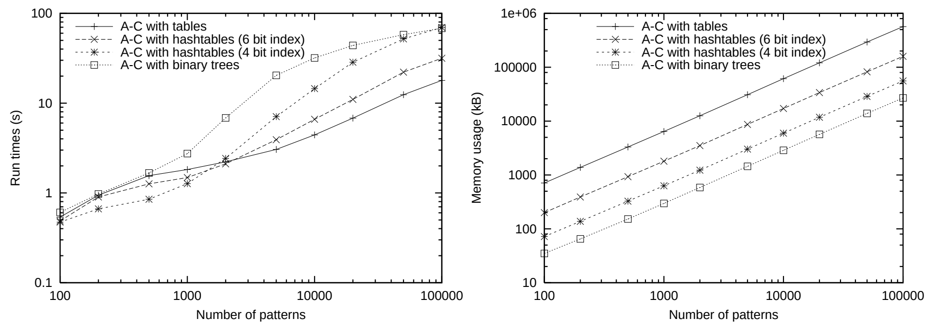

Figure 2. Performance of different trie implementations of the Aho-Corasick algorithm.

Note that even C_2 is not accurate, because consecutive overlapping 2-grams are not independent. However, the difference from the exact value is insignificant for huge sets of patterns.

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 [13]. 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 the average, i.e. it does not inspect every text character. Switching to q-grams, $q \geq 2$, guarantees the sublinearity for smaller values of c and larger values of r.

7 Experiments with Earlier Algorithms

We ran tests on several algorithms. We used a 32 MB randomly created text in the alphabet of 256 characters. Also the patterns were randomly generated in the same alphabet. Note that in our case random data is in a sense harder than real data. For example, 'zg' is rarer in an English text than in a random text.

If not otherwise stated, m=8 and c=256 hold. 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 1.8 GHz Pentium 4 processor, 1 GB of memory, and 256 kB on-chip cache. The computer was running Linux 2.4.18. The algorithms were written in C and compiled with the gcc compiler.

Aho-Corasick. We used a code based on the case-sensitive implementation by Fisk and Varghese [8] to test the Aho-Corasick algorithm [1]. We tested three alternative implementations of the goto-function: table, hash table, and binary tree. The hash table version was tested with table sizes 16 and 64 (resulting in 4- and 6-bit indexes), see Figure 2.

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 2 shows a steady increase. Given the memory graph of Figure 2, the hierarchical memory could explain this behavior. For pattern set sizes between 100 and 2,000, the hash table version of the goto-function is preferable. When there are more than 2,000 patterns, the table version is the fastest but its memory requirement does not make it very attractive.

RKBT. 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.

Set Horspool. We used the code of Fisk and Varghese [8] to test the set Horspool algorithm. The same variations as for the Aho-Corasick algorithm were made. The results on memory usage were similar to those of the Aho-Corasick algorithm because the trie structure is very similar. Also the test results on run times resemble those of the Aho-Corasick algorithm especially with very large pattern sets. This is probably due to the memory usage. Differences with less than 1,000 patterns were not significant between modifications.

Agrep. We also tested the agrep tool [16]. 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. The run times measured do not contain the time used to preprocess the patterns.

In the experiments of Navarro and Raffinot [14] agrep was the fastest algorithm for 1,000 patterns for m=8. This holds true also for our experiments (excluding the new algorithms). The agrep tool is the fastest up to 2,000 patterns, the RKBT method is the fastest between 2,000 and 20,000 patterns and the set Horspool algorithm is the fastest with more than 20,000 patterns although its memory usage is excessive.

Figure 3 shows a comparison of the four earlier algorithms mentioned above. The times include verification but exclude preprocessing.

8 Experiments with New Algorithms

The test setting is the same as in the previous section. The new algorithms are all filtering algorithms, which use the RKBT method for verification of candidates. Each run time contains the verification time (if not otherwise specified) and excludes the preprocessing time.

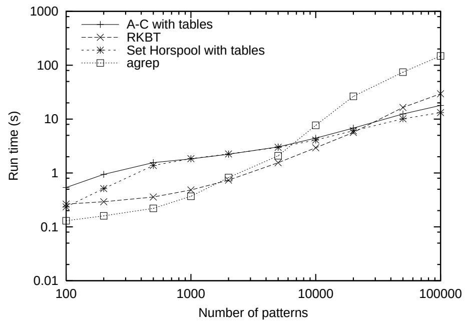

Figure 3. Run time comparison of the earlier algorithms.

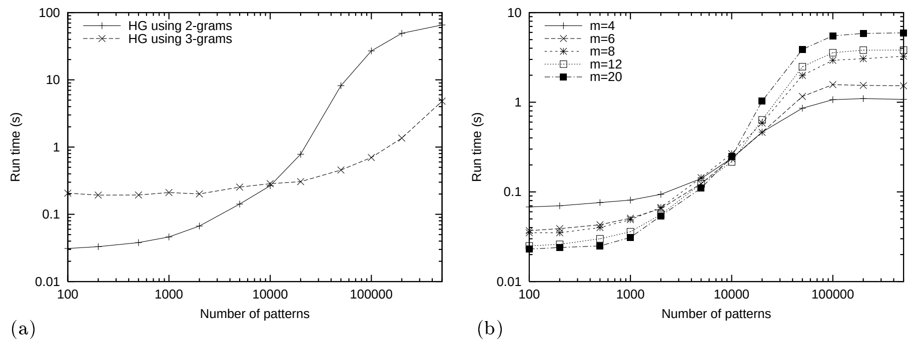

Figure 4. The HG algorithm: (a) comparison of 2-gram and 3-gram versions and (b) run times of the 2-gram version for different pattern lenghts.

8.1 HG

The HG algorithm was tested both with the 2-gram 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 due to 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 tested the HG algorithm also 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.

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. So, when there are more than 50,000 pat-

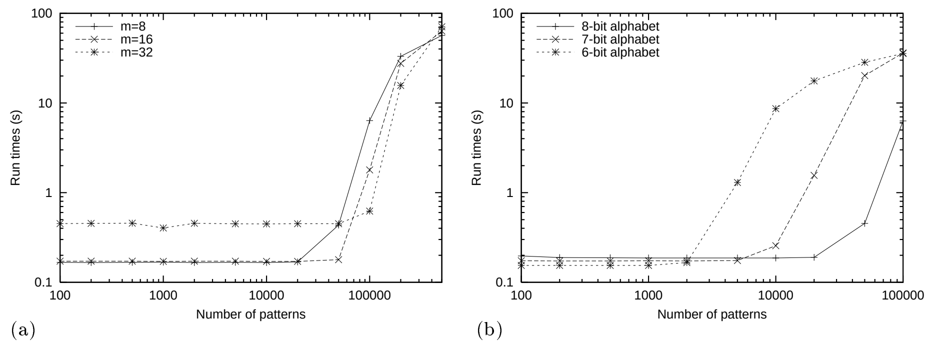

Figure 5. The SOG algorithm: (a) the effect of pattern length and (b) the effect of alphabet size.

terns, 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.

8.2 SOG

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 significantly slower for m=32 than for m=8 and 16. This is likely due to the data cache. The structures of the SOG algorithm take 64 kB memory for m=8, 128 kB for m=16, and 256 kB for m=32. Given the cache size of 256 kB, it is clear that the structures for m=32 cannot be held in the cache all the time because also the text to be searched has to be there.

The behavior of SOG with alphabet sizes 64, 128, and 256 is shown in Figure 5b. Given the alphabet size 64, there are 4,096 different 2-grams, and so the performance of the SOG algorithm was expected to degrade after 4,000 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 5b follow nicely this prediction.

The 3-gram version of the SOG algorithm was tested for m=8. Figure 6a shows a comparison of the 2-gram and 3-gram versions. With less than 500,000

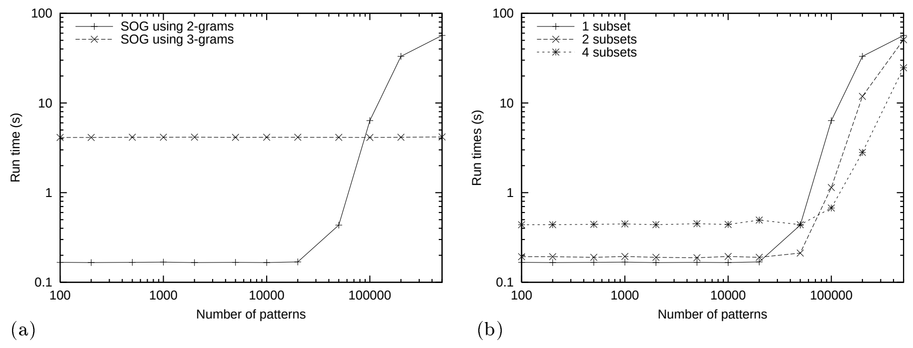

Figure 6. The SOG algorithm: (a) comparison of the 2- and 3-gram versions and (b) the effect of one, two and four subsets.

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 $2 \cdot 10^6$ different 3-grams. The 3-gram version is, however, much slower than the 2-gram version due to the hashing overhead and the greater memory requirement which causes cache misses.

The use of subsets with the SOG algorithm was tested for m=8. We tried versions with one, two and four subsets, see Figure 6b. The versions using one or two subsets are almost as fast up to 20,000 patterns. After that the version using two subsets is slightly faster. The version using four subsets is significantly slower than the other two versions with small pattern set sizes. The problem here is that the table needed to store the 32-bit vectors is as large as the data cache. In computers with larger caches this version would likely perform as well as the other two. 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 due to the more homogeneous sets produced by the division of patterns.

8.3 BG

We tested the performance of the BG algorithm for m = 8, 16 and 32. The algorithm is faster for m=16 than for m=8. In the case of m=32, the algorithm suffers from the large table which cannot be kept in the cache all the time. However, the filtering efficiency improves slightly with longer patterns.

The 3-gram version of the BG algorithm was also tested. The result was 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.

Table 1. Run times of the algorithms when r varies for m=8 and c=256.

	100	200	500	1,000	2,000	5,000	10,000	20,000	50,000	100,000
Aho-Corasick	0.538	0.944	1.559	1.824	2.221	3.055	4.433	6.804	12.427	17.951
RKBT	0.265	0.293	0.358	0.483	0.735	1.551	2.942	5.660	16.423	29.567
Set Horspool	0.235	0.513	1.375	1.848	2.252	2.990	4.083	6.068	10.154	13.225
agrep	0.130	0.160	0.220	0.370	0.820	2.090	7.670	26.480	74.370	148.690
SOG	0.167	0.166	0.167	0.168	0.166	0.167	0.166	0.169	0.435	6.357
HG	0.031	0.033	0.038	0.046	0.067	0.142	0.266	0.784	8.182	26.884
BG	0.027	0.029	0.035	0.041	0.056	0.106	0.151	0.206	0.649	7.389

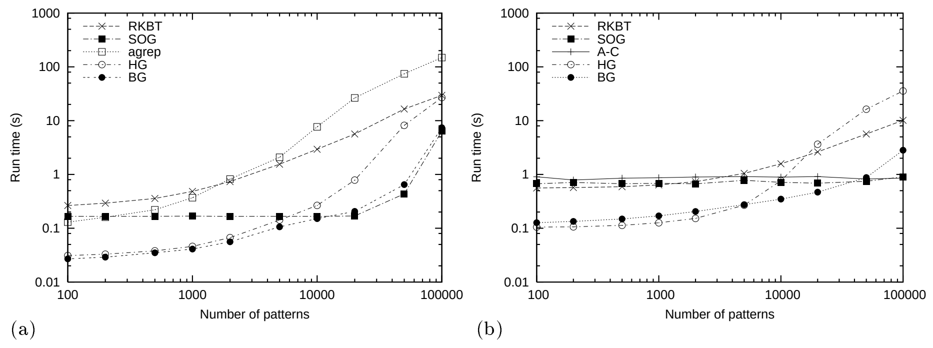

Figure 7. Run time comparison of the algorithms for (a) random data (m=8, c=256) and (b) dna data (m=32).

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

8.4 Comparison of the Algorithms

A run-time comparison of the algorithms is shown in Figures 3 and 7a based on Table 1. These times include verification but exclude preprocessing.

The memory usage and the preprocessing times of the algorithms are shown in Table 2. These are results from tests with patterns of eight characters, where HG, SOG, and BG use 2-grams.

Figure 7a 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 5,000 patterns. Between 5,000 and 100,000 patterns the SOG and BG algorithms are 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 2. Memory usage and preprocessing times of the algorithms for $r = 100$ and 100,000.

Algorithm	*Memory (kB)*		*Preproc. (s)*	
	100	100,000	100	100,000
RKBT	13	1,184	0.02	0.20
HG	69	1,240	0.03	0.23
SOG	77	1,248	0.03	0.21
BG	77	1,248	0.03	0.21
A-C (with tables)	799	663,174	0.54	5.10
Set Horspool (with tables)	793	656,338	0.19	1.68

Table 2 shows that the preprocessing phase of our algorithms is fast. Table 2 also shows that the memory usage of our algorithms is fairly small. In fact, the memory usage of our filtering techniques is constant. Because our algorithms use RKBT as a subroutine, their numbers cover also all the structures of RKBT including the second hash table. The space increase in Table 2 is due to 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 a preliminary test on dna data. Our text was the genome of fruit fly (20 MB). We used random patterns of 32 characters for q=8. The results are shown in Figure 7b. This test was made on a 1.0 GHz computer with 256 MB of memory and 512 kB cache. The algorithms HG and BG worked very well for sets of less than 10,000 patterns.

9 Concluding Remarks

We have presented efficient solutions for multiple string matching based on filtering with q-grams and bit-parallelism. We showed that on random test data, our algorithms perform faster and use a smaller amount of memory than the earlier ones. The preprocessing phase of our algorithms is fast. We tuned the algorithms to handle efficiently up to 100,000 patterns of eight characters. Our algorithms suit especially well to the searching of huge static sets of rare patterns.

Our approach seems to be sensitive to cache effects. We need to test the algorithms in several computers of different types in order to get additional information on their behavior.

We utilized overlapping q-grams. We tested our algorithms also with consecutive non-overlapping q-grams, but this modification brought clearly worse results. We used mainly the alphabet c=256. In the near future we will try small alphabets and compare our algorithm with the SBOM algorithm of Navarro and Raffinot [14]. We will also consider approximate matching (see e.g. [12]).

References

1. A. Aho, M. Corasick: Efficient string matching: An aid to bibliographic search. *Communications of the ACM* 18, 6 (1975), 333–340.
2. R. Baeza-Yates. Improved string searching. *Software – Practice and Experience*, 19, 3 (1989), 257–271.
3. R. Baeza-Yates, G. Gonnet: A new approach to text searching. *Communications of ACM* 35, 10 (1992), 74–82.
4. R. Boyer, S. Moore: A fast string searching algorithm. *Communications of the ACM* 20 (1977), 762–772.
5. B. Commentz-Walter: A string matching algorithm fast on the average. *Proc. 6th International Colloquium on Automata, Languages and Programming, Lecture Notes on Computer Science* 71, 1979, 118–132.
6. M. Crochemore, W. Rytter: *Text algorithms.* Oxford University Press, 1994.
7. K. Fredriksson: Fast string matching with super-alphabet. *Proc. SPIRE '02, String Processing and Information Retrieval, Lecture Notes in Computer Science* 2476, 2002, 44–57.
8. M. Fisk, G. Varghese: Fast content-based packet handling for intrusion detection. UCSD Technical Report CS2001-0670, 2001.
9. B. Gum, R. Lipton: Cheaper by the dozen: batched algorithms. *Proc. First SIAM International Conference on Data Mining*, 2001
10. N. Horspool: Practical fast searching in strings. *Software – Practice and Experience* 10 (1980), 501–506.
11. R. Karp, M. Rabin: Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development* 31 (1987), 249–260.
12. R. Muth, U. Manber: Approximate multiple string search. *Proc. CPM '96, Combinatorial Pattern Matching, Lecture Notes in Computer Science* 1075, 1996, 75–86.
13. G. Navarro, M. Raffinot: Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM Journal of Experimental Algorithms* 5, 4 (2000), 1–36.
14. G. Navarro, M. Raffinot: *Flexible pattern matching in strings.* Cambridge University Press, 2002.
15. S. Wu, U. Manber: A fast algorithm for multi-pattern searching. Report TR-94-17, Department of Computer Science, University of Arizona, 1994.
16. S. Wu, U. Manber: Agrep – A fast approximate pattern-matching tool. *Proc. Usenix Winter 1992 Technical Conference*, 1992, 153–162.
17. R. Zhu, T. Takaoka: A technique for two-dimensional pattern matching. *Communications of the ACM* 32 (1989), 1110–1120.