SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 27(7), 851–861 (JULY 1997)

String Matching in the DNA Alphabet

JORMA TARHIO

Department of Computer Science, University of Joensuu, P.O. Box 111, FIN-80101 Joensuu, Finland (email: tarhio@cs.joensuu.fi, tarhio@cs.helsinki.fi)

AND

HANNU PELTOLA

Department of Computer Science, P.O. Box 26 (Teollisuuskatu 23), FIN-00014 University of Helsinki, Finland (email: hannu.peltola@helsinki.fi, hannu.peltola@iki.fi)

SUMMARY

Searching for long DNA strings is studied. A q-gram variation of the Boyer–Moore algorithm is considered. An alphabet transformation with precomputed tables is utilized to reduce the processing time. Experimental results show that the new algorithm is efficient in practice. ©1997 by John Wiley & Sons, Ltd.

KEY WORDS: string matching; Boyer–Moore algorithm

INTRODUCTION

Searching for occurrences of string patterns is a common problem in many applications. Various good solutions have been presented for string matching. The most efficient solutions in practice are based on the Boyer–Moore algorithm.[1]

A typical question in molecular biology is whether a given sequence has appeared elsewhere. In the following, we will concentrate on searching for exact occurrences of long patterns in the DNA alphabet which in a typical case contains four characters, namely `a`, `c`, `g`, and `t`. However, the biologists are often interested in finding similar sequences. Nevertheless, exact searching can be used as a fast subroutine of approximate searching. At low error levels any algorithm for exact searching can be used as a fast filtering method. Assume that we allow e errors. If we divide the pattern in $e+1$ distinct blocks, every approximate occurrence contains an exact occurrence of at least one of the blocks.[2] Thus an occurrence of any block defines a potential approximate occurrence of the pattern, which can be checked with a slower dynamic programming method.

Hume and Sunday[3] review several techniques how to improve the practical efficiency of the Boyer–Moore algorithm using different shift heuristics, tight loops, unrolling of loops, and some other approaches. Their study mainly deals with searching for words of an English text, but they also tested their algorithms on DNA strings. Later Kim and Shawe-Taylor[4] present more efficient solutions for DNA strings based on an implementation of an algorithm introduced by Baeza-Yates.[5]

We will introduce a new version of the Baeza-Yates algorithm,[4,5] which is a modification of the Boyer–Moore–Horspool algorithm[6] for small alphabets. In the Baeza-Yates algorithm the

CCC 0038–0644/97/070851–11 $17·50

Received 14 November 1995
Revised 1 July and 24 October 1996

shift is based on q-grams which are strings of q consecutive characters. We will present a new way to transform such a substring to an integer. Experimental results show that our approach is more efficient than the previous algorithms for long patterns.

BOYER–MOORE–HORSPOOL ALGORITHM

The characteristic feature of the Boyer–Moore algorithm[1] is the right-to-left scan over the pattern. Let us think a pattern sliding above a text. At each alignment of the pattern with the text, characters of the text below the pattern are examined from right to left, starting by comparing the rightmost character of the pattern with the character in the text currently below it. Between alignments, the pattern is shifted to the right along the text.

After each mismatch, the original Boyer–Moore algorithm chooses the larger shift given by two heuristics. In the *match* heuristic, the pattern has to match all the text characters that were found to match at the previous alignment. In the *occurrence* heuristic, the character in the text that caused a mismatch will be aligned with the rightmost occurrence of that character in the pattern.

If the alphabet is large enough, the match heuristic is not very useful in practice, because the patterns are seldom periodic. So it is wise to use only the occurrence heuristic. Because it is not necessary to base the shift of the pattern on the text character that caused the mismatch, one can use the text character under the rightmost character of the pattern. This modification was presented by Horspool.[6] In practice the Boyer–Moore–Horspool algorithm is clearly faster than the original Boyer–Moore algorithm,[1] e.g. for English texts.

Although the Boyer–Moore–Horspool algorithm is slower for long patterns than the original Boyer–Moore algorithm in the case of a small alphabet[7] and though a related variation by Sunday[8] is faster for short patterns,[7] the Boyer–Moore–Horspool algorithm clearly provides the best platform to develop more efficient versions. Algorithm 1 is a simple form of the Boyer–Moore–Horspool algorithm. The text is denoted by `T[1]... T[n]` and the pattern by `P[1]... P[`m`]`.

Algorithm 1

```
1  Preprocess D;
2  p:=P[m];
3  k:=m;
4  repeat
5     t:=T[k];
6     if t=p then
7        Check other positions;
8     k:=k+D[t];
9  until k > n;
```

On line 7 characters `P[1]... P[`$m-1$`]` are compared with the corresponding text characters until a mismatch is found or a match is completed. Horspool's shift heuristic makes it possible to do this in any order. The most common orders are a forward scan (left-to-right) and a reverse scan (right-to-left). It depends both on the string matching problem and mostly on the computing environment, which order is advantageous, although the difference is often insignificant. We apply only the forward scan in our algorithms.

The computation of the shift table D is based on the following definition:

$$D[x] = \min(m, \min\{m - d \mid 1 \leq d < m, P[d] = x\}).$$

FINGERPRINT METHOD

A q-gram (or a q-tuple) is a substring of q characters. In a way, a q-gram represents a character of a larger alphabet. It is widely known that the Boyer–Moore algorithm is able to take longer shifts by examining a q-gram at a time instead of a single text character.[1,4,5,9,10] The Baeza-Yates algorithm[4,5] is one of the applications of this idea. Kim and Shawe-Taylor[4] present an efficient implementation of that algorithm for DNA strings. We will consider variations of the Baeza-Yates algorithm and suggest implementation techniques, which improve the total speed of the algorithm in the case of the DNA alphabet.

From a q-gram $a_0 \ldots a_{q-1}$ we compute a *fingerprint* $t = \sum_{i=0}^{q-1} c^i \cdot a_i$, which is a (reversed) number of base c, where c is the size of the alphabet.

At each alignment of the pattern, the last q-gram of the pattern is compared with the corresponding q-gram in the text by testing the equality of their fingerprints. If the fingerprints match, we have found a *potential occurrence*, which has to be checked. Let us call the fingerprint of the last q-gram of the pattern the fingerprint of the pattern. In Algorithm 1, which works with unigrams, p is the fingerprint of the pattern and t is a fingerprint computed from the text. This algorithm can be easily modified to Algorithm 2 which handles larger values of q. The notation $f(T, k, q)$ refers to the fingerprint of the q-gram $T[k - q + 1], \ldots, T[k]$ and $f(P, m, q)$ refers to the fingerprint of the pattern. Here fingerprints are computed using Horner's rule: $t = a_0 + c(a_1 + c(a_2 + \cdots c(a_{q-2} + ca_{q-1}) \cdots))$.

Note that a straightforward implementation of Algorithm 2 (or the Baeza-Yates algorithm) cannot process patterns shorter than q, but this problem is easy to avoid by incorporating Algorithm 1 to Algorithm 2 for short patterns.

Algorithm 2

```
1  Preprocess D;
2  p:=f(P,m,q);
3  k:=m;
4  repeat
5     t:=f(T,k,q);
6     if t=p then
7        Check other positions;
8     k:=k+D[t];
9  until k > n;
```

At each alignment, fingerprints *t* and *p* are compared. If they are equal, text positions *T* [*k-m+1*], . . ., *T* [$k - q$], corresponding to the $m - k$ characters long prefix of the pattern, are examined until a mismatch is found or a match is completed. Note that only the fingerprint of the pattern, not the pattern itself, is needed in the loop on lines 4–9 , which saves time.[4] In the end of the loop, a shift is taken according to the precomputed shift table D indexed by fingerprints. The computation of D is based on the definition:

$$D[x] = \min(m, \min\{m-d \mid 1 \le d < m, P[d-e+1] \ldots P[d] = \text{suffix}(x, e), e = \min(q, d)\}),$$

where $\text{suffix}(x, e)$ is the string of the last e characters of x. Baeza-Yates[5] as well as Kim and Shawe-Taylor[4] use a simpler alternative for the shift table D, which is faster to compute but which leads slightly shorter shifts:

$$D'[x] = \min(m - q + 1, \min\{m - d \mid q \le d < m, P[d - q + 1] \ldots P[d] = x\}).$$

We mostly use D in our algorithms.

At the implementation level, we use fixed code instead of a loop for computing fingerprints, because a loop would make the algorithm slower. Kim and Shawe-Taylor[4] use a loop where q has been defined as a constant, but the efficiency of this solution depends on the optimization skills of the compiler.

ALPHABET TRANSFORMATION

The straightforward implementation of Algorithm 2 contains a space problem, because the shift table D is indexed by the fingerprints. If the alphabet contains characters `a`, `c`, `g`, and `t`, whose ASCII codes are 97, 99, 103, and 116, respectively, the size of the actual alphabet is not four but 117 or 256. In order to handle case $q = 4$, we might need a shift table of at least 117^4 elements. A large shift table also makes the preprocessing phase of the algorithm slow, because the table must be initialized. In practice, the initialization time of the table D dominates the total running time for large values of q.

An alternative for multiplication in the computation of a fingerprint is to round the alphabet size c upwards to a power of two and to apply bitwise shifting.[5] Bitwise shifting is faster than multiplication in most computers, and the gain depends on the computer architecture. If c is not originally a power of two, this approach requires a larger table D.

A third alternative is to replace multiplication by a table look-up, e.g. for $q = 4$ we have

$$a_0 + a_1 * c + a_2 * c^2 + a_3 * c^3 = a_0 + s[a_1 + s[a_2 + s[a_3]]],$$

where $s[a] = a \cdot c$. Kärkkäinen[11] uses this approach in an implementation of a two-dimensional string matching algorithm.[12] This method needs some additional space and preprocessing time for the look-up table s.

One solution to the space problem is to apply hashing, which, however, makes the searching phase slower than table access. Zhu and Takaoka[10] present a two-level hashing scheme for $q = 2$. They maintain synonym links, but our experiments suggest that on the average it is more advantageous in practice to use the smaller shift value in the case of a collision and to examine also the last q-gram of the potential occurrence, if necessary, when the fingerprints match.

Another approach to save space is to cluster the alphabet by mapping input characters to a narrow range. The simplest mapping is to subtract 97 from the ASCII codes of characters `a`, `c`, `g`, and `t` to get 0, 3, 7, and 19, respectively, which means that the size of the resulting alphabet is 20. Kim and Shawe-Taylor[4] introduce another mapping: they use the last three bits of the character code, which means that the size of the resulting alphabet is eight. However, the initialization of the shift table is still slow for large values of q in the alphabet of eight characters. Note also that the transformation of Kim and Shawe-Taylor does not work for arbitrary character codes.

Most of these simple approaches may produce errors, if the input happens to contain improper symbols. A safe way for mapping is to apply a direct alphabet transformation. We map the ASCII codes to the range of 4: $0 \leq r[j] \leq 3$ such that characters `a`, `c`, `g`, and `t` get different codes and possible other characters get, for example, code 0. In this way we are able to limit the computation to the effective alphabet of four characters. We use a separate transformation table h_i for each position i of a q-gram and incorporate multiplications into

the tables: $h_i[j] = r[j] \cdot 4^i$. For $q = 4$, the fingerprint of $a_0 \ldots a_3$ is then computed as

$$h_0[a_0] + h_1[a_1] + h_2[a_2] + h_3[a_3].$$

Algorithm PP describes the preprocessing of the transformation tables and the shift table D for $q = 4$. The algorithm has a different code for each value of q (q can be a compilation parameter in C). Note that $r[j] = h_0[j]$ and $h_3[x] = h_0[x] * u/c$ hold.

Algorithm PP

```
 1  c:=4; q:=4;
 2  for i:=0 to 255 do h0[i]:=0;
 3  h0['a']:=0; h0['c']:=1;
 4  h0['g']:=2; h0['t']:=3;
 5  for i:=0 to 255 do begin
 6     h1[i]:=c*h0[i];
 7     h2[i]:=c*h1[i];
 8     h3[i]:=c*h2[i] end;
 9  u:=c**q;
10  for i:=0 to u-1 do D[i]:=m;
11  s:=0; a:=u;
12  for i:=1 to q-1 do begin
13     s:=s div c+h3[P[i]];
14     a:=a/c;
15     for j:=s to s+a-1 do D[j]:=m-i end;
16  s:=s div c+h3[P[q]];
17  for i:=q+1 to m do begin
18     D[s]:=m-i+1;
19     s:=s div c+h3[P[i]] end;
```

Note that when the data contains more than four different characters, the mapping r is not an injection. So this mapping defines a simple hashing scheme. In Algorithm 2, the q-gram corresponding to t needs never to be re-examined in the checking phase. In the case of a non-injective mapping that q-gram must be re-examined, if the fingerprints are equal and $P[1], \ldots, P[m-q]$ match with the text. However, the slow down due to this is insignificant in the average case while using the forward scan.

When the data contains more than four different characters, there is also another problem. Namely, if a q-gram of the text contains a rare character, the shift for it is possibly shorter with the mapping r than without it. For example, if the pattern is `cacgtcccc` and the q-gram is `xcgt` and *r*[a] = *r*[x], the shift for this q-gram is 5 with the mapping and 9 without it. However, the frequency of other characters than `a`, `c`, `g`, and `t` is low in real DNA data so that this slow down is insignificant in practice.

When applying the character transformation to a larger alphabet, where there are several text characters that do not occur in the pattern, all these characters can be mapped together. If the total frequency of these characters is low, the size of the effective alphabet is the number of different characters in the pattern, otherwise one more.

Algorithm 3

```
1  Execute Algorithm PP;
2  p:=f2(P,m,q);
3  k:=m;
4  repeat
5     t:=f2(T,k,q);
6     if t=p then
7        Check the potential occurrence;
8     k:=k+D[t];
9  until k>n;
```

Algorithm 3 is Algorithm 2 with the alphabet transformation. On line 7 characters $P[1], \ldots, P[m]$ are compared with the corresponding text characters until a mismatch is found or a match is completed. The notation $f2$ refers to the computation of fingerprints based on transformation tables created by Algorithm PP.

SKIP LOOP

In Algorithm 3 there are two tests in the main loop: on lines 6 and 9. The loop would be faster, if one of the tests could be removed. Such a loop with only one test is called a skip loop[3] or a fast loop.[1] Algorithm 4 is a modification of Algorithm 3 containing the variation of a skip loop called 'unrolled fast' by Hume and Sunday.[3]

Algorithm 4

```
 1  Execute Algorithm PP;
 2  p:=f2(P,m,q);
 3  x:=D[p];
 4  D[p]:=0;
 5  k:=m;
 6  s:=D[f2(T,k,q)];
 7  fast:
 8   repeat
 9      k:=k+s;
10      k:=k+D[f2(T,k,q)];
11      k:=k+D[f2(T,k,q)];
12      s:=D[f2(T,k,q)];
13   until s=0;
14   if k>n then exit;
15   Check the potential occurrence;
16   s:=x;
17  goto fast;
```

The value of $D[p]$ is stored to a variable x and $D[p]$ is set to zero in order to make the skip loop on lines 8–13 work properly. That loop is repeated until a q-gram with a matching fingerprint is found. In order to be able to detect the end of the text, an occurrence of the pattern must be copied to $T[n+1] \ldots T[n+m]$. The loop contains 3-fold unrolling. (A basic skip loop without unrolling is got by leaving lines 10 and 11 out.) The condition $D[p] = 0$

Table I. Searching times (in seconds)

m	G	RF	KS	B.4	B.6	BC.6	BC.7	A3.6	A4.4	A4.5	A4.6
25	15.57	17.02	15.93	12.96	21.34	22.34	27.11	15.90	**9.30**	10.36	13.30
50	12.35	10.30	9.37	7.60	11.87	11.65	13.83	8.95	**5.80**	6.39	8.07
100	9.89	6.39	4.75	4.08	5.72	5.66	6.55	4.34	**3.19**	3.28	4.06
200	7.75	3.79	2.37	2.35	2.84	2.77	3.26	2.15	1.89	**1.72**	2.06
400	6.70	2.44	1.22	1.56	1.45	1.37	1.56	1.16	1.24	**0.93**	1.05
800	5.67	1.55	0.70	1.16	0.73	0.71	0.80	0.55	0.94	**0.54**	**0.54**
1600	4.83	1.00	0.41	0.96	0.41	0.40	0.40	**0.31**	0.78	0.36	**0.31**
3200	4.38	0.73	0.27	0.95	0.25	0.24	0.22	**0.19**	0.80	0.29	0.21

guarantees that the variable k is not increased before exiting the loop, though a q-gram with a matching fingerprint is detected on any phase of the loop.

The unrolling factor of three works well in practice. The best value of the factor depends on the pattern, the text, and the computer architecture. If the control often leaves the skip loop, a smaller factor is better. However, the effect of unrolling is rather small for long DNA patterns.

EXPERIMENTAL RESULTS

We tested several algorithms on DNA data. Our DNA data consists of a text of 1,982,672 characters (nucleotides) and eight pattern files each containing 200 patterns of a fixed length, from 25 to 3200 characters.* The total number of occurrences of the pattern classes in the text ranges from 14 to 40. The text and the patterns were picked from the EMBL sequence database.[13] The text is a combination of nucleotide sequence strings shorter than 20,000 and longer than 8500 characters. The patterns are randomly selected continuous nonoverlapping strings of the database. The alphabet consists of characters `a`, `c`, `g`, `t`, and `n` besides the line feed. The character `n` represents an uncertain character, and its frequency is low.

We implemented Algorithms 3 and 4 in C and tested them with different values of q. Our testing framework is a modification of the one used by Hume and Sunday.[3] Tables I–III report the results of the experiments carried out in a Linux workstation with 16 MB RAM and a 90 MHz Pentium processor. The searching, preprocessing, and total times were recorded for four versions A3.6, A4.4, A4.5, and A4.6, where the suffix expresses the value of q. We also tested a variation of the Baeza-Yates algorithm[5] where the direct character transformation is the only addition the original algorithm. These versions are denoted by BC.6 and BC.7 for $q = 6, 7$. As a comparison, we executed the same tests with implementations of five earlier algorithms. For each algorithm, the mean timing of one hundred runs is given and the best value is shown in boldface on each row. Each run includes the processing of 200 different patterns.

Algorithm G is `ufast.rev.gd2` of Hume and Sunday,[3] and it applies a two-dimensional shift table indexed by the pattern position and the mismatching text character. This heuristic[1] gives longer shifts than the two usual Boyer–Moore heuristics, because the shift is computed for each possible mismatching character for each suffix of the pattern. Algorithm G was the best for DNA data in the comparison of Hume and Sunday.[3]

Algorithm RF is the reverse-factor algorithm by Lecroq.[7,14] The key feature of this algorithm

* The algorithms and the data are available at <http://www.cs.helsinki.fi/∼tarhio/sm/>.

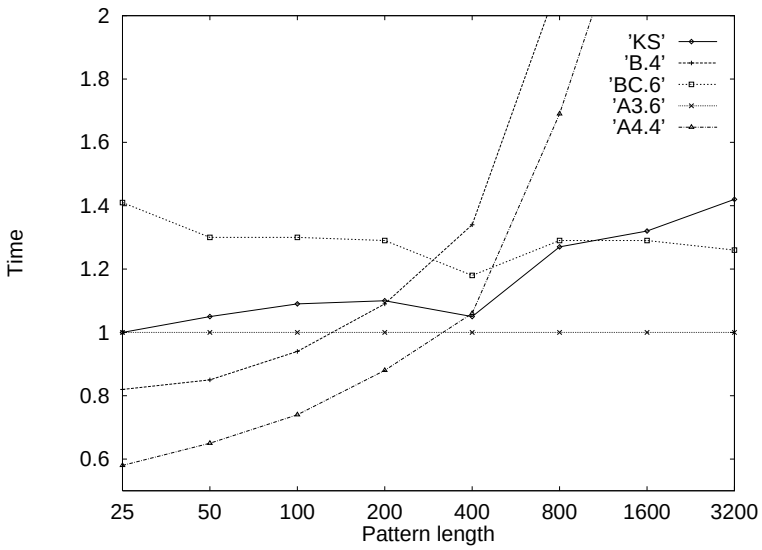

Figure 1. Relative searching times (A3.6 = 1)

is to control searching with the suffix automaton of a reversed pattern. Algorithm RF was the best for DNA data in the comparison performed by Lecroq.[7]

Algorithm RF is related to the optimized n-gram algorithm presented by Kim and Shawe-Taylor.[4] This algorithm (named here as Algorithm KS) uses on the first level a trie of reversed q-grams of the pattern, where each level of the trie holds a separate index. Algorithms B.4 and B.6 are the block algorithms of Kim and Shawe-Taylor[4] for $q = 4, 6$. The block algorithms were the best for DNA data in the comparison performed by Kim and Shawe-Taylor.[4]

In searching times (Table I), Algorithm A3.6 is the fastest for patterns of 1600–3200 characters, Algorithm A4.6 is the fastest for patterns of 800–1600 characters, Algorithm A4.5

Table II. Preprocessing times (in seconds)

m	G	RF	KS	B.4	B.6	BC.6	BC.7	A3.6	A4.4	A4.5	A4.6
25	0.10	0.38	0.89	0.10	7.26	0.07	0.31	0.10	**0.03**	0.05	0.13
50	0.19	0.82	0.90	0.10	7.27	0.08	0.34	0.11	**0.03**	0.05	0.14
100	0.38	1.77	0.95	0.11	7.27	0.08	0.34	0.11	**0.03**	0.05	0.14
200	0.75	3.64	1.02	0.12	7.27	0.08	0.37	0.11	**0.03**	0.05	0.14
400	1.50	7.83	1.17	0.14	7.34	0.09	0.35	0.12	**0.04**	0.06	0.15
800	2.98	16.71	1.48	0.18	7.39	0.12	0.40	0.13	**0.05**	0.08	0.16
1600	5.97	34.80	2.04	0.26	7.51	0.16	0.44	0.17	**0.08**	0.10	0.19
3200	11.98	70.58	3.08	0.41	7.71	0.24	0.52	0.23	**0.14**	0.16	0.26

Table III. Total times (in seconds)

m	G	RF	KS	B.4	B.6	BC.6	BC.7	A3.6	A4.4	A4.5	A4.6
25	15.67	17.40	16.82	13.06	28.60	22.41	27.43	16.00	**9.32**	10.41	13.43
50	12.54	11.12	10.27	7.70	19.13	11.72	14.17	9.06	**5.82**	6.44	8.22
100	10.27	8.15	5.70	4.19	12.99	5.74	6.89	4.45	**3.22**	3.33	4.20
200	8.50	7.43	3.39	2.47	10.12	2.85	3.63	2.26	1.92	**1.78**	2.19
400	8.19	10.27	2.39	1.69	8.80	1.46	1.91	1.28	1.28	**0.99**	1.19
800	8.65	18.27	2.18	1.33	8.12	0.82	1.20	0.69	0.99	**0.61**	0.70
1600	10.80	35.80	2.45	1.21	7.93	0.55	0.83	0.48	0.86	**0.46**	0.49
3200	16.36	71.30	3.35	1.35	7.97	0.48	0.75	**0.42**	0.94	0.46	0.46

is the fastest for patterns of 200–800 characters, and Algorithm A4.4 is the fastest for patterns of 25–100 characters. Figure 1 shows the relative searching times of KS, B.4, BC.6, A3.6, and A4.4.

Algorithms A4.4, A4.5, and A4.6 contain the skip loop. We tried them also without the skip loop. The benefit of the skip loop decreases when the pattern gets longer. For $m = 25$ the speed benefit is 4% and for $m = 3200$ the algorithms are already faster without the skip loop. When searching for short words in an English text, the benefit of the skip loop may be even 20%.[3] The advantage in our setting is considerable smaller.

In preprocessing times (Table II) Algorithm A4.4 is the fastest for all pattern sizes. Algorithm BC.4 would beat Algorithm A4.4 in the preprocessing speed, but because its searching speed is even slower than that of Algorithm B.4, we left it out. The preprocessing of Algorithms A3.6 and A4.4–A4.6 could be made a bit faster by initializing transformation tables during compilation time.

Algorithms A4.4, A4.5, and A4.6 use the shift table D and B.4, B.6, BC.6, BC.7, and A3.6 the shift table D'.

In total times (Table II) Algorithm A3.6 is the fastest for patterns of 3200 characters, Algorithm A4.5 is the fastest for patterns of 200–1600 characters, and Algorithm A4.4 is the fastest for patterns of 25–100 characters.

Comparing total times is troublesome, because the length of text affects the relative proportion of the preprocessing time. However, it would not be fair to neglect preprocessing times totally, because the preprocessing time is considerable in the case of long patterns.

We also tested our algorithms with other values of q. The average searching time of Algorithm A4.3 is 8.88 seconds for patterns of 25 characters, and so it is slightly faster than A4.4 for this length. The average searching time of Algorithm A3.7 is 0.17 seconds for patterns of 3200 characters, and so it is somewhat faster than A3.6 for this length. However, the preprocessing time of A3.7 is 0.50 seconds, which makes it slower in the total time. Because the advantage of A4.3 and A3.7 is only marginal, we left them out from Tables I–III. Algorithms B.q and BC.q for other values of q were not competitive in our setting.

MEMORY REQUIREMENTS

To compare the memory requirements of the algorithms, we calculated the maximal size of global data structures used during searching in the C implementation. These figures are given

Table IV. Memory usage of data structures (1000 bytes)

m	G	RF	KS	B.4	B.6	BC.6	BC.7	A3.6	A4.4	A4.5	A4.6
25	13	52	133	16	1049	17	67	23	5	9	23
50	26	103	134	16	1049	17	67	23	5	9	23
100	52	206	138	16	1049	18	67	23	5	9	23
200	103	412	145	17	1049	18	67	23	5	9	23
400	207	825	160	17	1049	18	67	23	6	10	23
800	414	1650	188	17	1049	18	67	23	6	10	23
1600	827	3299	246	18	1050	19	68	24	7	11	24
3200	1654	6598	361	20	1052	21	70	26	8	12	26

in Table IV. Note that the memory requirements of the reference methods are based on the original implementations of these algorithms, and those implementations are not necessarily tuned to achieve the maximal space-efficiency while maintaining their speed.

Algorithm KS uses the alphabet of 8 characters and the tested implementation of Algorithm G uses the alphabet of 128 characters. The memory usage of KS is upper limit, because it initializes data structures only when it needs them.

In these calculations, we assume that we have integers and pointers (memory addresses) of four bytes and no padding in and between records (structs in C).

CONCLUSION

Our approach reduces the processing time and the space requirement of the Baeza-Yates algorithm, and allows the use of longer q-grams for a still faster searching phase in the DNA alphabet. Our algorithms reduce the need for paging of the virtual memory, and they even fit computers with a small main memory. Our approach is independent of the character codes, and it is also applicable to other small alphabets having less than 15 common characters. The alphabet clustering technique can even be used with large alphabets.

None of the algorithms was the best for all pattern lengths tested. A hybrid algorithm, which selects q according to the length of the pattern, would have the best general efficiency. A good estimate[9] for the value of q is $\log_4(4m)$.

The implementation techniques we presented are not limited to C. That is why we gave our algorithms in a pseudocode close to Pascal. We have also tested Algorithms 1–4 in Pascal with comparable results.

Algorithms RF[7] and KS[4] are theoretically appealing, because they examine fewer text characters than the other algorithms we tested. However, they are slower than Algorithm A3.6 because of the overhead of transitions in an automaton or in a tree. Also, the space requirements of RF and KS are large. Algorithm KS was clearly faster than RF in our tests. The obvious reason for this is that it restricts the height of the suffix trie. An interesting topic for a further study would be to find faster ways to traverse a suffix trie.

ACKNOWLEDGEMENTS

We thank K. Rämö for fruitful discussions. We also thank J. Shawe-Taylor and T. Lecroq for

providing the C codes of their algorithms. We thank a referee for suggesting of testing the BC algorithms. The financial support of the Academy of Finland is appreciated.

REFERENCES

1. R. S. Boyer and S. Moore, 'A fast string searching algorithm', *Communications of the ACM*, **20**, 762–772 (1977).
2. S. Wu and U. Manber, 'Fast text searching allowing errors', *Communications of the ACM*, **35**(10), 83–91 (1992).
3. A. Hume and D. Sunday, 'Fast string searching', *Software—Practice and Experience*, **21**(11), 1221–1248 (1991).
4. J. Kim and J. Shawe-Taylor, 'Fast string matching using an n-gram algorithm', *Software—Practice and Experience*, **24**, (1), 79–88 (1994).
5. R. Baeza-Yates, 'Improved string searching', *Software—Practice and Experience*, **19**(3), 257–271 (1989).
6. N. Horspool, 'Practical fast searching in strings', *Software—Practice and Experience* **10**(6), 501–506 (1980).
7. T. Lecroq, 'Experimental results in string matching algorithms', *Software — Practice and Experience* **25**(7), 727–765 (1995).
8. D. Sunday, 'A very fast substring search algorithm', *Communications of the ACM*, **33**, 132–142 (1990).
9. R. Sedgewick, *Algorithms*, Addison-Wesley, 1983.
10. R. Zhu and T. Takaoka, 'On improving the average case of Boyer–Moore string matching algorithm', *Journal of Information Processing*, **10**, 173–177 (1987).
11. J. Kärkkäinen, Personal communication, 1994.
12. J. Kärkkäinen and E. Ukkonen, 'Two and higher dimensional pattern matching in optimal expected time', *Proceedings of the Fifth Symposium on Discrete Algorithms*, ACM-SIAM, 1994, pp. 715–723.
13. The EMBL Sequence Database, Version 41, 1995, URL: ftp://ftp.ebi.ac.uk/.
14. T. Lecroq, 'A variation on the Boyer–Moore algorithm', *Theoretical Computer Science*, **92**, 119–144 (1992).