Comparison of Exact String Matching Algorithms for Biological Sequences*

Petri Kalsi, Hannu Peltola**, and Jorma Tarhio

Department of Computer Science and Engineering
Helsinki University of Technology
P.O. Box 5400, FI-02015 HUT, Finland
petri.kalsi@iki.fi, {hpeltola, tarhio}@cs.hut.fi

Abstract. Exact matching of single patterns in DNA and amino acid sequences is studied. We performed an extensive experimental comparison of algorithms presented in the literature. In addition, we introduce new variations of earlier algorithms. The results of the comparison show that the new algorithms are efficient in practice.

1 Introduction

String matching is used for locating nucleotide or amino acid sequence patterns in the biological sequence databases. Nucleotide or DNA sequences have an alphabet of four characters, and amino acid sequences have an alphabet of 20 characters, respectively. Because the total number of sequences is rapidly increasing, efficient methods are needed. Recently three novel algorithms, namely Lecroq's 'New' [13], SSABS [16], and TVSBS [19], were proposed for searching single exact pattern in DNA and amino acid sequences. This motivated us to compare these algorithms with older methods known to be efficient. Another reason is advances in the processor and memory technology and in compilers. Old algorithms do not necessarily behave with new processors and compilers in the same way they did ten years ago in our extensive comparison [18]. Besides the comparison in the DNA and amino acid alphabets, we developed new variations of our earlier algorithm [18].

It turned out that SSABS and TVSBS were poor for DNA sequences. The new variations of our algorithm were slightly faster than Lecroq's algorithm in most cases in the both alphabets. The clear winner for amino acid patterns of moderate length was SBNDM2 [6], which has not been specifically designed for biological alphabets.

In terms of computer science, a biological sequence is a string, and therefore we use both terms. The aim of string matching is to report all exact occurrences of a pattern string in a longer text string. We use the following notations

* Supported by the Academy of Finland
** The corresponding author: hpeltola@cs.hut.fi

throughout the paper. The length of the pattern and the text are m and n, respectively. The size of the alphabet is σ.

The rest of the paper is organized as follows. We review earlier solutions in Section 2 and introduce the new variations of our algorithm in Section 3. Section 4 reports the results of our experiments before the conclusion in Section 5.

2 Known Solutions for DNA Sequences

Most of the efficient string matching algorithms in the DNA alphabet are modifications of the Boyer–Moore algorithm [3]. Most often only the occurrence heuristic (also called the bad character heuristic) is applied for shifting. Algorithms of this type are greedy in the sense that the pattern is moved forward after the first character mismatch of an alignment is observed. Shifts may then be unnecessarily short for the nucleotide alphabet if shifting is based on a single character. Therefore it is advantageous to apply q-grams, strings of q characters, instead of single characters. This technique was already mentioned in the original paper of Boyer and Moore [3, p. 772], and Knuth [12, p. 341] analyzed theoretically its gain. Zhu and Takaoka [21] presented the first algorithm utilizing the idea. Their algorithm uses two characters for indexing a two dimensional array. They gave also another version based on hashing. The size of the hash table was two times the pattern length.

Baeza-Yates [1] introduced an extension to the Boyer–Moore–Horspool algorithm [7] where the shift array is indexed with an integer formed from a q-gram with `shift` and `add` instructions. For this kind of approach the practical upper limit with 8 bit characters is two characters.

For the DNA alphabet Kim and Shawe-Taylor [10] introduced a convenient alphabet compression by masking the three lowest bits of ASCII characters. In addition to the `a`, `c`, `g`, and `t` one gets distinguishable codes also for `n` and `u`. Even important control code \n=LF has distinct value, but \r=CR gets the same code as `u`. With this method they were able to use q-grams of up to six characters. Indexing of the shift array was similar to Baeza-Yates' algorithm.

With a small alphabet the probability of an arbitrary q-gram appearing in long pattern is high. This restricts the average shift length. Kim and Shawe-Taylor [10] introduced also an interesting variation for the cases where the q-gram in the text occurs in the pattern. Then two additional characters are checked one by one to achieve a longer shift.

In most cases the q-gram that is taken from the text does not match with the suffix of the pattern, and the pattern can be shifted forward. For efficiency one should use a *ufast* skip loop [8], where the pattern is moved forward until the last q-gram of the pattern matches with a q-gram in the text. The easiest way to implement this idea is to artificially define the shift of the last q-gram of the pattern to be zero. We must also add to the end of the text a copy of the pattern as a stopper. After the skip loop the pattern is compared with the corresponding text positions. An advantage of skip loop is that we use text characters both for

preliminary comparison and for computing the shift. So we fetch the characters only once.

Our earlier algorithm [18] applies a skip loop. The key feature of this algorithm is efficient handling of q-grams. The details are presented in Section 3.1. Recently Lecroq [13] presented a related algorithm. Its implementation is based on the Wu–Manber algorithm [20] for multiple string matching, but as suggested above, the idea is older [3,21].

SSABS [16] and TVSBS [19] were developed for biological sequences. SSABS is a Boyer–Moore type algorithm. In the search phase the algorithm verifies that the first and last character of the pattern matches with the current alignment before checking the rest of the alignment (a.k.a. guard tests). TVSBS uses a 2-gram for calculating the shift, adopted from the Berry–Ravindran algorithm [2], which is a cross of the Zhu–Takaoka algorithm and Sunday's QS algorithm [17]. Instead of the two-dimensional shift table of Berry–Ravindran, TVSBS uses a hash function to compute an index to a one-dimensional table.

In Section 4, we will present an experimental comparison of several algorithms. Experimental results on some earlier algorithms for DNA sequences can be found in [18]. We have included three efficient algorithms, namely FAOSO [4], SBNDM [14,15], and SBNDM2 [6] in the comparison, although they have not been designed for biological sequences.

Recently Kim et al. [11] presented an algorithm for packed DNA. We excluded it from the comparison, because there is no fair way to compare other algorithms with it.

3 Variations of Our Algorithm

3.1 Our Earlier Algorithm

Our algorithm [18] is a modification of the Boyer–Moore–Horspool algorithm [7] for the DNA alphabet. Instead of inspecting a single character at each alignment of the pattern, the algorithm reads a q-gram and computes an integer called fingerprint from it. This idea of applying q-grams was already present in the earlier algorithms [1,10,21], but we introduced an efficient way to handle q-grams.

The ASCII codes of `a`, `c`, `g`, and `t` are 97, 99, 103, and 116, respectively. The ASCII codes are mapped to the range of 4: $0 \leq r[x] \leq 3$, where $r[x]$ is the new code of x, such that characters `a`, `c`, `g`, and `t` get different codes and other possible characters get e.g. code 0. In this way the computation is limited to the effective alphabet of four characters. The fingerprint is simply a reversed number of base 4. A separate transformation table h_i is used for each position i of a q-gram and multiplications are incorporated during preprocessing into the tables: $h_i[x] = r[x] \cdot 4^i$. For $q = 4$, the fingerprint of $x_0 \cdots x_3$ is $\sum_{i=0}^{3} r[x_i] \cdot 4^i$ which is then computed as

$$h_0[x_0] + h_1[x_1] + h_2[x_2] + h_3[x_3].$$

The algorithm is given below as BMHq. It corresponds to Algorithm 4 in [18] without unrolling. In the algorithm, $T = t_1 \cdots t_n$ denotes the text and $P =$

$p_1 \cdots p_m$ the pattern. 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, a potential occurrence has been found, which has to be checked. We need to check only the first $m - q$ characters of the pattern because our fingerprint method does not cause hash collisions, assuming that the searched text contains only DNA characters. The algorithm applies a variation of a skip loop called "unrolled fast" (=*ufast*) by Hume and Sunday [8].

Algorithm 1 BMHq($P = p_1 p_2 \cdots p_m, T = t_1 t_2 \cdots t_n$)

```
 1: Initialize D[*]
 2: t_{n+1} ... t_{n+m} ← P        /* adding stopper */
 3: p ← f(P, m, q)
 4: r ← D[p]; D[p] ← 0; k ← m
 5: s ← D[f(T, k, q)]
 6: loop
 7:     while s > 0 do
 8:         k ← k + s
 9:         s ← D[f(T, k, q)]
10:     if k > n then exit        /* CORRECTED LINE */
11:     Check the potential occurrence
12:     s ← r
```

In the algorithm, $f(T, k, q)$[1] denotes the fingerprint of $t_{k-q+1} \cdots t_k$. For computation of the shift table D see [18]. The pattern needs to be copied to the end of the text, so that the *ufast* skip loop will end when the search is complete.

Lecroq's 'New' algorithm [13] is closely related to BMHq. However, Lecroq applies a different method based on hashing for computing fingerprints of q-grams. Moreover, the maximal shift of his algorithm is $m - q + 1$, while that of BMHq is m, because BMHq is able to handle all prefixes of the first q-gram of the pattern.

3.2 Variations for DNA

We tested several ways to compute the fingerprint in order to make our algorithm BMH4 faster. When considering 4-grams, they fit into a 32-bit word in 8-bit characters. Some CPU architectures, notably the x86, allow unaligned memory reads of several bytes. This inspired us to try reading several bytes in one instruction, instead of four separate character reads. Reading several bytes at a time is by no means a new technique. Many researchers have used it [5,9]. But we have not seen any comparison with standard bytewise reading. One may argue that is not fair to apply multiple reading, because all CPU architectures

[1] This is f2 in [18].

do not support it. But the x86 architecture is nowadays so dominant that it is reasonable to tune algorithms for it.

We will present two variations of BMH4. BMH4b reads a 32-bit word, and BMH4c reads two consecutive halfwords. Because in BMH4b we have access to an integer consisting of four characters, it would be inefficient to use the old character-based fingerprint method. The fingerprint calculation arrays of BMH4 are replaced with hashing expression, where the input is a whole 4-gram as a 4-character long integer. ASCII codes for `a`, `c`, `g` and `t` are distinguishable by the last three bits. The following expression packs the unique bits of the four characters together in a few instructions, to form an integer in the range of 2313...16191 = 00100100001001...11111100111111.

```
FP(x) = ((x >> 13) & 0x3838) | (x & 0x0707)
```

Preprocessing of BMH4b is similar to the earlier algorithm, we just calculate the fingerprints of the 4-grams in the pattern with the new hash function. So the main difference between BMH4b and BMH4 is in the computing of f. In BMH4b this is done with masking, shifting, and bitwise OR. Hashing receives the current text location pointer as an argument, and reads the 32-bit integer from that address with `FP(*(k-3))`. $D[x]$ contains the preprocessed shift values for each hashed q-gram of the pattern.

Based on our tests, unaligned memory reads on x86 processors incur a speed penalty of up to 70% when compared with aligned reads. This unfortunately reduces the speed of BMH4b, because 75% of the reads are unaligned on the average. So we made another variation BMH4c, which reads two consecutive halfwords. In the case of BMH4c, only 25% of the reads are unaligned ones getting the speed penalty (coming on the border of 4 bytes). In BMH4c, the value of a fingerprint is got as $a_1[x_1] + a_2[x_2]$ where x_i is a halfword and a_i a preprocessed transformation table, for $i = 1, 2$.

3.3 Variations for Amino Acids

The q-gram approach is valid also for larger alphabets, although with a larger alphabet the optimal value of q is smaller. A larger alphabet size required only minor changes to the algorithm. A new variation BMH2 was created based on BMHq. The range of the ASCII code mapping $r[x]$ was increased from 4 to 20, to cover the amino acid alphabet "ACDEF GHIKL MNPQR STVWY" instead of the DNA alphabet. Otherwise the algorithm is the same as BMHq.

We made also BMH2c, which reads a 2-gram as a 16-bit halfword. The shift array is indexed directly with halfwords.

These algorithms can be used with any text, e.g. English text. For this kind of data we mapped each character to a smaller range with a modulo function in preprocessing. The best results for English text were obtained with modulo 25. Because the mapping tables are created in the preprocessing phase, the modulo operation does not affect the search time directly.

4 Experimental Results

Algorithms. Among tested algorithms were SSABS [16], TVSBS [19], BMH4, BMH4b, BMH4c, BMH2, and BMH2c. KS by Kim and Shawe-Taylor [10] uses a trie of reversed q-grams of the pattern. The Fast Average Optimal Shift-Or (FAOSO) [4] was tested with different values of the step size and the unrolling factor. The given run times of FAOSO are based on the best possible parameter combination for each pattern length.

SBNDM [15] is based on the BNDM [14] algorithm, with simplified shift calculations. SBNDM2 [6] is a modification of SBNDM. One of the key points of SBNDM2 is loop unrolling, i.e. handling of a 2-gram before entering to the inner loop.

We also tested LEC, which is the 'New' algorithm of Lecroq [13], and which uses q-grams and hashing. The run times of LEC are given based on the best possible q for each pattern length.

SSABS and TVSBS were implemented as described in the articles [16] and [19]. Because TVSBS is based on Berry–Ravindran algorithm [2], we also implemented two versions of the latter, BR and BRX. In BRX we have modified the algorithm to calculate shifts based on the last character of the current text alignment t_s and the next character t_{s+1}, instead of t_{s+1} and t_{s+2} as in BR which corresponds to the original Berry–Ravindran. The probability of a shift of the pattern by one position is approximately $1/\sigma$ for BR and $1/\sigma^2$ for BRX. Thus BRX produces with small alphabets longer shifts than BR on the average. This is due to the inherent weakness of Sunday's QS algorithm [17]: if the last character of the pattern is common, then the probability of a shift of one is high. In such a case, t_{s+2} is often useless in the computation of shift in BR. Our test results show that this weakness of BR is noticeable even with amino acids.

All algorithms were implemented in C. The codes of KS, SBNDM2, FAOSO, and LEC were got from the original authors. Other codes were implemented by us.

Test setting. The tests were run on a 2.8GHz Pentium D (dual core) CPU with 1 GB of memory. Both cores have 16 KB L1 data cache and 1024 KB L2 cache. The computer was running Fedora 8 Linux.

All the algorithms were tested in a testing framework of Hume and Sunday [8]. All programs were compiled with Intel's C compiler `icc 10.0` producing x86 "32-bit" code and using the optimization level `-O3`.

The test patterns were generated based on the text files so that roughly half of the patterns have matches in the text. These longer patterns were then cut to shorter patterns. Every pattern set contains 200 patterns for DNA and 100 for amino acids of the same length. For DNA, the test data was taken from the genome of the fruit fly. The amino acid text is the peptide sequences of *Arabidopsis thaliana*. All text files consisted of about $2 \cdot 10^6$ characters. Test runs were executed on otherwise unloaded computer. To achieve accurate timings the search for each pattern was repeated 20 times (with DNA and 50 times with amino acids). Reported results are medians for five successive test runs using

corresponding pattern set. The change of the process from one processor core to another empties cache memories with various degree. This would slow down reads from memory and induce annoying variation to the timing of test runs. To avoid it we have used Linux function `sched_setaffinity` to bind the process to only one processor or core.

DNA sequences. The search times for DNA sequences are listed in Table 1. The reported times do not include preprocessing. For most of the tested algorithms the preprocessing times were less than 1% of search times. The preprocessing time was slightly more for the BMH2c, BMH4b, BMH4c, TVSBS, KS, BR, and BRX algorithms, because they have large memory structures, which are initialized during preprocessing.

Table 1. Search times in milliseconds for DNA sequences ($\sigma = 4$)

m	BMH4	BMH4C	SSABS	TVSBS	KS	BR	BRX	SBNDM	SBNDM2	FAOSO	LEC$_{3-5}$
4	1004	806	1991	1559	-	1545	1294	1754	1152	601	1390
6	678	540	1750	1260	1126	1250	984	1246	885	558	762
8	534	429	1612	1073	592	1063	811	984	743	357	551
10	447	355	1570	945	411	936	696	806	657	306	450
12	391	305	1586	866	324	860	624	692	582	276	379
14	351	270	1556	802	281	795	573	603	515	278	327
16	317	250	1551	755	241	747	538	538	462	228	295
18	286	236	1526	716	212	710	506	482	417	235	272
20	257	224	1516	683	196	672	477	440	381	203	256
22	240	210	1536	663	179	654	462	406	349	205	244
24	232	199	1519	645	170	633	442	375	321	227	231
26	220	191	1491	620	162	612	431	348	299	244	220
28	209	182	1526	602	154	593	417	327	280	228	209
30	201	178	1529	589	149	580	407	308	263	242	198
40	173	156	1492	545	126	536	377	-	-	-	168
50	156	144	1481	521	113	511	361	-	-	-	150
60	146	135	1432	501	106	491	346	-	-	-	139
70	135	126	1495	498	105	489	343	-	-	-	129
80	125	112	1551	498	102	489	334	-	-	-	121
90	117	110	1588	494	102	483	334	-	-	-	114
100	111	108	1569	502	107	491	335	-	-	-	108

FAOSO, KS, and BMH4c were winners of the DNA test. FAOSO was the fastest for $m \leq 16$ excluding $m = 6$ and $m = 14$ where BMH4c was the best. KS was the fastest for $18 \leq m \leq 100$.

Multiple reading is advantageous, because BMH4c was approximately 20% faster than BMH4 for short patterns. The times of BMH4b (not shown) were

Table 2. Search times in milliseconds for amino acid sequences ($\sigma = 20$)

m	BMH2	BMH2C	SSABS	TVSBS	BR	BRX	SBNDM	SBNDM2	FAOSO	LEC_{3-4}
4	320	277	326	336	316	288	346	241	164	622
6	225	195	252	260	246	214	294	138	123	340
8	280	156	207	213	206	172	250	104	101	245
10	152	133	186	187	177	149	219	84	91	201
12	131	118	173	170	161	133	196	74	79	166
14	119	106	155	148	142	119	175	67	79	142
16	112	98	145	137	132	109	157	62	75	128
18	107	91	137	128	121	101	141	59	69	118
20	101	85	132	120	115	96	127	55	70	112
22	94	81	127	116	109	90	116	53	70	105
24	90	77	122	107	103	85	106	51	68	99
26	86	74	118	102	98	82	97	50	69	93
28	83	71	115	98	93	79	90	49	69	89
30	80	69	114	96	93	77	84	48	68	86
40	72	62	103	81	79	67	-	-	-	74
50	67	58	98	74	71	61	-	-	-	67
60	63	55	95	69	67	58	-	-	-	62
70	58	51	92	63	61	54	-	-	-	59
80	53	48	92	58	57	51	-	-	-	55
90	51	48	91	56	53	50	-	-	-	52
100	51	49	91	54	52	49	-	-	-	50

between those of BMH4 and BMH4c for patterns longer than 25 nucleotides; for shorter patterns is was a little slower than BMH4. Even BMH2c (not shown) was among the best for patterns shorter than 10.

TVSBS proves to be an improvement on SSABS for long patterns, but its performance still falls in line with the other algorithms of the comparison. However, the test results show that SSABS and TVSBS are not appropriate for DNA sequences. The original Berry–Ravindran algorithm was slightly faster than TVSBS. The results show that the shift modification in BRX is a clear improvement on BR for DNA sequences.

Amino acid sequences. The search times in the amino acid alphabet in Table 2 are more even. BMH2c works efficiently for all tested pattern lengths, but SBNDM2 is the overall winner of this test. Because the bitvectors were 32 bits long, BMH2c and BRX dominate when $m > 32$. Performance of BMH2 is close to BR and BRX for all tested pattern lengths.

Additional notes. Further optimizations might improve the search times of TVSBS, especially the same one we used in BRX. With 64-bit bitvectors FAOSO,

SBNDM, and SBNDM2 can handle longer patterns than in these tests. On separate test with "64-bit code", SBDNM2 was the fastest also for pattern lengths up to 64. Currently we have only 32-bit version of FAOSO, so its performance with longer bitvectors remained unknown.

It was a surprise that KS was the winner for long patterns, because it was slower in our earlier test [18] and also in the preliminary test with an older compiler for this paper. The improvement of KS is due to new compilers. With `gcc 4.1` the performance is nearly the same as in Table 1, but when compiled with `gcc 4.0.1`, `3.4.4`, or `3.3.5` the search times are at least 12% longer on several computers tested, and even 100% on the set of the shortest patterns.

We repeated the same tests on another computer having two 2.0GHz AMD Opteron DP 246 processors each having 6KB L1 cache and 1MB L2 cache. Size of main memory is 6GB (400MHz DDR PC3200). Unfortunately there is a unusually large variation in timings on that computer so that it was difficult to get reliable results. On tests with the DNA data, BMH4 was good for short patterns. KS was good for patterns longer than 15. BMH4c was the fastest for $m \leq 22$. On amino acids BMH2c was superior for all tried pattern sets. For most algorithms, the relative speed improvement was clearly smaller than in the tests reported above, when patterns got longer.

Although FAOSO was fast for short patterns, it is rather unpractical. Namely it has two constant parameters and it is a tedious process to find out the best combination of them for each type of input. Without varying both of them, one will likely get slower run times.

5 Conclusion

We demonstrated that it is possible to speed up q-gram calculation by reading several characters at the same time. According to the comparison, the new variations BMH4c and BMH2c were among the fastest ones for DNA and amino acid data, respectively. Our intention is to continue working on multiple reading. We expect that multiple reading would speed up also other algorithms handling continuous q-grams, like BR, LEC, and SBNDM2. KS is promising for searching of long DNA patterns on modern processors while using an up-to-date compiler.

We showed how to fix the inefficiency of the Berry–Ravindran algorithm in the case of DNA data. This modification is advantageous also for amino acid data.

We noticed that the results depend more on the processor and compiler than we expected. Algorithm $\mathcal{A}$ may be faster than algorithm $\mathcal{B}$ on computer $\mathcal{C}$ and the other way around on computer $\mathcal{D}$. For more profound practical evaluation of algorithms, several environments should be tried.

References

1. R. Baeza-Yates: Improved string searching. *Software: Practice and Experience*, **19** (3):257–271, 1989.

2. T. Berry and S. Ravindran: A fast string matching algorithm and experimental results. *Proc. of the Prague Stringology Club Workshop '99*, Czech Technical University, Prague, Czech Republic, Collaborative Report DC-99-05, pp. 16–28, 1999.
3. R.S. Boyer and J S. Moore: A fast string searching algorithm. *Communications of the ACM*, **20**(10):762–772, 1977.
4. K. Fredriksson and Sz. Grabowski: Practical and optimal string matching. In *Proc SPIRE'05, Lecture Notes in Computer Science* **3772**:376–387, 2005.
5. K. Fredriksson: Personal communication.
6. J. Holub and B. Ďurian: Fast variants of bit parallel approach to suffix automata. *(Unpublished Lecture) University of Haifa.* 2005-04-05.
7. R.N. Horspool: Practical fast searching in strings. *Software: Practice and Experience*, **10**(6):501–506, 1980.
8. A. Hume and D. Sunday: Fast string searching. *Software: Practice and Experience*, **21**(11):1221–1248, 1991.
9. H. Hyyrö: Personal communication.
10. J.Y. Kim and J. Shawe-Taylor: Fast string matching using an n-gram algorithm. *Software: Practice and Experience*, **24**(1):79–88, 1994.
11. J.W. Kim, E. Kim, and K. Park: Fast matching method for DNA sequences. In *Proc ESCAPE 2007, Lecture Notes in Computer Science* **4614**:271–281, 2007.
12. D.E. Knuth, J.H. Morris, and V.R. Pratt: Fast pattern matching in strings. *SIAM Journal on Computing*, **6**(1): 323–350, 1977.
13. T. Lecroq: Fast exact string matching algorithms. *Information Processing Letters*, **102**(6): 229–235, 2007.
14. G. Navarro and M. Raffinot: Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM Journal of Experimental Algorithms*, **5**(4):1–36, 2000.
15. H. Peltola and J. Tarhio: Alternative algorithms for bit-parallel string matching. In *Proc SPIRE'03, Lecture Notes in Computer Science* **2857**:80–93, 2003.
16. S.S. Sheik, S.K. Aggarwal, A. Poddar, N. Balakrishnan, and K. Sekar: A FAST pattern matching algorithm. *J. Chem. Inf. Comput. Sci.*, **44**(4):1251–1256, 2004.
17. D.M. Sunday: A very fast substring search algorithm. *Communications of the ACM*, **33**(8):132–142, 1990.
18. J. Tarhio and H. Peltola: String matching in the DNA alphabet. *Software: Practice and Experience*, **27**(7):851–861, 1997.
19. R. Thathoo, A. Virmani, S.Sai Lakshmi, N. Balakrishnan, and K. Sekar: TVSBS: A fast exact pattern matching algorithm for biological sequences. *Current Science* **91**(1):47–53, 2006.
20. S. Wu and U. Manber: A fast algorithm for multi-pattern searching, Report TR-94-17, Department of Computer Science, University of Arizona, Tucson, AZ, 1994.
21. R.F. Zhu and T. Takaoka: On improving the average case of the Boyer–Moore string matching algorithm. *Journal of Information Processing*, **10**(3):173–177, 1987.