Approximate String Matching with Reduced Alphabet*

Leena Salmela[1] and Jorma Tarhio[2]

[1] University of Helsinki, Department of Computer Science
leena.salmela@cs.helsinki.fi
[2] Aalto University
Deptartment of Computer Science and Engineering
jorma.tarhio@tkk.fi

Abstract. We present a method to speed up approximate string matching by mapping the factual alphabet to a smaller alphabet. We apply the alphabet reduction scheme to a tuned version of the approximate Boyer–Moore algorithm utilizing the Four-Russians technique. Our experiments show that the alphabet reduction makes the algorithm faster. Especially in the k-mismatch case, the new variation is faster than earlier algorithms for English data with small values of k.

1 Introduction

The approximate string matching problem is defined as follows. We have a pattern $P[1...m]$ of m characters drawn from an alphabet Σ of size σ, a text $T[1...n]$ of n characters over the same alphabet, and an integer k. We need to find all such positions i of the text that the distance between the pattern and a substring of the text ending at that position is at most k. In the k-difference problem the distance between two strings is the standard edit distance where substitutions, deletions, and insertions are allowed. The k-mismatch problem is a more restricted one using the Hamming distance where only substitutions are allowed.

Among the most cited papers on approximate string matching are the classical articles [1,2] by Esko Ukkonen. Besides them he has studied this topic extensively [3,4,5,6,7,8,9,10,11]. In this paper we present a practical improvement for Esko Ukkonen's approximate Boyer–Moore algorithm (ABM) [7] developed together with J. Tarhio. The ABM algorithm is based on Horspool's variation [12] of the Boyer–Moore algorithm [13]. ABM has variations both for the k-difference problem and for the k-mismatch problem. The k-difference variation is a filtration method. Recently Salmela et al. [14] introduced a tuned version of ABM for small alphabets. Here we consider an alphabet reduction technique which makes the tuned ABM more practical in the case of large alphabets. Our approach reduces the preprocessing time and the space usage of the algorithm. Our experiments show that the new variation is faster than the original ABM. Moreover,

* Supported by Academy of Finland grants 118653 (ALGODAN) and 134287.

the new variation is faster than earlier mismatch algorithms for English data with small values of k.

The rest of the paper is organized as follows. We start with a short review on alphabet transformations in string matching. After that we review earlier versions of ABM. Then we explain our alphabet transformations in detail. Before conclusions we review results of our experiments.

2 Alphabet Transformations in String Matching

Alphabet transformation is a widely used method to increase the efficiency of string matching. There are several types of alphabet transformations. One of the most common transformations is hashing [15]. One can check in a hash table, whether a window of the text possibly equals the pattern. By selecting a suitable hash function, one can control the probability of false matches [15]. In case of multiple patterns, one can apply binary search [16] or two-level hashing [17] for checking candidate matches.

In algorithms of Boyer–Moore type it is common to use q-grams, i.e. substrings of q characters, instead of single characters in shift calculation. This technique was already mentioned in the classical paper by Boyer and Moore [13]. The aim is to increase the size of effective alphabet, which leads to longer shifts especially in the case of small alphabets. The approach extends to multidimensional [18,19] and parameterized matching [20]. In many algorithms [21,22], q-grams and hashing occur together in shift calculation.

It is common to use q-grams instead of single characters also for other purposes than shifting. The aim is to increase practical scanning speed [23,24,25] or to improve selectivity [3,4,26]. Grams are not always continuous, but they may be gapped [27] or equidistant [28].

Still another type is relaxed preprocessing with a reduced alphabet [29,30]. In approximate string matching this extends the applicability of the Four-Russians technique [31,32], which is used to precompute edit distances between arbitrary q-grams and the q-grams of a pattern. With a reduced alphabet one can apply a larger q without extra space and preprocessing time. In Section 4, we will consider an application of transformations of this type in detail.

3 Tuned Version of ABM

In this section we will describe a tuned version of the ABM algorithm [7]. In the next section we will then use this algorithm to illustrate how to apply an alphabet reduction technique to speed up an approximate string matching algorithm that uses q-grams.

As preprocessing the ABM algorithm computes the shifts for each character of the alphabet as in the Boyer–Moore–Horspool algorithm [12]. During searching the shift is then computed by considering last $k + 1$ characters of the current window. The shift is the minimum of the precomputed shifts for each of those

$k+1$ characters. After shifting, at least one of these characters will be aligned correctly with the pattern or the pattern is shifted past the first one of these characters.

Liu et al. [33] tuned the k-mismatch version of ABM for smaller alphabets. Their algorithm, called FAAST, uses a stronger shift function based on a variation of the Four-Russians technique [31,32] to speed up the search. Instead of minimizing $k+1$ shifts during search, it uses a precomputed shift table for a q-gram aligned with the end of the pattern, where $q \geq k+1$ is a parameter of the algorithm. (The original paper used the notation $(k+x)$-gram.) The shift table is calculated so that after the shift at least $q-k$ characters are aligned correctly or the window is shifted past the last q-gram of the previous window.

Salmela et al. [14] further refined the FAAST algorithm and adapted it also to the k-difference problem. Their algorithm stores the number of substitutions or differences for aligning each q-gram with the end of the pattern and uses this precomputed value in the searching phase instead of recomputing it. We will use Algorithms 1 and 2 from [14] as a basis for our algorithm with alphabet reduction. The first one is an algorithm for the k-mismatch problem and the second one solves the k-difference problem. The pseudo code of these algorithms is shown as Algorithm 1. The preprocessing of these algorithms takes $O(m\sigma^q)$ time[3] and the average complexity of searching is $O(n(\log_\sigma m+k)/m)$ when $q=\Theta(\log_\sigma m+k)$. In the average complexity of searching and when computing the value of q, σ should be replaced by $1/p$, where p is the probability of two random characters matching, if the alphabet is not uniform. The space complexities of the mismatch version and the difference version of the algorithm are $O(m\sigma^q+mq)=O(m\sigma^q)$ and $O(m\sigma^q+mq+m^2)=O(m\sigma^q)$, respectively. We see now that the naive approach of using an alphabet of size 256 for English text is not feasible as the space (and preprocessing) requirement of the algorithm grows exponentially when q is increased. Even if we map each character to a unique integer, the alphabet size is too large to be practical for larger values of q.

4 Algorithm with Alphabet Reduction

We are now ready to present an alphabet reduction scheme for approximate string matching. The scheme can be applied to any algorithm using q-grams. As an example, we apply it to the tuned version of the ABM algorithm presented in the previous section. A similar alphabet reduction scheme has been earlier presented by Fredriksson and Navarro [30], but their alphabet mapping is different from ours.

We will first present the algorithm with alphabet reduction assuming we have a mapping function $g : \Sigma \mapsto \hat{\Sigma}$ which maps each character of the alphabet to a character in the reduced alphabet $\hat{\Sigma}$ of size $\hat{\sigma}$. We first note that if a pattern has an (approximate) occurrence in a text, then the pattern that is mapped to the reduced alphabet has the same (approximate) occurrence in a text that is

[3] See [14] for details on how to implement the preprocessing phase to reach this bound.

Algorithm 1 Search for $P[1...m]$ in $T[1...n]$ with at most k errors

```
1: Preprocessing:
2: for all G ∈ Σ^q do
3:    for i ← 1 to m do
4:       D[i] ← the minimum number of errors for aligning G with P[1...i] when
                deletions in the beginning of either G or P[1...i] are free
5:    M[G] ← D[m]
6:    D_s[G] ← m − max{i | i < m and D[i] ≤ k}
7: Searching:
8: j ← m − k {j ← m for the mismatch version of the algorithm}
9: while j ≤ n do
10:   G ← T[j − q + 1...j]
11:   if M[G] ≤ k then
12:      verify the potential match
13:   j ← j + D_s[G]
```

also mapped to the reduced alphabet. However, the mapped pattern might also have additional (approximate) occurrences in the mapped text.

Instead of mapping the whole text to the reduced alphabet, we will use the following method which only maps the needed q-grams of the text to the reduced alphabet. The preprocessing phase will now operate with the reduced alphabet. That is, we map each character of the pattern to the reduced alphabet and compute the arrays M and D_s for all q-grams in the reduced alphabet. The searching phase uses the same mapping of the q-grams of the text when accessing the arrays M and D_s but the verification of a potential match is performed with the original alphabet. The time complexity of the preprocessing phase is reduced to $O(m\hat{\sigma}^q)$ and the average complexity of searching becomes $O(n(\log_{\hat{\sigma}} m+k)/m)$ when $q = \Theta(\log_{\hat{\sigma}} m+k)$. The space complexities of the mismatch and difference versions of the algorithm are also reduced to $O(m\hat{\sigma}^q + mq) = O(m\hat{\sigma}^q)$ and $O(m\hat{\sigma}^q + mq + m^2) = O(m\hat{\sigma}^q)$, respectively. We now see that if we map the English alphabet for example to a reduced alphabet of size 8, using much larger values of q becomes feasible.

We notice that the average complexity of searching in the reduced alphabet scheme is theoretically larger than in the plain algorithm. However, our experiments show that in practise searching is faster in the reduced alphabet scheme. First we note that if the alphabet is nonuniform, some characters may be very rare and without alphabet reduction these characters increase the q-gram space unnecessarily as they are rarely accessed and thus do not improve filtering noticeably. Another issue is that q must be an integer and therefore we might have to make compromises when choosing the value of q as the optimal q is $c(\log_{\sigma} m+k)$ for some constant c and this optimal q might not be an integer. This problem is emphasized when the alphabet is large as there are fewer feasible choices for the value of q. Furthermore if k is large, even choosing $q = k+1$, which is the minimum possible value for q in the algorithm, might be infeasible with the original

alphabet and then reducing the alphabet size can make it feasible to use a large enough q.

The remaining problem is to find the mapping function g given the size of the reduced alphabet. The mapping function should minimize the probability that two random characters match. This probability is minimized by a mapping that produces the most uniform reduced alphabet [30]. Fredriksson and Navarro [30][4] have earlier used the following method to find this mapping. They first sort the characters in ascending order of frequency. The i:th character in this order is then mapped to the ($i \bmod \hat{\sigma}$):th character in the reduced alphabet.

The problem of finding the mapping is defined formally as follows. We are given the frequency f_c of each character $c \in \Sigma$ and an integer $\hat{\sigma}$ that defines the size of the reduced alphabet. Our task is now to partition the characters in Σ into $\hat{\sigma}$ subsets S_i such that the following objective function is minimized:

$$\max_{i \in [1,\hat{\sigma}]} \left\{ \sum_{c \in S_i} f_c \right\} - \min_{i \in [1,\hat{\sigma}]} \left\{ \sum_{c \in S_i} f_c \right\}.$$

Each of the subsets S_i is then mapped to a unique character in $\hat{\Sigma}$. This formulation is equivalent to the $\hat{\sigma}$-way number partitioning problem.

The number partitioning problem has been shown to be NP-complete [34] and thus we resort to the following well known greedy algorithm to find the mapping function. We first sort the characters in decreasing order of frequency. Starting from the most frequent character we map that character to the least frequent character of the reduced alphabet.

We considered the following schemes for reducing the alphabet:

Pattern Alphabet The reduced alphabet consists of all characters that occur in the pattern and one extra character [29]. All characters that do not occur in the pattern are mapped to this extra character.

Reduced Alphabet We compute the reduced alphabet using the greedy algorithm outlined above.

Reduced Pattern Alphabet We combine the two above methods. We first form an alphabet consisting of the characters occurring in the pattern and an extra character as in the pattern alphabet method. Then we use the reduced alphabet method to reduce this alphabet.

We also tried first classifying the characters into separators and letters or separators, vowels, and consonants, and then applying alphabet reduction to these groups separately but this approach was not competitive.

5 Experimental Results

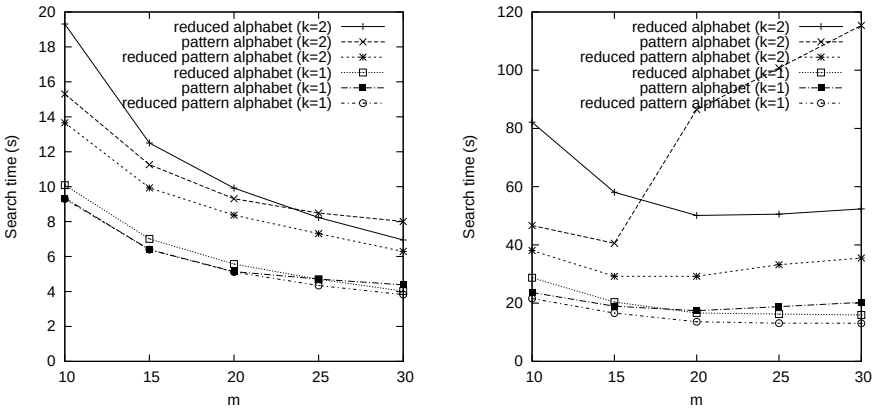
Experiments were run on an Intel 3.16 GHz dual core CPU with 3.7 GB of memory and 32 kB L1 cache and 6144 kB L2 cache. The computer was running Linux 2.6.27. The algorithms were written in C and compiled with the gcc

[4] This method is not outlined in the paper but can be found in the corresponding code.

4.3.2 compiler producing 32-bit code. We used the 50 MB English text from the *PizzaChili* site, `http://pizzachili.dcc.uchile.cl`. Each pattern set consists of 200 different patterns of the same length. The patterns are randomly chosen from the text and a substitution, insertion, or deletion is introduced at every position with probability 0.05. As an example, the 200 patterns of length 20 have 261 total matches when searching allowing one substitution and a total of 680 matches when allowing one substitution, deletion, or insertion.

5.1 Comparison of Alphabet Reduction Techniques

The value of q was varied from 2 to 7, and we tried reduced alphabet sizes of 4, 8, 16, and 32. We show the results for the best observed values of these parameters. Figure 1 shows the searching times excluding the preprocessing time for the k-mismatch and k-difference problems for $k = 1$ and $k = 2$. Table 1 shows the best parameter values for each of the methods. As can be seen, the best method in all cases is the reduced pattern alphabet method.



(a) k-mismatch problem (b) k-difference problem

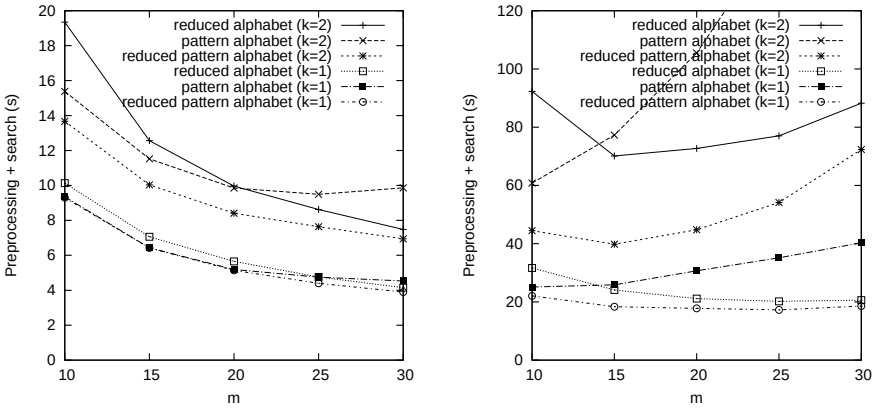
Fig. 1. Search times

Figure 2 shows the combined preprocessing and search times. Again we see that the reduced pattern alphabet method is the best. Furthermore, comparing Figures 1(a) and 2(a) we note that in the k-mismatch problem the preprocessing time is much smaller than the searching time. In the k-difference problem, especially the pattern alphabet method has a high preprocessing cost but also the other methods have a moderately high preprocessing cost when $k = 2$.

The increase in the preprocessing time is due to using a larger value of q to speed up searching. Figure 3 shows how the preprocessing time grows when q is increased in the k-difference algorithms when $m = 10$ and $k = 1$.

Table 1. The parameters yielding the best search time for each of the methods and the corresponding number of different q-grams in the reduced alphabet. If different values were best for different pattern lengths, the alternative values are shown on subsequent rows. To compute the number of different q-grams in the pattern alphabet method, we use the average reduced alphabet size for pattern length 30 which was 17.21. Note that the reduced pattern alphabet method is equivalent to the pattern alphabet method for the 1-mismatch problem because the size of the reduced alphabet is larger than the length of the pattern.

Algorithm	1-mismatch		2-mismatch		1-difference		2-difference	
	Param.	$\hat{\sigma}^q$	Param.	$\hat{\sigma}^q$	Param.	$\hat{\sigma}^q$	Param.	$\hat{\sigma}^q$
Pattern Alphabet	$q = 3$	$< 2^{13}$	$q = 4$	$< 2^{17}$	$q = 5$	$< 2^{21}$	$q = 6$	$< 2^{25}$
							$q = 5$	$< 2^{21}$
Reduced Alphabet	$\hat{\sigma} = 16, q = 3$	2^{12}	$\hat{\sigma} = 4, q = 6$	2^{12}	$\hat{\sigma} = 8, q = 6$	2^{18}	$\hat{\sigma} = 8, q = 7$	2^{21}
	$\hat{\sigma} = 8, q = 4$	2^{12}	$\hat{\sigma} = 8, q = 5$	2^{15}	$\hat{\sigma} = 4, q = 8$	2^{16}		
Reduced Pattern	$\hat{\sigma} = 32, q = 3$	2^{15}	$\hat{\sigma} = 8, q = 4$	2^{12}	$\hat{\sigma} = 8, q = 6$	2^{18}	$\hat{\sigma} = 8, q = 7$	2^{21}
Alphabet			$\hat{\sigma} = 8, q = 5$	2^{15}				



(a) k-mismatch problem (b) k-difference problem

Fig. 2. Combined preprocessing and search times for a text of length 50 MB

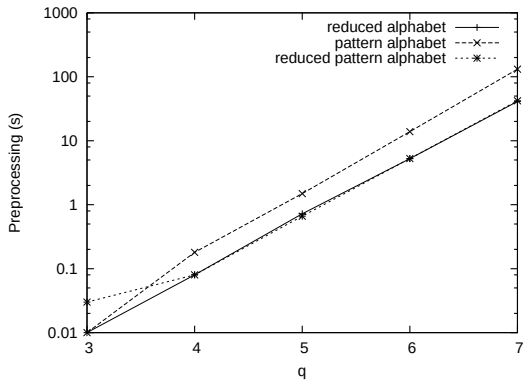


Fig. 3. Preprocessing time of the k-difference algorithm for varying values of q when $m = 10$ and $k = 1$. Reduced alphabet and reduced pattern alphabet methods use reduced alphabet size 8.

We also ran similar experiments with the 50 MB protein text from the *PizzaChili* site, `http://pizzachili.dcc.uchile.cl`. The results with protein data were very similar to our results with English text.

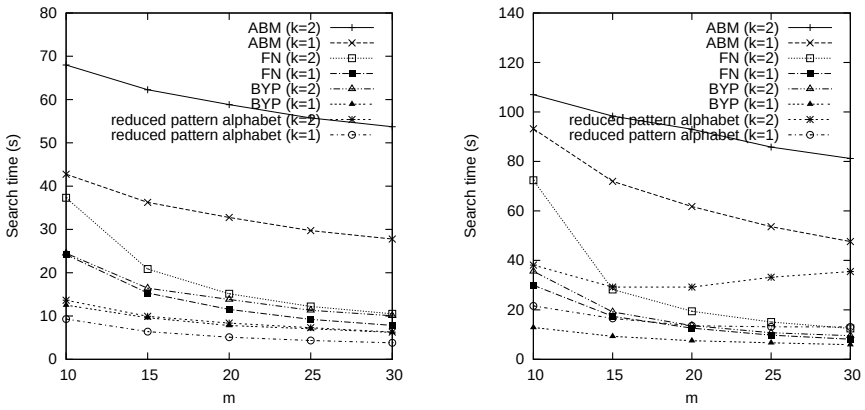
5.2 Comparison with Other Algorithms

We compared the performance of the best alphabet reduction scheme, reduced pattern alphabet, with the following algorithms:

- ABM: The original ABM algorithm.
- BYP: The algorithm by Baeza-Yates and Perleberg [35] divides the pattern into smaller pieces so that if the pattern occurs at some position, at least one of the pieces must have an exact occurrence at that position. The algorithm then searches for exact matches of the pieces and verifies the occurrences found by the exact search.
- FN: The algorithm by Fredriksson and Navarro [30] reads non-overlapping q-grams (ℓ-grams in the original paper) in an alignment and with the help of preprocessed tables determines the minimum number of substitutions or differences for aligning the q-grams with the pattern in some way. When the minimum number of substitutions or differences exceeds k, the pattern is shifted so that the first of these q-grams is no longer aligned with the pattern. The potential matches must be verified. To make the comparison fair, we modified their algorithm so that it uses the reduced pattern alphabet method, which improved its performance although the improvement was not as clear as in our algorithm.

Figure 4 shows the searching times of the algorithms. As can be seen, the alphabet reduction technique combined with the use of q-grams makes the new

algorithm significantly faster than the plain ABM algorithm. In the k-mismatch case, our new algorithm is the fastest, BYP being the second fastest, FN third, and ABM clearly the slowest. In the k-difference case BYP takes the lead which was also the case in the experiments by Fredriksson and Navarro [30]. Overall FN is the second best although our algorithm is faster for short patterns which are important in practise. Again ABM is clearly the slowest.



(a) k-mismatch problem (b) k-difference problem

Fig. 4. Comparison of the best alphabet reduction scheme, reduced pattern alphabet, ABM, BYP, and FN

6 Conclusions

We have presented an alphabet reduction technique to speed up algorithms for approximate string matching. We applied the technique to a Boyer–Moore style algorithm which uses the Four-Russians technique to compute shifts with small alphabets. When improved with alphabet reduction, the algorithm performs surprisingly well on large alphabets too. The space usage of the Four-Russians approach used in the algorithm is not feasible if the whole large alphabet is used but becomes practical with alphabet reduction. Our experiments on English data show that the algorithm with alphabet reduction is the fastest algorithm in the k-mismatch problem for small values of k.

Acknowledgements We thank the referee, who helped us to improve this paper.

References

1. Ukkonen, E.: Algorithms for approximate string matching. Information and Control **64**(1-3) (1985) 100–118
2. Ukkonen, E.: Finding approximate patterns in strings. J. Algorithms **6**(1) (1985) 132–137
3. Jokinen, P., Ukkonen, E.: Two algorithms for approximate string matching in static texts. In: Proc. 16th Symposium on Mathematical Foundations of Computer Science. Vol. 520 of LNCS, Berlin, Springer (1991) 240–248
4. Ukkonen, E.: Approximate string matching with q-grams and maximal matches. Theor. Comput. Sci. **92**(1) (1992) 191–211
5. Ukkonen, E.: Approximate string-matching over suffix trees. In: Proc. 4th Symposium on Combinatorial Pattern Matching. Vol. 684 of LNCS, Berlin, Springer (1993) 228–242
6. Ukkonen, E., Wood, D.: Approximate string matching with suffix automata. Algorithmica **10**(5) (1993) 353–364
7. Tarhio, J., Ukkonen, E.: Approximate Boyer–Moore string matching. SIAM J. Comput. **22**(2) (1993) 243–260
8. Jokinen, P., Tarhio, J., Ukkonen, E.: A comparison of approximate string matching algorithms. Software–Pract. Exp. **26**(12) (1996) 1439–1458
9. Fredriksson, K., Navarro, G., Ukkonen, E.: Optimal exact and fast approximate two dimensional pattern matching allowing rotations. In: Proc. 13th Symposium on Combinatorial Pattern Matching. Vol. 2373 of LNCS, Berlin, Springer (2002) 235–248
10. Kärkkäinen, J., Navarro, G., Ukkonen, E.: Approximate string matching on Ziv–Lempel compressed text. J. Discrete Algorithms **1**(3-4) (2003) 313–338
11. Mäkinen, V., Ukkonen, E., Navarro, G.: Approximate matching of run-length compressed strings. Algorithmica **35**(4) (2003) 347–369
12. Horspool, R.N.: Practical fast searching in strings. Software–Pract. Exp. **10**(6) (1980) 501–506
13. Boyer, R.S., Moore, J.S.: A fast string searching algorithm. Commun. ACM **20**(10) (1977) 762–772
14. Salmela, L., Tarhio, J., Kalsi, P.: Approximate Boyer–Moore string matching for small alphabets. Algorithmica (In press, online version available)
15. Karp, R.M., Rabin, M.O.: Efficient randomized pattern-matching algorithms. IBM J. Research and Development **31**(2) (1987) 249–260
16. Zhu, R., Takaoka, T.: A technique for two-dimensional pattern matching. Commun. ACM **32**(9) (1989) 1110–1120
17. Muth, R., Manber, U.: Approximate multiple strings search. In: Proc. 7th Symposium on Combinatorial Pattern Matching. Vol. 1075 of LNCS, Berlin, Springer (1996) 75–86
18. Kärkkäinen, J., Ukkonen, E.: Two- and higher-dimensional pattern matching in optimal expected time. SIAM J. Comput. **29**(2) (1999) 571–589
19. Tarhio, J.: A sublinear algorithm for two-dimensional string matching. Pattern Recogn. Lett. **17**(8) (1996) 833–838
20. Salmela, L., Tarhio, J.: Fast parameterized matching with q-grams. J. Discrete Algorithms **6**(3) (2008) 408–419
21. Lecroq, T.: Fast exact string matching algorithms. Inf. Process. Lett. **102**(6) (2007) 229–235

22. Wu, S., Manber, U.: A fast algorithm for multi-pattern searching. Technical report, Dept. of Computer Science, U. of Arizona (1994)
23. Kim, S.: A new string-pattern matching algorithm using partitioning and hashing efficiently. J. Exp. Algorithmics **4**(2) (1999)
24. Fredriksson, K.: Shift-or string matching with super-alphabets. Inf. Process. Lett. **87**(4) (2003) 201–204
25. Ďurian, B., Holub, J., Peltola, H., Tarhio, J.: Tuning BNDM with q-grams. In: Proc. ALENEX '09, SIAM (2009) 29–37
26. Salmela, L., Tarhio, J., Kytöjoki, J.: Multipattern string matching with q-grams. J. Exp. Algorithmics **11** (2006)
27. Fontaine, M., Burkhardt, S., Kärkkäinen, J.: BDD-based analysis of gapped q-gram filters. Int. J. Found. Comput. Sci. **16**(6) (2005) 1121–1134
28. Fredriksson, K., Grabowski, S.: Practical and optimal string matching. In: Proc. 12th International Conference on String Processing and Information Retrieval. Vol. 3772 of LNCS, Berlin, Springer (2005) 376–387
29. Berry, T., Ravindran, S.: Tuning the Zhu–Takaoka string matching algorithm and experimental results. Kybernetika **38**(1) (2002) 67–80
30. Fredriksson, K., Navarro, G.: Average-optimal single and multiple approximate string matching. J. Exp. Algorithmics **9** (2004)
31. Arlazarov, V.L., Dinic, E.A., Kronrod, M.A., Faradzev, I.A.: On economic construction of the transitive closure of a directed graph. Doklady Academi Nauk SSSR **194** (1970) 487–488 (in Russian). English translation in Soviet Mathematics Doklady **11** (1975) 1209–1210
32. Masek, W.J., Paterson, M.S.: A faster algorithm for computing string edit distances. J. Comput. Syst. Sci. **20**(1) (1980) 18–31
33. Liu, Z., Chen, X., Borneman, J., Jiang, T.: A fast algorithm for approximate string matching on gene sequences. In: Proc. 16th Symposium on Combinatorial Pattern Matching. Vol. 3537 of LNCS, Berlin, Springer (2005) 79–90
34. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. Freeman (1979)
35. Baeza-Yates, R., Perleberg, C.: Fast and practical approximate string matching. Inf. Process. Lett. **59**(1) (1996) 21–27