Algorithms for Weighted Matching

Leena Salmela and Jorma Tarhio[⋆]

Helsinki University of Technology
{lsalmela, tarhio}@cs.hut.fi

Abstract. We consider the matching of weighted patterns against an unweighted text. We adapt the shift-add algorithm for this problem. We also present an algorithm that enumerates all strings that produce a score higher than a given score threshold when aligned against a weighted pattern and then searches for all these strings using a standard exact multipattern algorithm. We show that both of these approaches are faster than previous algorithms on patterns of moderate length and high significance levels while the good performance of the shift-add algorithm continues with lower significance levels.

1 Introduction

In a weighted matching problem the text or the pattern is a weighted sequence where in each position a weight is assigned to each character of the alphabet. In this paper we consider the case where the pattern is weighted and the text unweighted and we are interested in finding alignments where the score, which is the sum of the weights in the pattern corresponding to the aligned characters in the text, is larger than some given score threshold.

Weighted patterns arise for example in the modeling of transcription factor binding sites in bioinformatics. In bioinformatics weighted patterns are called position weight matrices, position specific scoring matrices or profiles. The weight of a nucleotide in a given position describes the log probability of that nucleotide appearing in that position in a transcription factor binding site. Therefore the score of an alignment is the log probability of that alignment being a transcription factor binding site. Many methods in bioinformatics rely on the large scale scanning of these weighted patterns against a genome and there are large public databases, like TRANSFAC [5] containing such patterns.

In this paper we adapt some standard string matching techniques to the weighted matching problem and compare the performance of these algorithms against the algorithm by Liefooghe et al. [4]. In Section 4, we adapt the shift-add [1] algorithm to handle weighted patterns and in Section 5 we consider the enumeration of all strings matching a given weighted pattern and searching for these strings by a standard multipattern algorithm. We compare our new approaches to the previous algorithm by Liefooghe et al. [4] in Section 6. The preliminary experimental results show that for high significance levels the enumeration approach is the fastest for pattern lengths 7 to 19 while the shift-add algorithm is

[⋆] Work by Jorma Tarhio was supported by Academy of Finland.

i	1	2	3	4	5	6	7	8	9	10	11	12
a	7	−6	−5	−10	−8	−10	4	−10	−10	−2	−10	−10
c	−5	−8	−10	14	−10	−8	−10	−10	−10	11	−10	−10
t	6	13	−10	−8	−10	12	−10	−10	−10	−3	−10	9
g	−5	−6	13	−10	14	−1	11	14	14	−10	14	6

Fig. 1. An example weighted pattern corresponding to the EGR-1 family extracted from TRANSFAC.

the fastest for shorter and longer patterns. For the longest patterns either the algorithm by Liefooghe et al. or the shift-add algorithm is the fastest. For lower significance levels the shift-add algorithm is the fastest.

After submitting this paper we learned that Pizzi et al. [7] have also developed an algorithm based on the enumeration approach. However, they use a different multipattern algorithm to search for the enumerated strings while we use an algorithm tuned for very large pattern sets and low expected number of hits.

2 Definitions

We consider the matching of weighted patterns against an unweighted text. The text is a sequence of characters from an alphabet Σ of size σ. The weighted pattern assigns weights to all characters of the alphabet for each position of the pattern.

Definition 1. *A weighted pattern of length m is an $m \times \sigma$ matrix p of integer coefficients $p[i, c]$ which give the weight of the character $c \in \Sigma$ at position i where $1 \leq i \leq m$.*

Figure 1 shows an example of a weighted pattern. Here we will only consider weighted patterns with integer weights. Weighted patterns are obtained from entropy or log odd matrices that have real coefficients but in practice these are rounded to integer matrices to allow for more efficient computation.

Given a weighted pattern and a string of characters from the alphabet Σ the score of this string is defined as follows:

Definition 2. *Given a weighted pattern p of length m and a string t of length m from the alphabet Σ, the score of the pattern aligned with the string is defined as:*

$$\mathrm{score}(p, t) = \sum_{i=1}^{m} p[i, t_i]$$

In the weighted matching problem we are interested in finding all those alignments of a text with the pattern that yield a large enough score:

Definition 3. *Given a weighted pattern p of length m, a score threshold α and an unweighted text $t_{1\ldots n}$, find all such alignments i of the pattern with the text that* $\mathrm{score}(p, t_{i\ldots i+m-1}) \geq \alpha$.

Given a weighted matching problem, p-value [2,10] is a measure that can be used to estimate the statistical significance of the returned alignments. The p-value is defined as follows:

Definition 4. *Given a weighted matching problem with pattern p and score threshold α, $p\text{-value}(p, \alpha)$ is the probability that a given background model of the sequence produces a score equal to or greater than the score threshold α.*

In this paper we assume that the background model is the standard random string model where each character of the sequence is chosen independently and uniformly. In this case the p-value can be computed with the following recursion:

$$p\text{-value}(p[1...0], \alpha) = \begin{cases} 1 \text{ if } \alpha \leq 0 \\ 0 \text{ otherwise} \end{cases}$$
$$p\text{-value}(p[1...i], \alpha) = \frac{1}{\sigma} \sum_{c \in \Sigma} p\text{-value}(p[1...i-1], \alpha - p[i, c])$$

3 Previous Work

The brute force algorithm for the weighted matching problem calculates the score for each alignment of the pattern with the text and reports those alignments that yield a score higher than the score threshold. Lately various techniques have been proposed to speed up this scheme. Here we will review those techniques that are relevant to our work. See [8] for a survey on previous work.

Several algorithms use the lookahead technique [11] which provides a way to prune the calculation in a single alignment. For all suffixes of the pattern, there is a maximum score that they can contribute to the overall score. If after matching the prefix of the pattern, the score is not at least the score threshold minus maximum score of the suffix, there cannot be a match at this alignment. By calculating the maximum score for each pattern suffix, the overall computation time can be significantly reduced.

In Section 6 we will compare our algorithms to the algorithm by Liefooghe et al. [4]. Their algorithm uses the lookahead technique and in addition it divides the pattern into submatrices and precalculates for all possible strings the score yielded by each submatrix. For example, if we had a pattern of length 12, we could divide it to three submatrices of length four and then precalculate the scores of each submatrix for all the σ^4 possible strings. At matching time we can then just lookup the scores of each submatrix in a table.

4 Shift-Add for Weighted Matching

In this section we will adapt the shift-add algorithm [1] to weighted matching. Originally the shift-add algorithm was designed for the k-mismatch problem where the task is to find all substrings of the text that match the pattern with at most k mismatches. The algorithm works as follows.

For each pattern position i from 1 to m the algorithm has a variable s_i indicating with how many mismatches the suffix of length i of the text read so far matches the pattern prefix of length i. If the variables s_i can be represented in b bits, we can concatenate all these variables into a single vector $s = s_m s_{m-1} \dots s_1$ of length mb. In the preprocessing phase we initialize for each symbol c in the alphabet a vector $T[c]$ where the bits in the position of s_i are 0^b if c equals p_i and $0^{b-1}1$ otherwise. The vector s (and hence also the variables s_i) can then in the matching phase be all updated at the same time when the next character c from the text is read:

$$s = (s \ll b) + T[c]$$

The algorithm has found a match if $s_m \leq k$.

If the variables s_i count mismatches, the maximum value that they can reach is m. However, in the k-mismatch problem it is enough to be able to represent values in the range $[0, k+1]$ yielding $b = \lceil \log(k+1) \rceil$. However, we need an additional bit so that the possible carry bits do not interfere with the next variable. With this modification the update operation of the algorithm becomes:

$$\begin{aligned} s &= (s \ll b) + T[c] \\ of &= (of \ll b) \mid (s \ \& \ (10^{b-1})^m) \\ s &= s \ \& \ (01^{b-1})^m \end{aligned}$$

Here the first line updates the variables s_i, the second one keeps track of those variables s_i that have overflowed and the last one clears the carry bits. When checking for a match, we now also need to check that the variable s_m has not overflowed which can be seen from the of vector. The shift-add algorithm for the k-mismatch problem has time complexity $O(n \lceil \frac{mb}{w} \rceil)$ where $b = \lceil \log(k+1) \rceil + 1$ and w is the size of the computer word in bits.

We will now present the shift-add algorithm for weighted matching with positive restricted weights. Then we will show how a general weighted pattern matching problem can be transformed into such a restricted problem. The weights of the weighted matching problem with positive restricted weights have the following properties:

1. $\forall i, 1 \leq i \leq m, \forall c \in \Sigma, \ 0 \leq p[i,c] \leq \alpha$
2. $\forall i, 1 \leq i \leq m \ \exists c \in \Sigma$ such that $p[i,c] = 0$

where p is the weighted pattern of length m and α is the score threshold. Property 1 is needed for the correct operation of the shift-add algorithm while Property 2 merely serves as a way to lower the score threshold and thus lower the number of bits needed for the variables s_i as will be seen later.

The adaptation of the shift-add algorithm to weighted matching with positive restricted weights is quite straightforward. Now instead of counting mismatches, we will be calculating scores so the variables s_i contain the score of the suffix of length i of the text read so far as compared to the prefix of length i of the pattern. For the update operation the bits corresponding to s_i in the preprocessed vectors $T[c]$ now contain the weight of the character c at position i. The update operation is exactly as in the shift-add algorithm for the k-mismatch problem. If after the update operation the score $s_m \geq \alpha$ or the variable s_m has overflowed, a match is reported.

Property 1 of the weighted matching problem with positive restricted weights states that all weights are non-negative and thus

$$\text{score}(p_{1\ldots i}, t_{j\ldots j+i+1}) \leq \text{score}(p_{1\ldots i+1}, t_{j\ldots j+i+2}) \ .$$

Because the score can only increase when reading a new character, we can truncate the score values to α. Property 1 further states that all weights are at most α. Thus, if we truncate the score values to α, after the update operation the variables $s_i \leq 2\alpha$ so 1 carry bit is enough. Therefore we need to reserve $b = \lceil \log \alpha \rceil + 1$ bits for each variable s_i and the time complexity of the algorithm is $O(n\lceil \frac{m(\lceil \log \alpha \rceil + 1)}{w} \rceil)$.

In the weighted matching problem the weights can be, and in practice often are, negative. The following observation points us to a way to transform any weighted matching problem to a weighted matching problem with positive restricted weights. Let p be a weighted pattern of length m and let p' be a weighted pattern such that for some i, $1 \leq i \leq m$, $p'[i,c] = p[i,c] + h$ for all $c \in \Sigma$ and some constant h, and for all $j \neq i$, $1 \leq j \leq m$, and all $c \in \Sigma$, $p'[j,c] = p[j,c]$. Then the following holds for the scores of p and p' aligned with any string t of length m:

$$\text{score}(p', t) = \text{score}(p, t) + h$$

Therefore the weighted pattern matching problem for a text t, pattern p and score threshold α returns exactly the same alignments as the weighted pattern matching problem for a text t, pattern p' and score threshold $\alpha' = \alpha + h$.

Now given a weighted pattern matching problem with a score threshold α and a pattern p containing any integer weights we can transform the problem into an equivalent problem with a score threshold α' and a pattern p' containing only non-negative weights.

To reduce the score threshold (and thus also the number of bits needed for the variables s_i) we further transform the pattern so that in each position at least one of the weights equals zero by adding an appropriate negative constant h to all weights in that position and by adjusting the score threshold also by h. Furthermore, if now any weight is larger than the score threshold, it can be truncated to the score threshold without affecting the returned alignments because the score of an alignment cannot get smaller as more characters are read. The scores of those alignments will however be lower. As a result we have transformed a weighted matching problem into a weighted matching problem with positive restricted weights.

In practice weighted patterns are obtained by rounding log-odd or entropy matrices to integer matrices. Thus the values of the weights depend on how much precision is preserved by this rounding and furthermore practical values of the threshold α depend on the weights. Because of the $\lceil \log \alpha \rceil + 1$ factor in the running time the shift-add algorithm is somewhat sensitive to the precision of this rounding unlike other algorithms.

```
**enumerate**($p$, $\alpha$)
1.  recurse(1, 0)

string $s$

**recurse**($i$, score)
1.  if ($\alpha >$ score + max_score($i...m$))
2.      return
3.  if ($i > m$ and score $\geq \alpha$)
4.      add_string($s$)
5.  else
6.      for each $c \in \Sigma$
7.          $s[i] = c$
8.          recurse($i + 1$, score + $p[i, c]$)
```

Fig. 2. Pseudo code for enumerating all strings that produce a score higher than or equal to the score threshold α.

5 Enumeration Algorithms

For short patterns it is possible to enumerate all matching strings which are the strings that produce a score higher than the score threshold when aligned with the weighted pattern. The enumerated strings can then be searched for with an exact multipattern matching algorithm.

The enumeration of matching strings is done with a recursive algorithm. At recursion level i we have constructed a string of length $i - 1$ that is a possible prefix of a matching string and we try to expand that prefix with all characters of the alphabet. This way we have to calculate the score of each prefix only once. The recursion can further be pruned with the lookahead technique. Suppose we have enumerated a prefix of length $i - 1$ with score score_i and the maximum score of a suffix of length $m - i$ is max_score($i...m$) then if the score threshold $\alpha > \text{score}_i + \text{max\_score}(i...m)$ then at this branch of the recursion no matching strings can be found. The pseudo code for enumerating the matching strings is given in Fig. 2.

Because the number of enumerated strings is often very large, we used the multipattern BNDM with q-grams (BG) [9] algorithm which is especially tuned for large pattern sets. The BG algorithm first builds a filter, which is a pattern of classes of characters. In this filter all characters that appear in any of the single patterns in position i are accepted at that position. The backward nondeterministic DAWG matching (BNDM) [6] algorithm is then used to scan the text with this filter. The returned alignments are verified with a Rabin-Karp [3] style algorithm. When the number of patterns grows the filtering is no longer efficient enough because almost every alignment will match the filter. To boost the filtering efficiency, the BG algorithm uses q-grams instead of single characters in the filtering phase. If matches are sufficiently rare (i.e. the p-value(p, α) is sufficiently

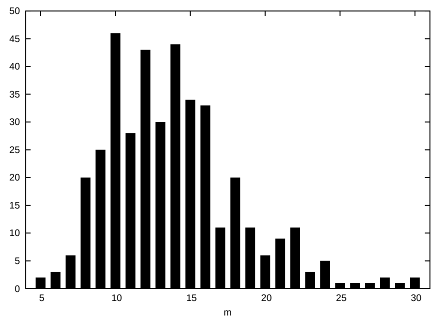

Fig. 3. The length distribution of patterns in the TRANSFAC database.

low), the BG algorithm has average case running time $O(n \log_{1/d} m/m)$ where $d = 1 - (1 - 1/\sigma^q)^r$ where r is the number of patterns.

p-value(p, α) gives the probability of a random string to produce a score equal to or greater than α when aligned with the weighted pattern p. If the background model assumes that all characters are chosen independently and uniformly, p-value(p, α) gives the proportion of all possible strings for which the score is at least α. Thus the expected number of enumerated strings is $\sigma^m p$-value(p, α) because there are σ^m different strings of length m.

In practice, it turned out to be reasonably fast to enumerate matching strings up to pattern length 16. With larger patterns we enumerated only 16 characters long prefixes of the matching strings and the algorithm verifies the found matches later.

The enumeration approach is easy to adjust to searching for multiple weighted patterns at once. All we need to do is to enumerate for all of the weighted patterns the strings producing high enough scores and then search for all these enumerated strings.

6 Experimental Results

For all experimental testing we used a computer with a 2.0 GHz AMD Opteron dual-processor and 6 GB of memory. The machine was running the 64-bit version of Linux 2.6.15. The tests were written in C and compiled with the gcc 4.1.0 compiler. The patterns were extracted from the TRANSFAC database [5]. Figure 3 shows the length distribution of the patterns. As can be seen the length of most patterns is between 8 and 22 nucleotides. In particular there are only a few patterns of length over 22 and thus the results concerning these pattern lengths are only tentative. The text we used was a chromosome from the fruitfly genome (20 MB).

Figure 4 shows a runtime comparison of the algorithm by Liefooghe, Touzet and Varré (LTV) [4], shift-add algorithm (sa) and the enumeration algorithm (ebg) for two p-values. The algorithms were run 10 times with each pattern

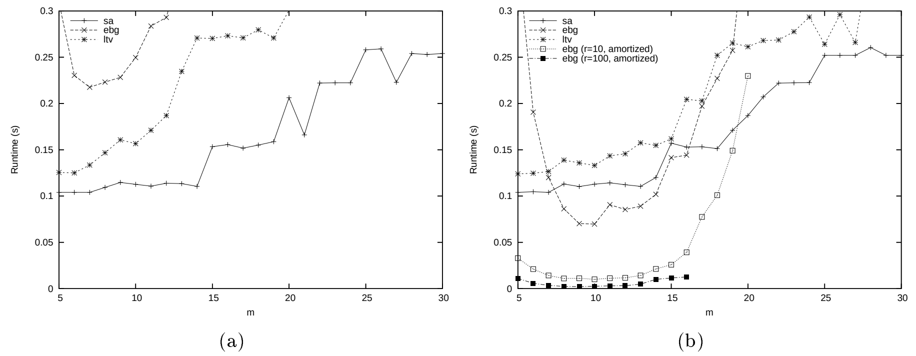

Fig. 4. Runtime comparison of different methods for p-values (a) 10^{-3} and (b) 10^{-5}.

and the average runtime was calculated. The figure shows average runtimes of patterns of same length. The measured runtime excludes the time used for preprocessing.

For the LTV algorithm we did not count the optimum length of the submatrices as presented in the original paper by Liefooghe et al. [4] because the optimum length calculation does not take into account cache effects and these surely have a significant effect on the runtime. Instead we tried the algorithm with submatrix lengths from 4 to 8 and included the best results in the comparison. With this modification the method is actually the same as the superalphabet algorithm of Pizzi et al. [7].

The optimal value for q in the LTV algorithm is lower for shorter patterns and for higher p-values but it does not affect the runtime of the algorithm very much until it reaches the value 8 when the tables no longer all fit into the cache. We can see that for the p-value 10^{-3} the runtime increases slowly until pattern length 11 and for the p-value 10^{-5} the runtime stays almost constant until pattern length 15. Until that time it is almost always sufficient to calculate the index of the first precalculated score table corresponding to the first submatrix because the lookahead technique then reports that a match at that position is not possible. When the pattern length increases further, more and more accesses are needed to the second precalculated table until at pattern length 14 for the p-value 10^{-3} and at pattern length 19 for the p-value 10^{-5} at almost every position we need to consult both the first and the second precalculated table.

Figure 4 shows that the runtime of the shift-add algorithm increases each time we need more words to represent the state vector. For pattern lengths $\{5-8, 8-14, 15-21, 19-24, 25-30\}$ we need state vectors of size $\{1, 2, 3, 4, 5\}$ words, respectively. Between lengths 19 and 21 some patterns need state vectors of 3 words while others need 4 words. Similarly for pattern length 8 some patterns need state vectors of 1 word while others need already 2 words. The number of words needed does not change from the p-value 10^{-3} to the p-value 10^{-5}.

We ran the enumeration algorithm with several different values of q and chose the value that gives the best runtime. For the p-value 10^{-3} and pattern lengths $\{5-7, 8-9, 10, 11, 12-15\}$ the values $\{4, 5, 6, 7, 8\}$, respectively, gave the best results and for the p-value 10^{-5} and pattern lengths $\{5-11, 12, 13, 14, 15-20\}$ the values $\{4, 5, 6, 7, 8\}$, respectively, gave the best results. We did not run the enumeration algorithm for longer pattern lengths because the number of enumerated patterns grew too large and already with these pattern lengths the algorithm started to significantly slow down.

Overall Fig. 4 shows that for low significance levels (i.e. high p-values) the shift-add algorithm is the fastest. For higher significance levels (i.e. smaller p-values) the shift-add algorithm is the fastest for pattern lengths smaller than 7. The enumeration algorithm is fastest for patterns lengths 8 to 16. For longer patterns the shift-add algorithm is the fastest at least until pattern length 25. After that the differences between shift-add and LTV are so small that it is hard to say anything conclusive because the TRANSFAC database contained so few long patterns.

The preprocessing of the shift-add algorithm is very fast taking less than 0.01 s regardless of the pattern length. The preprocessing time for the LTV algorithm ranges from less than 0.01 s to 0.09 s. The preprocessing time of the enumeration algorithm is exponential in the length of the pattern. It stays under 0.01 s until pattern length 12 for the p-value 10^{-3} and until pattern length 16 for the p-value 10^{-5}. For longer patterns the preprocessing time increases to 0.93 s for the p-value 10^{-3} and pattern length 15 and to 0.40 s for the p-value 10^{-5} and pattern length 20.

We also ran some experiments with the multiple pattern version of the enumeration algorithm. Because the single pattern algorithm worked well only for high significance levels we ran the multiple pattern version only for the p-value 10^{-5}. To get reliable results, we needed more patterns of each length than is provided by the TRANSFAC database. To increase the number of patterns for each pattern length we took prefixes of longer patterns and added these to our pool of patterns until we had a hundred patterns of each length. This worked up to pattern length 16 after which including prefixes of all longer patterns did not bring the number of patterns to one hundred.

Figure 5 shows how the runtime of the algorithm behaves as a function of pattern length and pattern set size r. As can be seen, the runtime decreases for all pattern sets as pattern length increases until pattern length 8 because the BG algorithm can make longer shifts. After pattern length 12 the filtering efficiency of the BG algorithm starts to deteriorate and we need to make more verifications which increases the runtime. The filtering efficiency could be boosted by increasing the value of parameter q but this would increase the amount of memory needed so that the structures frequently used by the algorithm no longer fit in the data cache and this imposes an even larger penalty on the runtime.

Figure 5b shows that the runtime increases only slightly when the pattern set size is increased for pattern lengths 8 through 14. For shorter pattern lengths the performance of the algorithm deteriorates faster because so many positions

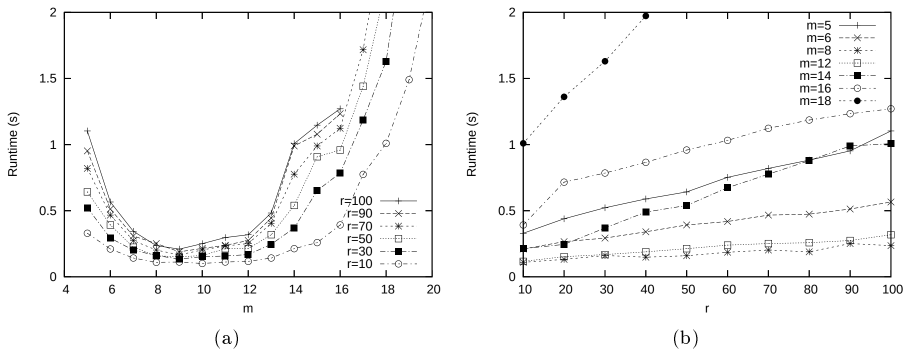

Fig. 5. The runtime of the multipattern enumeration algorithm as a function of (a) pattern length and (b) pattern set size.

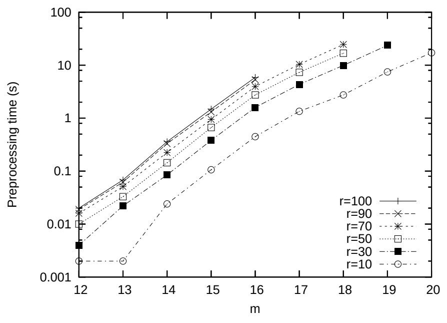

Fig. 6. Preprocessing times for the multiple pattern enumeration algorithm.

match at least one of the patterns. For longer patterns the filtering efficiency is a problem even when searching for a single pattern and this problem is further emphasized by increasing the pattern set size.

Preprocessing time of the multipattern algorithm is less than 0.01 s for all pattern set sizes when the pattern length is at most 11. Figure 6 shows the preprocessing times for longer patterns and various pattern set sizes.

The amortized running times (i.e. the running times per pattern) for the multipattern enumeration algorithm are shown also in Fig. 4b for pattern set sizes 10 and 100. As can be seen these times are much lower than the running times of the other algorithms until pattern length 16. After that the runtime starts to increase and after pattern length 20 it is probably faster to match one pattern at a time using either the shift-add or the LTV algorithm.

7 Conclusions

We have presented two efficient algorithms for searching weighted patterns in an unweighted text. We have showed that the algorithms are fast in practice by comparing their performance on real data against the previous algorithm by Liefooghe et al. [4].

References

1. Baeza-Yates, R., Gonnet, G.: A new approach to text searching. Communications of the ACM **35**(10) (1992) 74–82
2. Claverie, J.M., Audic, S.: The statistical significance of nucleotide position-weight matrix matches. Computer Applications in Biosciences **12**(5) (1996) 431–439
3. Karp, R., Rabin, M.: Efficient randomized pattern-matching algorithms. IBM Journal of Research and Development **31** (1987) 249–160
4. Liefooghe, A., Touzet, H., Varré, J.S.: Large scale matching for position weight matrices. In: Proceedings of 17th Symposium on Combinatorial Pattern Matching. Volume 4009 of LNCS, Berlin, Springer-Verlag (2006) 401–412
5. Matys, V., Fricke, E., Geffers, R., Gößling, E., Haubrock, M., Hehl, R., Hornischer, K., Karas, D., Kel, A., Kel-Margoulis, O., Kloos, D., Land, S., Lewicki-Potapov, B., Michael, H., Münch, R., Reuter, I., Rotert, S., Saxel, H., Scheer, M., Thiele, S., Wingender, E.: TRANSFAC: transcriptional regulation, from patterns to profiles. Nucleic Acids Res. **31** (2003) 374–378
6. Navarro, G., Raffinot, M.: Fast and flexible string matching by combining bit-parallelism and suffix automata. ACM Journal of Experimental Algorithmics **5**(4) (2000) 1–36
7. Pizzi, C., Rastas, P., Ukkonen, E.: Fast search algorithms for position specific scoring matrices. In: Proceedings of 1st International Conference on Bioinformatics Research and Development. Volume 4414 of LNBI, Berlin, Springer-Verlag (2007) 239–250
8. Pizzi, C., Ukkonen, E.: Fast profile matching algorithms – a survey. Theoretical Computer Science (to appear).
9. Salmela, L., Tarhio, J., Kytöjoki, J.: Multi-pattern string matching with q-grams. ACM Journal of Experimental Algorithmics **11** (2006) 1–19
10. Staden, R.: Methods for calculating the probabilities of finding patterns in sequences. Computer Applications in Biosciences **5** (1989) 89–96
11. Wu, T., Neville-Manning, C., Brutlag, D.: Fast probabilistic analysis of sequence function using scoring matrices. Bioinformatics **16**(3) (2000) 233–244