Weighted Degenerated Approximate Pattern Matching*

Costas S. Iliopoulos[1], Inuka Jayasekera[1], Bořivoj Melichar[2], and Jan Šupol[2]

[1] Department of Computer Science, King's College London, London, WC2R 2LS
{csi, jayasek}@dcs.kcl.ac.uk,
[2] Department of Computer Science & Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, Karlovo nám. 13, 121 35 Prague 2,
melichar@fel.cvut.cz, jan.supol@gmail.com

Abstract. We present a bit-parallel approach to degenerated approximate pattern matching problem. That is the problem of finding approximate matches of a "special" pattern in a text of degenerate symbols. The special pattern $P = s_1 *^{(a_1,b_1)} \dots s_\ell *^{(a_\ell,b_\ell)} s_{\ell+1} *^{(a_{\ell+1},b_{\ell+1})} \dots s_\omega$, such that symbol $*^{(a,b)}$ is a sequence of at most b but at least a "don't care" symbols which match any symbol within the alphabet, i.e. a sequence of subpatterns with gaps; the pattern is associated with integer weights in each subpattern s_ℓ for replacements, insertions, and deletions. The problem is to match the pattern such that the minimum sum of weights is achieved. The total time complexity is $(k(\log(k+2)+1)mn)/w$, where m is the length of the pattern P, n is the length of text of degenerate symbols, k is the maximum number of edit operations performed, and w is the length of the computer word.

1 Introduction

Suppose pattern $P = p_1p_2 \dots p_m$ in text $T = t_1t_2 \dots t_n$, where each edit operation has different weight. The weighted pattern matching problem is the problem of finding approximate occurrences with the lowest total weight, that is the most probable occurrences in a text. Traditional sequence comparison tools, such as BLAST [AMS+97] extend the original problem by considering approximate pattern matching, and by computing the highest possible scores between P and a factor of T, offering a wide range of applications in computational molecular biology. Among the rather large spectrum of distances one can use to compare two sequences of symbols, we will consider in what follows the Levenshtein distance, that makes use of three edit operations: insertion, deletion and substitution. The protein sequences are traditionally built upon the 20-symbol alphabet of amino acids.

The PROSITE [HBB+06] database is based on the observation that, while there is a huge number of different proteins, most of them can be grouped, on the basis of

* This research has been partially supported by the Ministry of Education, Youth, and Sport of the Czech Republic under research program MSM6840770014, by the Czech Science Foundation as project No. 201/06/1039, and the Engineering and Physical Sciences Research Council (EPSRC) under the Doctoral Training Award (DTA).

similarities in their sequences, into a limited number of families. Proteins or protein domains belonging to a particular family generally share functional attributes and are derived from a common ancestor.

In what follows, we will use the term *degenerate strings* to refer to strings built upon sets of symbols instead of individual ones. The need for the approximate pattern matching algorithms arises due to environmental and experimental factors which affect the protein assembly. Patterns with error bounds for each subpattern were introduced in [LAIP03] and an efficient algorithm was discussed in [IJM06]. This is based on the assumption that errors are randomly distributed, in such sequences and proteins. If only one error bound is used for the whole pattern P, there can be an approximate occurrence of P such that most errors are concentrated in a certain sub-pattern leaving almost no errors in the remaining sub-pattern, which may possibly be a "false positive". In this paper, we introduce a bit-parallel approach to degenerated approximate pattern matching such that we match the occurrence with the lowest weight, and we allow for each edit operation in each subpattern different integer weight.

The paper is organised as follows. In Section 2, we give the basic definitions and formally describe the computational problems. In Section 3, we outline previous work on the problems. In Section 4, we present the main algorithm. Finally, we conclude and discuss open problems in Section 5.

2 Preliminaries

Let A be a set of symbols called an *alphabet*. A *string* $X = x_1x_2\dots x_n$ is a concatenation of symbols from A; ε is said to be the *empty* string. A *degenerate symbol*, p, is said to be a subset of symbols from A; the symbol set is denoted using square brackets. We say that $T = t_1t_2\dots t_n$ is a *degenerate string*, if t_i are degenerate symbols, for $1 \le i \le n$. A degenerate symbol p matches the i-th position of T, if and only if $t_i \cap p \neq \emptyset$.

The symbol $*$, called *don't care*, is a special symbol that matches every symbol in S. An elastic gap or simply *gap*, $*^{(a,b)}$ is a string of q consecutive don't cares, where $a \le q \le b$. We define the occurrence of the 1-gap pattern $P^{(1)} = s_1{*}^{(a_1,b_1)}$ in T as the occurrence of the subpattern s_1 in T, immediately followed by any q symbols, with q in $[a_1, b_1]$. Then, we recursively define the occurrence of the ω-gap pattern in T

$$P^{(\omega)} = s_1 *^{(a_1,b_1)} \dots s_\ell *^{(a_\ell,b_\ell)} s_{\ell+1} *^{(a_{\ell+1},b_{\ell+1})} \dots s_\omega{*}^{(a_\omega,b_\omega)}$$

as an occurrence of $(\omega-1)$-gap pattern in T with

$$P^{(\omega-1)} = s_1 *^{(a_1,b_1)} \dots s_\ell *^{(a_\ell,b_\ell)} s_{\ell+1} *^{(a_{\ell+1},b_{\ell+1})} \dots s_{\omega-1}{*}^{(a_{\omega-1},b_{\omega-1})}$$

immediately followed by an occurrence of pattern with 1-gap pattern $s_\omega{*}^{(a_\omega,b_\omega)}$, $1 \le \ell < \omega$. We define m as the length of the longest possible occurrence of pattern, P, at sub-text $t_{i-m}t_{i-m+1}\dots t_i$, where there is a match at position i in the text, T.

The Weighted Approximate Degenerate Pattern Matching Problem: Let T be a degenerate text, and $P = s_1 *^{(a_1,b_1)} \dots s_\ell *^{(a_\ell,b_{\ell,})} s_{\ell+1} \dots s_\omega$ be the degenerate pattern. Let w_ℓ^R, w_ℓ^I, w_ℓ^D, and w_ℓ^{MAX}, for $1 \le \ell \le \omega$, be integers such that

$\max(w_\ell^R, w_\ell^I, w_\ell^D) \leq w_\ell^{MAX}$, where $w_\ell^R, w_\ell^I, w_\ell^D$ represent weights of replacements, insertions, and deletions in subpattern s_ℓ. Also, let k be an integer denoting the maximum number of allowed edit operations (errors) in all subpatterns. Compute all the substrings of T, which approximately match P, such that subpattern s_ℓ has the lowest value of

$$c_\ell^R w_\ell^R + c_\ell^I w_\ell^I + c_\ell^D w_\ell^D \leq w_\ell^{MAX}, 1 \leq \ell \leq \omega$$

integers $c_\ell^R, c_\ell^I, c_\ell^D$ are the number of replacements, insertions, and deletions performed in subpattern s_ℓ respectively, and the total number of errors be no greater than k.

2.1 Example

Let the set of characters in brackets be a degenerate symbol and let $T = SRD[PL]ND$ $[MY]YRSPYM$ and $P = RK *^{(2,3)} E[ML]Y *^{(2,3)} Y$ with $w_\ell^D = 1$, $w_\ell^I = 1$, $w_\ell^R = 2$ and $w_\ell^{MAX} = 2$ for all three sub-patterns, $s_1 = RK, s_2 = E[ML]Y$, and $s_3 = Y$. Also let $k = 3$. We say there is one occurrence of P at position 12 of T. Although, $p_j \cap t_i \neq \emptyset, 2 \leq i \leq 12, 1 \leq j \leq m$ is not true for all symbols in P, this is a valid occurrence as we are within the allowed edit operation limits for all replacements, insertions, and deletions. When observing the occurrence, we can see that there is one replacement at position 3, and one deletion between positions 6 and 7 in the text. We can also see that the total number of errors 2 is less than k, and since there is at most one error in each subpattern, the weight does not exceed w^{MAX}. Thus this is a valid occurrence according to the given criteria.

i	1	2	3	4	5	6		7	8	9	10	11	12	13
T	S	R	D	[PL]	N	D		[MY]	Y	R	S	P	Y	M
P		R	K	*	*	*	E	[ML]	Y	*	*	*	Y	
Error weight		0	2	0	0	0	1	0	0	0	0	0	0	
			w^R				w^D							

Although this example considers a case, where the error weights are identical for all sub-patterns, our algorithm is able to deal with different weights (w_ℓ^I, w_ℓ^R, w_ℓ^D and w_ℓ^{MAX}) for each sub-pattern s_ℓ.

3 Background

The traditional approach to gapped pattern matching has been to use a regular expression to represent the pattern and find occurrences of this pattern in a text using regular expression matching. For example, a pattern $R *^{(2,3)} D *^{(3,4)} Y$ that contains two gaps $*^{(2,3)}$ and $*^{(3,4)}$ can be represented as a regular expression $R(*|\varepsilon)*^2 D(*|\varepsilon)*^3 Y$. Occurrences of this pattern are then looked for in the text by creating a non-deterministic automaton (*NFA*) and either using this directly or converting it to a deterministic automaton (*DFA*) for matching [Gus97].

However for our purposes, this approach is too general as it considers a complex set of expressions and introduces unnecessary computations. The set of patterns that contain gaps is a proper subset of strings that can be represented by the regular expression because there is no Kleene closure and the | operation is only used between don't care characters ($*$) and an empty string ε. Furthermore, finding the approximate matching occurrences of a regular expression is difficult and time-consuming. In short, the regular expression matching approach adds unnecessary complexity to our pattern matching problem. We now look at several other approaches which have previously been taken.

Navarro and Raffinot [NR01] presented an $\mathcal{O}(mn/w)$ time algorithm for the exact gapped pattern matching with bounded size gaps, where w is the word size of the target machine, m is the length of the pattern when taking maximal gap constraint lengths and n is the length of the text used. Their algorithm uses the bit-vector simulation of *NFA*, based on the "shift-or" algorithm [BYG92] for the pattern matching. They also represented an $\mathcal{O}(mnk/w)$ time algorithm for the approximate gapped matching problem where the error bound k is for the whole pattern P. Navarro and Raffinot however used the Hamming Distance error model meaning that, an occurrence of P in T, could differ from P by k replacements only. Approximate matching is looked into more detail in Section 4.2.

Akutsu [Aku96] gave an $\mathcal{O}(mn \log n)$ time algorithm for the approximate gapped pattern matching problem. It uses a combination of a balanced search tree and the traditional dynamic programming approach for the approximate pattern matching problem with k errors. The error bound k is for the whole pattern as in [NR01]. However, the algorithm proposed in [NR01] does not handle *classes* of characters.

In addition, there are many existing works on pattern matching in degenerate strings including [HSW05] and [LAIP03].

4 Outline of Algorithm

4.1 Bit-Parallelism

In this paper we solve the weighted approximate degenerate pattern matching using the bit-parallel simulation technique of the non-deterministic pattern matching automaton (*PMA*). This technique has been introduced by Dömölki in [Döm64] (the "shift-and" variation), and improved in [BYG92,WM92,Hol00] (the "shift-or" variation used in this paper).

In what follows, we use some bitwise operations. Operation **or** is standard bitwise OR operation and operation **and** is standard bitwise AND operation. Operation **shl** is a standard shift-left bitwise operation. We also use the operation $\mathbf{shl}^i(x)$ as the operation **shl** performed i times on bit-vector x.

Here, we explain the "shift-or" variation of the bit-parallel algorithm for the simulation of *PMA* as presented by Holub in [Hol00,MHP05], and by Holub, Iliopoulos, Melichar, and Mouchard in [HIMM01]. It uses matrices $R^l, 0 \le l \le k$, each of size $m \times (n+1)$, and matrix D of size $m \times |A|$, where m is the length of the pattern, n is the length of text, and k is the maximum number of edit operations in pattern P. Each element $r^l_{j,i},\ 0 \le i \le n$, contains 0, if the Levenstein distance between string $p_1p_2 \ldots p_j$ and string ending at position i in text $T = t_1t_2 \ldots t_n$ is $\le l$, or 1, otherwise. Each element $d_{j,x},\ 0 < j \le m$, $x \in A$, contains 0, if $p_j = x$, or 1, otherwise.

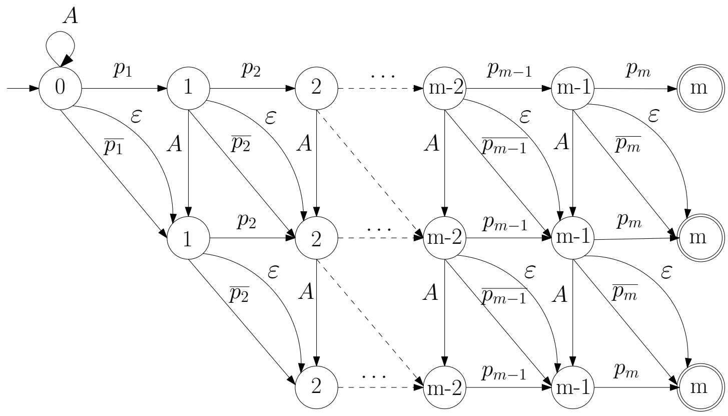

Fig. 1. Levenshtein pattern matching automaton, k=2

In exact pattern matching (consider top most level from Fig. 1 only), vectors $R_i^0,\ 0 \le i \le n$, are computed as follows:

$$\begin{array}{ll} r_{j,0}^0 = 1, & 0 < j \le m \\ R_i^0 \ = \mathbf{shl}(R_{i-1}^0) \ \mathbf{or} \ D[t_i], & 0 < i \le n \end{array} \qquad (1)$$

In approximate pattern matching using the Levenshtein distance, vectors $R_i^l,\ 0 \le l \le k,\ 0 \le i \le n$, are computed as follows:

$$\begin{array}{ll} r_{j,0}^l = 0, & 0 < j \le l,\ 0 \le l \le k \\ r_{j,0}^l = 1, & l < j \le m,\ 0 \le l \le k \\ R_i^0 \ = \mathbf{shl}(R_{i-1}^0) \ \mathbf{or} \ D[t_i], & 0 < i \le n \\ R_i^l \ = (\mathbf{shl}(R_{i-1}^l) \ \mathbf{or} \ D[t_i]) & \\ \quad \mathbf{and\ shl}(R_{i-1}^{l-1} \ \mathbf{and} \ R_i^{l-1}) & \\ \quad \mathbf{and} \ (R_{i-1}^{l-1} \ \mathbf{or} \ V), & 0 < i \le n,\ 0 < l \le k \end{array} \qquad (2)$$

The auxiliary vector V is initialized as $V = 0^{m-1}1$.

The term $\mathbf{shl}(R_{i-1}^l)$ **or** $D[t_i]$ represents matching ($p_1, p_2, \ldots, p_m$ transitions in Fig. 1) – position i in text T is increased, the position in pattern P is increased by operation **shl**, and the positions corresponding to the input symbol t_i are selected by the term **or** $D[t_i]$. The term $\mathbf{shl}(R_{i-1}^{l-1})$ represents edit operation *replace* ($\overline{p_1}, \overline{p_2}, \ldots, \overline{p_m}$ transitions in Fig. 1) – position i in text T is increased, the position in pattern P is increased, and edit distance l is increased. The term $\mathbf{shl}(R_i^{l-1})$ represents edit operation *delete* (ε-transitions in Fig. 1) – the position in the pattern is increased, the position in the text is not increased, and edit distance l is increased. The term R_{i-1}^{l-1} represents edit operation *insert* (transitions for any symbol represented by whole alphabet symbol A in Fig. 1 – note that the loop for A in state 0 is not for *insert*) – position in pattern is not increased, position in the text is increased, and edit distance l is increased. The term **or** V provides that no *insert* transition leads from any final state.

4.2 Weighted approximate pattern matching without gaps

We now formulate the algorithm without gaps, i.e. without "don't care" symbols, therefore we have weights w^R for replaces, w^I for insertions, and w^D for deletions only. Since we consider each degenerate symbol given in binary representation $[b_1 b_2 \ldots b_{|A|}]$, where each bit b_i represents presence of an i-th symbol from the alphabet in the set of symbols. It is clear that the algorithm is the same for symbols of the alphabet and for degenerate symbols, except that matrix D has to be extended for the degenerate symbols.

Our algorithm is derived from Formula 2, but instead having active state represented by one bit, our active state is represented by an integer value corresponding to a reached weight. To handle this, we use technique explained in [Nav04], where a cell of bits $[C_j]_\lambda$ corresponds to state j. Therefore each bit-vector $R_i^l,\ 0 < i \leq n,\ 0 < l \leq k$ has the following structure:

$$0[C_m]_\lambda 0[C_{m-1}]_\lambda \ldots 0[C_1]_\lambda,$$

where m is the length of the pattern, and λ is the number of bits per state. One extra bit extension before each cell is for computational purposes.

Let us suppose, that $max(w^R, w^D, w^I) \leq w^{MAX}$, where w^{MAX} is maximum allowed weight. Then each cell $[C_j]_\lambda,\ 1 \leq j \leq m$ stores values of weights $0, 1, \ldots, w^{MAX}$ or the saturation of at least $w^{MAX}+1$ for inactive state. Therefore $\lambda = \log(w^{MAX}+2)$ and inactive state is represented as 1^λ.

Since we do not need weights w^R, w^I, w^D for exact pattern matching, the formula is similar to Formula 1 and it is as follows:

$$\begin{array}{lll} r_{j,0}^0 = 1, & & 0 < j \leq m(\lambda+1) \\ R_i^0 & = \mathbf{shl}^{\lambda+1}(R_{i-1}^0) \ \mathbf{or}\ D[t_i], & 0 < i \leq n \end{array} \tag{3}$$

Note, that the bit-vector $D[t_i]$ has cell $[C_j]_\lambda = 1^\lambda$ for inactive state j. For active state the cell is $[C_j]_\lambda = 0^\lambda$ and thus the operation **or** does not change the value of weight.

The values w^R, w^I, w^D, and w^{MAX} are needed for approximate pattern matching. Thus we use three additional bit-vectors WR, WI, and WD which refer to the weights added when one of the edit operations *replace*, *insert*, or *delete* is used. Each cell of these bit-vectors contains binary value of corresponding weight, for instance, for ordinary approximate pattern matching, these operation have the same weights equal to 1, and thus each cell $[C_j]_\lambda$ has form $0^{\lambda-1}1$.

We use also an additional bit-vector WM which refers to the maximum allowed reached weight w^{MAX} in each state, for greater weight the algorithm sets the state as inactive. If the maximum is not given, the bit-vector $(01^\lambda)^m$ can be used.

The approximate pattern matching computes weight in a state when approaching from four different states via *match*, *replace*, *insert*, or *delete* transition. Only a minimum from these four states with an addition of a new weight according to w^R, w^I, or w^D is stored in each state as shows the main iteration function 2.

The minimum computation in Fig. 3 has been originally published in [Nav04]. The meaning is as follows. Line 3 set all λ bits to 1, if X is greater then Y and to 0 otherwise. The return value is then minimum of X and Y.

$\mathbf{MainIteration}(l, i)$
$Temp \leftarrow \mathbf{shl}^{\lambda+1}(R^{l}_{i-1})$ **or** $D[t_i]$
$Temp \leftarrow \mathbf{Min}(Temp,\ \mathbf{Add}(\mathbf{shl}^{\lambda+1}(R^{l-1}_{i-1}), WR))$
$Temp \leftarrow \mathbf{Min}(Temp,\ \mathbf{Add}(\mathbf{shl}^{\lambda+1}(R^{l-1}_{i}), WD))$
$Temp \leftarrow \mathbf{Min}(Temp,\ \mathbf{Add}(\mathbf{shl}^{\lambda+1}(R^{l-1}_{i-1})$ **or** $V), WI))$
Return $Temp$

Fig. 2. The main iteration computation

$\mathbf{Min}(X, Y)$
$T \leftarrow (10^{\lambda})^{m}$
$Z \leftarrow ((X$ **or** $T) - Y)$ **and** T
$Z \leftarrow Z - \mathbf{shr}^{\lambda}(Z)$
Return $(Y$ **and** $Z)$ **or** $(X$ **and not** $Z)$

Fig. 3. Bit-parallel minimum computation

The algorithm of addition in Fig. 4 is very similar to the minimum computation. If the weight of X in addition to appropriate edit operation weight W is greater then maximum allowed WM (i.e. for inactive state), then the state is set as inactive (all ones acquired from the negation of temporary bit-vector T). The term $(X + W)$ **and not** $(Z$ **or** $T)$ has the following meaning. Even though $X + W$ could be greater number then to fit in λ bits, it cannot be greater then to fit in $\lambda + 1$ bits (because of the most significant bit set to 0) and the negation of T ensures the most significant bit set back to 0. The negation of Z gives all ones if $X + W$ is less then maximum allowed and thus the result is $X + W$.

$\mathbf{Add}(X, W)$
$T \leftarrow (10^{\lambda})^{m}$
$Z \leftarrow (((X + W)$ **or** $T) - WM)$ **and** T
$Z \leftarrow Z - \mathbf{shr}^{\lambda}(Z)$
Return $((X + W)$ **and not** $(Z$ **or** $T))$ **or** $($**not** T **and** $Z)$

Fig. 4. Bit-parallel addition computation

The approximate pattern matching is then described as Formula 4. Note the bit-vector $V = 01^{\lambda}(00^{\lambda})^{m-1}$.

$$\begin{array}{lll}
r^{l}_{j,0} = 0[C_j]_{\lambda}, & 0 < j \leq l,\ 0 \leq l \leq k,\ [C_j]_{\lambda} = jw^{D} \leq w^{MAX} \\
r^{l}_{j,0} = 01^{\lambda}, & 0 < j \leq l,\ 0 \leq l \leq k,\ jw^{D} > w^{MAX} \\
r^{l}_{j,0} = 01^{\lambda}, & l < j \leq m,\ 0 \leq l \leq k \\
R^{0}_{i} \ = \mathbf{shl}^{\lambda+1}(R^{0}_{i-1}) \ \mathbf{or}\ D[t_i], & 0 < i \leq n \\
R^{l}_{i} \ = \mathbf{MainIteration}(l, i) & 0 < i \leq n,\ 0 < l \leq k
\end{array} \tag{4}$$

4.3 Different conditions for each subpattern

Assume a pattern with the following structure:

$$\hat{P} = s_1 s_2 \dots s_\omega,$$

where s_ℓ is the ℓ-th subpattern. Then for each s_ℓ there is a given weight for edit operation *replace*, *insert*, and *delete* $w_\ell^R, w_\ell^I, w_\ell^D$ and also given maximum w_ℓ^{MAX}. Moreover, we bound the pattern matching condition by additional parameter k, which represents the maximum allowed sum of all differences in the approximate occurrence of the pattern.

Now four changes are needed to make. The settings of bit-vector WM, the setting of bit-vectors of weights WR, WI, and WD, the guarantee of resetting of the weights in bit-vector R at the beginning of each subpattern s_ℓ, and initial bit-vector settings.

The cells $[C_j]_\lambda$ of bit-vector WR are set as follows. For each state j corresponding to a symbol p_j in s_ℓ we set $[C_j]_\lambda = w_\ell^R$. We set bit-vectors WI and WD in a similar way. For each state j corresponding to a symbol p_j in s_ℓ we set $[C_j]_\lambda = w_\ell^{MAX}$ in maximum bit-vector WM.

For the reset of weights we use an auxiliary bit-vector B. For each cell $[C_j]_\lambda$, such that state j is the first state after some gap, we set $[C_j]_\lambda = 0^\lambda$. We set all other cells as $[C_j]_\lambda = 1^\lambda$. Then we use algorithm in Fig. 5 to reset weights when entering a new subpattern via *match*, *replace*, and *delete* transitions, but not *insert* transition which remains in the same subpattern.

Reset(X)
1 $T \leftarrow (10^\lambda)^m$
2 $Z \leftarrow ((X$ **or** $T) -$ **not** $T)$ **and** T
3 $Z \leftarrow Z - \mathbf{shr}^\lambda(Z)$
4 **Return** $(B$ **or** $Z)$ **and** X

Fig. 5. Bit-parallel reset computation

The meaning is as follows. If state in X is active, the weight has to be lower than (**not** T). Therefore Z is equal to T on line 2 if and only if the state is not active. If the state is not active or there is not a beginning of new subpattern (B **or** Z) is all ones, and the function returns X. If both the state is active and there is a beginning of a new subpattern, the return value is all zeros.

The MainIteration algorithm has the form in Fig. 6.

The initial bit-vectors may not be straight forward. The *delete* operation is performed if and only if the maximum weight was not exceeded somewhere previously. Therefore the algorithm Set is depicted in Fig. 7.

The meaning is as follows. If there is the x-th symbol in subpattern s_ℓ such that x weights of *delete* is more than maximum allowed weight $xw_\ell^D > w_\ell^{MAX}$, set this cell and all consecutive cells to zero. Also, do not allow to set more total cells then l.

MainIteration(l, i)

$Temp \leftarrow \mathbf{Reset}(\mathbf{shl}^{\lambda+1}(R^l_{i-1})\ \mathbf{or}\ D[t_i])$
$Temp \leftarrow \mathbf{Min}(Temp,\ \mathbf{Add}(\mathbf{Reset}(\mathbf{shl}^{\lambda+1}(R^{l-1}_{i-1})), WR, M))$
$Temp \leftarrow \mathbf{Min}(Temp,\ \mathbf{Add}(\mathbf{Reset}(\mathbf{shl}^{\lambda+1}(R^{l-1}_{i})), WD, M))$
$Temp \leftarrow \mathbf{Min}(Temp,\ \mathbf{Add}(\mathbf{shl}^{\lambda+1}(R^{l-1}_{i-1})\ \mathbf{or}\ V), WI, M))$
Return $Temp$

Fig. 6. The improved main iteration computation

Set(l)

$Reached \leftarrow$ **false**
$R^l_0 \leftarrow (00^\lambda)^m$
for $j \leftarrow 1, 2, \ldots, m$
  if $Reached$ **or** $j > l$ **or** $xw^D_\ell > w^{MAX}_\ell, p_j = s_\ell[x]$ **then**
    $Z \leftarrow 0$; $Reached \leftarrow$ **true**
  else
    $Z \leftarrow xw^D_\ell, p_j = s_\ell[x]$
  $[C_j]_\lambda \leftarrow Z$
endfor

Fig. 7. Initial settings

4.4 Pattern with gaps

The matching of patterns with gaps has been introduced e.g. in [NR01,IJM06]. Navarro and Raffinot [NR01] presented a bit-parallel "shift-and" algorithm for computing patterns with gap of length between two given constants a and b, $b \geq a \geq 0$. We can reformulate it in "shift-or" terms (we chose the active state represented by 0) as follows:

$$R^l_i = R^l_i \ \mathbf{and\ not}\ (((I\ \mathbf{or}\ R^l_i) - F)\ \mathbf{and}\ F), \quad 0 \leq l \leq k,\ 0 < i \leq n \tag{5}$$

where vectors F and I have the following meaning. Vector I represents "gap-initial" states which are states S_i from where an ε-transition leaves, thus state S_1 considering Fig.8. Vector F represents "gap-final" states which are states $S_{i+b-a+1}$, thus state S_4 in Fig.8. Term I **or** R^l_i isolates active "gap-initial" states, and having active states represented by 0, the value of term I **or** R^l_i is greater then value of F. The substraction results states active via the ε-transitions, but active states are now represented by 1. The operation **and** F removes "gap-final" states and the negation **not** changes the representation of active states back to 0.

We may observe that the term $(I$ **or** $R^l_i) - F$ in Formula 5 fills bit between two zeros with ones. This observation we use when redefine "gap-initial" and "gap-final" bit-vectors I, and F for our case. Both vectors have the same structure $b_m[C_m]_\lambda b_{m-1}$ $[C_{m-1}]_\lambda \ldots b_1[C_1]_\lambda$.

First for the "gap-initial" states, we need to set cells $[C_j]_\lambda = 1^{\lambda-1}0$. Each bit $b_j = 1$. Second for the "gap-final" states j, we set the bit corresponding the one state before to $b_{j-1} = 0$, all other to $b_j = 1$, cells $[C_j]_\lambda = 1^\lambda$. For our four-states example in Fig. 8, suppose $\lambda = 2$, the bit-vectors I, F are as follows: $I = 1[11]1[11]1[11]1[10]$, $F = 1[11]0[11]1[11]1[11]$.

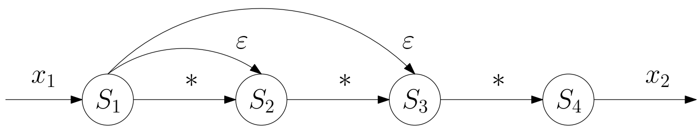

Fig. 8. Part of the automaton with gap $*^{(1,3)}$

However, it will be incorrect to use the bit-vector R_i^l with the structure $b_m[C_m]_\lambda$ $b_{m-1}[C_{m-1}]_\lambda \dots b_1[C_1]_\lambda$, because each bit b_j is set to zero but needed to be one and also each cell $[C_i]_\lambda$ can contain zeroes. Therefore, we use algorithm in Fig. 9 to handle this.

Gaps(X)
1 $T \leftarrow (10^\lambda)^m$
2 $Z \leftarrow ((X$ **or** $T)\ -\$ **not** $T)$ **and** T
3 $Z \leftarrow (Z - \mathbf{shr}^\lambda(Z))$ **or** T
4 $Z \leftarrow ((I$ **or** $Z)\ -\ F)$ **or** T
5 **Return not** Z

Fig. 9. Bit-parallel gap computation

The meaning is similar to algorithm in Fig. 5 and Formula 5. The value of Z on line 3 is 11^λ for inactive state, 10^λ otherwise. The term (I **or** Z) choose the active initial state, the substraction of F sets new active states, but active are those with cells 1^λ. Also some bits b_j were set to 0, and therefore we have to add the term (**or** T) on line 4, so that after negation those bits b_j will be back to 0. Because of that, one can see that the term (**and** F) is not needed on line 4 in this case compared to Formula 5.

Thus the following equation has to be performed after each iteration:

$$R_i^l = R_i^l \textbf{ and } \mathbf{Gaps}(R_i^l), \quad 0 \le l \le k,\ 0 < i \le n \tag{6}$$

This operation will add some new active states reached via ε transitions and these states have weight equal to 0. Since we are not interested in weights in gaps and since this value is reset instantly after the gap, this value of 0 is exactly what we need.

4.5 Complexities

Both functions Add and MainIteration have $\mathcal{O}(1)$ steps, and the function MainIteration is called n times from Formula 4 for each k. The bit-vectors are $m(\lambda + 1)$ bits, which should be fit into w bits of computer words. Therefore total time complexity is $(k(\log\ (k+2)+1)mn)/w$. The motivation is to fit all $(\log\ (k+2)+1)m$ bits into one computer word. In this case the final time complexity is $\mathcal{O}(kn)$.

5 Conclusion

We considered the approximate weighted gapped pattern matching problem in protein sequences. We also presented an $(k(\log(k+2)+1)mn)/w$ time algorithm for the approximate weighted gapped pattern matching problems.

Supposing $w = (\log(k+2)+1)m$, the time complexity is $\mathcal{O}(kn)$. Therefore this algorithm both improves existing algorithm [IJM06], which solves the same problem but with weights equal to one in $\mathcal{O}(mn)$ time, and provides a new possibility of parametrization of approximate pattern matching and thus we successfully answer the question in conclusion of the paper by Navarro and Raffinot [NR01].

The algorithm that we have presented can be used effectively in the motif searching of protein sequences in molecular biology.

A remaining open problem would be for the consideration of an overall maximum weight, w^{MAX}, for the entire pattern. However, at this time we believe individual maximum weights, w_ℓ^{MAX} for each sub-pattern is more useful as it will help to filter out any false positives arising as a result of errors concentrated in a single sub-pattern.

References

[Aku96] T. Akutsu. Approximate string matching with variable length don't care characters. *TIEICE: IEICE Transactions on Communications/Electronics/Information and Systems*, E79-D:1353–1354, 1996.

[AMS+97] S. F. Altschul, T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped BLAST and PSI–BLAST: a new generation of protein database search programs. *Nucleic Acids Res.*, 25:3389–3402, 1997.

[BYG92] Ricardo A. Baeza-Yates and Gaston H. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10):74–82, 1992.

[Döm64] B. Dömölki. An algorithm for syntactic analysis. *Computational Linguistics*, 8:29–46, 1964.

[Gus97] D. Gusfield. *Algorithms on strings, trees and sequences.* Cambridge University Press, 1997.

[HBB+06] N. Hulo, A. Bairoch, V. Bulliard, L. Cerutti, E. De Castro, P.S. Langendijk-Genevaux, M. Pagni, and C.J.A. Sigrist. The prosite database. *Nucleic Acids Research*, 34:227–23, 2006.

[HIMM01] Jan Holub, Costas S. Iliopoulos, Bořivoj Melichar, and Laurent Mouchard. Distributed pattern matching using finite automata. *J. Autom. Lang. Comb.*, 6(2):191–204, 2001.

[Hol00] J. Holub. *Simulation of Nondeterministic Finite Automata in Pattern Matching.* Ph.D. Thesis, Czech Technical University in Prague, February 2000.

[HSW05] J. Holub, W. F. Smyth, and S. Wung. Fast pattern-matching on indeterminate strings (preliminary version). In *Proc. 16th Australasian Workshop on Combinatorial Algorithms*, pages 415–428, 2005.

[IJM06] C. Iliopoulos, I. Jayasekera, and L. Mouchard. Fast practical exact and approximate pattern matching in protein sequences. In *Proceedings of the 17th Australasian Workshop on Combinatorial Algorithms*, Central Australia, July 2006.

[LAIP03] I. Lee, A. Apostolico, C. S. Iliopoulos, and K. Park. Finding approximate occurrences of a pattern that contains gaps. In SNU Press, editor, *Proceedings of the 14th Australasian Workshop on Combinatorial Algorithms*, pages 89–100, 2003.

[MHP05] Bořivoj Melichar, Jan Holub, and Tomáš Polcar. *Text Searching Algorithms, Vol. I.* `http://stringology.org/athens`, 2005.

[Nav04] Gonzalo Navarro. Approximate regular expression searching with arbitrary integer weights. *Nordic J. of Computing*, 11(4):356–373, 2004.

[NR01] Gonzalo Navarro and Mathieu Raffinot. Fast and simple character classes and bounded gaps pattern matching, with application to protein searching. In *RECOMB '01: Proceedings of the fifth annual international conference on Computational biology*, pages 231–240, New York, NY, USA, 2001. ACM Press.

[WM92] Sun Wu and Udi Manber. Fast text searching allowing errors. *Commun. ACM*, 35(10):83–91, 1992.