Inf Retrieval (2008) 11:335–357
DOI 10.1007/s10791-008-9054-z

Efficient algorithms for pattern matching with general gaps, character classes, and transposition invariance

Kimmo Fredriksson · Szymon Grabowski

Published online: 6 March 2008

Abstract We develop efficient dynamic programming algorithms for pattern matching with general gaps and character classes. We consider patterns of the form $p_0g(a_0,b_0)$ $p_1g(a_1,b_1)\ldots p_{m-1}$, where $p_i \subset \Sigma$, Σ is some finite alphabet, and $g(a_i,b_i)$ denotes a gap of length $a_i\ldots b_i$ between symbols p_i and p_{i+1}. The text symbol t_j matches p_i iff $t_j \in p_i$. Moreover, we require that if p_i matches t_j, then p_{i+1} should match one of the text symbols $t_{j+a_i+1}\ldots t_{j+b_i+1}$. Either or both of a_i and b_i can be negative. We also consider transposition invariant matching, i.e., the matching condition becomes $t_j \in p_i + \tau$, for some constant τ determined by the algorithms. We give algorithms that have efficient average and worst case running times. The algorithms have important applications in music information retrieval and computational biology. We give experimental results showing that the algorithms work well in practice.

Keywords String matching · Sparse dynamic programming · Bounded length gaps · Character classes · Transposition invariance

1 Introduction

Background Many notions of approximateness have been proposed in string matching literature, usually motivated by some real problems. One of seemingly underexplored problem with applications in music information retrieval (MIR) and molecular biology (MB) is *pattern matching with gaps* (Crochemore et al. 2002). In this problem, gaps (text substrings) of length up to α are allowed between each pair of matching pattern characters. Moreover, in MIR applications the character matching can be relaxed with δ-matching,

K. Fredriksson (✉)
Department of Computer Science, University of Kuopio, P.O. Box 1627, 70211 Kuopio, Finland
e-mail: kimmo.fredriksson@cs.uku.fi

S. Grabowski
Department of Computer Engineering, Technical University of Łódź, Al. Politechniki 11, 90-924 Lodz, Poland
e-mail: sgrabow@kis.p.lodz.pl

i.e., the pattern character matches if its numerical value differs at most by δ to the corresponding text character. In MB applications the singleton characters can be replaced by classes of characters, i.e., text character t matches a pattern character p if $t \in p$, where p is some subset of the alphabet.

In MIR, practical matching models usually also incorporate *transposition invariance*, i.e., invulnerability to shifting the whole pattern (over an integer alphabet) by any fixed value. It is motivated by the fact that humans recognize a melody by the intervals between successive notes rather than the pitches themselves.

Previous work Let us start the review from the problem without transposition invariance. The first algorithm in this setting (Crochemore et al. 2002) is based on dynamic programming, and runs in $O(nm)$ time, where n and m are the lengths of the text and pattern, respectively. This basic dynamic programming solution can also be generalized to handle more general gaps while keeping the $O(nm)$ time bound (Pinzón and Wang 2005). The basic algorithm was later reformulated (Cantone et al. 2005a) to allow to find all pattern occurrences, instead of only the positions where the occurrence ends. This needs more time, however. The algorithm in Cantone et al. (2005b) improves the average case of the one in Cantone et al. (2005a) to $O(n)$, but they assume constant α. Bit-parallelism can be used to improve the dynamic programming-based algorithm to run in $O(\lceil n/w \rceil m + n\delta)$ and $O(\lceil n/w \rceil \lceil \alpha\delta/\sigma \rceil + n)$ time in worst and average case, respectively, where w is the number of bits in a machine word, and σ is the size of the alphabet (Fredriksson and Grabowski 2006).

For the α-matching with classes of characters there exists an efficient bit-parallel nondeterministic automaton solution (Navarro and Raffinot 2003). This also allows gaps of different lengths between each pair of successive pattern characters. This algorithm can be trivially generalized to handle (δ,α)-matching (Cantone et al. 2005b), but the time complexity becomes $O(n\lceil \alpha m/w \rceil)$ in the worst case. For small α the algorithm can be made to run in $O(n)$ time on average. The worst case time can be improved to $O(n\lceil m\log(\alpha)/w \rceil)$ (Fredriksson and Grabowski 2006), but this assumes equal length gaps.

Sparse dynamic programming can be used to solve the problem in $O(n + |\mathcal{M}|\min\{\log(\delta+2), \log\log(m)\})$ time, where $\mathcal{M} = \{(i,j) | |p_i - t_j| \le \delta\}$ (and thus $|\mathcal{M}| \le nm$) (Mäkinen 2003). This can be extended for the harder problem variant where transposition invariance and character insertions, substitutions or mismatches are allowed together with (δ,α)-matching (Mäkinen et al. 2005). In this case the $|\mathcal{M}|$ factor becomes nm.

Our results We develop several algorithms, for both major problem variants. Our techniques are based on sparse dynamic programming and bit-parallelism. Our first algorithm for (δ,α)-matching without transposition invariance is essentially a reformulation of the algorithm in Mäkinen et al. (2005). The worst case running time of the algorithm is $O(n + |\mathcal{M}|)$. Our variant has the benefit that it generalizes in straight-forward way to handle general and even negative gaps, important in some MB applications (Mehldau and Myers 1993; Myers 1996). We then give several variants of this algorithm to improve its average case running time to close to linear, while increasing the worst case time only up to $O(n + |\mathcal{M}|(\log(n) + \alpha))$. This algorithm assumes fixed integer alphabet. We also present two simple and practical algorithms that run in $O(n)$ time on average for $\alpha = O(\sigma/\delta)$, but have $O(n + \min(nm, |\mathcal{M}|\alpha))$ worst case time, for any unbounded real alphabets. One of these algorithms is then modified to work in sublinear time in average. Finally, we extend our recent non-deterministic finite automaton-based algorithm (Fredriksson and Grabowski 2006) in order to improve its average case time complexity to sublinear for realistic parameter combinations, without compromising its worst case time complexity.

These are the first algorithms that achieve good average and worst case complexities simultaneously, and they are shown to perform well in practice too.

We also present two algorithms handling the problem with transposition invariance, both based on bit-parallelism and having attractive average case time complexities and performing reasonably well in the worst case.

2 Preliminaries

Let the pattern $P = p_0p_1p_2 \ldots p_{m-1}$ and the text $T = t_0t_1t_2 \ldots t_{n-1}$ be numerical strings, where p_i, $t_j \in \Sigma$ for some integer alphabet Σ of size σ. The number of distinct symbols in the pattern is denoted by σ_p. We sometimes call the set of distinct symbols in the pattern the pattern alphabet.

In δ-approximate string matching the symbols a, $b \in \Sigma$ match, denoted by $a =_\delta b$, iff $|a - b| \le \delta$. Pattern P (δ,α)-matches the text substring $t_{j_0}t_{j_1}t_{j_2} \ldots t_{j_{m-1}}$, if $p_i =_\delta t_{j_i}$ for $i \in \{0,\ldots, m-1\}$, where $j_i < j_{i+1}$, and $j_{i+1} - j_i \le \alpha + 1$. If string A (δ,α)-matches string B, we sometimes write $A =_\delta^\alpha B$.

In all our analyses we assume uniformly random distribution of characters in T and P, and constant α and δ/σ, unless otherwise stated. Moreover, we often write δ/σ to be terse, but the reader should understand that we mean $(2\delta + 1)/\sigma$, which is the upper bound for the probability that two randomly picked characters match.

The dynamic programming solution to (δ,α)-matching is based on the following recurrence (Crochemore et al. 2002; Cantone et al. 2005a):

$$D_{i,j} = \begin{cases} j & t_j =_\delta p_i \text{ AND } (i = 0 \text{ OR } (i,j \ge 1 \text{ AND } D_{i-1,j-1} \ge 0)) \\ D_{i,j-1} & t_j \ne_\delta p_i \text{ AND } j > 0 \text{ AND } j - D_{i,j-1} < \alpha + 1 \\ -1 & \text{otherwise} \end{cases} \quad (1)$$

In other words, if $D_{i,j} = j$, then the pattern prefix $p_0 \ldots p_i$ has an occurrence ending at text character t_j, i.e., $p_i =_\delta t_j$ and the prefix $p_0 \ldots p_{i-1}$ occurs at position $D_{i-1,j-1}$, and the gap between this position and the position j is at most α. If $p_i \ne_\delta t_j$, then we try to extend the match by extending the gap, i.e., we set $D_{i,j} = D_{i,j-1}$ if the gap does not become too large. Otherwise, we set $D_{i,j} = -1$. The algorithm then fills the table $D_{0\ldots m-1,0\ldots n-1}$, and reports an occurrence ending at position j whenever $D_{m-1,j} = j$. This is simple to implement, and the algorithm runs in $O(nm)$ time using $O(nm)$ space.

We first present efficient algorithms to the above problem, and then show how these can be generalized to handle arbitrary gaps, tackling with both upper and lower bounded gap lengths, and even negative gap lengths, and using general classes of characters instead of δ-matching.

3 Row-wise sparse dynamic programming

The algorithm we now present can be seen as a row-wise variant of the sparse dynamic programming algorithm of the algorithm in Mäkinen et al. (2005, Sect. 5.4). We show how to improve its average case running time. Our variant can also be easily extended to handle more general gaps (see Sect. 7).

3.1 Efficient worst case

From the recurrence of D it is clear that the interesting computation happens when $t_j =_\delta p_i$, and otherwise the algorithm just copies previous entries of the matrix or fills some of the cells with a constant.

Let $\mathcal{M} = \{(i,j)|p_i =_\delta t_j\}$ be the set of indexes of the δ-matching character pairs in P and T. For every $(i,j) \in \mathcal{M}$ we compute a value $d_{i,j}$. For the pair (i,j) where $d_{i,j}$ is defined, it corresponds to the value of $D_{i,j}$. If $(i,j) \notin \mathcal{M}$, then $d_{i,j}$ is not defined. Note that $d_{m-1,j}$ is always defined if P occurs at $t_{h\ldots j}$ for some $h < j$. The new recurrence is

$$d_{i,j} = j|(i-1,j') \in \mathcal{M} \text{ AND } 0<j-j' \leq \alpha+1 \text{ AND } d_{i-1,j'} \neq -1,$$

and -1 otherwise. Computing the d values is easy once $\mathcal{M}$ is computed. As we have an integer alphabet, we can use table look-ups to compute $\mathcal{M}$ efficiently. Instead of computing $\mathcal{M}$, we compute lists $L[p_i]$, where $L[p_i] = \{j|p_i =_\delta t_j\}$. These are obtained by scanning the text linearly, and inserting j into each list $L[p_i]$ such that p_i δ-matches t_j. Clearly, there are at most $O(\delta)$ and in average only $O(\delta\sigma_\mathrm{P}/\sigma)$ symbols p_i that δ-match t_j. Therefore this can be obtained in $O(\delta n)$ worst case time, and the average case complexity is $O(n(\delta\sigma_\mathrm{P}/\sigma + 1))$. Note that $|\mathcal{M}|$ is $O(nm)$ in the worst case, but the total length of all the lists is at most $O(\min\{\sigma_\mathrm{P},\delta\}\ n)$, hence L is a compact representation of $\mathcal{M}$. The indexes in $L[p_i]$ will be in increasing order.

Consider a row-wise computation of d. The values of the first row $d_{0,j}$ correspond one to one to the list $L[p_0]$, that is, the text positions j where $p_0 =_\delta t_j$. The subsequent rows d_i correspond to $L[p_i]$, with the additional constraint that $j-j' \leq \alpha + 1$, where $j' \in L[p_{i-1}]$ and $d_{i-1,j'} \neq -1$. Since the values in $L[p_i]$ and d_{i-1} are in increasing order, we can compute the current row i by traversing the lists $L[p_i]$ and d_{i-1} simultaneously, trying to enforce the condition that $L[p_i][h] - d_{i-1,k} \leq \alpha + 1$ for some h, k. If the condition cannot be satisfied for some h, we store -1 to $d_{i,h}$, otherwise we store the text position $L[p_i][h]$. The algorithm traverses L and $\mathcal{M}$ linearly, and hence runs in $O(n + |\mathcal{M}|)$ worst case time. We now consider improving the average case time of this algorithm.

3.2 Efficient average case

The basic sparse algorithm still does some redundant computation. To compute the values $d_{i,j}$ for the current row i, it laboriously scans through the list $L[p_i]$, for all positions, even for the positions close to where $p_0 \ldots p_{i-1}$ did not match. In general, the number of text positions with matching pattern prefixes decreases exponentially on average when the prefix length i increases. Yet, the list length $|L[p_i]|$ will stay approximately the same. The goal is therefore to improve the algorithm so that its running time per row depends on the number of matching pattern prefixes on that row, rather than on the number of δ-matches for the current character on that row.

The modifications are simple: (1) the values $d_{i,j} = -1$ are not maintained explicitly, they are just not stored since they do not affect the computation; (2) the list $L[p_i]$ is not traversed sequentially, position by position, but binary search is used to find the next value that may satisfy the condition that $L[p_i][h] - d_{i-1,k} \leq \alpha + 1$ for some h, k.

Consider now the average search time of this algorithm. The average length of each list $L[p_i]$ is $O(n\delta/\sigma)$. Hence this is the time needed to compute the first row of the matrix, i.e., we just copy the values in $L[p_0]$ to be the first row of d. For the subsequent rows we execute one binary search over $L[p_i]$ per each stored value in row i of the matrix. Hence in general, computing the row i of the matrix takes time $O(|d_{i-1}| \log(n\delta/\sigma))$, where $|d_i|$ denotes the number of stored values in row i. For $i > 0$ this decreases exponentially as $|d_i| = O(n(\delta/\sigma) \times \rho^i)$, where $\rho = 1 - (1 - \delta/\sigma)^{\alpha+1} < 1$ is the probability that a pattern symbol δ-matches in a text window of length α symbols. Summing up the resulting geometric series over all rows we obtain $O(n\frac{\delta}{\sigma(1-\delta/\sigma)^{\alpha+1}})$, which is $O(n\alpha\delta/\sigma)$ for $\delta/\sigma < 1 - \alpha^{-1/(\alpha+1)}$. In particular this is $O(n)$ for $\alpha = O(\sigma/\delta)$. Hence the average search time is $O(n + n\alpha\delta/\sigma\log(n\delta/\sigma))$. However, the worst case search time is also increased to $O(n + |\mathcal{M}|\log(|\mathcal{M}|/m))$. We note that this can be improved to $O(n + |\mathcal{M}|\log\log((nm)/|\mathcal{M}|))$ by using efficient priority queues (Johnson 1982) instead of binary search.

3.3 Faster preprocessing

The $O(\delta n)$ (worst case) preprocessing time can dominate the average case search time in some cases. Note however, that the preprocessing time can never exceed $O(n + |\mathcal{M}|)$. We now present two methods to improve the preprocessing time. The first one reduces the worst case preprocessing cost to $O(\sqrt{\delta}n)$, and improves its average case as well. The second method achieves $O(n)$ preprocessing time, but the worst case search time is slightly increased.

3.3.1 $O(\sqrt{\delta}n)$ time preprocessing

The basic idea is to partition the alphabet into $\sigma/\sqrt{\delta}$ disjoint intervals $I_h, h = 0\ldots\sigma/\sqrt{\delta} - 1$ of size $\sqrt{\delta}$ each (w.l.o.g. we assume that δ is a square number and $\sqrt{\delta}$ divides σ). Then, for each alphabet symbol s, its respective $[s - \delta, s + \delta]$ interval wholly covers $\Theta(\sqrt{\delta})$ intervals I_h, and also can partially cover at most two I_h intervals. Two kinds of lists are computed in the preprocessing, L_B (for "boundary" cases) and L_C (for "core"). For each character t_j from text T, at most $2(\sqrt{\delta} - 1)$ lists $L_B[p_i]$ are extended with one entry, the text position j, and those lists correspond to the pattern alphabet symbols from the partially covered intervals I_h. For example, if $\Sigma = \{0, \ldots, 29\}$, $\sqrt{\delta} = 3$, and $t_j = 10$, then the $[t_j - \delta, t_j + \delta]$ interval is [1, 19], and j is appended to the lists $L_B[1]$, $L_B[2]$, $L_B[18]$, $L_B[19]$, assuming that P contains all the symbols 1, 2, 18, and 19 (if not, the respective lists are not built at all). Figure 1 illustrates. Similarly, each character t_j also causes to append j to $O(\sqrt{\delta})$ lists $L_C[p_i/\sqrt{\delta}]$, those that correspond to the I_h intervals wholly covered by $[t_j - \delta, t_j + \delta]$.

More formally (and still assuming for simplicity that δ is a square number) text position j is appended to the lists $L_B[p_i]$ for

$$p_i \in \{t_j - \delta\ldots\lceil(t_j - \delta)/\sqrt{\delta}\rceil\sqrt{\delta} - 1,\ \lfloor(t_j + \delta + 1)/\sqrt{\delta}\rfloor\sqrt{\delta}\ldots t_j + \delta\}.$$

Likewise, j is appended to the lists $L_C[p_i/\sqrt{\delta}]$ for

$$p_i/\sqrt{\delta} \in \{\lceil(t_j - \delta)/\sqrt{\delta}\rceil\ldots\lfloor(t_j + \delta + 1)/\sqrt{\delta}\rfloor - 1\}.$$

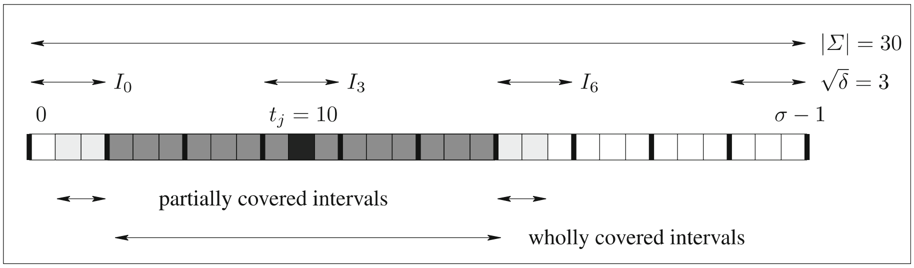


Fig. 1 $O(\sqrt{\delta}n)$ time preprocessing. The current text symbol is $t_j = 10$, and $\sqrt{\delta} = 3$. Its δ-interval spans over the dark-shaded and light-shaded cells. The light-shaded symbols (1, 2, 18, 19) are the "boundary" cases corresponding to the two partially covered intervals, and j is appended to the corresponding L_B lists. The dark-shaded intervals (1, 2, 3, 4, 5) are the fully covered "core" cases, and j is appended to the corresponding L_C lists

Clearly, the preprocessing is done in $O(\sqrt{\delta}n)$ worst case time and in $O(n\sqrt{\delta}\sigma_{\mathrm{P}}/\sigma)$ average time.

The search is again based on a binary search routine, but in this variant we binary search two lists: $L_{\mathrm{B}}[p_i]$ and $L_{\mathrm{C}}[p_i/\sqrt{\delta}]$, as the δ-matches to p_i may be stored either at some L_{B}, or at some L_{C} list. This increases both the average and worst case search cost only by a constant factor.

We can generalize this idea and have a preprocessing/search trade-off. Namely, we may have k levels, turning the preprocessing cost into $O(k\delta^{1/k}n)$, for the price of a multiplicative factor k in the search. For $k = \log(\delta)$ the preprocessing cost becomes $O(n\log(\delta))$, and both the average and worst case search times are multiplied by $\log(\delta)$ as well.

3.3.2 O(n) time preprocessing

We partition the alphabet into $\lceil\sigma/\delta\rceil$ disjoint intervals of width δ. With each interval a list of character occurrences will be associated. Namely, each list $L[i], i = 0\ldots\lceil\sigma/\delta\rceil - 1$, corresponds to the characters $i\delta\ldots\min\{(i+1)\,\delta - 1, \sigma - 1\}$. During the scan over the text in the preprocessing phase, we append each index j to up to three lists: $L[k]$ for such k that $k\,\delta \le t_j \le (k+1)\,\delta - 1$, $L[k-1]$ (if $k - 1 \ge 0$), and $L[k+1]$ (if $k + 1 \le \lceil\sigma/\delta\rceil - 1$). Note that no character from the range $[t_j - \delta\ldots t_j + \delta]$ can appear out of the union of the three corresponding intervals. Such preprocessing clearly needs $O(n)$ space and time in the worst case.

Now the search algorithm runs the binary search over the list $L[k]$ for such k that $k\delta \le p_i \le (k+1)\delta - 1$, as any j such that $t_j =_\delta p_i$ must have been stored at $L[k]$. Still, the problem is there can be other text positions stored on $L[k]$ too, as the only thing we can deduce is that for any j in the list $L[k]$, t_j is $(2\delta - 1)$-match to p_i. To overcome this problem, we have to verify if t_j is a real δ-match. If $t_j \ne_\delta p_i$, we read the next value from $L[k]$ and continue analogously. After at most $\alpha + 1$ read indexes from $L[k]$ we either have found a δ-match prolonging the matching prefix, or we have fallen off the $(\alpha + 1)$-sized window. As a result, the worst case time complexity is $O(n + |\mathcal{M}|(\log(n) + \alpha))$. The average time in this variant becomes $O(n + n\alpha\delta/\sigma\log(n))$. Algorithm 1 shows the complete pseudo code.

Alg. 1 SDP-rows($T, n, P, m, \delta, \alpha$).

```
for $j \leftarrow 0$ to $n - 1$ do
    for $c \leftarrow \max\{0, \lfloor t_j/\delta \rfloor - 1\}$ to $\min\{\lfloor(\sigma - 1)/\delta\rfloor, \lfloor t_j/\delta \rfloor + 1\}$ do
        $L[c] \leftarrow L[c] \cup \{j\}$
for $i \leftarrow 0$ to $|L[p_0]| - 1$ do
    $j \leftarrow L[p_0][i]$
    if $|t_j - p_0| \le \delta$ then $D'_i \leftarrow j$
$h \leftarrow |L[p_0]|$
for $i \leftarrow 1$ to $m - 1$ do
    $c \leftarrow p_i; pl \leftarrow h; k \leftarrow 0; h \leftarrow 0; u \leftarrow 0$
    while $u < |L[c]|$ AND $k < pl$ do
        $j \leftarrow L[c][u]$
        do  $j' \leftarrow D'_k$
            if $j - j' > \alpha + 1$ AND $k < pl$ then $k \leftarrow k + 1$
        while $j - j' > \alpha + 1$ AND $k < pl$
        if $j' < j$ AND $k < pl$ AND $|t_j - c| \le \delta$ then
            $D_h \leftarrow j; h \leftarrow h + 1$
            if $i = m - 1$ then report match
        if $k < pl$ then $u \leftarrow \min\{v \mid D'_k < L[c][v], v > u\}$
    swap($D$, $D'$)
```

3.4 Improved algorithm for large α

In this section we present a variant of the row-wise SDP algorithm, particularly suited to problem instances with large α.

In the preprocessing, we again compute lists $L[p_i] = \{j | t_j =_\delta p_i\}$. But now we also store $2\lfloor n/(\alpha + 1)\rfloor$ pointers to each list. In each list, for each $j = k\ (\alpha + 1)$ where $k \in 0 \ldots n/(\alpha + 1) - 1$, there are two pointers, showing the leftmost and the rightmost item with the value from the interval $[j, j + \alpha + 1]$. These pointers are kept in two 2-dimensional arrays, named $\mathcal{L}$ and $\mathcal{R}$. More formally, the elements of $\mathcal{L}$ and $\mathcal{R}$ are defined in the following way:

$$\mathcal{L}[p_i, k] = \min\{j | j \in L[p_i] \text{AND}\, j \in [k(\alpha + 1) \ldots (k + 1)(\alpha + 1)]\}$$
$$\mathcal{R}[p_i, k] = \max\{j | j \in L[p_i] \text{AND}\, j \in [k(\alpha + 1) \ldots (k + 1)(\alpha + 1)]\}$$

assuming the minimum and the maximum is seeked over a non-empty slice of a list $L[p_i]$. If this is not the case, the respective pointers are set to null. In total, the extra preprocessing cost is $O(\delta n + \sigma_P n/\alpha)$ in time, and $O(\sigma_P n/\alpha)$ space, in the worst case.

The search is basically prefix prolongation. A specific trait of the algorithm is that during the search we are not interested in finding all matching prefixes: what is enough are (at most) two prefixes per an $(\alpha + 1)$-sized chunk of each row (except for the last row, where we perform an extra scan, to be described later). The end positions of those prefixes are maintained in two auxiliary arrays, $\mathcal{C}_L$ and $\mathcal{C}_R$, of size $\lfloor n/(\alpha + 1)\rfloor$ each. They are initialized with the exact copy of the rows $\mathcal{L}[p_0]$ and $\mathcal{R}[p_0]$, respectively.

Now we assume the matrix row i we are in is at least 1. W.l.o.g. we also assume that we are in the column at least $\alpha + 1$. For each $k \in 1 \ldots n/(\alpha + 1) - 1$ we read $\mathcal{L}[p_i, k]$ and $\mathcal{R}[p_{i-1}, k - 1]$, and if both are non-null and $\mathcal{L}[p_i, k] - \mathcal{R}[p_{i-1}, k - 1]$ is at most $\alpha + 1$, then we have found a relevant prefix, which we write to $\mathcal{C}_L$. If not, we check if $\mathcal{L}[p_i, k] - \mathcal{L}[p_{i-1}, k] > 0$ (note that this difference cannot be greater than $\alpha + 1$, so testing for a positive difference of non-null values is all we need). Affirmative answer again corresponds to finding a relevant prefix (and requires updating $\mathcal{C}_L[k]$), but a negative one means that we have to look for a prefix prolongation somewhere further in the current chunk. In such case, we perform a binary search over the fragment of the list $L[p_i]$ with the boundaries kept in the pointers $\mathcal{L}[p_i, k]$ and $\mathcal{R}[p_i, k]$, to find the smallest value being greater than $\mathcal{L}[p_{i-1}, k]$. The interval has as most $\alpha + 1$ items, so the binary search cost is $O(\log(\alpha))$. If this results in a failure (which happens only if the considered interval is empty), it means that we do not have a prefix ended in the current chunk, and $\mathcal{C}_L[k]$ should be updated with a null value.

Analogously we proceed at the right boundary of each chunk. The invariant for the procedure is that after processing a row i, all the end positions of $p_0 \ldots p_i$ in the text chunk $t_{k(\alpha+1)} \ldots t_{(k+1)(\alpha+1)}$ are exactly those $\mathcal{C}_L[k] \le j \le \mathcal{C}_R[k]$ for whose t_j δ-matches p_i, assuming non-null values of $\mathcal{C}_L[k]$ and $\mathcal{C}_R[k]$. If either $\mathcal{C}_L[k]$ or $\mathcal{C}_R[k]$ is null, there are no prefixes ending in the given text chunk.

As mentioned, the last row requires an extra scan, to find all the δ-matches between the positions stored in $\mathcal{C}_L[k]$ and $\mathcal{C}_R[k]$, for all $k \in 0 \ldots n/(\alpha + 1) - 1$. Note that it is possible that $\mathcal{C}_L[k] = \mathcal{C}_R[k]$ or $\mathcal{C}_R[k] = \mathcal{C}_L[k + 1]$, so we must be careful not to count duplicates more than once. This stage needs $O(n)$ time, i.e., is always dominated by the preprocessing time.

The overall search complexity can be bounded by $O(n + nm \log(\alpha)/\alpha)$, but a closer look tells we can bound it better: with $O(n + nm\log(\alpha|\mathcal{M}|/(nm))/\alpha)$. Indeed, a single chunk may have up to $\alpha + 1$ items over which we binary search, but in total there are only $|\mathcal{M}|$ matches in the matrix, which can be much less than nm. This means that on average there are $O(\alpha|\mathcal{M}|/(nm))$ items in a chunk, and equal number of matches in chunks leads also to the worst overall case, which is trivially implied from the convexity of the log function.

On the theoretical side, we note that the achieved worst-case complexity dominates over existing algorithms on the pointer machine (where, e.g., bit-parallelism is forbidden), in the case $|\mathcal{M}| = O(nm)$.

4 Column-wise sparse dynamic programming

In this section we present a column-wise variant. This algorithm runs in $O(n + n\alpha\delta/\sigma)$ and $O(n + \min(|\mathcal{M}|\alpha, nm))$ average and worst case time, respectively.

The algorithm processes the dynamic programming matrix column-wise. Let us define *last prefix occurrence*$\mathcal{D}$ as

$$\mathcal{D}_{i,j} = \begin{cases} j' & \max j' \leq j | p_0 \ldots p_i =^{\alpha}_{\delta} t_h \ldots t_{j'} \\ -\alpha - 1 & \text{otherwise} \end{cases} \tag{2}$$

Note that $\mathcal{D}_{0,j} = j$ if $p_0 =_\delta t_j$. Note also that $\mathcal{D}_{i,j}$ is just an alternative definition of $D_{i,j}$ (Eq. 1). The pattern matching task is then to report every j such that $\mathcal{D}_{m-1,j} = j$. As seen, this is easy to compute in $O(nm)$ time. In order to do better, we maintain a list of *window prefix occurrences*$\mathcal{W}_j$ that contains for the current column j all the rows i such that $j - \mathcal{D}_{i,j} \leq \alpha$ where $i \in \mathcal{W}_j$.

Assume now that we have computed $\mathcal{D}$ and $\mathcal{W}$ up to column $j - 1$, and want to compute $\mathcal{D}$ and $\mathcal{W}$ for the current column j. The invariant is that $i \in \mathcal{W}_{j-1}$ iff $j - \mathcal{D}_{i,j-1} \leq \alpha + 1$. In other words, if $i \in \mathcal{W}_{j-1}$ and $j' = \mathcal{D}_{i,j-1}$, then $p_0 \ldots p_i =^{\alpha}_{\delta} t_h \ldots t_{j'}$ for some h. Therefore, if $t_j =_\delta p_{i+1}$, then the (δ,α)-matching prefix from $\mathcal{D}_{i,j-1}$ can be extended to text position j and row $i + 1$. In such case we update $\mathcal{D}_{i+1,j}$ to be j, and put the row number $i + 1$ into the list $\mathcal{W}_j$. This is repeated for all values in $\mathcal{W}_{j-1}$. After this we check if also p_0 δ-matches the current text character t_j, and in such case set $\mathcal{D}_{0,j} = j$ and insert j into $\mathcal{W}_j$. Finally, we must put all the values $i \in \mathcal{W}_{j-1}$ to $\mathcal{W}_j$ if the row i was not already there, and still it holds that $j - \mathcal{D}_{i,j} \leq \alpha$. This completes the processing for the column j.

Algorithm 2 gives the code. Note that the additional space we need is just $O(m)$, since only the values for the previous column are needed for $\mathcal{D}$ and $\mathcal{W}$. In the pseudo code this is implemented by using $\mathcal{W}$ and $\mathcal{W}'$ to store the prefix occurrences for the current and previous column, respectively.

Alg. 2 SDP-columns$(T, n, P, m, \delta, \alpha)$.

```
1   for $i \leftarrow 0$ to $m-1$ do $\mathcal{D}_i \leftarrow -\alpha - 1$
2   $top \leftarrow 0$
3   for $j \leftarrow 0$ to $n-1$ do
4       $c \leftarrow t_j; h \leftarrow 0$
5       for $i \leftarrow 0$ to $top - 1$ do
6           $pr \leftarrow \mathcal{W}'_i$
7           if $|c - p_{pr+1}| \leq \delta$ then
8               if $pr + 1 < m - 1$ then
9                   $\mathcal{W}_h \leftarrow pr + 1; h \leftarrow h + 1$
10              else
11                  report match
12      if $|c - p_0| \leq \delta$ then
13          $\mathcal{W}_h \leftarrow 0; h \leftarrow h + 1$
14      for $i \leftarrow 0$ to $h - 1$ do $\mathcal{D}_{\mathcal{W}_i} \leftarrow j$
15      for $i \leftarrow 0$ to $top - 1$ do
16          if $\mathcal{D}_{\mathcal{W}_i} \neq j$ AND $j - \mathcal{D}_{\mathcal{W}_i} \leq \alpha$ then
17              $\mathcal{W}_h \leftarrow \mathcal{W}'_i; h \leftarrow h + 1$
18      $top \leftarrow h$
19      swap$(\mathcal{W}, \mathcal{W}')$
```

The average case running time of the algorithm depends on how many values there are on average in the list $\mathcal{W}$. Similar analysis as in Sect. 3 can be applied to show that this is $O(\alpha\delta/\sigma)$. Each value is clearly processed in constant worst case time, and hence the algorithm runs in $O(n + n\alpha\delta/\sigma)$ average time. In the worst case the total length of the lists for all columns is $O(\min(|\mathcal{M}|\alpha, nm))$, and therefore the worst case running time is $O(n + \min(|\mathcal{M}|\alpha, nm))$, since every column must be visited. The preprocessing phase only needs to initialize $\mathcal{D}$, which takes $O(m)$ time.

Finally, note that this algorithm can be seen as a simplification of the algorithm in Mäkinen et al. (2005, Sect. 5.4). We avoid the computation of $\mathcal{M}$ in the preprocessing phase and traversing it in the search phase. The price we pay is a deterioration in the worst case time complexity, but we achieve simpler algorithm that is efficient on average. This also makes the algorithm alphabet independent.

5 Simple algorithm

In this section we will develop a simple algorithm that in practice performs very well on small (δ,α). The algorithm inherits the main idea from Algorithm 1, and actually can be seen as its brute-force variant. The algorithm has two traits that distinguish it from Algorithm 1: (i) the preprocessing phase is interweaved with the searching (lazy evaluation); (ii) binary search of the next qualifying match position is replaced with a linear scan in an $\alpha + 1$ wide text window. These two properties make the algorithm surprisingly simple and efficient on average, but impose an $O(\alpha)$ multiplicative factor in the worst case time bound.

The algorithm begins by computing a list L of δ-matches for p_0:

$$L_0 = \{j | t_j =_\delta p_0\}.$$

This takes $O(n)$ time (and solves the (δ,α)-matching problem for patterns of length 1). The matching prefixes are then iteratively extended, subsequently computing lists:

$$L_i = \{j | t_j =_\delta p_i \text{ AND } j' \in L_{i-1} \text{ AND } 0 < j - j' \le \alpha + 1\}.$$

List L_i can be easily computed by linearly scanning list L_{i-1}, and checking if any of the text characters $t_{j'+1} \ldots t_{j'+\alpha+1}$, for $j \in L_{i-1}$ δ-matches p_i. This takes $O(\alpha|L_{i-1}|)$ time. Clearly, in the worst case the total length of all the lists is $\sum_i L_i = |\mathcal{M}|$, and hence the algorithm runs in $O(n + \alpha|\mathcal{M}|)$ worst case time.

With one simple optimization the worst case can be improved to $O(\min\{\alpha|\mathcal{M}|, nm\})$ (improving also the average time a bit). When computing the current list L_i, Simple algorithm may inspect some text characters several times, because the subsequent text positions stored in L_{i-1} can be close to each other, in particular, they can be closer than $\alpha + 1$ positions. In this case the $\alpha + 1$ wide text windows will overlap, and same text positions are inspected more than once. Adding a simple safeguard to detect this, each value in the list L_i can be computed in $O(\alpha)$ *worst case* time, and in $O(1)$ best case time. In particular, if $|\mathcal{M}| = O(nm)$, then the overlap between the subsequent text windows is $O(\alpha)$, and each value of L_i is computed in $O(1)$ time. This results in $O(nm)$ worst case time. The average case is improved as well. Algorithm 3 shows the pseudo code, including this improvement.

Alg. 3 SDP-simple($T, n, P, m, \delta, \alpha$).

```
1   $h \leftarrow 0$
2   **for** $j \leftarrow 0$ **to** $n - 1$ **do**
3       **if** $|t_j - p_0| \leq \delta$ **then**
4           $L[h] \leftarrow j$; $h \leftarrow h + 1$
5   **for** $i \leftarrow 1$ **to** $m - 1$ **do**
6       $pn \leftarrow h$; $h \leftarrow 0$; $L[pn] = n - 1$
7       **for** $j \leftarrow 0$ **to** $pn - 1$ **do**
8           **for** $j' \leftarrow L[j] + 1$ **to** $\min(L[j + 1], L[j] + \alpha + 1)$ **do**
9               **if** $|t_{j'} - p_i| \leq \delta$ **then**
10                  $L'[h] \leftarrow j'$; $h \leftarrow h + 1$
11                  **if** $i = m - 1$ **then** report match
12      swap($L, L'$)
```

Consider now the average case. List L_0 is computed in $O(n)$ time. The length of this list is $O(n\delta/\sigma)$ on average. Hence the list L_1 is computed in $O(\alpha n\delta/\sigma)$ average time, resulting in a list L_1, whose average length is $O(n\delta/\sigma \times \alpha\delta/\sigma)$. In general, computing the list L_i takes

$$O(\alpha|L_{i-1}|) = O(n\alpha^i(\delta/\sigma)^i) = O(n(\alpha\delta/\sigma)^i)$$

average time. This is exponentially decreasing if $\alpha\delta/\sigma < 1$, i.e., if $\alpha < \sigma/\delta$, and hence, summing up, the total average time is $O(n)$.

5.1 Sublinear average case

In this section we show how the average case time of Simple can be improved. The basic observation is that while building the list L_0 not all δ-matches need to be inserted, but rather only those that have hope to be extended to a complete match of the whole pattern. In other words, some of the δ-matches can be skipped. This can be achieved using Boyer–Moore–Horspool (BMH) (Horspool 1980) strategy. We therefore build the list L_0 using the BMH approach (*filtering*), and then continue with plain Simple to compute the lists $L_{1\ldots m-1}$. This can be seen as a *verification* phase.

In what follows, we build L_0 scanning the text backwards. Lists $L_{1\ldots m-1}$ are built as before, using Simple. We first need the following definition:

$$S[c] = \min\{i, m | p_i =_\delta c\}.$$

This implies that if $S[t_j] \neq m$, then $p_{S[t_j]} =_\delta t_j$. This gives us a *shifting* rule. Assume now that p_0 is aligned with t_j. We then execute the following algorithm:

1. If $|p_0 - t_j| \leq \delta$, then put j into the list L_0.
2. Check $t_{j-\alpha-1\ldots j-1}$ from right to left, computing $s = \text{argmin}_i\{S[t_{j-i}] | i \in [1\ldots\alpha + 1]\}$. If several values of i give the same minimum shift value, return the smallest i.
3. Shift the pattern with $j \leftarrow j-(S[t_{j-s}] + s)$ to align t_{j-s} with $p_{S[t_{j-s}]}$.
4. If $j \geq 0$, then go to 1.
5. Pass the computed list L_0 to Simple, and compute lists $L_{1\ldots m-1}$.

The core of the algorithm is the step 2. We scan the text window $t_{j-\alpha-1\ldots j-1}$. If some occurrence overlaps this window, then some pattern character must δ-match one of the characters in this window. We therefore compute the smallest shift to align some δ-matching pattern character to one of these text characters. If such character does not exist, then the pattern occurrence cannot overlap this window, and the whole pattern is shifted past the window, i.e., the shift is $m + \alpha + 1$ characters.

The text scanning is performed backwards, as we want to put the *starting* positions (instead of ending positions) of the possible occurrences into the list L_0. The only reason for this is to be compatible with Simple algorithm. Algorithm 4 gives the pseudo code.

Alg. 4 SDP-simple-compute-$L_0(T, n, P, m, \delta, \alpha)$.

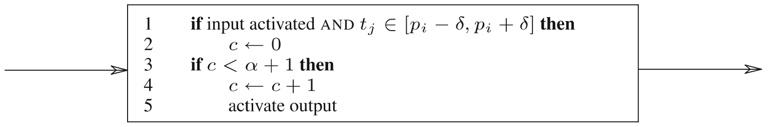
```
for $i \leftarrow 0$ to $\sigma - 1$ do $S[i] \leftarrow m$
for $i \leftarrow m - 1$ downto $0$ do
    for $j \leftarrow \max(0, p_i - \delta)$ to $\min(\sigma - 1, p_i + \delta)$ do $S[j] \leftarrow i$
$h \leftarrow 0; j \leftarrow n - m + 1$
while $j \geq 0$ do
    if $|t_j - p_0| \leq \delta$ then
        $L'[n - h - 1] \leftarrow j; h \leftarrow h - 1$
    $k \leftarrow \alpha; s \leftarrow m$
    for $i \leftarrow 0$ to $\alpha$ do
        if $j - i - 1 \geq 0$ AND $S[t_{j-i-1}] < s$ then
            $s \leftarrow S[t_{j-i-1}]; k \leftarrow i$
    $j \leftarrow j - (s + k + 1)$
$L[0 \ldots h - 1] \leftarrow L'[n - h \ldots n - 1]$
/* continue with Simple from row 1 */
```

As opposed to exact BMH matching, in this variant any shift requires $O(\alpha)$ prior character accesses. The average pattern shift can be lower-bounded by $O(\min(m,1/\rho))$, where ρ is the probability of a δ-matching symbol in $(\alpha + 1)$-window, that is, $\rho = 1 - (1-\delta/\sigma)^{\alpha+1}$. This probability is $O(\frac{\alpha+1}{\sigma/\delta})$ if $\alpha + 1 < \sigma/\delta$. Thus the average time for large m and small α is $O(n\alpha^2\delta/\sigma)$, which also dominates the verification phase.

6 Non-deterministic finite automata

In this section we present an algorithm based on non-deterministic finite automaton. Preliminary version of this algorithm appeared in Fredriksson and Grabowski (2006). We first review that algorithm and then improve its average case running time. The problem of the algorithm in Navarro and Raffinot (2003) is that it needs $m + (m - 1)\alpha$ bits to represent the search state. Our goal is to reduce this to $O(m\log(\alpha))$, and hence the worst case time to $O(n\lceil(m\log(\alpha))/w\rceil)$.

At a very high level, the algorithm can be seen as a novel combination of Shift-And and Shift-Add algorithms (Baeza-Yates and Gonnet 1992). The 'automaton' has two kinds of states: Shift-And states and Shift-Add states. The Shift-And states keep track of the pattern characters, while the Shift-Add states keep track of the gap length between the characters. The result is a systolic array rather than automaton; a high level description of a building



```
if input activated AND $t_j \in [p_i - \delta, p_i + \delta]$ then
    $c \leftarrow 0$
if $c < \alpha + 1$ then
    $c \leftarrow c + 1$
    activate output
```

Fig. 2 A building block for a systolic array detecting δ-matches with α-bounded gaps

block for character p_i is shown in Fig. 2. The final array is obtained by concatenating one building block for each pattern character. We call the building blocks *counters*.

To efficiently implement the systolic array in sequential computer, we need to represent each counter with as few bits as possible while still being able to update all the counters bit-parallelly.

We reserve $\ell = \lceil \log_2(\alpha + 1) \rceil + 1$ bits for each counter, and hence we can store $\lfloor w/\ell \rfloor$ counters into a single machine word. We use the value $2^{\ell-1} - (\alpha + 1)$ to initialize the counters, i.e., to represent the value 0. (This representation has been used before, e.g., in Crochemore et al. (2005).) This means that the highest bit (ℓth bit) of the counter becomes 1 when the counter has reached a value $\alpha + 1$, i.e., the gap cannot be extended anymore. Hence the lines 3–4 of the algorithm in Fig. 2 can be computed bit-parallelly as

$$C \leftarrow C + ((\sim C \gg (\ell - 1))\,\&\, msk),$$

where *msk* selects the lowest bit of each counter. That is, we negate and select the highest bit of each counter (shifted to the low bit positions), and add the result to the original counters. If a counter value is less than $\alpha + 1$, then the highest bit position is not activated, and hence the counter gets incremented by one. If the bit was activated, we effectively add 0 to the counter.

To detect the δ-matching characters we need to preprocess a table B, so that $B[c]$ has $i\ell$th bit set to 1, iff $|p_i - c| \le \delta$. We can then use the plain Shift-And step:

$$D' \leftarrow ((D \ll \ell)|1)\,\&\, B[t_j],$$

where we have reserved ℓ bits per character in D as well. Only the lowest bit of each field has any significance, the rest are only for aligning D and C appropriately. The reason is that a state in D may be activated also if the corresponding gap counter has not exceeded $\alpha + 1$. In other words, if the highest bit of a counter in C is not activated (the gap condition is not violated), then the corresponding bit in D should be activated:

$$D \leftarrow D'|((\sim C \gg (\ell - 1))\,\&\, msk).$$

The only remaining difficulty to solve is how to reinitialize (bit-parallelly) some subset of the counters to zero, i.e., how to implement the lines 1–2 of the algorithm in Fig. 2. The bit vector D' has value 1 in every field position that survived the Shift-And step, i.e., in every field position that needs to be initialized in C. Then

$$C \leftarrow C\,\&\sim(D' \times ((1 \ll \ell) - 1))$$
$$C \leftarrow C|(D' \times ((1 \ll (\ell - 1)) - (\alpha + 1)))$$

first clears the corresponding counter fields, and then copies the initial value $2^{\ell-1} - (\alpha + 1)$ to all the cleared fields.

This completes the algorithm. Algorithm 5 gives the pseudo code. Algorithm 5 runs in $O(n)$ worst case time, if $m(\lceil \log_2(\alpha + 1) \rceil + 1) \le w$. Otherwise, several machine words are needed to represent the search state, and the time grows accordingly. However, by using the well-known folklore idea, it is possible to obtain $O(n)$ average time for long patterns not fitting into a single word by updating only the "active" (i.e., non-zero) computer words. This works in $O(n)$ time on average as long as $\delta/(\sigma(1 - \delta/\sigma)^{\alpha+1}) = O(w/\log(\alpha))$. The preprocessing takes $O(m + (\sigma + \delta\sigma_P)\lceil m\log(\alpha)/w \rceil)$ time, which is $O(m + (\sigma + \delta\min\{m, \sigma\})\lceil m\log(\alpha)/w \rceil)$ in the worst case.

Alg. 5 DA-NFA($T, n, P, m, \delta, \alpha$).

```
$\ell \leftarrow \lceil \log_2(\alpha + 1) \rceil + 1$
**for** $i \leftarrow 0$ **to** $\sigma - 1$ **do** $B[i] \leftarrow 0$; $B'[i] \leftarrow 0$
**for** $i \leftarrow 0$ **to** $m - 1$ **do** $B'[p_i] \leftarrow B'[p_i] \mid (1 << (i \times \ell))$
**for** $i \leftarrow 0$ **to** $\sigma - 1$ **do if** $B'[i] \neq 0$ **then**
    **for** $j \leftarrow \max(0, i - \delta)$ **to** $\min(i + \delta, \sigma - 1)$ **do** $B[j] \leftarrow B[j] \mid B'[i]$
$msk \leftarrow 0$
**for** $i \leftarrow 0$ **to** $m - 1$ **do** $msk \leftarrow msk \mid (1 << (i \times \ell))$
$am \leftarrow (1 << (\ell - 1)) - (\alpha + 1)$
$D \leftarrow 0$; $C \leftarrow (am + \alpha + 1) \times msk$
$msk \leftarrow msk >> \ell$
$mm \leftarrow 1 << ((m - 1) \times \ell)$
**for** $i \leftarrow 0$ **to** $n - 1$ **do**
    $C \leftarrow C + ((\sim C >> (\ell - 1))\ \&\ msk)$
    $D' \leftarrow ((D << \ell) \mid 1)\ \&\ B[t_i]$
    $D \leftarrow D' \mid ((\sim C >> (\ell - 1))\ \&\ msk)$
    $C \leftarrow C\ \&\ \sim((D' << \ell) - D')$
    $C \leftarrow C \mid (D' \times am)$
    **if** $(D\ \&\ mm) = mm$ **then** report match
```

6.1 Sublinear average case

We note that the idea (Navarro and Raffinot 2003) of combining the forward matching automaton with BNDM (Navarro and Raffinot 2000) works with our algorithm as well. We briefly sketch the idea.

Denote the pattern in reverse as P^r. The set of its suffixes is $\{P^r_{i \ldots m-1} | 0 \leq i < m\}$ (note that this corresponds to the prefixes of the original pattern). The minimum length of a matching text substring is m. Assume that we are scanning the text window $t_{j \ldots j+m-1}$*backwards*. The invariant is that all occurrences that start before the position j are already reported. The algorithm matches the characters of the current window (backwards) as long as *any* of the suffixes (δ,α)-match, or we reach the beginning of the window. The algorithm remembers the longest suffix found from the window. The window is then shifted so that its starting position will become aligned with the last symbol of that suffix. This is the position of the next possible pattern occurrence. If the length of that longest suffix was ℓ, the next window to be searched is $t_{j+m-\ell \ldots j+m-1+m-\ell}$. This process is repeated until the whole text is scanned. However, if we reached the end of the window, then it is possible that there is an occurrence starting at the text position j, which must be verified.

To implement the above algorithm efficiently, we use Algorithm 5 for both the backward matching and verification. Consider first the backward matching phase. We build the automaton using P^r. For each window all the states are initialized to be active, in other words, the states corresponding to each of the suffixes of P^r is initialized to be active. This correctly models that we want to recognize every suffix of P^r ending at t_{j+m-1}. We also must remove the self-loop from the automaton, since the automaton is used for recognition, not for searching, i.e., the main Shift-And step becomes

$$D' \leftarrow (D \ll \ell)\ \&\ B[c].$$

To detect if some state is still active, we just check if D is not zero.

To verify the occurrences, we again use Algorithm 5 to scan the text from position j onwards using the original pattern P. Again the self-loop is removed from the initial state (which is the only state initialized to be active), hence the vector D must become zero after $m + (m - 1)\alpha$ steps, which is the maximum length of a pattern occurrence. This signals the end of the verification procedure.

The analysis is similar as that of plain BNDM. The differences are that we must always scan at least α characters in each window, and that the probability of a character match is not $1/\sigma$, but $\rho = 1 - (1 - \delta/\sigma)^{\alpha+1}$, hence the average time becomes $O(n\alpha\log_{1/\rho}(m)/m)$ for $m\log(\alpha) = O(w)$. Multiplying by $\lceil m\log(\alpha)/w \rceil$ we obtain $O(n\alpha\log(\alpha)\log_{1/\rho}(m)/w)$ asymptotically for large m. For m larger than $w/\log(\alpha)$ we could also use only pattern prefixes of length $w/\log(\alpha)$ in the backward search phase, resulting in $O(n\alpha\log(\alpha)\log_{1/\rho}(w/\log(\alpha))/w)$ average time.

The worst case time of this algorithm becomes quadratic, as in the worst case the length of the shift is always $O(1)$, i.e., each text character is scanned $O(m)$ times. However, there are some "standard tricks" that can be applied to combine the backward and forward (verification) scans so that either scans no text character twice (Crochemore et al. 1994; Navarro and Raffinot 2003). These work with our method as well. A somewhat simplified solution which achieves $O(1)$ accesses to any text character in the worst case is as follows. Assume that for the current window the backward scan touched more than $m/2$ text characters (i.e., it is possible, but not necessary, that the shift is less that $m/2$ characters). In this case we switch to forward matching. The window starts from the text position j, which is the next possible starting position of an occurrence. We search with the forward algorithm the text window $t_{j\ldots j+m+m+(m-1)\alpha}$, and then again switch to backward scanning, starting with text window $t_{j+m+1\ldots j+2m}$. Hence the backward scan never retraverses same text characters. The same is easily achieved for the forward scanning by saving the last scanned text position and the corresponding state vectors C and D, and resuming the search in the case of overlapped windows. This preserves the good worst case time of Algorithm 5. A more sophisticated solution is described in Navarro and Raffinot (2003), but the final result is the same.

7 Handling character classes and general gaps

We now consider the case where the gap limit can be of different length for each pattern character, and where the δ-matching is replaced with character classes, i.e., each pattern character is replaced with a set of characters.

7.1 Character classes

In the case of character classes $p_i \subset \Sigma$, and t_j matches p_i if $t_j \in p_i$. For Algorithms 2 and 3 we can preprocess a table $C[0\ldots m-1][0\ldots\sigma-1]$, where $C[i][c] := c \in p_i$. This requires $O(\sigma m)$ space and $O(\sigma\sum_i|p_i|)$ time, which is attractive for small σ, such as protein alphabet. The search algorithm can then use C to check if $t_j \in p_i$ in $O(1)$ time. For large alphabets we can use, e.g., hashing or binary search, to do the comparisons in $O(1)$ or in $O(\log(|p_i|))$ time, respectively.

Algorithm 1 is a bit more complicated, since we need to have $\mathcal{M}$ preprocessed. First compute lists $L'[c] = \{i \mid c \in p_i\}$. This can be done in one linear scan over the pattern. Then list $L[i]$ is defined as $L[i] = \{j \mid t_j \in p_i\}$. This can be computed in one linear scan over the text appending j into each list $L[i]$ where $i \in L'[t_j]$. The total time is then $O(n\delta)$, where we can consider δ as the average size of the character classes. The search algorithm can now be used as is, the only modification being that where we used $L[p_i]$ previously, we now use $L[i]$ instead (and the new definition of L).

7.2 Negative and range-restricted gaps

We now consider gaps of the form $g(a_i,b_i)$, where a_i denotes the minimum and b_i the maximum ($a_i \leq b_i$) gap length for the pattern position i. This problem variant has important applications, e.g., in protein searching (see Mehldau and Myers 1993; Myers 1996; Navarro and Raffinot 2003). General gaps were considered in Navarro and Raffinot (2003) and Pinzón and Wang (2005). This extension is easy or even trivial to handle in all our algorithms, i.e., it is equally easy to check if the formed gap length satisfies $g(a_i,b_i)$ as it is to check if it satisfies $g(0,\alpha)$. The column-wise sparse dynamic programming is a bit trickier, but still adaptable. Yet a stronger model (Mehldau and Myers 1993; Myers 1996) allows gaps of *negative* lengths, i.e., the gap may have a form $g(a_i,b_i)$ where $a_i < 0$ (it is also possible that $b_i < 0$). In other words, parts of the pattern occurrence can be overlapping in the text.

Consider first the situation where for each $g(a_i,b_i)$: (i) $a_i \geq 0$; or (ii) $b_i \leq 0$. In either case we have $a_i \leq b_i$. Handling the case (i) is just what our algorithms already do. The case (ii) is just the dual of the case (i), and conceptually it can be handled in any of our dynamic programming algorithms by just scanning the current row from right to left, and using $g(-b_i - 2, -a_i - 2)$ instead of $g(a_i,b_i)$.

The general case where we also allow $a_i < 0 < b_i$ is slightly trickier. Basically, the only modification for Algorithm 1 is that we change all the conditions of the form $0 \leq g \leq \alpha$, where g is the formed gap length for the current position, to form $a_i \leq g \leq b_i$. Note that this does not require any backtracking, even if $a_i < 0$.

Algorithm 3 can be adapted as follows. For computing the list L_i, the basic algorithm checks if any of the text characters $t_{j'+1}\ldots t_{j'+\alpha+1}$, for $j' \in L_{i-1}$ matches p_i. We modify this to check the text characters $t_{j'+a_i+1}\ldots t_{j'+b_i+1}$. This clearly handles correctly both the situations $b_i \leq 0$ and $a_i < 0 < b_i$. The scanning time for row i becomes now $O((b_i - a_i + 1)\,|L_{i-1}|)$. The average time is preserved as $O(n)$ if we now require that $(b_i - a_i + 1)\delta/\sigma < 1$. The optimization to detect and avoid overlapping text windows clearly works in this setting as well, and hence the worst case time remains $O(n + \min\{(b - a + 1)|\mathcal{M}|, nm\})$, where for simplicity we have considered that the gaps are of the same size for all rows.

8 Transposition invariance

In this section we consider transposition invariance. In this case pattern P (δ,α)-matches the text substring $t_{i_0}t_{i_1}t_{i_2}\ldots t_{i_{m-1}}$, if $p_j + \tau =_\delta t_{i_j}$ for $j \in \{0, \ldots, m - 1\}$, where $i_j < i_{j+1}$, $i_{j+1} - i_j \leq \alpha + 1$ and $\tau \in \{-\sigma + 1 \ldots \sigma - 1\}$. That is the condition is the same as before, but we now allow that the symbols can be "transposed" by some constant value. Now we also assume that the (integer) alphabet Σ is not arbitrary, but its symbols form a continuous range $0 \ldots \sigma - 1$. In MIR context transposition invariance means that the pattern and its occurrence in text can be in different key.

8.1 Transposition invariant Simple

It appears that our Simple algorithm can be modified to this setting relatively straightforwardly. We again maintain a list L_i of text positions where the pattern prefix $p_0 \ldots p_i$ matches the text substring, but this time we must also maintain the set of possible transpositions for each such text position. First notice that for any symbols p and t the transposition $\tau = t - p$ makes the symbols match exactly. Taking the δ condition into account, the set of possible transpositions becomes $\{\tau - \delta \ldots \tau + \delta\}$, i.e., for any single pair of symbols there are exactly $2\delta + 1$ allowed transpositions.

In the following we make the assumption that $2\delta + 1 \leq w$, where w is the number of bits in a machine word. In MIR applications this is practically never a restriction. We represent the set of possible transpositions as a pair $(\tau, \mathcal{T})$, where $\tau = t - p$ (the *base*) and $\mathcal{T}$ is the set of the $2\delta + 1$ possible *offsets* to the value τ. More precisely, $\mathcal{T}$ is a bitvector of $2\delta + 1$ bits. If the kth bit of $\mathcal{T}$ is set, then the transposition $\tau + k - \delta$ is valid.

Assume now that we have transpositions $(\tau_1, \mathcal{T}_1)$ and $(\tau_2, \mathcal{T}_2)$, and we want to compute the transposition $(\tau, \mathcal{T})$ that agrees with both, i.e.,

$$(\tau, \mathcal{T}) = (\tau_1, \mathcal{T}_1) \cap (\tau_2, \mathcal{T}_2).$$

If $\tau_1 = \tau_2$ then

$$(\tau, \mathcal{T}) = (\tau_2, \mathcal{T}_1 \,\&\, \mathcal{T}_2),$$

where the bit-wise & operation effectively intersects the two sets. If $|\tau_1 - \tau_2| > 2\delta$, then the intersection is an empty set, and we just set $\mathcal{T}$ to zero. Otherwise, if $|\tau_1 - \tau_2| \leq 2\delta$ the intersection can be non-empty. To compute the intersection we must first bring $\mathcal{T}_1$ and $\mathcal{T}_2$ into the same base. This is easily achieved by shifting the bitvectors. Assume that $\tau_1 < \tau_2$. Then

$$(\tau, \mathcal{T}) = (\tau_2, (\mathcal{T}_1 \gg (\tau_2 - \tau_1)) \,\&\, \mathcal{T}_2).$$

Symmetrically, if $\tau_1 > \tau_2$ we obtain

$$(\tau, \mathcal{T}) = (\tau_2, (\mathcal{T}_1 \ll (\tau_1 - \tau_2)) \,\&\, \mathcal{T}_2).$$

Let us now consider extending a (possible) prefix match. Let the current pattern position be i, and text position j. The set of candidate transpositions for this location is $(t_j - p_i, 1^{2\delta+1})$ (we use exponentiation to denote bit-repetition). This location is a prefix match, if in the previous row there are matching prefixes within α-window, and their corresponding transpositions agree with the pair $(t_j - p_i,\ 1^{2\delta+1})$. Let these transpositions be $(\tau_1, \mathcal{T}_1), \ldots, (\tau_k, \mathcal{T}_k),\ k \leq \alpha + 1$. Then the set of transpositions extending the prefix match to position (i,j) is

$$((t_j - p_i, 1^{2\delta+1}) \cap (\tau_1, \mathcal{T}_1)) \cup \cdots \cup ((t_j - p_i, 1^{2\delta+1}) \cap \cdots (\tau_k, \mathcal{T}_k)),$$

where the union $\cup$ is simply computed as bit-wise `or` of the bitvectors $\mathcal{T}$, as they are all brought to the same base by the intersection operation. Hence, assuming that $2\delta + 1 \leq w$, this computation takes $O(\alpha)$ time. If the resulting set is non-empty, we put the position j into the list L_i, just as in the Simple algorithm without transposition invariance. Algorithm 6 gives the complete pseudo code.

Alg. 6 SDP-TPI-simple($T, n, P, m, \delta, \alpha$).

```
tpm ← $\sim 0 >> (w - (2\delta + 1))$
for $j \leftarrow 0$ to $n - 1$ do
    $\tau[j] \leftarrow t_j - p_0$; $\mathcal{T}[j] \leftarrow tpm$; $\mathcal{T}'[j] \leftarrow 0$; $L[j] \leftarrow j$
$h \leftarrow n$
for $i \leftarrow 1$ to $m - 1$ do
    $pn \leftarrow h$; $h \leftarrow 0$; $L[pn] = n - 1$
    for $j \leftarrow 0$ to $pn - 1$ do
        for $j' \leftarrow L[j] + 1$ to $\min(n - 1, L[j] + \alpha + 1)$ do
            $ctpo \leftarrow t_{j'} - p_i$; $ptp \leftarrow \mathcal{T}[L[j]]$
            if $|ctpo - \tau[L[j]]| \leq 2\delta$ then
                if $\tau[L[j]] < ctpo$ then $ptp \leftarrow ptp >> (ctpo - \tau[L[j]])$ else
                if $\tau[L[j]] > ctpo$ then $ptp \leftarrow ptp << (\tau[L[j]] - ctpo)$
                $\mathcal{T}'[j'] \leftarrow \mathcal{T}'[j'] \mid (ptp \,\&\, tpm)$
                $\tau'[j'] \leftarrow ctpo$
    for $j \leftarrow 0$ to $pn - 1$ do
        $\mathcal{T}[L[j]] \leftarrow 0$
        for $j' \leftarrow L[j] + 1$ to $\min(L[j + 1], L[j] + \alpha + 1)$ do
            $\mathcal{T}[L[j']] \leftarrow 0$
            if $\mathcal{T}'[j'] \neq 0$ then
                $L'[h] \leftarrow j'$; $h \leftarrow h + 1$
                if $i = m - 1$ then report match
    swap($L, L'$); swap($\mathcal{T}, \mathcal{T}'$); swap($\tau, \tau'$)
```

The worst case time of this algorithm is $O(nm\alpha\lceil\delta/w\rceil)$. As in plain Simple, computing the list L_i takes $O(\alpha|L_{i-1}|)$ time (assuming that $\lceil\delta/w\rceil = O(1)$). However, this time the lists are longer on average. Clearly $|L_0| = n$, since pattern prefix of length 1 matches every text position. Hence computing L_1 costs $O(\alpha n)$ time, and the resulting list is of length $|L_1| = O(n\alpha\delta/\sigma)$, since the probability that two intervals intersect is upper-bounded by $(4\delta + 1)/\sigma$. In general, assuming that $\alpha\delta/\sigma < 1$, the ith list is of length

$$|L_i| = O(n(\alpha\delta/\sigma)^i).$$

This is exponentially decreasing with the above assumption. Thus the average time becomes $O(\alpha n)$.

8.2 Transposition invariant DP

We now present a basic dynamic programming solution that has better worst case complexity than the Simple algorithm. The algorithm (conceptually) maintains a matrix $D_{0\ldots m-1,0\ldots n-1}$ (but only $\alpha + 2$ columns are active at any time), where each $D_{i,j}$ is a binary vector of size $2\sigma + 1$. If the kth item of this vector is set, that is, iff $D_{i,j,k} = 1$, then $p_0\ldots p_i$ matches $t_h\ldots t_j$, for some h, with transposition $k - \sigma$. Let us define a helper matrix $\mathcal{T}$ as

$$\mathcal{T}_{i,j,k} = 1 | k \in [t_j - p_i + \sigma - \delta \ldots t_j - p_i + \sigma + \delta].$$

Now $D_{0,j}$ is easy to compute: $D_{0,j} = \mathcal{T}_{0,j}$. In general, $D_{i,j,k}$ depends on the values of the $\alpha + 1$ sized window of the previous row:

$$D_{i,j,k} = 1 | \mathcal{T}_{i,j,k} = 1 \text{ AND } \exists j' : 0 < j - j' \leq \alpha + 1 \text{ AND } D_{i-1,j',k} = 1.$$

The (almost) naïve implementation of the above recurrence would result in algorithm with $O(nm\alpha\delta)$ running time. We first remove the $O(\alpha)$ factor of the trivial algorithm, then improve the average case, and finally reduce the $O(\delta)$ factor using bit-parallelism.

Trivial algorithm implementing our recurrence for $D_{i,j,k}$ would need to scan $\alpha + 1$ vectors from the previous row. This can be avoided by using *counters* maintaining the total number of "voted" transpositions for each $(\alpha + 1)$-window:

$$C_{i,j,k} = \sum_{j'=j-\alpha}^{j} D_{i,j',k}.$$

Thus we can rewrite our main recurrence as

$$D_{i,j,k} = 1 | \mathcal{T}_{i,j,k} = 1 \text{ AND } C_{i-1,j-1,k} > 0.$$

The counters can be easily updated in $O(1)$ time per value by incremental computation:

$$C_{i,j,k} = C_{i-1,j,k} - D_{i-\alpha-1,j,k} + D_{i,j-1,k}.$$

This gives us $O(nm\delta)$ worst case time. Note that only $O(m\alpha\sigma)$ space is needed for D since only the past $O(\alpha)$ columns are needed at any time. This could be reduced to $O(m\alpha\delta)$ by using the technique we used in Sect. 8.1. Similarly C takes only $O(m\sigma)$ space, since only one column of counters is needed to be active at any time. Finally, $\mathcal{T}$ is not needed explicitly at all, we used it only as a tool for the presentation. Algorithm 7 gives the pseudo code, omitting initialization of the arrays, which are assumed to be all zero before the main loop. It also implements a cut-off trick discussed next.

Alg. 7 TPI-DP$(T, n, P, m, \delta, \alpha)$.

```
for $k \leftarrow t_0 - p_0 + \sigma - \delta$ to $k \leftarrow t_0 - p_0 + \sigma + \delta$ do $D[0][0][k] \leftarrow 1$;
$top \leftarrow m - 1$
for $i \leftarrow 1$ to $n - 1$ do
    $Dco[0] \leftarrow 1$
    for $j \leftarrow 1$ to $top$ do $Dco[j] \leftarrow 0$
    for $k \leftarrow t_i - p_0 + \sigma - \delta$ to $k \leftarrow t_i - p_0 + \sigma + \delta$ do $D[0][i \,\%\, (\alpha + 3)][k] \leftarrow 1$
    for $j \leftarrow 1$ to $top + 1$ do
        for $k \leftarrow t_{i-\alpha-2} - p_{j-1} + \sigma - \delta$ to $t_{i-\alpha-2} - p_{j-1} + \sigma + \delta$ do
            $c \leftarrow D[j-1][(i - \alpha - 2) \,\%\, (\alpha + 3)][k]$
            $D[j-1][(i - \alpha - 2) \,\%\, (\alpha + 3)][k] \leftarrow 0$
            $C[j-1][k] \leftarrow C[j-1][k] - c$
            $Cco[j-1] \leftarrow Cco[j-1] - c$
        for $k \leftarrow t_{i-1} - p_{j-1} + \sigma - \delta$ to $t_{i-1} - p_{j-1} + \sigma + \delta$ do
            $c \leftarrow D[j-1][(i-1) \,\%\, (\alpha + 3)][k]$
            $C[j-1][k] \leftarrow C[j-1][k] + c$
            $Cco[j-1] \leftarrow Cco[j-1] + c$
        if $j \leq top$ then
            for $k \leftarrow t_i - p_j + \sigma - \delta$ to $t_i - p_j + \sigma - \delta$ do
                $c \leftarrow \min(1, C[j-1][k])$
                $D[j][(i-1) \,\%\, (\alpha + 3)][k] \leftarrow c$
                $Dco[j] \leftarrow Dco[j] + c$
            if $j = m - 1$ AND $Dco[j] > 0$ then report match
    while $top \geq 1$ AND $Cco[top] = 0$ AND $Dco[top] = 0$ do $top \leftarrow top - 1$
    if $top < m - 1$ then $top \leftarrow top + 1$
```

8.2.1 Cut-off

We make the following observation: if $D_{i\ldots m-1, j-\alpha\ldots j,k} = 0$, for some i, j and all k, then $D_{i+1\ldots m-1,\, j+1,k} = 0$. This is because there is no way the recurrence can introduce any other value for those matrix cells. In other words, if $p_0\ldots p_i$ does not (δ,α)-match $t_h\ldots t_{j-j'}$ for any $j' = 0\ldots\alpha$, then the match at the position $j + 1$ cannot be extended to $p_0\ldots p_{i+1}$. This can be utilized by keeping track of the highest row number *top* of the current column j such that

$D_{top+1\ldots m-1,j-\alpha\ldots j} = 0$, and computing the next column only up to row *top* + 1. For this sake we maintain an array *Cco* so that $Cco_{i,j}$ gives the total number of "voted" transpositions for the last ($\alpha + 1$)-window:

$$Cco_{i,j} = \sum_k C_{i,j,k}.$$

This is again trivial to incrementally maintain in $O(1)$ time per computed D value. Hence after the row *top* for the column j is processed, the new value of top is computed as

$$top = \text{argmin}_i\{Cco_{i\ldots top,j} = 0\}.$$

Now consider the average time of this algorithm. Computing a single cell $D_{i,j}$ costs $O(\delta)$ time. Maintaining *top* costs only $O(n)$ time in total, since it can be incremented only by one per text symbol, and the number of decrements cannot be larger than the number of increments. The average time of this algorithm also depends on the average value of *top*, i.e., the total time is $O(n \text{ avg}(top)\ \delta)$. For p_0 the probability of a match for any text position is obviously 1 (and *top* is at least 1). For rows $i > 0$ the probability that $\mathcal{T}_{i,j}$ intersects with $D_{i-1,j-1\ldots j-\alpha-1}$ is upper bounded by

$$\rho = 1 - (1 - ((4\delta + 1)/\sigma))^{\alpha+1}.$$

Hence the expected length of a prefix match is at most

$$\sum_{i=0}^{\infty} \rho^i = \frac{1}{(1 - \frac{4\delta+1}{\sigma})^{\alpha+1}},$$

i.e., $\text{avg}(top) = O\left(\frac{1}{(1-\delta/\sigma)^{\alpha+1}}\right)$, and the average time becomes $O\left(n\frac{\delta}{(1-\delta/\sigma)^{\alpha+1}}\right)$. It is easy to show that this is $O\left(n\frac{\delta}{1-\delta(\alpha+1)/\sigma}\right)$ if $\alpha + 1 < \sigma/\delta$.

8.2.2 Bit-parallel algorithm

We note that the $O(\delta)$ factor can be easily reduced to $O(\lceil\delta\log(\alpha)/w\rceil)$, which is practically $O(1)$ in MIR applications. To see this, note that the counter values cannot exceed $\alpha + 1$, so we can pack $O(w/\log(\alpha))$ counters into a single computer word. All the inner loops (involving $2\delta + 1$ iterations) can then be computed parallelly, updating $O(w/\log(\alpha))$ counters in $O(1)$ time. The only non-trivial detail is the computation of minima of two sets of counters (parallelization of the line 19 of Algorithm 7), but the solution exists (Paul and Simon 1980), and is reasonably well-known. Note that for realistic assumptions (for MIR data) of $(4\delta + 1)\alpha < c\sigma$, for some constant $c < 1$, and for $\delta\log(\alpha) = O(w)$, this variant achieves $O(n)$ time on average. However, in practice δ is often so small that the benefit of this parallelization is negligible.

9 Experimental results

We have run experiments to evaluate the performance of our algorithms. The experiments were run on Pentium4 2 GHz with 512 MB of RAM, running GNU/Linux 2.4.18 operating system. We have implemented all the algorithms in C, and compiled with `icc 7.0`.

We first experimented with (δ,α)-matching, which is an important application in MIR. For the text we used a concatenation of 7,543 music pieces, obtained by extracting the pitch values from MIDI files. The total length is 1,828,089 bytes. The pitch values are in the range

[0…127]. This data is far from random; the six most frequent pitch values occur 915,082 times, i.e., they cover about 50% of the whole text, and the total number of different pitch values is just 55. A set of 100 patterns were randomly extracted from the text. Each pattern was then searched for separately, and we report the average user times. Figure 3 shows the timings for different pattern lengths. The timings are for the following algorithms:

DP: Plain Dynamic Programming algorithm (Crochemore et al. 2002);
DP Cut-off: "Cut-off" version of DP (as in Cantone et al. (2005b);
SDP RW: Basic Row-Wise Sparse Dynamic Programming;
SDP RW fast: Binary search version of SDP;
SDP RW fast PP: linear preprocessing time variant of SDP RW fast (Algorithm 1);
SDP CW: Column-Wise Sparse Dynamic Programming (Algorithm 2);
Simple: Simple algorithm (Algorithm 3);
BMH + Simple: BMH followed by Simple algorithm (Algorithm 4);
BP Cut-off: Bit-Parallel Dynamic Programming (Fredriksson and Grabowski 2006);
NFA α: Non-deterministic finite automaton, forward matching variant (Navarro and Raffinot 2003), using $O(\alpha)$ bits per symbol;
NFA $\log(\alpha)$: Non-deterministic finite automaton, forward matching variant (Algorithm 5), using $O(\log(\alpha))$ bits per symbol.

We also implemented the SDP RW variant with $O(\sqrt{\delta n})$ worst case preprocessing time, but this was not competitive in practice, so we omit the plots.

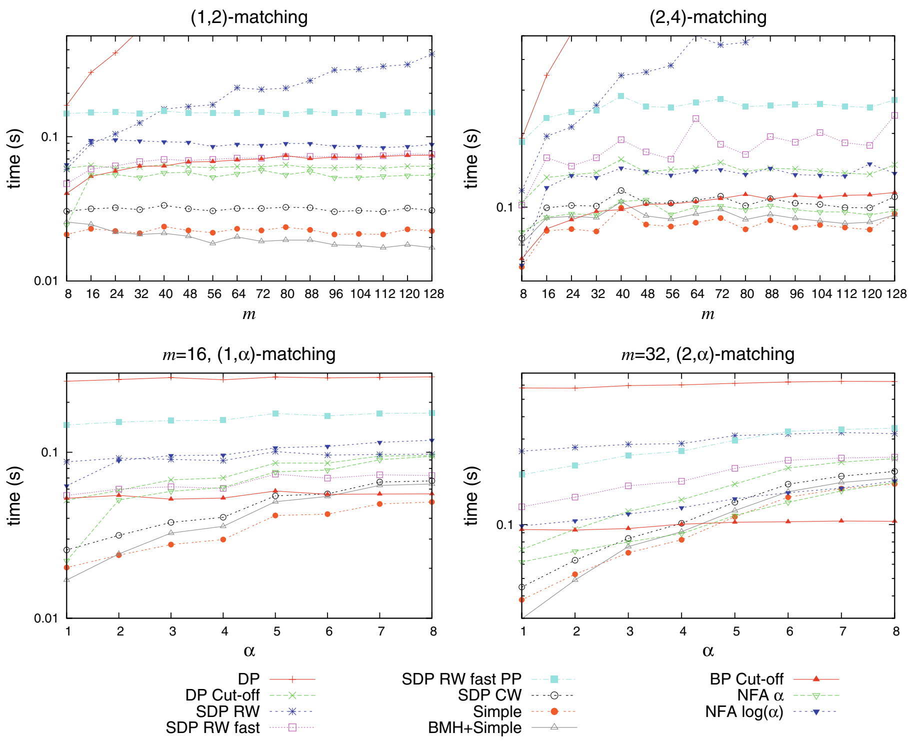


Fig. 3 Running times for (δ,α)-matching in seconds for $m = 8\ldots128$ (top) and for $\alpha = 1\ldots8$ (bottom). Note the logarithmic scale

SDP is clearly better than DP, but both show the dependence on m. The "cut-off" variants remove this dependence. The linear time preprocessing variant of the SDP "cut-off" is always slower than the plain version. This is due to the small effective alphabet size of the MIDI file. For large alphabets with flat distribution the linear time preprocessing variant quickly becomes faster as m (and hence the pattern alphabet) increases. The column-wise SDP algorithm and especially Simple algorithm are very efficient, beating everything else if δ and α are reasonably small. For very small δ and α and moderate m the BMH variant of Simple is even faster. For large (δ,α) the differences between the algorithms become smaller. The reason is that a large fraction of the text begins to match the pattern. However, this means that these large parameter values are less interesting for this application. The bit-parallel algorithm (Navarro and Raffinot 2003) is competitive but suffers from requiring more bits than fit into a single machine word, yet Algorithm 5 is even slower, besides having more efficient packing. This is attributed to the additional (constant time per text character) overhead due to the more complex packing.

9.1 Transposition invariance

We also experimented with the transposition invariant algorithms. The following algorithms were tested:

BF-Simple: Plain Simple executed $O(\sigma)$ times;
Simple: Transposition invariant Simple (Algorithm 6);

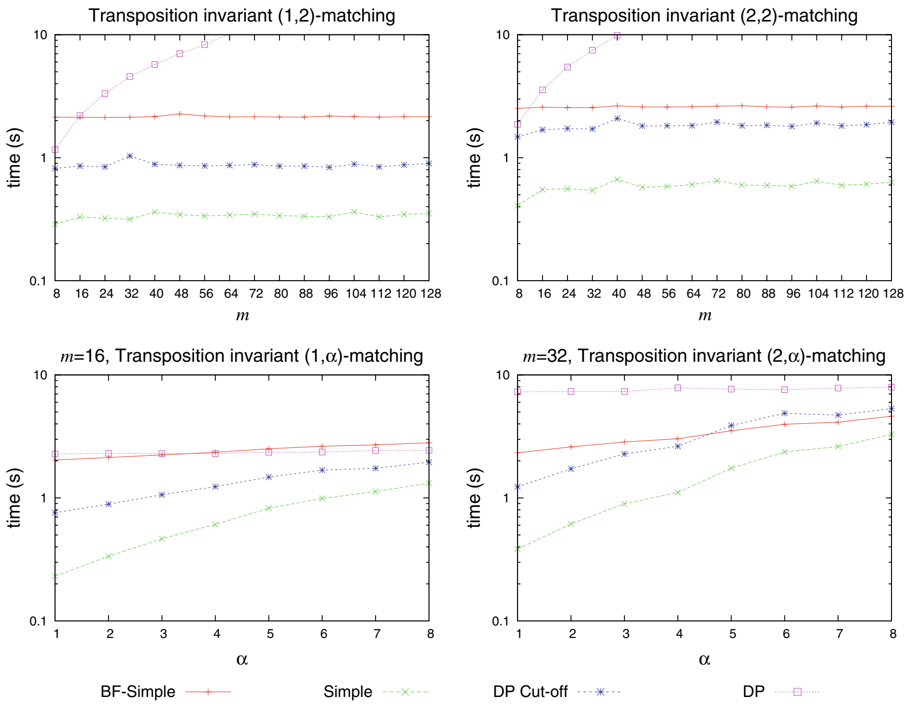


Fig. 4 Running times for transposition invariant (δ,α)-matching in seconds for $m = 8\ldots128$ (top) and for $\alpha = 1\ldots8$ (bottom). Note the logarithmic scale

DP: (Transposition invariant) Dynamic Programming algorithm;
DP Cut-off: "Cut-off" version of DP (Algorithm 7).

The results are shown in Fig. 4. In this case Simple is again clear winner, despite of the theoretical superiority of DP Cut-off. For large α DP (Cut-off) would eventually beat Simple, but in practical applications such large parameters are not interesting.

9.2 PROSITE patterns

We also ran preliminary experiments on searching PROSITE patterns from a 5 MB file of concatenated proteins. The PROSITE patterns include character classes and general bounded gaps. Searching 1,323 patterns took about 0.038 s per pattern with Simple, and about 0.035 s with NFA. Searching only the short enough patterns that can fit into a single computer word (and hence using specialized implementation), the NFA times drops to about 0.025 s. However, we did not implement the backward search version, which is reported to be substantially faster in most cases (Navarro and Raffinot 2003). Finally, note that the time for Simple would be unaffected even if the gaps were negative, since only the magnitude of the gap length affect the running time.

10 Conclusions

We have presented new efficient algorithms for string matching with bounded gaps and character classes. Some of those algorithms are designed to work under transposition invariance. Besides having theoretically good worst and average case complexities, the algorithms are shown to work well in practice. Finally, despite that the algorithms were designed for MIR, they have important applications in MB as well. In particular, we can handle even negative gaps efficiently.

Acknowledgment K. Fredriksson was supported by the Academy of Finland. This work was done while the author worked in the Department of Computer Science and Statistics, University of Joensuu.

References

Baeza-Yates, R. A., & Gonnet, G. H. (1992). A new approach to text searching. *Communications of the ACM, 35*(10), 74–82.

Cantone, D., Cristofaro, S., & Faro, S. (2005a). An efficient algorithm for δ-approximate matching with α-bounded gaps in musical sequences. In *Proceesings of WEA'05. LNCS* (Vol. 3503, pp. 428–439). Springer.

Cantone, D., Cristofaro, S., & Faro, S. (2005b). On tuning the (δ,α)-sequential-sampling algorithm for δ-approximate matching with α-bounded gaps in musical sequences. In *Proceedings of ISMIR'05.*

Crochemore, M., Czumaj, A., Gąsieniec, L., Jarominek, S., Lecroq, T., Plandowski, W., & Rytter, W. (1994). Speeding up two string matching algorithms. *Algorithmica, 12*(4–5), 247–267.

Crochemore, M., Iliopoulos, C., Makris, C., Rytter, W., Tsakalidis, A., & Tsichlas, K. (2002). Approximate string matching with gaps. *Nordic Journal of Computing, 9*(1), 54–65.

Crochemore, M., Iliopoulos, C., Navarro, G., Pinzon, Y., & Salinger, A. (2005). Bit-parallel (δ,γ)-matching suffix automata. *Journal of Discrete Algorithms (JDA), 3*(2–4), 198–214.

Fredriksson, K., & Grabowski, S. (2006). Efficient bit-parallel algorithms for (δ,α)-matching. In *Proceedings of WEA'06. LNCS* (Vol. 4007, pp. 170–181). Springer.

Horspool, R. N. (1980). Practical fast searching in strings. *Software Practice and Experience, 10*(6), 501–506.

Johnson, D. B. (1982). A priority queue in which initialization and queue operations take $O(\log \log D)$ time. *Mathematical Systems Theory, 15*, 295–309.

Mäkinen, V. (2003). *Parameterized approximate string matching and local-similarity- based point-pattern matching*. PhD thesis, Department of Computer Science, University of Helsinki.

Mäkinen, V., Navarro, G., & Ukkonen, E. (2005). Transposition invariant string matching. *Journal of Algorithms, 56*(2), 124–153.

Mehldau, G., & Myers, G. (1993). A system for pattern matching applications on biosequences. *Computer Application in the Bioscience, 9*(3), 299–314.

Myers, G. (1996). Approximate matching of network expression with spacers. *Journal of Computational Biology, 3*(1), 33–51.

Navarro, G., & Raffinot, M. (2000). Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM Journal of Experimental Algorithmics (JEA), 5*(4). http://www.jea.acm.org/2000/NavarroString.

Navarro, G., & Raffinot, M. (2003). Fast and simple character classes and bounded gaps pattern matching, with applications to protein searching. *Journal of Computational Biology, 10*(6), 903–923.

Paul, W., & Simon, J. (1980). Decision trees and random access machines. In *ZUERICH: Proceedings of the Symposium Logik und Algorithmik* (pp. 331–340).

Pinzón, Y. J., & Wang, S. (2005). Simple algorithm for pattern-matching with bounded gaps in genomic sequences. In *Proceedings of ICNAAM'05* (pp. 827–831).