UNIVERSITY OF JOENSUU

DEPARTMENT OF COMPUTER SCIENCE

Report Series A

Efficient Algorithms for (δ, α)-matching

Kimmo Fredriksson and Szymon Grabowski

A-2005-2

ACM F.2.2, H.3.3

ISBN 952-458-758-0

ISSN 0789-7316

Efficient Algorithms for (δ, α)-matching

Kimmo Fredriksson[*]
Department of Computer Science
University of Joensuu
kfredrik@cs.joensuu.fi

Szymon Grabowski
Computer Engineering Department
Technical University of Łódź
sgrabow@kis.p.lodz.pl

Abstract

We consider the following string matching problem. Pattern $p_0p_1p_2 \ldots p_{m-1}$ (δ, α)-matches the text substring $t_{i_0}t_{i_1}t_{i_2} \ldots t_{i_{m-1}}$, if $|p_j - t_{i_j}| \leq \delta$ for $j \in \{0, \ldots, m-1\}$, where $i_j < i_{j+1}$, and $i_{j+1} - i_j \leq \alpha + 1$. The task is then to find all text positions i_{m-1} that (δ, α)-match the pattern. The best previously known algorithm for this string matching problem runs in time $O(nm)$ using $O(\alpha m)$ space, where n is the length of the text and m is the length of the pattern. We develop several algorithms that improve the previous bounds. Our techniques are based on novel combinations of sparse dynamic programming, pre-emptying the computation early, bit-parallelism and finite automata. Our complexities vary from $O(n + |\mathcal{M}|)$ worst case time to $O(n)$ average time, where $\mathcal{M} = \{(i,j) \mid p_i =_\delta t_j\}$, using dynamic programming based approach. With bit-parallel dynamic programming we obtain worst case time $O(n\delta + \lceil n/w \rceil m)$ where w is the number of bits in machine word. Some of the ideas can be combined to obtain good average case complexities while retaining good worst case complexities. With bit-parallelism and nondeterministic finite automata we obtain $O(n\lceil m \log_2(\alpha)/w \rceil)$ worst case time, which is $O(n)$ for short patterns. Finally, we show that all our algorithms can be generalized to solve several other problem variants. Our experimental results show that the algorithms work extremely well in practice.

Keywords: approximate string matching, music information retrieval, protein matching, gaps, negative gaps, sparse dynamic programming, bit-parallelism, nondeterministic finite automata.

ACM Classification: F.2.2 [Analysis of algorithms and problem complexity]: Non-nunmerical algorithms and problems — *Pattern matching, Computations on discrete structures*; H.3.3 [Information storage and retrieval]: Information Search and Retrieval — *Search process.*

1 Introduction

Background and problem setting. Many notions of approximateness have been proposed in string matching literature, usually motivated by some real problems. One of seemingly underexplored problems with applications in music information retrieval and molecular biology is (δ, α)-matching [5]. In this problem, the pattern $p_0p_1 \ldots p_{m-1}$ is allowed to match a substring of the text $t_0t_1 \ldots t_{n-1}$ with α-limited gaps, and the respective pairs of matching characters may be different, only if their numerical values do not differ by more than δ. Translating this model into a music (melody seeking) application, we say can that we allow for small distortions of the original melody because the (presumably unskilled) human user may sing or whistle the melody imprecisely. The gaps, on the other hand, allow to skip over ornamenting notes (e.g., arpeggios), which appear especially in classical music. Other assumptions here, that is, monophonic melody and using pitch values only (without note durations), are reasonable in most practical cases. In biology, somewhat relaxed version of the α-matching problem is important for protein matching,

[*]Supported by the Academy of Finland, grant 202281.

especially together with allowing for matching classes of characters. Fortunately, in all the new algorithms we are going to present in this paper, δ-matching can be straightforwardly changed into matching classes of characters, without any penalty in the complexities if the size of the character class is on the order of δ.

Previous work. There are not many algorithms for (δ, α)-matching. The ones we are aware of are: by Crochemore et al. [5] and Cantone et al. [3, 4]. The algorithms in [5, 3] need $O(nm)$ time in the worst and the average case. The advantage of the algorithm by Cantone et al. is that it needs $O(\alpha m)$ space as opposed to $O(nm)$ needed by the previous algorithm. Also, Cantone et al. claim that their algorithm is faster in practice (and give timings), but our experiments, as the reader will see, do not confirm it. However, their algorithm have the advantage that it can be extended to find all pattern occurrences, instead of only the positions where the occurrence ends. This needs more time, however. The algorithm in [4] improves the average case of the one in [3] to $O(n)$.

For the α-matching with classes of characters there exists an efficient bit-parallel solution [12]. In this algorithm the gap limits for each pattern character may be of different length, in particular, it is assumed that for many characters it is zero. We note that this algorithm can be generalized to handle (δ, α)-matching, but the time complexity becomes $O(\alpha mn/w)$ in the worst case, where w is the length of the machine word. For small α the algorithm can be made to run in $O(n)$ time on average.

Note that there exist algorithms for the harder problem variant where transposition invariance and character insertions, substitutions or mismatches are allowed together with (δ, α)-matching [9].

Our results. The number of distinct symbols in the pattern and in the text are denoted by σ_{P} and σ_{T}, respectively, and $\sigma_{\mathrm{P}\cap\mathrm{T}}$ denotes the number characters that occur both in P and T simultaneously. Note that $\sigma_{\mathrm{P}\cap\mathrm{T}} \le \sigma_{\mathrm{P}}, \sigma_{\mathrm{T}}, m$. Moreover, let $\mathcal{M} = \{(i,j) \mid |p_i - t_j| \le \delta\}$. The number of bits in machine word is w. Our results can then be summarized as follows:

Plain dynamic programming with 'cut-off' trick: $O(n)$ and $O(nm)$ average and worst case complexities, respectively.

Sparse dynamic programming: $O(n + |\mathcal{M}|)$ time, where $\mathcal{M} = \{(i,j) \mid |p_i - t_j| \le \delta\}$.

Row-wise sparse dynamic programming with 'cut-off' trick: $O(n(\delta\sigma_{\mathrm{P}\cap\mathrm{T}}/\sigma_{\mathrm{T}} + 1))$ average case time, and $O((n\delta + |\mathcal{M}|)\log\log n)$ worst case time.

Column-wise sparse dynamic programming: $O(m+n)$ average case time, and $O(m+n+|\mathcal{M}|)$ worst case time.

Bit-parallel dynamic programming: $O(\lceil n/w \rceil \sigma_{\mathrm{P}} + n(\delta\sigma_{\mathrm{P}\cap\mathrm{T}}/\sigma_{\mathrm{T}} + 1))$ and $O(n\delta + \lceil n/w \rceil m)$ average and worst case time, respectively.

Simple algorithm: $O(n)$ average case time, and $O(\min\{\alpha|\mathcal{M}|, nm\})$ worst case time. (But in practice this is one of the best alternatives).

Nondeterministic finite automaton: $O(n\lceil m \log_2(\alpha)/w \rceil)$ worst case time, i.e. $O(n)$ worst case time for $m = O(w/\log_2(\alpha))$.

Some of these bounds can still be improved. In particular, we show how to improve $O(n\delta)$ and $O(n\delta\sigma_{\mathrm{P}\cap\mathrm{T}}/\sigma_{\mathrm{T}})$ terms to $O(n)$. In some cases this makes the worst case search complexity higher. The nondeterministic finite automaton based algorithm can be improved to run in sublinear time on average. These algorithms can be generalized to handle some other problem variants as well, as discussed in Sec. 10.

Alg. 1 DA-dp($T, n, P, m, \delta, \alpha$).

```
for $i \leftarrow 0$ to $m-1$ do $D_{i,0} \leftarrow -1$
if $|p_0 - t_0| \le \delta$ then $D_{0,0} \leftarrow 0$
for $i \leftarrow 0$ to $m-1$ do
    for $j \leftarrow 1$ to $n-1$ do
        if $|p_i - t_j| \le \delta$ AND ($i = 0$ OR $D_{i-1,j-1} \ge 0$) then
            $D_{i,j} \leftarrow j$
            if $i = m-1$ then report match
        else if $D_{i,j-1} \ge j - \alpha$ then $D_{i,j} \leftarrow D_{i,j-1}$
        else $D_{i,j} \leftarrow -1$
```

Throughout the paper we assume $\sigma = O(n)$. Efficient variations of the presented algorithms for the case of large alphabets may require different preprocessing and modified search routines.

Note added to the report. This report is a significantly extended version of our paper [7]. When writing it we did not know the paper [4] where an idea analogous to the one described in Sec. 3 was presented. Still, in details our algorithms differ.

2 Preliminaries

Let the pattern $P = p_0 p_1 p_2 \dots p_{m-1}$ and the text $T = t_0 t_1 t_2 \dots t_{n-1}$ 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 and in the text are denoted by σ_{P} and σ_{T}, respectively. Moreover, we use $\sigma_{\text{P}\cap\text{T}}$ to denote the number characters that occur both in P and T simultaneously. Note that $\sigma_{\text{P}\cap\text{T}} \le \sigma_{\text{P}}, \sigma_{\text{T}}, m$.

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 $t_{i_0} t_{i_1} t_{i_2} \dots t_{i_{m-1}}$, if $p_j =_\delta t_{i_j}$ for $j \in \{0, \dots, m-1\}$, where $i_j < i_{j+1}$, and $i_{j+1} - i_j \le \alpha + 1$. 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 [5, 3]:

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

In other words, if $D_{i,j} = j$, then the pattern prefix $p_0 \dots p_i$ has an occurrence ending at text character t_j, i.e. $p_i =_\delta t_j$ and the prefix $p_0 \dots p_{i-1}$ 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 $D_{0\dots m-1, 0\dots n-1}$, and reports an occurrence ending at position j whenever $D_{m-1,j} = j$. This is simple to implement, and the algorithm runs in $O(mn)$ time using $O(mn)$ space. Alg. 1 gives the pseudo code.

3 Dynamic programming with cut-off trick

It should be stressed that both cited algorithms from the literature have quadratic time complexity even in the best case. We propose two algorithms which are also $O(nm)$ time in the worst case, but are faster in practice. The first of them has $O(n)$ time in the average case, the second runs in $O(n + |\mathcal{M}|)$ worst case time, where $\mathcal{M} = \{(i, j) \mid p_i =_\delta t_j\}$ (note that $|\mathcal{M}| = O(nm)$ in the worst case).

Alg. 2 DA-dpco$(T, n, P, m, \delta, \alpha)$.

```
for $i \leftarrow 0$ to $m-1$ do $D'_i \leftarrow -1$
if $|p_0 - t_0| \le \delta$ then $D'_0 \leftarrow 0$
$top \leftarrow m-1$
for $j \leftarrow 1$ to $n-1$ do
    for $i \leftarrow 0$ to $top$ do
        if $|p_i - t_j| \le \delta$ AND $(i = 0$ OR $D'_{i-1} \ge 0)$ then
            $D_i \leftarrow j$
            if $i = m-1$ then report match
        else if $D'_i \ge j - \alpha$ then $D_i \leftarrow D'_i$
        else $D_i \leftarrow -1$
    while $top \ge 0$ AND $D_{top} = -1$ do $top \leftarrow top - 1$
    if $top < m-1$ then $top \leftarrow top + 1$
    $Dt \leftarrow D$; $D \leftarrow D'$; $D' \leftarrow Dt$
```

As already mentioned in [5] the space requirement of Alg. 1 can be made $O(n)$ by noticing that the computation of the current row of D depends only on the previous row. However, it is also true that the computation of the current *column* of D depends only on the previous column. This can be used to make the space complexity just $O(m)$, by computing the matrix column-wise, instead of row-wise, and storing only the current and previous columns.

The authors of the cited work [5] note that the average time can be made just $O(n)$, but they do know explain explicitly how, so we do it now. A similar algorithm was obtained in [4], where the authors apply the same idea to a different but related algorithm.

We make the following observation: if $D_{i\ldots m-1,j} = -1$, for some i, j, then $D_{i+1\ldots m-1,j+1} = -1$. 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 $t_h \ldots t_{j-k}$ for any $k = 0 \ldots \alpha$, 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 $D_{top,j} \ne -1$, and computing the next column only up to row $top + 1$. More formally, we define

$$top_j = \text{argmax}_i \{D_{i,j-1} = -1 \text{ AND } D_{i-1,j-1} \ne -1\}, \tag{2}$$

and at text position j compute the column j only up to row top_j. We call this a *cut-off* trick. This technique was first used (in a different context) by Ukkonen [14].

Alg. 2 gives the pseudo code to implement these two tricks. The space complexity is clearly $O(m)$, and we show that the average case time complexity is $O(n)$. First consider the time taken for keeping track of the last active row *top*. For each text position *top* can increase only by at most 1, giving a total of $O(n)$ increments. While *top* can decrease by $O(m)$ for a processed column, which costs $O(m)$ time, the total number of decrements cannot be larger than the number of increments plus $m-1$, so the amortized cost of decrements is at most $O(n)$ as well. In total *top* can be maintained in $O(n)$ worst case time during the whole computation.

We now show that the average value of *top* is $O(1)$, which gives a total $O(n)$ average time. The average value of *top* is the same as the average length $i+1$ of the longest prefix $p_0 \ldots p_i$ that matches at some text position. The probability that $a =_\delta b$ for some $a, b \in \Sigma$ is $O(\delta/\sigma)$ assuming uniform Bernoulli model of probability. We assume that δ/σ is constant here. Hence the probability that $p_0 \ldots p_i$ (δ, α)-matches $t_h \ldots t_j$ is

$$Pr(i) = O((1 - (1 - \delta/\sigma)^{\alpha+1})^i (\delta/\sigma)), \tag{3}$$

and the expected value of top is

$$\sum_{i=0}^{m-1} i \; Pr(i) \;=\; O\left(\sum_{i=0}^{\infty} i \; Pr(i)\right) \;=\; \frac{\delta}{\sigma(1 - \delta/\sigma)^{\alpha+1}}, \tag{4}$$

Alg. 3 DA-sdp-preprocess($T, n, P, m, \delta, \alpha$).

```
for $i \leftarrow 0$ to $m-1$ do $A[p_i] \leftarrow 0$
for $i \leftarrow 0$ to $m-1$ do
    if $A[p_i] = 0$ then
        $A[p_i] \leftarrow 1$
        for $j \leftarrow -\delta$ to $\delta$ do
            $c \leftarrow \max(0, \min(p_i + j, \sigma - 1))$
            $L'[c] \leftarrow L'[c] \cup \{p_i\}$
for $i \leftarrow 0$ to $n-1$ do
    for $j \leftarrow 0$ to $|L'[t_i]| - 1$ do
        $c \leftarrow L'[t_i][j]$
        $L[c] \leftarrow L[c] \cup \{i\}$
return $L$
```

as the sum is a geometric series, which does not depend on m, and is $O(1)$ for constant α. A similar analysis can be found in [4] for the Sequential Sampling algorithm.

Note that the average case analysis may have little practical value for music, which is far from random and most of its pitch alphabet is hardly ever used. Hence, practical evaluation of this algorithm (see Sec. 11) will make more sense.

4 Sparse dynamic programming

We now aim at good worst case complexity. From the recurrence of D it is clear that the interesting computation in Alg. 1 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 $\mathcal{M} = \{(i,j) \mid p_i =_\delta t_j\}$ 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 $t_{h...j}$. The new recurrence is

$$d_{i,j} = \begin{cases} j & (i-1,j') \in \mathcal{M} \text{ AND } 0 < j - j' \le \alpha + 1 \text{ AND } d_{i-1,j'} \neq -1 \\ -1 & \text{otherwise} \end{cases} \tag{5}$$

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. We first compute a table L', such that for all $c \in \Sigma$ the list $L'[c]$ contains all the characters p_i that satisfy $p_i =_\delta c$. Using this table another table L is computed so that $L[c] = \{j \mid c =_\delta t_j\}$. This can be done by scanning through the text, and adding to $L[c]$ the index j for all $c \in L'[t_j]$. This takes $O(m + \sigma + \delta\sigma_{\text{P}} + \delta n) = O(\delta n)$ worst case time and space. Note that $|\mathcal{M}| = O(mn)$ in the worst case. The probability that two characters δ-match is $O(\delta/\sigma)$, and hence the expected number of matching pattern characters for each text character is $O(\delta\sigma_{\text{P}\cap\text{T}}/\sigma_{\text{T}})$. Therefore, the average case complexity of the preprocessing is $O(n(\delta\sigma_{\text{P}\cap\text{T}}/\sigma_{\text{T}} + 1))$. We will use L as a terse representation of $\mathcal{M}$. Note that $L[p_i]$ corresponds to the set $\{(i,j) \mid p_i =_\delta t_j\}$ for a fixed i, i.e. $L[p_i] = \{j \mid p_i =_\delta t_j\}$. Alg. 3 gives the pseudo code for computing L. The text positions in $L[c]$ are computed in increasing order, that is $L[c][h] < L[c][h+1]$, which will be crucial for computing d.

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 $j - j' \le \alpha + 1$, 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 $L[p_i][h] - d_{i-1,k} \le \alpha + 1$ 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 $L[p_i][h]$. Alg. 4 gives the pseudo code. The algorithm processes each $(i,j) \in \mathcal{M}$ in $O(1)$ time, and hence runs in $O(n + |\mathcal{M}|)$ worst case time.

Alg. 4 DA-sdp$(T, n, P, m, \delta, \alpha)$.

```
$L \leftarrow$ DA-sdp-preprocess$(T, n, P, m, \delta, \alpha)$
for $i \leftarrow 0$ to $m - 1$ do
    for $j \leftarrow 0$ to $|L[p_i]| - 1$ do $D_{i,j} \leftarrow -1$
for $i \leftarrow 0$ to $|L[p_0]| - 1$ do $D_{0,i} \leftarrow L[p_0][i]$
for $i \leftarrow 1$ to $m - 1$ do
    $c \leftarrow p_i$; $pc \leftarrow p_{i-1}$; $k \leftarrow 0$; $h \leftarrow 0$
    while $h < |L[c]|$ AND $k < |L[pc]|$ do
        $j \leftarrow L[c][h]$
        do  $j' \leftarrow D_{i-1,k}$
            if $j - j' > \alpha + 1$ AND $k < |L[pc]|$ then $k \leftarrow k + 1$
        while $j - j' > \alpha + 1$ AND $k < |L[pc]|$
        if $j' < j$ AND $k < |L[pc]|$ AND $D_{i-1,k} \neq -1$ then
            $D_{i,h} \leftarrow L[c][h]$
            if $i = m - 1$ then report match
        $h \leftarrow h + 1$
```

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

```
$L \leftarrow$ DA-sdp-preprocess$(T, n, P, m, \delta, \alpha)$
for $i \leftarrow 0$ to $|L[p_0]| - 1$ do $D'_i \leftarrow L[p_0][i]$
$h \leftarrow |L[p_0]|$
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 $k < pl$ then
            $D_h \leftarrow L[c][u]$
            $h \leftarrow h + 1$
            if $i = m - 1$ then report match
        if $k < pl$ then $u \leftarrow \min\{v \mid D'_k < L[c][v], v > u\}$
    $Dt \leftarrow D$; $D \leftarrow D'$; $D' \leftarrow Dt$
```

4.1 Adapting the cut-off trick

Alg. 4 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.

Alg. 5 gives the code. It also uses the previous trick to make the space complexity smaller by storing only the current and previous rows. The total space complexity is therefore $O(n(\delta\sigma_{\mathrm{P}\cap\mathrm{T}}/\sigma_{\mathrm{T}} + 1))$.

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 per each stored value in row i of the matrix. Hence in general, computing the row i of the matrix takes time $O(|d_{i-1}| \log_2(n\delta/\sigma))$, where $|d_i|$ denotes the number of stored values in row i.

For $i > 0$ this decreases exponentially as $|d_i| = O(|d_{i-1}| \times \gamma^i)$, where $\gamma = 1 - (1 - \delta/\sigma)^{\alpha+1} < 1$ is the probability that a pattern symbol δ-matches in a text window of length α symbols. Summing up, the total average time is then dominated by the time for the two first rows, i.e. this is a geometric series and can be bounded as in Eq. (4). Hence the average search time is $O(n\delta/\sigma \log_2(n\delta/\sigma)) = O(|\mathcal{M}|/m \log_2(|\mathcal{M}|/m))$. However, the worst case search time is also increased to $O(|\mathcal{M}| \log_2(|\mathcal{M}|/m))$.

Note that this analysis assumes constant δ/σ, i.e. the time would grow if we let δ/σ grow, but the time does not depend on m.

We now consider several approaches to improve the binary search stage.

Combining linear and binary searches. The binary search stage can be improved as follows. Assume that at first we used a linear search, skipping the list by s entries in each step, until we see that we have gone past the correct item. Then the correct item must be among the last s entries in the list. This interval is binary searched with cost $O(\log_2 s)$. Hence the (amortized) worst case search time is just $O(|\mathcal{M}|/s + |\mathcal{M}| \log_2 s)$. By setting $s = |\mathcal{M}|/n$ we obtain $O(n + |\mathcal{M}| \log_2(|\mathcal{M}|/n))$. This means $O(n/m + |\mathcal{M}|/m \log_2(|\mathcal{M}|/n))$ average time. The value of $|\mathcal{M}|$ must be known before the search, but this is no problem, since $|\mathcal{M}| = \sum_i |L[p_i]|$ and thus can be computed in $O(m)$ time after the preprocessing (i.e., when the lengths of the lists $L[p_i]$ are known). Note that it is not a good idea to set $s = O(1)$, which improves the worst case but also prevents any gain in the average case, and effectively leads back to the original sparse dynamic programming algorithm.

Combining binary and interpolation searches. If the distribution of the δ-matching character positions in $\mathcal{M}$ is "regular" [16], then interpolation search [17, 16] can be used to reduce the $\log(x)$ factors to $\log\log(x)$ on average. The worst case remains the same as with binary search by using the following trick [13]: the binary and interpolation searches are interweaved so that in even iterations of the search one step of the binary search is executed, and in odd iterations one step of interpolation search is executed. This guarantees $O(\log(x))$ worst case time, without destroying the $O(\log\log(x))$ average time.

Using efficient priority queues. Van Emde Boas priority queues [15] support insertions and successor searches of integers in $O(\log\log u)$ time, where u is the size of the universe, i.e. n in our case. By using this structure we can obtain average and worst case times (including preprocessing) of $O((n\delta\sigma_{\text{P}\cap\text{T}}/\sigma_{\text{T}} + n\delta/\sigma) \log\log n)$ and $O((n\delta + |\mathcal{M}|) \log\log n)$, respectively. The penalty is $O(n\delta)$ additional space in the worst case. Unfortunately the trick of linear search and skipping by s positions does not work with this approach.

However, we can use Johnson's data structure [8], which supports a *homogeneous* sequence of insertions and successor searches in time $O(\log\log\Delta)$, where Δ is the gap between the successive values inserted or searched. Note that all the successive operations must be of the same type, i.e. insertions and searches cannot be mixed. Fortunately this is not a problem in our case, since the preprocessing phase consists of just inserting all the lists $L[p_i]$ into the queue and in the search phase we have only successor searches. For insertions $\Delta = \sigma/\delta$ on average, and hence the preprocessing takes $O(n\delta\sigma_{\text{P}\cap\text{T}}/\sigma_{\text{T}} \times \log\log(\sigma/\delta))$ time on average. in the search phase the value of Δ depends on the row. For the row 1 it is again σ/δ, but then grows exponentially (up to $O(n)$) on average. However, this is because the length of d_i decreases exponentially, i.e. $\Delta = n/|d_{i-1}|$ for row i on average. This does not affect the average complexity, i.e. by using the same argument as for the binary search, the average search time is $O(n\delta/\sigma \log\log(\sigma/\delta))$. This is $O(n)$ if $\delta = O(\sigma/\log\log\sigma)$. In this case the preprocessing cost is $O(n\sigma \log\log\log\log\sigma/\log\log\sigma)$.

The above bound can still be improved with exponential search trees [1]. The $O(\log\log\Delta)$ bound can be improved to $O(\sqrt{\log q/\log\log q})$, where q is the number of stored keys between two subsequent searches, rather than the absolute value of the difference of the keys (Δ). Note that $q \leq \Delta$ holds always, and in our case $q = 1$ in the preprocessing phase and $O(1)$ while searching in the first row of the matrix.

5 Column-wise sparse dynamic programming

In this section we present another cut-off variant. This algorithm does not depend on any complex helper data structures, and runs in $O(\sigma + m\sigma_{\mathrm{T}} + n)$ and $O(\sigma + m\sigma_{\mathrm{T}} + |\mathcal{M}|)$ average and worst case time, respectively.

The algorithm processes the dynamic programming matrix column-wise. Let us define *Last Prefix Occurrence LPO* as

$$LPO_{i,j} = \begin{cases} j' & \max j' \leq j \mid p_0 \ldots p_i =_\delta^\alpha t_h \ldots t_{j'} \\ -\alpha - 1 & \text{otherwise} \end{cases} \tag{6}$$

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

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

Alg. 6 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 LPO and WPO.

The average case running time of the algorithm depends on how many values there on average in the list WPO, i.e. $O(1)$. Each value is clearly processed in constant worst case time, and hence the algorithm runs in $O(n)$ average time. In the worst case the total length of the lists for all columns is $O(|\mathcal{M}|)$, and therefore the worst case running time is $O(n + |\mathcal{M}|)$, since every column must be visited. The preprocessing phase only needs to initialize LPO, which takes $O(m)$ time.

6 Bit-parallel dynamic programming

The worst case complexity of all the previous algorithms is $O(mn)$. In this section we show how bit-parallelism can be used to bring the worst case complexity down to $O(n\delta + \lceil n/w \rceil m)$ and the average case complexity to $O(\lceil n/w \rceil \sigma_{\mathrm{P}} + n(\delta\sigma_{\mathrm{P \cap T}}/\sigma_{\mathrm{T}} + 1))$, where w is the number of bits in computer word (typically 32 or 64). We number the bits from the least significant bit (0) to the most significant bit ($w-1$). C–like notation is used for the bit-wise operations of words; & is bit-wise `and`, | is `or`, $\sim$ negates all bits, << is shift to left, and >> shift to right, both with zero padding.

Let us first redefine the matrix D. Let $D_{i,j} = 1$ if $p_0 p_1 \ldots p_i =_\delta^\alpha t_h t_{h+1} \ldots t_j$. Otherwise, $D_{i,j} = 0$. This can be expressed as:

$$D_{i,j} = \begin{cases} 1 & p_i =_\delta t_j \text{ AND } \exists j' : 0 < j - j' \leq \alpha + 1 \text{ AND } D_{i-1,j'} = 1 \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

Alg. 6 DA-sdpco-columns$(T, n, P, m, \delta, \alpha)$.

```
for $i \leftarrow 0$ to $m-1$ do $LPO_i \leftarrow -\alpha - 1$
$top \leftarrow 0$
for $i \leftarrow 0$ to $n-1$ do
    $c \leftarrow t_i$; $h \leftarrow 0$
    for $j \leftarrow 0$ to $top - 1$ do
        $pr \leftarrow WPO'_j$
        $pc \leftarrow LPO_{pr}$
        if $|c - p_{pr+1}| \leq \delta$ then
            if $pr + 1 < m - 1$
                $WPO_h \leftarrow pr + 1$
                $h \leftarrow h + 1$
            else
                report match
    if $|c - p_0| \leq \delta$ then
        $WPO_h \leftarrow 0$
        $h \leftarrow h + 1$
    for $j \leftarrow 0$ to $h - 1$ do $LPO_{WPO_j} \leftarrow i$
    for $j \leftarrow 0$ to $top - 1$ do
        if $LPO_{WPO_j} \neq i$ AND $i - LPO_{WPO_j} \leq \alpha$ then
            $WPO_h \leftarrow WPO'_j$
            $h \leftarrow h + 1$
    $top \leftarrow h$
    $Lt \leftarrow WPO; WPO \leftarrow WPO'; WPO' \leftarrow Lt$
```

This definition can be used instead of the definition in Sec. 2. However, at a first glance it seems that this recurrence would lead to $O(\alpha nm)$ time, i.e. by far to the worst complexity. However, we show how to compute $O(w)$ columns in each row of the new matrix in $O(\alpha/w)$ time, i.e. in $O(1)$ time for small α.

To this end, assume that in the preprocessing phase we have computed a helper bit-matrix (whose efficient computation we will consider later) V:

$$V_{i,j} = \begin{cases} 1 & p_i =_\delta t_j \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

The computation of D will proceed column-wise, w columns at once. Each matrix element takes only one bit of storage, so we can store w columns into a single machine word. Assume that we have computed all rows of the columns $(j-1)w \dots jw-1$, and columns $jw \dots (j+1)w-1$ up to row $i-1$, and we want to compute the columns $jw \dots (j+1)w-1$ at row i. Assume also that $\alpha < w$. We adopt the notation $D^w_{i,j} = D_{i,jw\dots(j+1)w-1}$, and analogously for V. The goal is then to produce $D^w_{i,j}$ from $V^w_{i,j}$ and $D^w_{i-1,j}$ and $D^w_{i-1,j-1}$. $D^w_{i,j}$ does not depend on any other D^w element, according to the definition of D, and given our assumption that $\alpha < w$.

Now, according to Eq. (7), the kth bit in $D^w_{i,j}$ should be set iff (i) the kth bit in $V^w_{i,j}$ is set (i.e. $p_i =_\delta t_{jw+k}$), and (ii) any of the bits $k - \alpha - 1 \dots k - 1$ in $D^w_{i-1,j}$ or any of the bits $k + w - \alpha - 1 \dots w - 1$ in $D^w_{i-1,j-1}$ is set (i.e. the gap length to the previous match is at most α). To compute item (ii) efficiently we assume that we have available function $M(x)$, that for a binary number x is defined as:

$$M(x) = (x << 1) \mid (x << 2) \mid \dots \mid (x << (\alpha + 1)) \tag{9}$$

In other words, $M(x)$ copy-propagates all bits in x to left $1 \dots \alpha + 1$ positions. This means that if the 1 bits in x correspond to the matching positions of a pattern prefix, then $M(x)$ will have those 1 bits aligned in all positions where the matching prefix could be extended. We can now write the recurrence for D^w:

$$D^w_{i,j} = V^w_{i,j} \;\&\; (M(D^w_{i-1,j}) \mid (M(D^w_{i-1,j-1}) >> w)) \tag{10}$$

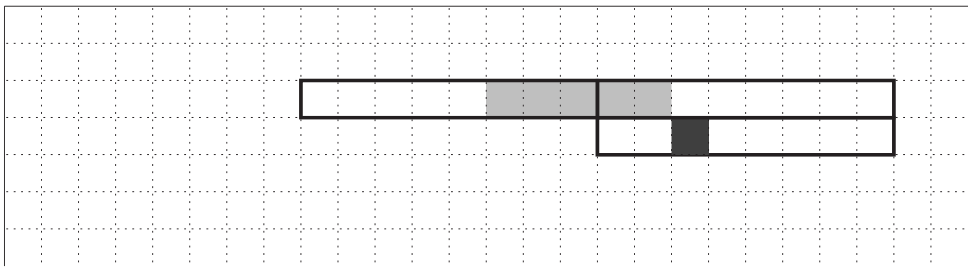


Figure 1: Tiling the dynamic programming matrix with $w \times 1$ vectors ($w = 8$). The dark gray cell of the current tile depends on the light gray cells of the two tiles in the previous row ($\alpha = 4$).

Fig. 1 illustrates the bits in the previous row affecting the bits in the current row.

We are not able to compute $M(x)$ in constant time, hence we use a precomputed look-up table instead[1]. Since w can be too large to make this approach feasible, we can precompute the answers e.g. to only $w/2$ or $w/4$ bit numbers, and correspondingly compute $M(x)$ in 2 or 4 pieces without affecting the time complexity (in our implementation we used $w/2 = 16$ bit numbers for computing $M(x)$).

Finally, the preprocessing of V can be done in the same way as the preprocessing of the list L in Alg. 3. However, the preprocessing has $O(\lceil n/w \rceil \sigma_{\mathrm{P}} + n(\delta \sigma_{\mathrm{P} \cap \mathrm{T}}/\sigma_{\mathrm{T}} + 1))$ cost now, since V must be initialized to all zeros. Searching clearly takes only $O(\lceil n/w \rceil m)$ time.

6.1 Adapting the cut-off trick

The cut-off trick given in Sec. 3 can be applied even if we are computing w columns in parallel. The new definition for the maximum row top_j^w for the column j is:

$$top_j^w = \mathrm{argmax}_i \ \{(D^w_{i,j-1} \ \& \ a = 0 \ \text{AND} \ D^w_{i-1,j-1} \ \& \ a \neq 0) \quad (11)$$

$$\text{OR} \ D^w_{i-1,j} \neq 0\} \quad (12)$$

where the bitmask $a = {\sim}0 << (w - \alpha - 1)$. Consider first the part

$$D^w_{i,j-1} \ \& \ a = 0 \ \text{AND} \ D^w_{i-1,j-1} \ \& \ a \neq 0.$$

The rationale is as follows. When we are computing $D^w_{i,j}$, only the $\alpha + 1$ highest non-zero bits of $D^w_{i-1,j-1}$ can affect the bits in $D^w_{i,j}$. These are selected by the $\&\ a$ operation. However, since we are computing w columns in parallel, the non-zero bits in $D^w_{i-1,j}$ (the second part), i.e. in the previous row of the *current* set of columns, can affect the bits in $D^w_{i,j}$ as well. Obviously, this second part cannot be computed at column $j - 1$.

We solve this problem simply by computing the first part (11) of top_j^w after the column $j-1$ have been computed, and when processing the column j, we increase top_j^w if needed according to the second part (12).

Alg. 7 gives the pseudo code. It uses $w' = w/2$ bits for the precomputed table for the $M(\cdot)$ function. For simplicity, the code also assumes that $\alpha < w'$ (but w columns are still processed in parallel).

6.2 Handling large α in $O(1)$ time

Alg. 7 assumes that $\alpha < w$. For larger α the time increases by $O(\alpha/w)$ factor, as the gap may span over several machine words. We now show how to remove this limit while maintaining the $O(1)$ cost for processing w columns.

[1] $M(x)$ obviously belongs to AC^0, and hence in principle could be computed in $O(1)$ time with a constant depth polynomial size circuit.

Alg. 7 DA-bpdp$(T, n, P, m, \delta, \alpha)$.

```
for $i \leftarrow 0$ to $m-1$ do $A[p_i] \leftarrow 0$
for $i \leftarrow 0$ to $m-1$ do
    if $A[p_i] = 0$ then
        $A[p_i] \leftarrow 1$
        for $j \leftarrow -\delta$ to $\delta$ do
            $c \leftarrow \max(0, \min(p_i + j, \sigma - 1))$
            $L'[c] \leftarrow L'[c] \cup \{p_i\}$
        for $j \leftarrow 0$ to $\lceil n/w \rceil - 1$ do $V[p_i][j] \leftarrow 0$
for $i \leftarrow 0$ to $n-1$ do
    for $j \leftarrow 0$ to $|L'[t_i]| - 1$ do
        $c \leftarrow L'[t_i][j]$
        $V[c][i/w] \leftarrow V[c][i/w] \mid (1 << (i \,\%\, w))$
$w' \leftarrow w/2$; $msk \leftarrow (1 << w') - 1$
for $i \leftarrow 0$ to $(1 << w') - 1$ do
    $M[i] \leftarrow 0$
    for $j \leftarrow 0$ to $\alpha$ do $M[i] \leftarrow M[i] \mid (i << (j+1))$
$top \leftarrow m - 1$
$D_0 \leftarrow V[p_0][0]$
for $i \leftarrow 1$ to $top$ do
    $D_i \leftarrow V[p_i][0] \,\&\, (M[D_{i-1} \,\&\, msk] \mid (M[D_{i-1} >> w'] << w'))$
if $D_{m-1} \neq 0$ then report matches
for $j \leftarrow 1$ to $\lceil n/w \rceil$ do
    $D'_0 \leftarrow V[p_0][j]$
    $i \leftarrow 1$
    while $i \leq top$ do
        $x \leftarrow M[D'_{i-1} \,\&\, msk] \mid (M[D'_{i-1} >> w'] << w')$
        $y \leftarrow M[D_{i-1} >> w'] >> w'$
        $D'_i \leftarrow V[p_i][j] \,\&\, (x \mid y)$
        if $i = top$ AND $top < m-1$ AND $D'_i \neq 0$ then
            $D_i \leftarrow 0$
            $top \leftarrow top + 1$
        $i \leftarrow i + 1$
    if $top = m-1$ AND $D'_{m-1} \neq 0$ then report matches
    while $top > 0$ AND $D'_{top} \,\&\, (\sim 0 << (w - \alpha - 1)) = 0$ do $top \leftarrow top - 1$
    if $top < m-1$ then $top \leftarrow top + 1$
    $Dt \leftarrow D$; $D \leftarrow D'$; $D' \leftarrow Dt$;
```

Let us define *Last Prefix Occurrence*:

$$LPO_{i,j} = \begin{cases} j' & D^w_{i,j'} \neq 0 \mid \max j' \leq j \\ -\alpha - 1 & \text{otherwise} \end{cases} \qquad (13)$$

I.e. for $LPO_{i,j} = j'$, $D^w_{i,j'}$ is the vector that corresponds to the last (δ, α)-match(es) of the prefix $p_0 \dots p_i$ in the text area $t_0 \dots t_{wj-1}$. If such vector does not exist (e.g. when $j = 0$) we set $LPO_{i,j} = -\alpha - 1$.

Assume that $\alpha \geq w$ and consider the computation of $D^w_{i,j}$. The recurrence becomes

$$D^w_{i,j} = V^w_{i,j} \,\&\, (M(D^w_{i-1,j}) \mid ov) \qquad (14)$$

The vector ov is computed according to the last prefix occurrence information. Let $j' = LPO_{i-1,j-1}$. We have the following four cases (see also Fig. 2):

1. $j' = -1$: no matching prefixes have been found, hence $ov = 0$.
2. $w(j - j') - w + 1 > \alpha + 1$: no bit of $D^w_{i-1,j'}$ can affect any bit in $D^w_{i,j}$, hence we set $ov = 0$.
3. $w(j - j') + w - 1 \leq \alpha + 1$: any set bit in $D^w_{i-1,j'}$ is close enough to affect any bit in $D^w_{i,j}$, hence we set $ov = \sim 0$.

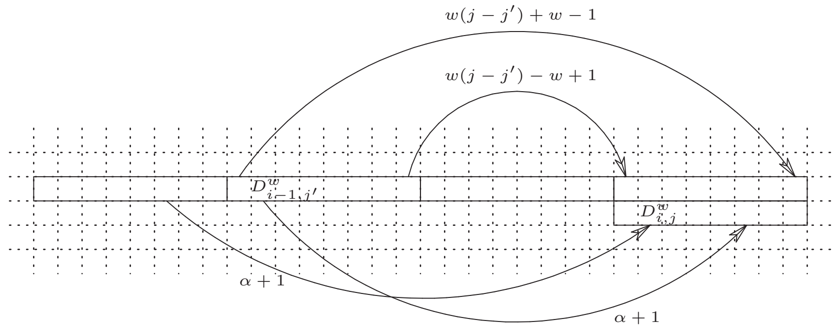


Figure 2: Some bit distances illustrated.

4. Otherwise some bits of $D^w_{i-1,j'}$ or $D^w_{i-1,j'-1}$ can be close enough to affect some bits of $D^w_{i,j}$.

The condition 4 is the tricky one. Assume that the function $M(\cdot)$ uses the value $\alpha' = w + \alpha \bmod w$ instead of α. Note that in Eq. (14) $M(D^w_{i-1,j})$ is still correctly computed as we assume that $\alpha \geq w$. The contribution of $D^w_{i-1,j'}$ is therefore $ov = M(D^w_{i-1,j'}) >> w$. If $D^w_{i-1,j'-1}$ is not zero, then its high order bits may contribute as well, and we set

$$ov = (M(D^w_{i-1,j'}) >> w) \mid (M(D^w_{i-1,j'-1}) >> 2w). \tag{15}$$

Obviously, $LPO_{i,j}$ can be easily maintained in constant time for each i, j. $LPO(i, -1)$ is initialized to $-\alpha - 1$ for all i, which takes $O(m)$ time. Then, the computation of D^w proceeds column-wise. After $D^w_{i,j}$ is computed, we simply set $LPO_{i,j} = j$ iff $D^w_{i,j} \neq 0$, otherwise we set $LPO_{i,j} = LPO_{i,j-1}$. In practice we can store only the latest value of LPO for each row, so only $O(m)$ space is needed.

Finally, note that since $\alpha \geq w$, and hence $w \leq \alpha' < 2w$, the function $M(\cdot)$ is now much simpler to compute. That is, $M(x) = 2^{2w} - LSB(x) - LSB(x)$, where $LSB(x)$ extracts the least significant set bit of x. The first subtraction operation then propagates the LSB to every higher position as well, while the second subtraction then clears the least significant bit of the result. The last obstacle is the computation of $LSB(\cdot)$. However, the solution is part of the computing folklore, and can be computed as $LSB(x) = (x \;\&\; (x-1)) \wedge x$ in $O(1)$ time. Thus, for $\alpha \geq w$ we can compute $M(\cdot)$ in $O(1)$ time without table look-ups.

Hence we can conclude that the value of α does not affect the running time of the algorithm.

7 Linear time preprocessing

For the "cut-off" variants of our algorithm the $O(\delta n)$ (worst case) preprocessing time can be the dominating factor in some cases. In this section we present a preprocessing variant not dependent on the pattern or δ in time and space. The idea is to 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 $k\delta \leq t_j \leq (k+1)\delta - 1$, $L[k-1]$ (if $k - 1 \geq 0$), and $L[k+1]$ (if $k + 1 \leq \lceil \sigma/\delta \rceil - 1$). 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. Alg. 8 gives the pseudo code.

The search in the original sparse dynamic programming algorithm with the "cut-off" idea is based (at least in the simple practical implementation) on binary searching over a list of all δ-occurrences of a current p_i character. The variant with linear preprocessing is also based on binary search, namely it runs the binary search over the list $L[k]$ for such k that $k\delta \leq p_i \leq (k+1)\delta - 1$, as any j such that $t_j =_\delta p_i$ must have been stored at $L[k]$. Still, the problem is

Alg. 8 DA-sdp-preprocess($T, n, P, m, \delta, \alpha$).

```
1    for $j \leftarrow 0$ to $n-1$ do
2        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
3            $L[c] \leftarrow L[c] \cup \{j\}$
4    return $L$
```

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 $t_j \neq_\delta p_i$, 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 $O(n + |\mathcal{M}|(\log_2 n + \alpha))$, or $O(n \log\log n + |\mathcal{M}|(\log\log n + \alpha))$ if van Emde Boas trees [15] are used. The average time in this variant becomes $O(n \log_2 n)$ or $O(n \log\log n)$, respectively. The other alternatives to binary search can be adapted analogously.

Note this idea in principle could be used also for SDP without the cut-off trick, but there it is of little value as in that algorithm the preprocessing (Alg. 4) either takes $O(n)$ time or it takes no more time (in the complexity terms) than the successive search routine. The only benefit of this variant for the plain SDP algorithm is reducing the worst case space complexity to $O(n)$.

For the bit-parallel algorithm the preprocessing takes $O(\lceil n/w \rceil \sigma_{\text{P}} + n(\delta\sigma_{\text{P}\cap\text{T}}/\sigma_{\text{T}} + 1))$ time. Using the above idea this can be reduced to $O(\lceil n/w \rceil \sigma_{\text{P}}/\delta + n)$. However, the search algorithm becomes a filter, since we are not able to verify the δ-matches bit-parallelly. This means that the candidate pattern occurrences must be verified using e.g. plain dynamic programming algorithm, which increases the worst case cost to $O(mn)$. The good average case performance is preserved. Finally, note that we can use any interval $h = 1 \ldots \sigma$, instead of $h = \delta$, for partitioning the alphabet. Using larger intervals decreases the preprocessing cost, but as a trade-off makes the filter slower and triggers more verifications.

8 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 the the cut-off variant of the sparse dynamic programming (Sec. 4.1), and actually can be seen as its brute-force variant. The algorithm has two traits that distinguish it from Alg. 5: (*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:

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

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:

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

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 $O(\alpha|\mathcal{M}|)$ worst case time.

With one simple optimization the worst case can be improved to $O(\min\{\alpha|\mathcal{M}|, mn\})$ (improving also the average time a bit). When computing the current list L_i, Simple algorithm may

Alg. 9 DA-simple$(T, n, P, m, \delta, \alpha)$.

```
$h \leftarrow 0$
**for** $j \leftarrow 0$ **to** $n-1$ **do**
    **if** $|t_j - p_0| \leq \delta$ **then**
        $L[h] \leftarrow j$; $h \leftarrow h+1$
**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** $\min(L[j+1], L[j]+\alpha+1)$ **do**
            **if** $|t_{j'} - p_i| \leq \delta$ **then**
                $L'[h] \leftarrow j'$; $h \leftarrow h+1$
                **if** $i = m-1$ **then** report match
    $Lt \leftarrow L; L \leftarrow L'; L' \leftarrow Lt$
```

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 $|\mathcal{M}| = O(mn)$, 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 $O(mn)$ worst case time. The average case is improved as well. Alg. 9 shows the pseudo code, including this improvement.

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

$$O(\alpha|L_{i-1}|) = O(n\alpha^i(\delta/\sigma)^i) = O(n(\alpha\delta/\sigma)^i) \tag{16}$$

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 $O(n)$.

9 Non-deterministic finite automata based approaches

In this section we present two algorithms based on non-deterministic finite automaton. The first algorithm is a trivial adaptation of the algorithm in [12] and uses $m + (m-1)\alpha$ bits to represent the automaton, and hence runs in $O(n\lceil m\alpha/w\rceil)$ time. This observation was done also in [4], but our adaptation is somewhat simpler and more efficient. The second algorithm is more complex but uses only $O(m \log_2 \alpha)$ bits, and therefore runs in $O(n\lceil (m \log_2 \alpha)/w\rceil)$ time.

9.1 (δ, α)-matching with $O(m\alpha)$ bits

In [12] the authors present a bit-parallel algorithm for string matching with bounded gaps and character classes, the motivating application being protein searching. The algorithm handles patterns of form $p_0 g(a_0, b_0) p_1 g(a_1, b_1) p_2 \ldots g(a_{m-2}, b_{m-2}) p_{m-1}$, where $p_i \subset \Sigma$ and $g(a_i, b_i) \in \{\Sigma^\ell \mid a_i \leq \ell \leq b_i\}$. In other words, a_i and b_i are the minimum and maximum gap lengths ($a_i \leq b_i$) between p_i and p_{i+1}. This algorithm can be trivially adapted to (δ, α)-matching: δ-matching is just a special case of matching with character classes, and we can use $g(a_i, b_i) = g(0, \alpha)$. The original algorithm runs in $O(n\lceil L/w\rceil)$ worst case time, where $L = m + \sum_i b_i$. In our case $L = m + (m-1)\alpha$. Preprocessing for (δ, α)-matching takes $O(\sigma + m\delta)$ time (the preprocessing in [4] takes $O(m\alpha\sigma)$ time). While the original technique is trivial to modify for our purposes, we briefly review the (modified) algorithm here for completeness. For more details refer to [12]. This also serves as the base of our new more efficient algorithm, see Sec. 9.2.

The algorithm is based on simulating a nondeterministic finite automaton with bit-parallelism, namely with the Shift-And [2] algorithm. The standard Shift-And automaton is

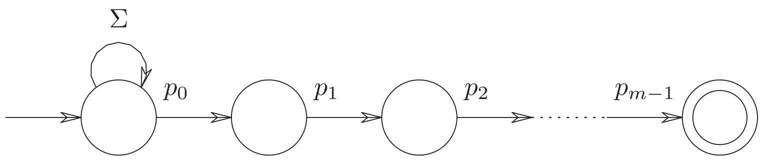


Figure 3: Shift-And automaton.

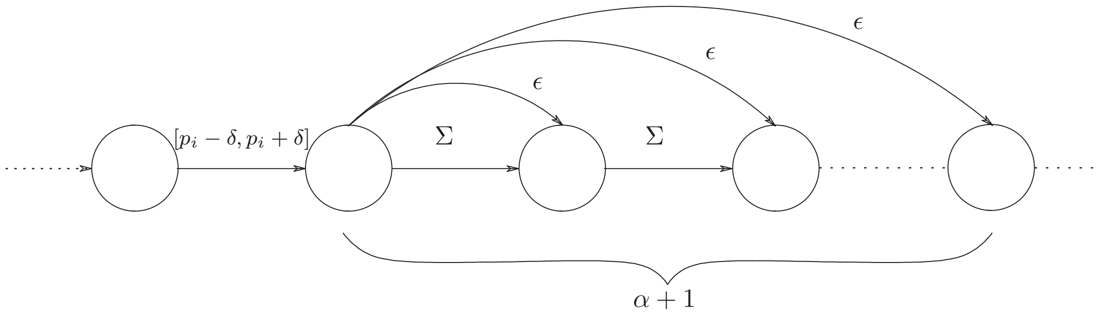


Figure 4: A building block for a Shift-And automaton allowing δ-matches and α-bounded gaps.

constructed as follows. The automaton has states $0, 1, \ldots, m$. The state 0 is the initial state, state m is the final (accepting) state, and for $i = 0, \ldots, m-1$ there is a transition from the state i to the state $i+1$ for character p_i. In addition, there is a transition for every $c \in \Sigma$ from and to the initial state, which makes the automaton nondeterministic. Fig. 3 illustrates the resulting automaton.

The preprocessing algorithm builds a table B, having one bit-mask entry for each $c \in \Sigma$. For $0 \leq i \leq m-1$, the mask $B[c]$ has ith bit set to 1, iff $p_i = c$. These correspond to the transitions of the implicit automaton. That is, if the bit i in $B[c]$ is 1, then there is a transition from the state i to the state $i+1$ with character c.

We also need a bit-vector D for the states of the automaton. The ith bit of the state vector is set to 1, iff the state i is active. Initially each bit is set to 0. For each text symbol c the vector is updated by

$$D \leftarrow ((D << 1) \mid 1) \ \& \ B[c].$$

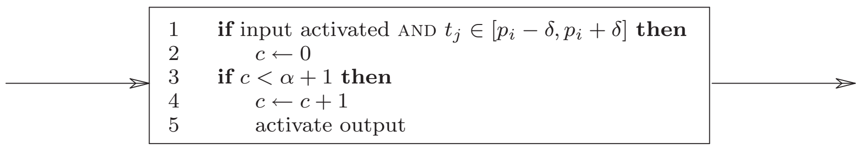
This simulates all the possible transitions of the nondeterministic automaton in a single step. If after the update the mth bit of D is one, then there is an occurrence of P. If $m \leq w$, then the algorithm runs in time $O(n)$.

To handle δ-matches, we can simply modify the definition of B so that now $B[c]$ has ith bit set to 1, iff $|p_i - c| \leq \delta$. Handling the gaps is more complicated, and requires adding new states into the automaton. Let s be the state that recognizes prefix of length i of the pattern. To allow a gap of length α after this prefix, we add a sequence of α states so that there is transition from state $s+j-1$ to state $s+j$ with all characters in Σ, where $j \in \{1 \ldots \alpha\}$. This forces a gap of exactly length α. The state s is called *gap initial* and the state $s + \alpha$ *gap final* state. To allow a gap of length $0 \ldots \alpha$, we add ϵ transitions from s to all states $s+j$, i.e. $0 \ldots \alpha$ states can be skipped, and hence the gap is correctly processed. Fig. 4 shows a building block to handle one character of the pattern, while the automaton for the whole pattern is obtained by concatenating the building blocks for all characters (the final gap may be omitted).

To handle the new Σ transitions is trivial, since these can be handled as any normal transition, i.e. we update the states as $D \leftarrow ((D << 1) \mid 1) \ \& \ B[c]$. The ϵ transitions need special treatment. If the state s recognizing a pattern prefix becomes active, then the states $s+j$, where $j \in \{1 \ldots \alpha\}$ should be activated as well. These transitions are handled as a second step, after the normal Shift-And step. Subtraction operation can be used to copy-propagate the activated bit to all these states as follows. Let I be a bit mask that has 1 in all positions corresponding

Alg. 10 DA-ma-bits$(T, n, P, m, \delta, \alpha)$.

```
$\ell \leftarrow m + (m-1) \times \alpha$
**for** $i \leftarrow 0$ **to** $\sigma - 1$ **do** $B[i] \leftarrow 0$
$I \leftarrow 0$; $h \leftarrow 0$; $msk \leftarrow 0$
**for** $i \leftarrow 0$ **to** $m - 1$ **do**
    **for** $j \leftarrow -\delta$ **to** $\delta$ **do**
        $c \leftarrow \max(0, \min(p_i + j, \sigma - 1))$
        $B[c] \leftarrow B[c] \mid (1 << h)$
    $h \leftarrow h + 1$
    **if** $i < m - 1$ **then**
        $I \leftarrow I \mid (1 << (h-1))$
        $msk \leftarrow msk \mid (((1 << \alpha) - 1) << h)$
        $h \leftarrow h + \alpha$
**for** $c \leftarrow 0$ **to** $\sigma - 1$ **do** $B[c] \leftarrow B[c] \mid msk$
$M \leftarrow 1 << (\ell - 1)$; $D \leftarrow 0$;
**for** $i \leftarrow 0$ **to** $n - 1$ **do**
    $D \leftarrow ((D << 1) \mid 1)$ & $B[t_i]$
    $AI \leftarrow D$ & $I$
    $D \leftarrow D \mid (((AI << (\alpha + 1)) - AI)$
    **if** $(D$ & $M) \neq 0$ **then** report match
```



```
**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
```

Figure 5: A building block for a systolic array detecting δ-matches with α-bounded gaps.

to the gap initial states. Then $AI \leftarrow D$ & I is the bit mask that has 1 in all position where the gap initial state is active. To propagate the active bits in all gap states we use

$$D \leftarrow D \mid ((AI << (\alpha + 1)) - AI).$$

If AI has 0 in the gap initial state, then this has no effect to the gap states. If the bit is 1, then it is propagated to all gap states due to carry effect of the subtraction, and zeroes the overflowed bits $(AI << (\alpha + 1))$.

Note that this solution is slightly simpler than the one given in [12]. Our solution is possible as the automaton has more regular structure now. Alg. 10 gives the pseudo code for the complete algorithm.

9.2 (δ, α)-matching with $O(m \log_2 \alpha)$ bits

The problem of Alg. 10 is that it needs $m + (m-1)\alpha$ bits to represent the search search state. Our goal is to reduce this to $O(m \log_2 \alpha)$.

On very high level, the algorithm can be seen as a novel combination of Shift-And and Shift-Add algorithms [2]. 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 block for character p_i is shown in Fig. 5. The final array is obtained by concatenating one building block for each pattern character. We call the building blocks as *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 [6].) 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. 5 can be computed bit-parallelly as

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

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 the 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 use the plain Shift-And step:

$$D' \leftarrow ((D << \ell) \mid 1) \ \& \ B[t_i],$$

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:

$$D \leftarrow D' \mid ((\sim C >> (\ell - 1)) \ \& \ msk).$$

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. 5. 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

$$C \leftarrow C \ \& \ \sim(D' \times ((1 << \ell) - 1))$$

clears the corresponding counter fields, and

$$C \leftarrow C \mid (D' \times ((1 << (\ell - 1)) - (\alpha + 1)))$$

copies the initial value $2^{\ell - 1} - (\alpha + 1)$ to all the cleared fields. Note that the multiplication in the clearing operation can be avoided by using $C \leftarrow C \ \& \ \sim((D' << \ell) - D')$ instead.

This completes the algorithm. Alg. 11 gives the pseudo code.

9.3 Handling long patterns

Both Alg. 10 and Alg. 11 run in $O(n)$ worst case time, if $m + (m-1)\alpha \leq w$ and $m(\lceil \log_2(\alpha + 1) \rceil + 1) \leq w$, respectively. Otherwise, several machine words are needed to represent the search state, and the time grows accordingly. However, to use the well-known folklore idea, it is enough to update the machine words that are *active*, i.e. only those words of D (correspondingly C) that are not zero (have counters that have not overflowed). On average only $O(1)$ words need to be updated for each text position.

Another solution is to search only a subpattern whose representation fits into a single machine word, and verify the whole pattern only when the subpattern matches [12]. This is attractive only if δ and α are small enough to keep the number of verifications small.

9.4 Backward searching

In [12] the authors combined their forward matching algorithm (i.e. their version of Alg. 10) with Backward Nondeterministic Dawg Matching (BNDM) [11] algorithm. The adaptation is straightforward, yet allows the algorithm to skip some text characters. We only note that exactly the same technique can be used with our Alg. 11.

Alg. 11 DA-mloga-bits$(T, n, P, m, \delta, \alpha)$.

```
1   $\ell \leftarrow \lceil \log_2(\alpha+1) \rceil + 1$
2   for $i \leftarrow 0$ to $\sigma - 1$ do $B[i] \leftarrow 0$
3   for $i \leftarrow 0$ to $m - 1$ do
4       for $j \leftarrow -\delta$ to $\delta$ do
5           $c \leftarrow \max(0, \min(p_i + j, \sigma - 1))$
6           $B[c] \leftarrow B[c] \mid (1 << (i \times \ell))$
7   $msk \leftarrow 0$
8   for $i \leftarrow 0$ to $m - 1$ do $msk \leftarrow msk \mid (1 << (i \times \ell))$
9   $am \leftarrow (1 << (\ell - 1)) - (\alpha + 1)$
10  $D \leftarrow 0$; $C \leftarrow (am + \alpha + 1) \times msk$
11  $msk \leftarrow msk >> \ell$
12  $mm \leftarrow 1 << ((m-1) \times \ell)$
13  for $i \leftarrow 0$ to $n - 1$ do
14      $C \leftarrow C + ((\sim C >> (\ell - 1))\ \&\ msk)$
15      $D' \leftarrow ((D << \ell) \mid 1)\ \&\ B[t_i]$
16      $D \leftarrow D' \mid ((\sim C >> (\ell - 1))\ \&\ msk)$
17      $C \leftarrow C\ \&\ \sim((D' << \ell) - D')$
18      $C \leftarrow C \mid (D' \times am)$
19      if $(D\ \&\ mm) = mm$ then report match
```

10 Extensions

Our algorithms can be extended in various ways, including the following cases.

10.1 Relaxing α and δ

All our dynamic programming algorithms can be generalized to 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. More precisely, pattern $p_0 p_1 p_2 \ldots p_{m-1}$, where $p_j \subset \Sigma$, matches $t_{i_0} t_{i_1} t_{i_2} \ldots t_{i_{m-1}}$, if $t_{i_j} \in p_j$ for $j \in \{0, \ldots, m-1\}$, where $i_j < i_{j+1}$, and $i_{j+1} - i_j \leq \alpha_j + 1$. This problem variant has important applications e.g. in protein searching, see [12].

The dynamic programming algorithms can easily handle general α gaps. All is needed is to use α_i in the row i of the matrix. The time complexities are preserved if we (pessimistically) consider that $\alpha = \max\{\alpha_i\}$.

Handling the character classes is slightly more complicated. For the basic dynamic programming algorithms 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.

The sparse dynamic programming algorithms become a bit more complicated, since we need to have $\mathcal{M}$ preprocessed. Inspired by Alg. 3, we do this as follows. First compute lists $L'[c] = \{i \mid c \in p_i\}$. This can be done in one linear scan over the pattern, and takes $O(m \log \sigma)$ total time. 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).

Finally, Alg. 10 as described in the original paper [12] handles just these extensions. Alg. 11 can trivially handle character classes. On the other hand, this algorithm makes an implicit assumption of fixed α values. Generalizing for different length gaps seems to be possible, but not worth the effort.

10.2 Handling negative and range restricted gaps

The generalization of Sec. 10.1 to allow different gap lengths according to pattern position is still restrictive. In fact, the algorithm in [12] allows 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. However, this extension is easy or even trivial to handle in all our algorithms, i.e. in most algorithms 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 $g(0, \alpha_i)$. The column-wise sparse dynamic programming is a bit trickier, but still adaptable.

Yet a stronger model [10] 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 $g(-b_i, -a_i)$ instead of $g(a_i, b_i)$.

The general case where we also allow $a_i < 0 < b_i$ is trickier, and automata based approaches do not work anymore. However, adapting the dynamic programming algorithms (with or without the cut-off trick), and Simple algorithm of Sec. 8 to this setting is still quite straightforward.

In the following we consider the modification needed for some of the algorithms.

10.2.1 Dynamic programming

Basically, the only modification for the sparse algorithms 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 $a_i < 0$.

Adapting the plain dynamic programming algorithm to the case $a_i < 0 < b_i$ is quite simple but requires scanning each row twice. The first pass, from left to right, follows exactly the original dynamic programming recurrence (cf. Alg. 1), with positive gaps limited to b_i. Immediately after that, the same row is scanned from right to left. This time, each δ-match in rows $1 \ldots m-1$ is verified with respect to its upper-right neighboring cell (as opposed to the upper-left neighbor during the first pass). Another difference is that the values -1 are never written in D during this pass. It implies from the fact that any (δ, α)-match in a row using (for the current row) a gap from 0 to b_i is also a valid (δ, α)-match if the allowed gap for the current row is from a negative a_i to b_i. In other words, the right-to-left scan may add some (δ, α)-matches but not invalidate any. Clearly, this dynamic programming routine visits each cell twice, but the $O(mn)$ time complexity is preserved.

10.2.2 Bit-parallel dynamic programming

The core of Alg. 7 is the use of $M(x)$ (Eq. (9)) to select the positions from the previous row where a matching pattern ends. To handle gaps of the form $g(a_i, b_i)$ where $a_i \geq 0$ we use

$$M_{\mathrm{L}}(x) = (x << (a_i + 1)) \mid (x << (a_i + 2)) \mid \ldots \mid (x << (b_i + 1)). \tag{17}$$

For the negative gaps $b_i \leq 0$ we just align the bits from right, and hence define:

$$M_{\mathrm{R}}(x) = (x >> -b_i) \mid (x >> (-b_i + 1)) \mid \ldots \mid (x >> -a_i)). \tag{18}$$

The general case $a_i < 0 < b_i$ is handled as a combination of these:

$$M(x) = (x >> -a_i)) \mid (x >> (-a_i + 1)) \mid \ldots \mid (x << 1) \mid \ldots \mid (x << (b_i + 1)). \tag{19}$$

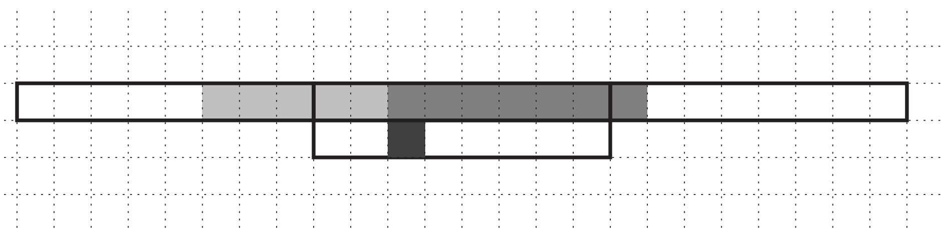


Figure 6: Tiling the dynamic programming matrix with $w \times 1$ vectors ($w = 8$). The dark gray cell of the current tile depends on the lighter gray cells of the three tiles in the previous row (the allowed gap is $g(-7, 4)$).

Fig. 6 illustrates the bits, i.e. the matching prefix positions, selected by $M()$. The final simple modification that we need is to take $D^w_{i-1,j+1}$ into account while computing $D^w_{i,j}$, since the negative gaps may span into it. Hence we modify Eq. (10) to:

$$D^w_{i,j} = V^w_{i,j} \ \& \ (M(D^w_{i-1,j}) \mid (M(D^w_{i-1,j-1}) >> w) \mid (M(D^w_{i-1,j+1} << w) >> w)). \qquad (20)$$

10.2.3 Simple

Finally, consider the Alg. 9 of Sec. 8. This algorithm 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 $j' \in L_{i-1}$ δ-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 $O(\min\{(b-a+1)|\mathcal{M}|, mn\})$, where for simplicity we have considered that the gaps are of the same size for all rows.

10.3 Multiple patterns

The sparse and especially the bit-parallel dynamic programming algorithms have relatively high preprocessing costs, $O(\delta n)$ and $O(\delta n + \sigma_{\text{P}}\lceil n/w \rceil)$ in the worst case, respectively. Both have also an $O(m + \sigma + \delta\sigma_{\text{P}})$ additional term, which does not affect the total complexity. However, if we want to search a set of r patterns, instead of only one pattern, the preprocessing remains essentially the same, since it depends only on the text and the pattern alphabet. The total (worst case) preprocessing times increase only to $O(\delta n + rm)$ and $O(\delta n + \sigma_{\text{P}}\lceil n/w \rceil + rm)$ for the sparse and bit-parallel dynamic programming algorithms, respectively, where we have pessimistically considered that m is the length of the longest pattern in the set, and that σ_{P} is the number of distinct symbols in the whole pattern set. The search times have to be multiplied by r, of course, but relative preprocessing cost per pattern is considerably reduced.

10.4 δ-occurrence of a set of strings with bounded gaps

In [5] another variant of multipattern matching was considered. Assume that we have a set of r patterns, $P_{1\ldots r}$. Each pattern is searched with $\alpha = 0$, and between the occurrence of P_i and P_{i+1} there can be a gap whose length is at most α_i. If all the patterns are found this way, we count an occurrence of the whole set. Note that the individual patterns must occur strictly in order $P_1, P_2, \ldots, P_r$. This variant can be easily solved by just concatenating all the r patterns, forming a single pattern of length m, and applying the techniques of Sec. 10.1.

10.5 Reducing space with block-wise computation

There is a simple way to save space in the algorithms based on a dynamic programming matrix, without a loss in the worst and the average case time complexities. The idea is to perform the preprocessing and the search in h-column blocks. Each block is processed only if the previous block is finished. As the leftmost column of any block, except the first block, depends on the previous block, it is clear that this approach cannot achieve $o(m)$ space complexity. Consequently, this idea cannot improve the plain dynamic programming variants (Alg. 1 and 2).

The sparse dynamic programming (cf. Alg. 4) can be performed in h-column blocks. The preprocessing is performed for each block separately, and in the search phase for each block k, $k \in \{0, \ldots, \lceil n/h \rceil - 1\}$, we scan all the lists $L[p_i]$ and for each $i \in \{0, \ldots, m-2\}$ we store the largest value on $L[p_i]$ which is less than $h(k+1)$. If the list $L[p_i]$ does not have any item in the row i of the current h-column block, then the desired value is simply copied from the previous block.

Hence, we have $O(mn/h)$ penalty in time and $O(h\delta + m)$ overall space. By setting $h = O(m)$ we obtain $O(n)$ time penalty, which never dominates, and only $O(m\delta)$ space in the worst case. Still, this improvement can hardly be recommended to the sparse DP algorithm with the cut-off trick, since its column-wise variant (Alg. 6) is faster and simpler (avoids complex search helper structures).

Using h-column blocks makes more sense for Simple algorithm. In a crude way we could use overlapping blocks, with the overlapping zones of $(\alpha + 1)(m - 1)$ columns. Setting $h = O(\alpha m)$ does not deteriorate the time complexity of Simple algorithm but reduces the space complexity if only $\alpha m = o(n)$, which is a realistic assumption. Moreover, in practice, a block of $O(\alpha m)$ characters should easily fit an L2 (or even L1) cache of any modern CPU, which should make this variant more cache friendly than the original one involving several (but $O(1)$ on average) scans over the entire text.

11 Experimental results

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

For the text we used a concatenation of 7543 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 $[0 \ldots 127]$. 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. Fig. 7 shows the timings for different pattern lengths. The timings are for the following algorithms:

DP: The plain Dynamic Programming algorithm [5], $O(nm)$ time and $O(m)$ space (a column-wise variation of Alg. 1);

SS: Sequential Sampling algorithm [3], $O(nm)$ time and $O(\alpha m)$ space;

DP Cut-off: "Cut-off" version of DP (Alg. 2);

SDP: Sparse Dynamic Programming (Alg. 4);

SDP Cut-off: "Cut-off" version of SDP (Alg. 5);

SDP Cut-off 3-L: $O(n)$ preprocessing time variant of SDP Cut-off (Sec. 7);

CWSDP: Column-Wise Sparse Dynamic Programming (Alg. 6);

BP Cut-off: Bit-parallel "cut-off" (Alg. 7);

BP Cut-off 3-L: Fast preprocessing time variant of BP Cut-off (Sec. 7).

Simple: Simple algorithm (Alg. 9).

NFA alpha: The nondeterministic finite automaton (Alg. 10).

NFA log alpha: The nondeterministic finite automaton (Alg. 11).

The experiments show that the new algorithms are substantially faster than the previous methods. This is especially true for small (δ, α) and large m. The fastest algorithms are the "cut-off" variants, the bit-parallel algorithm being the most efficient for large δ and α. Note that for large (δ, α) the bit-parallel cut-off algorithm with the fast preprocessing becomes faster for long patterns. This is because for long patterns the number of verifications decrease. The sequential sampling method is a bit slower but comparable to the plain dynamic programming algorithm, which is in contrast with the experimental results reported in [3].

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 not interesting anymore for this application.

We also repeated the experiments on uniformly random data, with $\sigma = 128$. The results are shown in Fig. 7. In this case the basic cut-off algorithm (Alg. 2) is clearly the best for all but very short patterns. Note however, that the bit-parallel cut-off variant (Alg. 7) has always shorter search time (not shown in the plot), but the high preprocessing cost makes it quite slow for if the pattern alphabet is large. Note that the performance of the algorithms with the linear time preprocessing are very stable, independent of the pattern lengths, i.e. independent on σ_{P}, as expected.

Fig. 8 shows timings for $\alpha = 1 \ldots 8$. For random data the times are quite insensitive to increasing α, except for Simple, DP Cut-off, CWSDP and the two NFA algorithms. For the NFA algorithms the explanation is that more computer words are needed for increasing α and large m. For the others, the 'constant' factor depends on α, but for large enough m and α the constant should still converge. For MIDI data the situation is worse, but again this mainly is due to the fact that for large α a significant portion of the text matches the pattern.

Finally, note that the performance is much more predictable for random than for MIDI data. Also, Simple algorithm is always the best alternative for random data, while for MIDI data it is useful mainly for relatively small δ and α.

12 Conclusions and future work

We have presented new efficient algorithms for string matching with bounded gaps and character classes. Our algorithms are based on sparse dynamic programming, pre-emptying the computation early where the match cannot be extended, bit-parallelism, nondeterministic finite automata, and the combination of these methods. Besides having theoretically good worst and average case complexities, the algorithms are shown to outperform the previous methods in practice as well.

We are working on several other algorithms to solve the present problem. Some of these seem to be more efficient than our current algorithms, depending on the search parameters. Our techniques can be used to obtain improved algorithms for some other related problems, such as allowing transpositions, and restricting the total accumulated differences $t_j - p_i$ to at most γ, where $\gamma < m\delta$. We are currently working on these problems.

References

[1] A. Andersson and M. Thorup. Tight(er) worst-case bounds on dynamic searching and priority queues. In *Proceedins of ACM STOC'00*, pages 335–342, 2000.

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

[3] D. Cantone, S. Cristofaro, and S. Faro. An efficient algorithm for δ-approximate matching with α-bounded gaps in musical sequences. In *Proceesings of WEA'05*, volume 3503 of *LNCS*, pages 428–439. Springer, 2005.

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

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

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

[7] K. Fredriksson and Sz. Grabowski. Efficient string matching with bounded gaps and character classes. 2005. Submitted.

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

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

[10] E. W. Myers. Approximate matching of network expression with spacers. *Journal of Computational Biology*, 3(1):33–51, 1996.

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

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

[13] N. Santoro and J. B. Sidney. Interpolation-binary search. *Inf. Process. Lett.*, 20(4):179–181, 1985.

[14] E. Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms*, 6(1–3):132–137, 1985.

[15] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[16] D. Willard. Searching unindexed and nonuniformly generated files in $\log \log n$ time. *SIAM J. Comput.*, 14(4):1023–1029, 1985.

[17] A. C. Yao and F. F. Yao. The complexity of searching an ordered random table. In *Proceedings of FOCS'76*, pages 173–177, 1976.

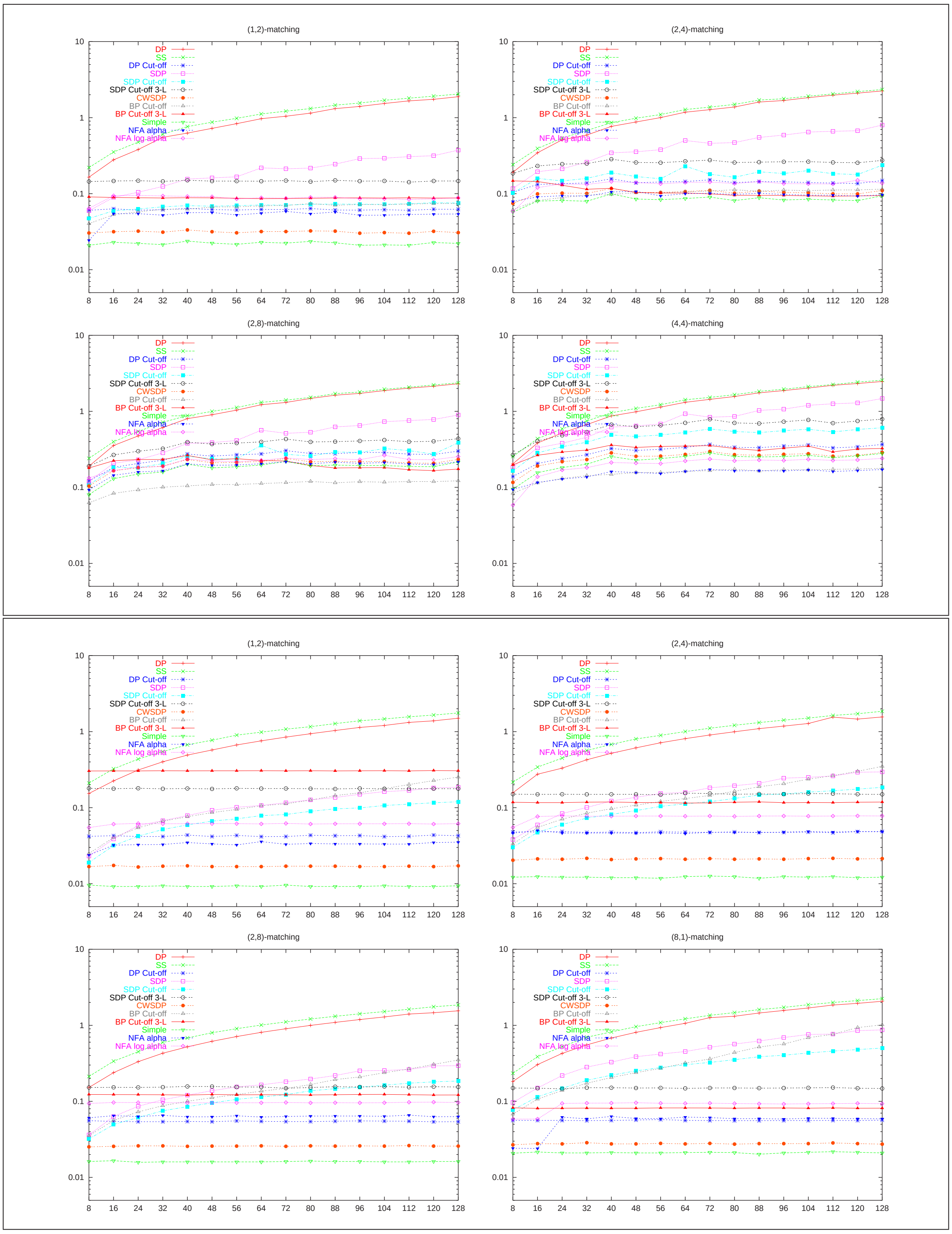


Figure 7: Running times in seconds for different pattern lengths. Top: MIDI data; bottom: random data ($\sigma = 128$). Note the logarithmic scale.

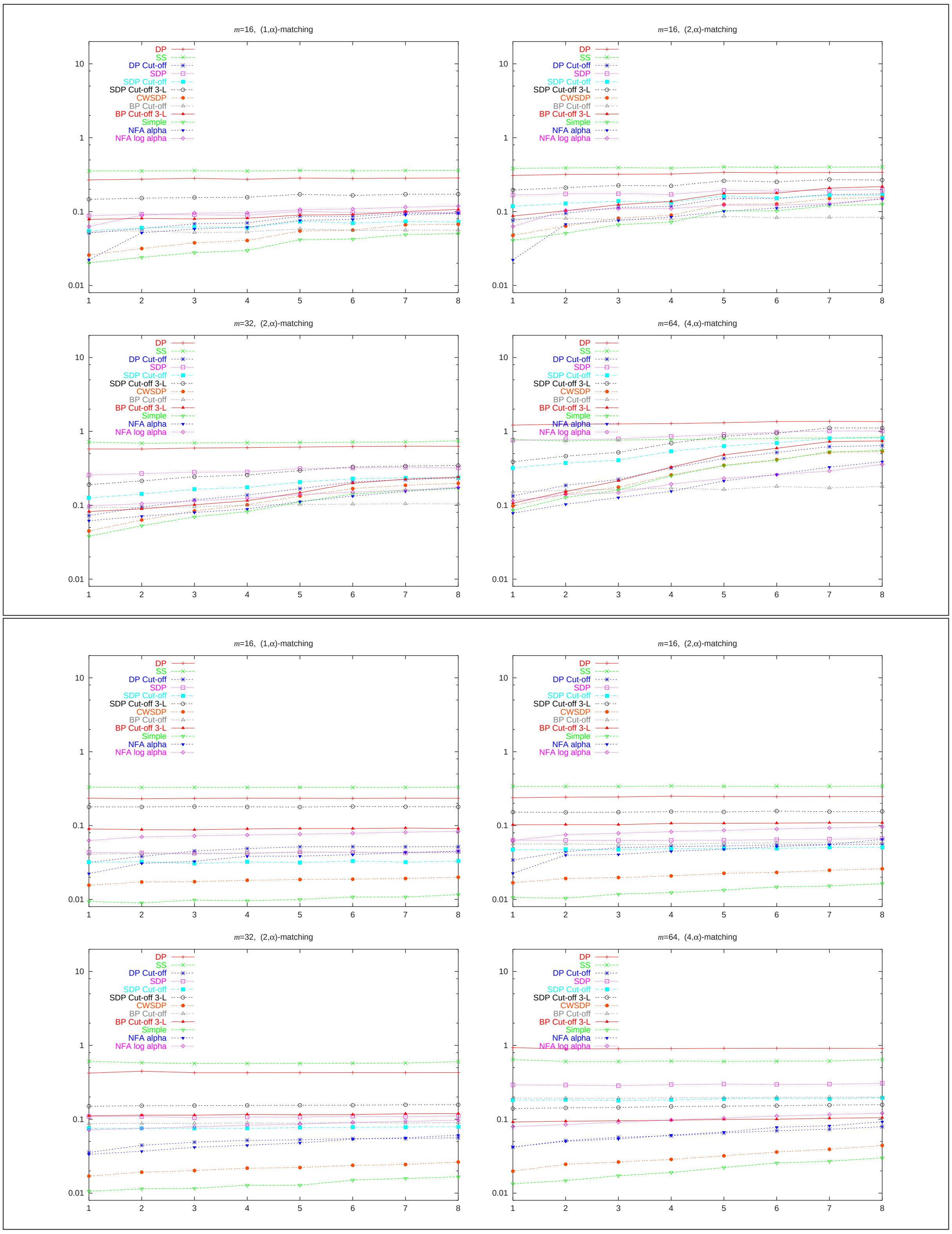


Figure 8: Running times in seconds for $\alpha = 1 \ldots 8$ and different pattern lengths. Top: MIDI data; bottom: random data ($\sigma = 128$). Note the logarithmic scale.