Efficient Algorithms for (δ, γ, α)-Matching

Kimmo Fredriksson[1]* and Szymon Grabowski[2]

[1] Department of Computer Science, University of Joensuu
PO Box 111, FIN–80101 Joensuu, Finland
kfredrik@cs.joensuu.fi
[2] Technical University of Łódź, Computer Engineering Department,
Al. Politechniki 11, 90–924 Łódź, Poland
sgrabow@kis.p.lodz.pl

Abstract. We propose new algorithms for (δ, γ, α)-matching. In this string matching problem we are given a pattern $P = p_0 p_1 \ldots p_{m-1}$ and a text $T = t_0 t_1 \ldots t_{n-1}$ over some integer alphabet $\Sigma = \{0 \ldots \sigma - 1\}$. The pattern symbol p_i matches the text symbol t_j iff $|p_i - t_j| \leq \delta$. The pattern P (δ, γ)-matches some text substring $t_j \ldots t_{j+m-1}$ iff for all i it holds that $|p_i - t_{j+i}| \leq \delta$ and $\sum |p_i - t_{j+i}| \leq \gamma$. Finally, in (δ, γ, α)-matching we also permit at most α length gaps (text substrings) between each matching text symbol. The only known previous algorithm runs in $O(mn)$ time. We give several algorithms that improve the average case up to $O(n)$ for small α, and the worst case to $O(\mathbf{min}\{mn, |\mathcal{M}|\alpha\})$ or $O(mn \log \gamma / w)$, where $\mathcal{M} = \{(i, j) \mid |p_i - t_j| \leq \delta\}$ and w is the number of bits in a machine word. We conclude with experimental results showing that the algorithms are very efficient in practice.

Keywords: approximate string matching, music information retrieval, bit-parallelism, sparse dynamic programming

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, γ, α)-matching [4] and its variations. In this problem, the pattern $p_0 p_1 \ldots p_{m-1}$ is allowed to match a substring of the text $t_0 t_1 \ldots t_{n-1}$ with α-limited gaps, and the respective pairs of matching characters' numerical values may differ only by δ, and the total sum of differences is limited to γ. Translating this model into a music (melody seeking) application, we can 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.

Previous work. There are many algorithms that solve some restricted variant of (δ, γ, α)-matching, such as δ-matching [3], (δ, γ)-matching [5, 6] and (δ, α)-matching [13, 1, 2, 8]. There are also algorithms that allow transpositions and insertions and deletions of symbols simultaneously with (δ, γ) or (δ, α)-matching [11, 12]. However, none of these algorithms can handle (δ, γ, α)-matching. We are aware of only one algorithm for (δ, γ, α)-matching problem [4]. This is based on dynamic programming, and runs in $O(nm)$ time.

* Supported by the Academy of Finland, grant 202281.

Our results. We improve the basic dynamic programming based algorithm [4] to run in $O(n\alpha\delta/\sigma)$ average time. We develop a simple sparse dynamic programming algorithm that runs in $O(n)$ average time, and in $O(\mathbf{min}\{mn, |\mathcal{M}|\alpha\})$ worst case time, where $\mathcal{M} = \{(i,j) \mid p_i =_\delta t_j\}$. Finally, we develop a bit-parallel dynamic programming algorithm that runs in $O(mn\log(\gamma)/w + n\delta)$ worst case time, where w is the number of bits in computer word. The average time of this algorithm is close to $O(n\log(\gamma)/w\ \alpha\delta/\sigma + n)$. The average case analyzes assume that α is small enough.

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 $\Sigma = \{0, 1, \ldots, \sigma-1\}$. The number of distinct symbols in the pattern is denoted by σ_p.

In δ-approximate string matching the symbols $a, b \in \Sigma$ match, denoted by $a =_\delta b$, iff $|a-b| \leq \delta$. Pattern P (δ,α)-matches the text substring $t_{i_0}t_{i_1}t_{i_2}\ldots t_{i_{m-1}}$, if $p_j =_\delta t_{i_j}$ for $j \in \{0,\ldots,m-1\}$, where $0 < i_{j+1} - i_j \leq \alpha + 1$. Finally, in (δ,γ,α)-matching we require also that $\sum |p_j - t_{i_j}| \leq \gamma$. If string A (δ,γ,α)-matches string B, we sometimes write $A =^{\alpha}_{\delta,\gamma} B$.

In all our analysis we assume uniformly random distribution of characters in T and P, and constant δ and σ. Note that $\gamma < m\delta$, as otherwise (δ,γ,α)-matching degenerates into (δ,α)-matching. It is also meaningless to have $\delta > \gamma$.

For the bit-parallel algorithms 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.

3 Dynamic programming

A straight-forward solution to (δ,γ,α)-matching is to use dynamic programming. The following recurrence can be used:

$$D_{i,j} = \begin{cases} D_{i-1,j'} + |p_i - t_j|, p_i =_\delta t_j \text{ AND } 0 < j - j' \leq \alpha + 1, \ \mathbf{min}\, D_{i-1,j'} \leq \gamma \\ \gamma + 1, \qquad\qquad \text{otherwise.} \end{cases} \quad (1)$$

If $D_{m-1,j} \leq \gamma$, then $P =^{\alpha}_{\delta,\gamma} t_h\ldots t_j$ for some h. The matrix D is simple to compute in $O(\alpha mn)$ time. As we are only interested in the matching text positions, the $O(mn)$ space complexity can be easily improved. Using row-wise computation only the current and the previous rows need to be in memory, and hence the space complexity is just $O(n)$. For column-wise computation the space complexity is $O(\alpha m)$ as up to $\alpha + 1$ columns have to be stored.

As shown in [4] the time complexity can be improved to $O(mn)$ using *min-queue* data structures [9]. However, in practical MIR applications α is usually so small that the simple brute-force evaluation is faster than using sophisticated data structures that have large (constant) overhead. Instead, we propose a simple cut-off trick that improves the average case.

3.1 Cut-off

We make the following observation: if $D_{i\ldots m-1,j-\alpha\ldots j} > \gamma$, for some i, j, then $D_{i+1\ldots m-1,j+1} > \gamma$. This is because there is no way the recurrence can introduce

Alg. 1 DPCO$(T, n, P, m, \delta, \gamma, \alpha)$.

```
1   for i ← 0 to α + 1 do for j ← 0 to m − 1 do D[i][j] ← γ + 1
2   for j ← 0 to m − 1 do C[j] ← −α − 1
3   D[0][0] ← |T[0] − P[0]|
4   if D[0][0] > δ then D[0][0] ← γ + 1
5   if D[0][0] ≤ γ then C[0] ← 0
6   top ← m − 1
7   for i ← 1 to n − 1 do
8       C′ ← C[0]
9       k ← i % (α + 2)
10      D[k][0] ← |T[i] − P[0]|
11      if D[k][0] > δ then D[k][0] ← γ + 1
12      if D[k][0] ≤ γ then C[0] ← i
13      for j ← 1 to top do
14          d ← |T[i] − P[j]|
15          min ← γ + 1
16          if d ≤ δ AND i − C′ ≤ α + 1 then
17              k′ ← (i − 1) % (α + 2)
18              min ← D[k′][j − 1]
19              for h ← max{0, i − α − 1} to i − 2 do
20                  k′ ← h % (α + 2)
21                  if D[k′][j − 1] < min then min ← D[k′][j − 1]
22          D[k][j] ← min + d
23          C′ ← C[j]
24          if D[k][j] ≤ γ then
25              C[j] ← i
26              if j = m − 1 then report match
27      while top ≥ 0 AND i − C[top] > α + 1 do top ← top − 1
28      if top < m − 1 then top ← top + 1
```

any other value for those matrix cells. In other words, if $p_0 \dots p_i$ does not (δ, γ, α)-match $t_h \dots t_{j-k}$ for any $k = 0 \dots \alpha$, then the match at the position $j + 1$ cannot be extended to $p_0 \dots p_{i+1}$. This can be utilized by keeping track of the highest row number *top* of the current column j such that $D_{top+1 \dots m-1, j-\alpha \dots j} > \gamma$, and computing the next column only up to row $top + 1$. For this sake we maintain an array C so that $C[i]$ gives the largest j such that $p_0 \dots p_i =^{\alpha}_{\delta,\gamma} t_h \dots t_j$. This is easy to do in $O(1)$ time per accessed matrix cell. Alg. 1 shows the complete pseudo code.

Now consider the average time of this algorithm. Computing a single cell $D_{i,j}$ costs $O(\alpha)$ in the worst case. However, this happens only if $p_0 \dots p_{i-1} =^{\alpha}_{\delta,\gamma} t_h \dots t_{j'}$ and $p_i =_{\delta} t_j$ for some $j' \geq j - \alpha - 1$, and otherwise the cost is just $O(1)$. Therefore on average each cell is computed in $O(\alpha\delta/\sigma)$ time. Maintaining *top* costs only $O(n)$ time in total, since it can be incremented only by one per text character, 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 $O(n \text{ avg}(top) \ \alpha\delta/\sigma)$. For $\gamma = \infty$ it can be shown that $\text{avg}(top) = O\left(\frac{\delta}{\sigma(1-\delta/\sigma)^{\alpha+1}}\right)$ [2]. This is $O(\alpha\delta/\sigma)$ for $\delta/\sigma < 1 - \alpha^{-1/(\alpha+1)}$, so the average time is at most $O(n(\alpha\delta/\sigma)^2)$. We have neglected the effect of γ, but by forcing the γ condition the time can only improve, hence our analysis is pessimistic. In the worst case the time is $O(\alpha mn)$, but this can be improved to $O(mn)$ as in [4], the only difference being that we need m queues, since we are computing column-wise (as opposed to row-wise in [4]).

4 Simple algorithm

In this section we will develop a variant of the Simple algorithm for (δ, α)-matching [7]. This performs very well on small (δ, γ, α).

Alg. 2 Simple$(T, n, P, m, \delta, \gamma, \alpha)$.

```
1   h ← 0
2   for i ← 0 to n − 1 do
3       M[i] ← γ + 1
4       d ← |T[i] − P[0]|
5       if d ≤ δ then
6           L1[h] ← i
7           G[h] ← d
8           h ← h + 1
9   for j ← 1 to m − 1 do
10      pn ← h; h ← 0;
11      for i ← 0 to pn − 1 do
12          g ← G[i]
13          for k ← L1[i] + 1 to min(L1[i] + α + 1, n − 1) do
14              d ← |T[k] − P[j]|
15              if d ≤ δ AND g + d ≤ γ then
16                  if M[k] ≤ γ then
17                      if g + d < M[k] then M[k] ← g + d
18                  else
19                      L2[h] ← k
20                      h ← h + 1
21                      M[k] ← g + d
22                  if j = m − 1 AND M[k] ≥ 0 then
23                      report match
24                      M[k] ← −1
25      if j < m − 1 then for i ← 0 to h − 1 do
26          k ← L2[i]
27          G[i] ← M[k]
28          M[k] ← γ + 1
29      Lt ← L1; L1 ← L2; L2 ← Lt;
```

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

$$L_0 = \{j \mid t_j =_\delta p_0\}. \tag{2}$$

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

$$L_i = \{j \mid p_i =_\delta t_j \text{ AND } D_{i-1,j'} + |p_i - t_j| \leq \gamma \text{ AND } j' \in L_{i-1} \text{ AND } 0 < j - j' \leq \alpha + 1\}. \tag{3}$$

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, and if so whether the sum of errors is still at most γ. When some j is appended into L_i, the corresponding matrix cell $D_{i,j}$ is also updated to hold the sum of errors for the matching pattern prefix $p_0 \ldots p_i$. Note that we put each j only once into L_i, but there can be up to $\alpha + 1$ different $j' \in L_{i-1}$ that may cause it. In the case that j is already in L_i we only update $D_{i,j}$ if the new sum is smaller. This takes $O(\alpha |L_{i-1}|)$ time. Alg. 2 shows the code.

Clearly, in the worst case the total length of all the lists is $\sum_i |L_i| = |\mathcal{M}|$, where $\mathcal{M} = \{(i, j) \mid p_i =_\delta t_j\}$, and hence the algorithm runs in $O(\alpha |\mathcal{M}|)$ worst case time. 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

$$O(\alpha |L_{i-1}|) = O(n\alpha^i (\delta/\sigma)^i) = O(n(\alpha\delta/\sigma)^i) \tag{4}$$

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 $O(n)$. Note that we did not use γ in this analysis, making it pessimistic.

4.1 Improving the worst case

As a theoretical option, we can improve the worst case of this algorithm to $O(\mathbf{min}\{mn, \alpha|\mathcal{M}|\})$. The idea is to avoid brute force handling of overlapping windows of size $\alpha + 1$. We make use of the min-queue data structure [9], similarly to the concept from [4] where the min-queue was used with plain dynamic programming.

For the current cell $D_{i+1,j}$, the keys in the queue are the values of $D_{i,j'}$, where $j' \in \{L_i \mid 0 < j - L_i < \alpha + 1\}$. For calculating $D_{i+1,j}$ it is enough to add its individual error to the minimum sum of errors from the queue. An algorithmic challenge is to update the queue quickly. For each processed cell only 0 or 1 values have to be inserted to the front of the queue and from 0 to $\alpha + 1$ deleted from the tail. Note however that only $O(1)$ cells (amortized) are inserted or deleted at each step. All the operations can then be done in $O(1)$ time with the min-queue data structure. This gives $O(\mathbf{min}\{mn, \alpha|\mathcal{M}|\})$ worst case time.

Finally, the $O(\alpha)$ factor can be removed by precomputing $\mathcal{M}$. This can be done in $O(\mathbf{min}\{|\mathcal{M}| + n, \delta n\})$ worst case time and $O(n(\delta\sigma_{\mathrm{p}}/\sigma + 1))$ average case time for integer alphabets (see Sec. 5). Having $\mathcal{M}$ available, we can avoid the brute force scanning for δ-matches. $\mathcal{M}$ can be stored e.g. in Johnson's data structure [10] which supports a homogeneous sequence of insertions and successor searches in $O(\log\log(mn/|\mathcal{M}|))$ time. This gives $O(|\mathcal{M}| \log\log(mn/|\mathcal{M}|))$ worst case time, but destroys the good average case because of the costly precomputation. Note that $O(|\mathcal{M}| + n)$ worst case algorithm is easy to obtain by simply scanning $\mathcal{M}$ linearly, but this then becomes also the average case.

5 Bit-parallel dynamic programming

We now show how the basic dynamic programming algorithm can be bit-parallelized. The algorithm is based on the bit-parallel dynamic programming algorithm for (δ, α)-matching [8]. All the interesting values in the matrix D are at most γ, and all other values can be represented as any value greater than γ. Hence $O(\log \gamma)$ bits per matrix cell is sufficient, and we can compute $O(w/\log\gamma)$ cells in parallel, where w is the number of bits in a machine word. Moreover, we show how to handle α up to $O(w/\log\gamma)$ efficiently. We obtain $O(mn \log\gamma/w)$ worst case time algorithm.

Each matrix cell is represented with

$$\ell = \lceil \log_2(2\gamma + 1) \rceil \tag{5}$$

bits, and number zero is represented (using ℓ bits) as $2^{\ell-1} - (\gamma+1)$. This representation has been used before e.g. for (δ, γ)-matching [5]. We still need an additional bit per cell, and hence each machine word packs

$$C = \lfloor w/(\ell + 1) \rfloor \tag{6}$$

cells, or *counters*. This representation solves three problems we are going to face shortly: (i) counter overflows can be handled in parallel; (ii) it is easy to check in parallel if some of the counters have exceeded γ; (iii) thanks to the additional bit it is easy to compute pair-wise minima over two sets of counters in parallel.

Assume then that in the preprocessing phase we have computed a helper matrix (whose efficient computation we will consider later) V:

$$V_{i,j} = \begin{cases} |p_i - t_j|, p_i =_\delta t_j \\ \gamma + 1, \quad \text{otherwise.} \end{cases} \tag{7}$$

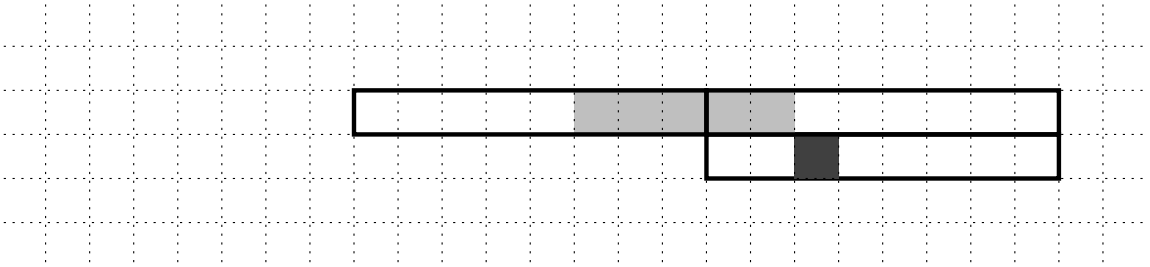

Figure 1. Tiling the dynamic programming matrix with $C = \lfloor w/(\ell+1) \rfloor \times 1$ vectors ($C = 8$). The dark gray cell of the current tile depends on the light gray cells of the two tiles in the previous row ($\alpha = 4$).

The computation of D will proceed column-wise, C columns at once. We adopt the notation $D^C_{i,j} = D_{i,jC\ldots(j+1)C-1}$, and analogously V^C for V, to make the parallelism explicit. Assume now that $\alpha < C$. The goal is then to produce $D^C_{i,j}$ from $V^C_{i,j}$, $D^C_{i-1,j}$ and $D^C_{i-1,j-1}$. $D^C_{i,j}$ does not depend on any other D^C element, according to the definition of D, and given our assumption that $\alpha < C$. Fig. 1 illustrates.

Now, according to the recurrence, the kth counter in $D^C_{i,j}$ is the sum of (i) the kth counter of $V^C_{i,j}$ (i.e. $|p_i - t_{jC+k}|$) and (ii) the minimum of the counters $k-\alpha-1 \ldots k-1$ in $D^C_{i-1,j}$ and the counter $k + C - \alpha - 1 \ldots C - 1$ in $D^C_{i-1,j-1}$ (i.e. the gap length to the previous match is at most α), see Fig. 1.

To compute item (ii) efficiently we assume that we have available function $M(x)$, that replaces each counter in x with the minima of the $\alpha + 1$ previous counters in x. The recurrence for D^C then becomes:

$$D^C_{i,j} = V^C_{i,j} + (M((D^C_{i-1,j} << w) \mid (D^C_{i-1,j-1} << (w - w \,\%\, C))) >> w), \qquad (8)$$

where for simplicity we have assumed that $M(x)$ can handle words of length $2w$. However, the above equation may cause counter overflow. To prevent this we use

$$D^C_{i,j} = (V^C_{i,j} + (M' \,\&\, \sim hmsk)) \mid (M' \,\&\, hmsk), \qquad (9)$$

instead, where

$$M' = M((D^C_{i-1,j} << w) \mid (D^C_{i-1,j-1} << (w - w \,\%\, C))) >> w, \qquad (10)$$

and $hmsk$ selects the ℓth bit of each counter. That is, $M' \,\&\, \sim hmsk$ clears the highest bit of each counter, so that the result can be safely added to $V^C_{i,j}$, and then $\mid (M' \,\&\, hmsk)$ restores the highest bit. This works correctly, as if the highest bit was set, then the sum is certainly greater than γ, and its exact value is not interesting anymore. The $(\ell+1)$th bit is not affected by the summation as the maximum value added is $\gamma + 1$.

Finally, to detect the possible pattern occurrences we must add our representation of zero ($2^{\ell-1} - (\gamma+1)$) to each counter. If some of the counters have still not overflowed, the corresponding text positions match. This can be detected as

$$q = \sim(((D^C_{m-1,j} \,\&\, \sim hmsk) + zeromsk) \mid D^C_{m-1,j}) \,\&\, hmsk, \qquad (11)$$

where $zeromsk$ has the value $2^{\ell-1} - (\gamma+1)$ in each counter position. Each set bit in q then indicates a pattern occurrence.

Alg. 3 $\mathrm{vmin}(x, y, msk)$.

```
1   F ← ((x | msk) − y) & msk
2   F ← F − (F >> ℓ)
3   return (x & ∼F) | (y & F)
```

Alg. 4 $M(x, y, \alpha, msk)$.

```
1   x ← (x << w) | (y << (w − w % C))
2   while α ≠ 0 do
3       r ← α % 2
4       α ← ⌊α/2⌋
5       x ← vmin(x, x << ((ℓ + 1)α), msk)
6       if r = 0 then continue
7       x ← vmin(x, x << (ℓ + 1), msk)
8   return (x << (ℓ + 1)) >> w
```

Consider now the computation of $M(x)$. One possible solution is to use table look-ups to compute it in constant time. 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 tests we used at most $w/2 = 16$ bit numbers for computing $M(x)$).

Another solution is to use repeated shifting and minimization. That is, assuming that $\mathrm{vmin}(x, y)$ computes pair-wise minima of the counter sets x and y, we compute $x \leftarrow \mathrm{vmin}(x, (x << (\ell+1)) \mid (\gamma+1))$ and repeat that α times, and then perform the final shift $x \leftarrow x << (\ell+1)$, which gives the desired result. The minimization can be done in $O(1)$ time [14], see Alg. 3. The total time for computing $M(x)$ is then $O(\alpha)$. This can be easily improved to $O(\log \alpha)$. Without loss of generality assume that α is a power of two. Instead of shifting one counter position at a time we first shift by $\alpha/2$ counter positions, then $\alpha/4$ counter positions, and so on $\log_2 \alpha$ times, performing the minimization at each step. Alg. 4 shows the code, handling the general case as well. This algorithm takes the counter sets $D^C_{i-1,j}$ and $D^C_{i-1,j-1}$, that can affect the current counters $D^C_{i,j}$, as parameter. For simplicity these are handled as a concatenated single word of $2w$ bits. Eq. (10) then becomes

$$M' = M(D^C_{i-1,j}, D^C_{i-1,j-1}, \alpha, msk), \tag{12}$$

where msk has every $(\ell+1)$th bit set, needed at the counter minimization.

We also need to compute V efficiently. This is easy with table look-ups as we have an integer alphabet. We first compute a table L, such that for all $c \in \Sigma$ the list $L[c]$ contains all the distinct characters p_i that satisfy $p_i =_\delta c$. Using this table we build a table V', which we will use as a terse representation of V, namely we have that $V'[p_i] = V_i$. This can be done by scanning through the text, and setting the jth counter of $V'[c]$ to $|c - t_j|$ for each $c \in L[t_j]$. This process takes $O(\lceil n/C \rceil \sigma_{\mathrm{P}} + m + \sigma + \delta\sigma_{\mathrm{P}} + \delta n) = O(\lceil n/C \rceil \sigma_{\mathrm{P}} + \delta n)$ worst case time. The probability that two characters δ-match is at most $(2\delta+1)/\sigma$, and hence the expected number of matching pattern characters for each text character is $O(\delta\sigma_{\mathrm{P}}/\sigma)$. Therefore, the average case complexity of the preprocessing is $O(\lceil n/C \rceil \sigma_{\mathrm{P}} + n(\delta\sigma_{\mathrm{P}}/\sigma + 1))$. Searching clearly takes only $O(\lceil n/C \rceil m) = O(\lceil n \log \gamma / w \rceil m)$ time if table look-ups are used for computing $M(x)$, and $O(\lceil n \log \alpha \log \gamma / w \rceil m)$ if Alg. 4 is used. For α larger than $O(w/\log \gamma)$ the search time must be multiplied by $O(\lceil \alpha \log \gamma / w \rceil)$.

Extended patterns. We note that this algorithm can be easily adapted to handle character classes, both in the pattern and the text. I.e. the pattern and text symbols

can be subsets of the alphabet, that is, $p_i, t_j \subseteq \Sigma$. The search algorithm does not change, we just change the definition (and preprocessing) of V:

$$V_{i,j} = \begin{cases} \textbf{min}\,|p-t|, p =_\delta t \text{ AND } p \in p_i, t \in t_j \\ \gamma + 1, \qquad \text{otherwise.} \end{cases} \tag{13}$$

5.1 Cut-off

The cut-off trick used in Sec. 3.1 obviously works for the bit-parallel algorithm as well. More formally, we define (for D^C) the maximum row top_j^C for the column j as:

$$top_j^C = \text{argmax}_i\{\ (\text{MatchMsk}(D^C_{i-1,j-1}) >> ((C-\alpha-1)(\ell+1))) \neq 0 \text{ OR} \tag{14}$$

$$(\text{MatchMsk}(D^C_{i-1,j}) << (\ell+1)) \neq 0\ \}, \tag{15}$$

where

$$\text{MatchMsk}(x) = \sim(((x\ \&\ \sim hmsk) + zeromsk)\ |\ x)\ \&\ hmsk. \tag{16}$$

Consider first the part (14). The rationale is as follows. When we are computing $D^C_{i,j}$, only the last $\alpha+1$ counters of $D^C_{i-1,j-1}$ that are at most γ can affect the counters in $D^C_{i,j}$. We therefore select the corresponding counter bits that indicate whether or not the sum have exceeded γ. However, since we are computing C columns in parallel, the $C-1$ first counters that have a value of at most γ in $D^C_{i-1,j}$ (15), i.e. in the previous row of the *current* set of columns, can affect the counters in $D^C_{i,j}$ as well. Obviously, this second part cannot be computed at column $j-1$. We solve this simply by computing the first part of top_j^C after the column $j-1$ have been computed, and when processing the column j, we increase top_j^C if needed according to the second part (15).

Alg. 5 gives the pseudo code. It uses the $O(\log\alpha)$ time algorithm for the $M(\cdot)$ function. The average case running time of this algorithm depends on what is the average value of top^C. For $C=1$ and $\gamma=\infty$ $\text{avg}(top^1) = O(\frac{\delta}{\sigma(1-\delta/\sigma)^{\alpha+1}})$, see Sec. 3.1. We are not able to analyze $\text{avg}(top^C)$ exactly, but we have trivially that $\text{avg}(top^1) \leq \text{avg}(top^C) \leq \text{avg}(top^1) + C - 1$, and hence the amortized average search time of Alg. 5 is at most $O((\lceil n/C\rceil\lceil\alpha\delta/\sigma\rceil + n)\log\alpha)$. The $\log\alpha$ factor can be easily removed with precomputation.

5.2 Lazy preprocessing

This can be still improved by interweaving the preprocessing and search phases, so that we initialize and preprocess V^C only for top_j^C length prefixes of the pattern for each j. At the time of processing the column j, we only know top_{j-1}^C, so we use an estimate $\varepsilon \times top_{j-1}^C$ for top_j^C, where $\varepsilon > 1$ is a small constant. If this turns out to be too small, we just increase the estimate and re-preprocess for the current column. The total preprocessing cost on average then becomes only $O(\lceil n/C\rceil\sigma_{top^C}\delta/\sigma + n)$, where σ_{top^C} is the alphabet size of top^C length prefix of the pattern. Hence the initialization time is at most $O(\lceil n/C\rceil\lceil\alpha\delta/\sigma\rceil + n)$ on average. This matches the search time, and together with the preprocessing the total is $O(\lceil n/C\rceil\lceil\alpha\delta/\sigma\rceil + n\lceil\alpha\delta/\sigma\rceil\delta/\sigma + n)$ on average.

Alg. 5 BPCO$(T, n, P, m, \delta, \gamma, \alpha)$.

```
1   ℓ ← ⌈log2(2γ + 1)⌉
2   f ← (w/(ℓ + 1))
3   Qb ← (f/2)(ℓ + 1)
4   zmsk ← (1 << (ℓ + 1)) − 1
5   for i ← 0 to σ − 1 do A[i] ← 0
6   for i ← 0 to m − 1 do
7       if A[P[i]] then continue
8       A[P[i]] ← 1
9       for j ← max{0, P[i] − δ} to min{P[i] + δ, σ − 1} do
10          Lt[j] ← Lt[j] ∪ {P[i]}
11  zero ← (1 << (l − 1)) − (γ + 1)
12  hhmsk ← 0
13  for i ← 0 to f − 1 do hhmsk ← hhmsk | (1 << ((i + 1)(ℓ + 1) − 1))
14  hmsk ← hhmsk >> 1
15  b ← (n + f − 1)/f
16  for i ← 0 to σ − 1 do
17      V[i] ← 0
18      if A[i] ≠ 0 then
19          for j ← 0 to b − 1 do V[i][j] ← hmsk
20  for i ← 0 to n − 1 do
21      for j ← 0 to |Lt[T[i]]| − 1 do
22          c ← Lt[T[i]][j]
23          V[c][i/f] ← V[c][i/f] & ∼(zmsk << ((i % f)(ℓ + 1)))
24          V[c][i/f] ← V[c][i/f] | (|c − T[i]| << ((i % f)(ℓ + 1)))
25  top ← m − 1
26  D1[0] ← V[P[0]][0]
27  for i ← 1 to top do
28      x ← M(D1[i − 1], hmsk, α, hhmsk)
29      D1[i] ← (V[P[i]][0] + (x & ∼hmsk)) | (x & hmsk)
30  zeromsk ← 0
31  for i ← 0 to f − 1 zeromsk ← zeromsk | (zero << (i(ℓ + 1)))
32  x ← ∼(((D1[m − 1] & ∼hmsk) + zeromsk) | D1[m − 1]) & hmsk
33  if x ≠ 0 then report matches
34  k ← ((f − α − 1)(ℓ + 1))
35  for j ← 1 to b − 1 do
36      D2[0] ← V[P[0]][j]
37      if top = 0 then
38          if (∼(((D2[0] & ∼hmsk) + zeromsk) | D2[0]) & hmsk) ≠ 0 then D1[0] ← hmsk; top ← top + 1
39      for i ← 1 to top do
40          x ← M(D2[i − 1], D1[i − 1], α, hhmsk)
41          D2[i] ← (V[P[i]][j] + (x & ∼hmsk)) | (x & hmsk)
42          x ← ∼(((D2[i] & ∼hmsk) + zeromsk) | D2[i]) & hmsk
43          if i = top AND top < m − 1 AND (x << (ℓ + 1)) ≠ 0 then D1[i] ← hmsk; top ← top + 1
44      if top = m − 1 AND x ≠ 0 then report matches
45      do  x ← (∼(((D2[top] & ∼hmsk) + zeromsk) | D2[top]) & hmsk) >> k
46          if x = 0 then top ← top − 1
47      while top ≥ 0 AND x = 0
48      if top < m − 1 top ← top + 1
49      Dt ← D1; D1 ← D2; D2 ← Dt
```

5.3 Multiple patterns

The algorithm has relatively high preprocessing cost $O(\delta n + \sigma_{\text{P}}\lceil n/C \rceil)$ in the worst case. 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 time increase only $O(\delta n + \sigma_{\text{P}}\lceil n/C \rceil + rm)$, 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, but the amortized preprocessing cost per pattern is considerably reduced. If r is small as compared to σ/δ, the search cost can be reduced by "superimposing" the patterns, that is we

define

$$V_{i,j} = \begin{cases} \mathbf{min}\,|p - t_j|, p =_\delta t_j \text{ AND } p \in p_i^h, h \in 0 \ldots r-1 \\ \gamma + 1, \qquad \text{otherwise,} \end{cases} \tag{17}$$

where we use the notation p_i^h to denote the ith symbol of the hth pattern. We then need only one search, but the potential matches must be verified. Superimposing works for the other algorithms as well.

5.4 Filtering

Alg. 5 is substantially more complex than its ancestor, the (δ, α)-matching algorithm [8]. In addition to being simpler, the previous algorithm achieves greater parallelism, the worst case search time being only $O(\lceil n/w \rceil m)$. However, we note that this algorithm (as any (δ, α)-matching algorithm) can be used as a filter, since it implicitly assumes that $\gamma = \infty$. The potential occurrences have to verified, which can be done using any of the algorithms given in this paper. The worst case time then becomes that of the verification algorithm.

6 Experimental results

We have run experiments to evaluate the performance of our algorithms. The experiments were run on Pentium4 2.4GHz with 512Mb of RAM, running GNU/Linux 2.4.20 operating system. We have implemented all the algorithms in C, and compiled with `icc 9.0`.

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. We also repeated the experiments on uniformly random data, with $\sigma = 128$. 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.

We experimented with the following algorithms:

BP Cut-off Bit-parallel dynamic programming with cut-off, Alg. 5 (without the lazy preprocessing);
BP Filter The (δ, α)-matching version of BP Cut-off [8] used as a filter, and Alg. 1 used for the verifications;
DP Cut-off Dynamic programming with cut-off, Alg. 1;
Simple Simple sparse dynamic programming, Alg. 2.

We omitted the results for basic dynamic programming based algorithms, since these are orders of magnitude slower. Fig. 2 shows the timings. Simple is the clear winner in most of the cases. BP Cut-off suffers from the large preprocessing cost, especially if the pattern alphabet is large. The same is true for the BP Filter, but this is more competitive in MIDI data, where the pattern alphabet is effectively very small.

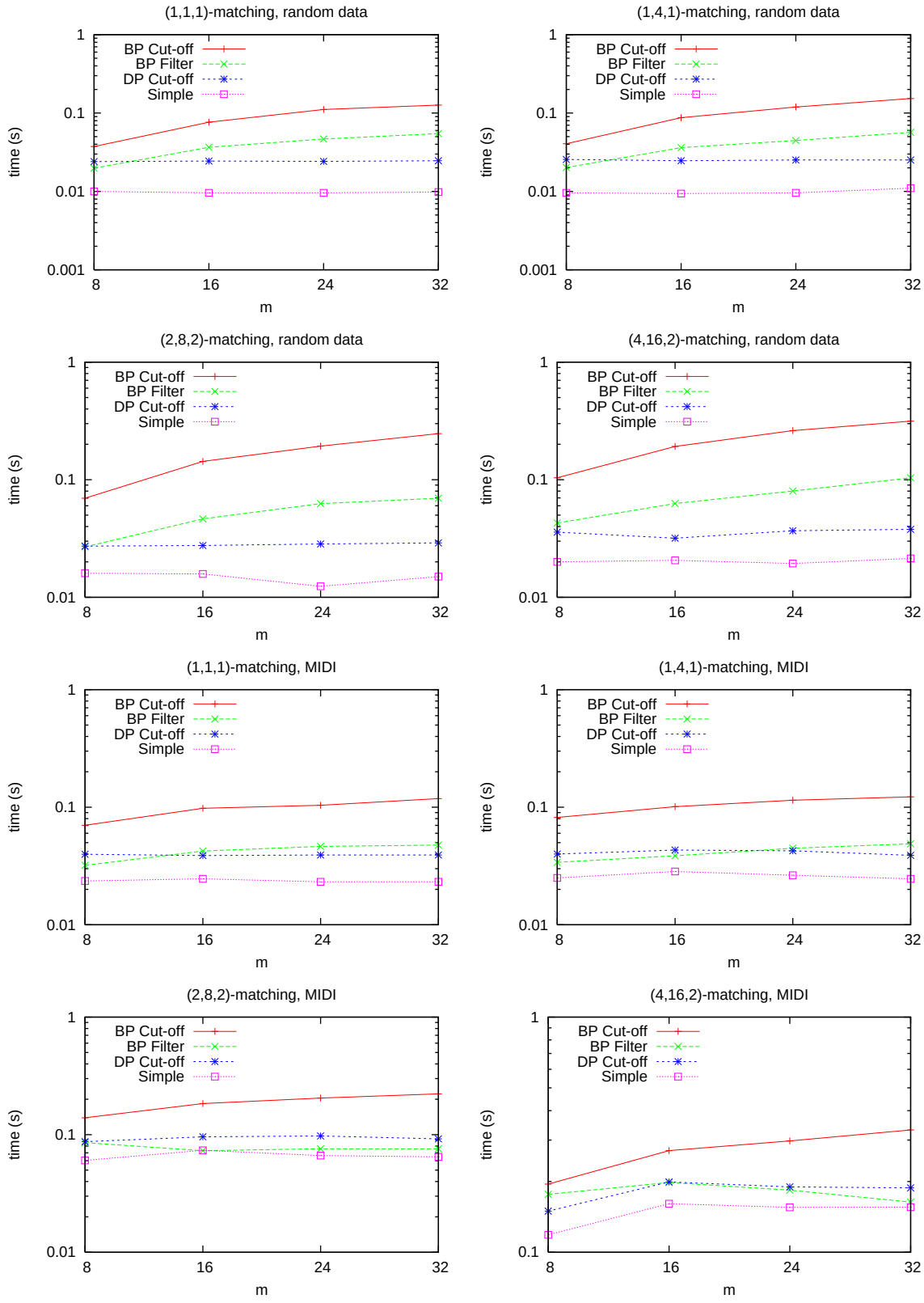

Figure 2. Execution times in seconds for $m = 8 \ldots 32$. Note the logarithmic scale.

7 Conclusions

We have presented new efficient algorithms for (δ, γ, α)-matching. Our algorithms are based on aborting the computation early where the match cannot be extended and on bit-parallelism. Besides having theoretically good worst and average case complexities, the algorithms are shown to work well in practice.

References

[1] 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, vol. 3503 of LNCS, Springer, 2005, pp. 428–439.

[2] 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.

[3] M. Crochemore, C. Iliopoulos, T. Lecroq, Y. Pinzon, W. Plandowski, and W. Rytter: *Occurrence and substring heuristics for δ-matching.* Fundamenta Informaticae, 56(1–2) 2003, pp. 1–15.

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

[5] 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) 2005, pp. 198–214.

[6] K. Fredriksson, V. Mäkinen, and G. Navarro: *Flexible music retrieval in sublinear time*, in Proceedings of the 10th Prague Stringology Conference (PSC'05), 2005, pp. 174–188.

[7] K. Fredriksson and Sz. Grabowski: *Efficient algorithms for pattern matching with general gaps and character classes*, in Proc. SPIRE'06, 2006, to appear.

[8] K. Fredriksson and Sz. Grabowski: *Efficient bit-parallel algorithms for (δ, α)-matching.*, in Proc. 5th Workshop on Efficient and Experimental Algorithms (WEA'06), LNCS 4007, 2006, pp. 170–181.

[9] H. Gajewska and R. E. Tarjan: *Dequeus with heap order.* Information Processing Letters, 22(4) 1986, pp. 197–200.

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

[11] V. Mäkinen: *Parameterized approximate string matching and local-similarity-based point-pattern matching*, PhD thesis, Department of Computer Science, University of Helsinki, Aug. 2003.

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

[13] 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) 2003, pp. 903–923.

[14] W. Paul and J. Simon: *Decision trees and random access machines*, in ZUERICH: Proc. Symp. Logik und Algorithmik, 1980, pp. 331–340.