Fast Two-Dimensional Approximate Pattern Matching*

Ricardo Baeza-Yates and Gonzalo Navarro

Dept. of Computer Science, University of Chile.
Blanco Encalada 2120, Santiago, Chile.
{rbaeza,gnavarro}@dcc.uchile.cl.

Abstract. We address the problem of approximate string matching in two dimensions, that is, to find a pattern of size $m \times m$ in a text of size $n \times n$ with at most k errors (substitutions, insertions and deletions). Although the problem can be solved using dynamic programming in time $O(m^2 n^2)$, this is in general too expensive for small k. So we design a filtering algorithm which avoids verifying most of the text with dynamic programming. This filter is based on a one-dimensional multi-pattern approximate search algorithm. The average complexity of our resulting algorithm is $O(n^2 k \log_\sigma m \;/m^2)$ for $k < m(m+1)/(5 \log_\sigma m)$, which is optimal and matches the best previous result which allows only substitutions. For higher error levels, we present an algorithm with time complexity $O(n^2 k/(w\sqrt{\sigma}))$ (where w is the size in bits of the computer word and σ is the alphabet size). This algorithm works for $k < m(m+1)(1-e/\sqrt{\sigma})$, where $e = 2.718...$, a limit which is not possible to improve. These are the first good expected-case algorithms for the problem. Our algorithms work also for rectangular patterns and rectangular text and can even be extended to the case where each row in the pattern and the text has a different length.

1 Introduction

A number of important problems related to string processing lead to algorithms for approximate string matching: text searching, pattern recognition, computational biology, audio processing, etc. Two dimensional pattern matching with errors has applications, for instance, in computer vision.

The *edit distance* between two strings a and b, $ed(a,b)$, is defined as the minimum number of *edit operations* that must be carried out to make them equal. The allowed operations are insertion, deletion and substitution of characters in a or b. The problem of *approximate string matching* is defined as follows: given a *text* of length n, and a *pattern* of length m, both being sequences over an alphabet Σ of size σ, find all segments (or "occurrences") in *text* whose edit distance to *pattern* is at most k, where $0 < k < m$. The classical solution is $O(mn)$ time and involves dynamic programming [19].

* Support from Fondecyt grants 1-95-0622 and 1-96-0881 are gratefully acknowledged.

Krithivasan and Sitalakshmi (KS) [14] proposed the following extension of edit distance for two dimensions. Given two images of the same size, the edit distance is the sum of the edit distance of the corresponding row images. This definition is justified when the images are transmitted row by row and there are not too many communication errors. On the other hand, it is not clear how to lift the row restriction (i.e. letting insertions and deletions along rows and columns) as then an approximate match is harder to define. Figure 1 gives an example.

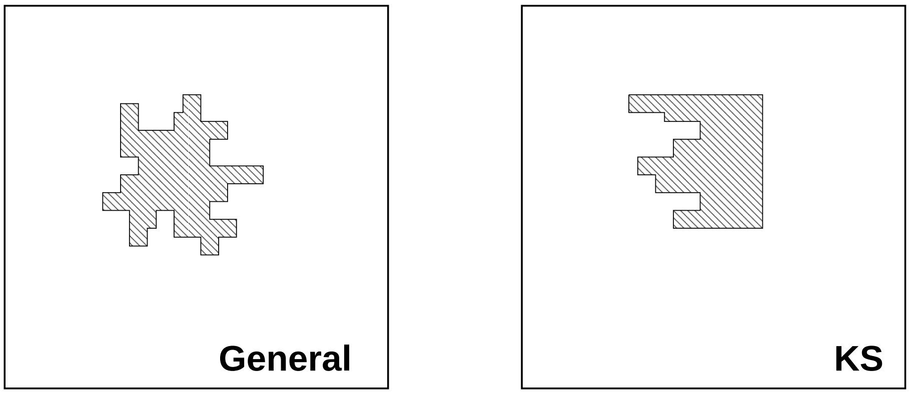

Fig. 1. Alternative error models.

Using this model they define an approximate search problem where a subimage of size $m \times m$ is searched into a large image of size $n \times n$, which they solve in $O(m^2 n^2)$ time using a generalization of the classical one-dimensional algorithm.

We use the same model and improve the expected case using a filter algorithm based in multiple one-dimensional approximate string matching, in the same vein of [9, 8, 7]. Our algorithm has $O(n^2 k \log_\sigma m \;/m^2)$ average-case behavior for $k < m(m+1)/(5 \log_\sigma m)$, using $O(m^2)$ space. This time matches the best known result for the same problem allowing only substitutions and is optimal [12], being the restriction on k only a bit more strict. For higher error levels, we present an algorithm with time complexity $O(n^2 k/(w\sqrt{\sigma}))$ (where w is the size in bits of the computer word), which works for $k < m(m+1)(1 - e/\sqrt{\sigma})$. We also show that this limit on k cannot be improved.

Given a two-dimensional string S, we denote as $S[i]$ its i-th row ($i \geq 1$), and $S[i][j]$ the j-th column of row i ($j \geq 1$). The two-dimensional strings we use are the pattern P and the text T.

2 Previous Work

The classical $O(mn)$ dynamic programming solution to the one-dimensional problem [19] keeps an array $C[0..m]$, which for each new text position $T[j]$ is updated to $C'[0..m]$ with the formula

$$C'[0] \leftarrow C[0],\ C'[i] \leftarrow \text{if } P[i] = T[j] \text{ then } C[i-1] \text{ else } 1+\min(C[i-1], C'[i-1], C[i])$$

and a match is reported whenever $C[m] \leq k$.

This solution was later improved by a number of algorithms. The different approaches can be divided in three main areas:

- Those that use cleverly the geometric properties of the dynamic programming matrix, e.g. [15, 21, 10]. These algorithms normally achieve $O(kn)$ time complexity in the worst or the average case.
- Those that filter the text, quickly leaving out most of the text and verifying only the areas that seem interesting, e.g. [20, 6]. They achieve sublinear expected time in many cases (e.g. $O(kn \log_\sigma m/m)$) for small k/m ratios.
- Those that parallelize the computation of a classical algorithm in the bits of computer words [22, 23, 4]. We call w the number of bits in the computer word, which is assumed to be $\Theta(\log n)$. These algorithms obtain in the best case a factor of $O(1/\log n)$ over their classical counterparts.

On the other hand, multi-pattern approximate search has only recently been considered. In [16], hashing is used to search thousands of patterns in parallel, although with only one error. In [5], extensions of [4] and [6] are presented based on superimposing automata. In [17], a counting filter is bit-parallelized to keep the state of many searches in parallel. Most multipattern algorithms consist of a filter which discards most of the text at low cost, and verify using dynamic programming the text areas that cannot be discarded. If the error level is low enough, the average number of verifications is so low that their total cost is of lower order and can be neglected. Otherwise the cost of verifications dominates and the algorithm is not useful, as it is as costly as plain dynamic programming.

Finally, the case of two dimensional approximate string matching usually considers only substitutions for rectangular patterns, which is much simpler than the general case with insertions and deletions. For substitutions, the pattern shape matches the same shape in the text (e.g. if the pattern is a rectangle, it matches a rectangle of the same size in the text). For insertions and deletions, instead, rows and/or columns of the pattern can match pieces of the text of different length.

If we consider matching the pattern with at most k substitutions, one of the best results on the worst case is due to Amir and Landau [2], which achieves $O((k + \log\sigma)n^2)$ time but uses $O(n^2)$ space. A similar algorithm is presented in Crochemore and Rytter [11]. Ranka and Heywood, on the other hand, solve the problem in $O((k + m)n^2)$ time and $O(kn)$ space. Amir and Landau also present a different algorithm running in $O(n^2 \log n \log\log n \log m)$ time. On average, the best algorithm is due to Karkkäinen and Ukkonen [12], with its analysis and space usage improved by Park [18]. The expected time is $O(n^2k/m^2\ \log_\sigma m)$ for

$$k \leq \left\lfloor \frac{m}{\lceil \log_\sigma(m^2) \rceil} \right\rfloor \frac{m}{2} - 1 \ \approx\ \frac{m^2}{4\log_\sigma m}$$

using $O(m^2)$ space ($O(k)$ space on average). This time result is optimal for the expected case.

Under the KS definition (i.e. allowing insertions and deletions along rows), Krithivasan [13] presents an $O(m(k+\log m)n^2)$ algorithm that uses $O(mn)$ space. This was improved (for $k < m$) by Amir and Landau [2] to $O(k^2n^2)$ worst case time using $O(n^2)$ space. Amir and Farach [1] also considered non-rectangular patterns achieving $O(k(k + \sqrt{m \log m}\sqrt{k \log k})n^2)$ time. This algorithm is very complicated, as it uses numerical convolutions.

3 Error Model for Two Dimensions

We assume that pattern and text are rectangular, of sizes $m_1 \times m_2$ and $n_1 \times n_2$ respectively (rows $\times$ columns). We use sometimes $M = m_1 m_2$ and $N = n_1 n_2$ as the size of the pattern and the text respectively. However, our algorithms can be easily extended to the more general case where each row in the pattern and the text has different length. For simplicity we only explain the rectangular case in this paper. Sometimes we even simplify more, considering the case $m_1 = m_2 = m$ and $n_1 = n_2 = n$.

In the KS error model we allow errors along rows, but errors cannot occur along columns. This means that, for instance, a single insertion cannot move all the characters of its column one position down. Or we cannot perform m_2 deletions along a row and eliminate the row. All insertions and deletions displace the characters of the row they occur in.

In this simple model every row is exactly where it is expected to be in an exact search. That is, we can see the pattern as an m_1-tuple of strings of length m_2, and each error is a one-dimensional error occurring in exactly one of the strings. Formally,

Definition: Given a pattern P of size $m_1 \times m_2$ and a text T of size $n_1 \times n_2$, we say that the pattern P *occurs in the text at position* (i,j) *with at most* k *errors* if

$$\sum_{r=1}^{m_1} led(T[i+r-1][1..j], P[r]) \quad \leq \quad k$$

where $led(t[1..j], pat) = \min_{i \in 1..j} ed(t[i..j], pat)$.

Observe that in this case the problem still makes sense for $k > m_2$, although it must hold $k < m_1 m_2$ (since otherwise every text position matches the pattern by performing $m_1 m_2$ substitutions).

The natural generalization of the classical dynamic programming algorithm for one dimension to the case of two dimensions was presented in [14]. Its complexity is $O(MN)$, which is also a natural extension of the $O(mn)$ complexity for one-dimensional text. The algorithm is presented in Figure 2 as it is the basic procedure for the verification phase of our filtering algorithm. Instead of the single column vector $C[j]$ of length $m+1$ used in [19], we have an $m_1 \times (m_2+1)$ matrix indexed by pattern rows and columns, $C[r][j]$, for $r \in 1..m_1,\ j \in 0..m_2$.

```
for i ← 1 to n1-m1
        --- initialize C ---
    for r ← 1 to m1
        for j ← 0 to m2
            C[r][j] ← j
        --- compute values for each text column j ---
    for j ← 1 to n2
        err ← 0
        for r ← 1 to m1
            for s ← 1 to m2
                if P[r][s] = T[i+r-1][j]
                   then C'[r][s] ← C[r][s-1]
                   else C'[r][s] ← 1 + min(C[r][s-1],C[r][s],C'[r][s-1])
            err ← err + C'[r][m2]
        exchange C and C'   --- just exchange pointers ---
        if err <= k then report match at (i,j)
```

Fig. 2. Two dimensional approximate matching by dynamic programming. The variable err sums up the errors along the rows of the pattern.

This algorithm uses $O(M)$ extra space, which is the only state information it needs to be started at any text position. Although Amir and Landau have an $O(k^2n^2)$ algorithm, notice that dynamic programming is always better if $k > m$, so depending on k we have to choose the best algorithm.

4 A Fast Algorithm on Average

We begin by proving a lemma which allows us to quickly discard large areas of the text.

Lemma: If the pattern occurs with k errors at position (i, j) in the text, and $r_1, r_2, ... r_s$ are s different rows in the range 1 to m_1, then

$$\min_{t=1..s} \{led(T[i + r_t - 1][1..j], P[r_t])\} \quad \leq \quad \lfloor k/s \rfloor \ .$$

Proof: Otherwise, $led(T[i + r_t - 1][1..j], P[r_t]) \geq 1 + \lfloor k/s \rfloor > k/s$ for all t. Just summing up the errors in the s selected rows we have strictly more than $s \times k/s = k$ errors and therefore a match is not possible.

The Lemma can be used in many ways. The simplest case is to set $s = 1$. This tells us that if we cannot find a row r of the pattern with at most k errors at text row i, then the pattern cannot occur at row $i - r + 1$. Therefore, we can search for *all* rows of the pattern at text row m_1. If we cannot find a match of any of the pattern rows with at most k errors, then no possible match begins at text rows $1..m_1$. There cannot be a match at text row 1 because pattern row m_1 was not found at text row m_1. There cannot be a match at text row 2 because pattern row $m_1 - 1$ was not found at text row m_1. Finally, there cannot be a match at text row m_1 because pattern row 1 was not found at text row m_1.

This shows that we can search only text rows $i \cdot m_1$, for $i = 1..\lfloor n_1/m_1 \rfloor$. Only in the case that we find a match of pattern row r at text position $(i \cdot m_1, j)$, we must verify a possible match beginning at text row $i \cdot m_1 - r + 1$. We must perform the verification from text column $j - m_2 - k + 1$ to j, using the dynamic programming algorithm. However, if $k > m_2$ we can start at $j - 2m_2 + 1$, since otherwise we would pay more than m_2 insertions, in which case it is cheaper to just perform m_2 substitutions. This verification costs $O(m_1 m_2^2) = O(m^3)$.

To avoid re-verifying the same areas due to overlapping verification requirements, we can force all verifications to be made in ascending row order and ascending column order inside rows. By remembering the state of the last verified positions we avoid re-verifying the same columns, this way keeping the worst case of this algorithm at $O(m^2n^2)$ cost instead of $O(m^3n^2)$.

Figure 3 shows how the algorithm works.

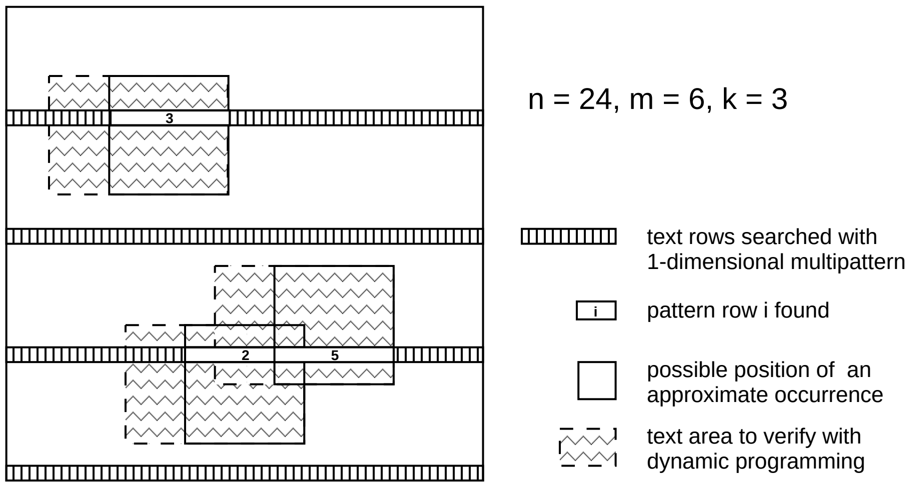

Fig. 3. Example of how the algorithm works.

We have still not explained how to perform a multi-pattern search for all the rows of the pattern at text rows numbered $i \cdot m_1$. We can use any available one-dimensional multi-pattern filtering algorithm. Each such algorithm has a different complexity and a maximum error level (i.e. k/m ratio) up to where it works well. For higher error levels, the filter triggers too many verifications, which dominate the search time.

A problem with this approach is that, if $k \geq m_2$ holds in our original problem, this filtration phase will be completely ineffective (since all text positions will match all the patterns, and all the text will be verified with dynamic programming). Even for $k < m_2$ the error level k/m_2 can be very high for the multipattern filter we choose.

This is where the s of the Lemma comes to play. We can search, instead of all text rows of the form $i \cdot m_1$, all text rows of the form $i \cdot \lfloor m_1/2 \rfloor$, for all patterns, with $\lfloor k/2 \rfloor$ errors. This corresponds to $s = 2$. If we find nothing at rows $i \cdot \lfloor m_1/2 \rfloor$ and $(i+1) \cdot \lfloor m_1/2 \rfloor$, then no occurrence can be found at text rows $(i-1) \cdot \lfloor m_1/2 \rfloor + 1$ to $i \cdot \lfloor m_1/2 \rfloor$, because that occurrence has already two rows with more than $k/2$ errors. In general, we can search only the text rows numbered $i \cdot \lfloor m_1/s \rfloor$, for all the patterns, with $\lfloor k/s \rfloor$ errors. In the extreme case, we can search all text rows with $\lfloor k/m_1 \rfloor$ errors (which is always $< m_2$ and therefore filtering is in principle possible).

There is another alternative way to use s, which is to search only the first $\lceil m_1/s \rceil$ rows of the pattern with k errors and consider the text rows of the form $i \cdot \lfloor m_1/s \rfloor$. That is, reduce the number of patterns instead of reducing the error level (this is because the tolerance to errors of some filters is reduced as the number of patterns grows). This alternative, however, is not promising since we pay s more times searches of $(1/s)$-th of the patterns. If the search cost for r patterns is $C(r)$, we pay $sC(r/s)$. The aim of any multi-pattern matching algorithm is precisely that $C(r) < sC(r/s)$ (since the worst thing that can happen is that searching for r patterns costs the same as r searches for one pattern, i.e. $C(r) = sC(r/s)$).

5 Average Case Analysis

Once we have selected a given one-dimensional multipattern search algorithm to support our two-dimensional filter, two values of the one-dimensional algorithm influence the analysis of the two-dimensional filter:

- $C(m,k,r)$, which is the cost per text character to search r patterns of length m with k errors. Notice that in our case, $m = m_2$ and $r = m_1$. Hence, the cost to search a text row with this algorithm is $n_2 C(m_2, k, m_1)$.
- $L(m,r)$, which is the maximum acceptable value for k/m up to where the one-dimensional algorithm works. That is, the cost of the search is $C(m,k,r)$ per text character, plus the verifications. If the error level is low enough (i.e. $k/m < L(m,r)$), the number of those verifications is so low that their cost can be neglected. Otherwise the cost of verifications dominates and the algorithm is not useful, as it is as costly as plain dynamic programming and our whole scheme does not work. Again, in our case, $m = m_2$ and $r = m_1$.

Given a multi-pattern search algorithm, our search strategy for the two-dimensional filter is as follows. If we search with $\lfloor k/s \rfloor$ errors, it must hold

$$\frac{\lfloor k/s \rfloor}{m_2} < L(m_2, m_1) \quad \Longrightarrow \quad s = \left\lfloor \frac{k}{m_2 L(m_2, m_1)} \right\rfloor \, . \qquad (1)$$

Since we traverse only the text rows of the form $i \cdot \lfloor m_1/s \rfloor$, we work on $O(n_1 s/m_1)$ rows, and therefore our total complexity to filter the text is

$$O(n_1 s/m_1 \; n_2 C(m_2, k/s, m_1)) \;\; = \;\; O\left(\frac{Nk}{M} \; \frac{C(m_2, m_2 L(m_2, m_1), m_1)}{L(m_2, m_1)}\right) \;, \quad (2)$$

where we recall that L has been selected so that the cost of verifications has, on average, lower order and therefore we neglect verification costs. The algorithm is applicable when it holds $s \leq m_1$, i.e. for

$$k < m_2(m_1 + 1) L(m_2, m_1) \;, \qquad (3)$$

since if it requires $s > m_1$, this means that the error level is too high even if we search all rows of the text ($s = m_1$).

We consider specific multi-pattern algorithms now, each one with a given C and L functions. As we only reference the algorithms, we do not include here their analysis leading to C and L, which is done in the original papers.

- **Exact Partitioning [5]** can be implemented such that $C(m, k, r) = O(1)$ (i.e. linear search time). For our $O(m_1 m_2^2) = O(rm^2)$ verification costs, we have $L(m, r) = 1/\log_\sigma(m^3 r^2)$. Therefore, using this algorithm we would select (Eq. (1))

$$s \;\; = \;\; \left\lfloor \frac{k \log_\sigma(m_1^2 m_2^3)}{m_2} \right\rfloor \;\; = \;\; \left\lfloor \frac{5k \log_\sigma m}{m} \right\rfloor \;,$$

our average search cost would be (Eq. (2))

$$O\left(\frac{Nk \log_\sigma \max(m_1, m_2)}{M}\right) \;\; = \;\; O\left(\frac{n^2 k \log_\sigma m}{m^2}\right)$$

and the algorithm would be applicable for $k < m_2(m_1+1)/\log_\sigma(m_1^2 m_2^3) = m(m+1)/(5 \log_\sigma m)$ (Eq. (3)).

- **Superimposed Automata [5]** has $L(m, r) = 1 - e/\sqrt{\sigma}$ (where $e = 2.718...$), and $C(m, k, r) = O(mr/(\sigma w(1 - k/m)))$ in its best version (automaton partitioning). Therefore, we have (Eq. (1))

$$s \;\; = \;\; \left\lfloor \frac{k}{m_2(1 - e/\sqrt{\sigma})} \right\rfloor \;\; = \;\; \left\lfloor \frac{k}{m(1 - e/\sqrt{\sigma})} \right\rfloor$$

the average complexity is (Eq. (2))

$$O\left(\frac{Nk}{M(1 - e/\sqrt{\sigma})} \; \frac{m_2 m_1}{\sqrt{\sigma} we}\right) \;\; = \;\; O\left(\frac{Nk}{\sqrt{\sigma} w}\right) \;\; = \;\; O\left(\frac{n^2 k}{\sqrt{\sigma} w}\right)$$

and the algorithm is applicable for $k < m_2(m_1+1)(1 - e/\sqrt{\sigma}) \;=\; m(m+1)(1 - e/\sqrt{\sigma})$ (Eq. (3)).

- **Counting [17]** has $L(m,r) = e^{-m/\sigma}$ and $C(m,k,r) = O(r/w \ \log m)$. Therefore, using this algorithm we would select (Eq. (1))

$$s = \left\lfloor \frac{ke^{m_2/\sigma}}{m_2} \right\rfloor = \left\lfloor \frac{ke^{m/\sigma}}{m} \right\rfloor ,$$

  the average search cost would be (Eq. (2))

$$O\left(\frac{Nke^{m_2/\sigma}}{M}\ \frac{m_1 \log m_2}{w}\right) = O\left(\frac{Nke^{m_2/\sigma}\log m_2}{m_2 w}\right) = O\left(\frac{n^2 ke^{m/\sigma}\log m}{mw}\right)$$

  and the algorithm would be applicable for $k < m_2(m_1+1)e^{-m_2/\sigma} = m(m+1)e^{-m/\sigma}$ (Eq. (3)).
  Notice that this algorithm is asymmetric with respect to the shape of the pattern, i.e. it works better on tall patterns than on wide ones. This is because its cost formula and error level are not symmetric in terms of m and r as the previous ones.
- **One Error [16]** can only search with $k = 1$ errors (i.e. $L(m,r) = 2/m$), with time cost $C(m,k,r) = m$. Therefore we must have $s = \lfloor k/2 \rfloor + 1$, which means that we can only apply the algorithm for $k < 2m_1$. In this case, the complexity would be

$$O\left(\frac{Nk}{M}\ \frac{m_2 m_2}{2}\right) = O\left(\frac{Nkm_2}{m_1}\right) = O(n^2 k) .$$

  This algorithm is asymmetric with respect to the error level it tolerates, also preferring taller rather than wider patterns.

The best algorithm on average turns out to be a hybrid. Counting is the best option for small patterns (i.e. $me^{-m/\sigma}/\log_2 m > \sqrt{\sigma}$), superimposed automata is the best option for intermediate patterns (i.e. $m^2/\log_2 m < w\sqrt{\sigma}/\log_2 \sigma$), and exact partitioning is the best option for larger patterns. The combined complexity is therefore

$$O\left(\frac{n^2 k \log m}{mw \max(m/w \log \sigma, \sqrt{\sigma}\log m/m, e^{-m/\sigma})}\right) .$$

As m grows, the best (and optimal) complexity is given by the exact partitioning, $O(n^2 k \log_\sigma m \ /m^2)$. However, this is true for $k < m(m+1)/(5\log_\sigma m)$, because otherwise the verification phase dominates. Once $s = 1$ and we cannot reduce the error level by reducing s (i.e. by searching on more rows), the approach most resistant to the error level is superimposed automata, which works up to $k < m(m+1)(1 - e/\sqrt{\sigma})$ (at that point its cost is $O(m^2 n^2/(w\sqrt{\sigma}))$, very close to simple dynamic programming, and the verification time becomes dominant).

Moreover, we prove in [4] that if $k/m_2 \geq 1 - e/\sqrt{\sigma}$ the number of text positions matching the pattern is high. Therefore, the limit for automaton partitioning is not just the limit of another filtering algorithm, but the true limit up to where it is possible at all to filter the text. In this sense, this filter has optimal tolerance to errors.

We summarize our results in Figure 4, where the best algorithm for each case is presented.

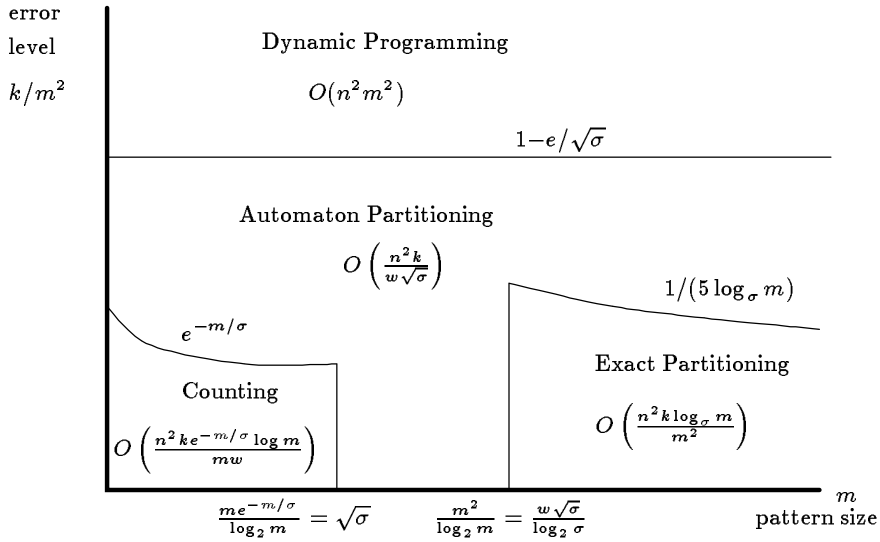

Fig. 4. The best algorithm with respect to the pattern length and error level. The complexity of each algorithm is also included.

6 Concluding Remarks

We present the first filtering algorithm for two dimensional approximate string matching allowing also insertions and deletions. This filter avoids verifying most of the text with the expensive dynamic programming algorithm, and is based on a one-dimensional multi-pattern approximate search algorithm. Our analysis gives the complexity of the filtering algorithm, obtaining expected case time $O(n^2k\log_\sigma m/m^2)$ for $k < m^2/(5\log_\sigma m)$. This time is optimal on average [12].

The edit distance that we use is simplified (row-wise) and does not model well simple cases of approximate matching in other settings. For example, we could have a match that only has the middle row of the pattern missing. In the KS definition (which we use), the edit distance would be $O(m^2)$ if all pattern rows are different. Intuitively, the right answer should be m, because only m characters were deleted in the pattern. We are currently working on more general error models [3], but as they are more general, the search complexity should be higher.

References

1. A. Amir and M. Farach. Efficient 2-dimensional approximate matching of non-rectangular figures. In *Proc. SODA'91*, pages 212–223, 1991.
2. A. Amir and G. Landau. Fast parallel and serial multidimensional approximate array matching. *Theoretical Computer Science*, 81:97–115, 1991.
3. R. Baeza-Yates. Similarity in two dimensional strings. Dept. of Computer Science, University of Chile, 1996.
4. R. Baeza-Yates and G. Navarro. A faster algorithm for approximate string matching. In *Proc. CPM'96*, LNCS 1075, pages 1–23, 1996. `ftp://ftp.dcc.uchile.cl/-pub/users/gnavarro/cpm96.ps.gz`.
5. R. Baeza-Yates and G. Navarro. Multiple approximate string matching. In *Proc. WADS'97*, LNCS 1272, pages 174–184, 1997. `ftp://ftp.dcc.uchile.cl/pub/-users/gnavarro/wads97.ps.gz`.
6. R. Baeza-Yates and C. Perleberg. Fast and practical approximate pattern matching. In *Proc. CPM'92*, LNCS 644, pages 185–192, 1992.
7. R. Baeza-Yates and M. Régnier. Fast two dimensional pattern matching. *Information Processing Letters*, 45:51–57, 1993.
8. T. Baker. A technique for extending rapid exact string matching to arrays of more than one dimension. *SIAM Journal on Computing*, 7:533–541, 1978.
9. R. Bird. Two dimensional pattern matching. *Inf. Proc. Letters*, 6:168–170, 1977.
10. W. Chang and J. Lampe. Theoretical and empirical comparisons of approximate string matching algorithms. In *Proc. CPM'92*, LNCS 644, pages 172–181, 1992.
11. M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, Oxford, UK, 1994.
12. J. Karkkäinen and E. Ukkonen. Two and higher dimensional pattern matching in optimal expected time. In *Proc. SODA'94*, pages 715–723. SIAM, 1994.
13. K. Krithivasan. Efficient two-dimensional parallel and serial approximate pattern matching. Technical Report CAR-TR-259, University of Maryland, 1987.
14. K. Krithivasan and R. Sitalakshmi. Efficient two-dimensional pattern matching in the presence of errors. *Information Sciences*, 43:169–184, 1987.
15. G. Landau and U. Vishkin. Fast string matching with k differences. *J. of Computer Systems Science*, 37:63–78, 1988.
16. R. Muth and U. Manber. Approximate multiple string search. In *Proc. CPM'96*, LNCS 1075, pages 75–86, 1996.
17. G. Navarro. Multiple approximate string matching by counting. In *Proc. WSP'97*, pages 125–139, 1997. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/-wsp97.1.ps.gz`.
18. K. Park. Analysis of two dimensional approximate pattern matching algorithms. In *Proc. CPM'96*, LNCS 1075, pages 335–347, 1996.
19. P. Sellers. The theory and computation of evolutionary distances: pattern recognition. *J. of Algorithms*, 1:359–373, 1980.
20. E. Sutinen and J. Tarhio. On using q-gram locations in approximate string matching. In *Proc. ESA'95*, LNCS 979, 1995.
21. Esko Ukkonen. Finding approximate patterns in strings. *J. of Algorithms*, 6:132–137, 1985.
22. S. Wu and U. Manber. Fast text searching allowing errors. *CACM*, 35(10):83–91, October 1992.
23. S. Wu, U. Manber, and E. Myers. A sub-quadratic algorithm for approximate limited expression matching. *Algorithmica*, 15(1):50–67, 1996.