New Models and Algorithms for Multidimensional Approximate Pattern Matching

RICARDO BAEZA-YATES[1], *Dept. of Computer Science, University of Chile. Blanco Encalada 2120, Santiago, Chile.*
rbaeza@dcc.uchile.cl

GONZALO NAVARRO[1], *Dept. of Computer Science, University of Chile. Blanco Encalada 2120, Santiago, Chile.*
gnavarro@dcc.uchile.cl

ABSTRACT: We focus on how to compute the edit distance (or similarity) between two images and 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 (character substitutions, insertions and deletions). Pattern and text are matrices over an alphabet of size σ. We present new models and give the first sublinear time search algorithms for the new and the existing models.

The only existing model just considers errors along one dimension. The associated approximate search algorithms use dynamic programming and are relatively expensive ($O(m^2n^2)$ or $O(k^2n^2)$). For this model we present a filtering algorithm which avoids verifying most of the text with dynamic programming. This filter is based on one-dimensional multipattern approximate searching. The average complexity of our resulting algorithm is $O(n^2k\log_\sigma m\,/m^2)$ for $k < m(m+1)/(5\log_\sigma m)$, which is optimal and matches the best previous result that allows only substitutions. We present other slower filtration algorithms that however work for higher error levels.

We then consider more general models to compare images. We present new similarity measures and the algorithms to compute them. We then focus on one of the models, which allows the errors to occur along any dimension, and extend it to the general case where pattern and text are d-dimensional. This edit distance can be computed in $O(d!m^{2d})$ time and $O(d!m^{2d-1})$ space. We also present the first sublinear-time (on average) searching algorithm (i.e. not all text cells are inspected), which is $O(kn^d/m^{d-1})$ time for $k < (m/(d(\log_\sigma(m/d))))^{d-1}$.

Keywords: Pattern matching in images, edit distance, Levenshtein distance

1 Introduction

Approximate pattern matching is the problem of finding a pattern in a text allowing errors (insertions, deletions, substitutions) of characters. 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.

[1]This work has been supported in part by Fondecyt grant 1-990627.

Approximate two dimensional pattern matching has applications, for instance, in computer vision (i.e. searching a subimage inside a large image) and OCR. In three dimensions, the problem has applications in some types of medical data (e.g. MRI brain scans) and in biocomputing (e.g. detecting protein patterns on the surface of three dimensional virus reconstructions).

For one dimension this problem is well-known, and is modeled using the edit distance. 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 T of length n, and a pattern P of length m, both being sequences over an alphabet Σ of size σ, find all segments (or "occurrences") in T whose edit distance to P is at most k, where $0 < k < m$. The classical solution is $O(mn)$ time and involves dynamic programming [32].

Krithivasan and Sitalakshmi (KS) [24] proposed a simple extension to 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 (e.g. photocopy images, where most errors come from the mechanical traction mechanism along one dimension only, or images transmitted by fax), but it is not appropriate otherwise. 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 $O(m^2n^2)$ time using a generalization of the classical one-dimensional algorithm.

Using this model we improve the expected case using a filter algorithm based on multiple one-dimensional approximate string matching, in the same vein of [14, 13, 12]. Our algorithm has $O(n^2k \log_\sigma m\ /m^2)$ 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 [22], being the restriction on k only a bit more strict. For higher error levels, we present an algorithm with time complexity $O(n^2k/(w\sqrt{\sigma}))$ (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.

However, for many other problems, the KS distance does not reflect well simple cases of approximate matching in different settings. For example, we could have a match that only has the middle row of the pattern missing. In the definition above, the edit distance would be $O(m^2)$ if all pattern rows are different. Intuitively, the right answer should be at most $2m$, because only m characters were deleted in the pattern and m characters are inserted at the bottom.

In this paper we extend the edit distance to two dimensions lifting the problem just mentioned and also extending the edit distance to images of different shapes. We define different distances and give algorithms to compute them, as well as the associated approximate search algorithms.

Among the more general extensions that we define, we focus in the RC model, where the errors can occur along rows or columns at any time. This model is much more robust and useful for more applications. We extend the model to d dimensions and present an edit distance algorithm with time complexity $O(d!m^{2d})$. We also give a new filtering algorithm that allows quickly discarding large parts of the text that cannot contain a match. This algorithm searches the pattern in average time $O(kn^d/m^{d-1})$ for $k < (m/(d(\log_\sigma(m/d))))^{d-1}$. After that error level the filter changes its cost but remains better than dynamic programming for $k \leq m^{d-1}/(d(\log_\sigma(m/d)))^{(d-1)/d}$.

This paper is organized as follows. First we discuss the basic concepts and previous work on pattern matching with errors and image similarity. Next, we consider the basic KS model and give new filters to speed up the search. In Section 4 we introduce new notions of similarity between two-dimensional images, together with algorithms to compute the edit distance. Section 5 presents how to search a pattern in a text under the new model. Then we extend one of the models to more dimensions and give fast filtering algorithms for approximate searching under that model. Finally, we give our conclusions. This work is an integrated and revised version of [7, 9, 28].

2 Basics and Previous Work

We give in this section some basic concepts and review the previous work in the area. We define some terminology first. Given a one dimensional string S we use S_i to denote its i-th character, the first one corresponding to $i = 1$. $S_{i..j}$ denotes the substring starting at character i and ending at character j, both inclusive. A character of a two-dimensional string S is addressed as $S_{i,j}$, meaning the character at row i and column j. Similarly, rows and columns can be extracted as $S_{i,j_1..j_2}$ and $S_{i_1..i_2,j}$, respectively, and even sub-matrices such as $S_{i_1..i_2,j_1..j_2}$.

2.1 One Dimensional Approximate String Matching

The classical dynamic programming algorithm [29] to compute the edit distance between two one-dimensional strings A and B of length m and n computes a matrix $C_{0..m,0..n}$. The value $C_{i,j}$ holds the edit distance between $A_{1..i}$ and $B_{1..j}$. The construction algorithm is as follows

$$\begin{array}{lcl} C_{i,0} & \leftarrow & i \ \ , \ \ C_{0,j} \ \leftarrow \ j \\ C_{i,j} & \leftarrow & \text{if } A_i = B_j \text{ then } C_{i-1,j-1} \quad \text{else } 1 + \min(C_{i-1,j-1}, C_{i-1,j}, C_{i,j-1}) \end{array}$$

and the distance $ed(A, B)$ is the final value of $C_{m,n}$. The rationale of the formula is that if $A_i = B_j$ then the cost to convert $A_{1..i}$ into $B_{1..j}$ is that of converting $A_{1..i-1}$ into $B_{1..j-1}$. Otherwise we have to make one error and select among three choices: (a) convert $A_{1..i-1}$ into $B_{1..j-1}$ and replace A_i by B_j, (b) convert $A_{1..i-1}$ into $B_{1..j}$ and delete A_i, and (c) convert $A_{1..i}$ into $B_{1..j-1}$ and insert B_j.

This algorithm takes $O(mn)$ space and time. It is easily adapted to search a pattern P in a text T allowing up to k errors [32]. In this case we want to report all the text positions j such that a suffix of $T_{1..j}$ matches P with at most k errors. This time the construction formula is

$$\begin{array}{lcl} C_{i,0} & \leftarrow & i \ \ , \ \ C_{0,j} \ \leftarrow \ 0 \\ C_{i,j} & \leftarrow & \text{if } P_i = T_j \text{ then } C_{i-1,j-1} \quad \text{else } 1 + \min(C_{i-1,j-1}, C_{i-1,j}, C_{i,j-1}) \end{array}$$

where the only change is that a pattern of length zero matches with no errors at any text position. All the positions j such that $C_{m,j} \leq k$ are reported. This takes $O(mn)$ time. The space can be reduced to $O(m)$ by noticing that only the old and new column of the matrix need to be stored.

This solution was later improved by a number of algorithms [27]. One of special interest to this work is a filtering algorithm [34, 11, 10]. A *filter* is a fast algorithm that can discard most of the text by checking a necessary condition. All filters have a maximum error level up to where they are useful. This filter cuts P in $k+1$ pieces. Any occurrence with up to k errors must contain one of those pieces unchanged. This is obvious since k errors cannot alter the $k+1$ pieces given that the edit operations that we consider cannot alter two pieces at the same time. The algorithm simply scans the text using a multipattern exact search algorithm for all the pieces. Each time a piece is found, it uses dynamic programming over an area of length $m+2k$ where the approximate occurrence can be found.

The multipattern search can be carried out in $O(n)$ worst-case search time by using an Aho-Corasick machine [1], or in $O(n/m)$ best-case time using Commentz-Walter [15] or another Boyer-Moore type algorithm adapted to multipattern search. The total cost of verifications keeps negligible if $k/m \leq 1/(3\log_\sigma m)$. We call *sublinear time* those algorithms that do not inspect all the text characters.

On the other hand, approximate multipattern search has only recently been considered. In [25], hashing is used to search thousands of patterns in parallel, although with only one error. In [8], extensions of [10] and [11] are presented based on superimposing automata. In [26], a counting filter is bit-parallelized to keep the state of many searches in parallel. Most multipattern algorithms are filters able to check for a necessary condition on many patterns at the same time.

2.2 Two Dimensional Pattern Matching

Two dimensional exact string matching was first considered by Bird and Baker [14, 13], who obtain $O(n^2)$ worst-case time. Good average results are presented by Zhu and Takaoka in [35]. The first good average case result is due to Baeza-Yates and Régnier [12], who obtain $O(n^2/m)$ time on average and $O(n^2)$ in the worst case. This was improved by Karkkäinen and Ukkonen [22] who achieve $O(n^2 \log_\sigma(m)/m^2)$ average case time, which is optimal.

Two-dimensional approximate string matching usually considers only substitutions for rectangular patterns, which is much simpler than the general case with insertions and deletions (because in this case, 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 [5] achieving $O((k+\log\sigma)n^2)$ time but using $O(n^2)$ space. A similar algorithm is presented in Crochemore and Rytter [16]. Ranka and Heywood [31], 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 [22], with its analysis and space usage improved by Park [30]. The expected time is $O(n^2 k \log_\sigma(m)/m^2)$ for $k \leq m^2/(4 \log_\sigma m)$, using $O(m^2)$ space ($O(k)$ space on average). This time result is optimal for the expected case. They extend their results to d dimensions achieving time $O(dn^d k \log_\sigma(m)/m^d)$.

2.3 The KS Model

Krithivasan and Sitalakshmi (KS) [24] defined the edit distance in two dimensions as the sum of the edit distance of the corresponding row images. Using this model they search a subimage of size $m \times m$ into a large image of size $n \times n$, in $O(m^2 n^2)$ time using a generalization of the classical one-dimensional algorithm. Krithivasan [23] presents for the same model an $O(m(k+\log m)n^2)$ algorithm that uses $O(mn)$ space. Amir and Landau [5] give an $O(k^2 n^2)$ worst case time algorithm using $O(n^2)$ space (note that k can be larger than m, so this is not necessarily better than the previous algorithms). Amir and Farach [4] also considered non-rectangular patterns achieving $O(k(k + \sqrt{m \log m}\sqrt{k \log k})n^2)$ time.

The KS model can in principle be extended to more than two dimensions, but it is less interesting because it allows errors along one of the dimensions only. Amir and Landau [5] also study this case, obtaining a $O(n^d(k(k+d)))$ worst case algorithm. We do not consider the KS model for $d > 2$.

2.4 Related Problems

Other problems related to comparing images is searching allowing rotations [20, 19, 18] and scaling [3, 2] (i.e. the pattern appears in the image at a different size).

Another related problem is geometric matching, where we have to match a geometric figure or a set of points. In this case, the problem is in a continuous space rather than a discrete space and usually the Hausdorff measure [6] is used.

There are other approaches to matching images, which are very different to ours (which belongs to what is called combinatorial pattern matching). Among them we can mention techniques used in pattern matching related to artificial intelligence (for example image processing and neural networks [33]) and techniques used in databases (extracting features of the image like color histograms [17]).

3 Fast Searching under the KS Model

We present now a fast filter to search a pattern allowing errors under the KS model. Although our algorithm can be used even for patterns and texts where each row has a different length, for simplicity we assume that the pattern P and the text T are rectangular, of sizes $m_1 \times m_2$ and $n_1 \times n_2$ respectively (rows $\times$ columns). We use also $M = m_1 m_2$ and $N = n_1 n_2$ as the size of the pattern and the text, respectively. Sometimes (especially for the analyses) we simplify and consider $m_1 = m_2 = m$ 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

$$\sum_{r=1}^{m_1} led(T_{i+r-1,1..j}, P_{r,1..m2}) \quad \leq \quad k$$

where $led(t_{1..j}, p) = \min_{i \in 1..j} ed(t_{i..j}, p)$ for one-dimensional strings t and p.

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 [24]. Its complexity is $O(MN)$, which is also a natural extension of the $O(mn)$ complexity for one-dimensional text. This algorithm uses $O(M)$ extra space, which is the only state information it needs to be started at any text position.

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

$$\min_{t=1..s} \{led(T_{i+r_t-1,1..j}, P_{r_t,1..m_2})\} \quad \leq \quad \lfloor k/s \rfloor \ .$$

Proof: Otherwise, $led(T_{i+r_t-1,1..j}, P_{r_t,1..m_2}) \geq 1 + \lfloor k/s \rfloor > k/s$ 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 text 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

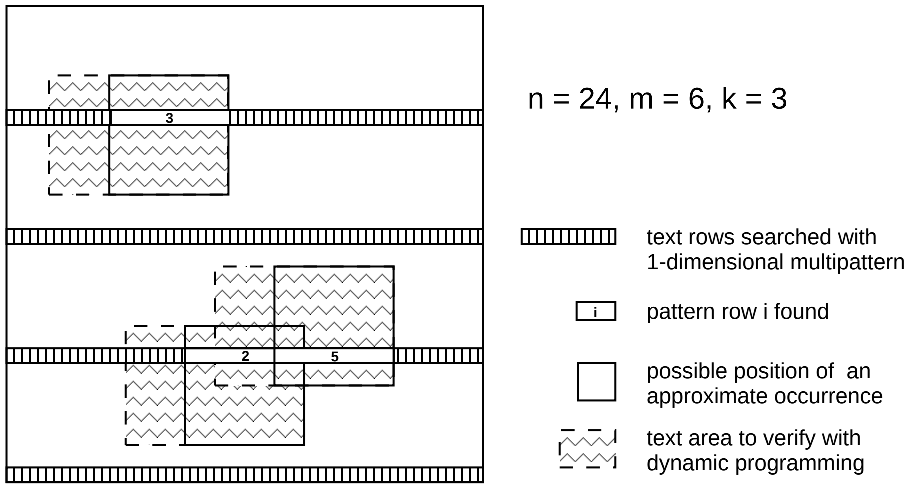
n = 24, m = 6, k = 3
3
2
5
i
text rows searched with 1-dimensional multipattern
pattern row i found
possible position of an approximate occurrence
text area to verify with dynamic programming

FIG. 1. Example of how the algorithm works.

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 $O(m_1 m_2^2) = O(m^3)$, which is formed by m_1 applications of the one-dimensional algorithm in a segment of length m_2.

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 1 shows how the algorithm works.

We have still not explained how to perform a multipattern approximate search for all the rows of the pattern at text rows numbered $i \cdot m_1$. We can use any available one-dimensional multipattern 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 each. 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 (the motivation for this is that 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 multipattern 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. $C(r) = sC(r/s)$).

3.1 Average Case Analysis

Once we have selected a given one-dimensional approximate 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 value for k/m from where the one-dimensional algorithm does not work anymore. 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 multipattern 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

$$\frac{\lfloor k/s \rfloor}{m_2} < L(m_2, m_1) \quad \Longrightarrow \quad s = 1 + \left\lfloor \frac{k}{m_2 L(m_2, m_1)} \right\rfloor \ . \tag{3.1}$$

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

$$O(n_1 s/m_1 \cdot n_2 C(m_2, k/s, m_1)) \;=\; O\left(\frac{Nk}{M} \, \frac{C(m_2, m_2 L(m_2, m_1), m_1)}{L(m_2, m_1)} \right) \ , \tag{3.2}$$

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

$$k < m_2(m_1 + 1) L(m_2, m_1) \ , \tag{3.3}$$

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 multipattern 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 [8]** 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. (3.1))

$$s \;=\; 1 + \left\lfloor \frac{k \log_\sigma(m_1^2 m_2^3)}{m_2} \right\rfloor \;=\; 1 + \left\lfloor \frac{5k \log_\sigma m}{m} \right\rfloor ,$$

  our average search cost would be (Eq. (3.2))

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

  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. (3.3)).

- **Superimposed Automata [8]** 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. (3.1))

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

  the average complexity is (Eq. (3.2))

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

  and the algorithm is applicable for $k < m_2(m_1+1)(1 - e/\sqrt{\sigma}) = m(m+1)(1 - e/\sqrt{\sigma})$ (Eq. (3.3)).

- **Counting [26]** 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. (3.1))

$$s \;=\; 1 + \left\lfloor \frac{k e^{m_2/\sigma}}{m_2} \right\rfloor \;=\; 1 + \left\lfloor \frac{k e^{m/\sigma}}{m} \right\rfloor ,$$

  the average search cost would be (Eq. (3.2))

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

  and the algorithm would be applicable for $k < m_2(m_1+1)e^{-m_2/\sigma} = m(m+1)e^{-m/\sigma}$ (Eq. (3.3)).

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 [25]** 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.

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 [10] that if $k/m_2 \geq 1 - e/\sqrt{\sigma}$ the number of text positions matching the pattern is high (observe that m_2 is the length of the strings that are searched, i.e. the width of the pattern). 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 2, where the best algorithm for each case is presented.

4 New Models

We present in this section new models for similarity in two dimensions. We also show how to compute the resulting distances and give basic search algorithms for them.

First, some notation used in this section. We consider two rectangular strings A and B of sizes $m_1 \times m_2$ and $n_1 \times n_2$, respectively. For the search problem, we replace A by P and B by T. Given a two-dimensional string S, we denote by $LS_{i,j}(S)$ the L-shaped string consisting of the first (left) j elements of the i-th row and the first (top) $i-1$ elements of the j-th column. This is related to the L-shape idea of Giancarlo [21] used for extending suffix trees to two dimensions.

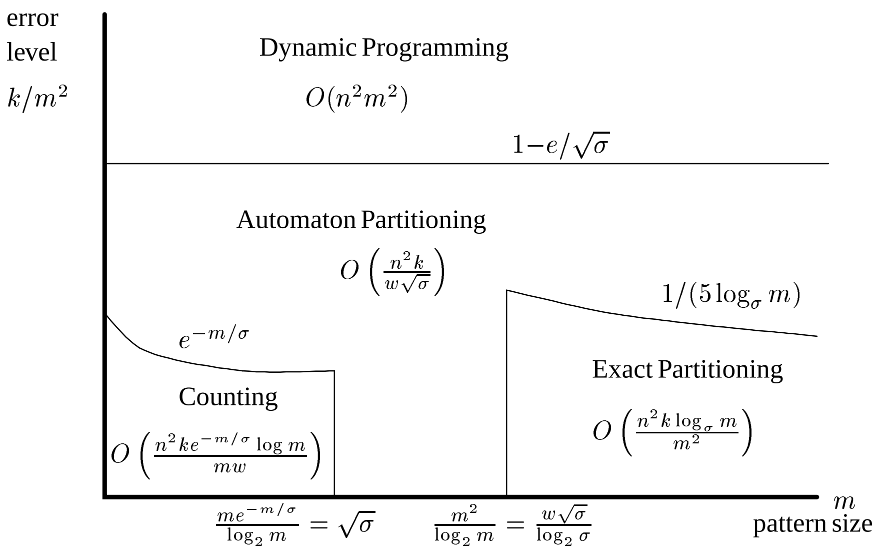

FIG. 2. The best algorithm with respect to the pattern length and error level. The complexity of each algorithm is also included.

4.1 Extending the Edit Distance

We start by solving the limitation of the KS model to handle deletions or insertions of whole rows. We introduce now the R model, where each row is treated as a single string which is compared to other rows using the one-dimensional edit distance, but whole rows can be inserted and deleted as well. We call $R_{i,j} = R(A_{1..i,1..m_2}, B_{1..j,1..n_2})$. Hence

$$R_{i,j} = \min(R_{i-1,j} + m_2, R_{i,j-1} + n_2, R_{i-1,j-1} + ed(A_{i,1..m_2}, B_{j,1..n_2}))$$

where the boundary conditions are $R_{i,0} = i \cdot m_2$ and $R_{0,j} = j \cdot n_2$, and the distance between the two images is given by $R(a,b) = R_{m_1,n_1}$.

In the example given in the Introduction, the distance is reduced to at most $2m$ instead of being $O(m^2)$ as in the KS model. Similarly, we could use columns instead of rows, obtaining another distance $C(a,b)$. This model is much more fair than the KS model. Although we use rectangular images, this measure can be extended to images where rows are connected and continuous, and that have different sizes.

Generalizing this idea to insertions and deletions at the same time in rows and/or columns is not as simple. Suppose that we have two subimages that we want to compare. One alternative is to decompose the border of a subimage in rows or columns. Then we can use the following decompositions:

1. removing one row or one column from one of the subimages or

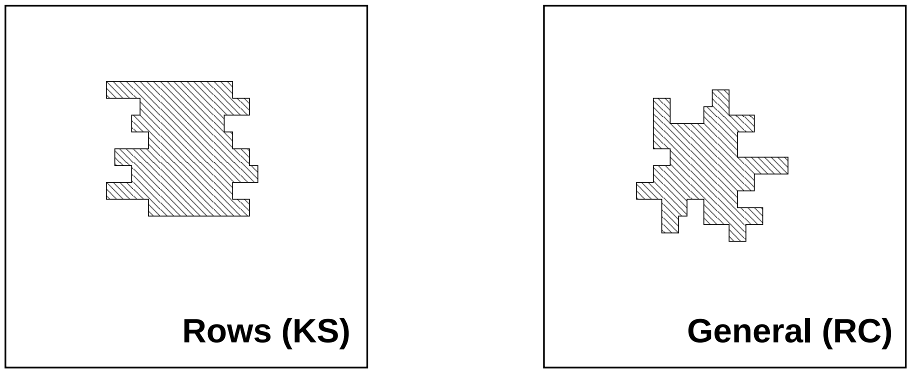

FIG. 3. KS and RC error models.

2. removing one row or one column in the same side of each subimage and computing the edit distance between them.

We can apply dynamic programming to find the best possible decomposition, and call RC the resulting model. Figure 3 shows the difference between the KS and the RC model. That is, if $RC_{i,j,p,q} = RC(A_{1..i,1..j}, B_{1..p,1..q})$, then we have that $RC_{i,j,p,q}$ is the minimum of the following values:

- $RC_{i-1,j,p,q} + j$, $RC_{i,j-1,p,q} + i$, $RC_{i,j,p-1,q} + q$, and $RC_{i,j,p,q-1} + p$, which corresponds to deleting one row or column in one sub-image (the cost is the size of the row or column removed); and
- $RC_{i-1,j,p-1,q} + ed(A_{i,1..j}, B_{p,1..q})$ and $RC_{i,j-1,p,q-1} + ed(A_{1..i,j}, B_{1..p,q})$, which corresponds to replacing one row or column of a subimage by that of the other. The one-dimensional edit distance $ed()$ gives the optimal way to do the change.

The boundary conditions are $RC_{0,0,i,j} = RC_{i,j,0,0} = i \cdot j$. The distance $RC(A, B)$ is given by RC_{m_1,m_2,n_1,n_2}. Figure 4 shows all these cases. This distance can also be applied to any convex image, for example circles or other regular polygons.

Nevertheless, this distance does not handle cases where we want to change at the same time a row and a column (for example, a motivation could be scaling). For that we use the L-shape mentioned earlier. So, we can also decompose the border of a subimage using L-shapes and we can have the same extensions as for rows or columns. To compare two L-shapes we see them as two one-dimensional strings. Then we have the following cases to find the minimal decomposed distance:

- $L_{i-1,j-1,p,q} + i + j - 1$ and $L_{i,j,p-1,q-1} + p + q - 1$ which corresponds to removing an L-shape in a subimage; and
- $L_{i-1,j-1,p-1,q-1} + ed(LS_{i,j}(A), LS_{p,q}(B)))$ which corresponds to comparing two L-shapes.

The boundary conditions are the same as the RC measure and the final distance is similarly given by $L(A, B) = L_{m_1,m_2,n_1,n_2}$. Figure 4 shows the decompositions

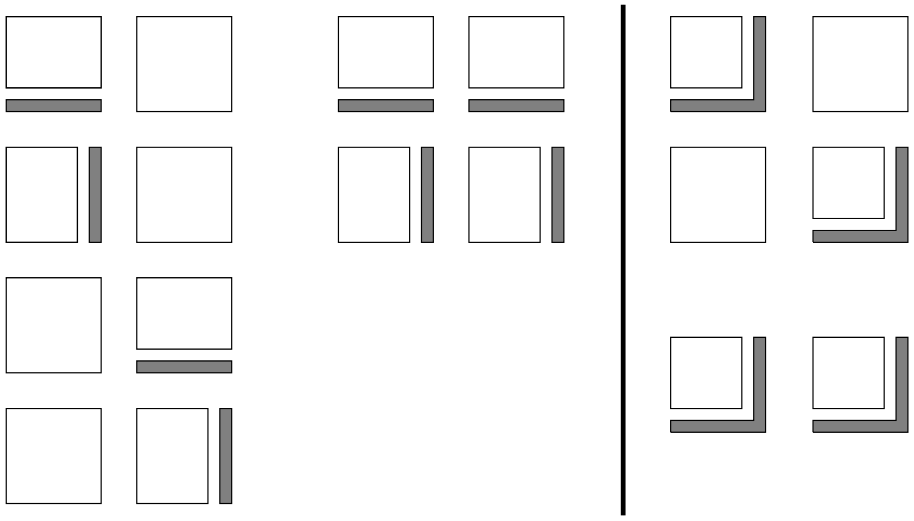
FIG. 4. Decomposition used in RC (left, 6 cases) and L (right, 3 cases).

associated to L. We will see later that this definition can be simplified by using the fact that one row and one column are considered at the same time when using L-shapes.

Finally, we can have a general distance $All(A, B)$ that uses both decompositions at the same time (RC and L) computing the minimal value of all possible cases. It is easy to show that $KS(A, B) \geq R(A, B) \geq RC(A, B) \geq All(A, B)$ and that $L(A, B) \geq All(A, B)$ because each case is a subset of the next. On the other hand, there are cases where $RC(A, B)$ will be less than $L(A, B)$ and vice versa. In fact, in Figure 5 this is shown together with other examples, where each color is a different symbol. The last example shows that combining RC and L can actually lead to a distance less than each separate case.

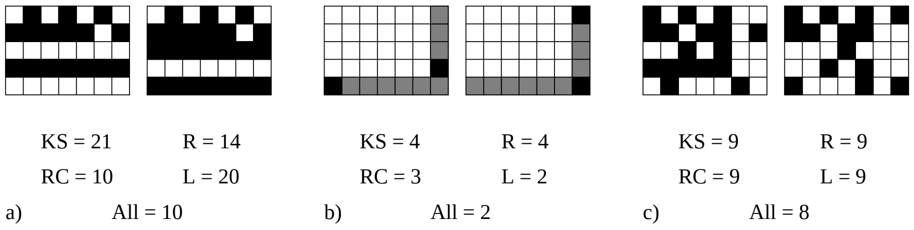

FIG. 5. Three examples for our new measures.

4.2 Computing the Distances

The definition of the above distances yields directly the algorithms to compute them: R and C can be computed in time $O(m_1n_1m_2n_2)$, RC in time $O(m_1n_1m_2n_2(m_1n_1+m_2n_2))$ and L in time $O(m_1m_2n_1n_2(m1+m2)(n1+n2))$. We simplify the exposition by considering that A and B are of size $m \times m$, in which case R and C cost $O(m^4)$ time $O(m^2)$ space, while RC and L cost $O(m^6)$ time and $O(m^4)$ space. This is prohibitive even for small images.

We show now how to do this better. First, the space usage is easily reduced to $O(m)$ for R and C and $O(m^3)$ to RC and L by noticing that we only need to store the boundary of the matrices of the dynamic programming computation as they are computed incrementally.

The computation of L can be simplified further by noticing that to compute the best decomposition $i-j$ and $p-q$ are always constant (in fact, equal to n_1-n_2 or m_1-m_2, which for squares images is 0). This is easily checked by observing the recurrence formulas for L. This means that only a quadratic number of entries must be computed, which implies a matrix boundary of size $O(m)$ and a running time of $O(m^4)$. However, this property does not hold if we use the All distance, since different $i-j$ and $p-q$ values may appear as the recurrence for L is mixed with others. Therefore, computing the All distance keeps $O(m^6)$ time.

Finally, we can also improve the computation of RC by precomputing all the edit distance matrices between pairs of rows and pairs of columns. That is,

$$Horiz_{i,j,p,q} = ed(A_{i,1..j}, B_{p,1..q}) \qquad Vert_{i,j,p,q} = ed(A_{1..i,j}, B_{1..p,q})$$

Note that $Horiz(i,*,p,*)$ is precisely the dynamic programming matrix used to compute the one-dimensional edit distance between $A_{i,1..m_2}$ and $B_{p,1..n_2}$; and the same happens to $Vert$. Hence, this preprocessing consists of computing the edit distances between all pairs of rows and all pairs of columns, and storing all the intermediate results. This takes $O(m^4)$ time and space.

Once this is precomputed, the one-dimensional edit distances in the RC formula can be obtained in constant time. The time to solve the recurrence drops to $O(m^4)$. Hence, RC can be computed in $O(m^4)$ time and space.

The idea of storing the boundary of the matrix can be applied to $Horiz$ and $Vert$ as well, reducing the space to $O(m^3)$. A very concrete way to see this is as follows: we select, say, i as the most external variable of the iteration to fill the matrices. Therefore, we need only the values at iteration $i-1$ to compute the values at iteration i. Hence, we do not need to store all the cells of all the i-th iterations, just the last one.

$Horiz$ and $Vert$ are therefore not precomputed completely but in coordination with the RC dynamic programming computation. For example, if we use i as the most external variable, we move from $i-1$ to i by computing and storing

$$Horiz'_{j,p,q} \;=\; ed(A_{i,1..j}, B_{p,1..q}) \;, \quad Vert'_{j,p,q} \;=\; ed(A_{1..i,j}, B_{1..p,q}) \;.$$

and we can see that $Horiz'$ does not need the previous value of $ed(A_{i-1,1..j}, B_{p,1..q})$, while $Vert'$ uses $ed(A_{1..i-1,j}, B_{1..p,q})$ (i.e. its old values) to obtain its new values.

Measure	Edit Distance		Searching	
	Time	Space	Time	Space
KS	m^3	m	m^3n^2	m
R, C	m^4	m	m^3n^2	$k+m$
L	m^4	m	m^4n^2	m
RC	m^4	m^3	m^4n^2	m^3
All	m^6	m^3	m^6n^2	m^3

TABLE 1: Time and space complexity for computing different distances and searching in two dimensions.

These optimizations allow us to handle patterns of reasonable size (say up to 50×50). Table 1 summarizes the space and time complexity obtained for all the measures, including KS. We can see that our measures need only one order of magnitude more time with respect to KS, using the same space, except for RC and All.

5 Searching Algorithms for the New Models

We consider now the problem of searching a pattern of size $m_1 \times m_2$ in a text of size $n_1 \times n_2$ (sometimes simplified to $m = m_1 = m_2$ and $n = n_1 = n_2$). To define the search problem using the new measures, we need to specify what is a match. Our working definition is that a match is a submatrix of the text of the form $T_{i..i+m_1-1,j..j+m_2-1}$. That is, there is a subrectangle in the text of the same shape of P whose edit distance to P is at most k.

One would also like to consider different definitions, for example allowing the pattern to match a subtext of different shape. For example, if P appears in T with one row inserted, our definition considers that P appears with $2m_2$ errors, while one could argue that the pattern matches a subtext of size $(m_1 + 1) \times m_2$ with only m_2 errors. We do not use this definition because we have not devised a good search algorithm for it. For instance, trying to extend the edit distances in the straightforward way [28] leads to an asymmetric search problem, where for example in the RC distance the matches can extend with arbitrary shape in their upper and left borders but have to finish sharply at the bottom and right borders.

The straightforward technique to search a pattern P in a text T is by considering all the $O(n^2)$ text positions as the possible origins of a match and applying the edit distance algorithm to the corresponding text rectangle. This simply multiplies by n^2 all the complexities given in Table 1 to compute the edit distance (even for KS). However (as shown in the same Table), this can be done better in some cases.

For R, we can precompute the one-dimensional distance between each pattern row and each horizontal text segment of length m_2. This takes $O(m_2^2 m_1 n_1 n_2) = O(m^3n^2)$ time. Once this is precomputed the R distance at each position can be solved in $O(m_1^2)$ time, so the total time keeps $O(m^3n^2)$. Of course not all the $O(mn^2)$ values have to be stored at any time. Just those relevant to the current pattern position in the text are kept, so that the new needed values are computed on the fly and those that are no more necessary are discarded. Hence, the extra space is only $O(k)$ because at each possible alignment each row can be displaced only in $\lfloor k/m_2 \rfloor$ rows, and therefore there are only $O(k/m)$ relevant text rows for each of the m pattern rows. The total space requirement is $O(m+k)$ because the computation of R needs $O(m)$ space anyway. The same can be done with the C distance.

The same technique can be applied to RC, but in this case the total time remains $O(m^4 n^2)$, because this is the cost of the recurrence even when the one-dimensional edit distances cost $O(1)$.

The next two subsections focus on fast average time filters for the new measures. They use the same filter used for the KS distance, with a few modifications.

5.1 A Filter for the R and C Distances

The filter used in Section 3 can also be applied to the R (and, rotating the problem, to the C) distance to obtain a fast algorithm on average. At most $\lfloor k/m_2 \rfloor$ insertions or deletions of rows can occur in a match with k errors. Therefore, if we check for all pattern rows in at least $s = 1 + \lfloor k/m_2 \lfloor$ rows of the text candidate area, we cannot miss a match. The s value used in Section 3 always satisfies this because $L(m_2, m_1) \leq 1$ in Eq. (3.1).

We only change the verification phase (each potential area found by the filter must be verified using the $O(m^4)$ worst case edit distance algorithm described in Section 4.2) by using R (or C) to compute the distance in a fixed area. The performance depends on which multipattern search algorithm we use. The best asymptotic performance is given by Exact Partitioning, which yields $O(n^2 k \log_\sigma(m)/m)$ search time for $k < m(m+1)/(6\log_\sigma m)$. This expected time is optimal [22].

5.2 A Filter for the RC Distance

The techniques presented in Section 3 can be adapted, in the two-dimensional case, to the RC distance function. In the KS definition, a single error could alter only a single row, while in the RC distance it can add one error to every row. However, there cannot be more than k errors in any row.

We can therefore use a multipattern search for all the pattern rows at the text rows of the form $i \cdot m$. Each time a row is found with k errors, the whole pattern is verified in a neighborhood of the occurrence. This corresponds to the case $s = 1$ in the Lemma of Section 3. However, this time we cannot use a larger s to reduce the number of errors. The reason is that in that case the total number of errors among all rows was limited by k, while now we can have k errors in each row. Hence, we are limited to the case $k < m$ for this filter. On the other hand, our verification cost is $O(m^4)$ instead of $O(m^3)$ as in KS.

The analysis of the multipattern search algorithms applied to this case is easy to derive. Using the same $C(m,k,r)$ and $L(m,r)$ terminology of Section 3, we have that the complexity of this algorithm is $n^2/m\ C(m,k,m)$ and it is applicable for

$k/m \leq L(m,m)$. Albeit we can use again many different search algorithms, we only show Exact Partitioning, which has the best asymptotic complexity. This multipattern algorithm has $C(m,k,m) = O(1)$ and, for our $O(m^4)$ verification costs, $L(m,m) = 1/(6\log_\sigma(m))$.

The resulting algorithm is thus $O(n^2/m)$ time overall. This is faster than the filters developed in the next section for an arbitrary number of dimensions (which for $d = 2$ yield $O(kn^2/m)$). However, it is less tolerant to errors, since it can be used for $k < m/(6\log_\sigma m)$ instead of $k < m/(2\log_\sigma m)$, which is the limit of the multidimensional filter for $d = 2$.

6 Extending the RC Model to More Dimensions

We concentrate now on RC, which can be nicely extended to the multidimensional case. We consider that A and B are d dimensional matrices of m^d cells. We call from now on $ed_d()$ the RC edit distance generalized to d dimensions. We show that it can be computed in $O(d!m^{2d})$ time and $O(m^{2d-1})$ space.

A $(2d)$-dimensional matrix RC is computed (d dimensions for A and d dimensions for B), and the $ed()$ of the two-dimensional formula (Section 4.1) is replaced by ed_{d-1}. If the values of ed_{d-1} are not precomputed then we have $O(m^{2d-1})$ space (by using the trick of selecting one variable as the most external in the iteration) plus the space needed to compute ed_{d-1} (only one at a time is computed). This gives the recurrence

$$S_1 \;=\; m\;, \qquad S_d \;=\; m^{2d-1} + S_{d-1}$$

which yields $O(m^{2d-1})$ space. The time, on the other hand, involves to fill m^{2d} cells, where each cell performs a minimum over $3d$ elements (i.e. insertion, deletion and ed_{d-1} in d dimensions). This makes it necessary to compute d times the function $ed_{d-1}()$. That is

$$T_1 \;=\; m^2\;, \qquad T_d \;=\; m^{2d}\,3d \;+\; m^{2d}\,d\,T_{d-1}$$

which yields $O(d!m^{d(d+1)})$. This matches the $O(m^6)$ result for two dimensions of Section 4.2.

However, as before, this can be done better. We may precompute all the necessary values of $ed_{d-1}()$. Along each one of the d dimensions, we take all the m^2 (i,p) possible combinations of values of the selected dimension in A and B, and compute $ed_{d-1}()$ between the $(d-1)$-dimensional objects which result from restricting the selected dimension to i in A and to p in B. Once this is done, the ed_{d-1} computations can be taken as constants in the formula of $ed_d()$. The time cost is now

$$T_1 \;=\; m^2\;, \qquad T_d \;=\; m^{2d}\,3d \;+\; dm^2T_{d-1}$$

which yields $O(d!m^{2d})$ time (this matches the improved $O(m^4)$ for two dimensions). This is a big improvement over the naive algorithm. The space requirements are, however, higher. We have to store, for the d-dimensional object, m^{2d} cells plus the precomputed values, along each dimension, of all the m^2 combinations of (i,p) values for that dimension, and all the space for the lower dimensions resulting for each pair (i, p). That is

$$S_1 \ = \ m \ , \quad S_d = m^{2d} \ + \ dm^2 S_{d-1}$$

which yields

$$S_d \ = \ d!m^{2d} \left(\frac{1}{1!} + \frac{1}{2!} + ... + \frac{1}{d!} \right) \ \leq \ d!m^{2d}e \ = \ O(d!m^{2d})$$

and we can use the trick of the external variable to reduce this to $O(d!m^{2d-1})$.

The time to search a pattern of size m^d in a text of size n^d is therefore $O(d!m^{2d}n^d)$. Our aim is to reduce this time. We first show a better worst-case algorithm and later develop a filter that is fast on average. For this filter, we need first to return to the simpler problem of multidimensional exact searching.

6.1 *Faster Computation of Limited RC Distances*

We saw that computing the edit distance requires $O(m^{2d})$ time using the previous algorithm. We show now that a restricted version of the problem can be solved better: given k, we want to determine which is the edit distance provided it is at most k, otherwise we just want to know that it is larger than k. We show how to do this in $O(dk^2m)$ time.

This serves two purposes: first, we can use it to solve the general problem: if D is the edit distance between two d-dimensional cubes of size m^d, then it can be computed in $O(dD^2m)$ time. This is better than the naive algorithm if the cubes are quite similar, i.e. if $D = o(m^{d-1/2})$. This is done by using the restricted algorithm for $k = 0$, 1, 2, 4, 8... until the exact distance is reached or surpassed by k. The complexity of the sum of runs is that of the last run, where $\lfloor k/2 \rfloor < D \leq k$ and therefore $k = \Theta(D)$.

A second use of the algorithm is for searching: we compare all the text positions but are only interested in the limited problem, hence obtaining a search algorithm of $O(dk^2mn^2)$ time. This is better than the naive algorithm if $k = o(m^{d-1/2})$.

So we concentrate on the algorithm for restricted k. The key idea is that, if we are limited in the maximum edit distance we want to find, we do not need to compare all the $(d-1)$ dimensional objects, since some are too far away to compensate the difference in sizes with k insertions. We give a complete argument for two dimensions and then show a simpler generalization to d dimensions.

6.1.1 Two Dimensions

We consider which subrectangles $A_{1..i,1..j}$ and $B_{1..p,1..q}$ are so different that they need not be compared because k insertions are not sufficient to compensate for the difference in sizes. We consider fixed i and p so that $i < p$, and study the relevant j and q. The case $i > p$ is symmetric. For fixed $i < p$ there are two different cases: $j < q$ and $j \geq q$ (see Figure 6).

In the first case, the subrectangle of A is totally contained in that of B, and the difference in areas is $pq - ij$, which must satisfy $pq - ij \leq k$. For fixed i and p, this

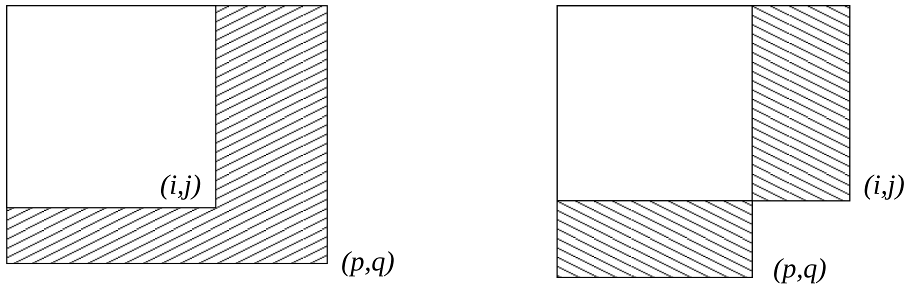

FIG. 6. The two possible relations between squares for $i < p$.

defines a straight line in the (q, j) coordinate space, whose extremes are $(k/p, 0)$ and $(k/(p-i), k/(p-i))$. In the second case, we have that the rectangles overlap and the difference in areas is $(p-i)q + (j-q)i$, which again cannot be larger than k. This defines another straight line with extremes $(k/(p-i), k/(p-i))$ and $(0, k/i)$. Figure 7 illustrates.

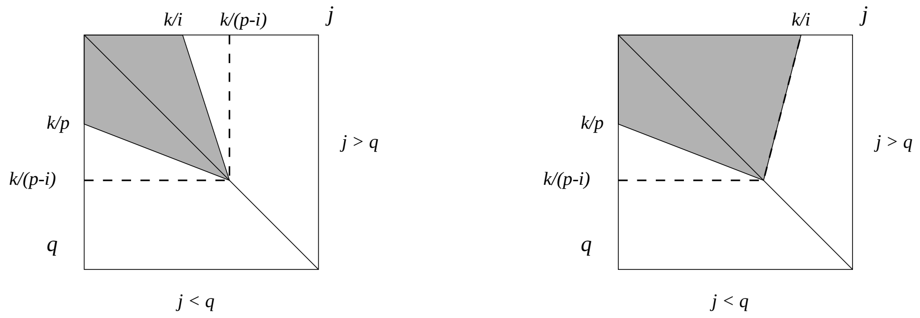

FIG. 7: The areas in the (q, j) coordinate space that must be computed have been shadowed, parameterized with p and i. They are different according to whether $p > 2i$ or not. The dashed lines show other areas that must be also computed to obtain the desired values.

To fill the required cells, two processes are necessary: (i) precompute the row-wise and column-wise distances, (ii) run the dynamic programming algorithm over the shadowed areas. The second process is proportional to the shadowed areas (summed over all (i, p)). In the first process we must compute all the distances between prefixes of rows and columns that lie in the shadowed area. However, this is not $O(1)$ per shadowed cell.

To compute the edit distance between row prefixes $A_{i,1..j}$ and $B_{p,1..q}$ we need to compute also the distance between all the prefixes of those prefixes. This takes constant time per prefix computed (hence the total $O(qj)$ quadratic cost). However, the shadowed area in Figure 7 is not prefix-closed. We have added dashed lines enclosing the square that must be computed to obtain the shadowed cells. Hence the total cost of the preprocessing is proportional to this extended set of cells, and therefore dominates the total processing time.

To measure the area enclosed in dashed lines we must separate the cases shown in Figure 7, which depend on whether $i < p \leq 2i$ or $p > 2i$. In the first case the total area is $(k/(p-i))^2$ and in the second $pk^2/(2i(p-i)^2)$. Summing all the relevant areas over $1 \leq i < p \leq m$ yields $O(k^2 m)$.

6.1.2 More than Two Dimensions

We can generalize the above scheme to $d > 2$ dimensions. In general, if we have two hypercubes, we need to compare them only if they are reasonably close in size. Considering again that the preprocessing that compares all the m^2 different $(d-1)$-dimensional objects along each dimension is the most expensive part, we see that two objects of "volume" V that are at positions i and p need only be compared if $V(p-i) \leq k$, since $V(p-i)$ is the minimum number of insertions necessary to make them comparable (their volume could be different, say V_1 and V_2, but the number of insertions is $\max(V_1, V_2)(p-i)$ and therefore the limit is reached for similar volumes). Comparing those objects of volume V, together with all their prefixes (objects of smaller volume) takes $O(V^2)$, which is limited by $(k/(p-i))^2)$. Summing over all (i, p) yields $O(k^2 m)$, and summing this over each dimension gives $O(dk^2 m)$.

7 Multidimensional Searching Algorithms

We first present some new results on exact multidimensional pattern matching which we later use for fast filter algorithms for multidimensional approximate pattern matching.

7.1 Exact Multidimensional Pattern Matching

In [12], they allow searching, in two dimensions, a pattern in a text in $O(n^2/m)$ average time. They traverse only the text rows of the form $i \times m$ searching for all the pattern rows at the same time (using Aho-Corasick [1]), and verify all potential matches. Clearly, no match can be missed with the filter.

In [12], the authors briefly mention that their technique can be extended to more dimensions by selecting one dimension and recursively using an algorithm for $(d-1)$ dimensions on the m-th "rows" of such text. However no more details are given, nor any analysis.

We give now a more detailed version of the algorithm and analyze it. We select one dimension (say, coordinate 1) and obtain n/m different $(d-1)$ dimensional objects of the form $T_{m,1..n,1..n,...}$, $T_{2m,1..n,1..n,...}$, ..., $T_{im,1..n,1..n,...}$, and so on. On the other hand, we obtain m patterns of $(d-1)$ dimensions, namely $P_{1,1..m,1..m,...}$, $P_{2,1..m,1..m,...}$, ..., $P_{p,1..m,1..m,...}$ and so on. All the m subpatterns are searched in each one of the $(d-1)$ dimensional subtexts. See Figure 8. Each time one of the $(d-1)$ dimensional subpatterns is found in a text position, the complete d-dimensional pattern

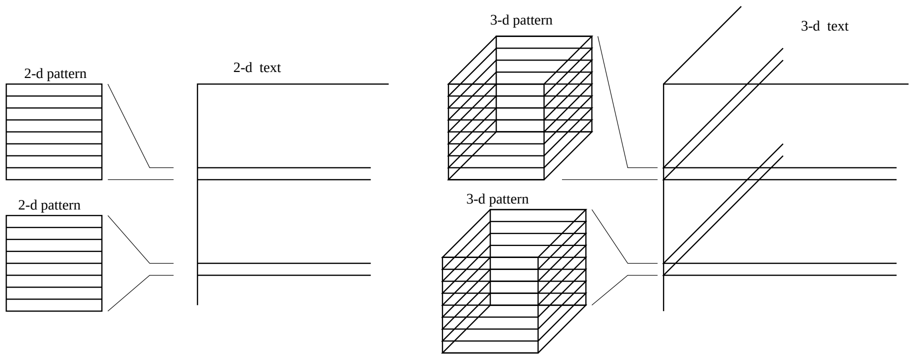

FIG. 8: Algorithm for exact searching. All the pattern "rows" are searched in n/m text "rows" at the same time.

is checked.

An important part of the analysis of [12] for two dimensions is that the total cost to verify potential matches is not too large. It is not immediate that this is still valid for more dimensions, since a very large number of verifications are finally triggered.

The cost to verify a potential match in d dimensions is always $O(1)$ on average, since we have to check if m^d letters of the pattern are equal to the text at a given position. Since we stop the checking as soon as we find a mismatch, we verify more than c characters with probability $1/\sigma^c$. Hence, the average number of characters checked is $\sum_c 1/\sigma^c = O(1)$ (even for patterns of unbounded size).

We denote by $E_{d,r}$ the average search cost for r patterns in d dimensions. The existence of the Aho-Corasick [1] algorithm implies that $E_{1,r} = n$. Now, for d dimensions, we perform n/m searches for rm patterns on $d-1$ dimensions, and check all the candidates that occur. The probability of a pattern of size m^{d-1} occurring in a text position is $1/\sigma^{m^{d-1}}$, but we multiply that by rm because we search for rm different patterns. As the average cost to verify each potential match is $O(1)$, and the $(d-1)$ dimensional texts are of size n^{d-1}, we have that

$$E_{d,r} \;=\; \frac{n}{m}\left(E_{d-1,rm} + n^{d-1}\frac{rm}{\sigma^{m^{d-1}}}\right) \;=\; \frac{n}{m}E_{d-1,rm} + \frac{n^d r}{\sigma^{m^{d-1}}}$$

which gives

$$E_{d,r} \;=\; \frac{n^d}{m^{d-1}} \;+\; \sum_{w=1}^{d-1}\frac{n^d r}{\sigma^{m^w}} \;=\; O\left(n^d\left(\frac{1}{m^{d-1}} + \frac{r}{\sigma^m}\right)\right)$$

(where the first term corresponds to the actual searches which are all done in one dimension).

To search for one pattern we replace r by 1 in this final formula (although the algorithm internally uses multipattern search). This formula matches the result for two dimensions, since $1/\sigma^m = o(1/m)$. In general, if d is considered fixed, the

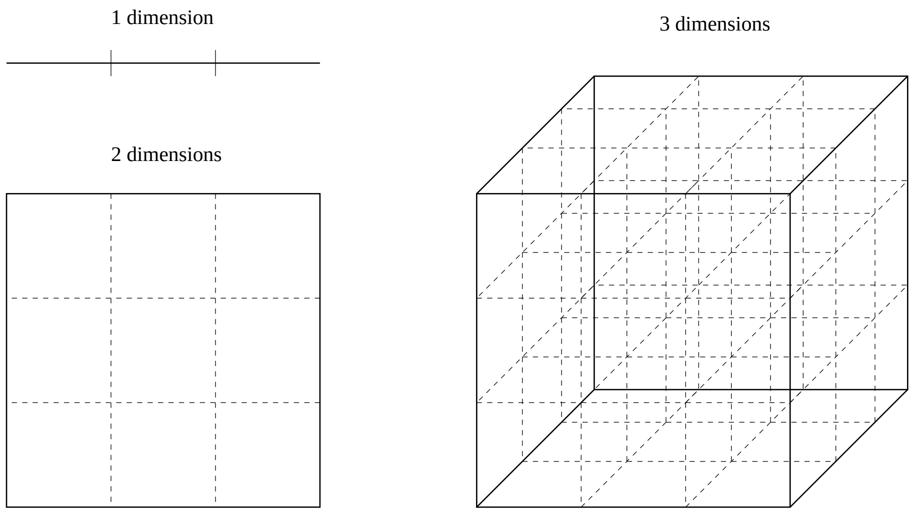

FIG. 9: Filtering algorithm for $j = 3$. The maximum possible k so that some block appears unchanged is 2, 2, and 8 as the dimension grows.

above result for $r = 1$ can be bounded by $O(n^d/m^{d-1})$. The worst case search cost corresponds to verifying all text positions like a brute force search, i.e. $O(rm^d n^d)$.

The space complexity of the algorithm corresponds to the Aho-Corasick machine, whose space requirements are proportional to the total size of all the patterns, i.e. $O(rm^d)$. We use this algorithm as a building block in the next section.

7.2 A Fast Filter for Multidimensional Approximate Searching

We present now an effective filter to quickly discard large parts of the text which cannot contain a match, so that we use the dynamic programming algorithm to verify only the text areas which could contain an occurrence of the pattern.

The filter is based on a generalization of the one-dimensional filter explained in Section 2. In that case, we cut the pattern in $(k + 1)$ pieces, and since each error can destroy at most one piece, we have always one piece left untouched inside each occurrence.

In two and more dimensions, we cut the pattern in j pieces along each dimension, for some $1 \leq j \leq m$ (see Figure 9). Since each error occurs along one dimension only, at most kj pieces are destroyed. Therefore, since there are j^d pieces in total, it is enough that $j^d > kj$ to ensure that at least one of the pieces is left untouched (although we do not know which one). Hence, we search for all the j^d pieces at the same time in the text without allowing errors. Those pieces are of size $(m/j)^d$, and can be searched with the algorithm of the previous section in $O(m^d)$ space and an average time of

$$n^d \left(\frac{1}{(m/j)^{d-1}} + \frac{j^d}{\sigma^{m/j}} \right) = j^d n^d \left(\frac{1}{j m^{d-1}} + \frac{1}{\sigma^{m/j}} \right)$$

Each time one such piece is found, we have to verify a surrounding text area to check for a possible match. Hence, the cost of a verification is the same as that of comparing the pattern with a subtext of size m^d allowing errors, which is $O(d! m^{2d})$. The total number of verifications is obtained by multiplying the number of pattern pieces j^d by the probability of a piece matching, i.e. $1/\sigma^{(m/j)^d}$. Hence, the total expected cost for verifications is $j^d d! m^{2d} n^d / \sigma^{(m/j)^d}$.

The space requirement of this algorithm is $O(d! m^{2d-1})$ (this corresponds to the verification phase, since the search of the pieces needs much less, i.e. $O(m^d)$).

Both the search and the verification cost worsen as j grows, so we are interested in the minimum j that works. As said, we need that $j^d > kj$, hence

$$j = \left\lfloor k^{\frac{1}{d-1}} \right\rfloor + 1$$

is the best choice. The formula does not work for one dimension (because it is not true that kj pieces are destroyed), and for 2 dimensions it sets $j = k + 1$ as in the traditional one-dimensional case. Notice that we need that $j \leq m$, and therefore the mechanism works for $k < k_3 = m^{d-1}$. Using this optimum (and minimum) j, the total cost of searching plus verifying is

$$n^d k^{\frac{d}{d-1}} \left(\frac{1}{m^{d-1} k^{\frac{1}{d-1}}} + \frac{1}{\sigma^{m/k^{1/(d-1)}}} + \frac{d! m^{2d}}{\sigma^{m^d / k^{d/(d-1)}}} \right) \qquad (7.1)$$

which worsens as k grows. This search complexity has three terms, each of which dominates for a different range of k values. The first one dominates for

$$k \leq k_0 = \frac{m^{d-1}}{(d \log_\sigma m)^{d-1}} (1 + o(1))$$

while the second dominates from $k > k_0$ until

$$k \leq k_1 = \frac{m^{d-1}}{(d(\log_\sigma d + 2 \log_\sigma m))^{\frac{d-1}{d}}} (1 + o(1))$$

In the maximum acceptable value $k = m^{d-1} - 1$, the search complexity becomes $O(d! m^{3d} n^d)$, which is worse than using dynamic programming. We want to know which is the k value for which the filter is better than dynamic programming. This is

$$k \leq k_2 = \frac{m^{d-1}}{(2d \log_\sigma m)^{\frac{d-1}{d}}} (1 + o(1))$$

Finally, the most stringent condition we can ask to the filter is to be sublinear, i.e. faster than $O(n^d)$. If we try to consider the third term of the search complexity as dominant, we arrive to a k value which is smaller than k_1, which means that the solution is in a stricter k range. By considering the second term of the search complexity, we arrive to the condition $k \leq k_0$. That is, the search time is sublinear precisely when the first term of the summation dominates.

To summarize, the search algorithm is sublinear (i.e. $O(kn^d/m^{d-1})$) for $k < (m/(d\log_\sigma m))^{d-1}$. Otherwise, it is not sublinear, but it improves over dynamic programming for $k \leq m^{d-1}/(2d\log_\sigma m)^{(d-1)/d}$. Figure 10 illustrates the result of the analysis.

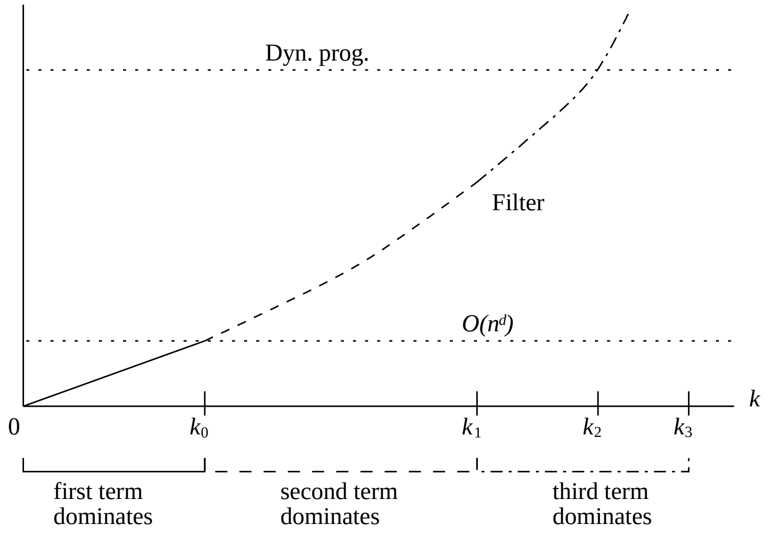

FIG. 10. The complexity of the proposed filter, depending on k.

7.3 *A Stricter Filter*

We have assumed up to now that we verify the presence of the pattern allowing errors as soon as any of the j^d pieces appears. However, we can do better. We know that $j^d - jk$ pieces must appear, at their correct positions, for a match to be possible. Therefore, whenever a piece appears, we can check the neighborhood for the exact occurrences of other pieces. On average, the verification of each piece will fail in $O(1)$ character comparisons, and we will check $O(jk)$ pieces until jk of them fail the test (this is because both are geometric processes). Therefore, we have a preverification test which occurs with probability $j^d/\sigma^{(m/j)^d}$, costs $O(jk)$ and is able to discard more text positions before actually verifying the candidate area. The probability that a text position passes the preverification test and undergoes the dynamic programming verification can be computed by considering that $j^d - jk$ cells need to match, which means that $m^d - km^d/j^{d-1}$ characters match. On the other hand, we can select as we want which jk cells match out of j^d.

The new search cost is therefore

$$n^d \left(\frac{j^{d-1}}{m^{d-1}} + \frac{j^d}{\sigma^{m/j}} + \frac{j^d jk}{\sigma^{(m/j)^d}} + \frac{\binom{j^d}{jk} d! m^{2d}}{\sigma^{m^d - km^d/j^{d-1}}} \right)$$

where the first term dominates for $j \leq m/(d \log_\sigma m)$, the second one up to $j \leq m/(\log_\sigma m + \log_\sigma k)^{1/d}$, and the third one for larger j. The fourth term decreases with j, and therefore it is not immediate that the minimum j is the optimum (in fact it is not). We have not been able to determine the optimum j, but we can still obtain the maximum k value up to where the filter is better than dynamic programming. The first two terms are never worse than dynamic programming, and the third improves over dynamic programming for

$$j \leq \frac{m}{(\log_\sigma m + \log_\sigma k - d \log_\sigma d)^{1/d}} \, (1 + o(1))$$

which gives a condition on k since $j^{d-1} > k$:

$$k \leq k_2' = \frac{m^{d-1}}{(d(\log_\sigma m - \log_\sigma d))^{\frac{d-1}{d}}} \, (1 + o(1))$$

Now, we introduce this maximum j value in the fourth term to determine whether it is also better than dynamic programming at that point. The result is that, using that j value, the fourth term is dominated by the third precisely for $k \leq k_2'$. Therefore we improve over dynamic programming for $k \leq k_2'$ (which is better than our previous k_2 limit). The proposed j is the best for high k values, but smaller values are better for lower k values. In particular, we may be interested in obtaining the sublinearity limit for this filter. The first three terms put an upper bound on j, the strictest one being

$$j \leq \frac{m}{d(\log_\sigma m - \log_\sigma d)} \, (1 + o(1))$$

and using this maximum j value the fourth term gives us the maximum k that allows sublinear search time:

$$k \leq k_0' = \frac{m^{d-1}}{(d(\log_\sigma m - \log_\sigma d))^{d-1}} \, (1 + o(1))$$

which is slightly better than our previous k_0 limit.

We could have used the algorithm of Kärkkäinen and Ukkonen [22] instead of that of Baeza-Yates and Régnier [12] considering that the former is faster on average. However, the former does not have the ability of searching many patterns simultaneously, which is the key issue in our case. In fact, using that algorithm, our filter is slower only in two cases:

i) $k < \left(\frac{m}{\frac{d^2}{d+1} \log_\sigma m} \right)^{\frac{d-1}{d+1}}$ which is a very small k; or

ii) $k \geq m^{d-1}$, which is too large and where the filters do not work anyway.

This resembles the difference between searching multiple patterns using a Boyer-Moore or an Aho-Corasick algorithm.

7.4 Adapting the Filter to Simpler Distances

Since the R and C distances are lower bounded by RC, the filter we have just designed for RC works for R and C as well, with the same complexities (albeit only the case $d = 2$ is interesting).

Another possible simplification is to use the filter to search a pattern allowing k substitutions. This problem is much simpler: a brute force search algorithm checks any possible text position until it finds k mismatches. Being a geometric process, this occurs after $O(k)$ character comparisons, which makes the total search cost $O(kn^d)$ on average.

Therefore, in this model the cost to verify a candidate text position is only $O(k)$. The search cost, as in Eq. (7.1), still has three terms:

$$n^d k^{\frac{d}{d-1}} \left(\frac{1}{m^{d-1} k^{\frac{1}{d-1}}} + \frac{1}{\sigma^{m/k^{1/(d-1)}}} + \frac{k}{\sigma^{m^d/k^{d/(d-1)}}} \right)$$

where the first term is dominant for $k \leq k_0$. The second term is now dominant for

$$k \;\leq\; k_1' \;=\; \frac{m^{d-1}}{(d \log_\sigma m)^{\frac{d-1}{d}}} \, (1 + o(1))$$

and the last one dominates for $k > k_1'$. This filter is sublinear (i.e. does not inspect all the text characters) on average for $k < k_0$ as before. On the other hand, it turns out to be better than brute force (i.e. $O(kn^d)$) for $k \leq k_1'$, i.e. before the verification step dominates the search cost. Overall, we achieve the $O(kn^d/m^{d-1})$ search time on average. However, Karkkäinen and Ukkonen algorithm's [22] for this case is faster, achieving $O(dn^d k \log_\sigma(m)/m^d)$ average time.

8 Concluding Remarks

We have focused on two and multidimensional approximate pattern matching. The contribution of this work is many fold. We have developed the first sublinear average time filters for the existing model on two dimensions. We have proposed new distances for two dimensions and have shown how to compute them and how to search a pattern in a text under those distances. The most promising of them, that we have called the RC distance, allows the errors to occur along rows and columns at any time. We have generalized the most promising of them to d dimensions and have presented a d-dimensional filtering algorithm that yields sublinear search time when the error level tolerated is low enough. For instance, in two dimensions the filter is sublinear time for $k < m/(2 \log_\sigma m)$ and better than a brute force search for $k \leq m/\sqrt{2 \log_\sigma m}$.

These are the first search algorithms and fast filters for the first model which extends successfully the concept of approximate string matching to more than one dimension. Although the algorithms have been presented for square d-dimensional pattern and text, they also work for hyper-rectangular elements and more complex shapes.

An open problem is how to design optimal worst-case time algorithms for approximate searching using the new measures, i.e. achieving $O(m^2 n^2)$ time complexity for the R, C, L, and RC measures. Another interesting problem is how to search efficiently using the L distance.

There are other open problems related to the models themselves. For example, we could try to define the *largest common image* of two images, which generalizes the concept of longest common subsequence of one-dimensional strings. Given two images, find a set of position pairs that match exactly in both images subject to the following restrictions:

1. The set of positions for the same pattern are disjoint;
2. a suitable order given by the position values is the same for both images (for example, image pixels can be sorted by their $i + j$ value, using the value of i in the case of ties); and
3. the total size of the set of positions is maximized.

For the edit distance, condition 3 has to be changed to:

3. Minimize the number of mismatches, insertions and deletions needed to obtain the set of matching positions.

Figure 11 gives an example. All pieces of the pattern not in the text corresponds to deletions and mismatches and should be counted. In the text, black regions are not counted, because they correspond to mismatches. All other pieces are insertions in the pattern. It is not clear that the minimal string editing solution gives the same answer as the largest common set of sub-images. Also, it could be argued that characters inserted/deleted on external borders should not be counted as errors.

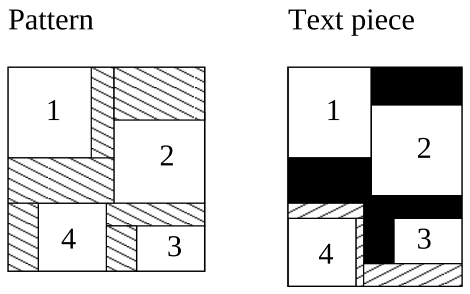

FIG. 11. Example of largest common image.

The approximate two-dimensional pattern matching problem can be stated as usual using the above definition as searching for all rectangular subimages of the text that have edit distance at most k with the pattern. An alternative definition would be to find all pieces of the text that have at least $m^2 - k$ matching positions with the pattern.

Our work is a (very preliminary) step towards presenting a combinatorial alternative to the current image processing technology. Other related approaches have focused on rotations [20] and scalings [3, 2] An open problem is how to combine those approaches to allow deformations in the occurrences.

Acknowledgements

We would like to thank the comments of Sven Schuerier regarding the new models (which in particular pointed out the simplification for computing the L measure).

References

[1] A. Aho and M. Corasick. Efficient string matching: an aid to bibliographic search. *CACM*, 18(6):333–340, June 1975.

[2] A. Amir, A. Butman, and M. Lewenstein. Real scaled matching. In *Proc. SODA 2000*, page To appear., San Francisco, January 2000.

[3] A. Amir and G. Calinescu. Alphabet independent and dictionary scaled matching. In *Proc. CPM'96*, number 1075 in LNCS, pages 320–334, 1996.

[4] A. Amir and M. Farach. Efficient 2-dimensional approximate matching of non-rectangular figures. In *Proc. SODA'91*, pages 212–223, 1991.

[5] A. Amir and G. Landau. Fast parallel and serial multidimensional approximate array matching. *Theoretical Computer Science*, 81:97–115, 1991.

[6] M. Atallah. A linear time algorithm for the Hausdorff distance between convex polygons. *Information Processing Letters*, 17:207–209, 1983.

[7] R. Baeza-Yates. Similarity in two-dimensional strings. In *Proc. COCOON'98*, number 1449 in LNCS, pages 319–328, Taipei, Taiwan, August 1998.

[8] R. Baeza-Yates and G. Navarro. Multiple approximate string matching. In *Proc. WADS'97*, LNCS 1272, pages 174–184, 1997.

[9] R. Baeza-Yates and G. Navarro. Fast two-dimensional approximate pattern matching. In *Proc. LATIN'98*, number 1380 in LNCS, pages 341–351. Springer-Verlag, 1998.

[10] R. Baeza-Yates and G. Navarro. Faster approximate string matching. *Algorithmica*, 23(2):127–158, 1999.

[11] R. Baeza-Yates and C. Perleberg. Fast and practical approximate pattern matching. In *Proc. CPM'92*, LNCS 644, pages 185–192, 1992.

[12] R. Baeza-Yates and M. Régnier. Fast two dimensional pattern matching. *Information Processing Letters*, 45:51–57, 1993.

[13] 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.

[14] R. Bird. Two dimensional pattern matching. *Inf. Proc. Letters*, 6:168–170, 1977.

[15] B. Commentz-Walter. A string matching algorithm fast on the average. In *Proc. ICALP'79*, number 6 in LNCS, pages 118–132. Springer-Verlag, 1979.

[16] M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, Oxford, UK, 1994.

[17] C. Faloutsos, R. Barber, M. Flickner, J. Hafner, W. Niblack, D. Petkovic, and W. Equitz. Efficient and effective querying by image content. *J. of Intelligent Information Systems*, 3:231–262, 1994.

[18] K. Fredriksson, G. Navarro, and E. Ukkonen. Fast filters for two dimensional string matching allowing rotations. Technical Report TR/DCC-99-9, Dept. of Computer Science, Univ. of Chile, November 1999.

[19] K. Fredriksson, G. Navarro, and E. Ukkonen. An index for two dimensional string matching allowing rotations. Technical Report TR/DCC-99-8, Dept. of Computer Science, Univ. of Chile, November 1999. Submitted to *IFIP TCS 2000*.

[20] K. Fredriksson and E. Ukkonen. A rotation invariant filter for two-dimensional string matching. In *Proc. CPM'98*, number 1448 in LNCS, pages 118–125, 1998.

[21] R. Giancarlo. A generalization of suffix trees to square matrices, with applications. *SIAM J. on Computing*, 24:520–562, 1995.

[22] J. Karkkäinen and E. Ukkonen. Two and higher dimensional pattern matching in optimal expected time. In *Proc. SODA'94*, pages 715–723. SIAM, 1994.

[23] K. Krithivasan. Efficient two-dimensional parallel and serial approximate pattern matching. Technical Report CAR-TR-259, University of Maryland, 1987.

[24] K. Krithivasan and R. Sitalakshmi. Efficient two-dimensional pattern matching in the presence of errors. *Information Sciences*, 43:169–184, 1987.

[25] R. Muth and U. Manber. Approximate multiple string search. In *Proc. CPM'96*, LNCS 1075, pages 75–86, 1996.

[26] G. Navarro. Multiple approximate string matching by counting. In *Proc. WSP'97*, pages 125–139, 1997.

[27] G. Navarro. A guided tour to approximate string matching. Technical Report TR/DCC-99-5, Dept. of Computer Science, Univ. of Chile, 1999. To appear in *ACM Computing Surveys*. `ftp://-ftp.dcc.uchile.cl/pub/users/gnavarro/survasm.ps.gz`.

[28] G. Navarro and R. Baeza-Yates. Fast multi-dimensional approximate string matching. In *Proc. CPM'99*, LNCS v. 1645, pages 243–257, 1999.

[29] S. Needleman and C. Wunsch. A general method applicable to the search for similarities in the amino acid sequences of two proteins. *J. of Molecular Biology*, 48:444–453, 1970.

[30] K. Park. Analysis of two dimensional approximate pattern matching algorithms. In *Proc. CPM'96*, LNCS 1075, pages 335–347, 1996.

[31] S. Ranka and T. Heywood. Two-dimensional pattern matching with k mismatches. *Pattern recognition*, 24(1):31–40, 1991.

[32] P. Sellers. The theory and computation of evolutionary distances: pattern recognition. *J. of Algorithms*, 1:359–373, 1980.

[33] P. Suetens, P. Fua, and A. Hanson. Computational strategies for object recognition. *ACM Computing Surveys*, 24:5–62, 1992.

[34] S. Wu and U. Manber. Fast text searching allowing errors. *CACM*, 35(10):83–91, October 1992.

[35] R. Zhu and T. Takaoka. A technique for two-dimensional pattern matching. *Comm. ACM*, 32(9):1110–1120, 1989.