Fast Multi-Dimensional Approximate Pattern Matching [*]

Gonzalo Navarro and Ricardo Baeza-Yates

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

Abstract. We address the problem of approximate string matching in d dimensions, that is, to find a pattern of size m^d in a text of size n^d with at most $k < m^d$ errors (substitutions, insertions and deletions along any dimension). We use a novel and very flexible error model, for which there exists only an algorithm to evaluate the similarity between two elements in two dimensions at $O(m^4)$ time. We extend the algorithm to d dimensions, at $O(d!m^{2d})$ time and $O(d!m^{2d-1})$ space. We also give the first search algorithm for such model, which is $O(d!m^d n^d)$ time and $O(d!m^d n^{d-1})$ space. We show how to reduce the space cost to $O(d!3^d m^{2d-1})$ with little time penalty. Finally, we present the first sublinear-time (on average) searching algorithm (i.e. not all text cells are inspected), which is $O(kn^d/m^{d-1})$ for $k < (m/(d(\log_\sigma m - \log_\sigma d)))^{d-1}$, where σ is the alphabet size. After that error level the filter still remains better than dynamic programming for $k \le m^{d-1}/(d(\log_\sigma m - \log_\sigma d))^{(d-1)/d}$. These are the first search algorithms for the problem. As side-effects we extend to d dimensions an already proposed algorithm for two-dimensional exact string matching, and we obtain a sublinear-time filter to search in d dimensions allowing k mismatches.

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. Two dimensional pattern matching with errors has applications, for instance, in computer vision (i.e. searching a subimage inside a large image).In three dimensions, our algorithms may be useful for searching allowing errors in video data (where the time would be the third dimension) or in some types of medical data (e.g. MRI brain scans).

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

[*] Supported in part by Fondecyt grant 1-990627.

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 [20].

Krithivasan and Sitalakshmi (KS) [17] 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.

In [5], Baeza-Yates (BY) defined a more general extension (there called *RC*), where the errors can occur along rows or columns at any time. This model is much more robust and useful for more applications. We are interested in this general model in this work. Figure 1 gives an example.

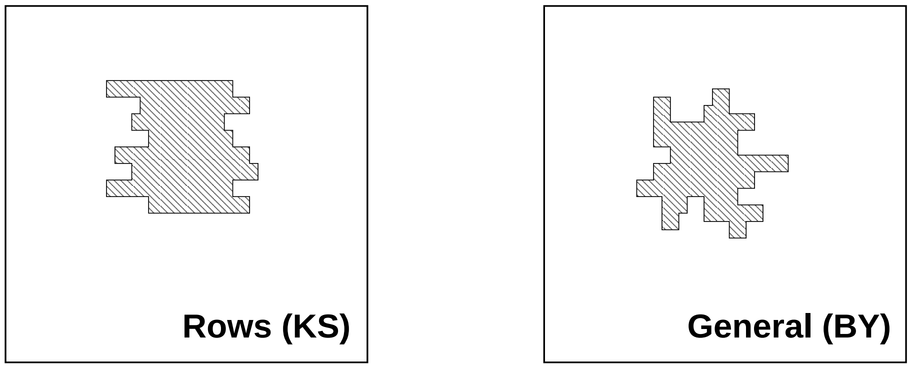

Fig. 1. Alternative error models.

Although in [5] they give an $O(m^4)$ time algorithm to compute the edit distance among two images of size $m \times m$, they do not give any algorithm to search a subimage inside a larger image allowing errors.

In this work, we first generalize the edit distance algorithm to d dimensions with complexity $O(d!m^{2d})$. We then give an $O(d!m^dn^d)$ time algorithm for the search problem, matching the same complexity of the simpler KS model in two dimensions, and show how to reduce the space requirements so that they depend only on the pattern size. We also give a new filtering algorithm that allows to quickly discard 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 $k < (m/(d(\log_\sigma m - \log_\sigma d)))^{d-1}$, where σ is the alphabet size. After that error level the filter changes its cost but remains better than dynamic programming for $k \leq m^{d-1}/(d(\log_\sigma m - \log_\sigma d))^{(d-1)/d}$. These are the first searching algorithms for this problem.

Two side-effects are obtained as well. First, we generalize to d dimensions and analyze a previously proposed algorithm to search in two dimensions not allowing errors. Second, we obtain a filter to search a pattern in d dimensions allowing up to k character substitutions.

2 Previous Work

The classical dynamic programming algorithm [20] to search a pattern in a text allowing errors uses dynamic programming and is $O(mn)$ time and $O(m)$ space.

This solution was later improved by a number of algorithms, which we do not cover here. The only one of interest to this work is a filtering algorithm [21, 8, 7]. It states that if a pattern is cut in $k+1$ pieces, then any occurrence with up to k errors must contain one of the pieces unchanged. This is obvious since k errors cannot alter the $k+1$ pieces given the edit operations that we consider (which 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 [12] or another Boyer-Moore type algorithm adapted to multipattern search. The total cost of verifications keeps below $O(n)$ if $k/m \leq 1/(3\log_\sigma m)$.

Two dimensional string matching was first considered by Bird and Baker [11, 10], who obtain $O(n^2)$ worst-case time. Good average results are presented by Zhu and Takaoka in [22]. The best average case result is due to Baeza-Yates and Régnier [9], who obtain $O(n^2/m)$ time on average and $O(n^2)$ in the worst case.

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. Under the substitutions model, one of the best results on the worst case is due to Amir and Landau [4], which achieves $O((k+\log\sigma)n^2)$ time but uses $O(n^2)$ space. A similar algorithm is presented in [13]. Ranka and Heywood solve the same 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 [15], with its analysis and space usage improved by Park [19]. The expected time is $O(n^2k/m^2\ \log_\sigma m)$

for $k < m^2/(4\log_\sigma m)$ using $O(m^2)$ space ($O(k)$ space on average). This time result is optimal for the expected case.

Krithivasan and Sitalakshmi (KS) [17] 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 $O(m^2n^2)$ time using a generalization of the classical one-dimensional algorithm. Krithivasan [16] presents for the same model an $O(m(k+\log m)n^2)$ algorithm that uses $O(mn)$ space. Amir and Landau [4] give an $O(k^2n^2)$ worst case time algorithm using $O(n^2)$ space. Amir and Farach [3] also considered non-rectangular patterns achieving $O(k(k+\sqrt{m\log m}\sqrt{k\log k})n^2)$ time.

In [6] we use the same model and improve the expected case to $O(n^2k\log_\sigma m/m^2)$ on average for $k < m(m+1)/(5\log_\sigma m)$, using $O(m^2)$ space. This time matches the optimal result allowing only substitutions, and is also optimal [15], being the restriction on k only a bit stricter. For higher error levels, [6] presents an algorithm with time complexity $O(n^2k/(\sqrt{\sigma}\log n))$, which works for $k < m(m+1)(1-e/\sqrt{\sigma})$. It is also shown that this limit on k cannot be improved.

In [5], Baeza-Yates defined more general models, where the errors can occur along rows or columns. Three distances R, C and L are defined, and for the first two it is shown that the filters of [6] can be applied to obtain the same complexity and slightly reduced tolerance to errors, i.e. $k < m(m+1)/(7\log_\sigma m)$. A fourth model defined in [5] is called RC, which generalizes R and C since the errors can occur along rows or columns at any time. This model is much more robust and useful for more applications, and is the one we use in this work. We cover this model in detail in the next section.

3 Multidimensional Approximate Searching

The classical dynamic programming algorithm [18] to compute the edit distance between two one-dimensional strings A and B of length m_1 and m_2 computes a matrix $C_{0..m_1,0..m_2}$. The value $C_{i,j}$ holds the edit distance between $A_{1..i}$ and $B_{1..j}$. The construction algorithm is as follows

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

and the distance $ed(A,B)$ is the final value of C_{m_1,m_2}. 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 $O(m_1m_2)$ space and time. It is easily adapted to search a pattern P in a text T allowing up to k errors [20]. 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 matrix is $C_{0..n,0..m}$ and the construction formula is

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

where the only change is that a pattern of length zero matches with no errors at any text position. All the positions i such that $C_{i,m} \leq k$ 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. We define $led(T, P)$ as the smallest edit distance among the pattern P and a suffix of T, and therefore $led(T_{1..i}, P) = C_{i,m}$.

In [5], a natural extension to the edit distance notion for two dimensional strings (or "images") A and B was defined (called RC in that paper, and ed_2 in this work). It allows the errors to occur along any dimension. An algorithm to compute the edit distance among two images is defined. For simplicity we assume that they are square and of the same size $m \times m$, although it is easy to remove that limitation. The algorithm computes a four-dimensional matrix $C_{0..m,0..m,0..m,0..m}$, so that $C_{i,j,p,q} = ed(A_{1..i,1..j}, B_{1..p,1..q})$. C is built using the formulas

$$\begin{array}{ll} C_{i,0,0,0} \leftarrow i \ , \ C_{0,j,0,0} \leftarrow j \\ C_{0,0,p,0} \leftarrow p \ , \ C_{0,0,0,q} \leftarrow q \\ C_{i,j,p,q} \leftarrow \min(& C_{i-1,j,p-1,q} + ed(A_{i,1..j}, B_{p,1..q}),\ C_{i-1,j,p,q} + j,\ C_{i,j,p-1,q} + q, \\ & C_{i,j-1,p,q-1} + ed(A_{1..i,j}, B_{1..p,q})\ C_{i,j-1,p,q} + i,\ C_{i,j,p,q-1} + p\ ) \end{array}$$

which has a very similar rationale of the one-dimensional case: at each point we can solve the last row (first line of the min() formula) or the last column (second line of the min() formula). In each case, we either insert the whole row, delete the whole row, or replace the row of A by the row of B (and $ed()$ gives the best way to do it). This algorithm is $O(m^6)$ time and $O(m^4)$ space. However, by precomputing all the values

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

(i.e. all the row-wise and column-wise alignments), the search time drops to $O(m^4)$ and the space does not change. This is because the $ed()$ of the C formula are obtained in constant time, and $Horiz$ consists of m^2 one-dimensional edit distance computations, among $A_{i,*}$ and $B_{p,*}$. The same holds for $Vert$.

The space can also be reduced to $O(m^3)$, as shown in [5]. We select, say, i as the most external variable of the iteration to fill the matrix. 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. The same can be done with $Horiz$ and $Vert$, by using i as the most external iteration variable.

In [5] they mention that this algorithm extends to d dimensions in time $O(m^{2d})$ but they do not give the details. We give a detailed algorithm in the next section and show that the exact complexity is $O(d!m^{2d})$. Also, no algorithm was given in [5] to search a subimage in a larger image using the above distance function. We do so in the following sections. We finally extend the one-dimensional filtering algorithm to more dimensions.

4 Edit Distance in More Dimensions

The idea of the previous section can be extended to compute $ed_d()$, i.e. the edit distance generalized to d dimensions. The algorithm is $O(d!m^{2d})$ time and $O(m^{2d-1})$ space.

A $(2d)$-dimensional matrix C is computed (d dimensions for A and d dimensions for B), and the $ed()$ of the above formula 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 mentioned in [5].

However, 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 j 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

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

which yields $O(d!m^{2d})$ time (which matches the improved $O(m^4)$ algorithm of [5] 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

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

which yields

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

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

5 A Dynamic Programming Search Algorithm

We modify the edit distance algorithm so that instead of computing the edit distance between two elements, it searches a small pattern P of size m^d inside a large text T of size n^d. The idea is a simple modification of the edit distance algorithm. For two dimensions the formula is as follows

$$\begin{array}{lll} C_{i,0,0,0} & \leftarrow 0 \ , \ C_{0,j,0,0} & \leftarrow 0 \\ C_{0,0,p,0} & \leftarrow p \ , \ C_{0,0,0,q} & \leftarrow q \\ C_{i,j,p,q} & \multicolumn{2}{l}{\leftarrow \min(\ C_{i-1,j,p-1,q} + led(A_{i,1..j}, B_{p,1..q}),\ C_{i-1,j,p,q} + q,\ C_{i,j,p-1,q} + q,} \\ & \multicolumn{2}{l}{\qquad C_{i,j-1,p,q-1} + led(A_{1..i,j}, B_{1..p,q})\ C_{i,j-1,p,q} + p,\ C_{i,j,p,q-1} + p\ )} \end{array}$$

where the only differences are that the basic values are zero when the pattern is of size zero, that we penalize insertions and deletions according to the pattern size, and that instead of $ed()$ we use $led()$, so that we select the best suffix of the text along each dimension. If we are searching allowing up to k errors, then we report all text (i,j) positions such that $C_{i,j,m,m} \leq k$.

The form to extend this to more dimensions is immediate. By repeating the analysis of the above section, we see that the naive algorithm is $O(d!(mn)^{\frac{d(d+1)}{2}})$ time and $O(m^d n^{d-1})$ space (since n is much larger than m, we select one of the text coordinates as the most external variable). By precomputing the distances in lower dimensions, the search algorithm is $O(d!m^d n^d)$ time and $O(d!m^d n^{d-1})$ space.

5.1 Correctness

We now prove that the above algorithm is correct (in two dimensions). This extends easily to more dimensions.

Lemma: For each text position (i,j), it is possible to perform $C_{i,j,m,m}$ edit operations in the pattern P (converting it into P') so that the pattern P' matches the text suffix $T..i,..j$, and this is not possible with less operations.

Proof: We prove the Lemma for any $C_{i,j,p,q}$. The Lemma is obviously true for the base case of the formula. For the recursive case, we inductively assume that the Lemma is true for the subproblems. We consider the first line of the update formula, which corresponds to the rows (the other cases are equivalent).

If the value for $C_{i,j,p,q}$ is obtained using a row insertion in the pattern, then we can inductively align $P_{1..p,1..q}$ at $T_{..i-1,j}$ with cost $C_{i-1,j,p,q}$, and then insert the text segment $T_{i,j-p+1..j}$ in P at the cost of p more errors so as to align $P_{1..p,1..q}$ at $T_{..i,j}$.

If the value for $C_{i,j,p,q}$ is obtained using a row deletion in the pattern, then we can inductively align $P_{1..p-1,1..q}$ at $T_{..i,j}$ with cost $C_{i,j,p-1,q}$, and then delete the pattern row $P_{p,1..q}$ from P at the cost of p more errors so as to align $P_{1..p,1..q}$ at $T_{..i,j}$.

Finally, if we obtain $C_{i,j,p,q}$ by replacing $P_{p,1..q}$ with a row suffix of $T_{i,..j}$, then the $led()$ of the formula gives the optimal way to do it, so that we align

$P_{1..p-1,1..q}$ at $T_{..i-1,j}$ with cost $C_{i-1,j,p-1,q}$, and then convert the pattern row $P_{p,1..q}$ to some text row suffix of $T_{i,..j}$, at $led(T_{i,1..j}, P_{p,1..q})$ cost.

Alternatively, we can use the recursion on the column values. It is also clear that this cannot be done better. On the other hand, we can use induction over the number of dimensions to show that the Lemma is correct for any d-dimensional problem.

5.2 Reducing the Space Requirements

The space requirement of the algorithm is $O(d!m^d n^{d-1})$, which is too high. This is awkward since the problem exhibits high locality. That is, the fact that a text position matches or not depends only on the last $(m+k)^d$-size text "suffix" that ends at that point. In fact, if $k > m$ we just need to start $2m$ positions behind the subtext at each dimension, since if more than m errors are made along a given line, it is better to just perform m replacements.

Therefore, if we cut the text in $(n/s)^d$ subtexts (of d dimensions) of size s^d, we can work separately at each subtext provided we start, at each dimension, $m + \min(m, k)$ positions behind the cube so as to have the context properly initialized when we reach the cube. The total time is $(n/s)^d d! m^d (m + \min(m, k) + s)^d$, and the total space is $d! m^d (m + \min(m, k) + s)^{d-1}$.

For instance, we may select $s = m$, and then we obtain an algorithm which is at most $O(d! 3^d m^d n^d)$ time and $O(d! 3^d m^{2d-1})$ space (and less if $k < m$), which is much more reasonable. The minimum possible space requirement is $O(d! 2^d m^{2d-1})$, at time cost $O(d! 2^d m^{2d} n^d)$ (that is, $s = 1$).

6 Multidimensional Exact String Matching

In [9], they allow to search, 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 [9], 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 i) 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 2. Each time one of the $(d-1)$ dimensional subpatterns is found in a text position, the complete d-dimensional pattern is checked.

An important part of the analysis of [9] for two dimensions is that the total cost to verify potential matches is not too large. It is not immediate that this

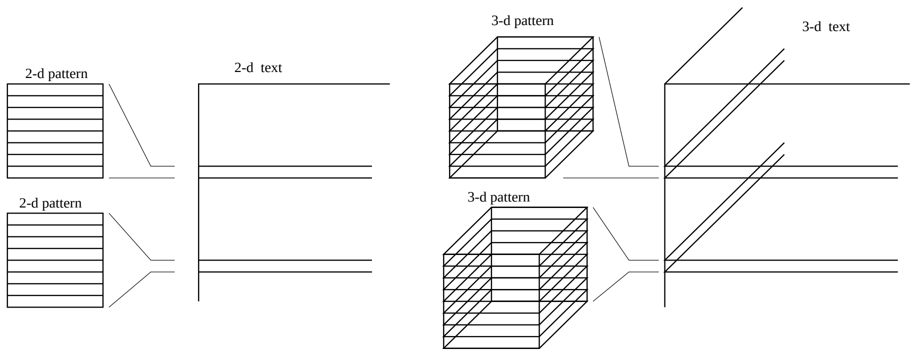

Fig. 2. Algorithm for exact searching. All the pattern "rows" are searched in n/m text "rows" at the same time.

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 $\sum 1/\sigma^c = O(1)$ (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 above result for $r = 1$ can be bounded by $O(n^d/m^{d-1})$.

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 now this algorithm as a building block.

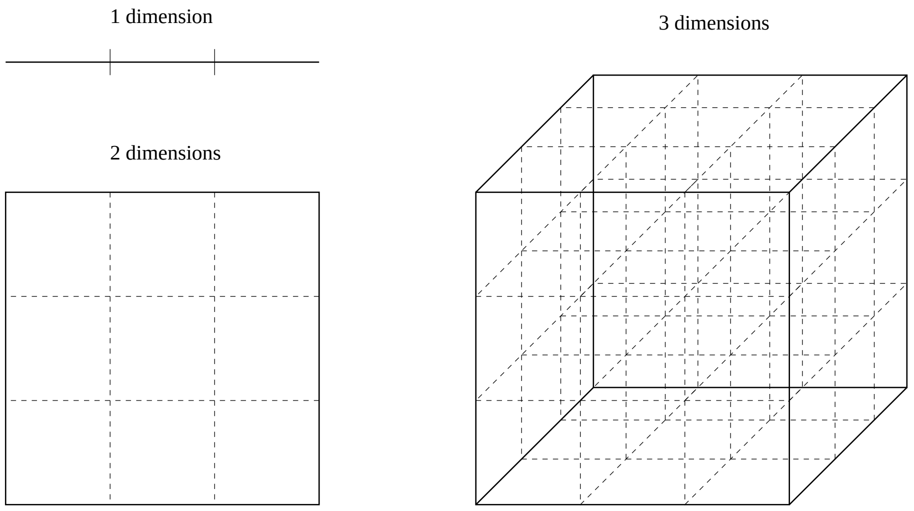

Fig. 3. Filtering algorithm for $j = 3$. The maximum possible k so that some block appears unchanged is 2, 2, and 8 as the dimension grows.

7 A Fast Filter for Multidimensional Approximate Search

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 3). 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

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

Each time one such piece is found, we have to verify a surrounding text area to check for a possible match. This area extends $(m + 2\min(m,k))$ positions along each dimension (since the match could start at most $\min(m,k)$ positions backward or finish up to $\min(m,k)$ positions forward). Hence, the cost of a verification is the same as that of searching the pattern in a text of size

$(m + 2\min(m,k))^d$ allowing errors, which is $O(d!m^d(m + 2\min(m,k))^d)$. 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 $j^d d!m^d(m + 2\min(m,k))^d n^d/\sigma^{(m/j)^d}$.

Notice that, since we only verify pieces of the text of size $(m+2\min(m,k))^d$, the space requirement of this algorithm is $O(d!m^d(m + 2\min(m,k))^{d-1})$ (this corresponds to the verification phase, since the search of the pieces needs much less, i.e. $O(m^d)$). This is a form of our previous technique to reduce space requirements (recall Section 5.2) equivalent to using $s = \min(m,k)$. However, in this case we only check a few portions of the text.

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 $j \le m$, 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

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

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

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

while the second dominates from $k > k_0$ until

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

In the maximum acceptable value $k = m^{d-1} - 1$, the search complexity becomes $O(d!3^d m^{3d} n^d)$, 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. We can compare against the version that uses the same amount of space (which corresponds to $s = \min(m,k)$), whose time complexity is $O(d!2^d m^{2d} n^d)$; or we can compare it against the fastest version of dynamic programming, which needs much more space and whose time cost is $O(d!m^d n^d)$. In either case we have that the k range for which the filter is better than dynamic programming is

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

where the difference in the version of dynamic programming used affects lower order terms only.

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 $k \le k_0$. 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 $k < (m/(d \log_\sigma m))^{d-1}$, and it improves over dynamic programming for $k \le m^{d-1}/(2d \log_\sigma m)^{(d-1)/d}$. Figure 4 illustrates the result of the analysis.

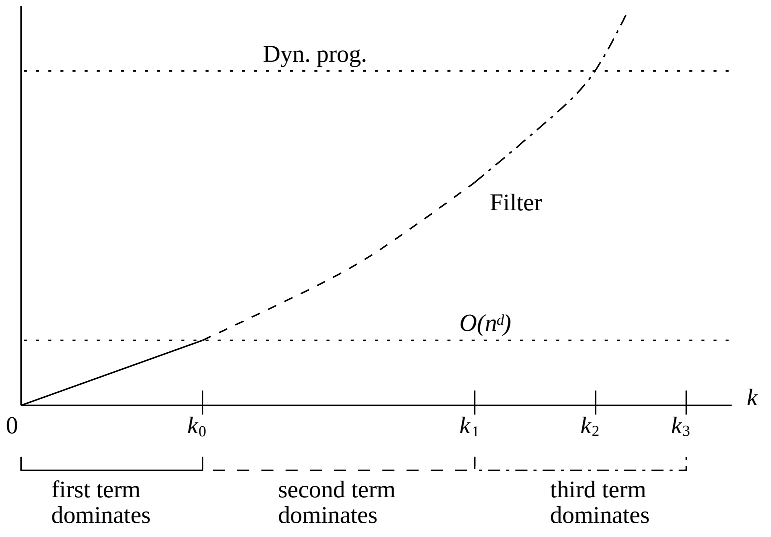

Fig. 4. The complexity of the proposed filter, depending on k.

7.1 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, which multiplies the probability by $\binom{j^d}{jk}$. Finally, if the text area passes this filter, we verify it at the same cost as before (i.e. $d!m^d(m + 2\min(m,k))^d$). The new search cost is therefore

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

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 into 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.

7.2 Adapting the Filter to Substitutions

The problem of searching a pattern allowing k substitutions is much simpler, and we can apply our machinery to that case as well. 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.

The same filter proposed in this section works for the case of k substitutions, the only difference being that in this case the cost to verify a candidate text position is $O(k)$, i.e. much cheaper. The search cost still has three terms, the first one being dominant for $k \leq k_0$. The second component 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.

8 Conclusions

We have presented the first algorithms to search a multidimensional pattern in multidimensional text allowing editing errors along any dimension. This is a new model recently proposed in [5]. We have generalized to d dimensions their algorithm to compute edit distance, where we obtained $O(d!m^{2d})$ time and $O(d!m^{2d-1})$ space (where the compared elements are of size m^d).

We have obtained and proved the correctness of the first search algorithm for this model, where a pattern of size m^d is searched in a text of size n^d at $O(d!m^dn^d)$ time and $O(d!m^dn^{d-1})$ space. We have shown how to trade time for space, for instance with $O(d!3^dm^{2d-1})$ space we have $O(d!3^dm^dn^d)$ time.

Finally, we have proposed a filter which obtains roughly $O(kn^d/m^{d-1})$ (i.e. sublinear) average search time for $k < (m/(d(\log_\sigma m - \log_\sigma d)))^{d-1}$, where σ is the alphabet size. After that error level the filter changes its cost but remains better than dynamic programming for $k \leq m^{d-1}/(d(\log_\sigma m - \log_\sigma d))^{(d-1)/d}$. For instance, in two dimensions the filter is sublinear for $k < m/(2\log_\sigma m)$ and better than dynamic programming 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 we present the algorithms for square d-dimensional pattern and text, they also work for hyper-rectangular elements.

Our work is a (very preliminary) step towards presenting a combinatorial alternative to the current image processing technology. However, for this to be successful, we must allow not only errors but also rotations, scalings and deformations in the images. There are some works addressing those issues separately [2,14], but they have not been merged. We are currently working on this integration.

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 and G. Calinescu. Alphabet independent and dictionary scaled matching. In *Proc. CPM'96*, number 1075 in LNCS, pages 320–334, 1996.
3. A. Amir and M. Farach. Efficient 2-dimensional approximate matching of non-rectangular figures. In *Proc. SODA'91*, pages 212–223, 1991.
4. A. Amir and G. Landau. Fast parallel and serial multidimensional approximate array matching. *Theoretical Computer Science*, 81:97–115, 1991.
5. R. Baeza-Yates. Similarity in two-dimensional strings. In *Proc. COCOON'98*, number 1449 in LNCS, pages 319–328, Taipei, Taiwan, August 1998.
6. 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.
7. R. Baeza-Yates and G. Navarro. Faster approximate string matching. *Algorithmica*, 23(2):127–158, 1999. To appear. Preliminary version in *Proc. CPM'96*.
8. R. Baeza-Yates and C. Perleberg. Fast and practical approximate pattern matching. In *Proc. CPM'92*, LNCS 644, pages 185–192, 1992.
9. R. Baeza-Yates and M. Régnier. Fast two dimensional pattern matching. *Information Processing Letters*, 45:51–57, 1993.
10. 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.
11. R. Bird. Two dimensional pattern matching. *Inf. Proc. Letters*, 6:168–170, 1977.
12. 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.
13. M. Crochemore and W. Rytter. *Text Algorithms.* Oxford University Press, Oxford, UK, 1994.
14. 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.
15. J. Karkkäinen and E. Ukkonen. Two and higher dimensional pattern matching in optimal expected time. In *Proc. SODA'94*, pages 715–723. SIAM, 1994.
16. K. Krithivasan. Efficient two-dimensional parallel and serial approximate pattern matching. Technical Report CAR-TR-259, University of Maryland, 1987.
17. K. Krithivasan and R. Sitalakshmi. Efficient two-dimensional pattern matching in the presence of errors. *Information Sciences*, 43:169–184, 1987.
18. 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.
19. K. Park. Analysis of two dimensional approximate pattern matching algorithms. In *Proc. CPM'96*, LNCS 1075, pages 335–347, 1996.
20. P. Sellers. The theory and computation of evolutionary distances: pattern recognition. *J. of Algorithms*, 1:359–373, 1980.
21. S. Wu and U. Manber. Fast text searching allowing errors. *CACM*, 35(10):83–91, October 1992.
22. R. Zhu and T. Takaoka. A technique for two-dimensional pattern matching. *Comm. ACM*, 32(9):1110–1120, 1989.