Rotation and Lighting Invariant Template Matching

Kimmo Fredriksson[1], Veli Mäkinen[2]*, and Gonzalo Navarro[3]**

[1] Department of Computer Science, University of Joensuu.
`kfredrik@cs.joensuu.fi`

[2] Department of Computer Science, University of Helsinki.
`vmakinen@cs.helsinki.fi`

[3] Center for Web Research, Department of Computer Science, University of Chile.
`gnavarro@dcc.uchile.cl`

Abstract. We address the problem of searching for a two-dimensional pattern in a two-dimensional text (or image), such that the pattern can be found even if it appears rotated and brighter or darker than its occurrence. Furthermore, we consider approximate matching under several tolerance models. We obtain algorithms that are almost worst-case optimal. The complexities we obtain are very close to the best current results for the case where only rotations, but not lighting invariance, are supported. These are the first results for this problem under a combinatorial approach.

1 Introduction

We consider the problem of finding the occurrences of a two-dimensional *pattern* of size $m \times m$ cells in a two-dimensional *text* of size $n \times n$ cells, when all possible rotations of the pattern are allowed and also pattern and text may have differences in brightness. This stands for *rotation and lighting invariant template matching.* Text and pattern are seen as images formed by cells, each of which has a gray level value, also called a color.

Template matching has numerous important applications from science to multimedia, for example in image processing, content based information retrieval from image databases, geographic information systems, processing of aerial images, to name a few. In all these cases, we want to find a small subimage (the pattern) inside a large image (the text) permitting rotations (a small degree or any). Furthermore, pattern and text may have been photographed under different lighting conditions, so one may be brighter than the other.

The traditional approach to this problem [2] is to compute the cross correlation between each text location and each rotation of the pattern template. This

* A part of the work was done while visiting University of Chile under a researcher exchange grant from University of Helsinki.

** Funded by Millennium Nucleus Center for Web Research, Grant P01-029-F, Mideplan, Chile.

can be done reasonably efficiently using the Fast Fourier Transform (FFT), requiring time $O(Kn^2 \log n)$ where K is the number of rotations sampled. Typically K is $O(m)$ in the two-dimensional (2D) case, and $O(m^3)$ in the 3D case, which makes the FFT approach very slow in practice. In addition, lighting-invariant features may be defined in order to make the FFT insensitive to brightness. Also, in many applications, "close enough" matches of the pattern are also accepted. To this end, the user may specify, for example, a parameter κ such that matches that have at most κ differences with the pattern should be accepted, or a parameter δ such that gray levels differing by less than δ are considered equal. The definition of the matching conditions is called the "matching model".

Rotation invariant template matching was first considered from a combinatorial point of view in [8,9]. Since then, several fast filters have been developed for diverse matching models [10,7,6]. These represent large performance improvements over the FFT-based approach. The worst-case complexity of the problem was also studied [1,7]. However, lighting invariance has not been considered in this scenario.

On the other hand, *transposition invariant* string matching was considered in music retrieval [3,11]. The aim is to search for (one-dimensional) patterns in texts such that the pattern may match the text after all its characters (notes) are shifted by some value. The reason is that such an occurrence will sound like the pattern to a human, albeit in a different scale. In this context, efficient algorithms for several approximate matching functions were developed in [12].

We note that transposition invariance becomes lighting invariance when we replace musical notes by gray levels of cells in an image. Hence, the aim of this paper is to enrich the existing algorithms for rotation invariant template matching [7] with the techniques developed for transposition invariance [12] so as to obtain rotation and lighting invariant template matching. It turns out that lighting invariance can be added at very little extra cost. The key technique exploited is *incremental distance computation*; we show that several transposition invariant distances can be computed incrementally taking the computation done with the previous rotation into account in the next rotation angle.

Let us now determine which are the reasonable matching models. In [7], some of the models considered were useful only for binary images, a case where obviously we are not interested in this paper. We will address models that make sense for gray level images. We define three transposition-invariant distances: $d_{\mathrm{H}}^{\mathrm{t},\delta}$, which counts how many pattern and text cells differ by more than δ; $d_{\mathrm{MAD}}^{\mathrm{t},\kappa}$, which is the maximum color difference between pattern and text cells when up to κ outliers are permitted; and $d_{\mathrm{SAD}}^{\mathrm{t},\kappa}$, which is the sum of absolute color differences between pattern and text cells permitting up to κ outliers. Table 1 shows our complexities to compute these distances for every possible rotation of a pattern centered at a fixed text position. Variable σ is the number of different gray levels (assume $\sigma = \infty$ if the alphabet is not a finite discrete range). A lower bound to this problem is $O(m^3)$, achieved in [7] without lighting invariance.

We also define two search problems, consisting in finding all the transposition-invariant rotated occurrences of P in T such that: (1) there are at most κ cells of P differing by more than δ from their text cell (δ-matching); or (2) the sum

Table 1. Worst-case complexities to compute the different distances defined.

Distance	Complexity
$d_{\mathrm{H}}^{\mathrm{t},\delta}$	$\min(\log m, \sigma + (\delta+1))m^3$
$d_{\mathrm{MAD}}^{\mathrm{t},\kappa}$	$(\min(\kappa,\sigma) + \log\min(m,\sigma))m^3$
$d_{\mathrm{SAD}}^{\mathrm{t},\kappa}$	$(\min(\kappa,\sigma) + \log\min(m,\sigma))m^3$

of absolute difference between cells in P and T, except for κ outliers, does not exceed γ (γ-matching). Note that δ-matching can be solved by examining every text cell and reporting it if $d_{\mathrm{H}}^{\mathrm{t},\delta}(P,T) \leq \kappa$, or if $d_{\mathrm{MAD}}^{\mathrm{t},\kappa}(P,T) \leq \delta$ around the text cell. Hence any $O(f(m))$ algorithm for computing $d_{\mathrm{H}}^{\mathrm{t},\delta}$ or $d_{\mathrm{MAD}}^{\mathrm{t},\kappa}(P,T)$ yields an $O(f(m)n^2)$ algorithm for δ-matching. Similarly, γ-matching can be reduced to checking whether $d_{\mathrm{SAD}}^{\mathrm{t},\kappa}(P,T) \leq \gamma$ around each cell. Without transposition invariance all searching worst cases are $O(m^3n^2)$ [7].

We remark that we have developed algorithms that work on arbitrary alphabets, but we have also taken advantage of the case where the alphabet is a discrete range of integer values.

A full version of this paper [5] considers also (δ,γ)-matching and optimal average case search complexities.

2 Definitions

Let $T = T[1..n, 1..n]$ and $P = P[1..m, 1..m]$ be arrays of unit squares, called *cells*, in the (x,y)-plane. Each cell has a value in an alphabet called Σ, sometimes called its gray level or its color. A particular case of interest is that of Σ being a finite integer range of size σ. The corners of the cell for $T[i,j]$ are $(i-1,j-1), (i,j-1), (i-1,j)$ and (i,j). The center of the cell for $T[i,j]$ is $(i-\frac{1}{2}, j-\frac{1}{2})$. The array of cells for pattern P is defined similarly. The center of the whole pattern P is the center of the cell in the middle of P. Precisely, assuming for simplicity that m is odd, the center of P is the center of cell $P[\frac{m+1}{2}, \frac{m+1}{2}]$.

Assume now that P has been moved on top of T using a rigid motion (translation and rotation), such that the center of P coincides exactly with the center of some cell of T (*center-to-center assumption*). The location of P with respect to T can be uniquely given as $((i,j),\theta)$ where (i,j) is the cell of T that matches the center of P, and θ is the angle between the x-axis of T and the x-axis of P. The (approximate) occurrence between T and P at some location is defined by comparing the values of the cells of T and P that overlap. We will use the centers of the cells of T for selecting the comparison points. That is, for the pattern at location $((i,j),\theta)$, we look which cells of the pattern cover the centers of the cells of the text, and compare the corresponding values of those cells (Fig. 1).

More precisely, assume that P is at location $((i,j),\theta)$. For each cell $T[r,s]$ of T whose center belongs to the area covered by P, let $P[r',s']$ be the cell of P such that the center of $T[r,s]$ belongs to the area covered by $P[r',s']$. Then $M(T[r,s]) = P[r',s']$, that is, our algorithms compare the cell $T[r,s]$ of T against the cell $M(T[r,s])$ of P.

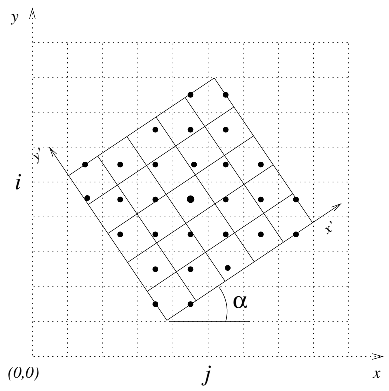

Fig. 1. Each text cell is matched against the pattern cell that covers the center of the text cell.

Hence the *matching function* M is a function from the cells of T to the cells of P. Now consider what happens to M when angle θ grows continuously, starting from $\theta = 0$. Function M changes only at the values of θ such that some cell center of T hits some cell boundary of P. It was shown in [8] that this happens $O(m^3)$ times, when P rotates full 2π radians. This result was shown to be also a lower bound in [1]. Hence there are $\Theta(m^3)$ relevant orientations of P to be checked. The set of angles for $0 \le \theta \le \pi/2$ is $A = \{\beta, \pi/2 - \beta \mid \beta = \arcsin \frac{h+\frac{1}{2}}{\sqrt{i^2+j^2}} - \arcsin \frac{j}{\sqrt{i^2+j^2}};\ i = 1, 2, \ldots, \lfloor m/2 \rfloor; j = 0, 1, \ldots, \lfloor m/2 \rfloor; h = 0, 1, \ldots, \lfloor \sqrt{i^2+j^2} \rfloor\}$. By symmetry, the set of possible angles θ, $0 \le \theta < 2\pi$, is $\mathcal{A} = A \ \cup \ A + \pi/2 \ \cup \ A + \pi \ \cup \ A + 3\pi/2$.

Furthermore, pattern P matches at location $((i,j),\theta)$ with lighting invariance if there is some integer transposition t such that $T[r,s]+t = P[r',s']$ for all $[r',s']$ in the area of P.

Once the position and rotation $((i,j),\theta)$ of P in T define the matching function, we can compute different kinds of *distances* between the pattern and the text. Lighting-invariance versions of the distances choose the transposition minimizing the basic distance. Interesting distances for gray level images follow.

Hamming Distance (H): The number of times $T[r,s] \neq P[r',s']$ occurs, over all the cells of P, that is, $d_{\mathrm{H}}(i,j,\theta,t) = \sum_{r',s'} [\textbf{if } T[r,s] + t \neq P[r',s'] \textbf{ then } 1 \textbf{ else } 0]$, and $d^{\mathrm{t}}_{\mathrm{H}}(i,j,\theta) = \min_t d_{\mathrm{H}}(i,j,\theta,t)$. This can be extended to distance d^{δ}_{H} and its transposition-invariant version $d^{\mathrm{t},\delta}_{\mathrm{H}}$, where colors must differ by more than δ in order to be considered different, that is, $T[r,s] + t \notin [P[r',s'] - \delta, P[r',s'] + \delta]$.

Maximum Absolute Differences (MAD): The maximum value of $|T[r,s] - P[r',s']|$ over all the cells of P, that is, $d_{\mathrm{MAD}}(i,j,\theta,t) = \max_{r',s'} |T[r,s] + t - P[r',s']|$, and $d^{\mathrm{t}}_{\mathrm{MAD}}(i,j,\theta) = \min_t d_{\mathrm{MAD}}(i,j,\theta,t)$. This can be extended to distance $d^{\kappa}_{\mathrm{MAD}}$ and its transposition-invariant version $d^{\mathrm{t},\kappa}_{\mathrm{MAD}}$, so that up to κ pattern cells are freed from matching the text. Then the problem is to compute the MAD distance with the best choice of κ outliers that are not included in the maximum.

Sum of Absolute Differences (SAD): The sum of the $|T[r,s]-P[r',s']|$ values over all the cells of P, that is, $d_{\mathrm{SAD}}(i,j,\theta,t) = \sum_{r',s'} |T[r,s]+t-P[r',s']|$, and $d^{\mathrm{t}}_{\mathrm{SAD}}(i,j,\theta) = \min_t d_{\mathrm{SAD}}(i,j,\theta,t)$. Similarly, this distance can be extended to $d^{\kappa}_{\mathrm{SAD}}$ and its transposition-invariant version $d^{\mathrm{t},\kappa}_{\mathrm{SAD}}$, where up to κ pattern cells can be removed from the summation.

3 Efficient Algorithms

In [1] it was shown that for the problem of the two dimensional pattern matching allowing rotations the worst case lower bound is $\Omega(n^2m^3)$. We have shown in [7] a simple way to achieve this lower bound for any of the distances under consideration (without lighting invariance). The idea is that we will check each possible text center, one by one. So we have to pay $O(m^3)$ per text center to achieve the desired complexity. What we do is to compute the distance we want for each possible rotation, by reusing most of the work done for the previous rotation. Once the distances are computed, it is easy to report the triples (i,j,θ) where these values are smaller than the given thresholds (δ and/or γ). Only distances d_{H} (with $\delta=0$) and d_{SAD} (with $\kappa=0$) were considered.

Assume that, when computing the set of angles $\mathcal{A} = (\beta_1, \beta_2, \ldots)$, we also sort the angles so that $\beta_i < \beta_{i+1}$, and associate with each angle β_i the set $\mathcal{C}_i$ containing the corresponding cell centers that must hit a cell boundary at β_i. This is done in a precomputation step that depends only on m, not on P or T. Hence we can evaluate the distance functions (such as d_{SAD}) incrementally for successive rotations of P. That is, assume that the distance has been evaluated for β_i, then to evaluate it for rotation β_{i+1} it suffices to re-evaluate the cells restricted to the set $\mathcal{C}_i$. This is repeated for each $\beta \in \mathcal{A}$. Therefore, the total time for evaluating the distance for P centered at some position in T, for all possible angles, is $O(\sum_i |\mathcal{C}_i|)$. This is $O(m^3)$ because each fixed cell center of T, covered by P, can belong to some $\mathcal{C}_i$ at most $O(m)$ times. To see this, note that when P is rotated the whole angle 2π, any cell of P traverses $O(m)$ cells of T.

If we want to add lighting invariance to the above scheme, a naive approach is to run the algorithm for every possible transposition, for a total cost of $O(n^2m^3\sigma)$. In case of a general alphabet there are $O(m^2)$ relevant transpositions at each rotation (that is, each pattern cell can be made to match its corresponding text cell). Hence the cost raises to $O(n^2m^5)$.

In order to do better, we must be able to compute the optimal transposition for the initial angle and then maintaining it when some characters of the text change (because the pattern has been aligned over a different text cell). If we take $f(m)$ time to do this, then our lighting invariant algorithm becomes worst-case time $O(n^2m^3f(m))$. In the following we show how can we achieve this for each of the distances under consideration.

3.1 Distance $d_{\mathrm{H}}^{\mathrm{t},\delta}$

As proved in [12], the optimal transposition for Hamming distance is obtained as follows. Each cell $P[r', s']$, aligned to $T[r, s]$, *votes* for a range of transpositions $[P[r', s'] - T[r, s] - \delta, P[r', s'] - T[r, s] + \delta]$, for which it would match. If a transposition receives v votes, then its Hamming distance is $m^2 - v$. Hence, the transposition that receives most votes is the one yielding distance $d_{\mathrm{H}}^{\mathrm{t},\delta}$. Let us now separate the cases of integer and general alphabets.

Integer alphabet. The original algorithm [12] obtains $O(\sigma + |P|)$ time on integer alphabet, by bucket-sorting the range extremes and then traversing them linearly so as to find the most voted transposition (a counter is incremented when a range starts and decremented when it finishes).

In our case, we have to pay $O(\sigma + m^2)$ in order to find the optimal transposition for the first rotation angle. The problem is how to recompute the optimal transposition once some text cell $T[r, s]$ changes its value (due to a small change in rotation angle). The net effect is that the range of transpositions given by the old cell value loses a vote and a new range gains a vote.

We use the fact that the alphabet is an integer range, so there are $O(\sigma)$ possible transpositions. Each transposition can be classified according to the number of votes it has. There are $m^2 + 1$ lists L_i, $0 \le i \le m^2$, containing the transpositions that currently have i votes. Hence, when a range of transpositions loses/gains one vote, the $2\delta + 1$ transpositions are moved to the lower/upper list. An array pointing to the list node where each transposition appears is necessary to efficiently find each of those $2\delta + 1$ transpositions. We need to keep control of which is the highest-numbered non-empty list, which is easily done in constant time per operation because transpositions move only from one list to the next/previous. Initially we pay $O(\sigma + m^2)$ to initialize all the lists and put all the transpositions in list L_0, then $O((\delta + 1)m^2)$ to process the votes of all the cells, and then $O(\delta + 1)$ to process each cell that changes. Overall, when we consider all the $O(m^3)$ cell changes, the scheme is $O(\sigma + (\delta + 1)m^3)$. This is our complexity to compute distance $d_{\mathrm{H}}^{\mathrm{t},\delta}$ between a pattern and a text center, considering all possible rotations and transpositions.

General alphabet. Let us resort to a more general problem of *dynamic range voting*: In the static case we have a multiset $S = \{[\ell, r]\}$ of one-dimensional closed ranges, and we are interested in obtaining a point p that is included in most ranges, that is $\mathrm{maxvote}(S) = \max_p |\{[\ell, r] \in S \mid \ell \le p \le r\}|$. In the dynamic case a new range is added to or an old one is deleted from S, and we must be able to return $\mathrm{maxvote}(S)$ after each update.

Notice that our original problem of computing $d_{\mathrm{H}}^{\mathrm{t},\delta}$ from one rotation angle to another is a special case of dynamic range voting; multiset S is $\{[P[r', s'] - T[r, s] - \delta, P[r', s'] - T[r, s] + \delta] \mid M(T[r, s]) = P[r', s']\}$ in one rotation angle, and in the next one some $T[r, s]$ changes its value. That is, the old range is deleted and the new one is inserted, after which $\mathrm{maxvote}(S)$ is requested to compute the distance $d_{\mathrm{H}}^{\mathrm{t},\delta} = m^2 - \mathrm{maxvote}(S)$ in the new angle.

We show that dynamic range voting can be supported in $O(\log |S|)$ time, which immediately gives an $O(m^3 \log m)$ time algorithm for computing $d_{\mathrm{H}}^{\mathrm{t},\delta}$ between a pattern and a text center, considering all rotations and transpositions.

First, notice that the point that gives $\mathrm{maxvote}(S)$ can always be chosen among the endpoints of ranges in S. We store each endpoint e in a balanced binary search tree with key e. Let us denote the leaf whose key is e simply by (leaf) e. With each endpoint e we associate a value $\mathrm{vote}(e)$ (stored in leaf e) that gives the number $|\{[\ell, r] \mid \ell \leq e \leq r, [\ell, r] \in S\}|$, where the set is considered as a multiset (same ranges can have multiple occurrences). In each internal node v, value $\mathrm{maxvote}(v)$ gives the maximum of the $\mathrm{vote}(e)$ values of the leaves e in its subtree. After all the endpoints e are added and the values $\mathrm{vote}(e)$ in the leaves and values $\mathrm{maxvote}(v)$ in the internal nodes are computed, the static case is solved by taking the value $\mathrm{maxvote}(\mathit{root}) = \mathrm{maxvote}(S)$ in the root node of the tree.

A straightforward way of generalizing the above approach to the dynamic case would be to recompute all values $\mathrm{vote}(e)$ that are affected by the insertion/deletion of a range. This would, however, take $O(|S|)$ time in the worst case. To get a faster algorithm, we only store the changes of the votes in the roots of certain subtrees so that $\mathrm{vote}(e)$ for any leaf e can be computed by summing up the changes from the root to the leaf e.

For now on, we refer to $\mathrm{vote}(e)$ and $\mathrm{maxvote}(v)$ as virtual values, and replace them with counters $\mathrm{diff}(v)$ and values $\mathrm{maxdiff}(v)$. Counters $\mathrm{diff}(v)$ are defined implicitly so that for all leaves of the tree it holds

$$\mathrm{vote}(e) = \sum_{v \in \mathrm{path}(\mathit{root}, e)} \mathrm{diff}(v), \tag{1}$$

where $\mathrm{path}(\mathit{root}, e)$ is the set of nodes in the path from the root to a leaf e (including the leaf). We note that there are several possible ways to choose $\mathrm{diff}(v)$ values so that they satisfy the definition. Values $\mathrm{maxdiff}(v)$ are defined recursively as

$$\max(\mathrm{maxdiff}(v.\mathit{left}) + \mathrm{diff}(v.\mathit{left}), \mathrm{maxdiff}(v.\mathit{right}) + \mathrm{diff}(v.\mathit{right})), \tag{2}$$

where $v.\mathit{left}$ and $v.\mathit{right}$ are the left and right child of v, respectively. In particular, $\mathrm{maxdiff}(e) = 0$ for any leaf node e. One easily notices that

$$\mathrm{maxvote}(v) = \mathrm{maxdiff}(v) + \sum_{v' \in \mathrm{path}(\mathit{root}, v)} \mathrm{diff}(v'),$$

which also gives as a special case Equation (1) once we notice that $\mathrm{maxvote}(e) = \mathrm{vote}(e)$ for each leaf node e.

Our goal is to maintain diff() and maxdiff() values correctly during insertions and deletions. We have three different cases to consider: (i) How to compute the value $\mathrm{diff}(e)$ for a new endpoint of a range, (ii) how to update the values of diff() and maxdiff() when a range is inserted/deleted, and (iii) how to update the values during rotations to rebalance the tree.

Case (i) is handled by storing in each leaf an additional counter end(e). It gives the number of ranges whose rightmost endpoint is e. Assume that this value is computed for all existing leaves. When we insert a new endpoint e, we either find a leaf labeled e or otherwise there is a leaf e' after which e is inserted. In the first case vote(e) remains the same and in the latter case vote(e) = vote(e') − end(e'), because e is included in the same ranges as e' except those that end at e'. Notice also that vote(e) = 0 in the degenerate case when e is the leftmost leaf. The +1 vote induced by the new range whose endpoint e is, will be handled in case (ii). To make $\mathrm{vote}(e) = \sum_{v' \in \mathrm{path}(root,e)} \mathrm{diff}(v')$, we fix diff($e$) so that $\mathrm{vote}(e) = \mathrm{diff}(e) + \sum_{v' \in \mathrm{path}(root,v)} \mathrm{diff}(v')$, where v is the parent of e. Once the maxdiff() values are updated in the path from e to the root, we can conclude that all the necessary updates are done in $O(\log |S|)$ time.

Let us then consider case (ii). Recall the one-dimensional range search on a balanced binary search tree (see e.g. [4], Section 5.1). We use the fact that one can find in $O(\log |S|)$ time the minimal set of nodes, say F, such that the range $[\ell, r]$ of S is *partitioned* by F; the subtrees starting at nodes of F contain all the points in $[\ell, r] \cap S$ and only them. It follows that when inserting (deleting) a range $[\ell, r]$, we can set diff(v) = diff(v) + 1 (diff(v) = diff(v) − 1) at each $v \in F$. This is because all the values vote(e) in these subtrees change by ± 1 (including leaves ℓ and r). Note that some diff(v) values may go below zero, but this does not affect correctness. To keep also the maxdiff() values correctly updated, it is enough to recompute the values in the nodes in the paths from each $v \in F$ to the root using Equation (2); other values are not affected by the insertion/deletion of the range $[\ell, r]$. The overall number of nodes that need updating is $O(\log |S|)$.

Finally, let us consider case (iii). Counters diff(v) are affected by tree rotations, but in case a tree rotation involving e.g. subtrees $v.left$, $v.right.left$ and $v.right.right$ takes place, values diff(v) and diff($v.right$) can be "pushed" down to the roots of the affected subtrees, and hence they become zero. Then the tree rotation can be carried out, also maintaining subtree maxima easily.

Hence, each insertion/deletion takes $O(\log |S|)$ time, and maxvote(S) = maxdiff($root$) + diff($root$) is readily available in the root node.

3.2 Distance $d^{\mathbf{t},\kappa}_{\mathrm{MAD}}$

Let us start with $\kappa = 0$. As proved in [12], the optimal transposition for distance $d^{\mathrm{t}}_{\mathrm{MAD}}$ is obtained as follows. Each cell $P[r', s']$, aligned to $T[r, s]$, votes for transposition $P[r', s'] - T[r, s]$. Then, the optimal transposition is the average between the minimum and maximum vote, and $d^{\mathrm{t}}_{\mathrm{MAD}}$ distance is the difference of maximum minus minimum, divided by two. An $O(|P|)$ algorithm follwed.

We need $O(m^2)$ to obtain the optimal transposition for the first angle, zero. Then, in order to address changes of text characters (because, due to angle changes, the pattern cell was aligned to a different text cell), we must be able to maintain minimum and maximum votes. Every time a text character changes, a vote disappears and a new vote appears. We can simply maintain balanced search trees with all the current votes so as to handle any insertion/deletion of votes in $O(\log(m^2)) = O(\log m)$ time, knowing the minimum and maximum at any time. If we have an integer alphabet of size σ, there are only $2\sigma + 1$ possible votes, so it is not hard to obtain $O(\log \sigma)$ complexity. Hence $d^{\mathrm{t}}_{\mathrm{MAD}}$ distance between a pattern and a text center can be computed in $O(m^3 \log m)$ or $O(m^3 \log \min(m, \sigma))$ time, for all possible rotations and transpositions.

In order to account for up to κ outliers, it was already shown in [12] that it is optimal to choose them from the pairs that vote for maximum or minimum transpositions. That is, if all the votes are sorted into a list $v_1 \ldots v_{m^2}$, then distance $d^{\mathrm{t},\kappa}_{\mathrm{MAD}}$ is the minimum among distances $d^{\mathrm{t}}_{\mathrm{MAD}}$ computed in sets $v_1 \ldots v_{m^2-\kappa}$, $v_2 \ldots v_{m^2-\kappa+1}$, and so on until $v_{\kappa+1} \ldots v_{m^2}$. Moreover, the optimum transposition of the i-th value of this list is simply the average of maximum and minimum, that is, $(v_{m^2-\kappa-1+i} + v_i)/2$.

So our algorithm for $d^{\mathrm{t},\kappa}_{\mathrm{MAD}}$ is as follows. We make our tree threaded (each node points to its predecessor and successor in the tree), so we can easily access the $\kappa + 1$ smallest and largest votes. After each change in the tree, we retraverse these $\kappa + 1$ pairs and recompute the minimum among the $v_{m^2-\kappa-1+i} - v_i$ differences. This takes $O(m^3(\kappa + \log m))$ time. In case of an integer alphabet, since there cannot be more than $O(\sigma)$ different votes, this can be done in time $O(m^3(\min(\kappa, \sigma) + \log \min(m, \sigma)))$.

3.3 Distance $d^{\mathrm{t},\kappa}_{\mathrm{SAD}}$

Let us first consider case $\kappa = 0$. This corresponds to the SAD model of [12], where it was shown that, if we collect votes $P[r', s'] - T[r, s]$, then the median vote (either one if $|P|$ is even) is the transposition that yields distance $d^{\mathrm{t}}_{\mathrm{SAD}}$. Then the actual distance can be obtained by using the formula for d_{SAD}, and an $O(|P|)$ time algorithm was immediate.

In this case we have to pay $O(m^2)$ to compute the distance for the first rotation, and then have to manage to maintain the median transposition and current distance when some text cells change their value due to small rotations.

We maintain a balanced and threaded binary search tree for the votes, plus a pointer to the median vote. Each time a vote changes because a pattern cell aligns to a new text cell, we must remove the old vote and insert the new one. When insertion and deletion occur at different halves of the sorted list of votes (that is, one is larger and the other smaller than the median), the median may move by one position. This is done in constant time since the tree is threaded.

The distance value itself can change. One change is due to the fact that one of the votes changed its value. Given a fixed transposition, it is trivial to remove the appropriate summand and introduce a new one in the formula for d_{SAD}. Another change is due to the fact that the median position can change from a value in the sorted list to the next or previous. It was shown in [12] how to modify in constant time distance $d^{\mathrm{t}}_{\mathrm{SAD}}$ in this case: If we move from transposition v_j to v_{j+1}, then all the j smallest votes increase their value by $v_{j+1} - v_j$, and the $m - j$ largest votes decrease by $v_{j+1} - v_j$. Hence distance d_{SAD} at the new transposition is the value at the old transposition plus $(2j - m)(v_{j+1} - v_j)$.

Hence, we can traverse all the rotations in time $O(m^3 \log m)$. This can be reduced to $O(m^3 \log \min(m, \sigma))$ on finite integer alphabet, by noting that there cannot be more than $O(\sigma)$ different votes, and taking some care in handling repeated values inside single tree nodes.

To compute distance $d^{\mathrm{t},\kappa}_{\mathrm{SAD}}$, we have again that the optimal values to free from matching are those voting for minimum or maximum transpositions. If we remove those values, then the median lies at positions $m - \lceil \kappa/2 \rceil \ldots m + \lceil \kappa/2 \rceil$ in the list of sorted votes, where m is the median position for the whole list.

Hence, instead of maintaining a pointer to the median, we maintain two pointers to the range of $\kappa+1$ medians that could be relevant. It is not hard to maintain left and right pointers when votes are inserted and deleted in the set. All the median values can be changed one by one, and we can choose the minimum distance among the $\kappa+1$ options. This gives us an $O(m^3(\kappa+\log m))$ time algorithm to compute $d^{\mathrm{t},\kappa}_{\mathrm{SAD}}$. On integer alphabet, this is $O(m^3(\kappa + \log\min(m,\sigma)))$, which can be turned into $O(m^3(\min(\kappa,\sigma) + \log\min(m,\sigma)))$ by standard tricks using the fact that there are $O(\sigma)$ possible median votes that have different values.

References

1. A. Amir, A. Butman, M. Crochemore, G. Landau, and M. Schaps. Two-dimensional pattern matching with rotations. In *Proc. CPM'03*, LNCS 2676, pages 17–31, 2003.
2. L. G. Brown. A survey of image registration techniques. *ACM Computing Surveys*, 24(4):325–376, 1992.
3. T. Crawford, C. Iliopoulos, and R. Raman. String matching techniques for musical similarity and melodic recognition. *Computing in Musicology*, 11:71–100, 1998.
4. M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications.* Springer-Verlag, 2nd rev. edition, 2000.
5. K. Fredriksson, V. Mäkinen, and G. Navarro. Rotation and lighting invariant template matching. Technical Report TR/DCC-2003-3, Dept. of Comp.Sci., Univ. of Chile, 2003. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/lighting.ps.gz`.
6. K. Fredriksson, G. Navarro, and E. Ukkonen. *Faster than FFT: Rotation Invariant Combinatorial Template Matching*, volume II, pages 75–112. Trans.Res.Net., 2002.
7. K. Fredriksson, G. Navarro, and E. Ukkonen. Optimal exact and fast approximate two dimensional pattern matching allowing rotations. In *Proc. CPM'02*, LNCS 2373, pages 235–248, 2002.
8. K. Fredriksson and E. Ukkonen. A rotation invariant filter for two-dimensional string matching. In *Proc. CPM'98*, LNCS 1448, pages 118–125, 1998.
9. K. Fredriksson and E. Ukkonen. Combinatorial methods for approximate image matching under translations and rotations. *Patt. Rec. Lett.*, 20(11–13):1249–1258, 1999.
10. G. Navarro K. Fredriksson and E. Ukkonen. An index for two dimensional string matching allowing rotations. In *Proc. IFIP TCS'00*, LNCS 1872, pages 59–75, 2000.
11. K. Lemström and J. Tarhio. Detecting monophonic patterns within polyphonic sources. In *Proc. RIAO'00*, pages 1261–1279, 2000.
12. V. Mäkinen, G. Navarro, and E. Ukkonen. Algorithms for transposition invariant string matching. In *Proc. STACS'03*, LNCS 2607, pages 191–202, 2003. Extended version as TR/DCC-2002-5, Dept. of Computer Science, Univ. of Chile.