1997

Faster Image Template Matching in the Sum of the Absolute Value of Differences Measure

Mikhail J. Atallah
Purdue University, mja@cs.purdue.edu

Report Number:
97-025

FASTER IMAGE TEMPLATE MATCHING IN THE SUM OF THE ABSOLUTE VALUE OF DIFFERENCES MEASURE

Mikhail J. Atallah

CSD-TR #97-025
April 1997
(Revised may 1998)

Faster Image Template Matching in the Sum of the Absolute Value of Differences Measure

Mikhail J. Atallah*
COAST Laboratory and
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
U.S.A.
mja@cs.purdue.edu

Abstract

Given an $m \times m$ image I and a smaller $n \times n$ image P, the computation of an $(m-n+1) \times (m-n+1)$ matrix C where $C(i,j)$ is of the form

$$C(i,j) = \sum_{k=0}^{n-1} \sum_{k'=0}^{n-1} f(I(i+k, j+k'), P(k,k')), 0 \leq i,j \leq m-n,$$

for some function f, is often used in template matching. Frequent choices for the function f are $f(x,y) = (x-y)^2$ and $f(x,y) = |x-y|$. For the case when $f(x,y) = (x-y)^2$, it is well known that C is computable in $O(m^2 \log n)$ time. For the case $f(x,y) = |x-y|$, on the other hand, the brute force $O((m-n+1)^2 n^2)$ time algorithm for computing C seems to be the best known. This paper gives an asymptotically faster algorithm for computing C when $f(x,y) = |x-y|$, one that runs in time $O(\min\{s, n/\sqrt{\log n}\} m^2 \log n)$ time, where s is the size of the alphabet, i.e., the number of distinct symbols that appear in I and P. This is achieved by combining two algorithms, one of which runs in $O(sm^2 \log n)$ time, the other in $O(m^2 n \sqrt{\log n})$ time. We also give a simple Monte Carlo algorithm that runs in $O(m^2 \log n)$ time and gives unbiased estimates of C.

Keywords: Image processing, template matching, algorithms, convolution

*Portions of this work were supported by sponsors of the COAST Laboratory.

1 Introduction

Template matching tries to answer one of the most basic questions about an image: Is there a certain object in that image ? If so, where ? The template is a description of that object (hence is an image itself), and is used to search the image by computing a difference measure between the template and all possible portions of the image that could match the template: If any of these produces a small difference, then it is viewed as a possible occurrence of the object. Various difference measures have different mathematical properties, and different computational properties. The measure considered in this paper is the *sum of absolute value of differences* one: We give a faster algorithm for performing the basic template matching computation for this measure. Although it is not the purpose of this paper to make any claim about the suitability of that particular measure as opposed to other measures, we do note that most textbooks on image processing mention it as a possible choice. Of course the literature contains many other measures, and interesting new ones continue to be proposed (for example, see [3] and the papers it references). For all of these measures, the speed of template matching is of crucial importance. Many approaches have been proposed for speeding up template matching computations. To mention a few: The use of parallel processing computer architectures [14], of hierarchical tree-based schemes [4, 8], of computational geometry techniques [11], of correlation techniques [7, 9, 12], and of methods that are very specific to a particular application domain, such as semiconductor chips [13]. We next state precisely the computational problem considered in this paper, and the nature of the paper's contribution.

Let I be an $m \times m$ matrix (called the *image* matrix), P be an $n \times n$

matrix (called the *pattern* matrix), $n \leq m$. The entries of both I and P come from some alphabet $A = \{a_1, \cdots, a_s\}$ where the a_i's are (possibly large) numbers, $a_1 < a_2 < \cdots < a_s$. Without loss of generality, we assume that the a_i's are positive integers; this simplifies the exposition. It is trivial to modify the paper for the case of negative a_i's (or, alternatively, one can add to everything a large enough constant to make it positive – the template matching function considered here is invariant to such a transformation).

The goal is to compute an $(m-n+1) \times (m-n+1)$ matrix C where $C(i,j)$ is of the form

$$C(i,j) = \sum_{k=0}^{n-1} \sum_{k'=0}^{n-1} f(I(i+k, j+k'), P(k,k')), 0 \leq i,j \leq m-n,$$

for some function f. Two reasonable choices for the function f that are often used in image processing [7, 9] are $f(x,y) = (x-y)^2$ and $f(x,y) = |x-y|$.

The case when $f(x,y) = (x-y)^2$ is known to be solvable in $O(m^2 \log n)$ time [7, 9]. This can easily be seen by expanding $f(x,y) = (x-y)^2$ into x^2+y^2-2xy in the definition of matrix C: The matrices corresponding to the x^2 term and (respectively) y^2 term are easy to compute in $O(m^2)$ time, and the matrix corresponding to the xy term can be computed in $O(m^2 \log n)$ by a judicious use of convolution; various elaborations and improvements on this basic idea can be found in the literature.

An $O(m^2 \log n)$ time Monte Carlo algorithm [2] has recently been given for the case when $f(x,y) = \delta_{x,y}$, where $\delta_{x,y}$ is the Kronecker symbol: $\delta_{x,y}$ is 1 if and only if $x = y$ and is 0 otherwise. The techniques used in [2] do not extend to the case of $f(x,y) = |x-y|$, and the method used in [2] has little in common with the method used in this paper. In the pattern matching community, the use of $f(x,y) = \delta_{x,y}$ is almost universal [6], but this is not so in the image processing community. In fact, most of the papers on pattern

matching not only use $f(x,y) = \delta_{x,y}$, but also focus on the problem of finding exact or almost-exact occurrences of the pattern (we refer the reader to the book [6] for an extensive bibliography on this subject).

We are not aware of any previous algorithm for the case $f(x,y) = |x-y|$ that is faster than the obvious brute force approach, which consists of taking $O(n^2)$ time for each entry of C, hence a total of $O((m-n+1)^2n^2)$ time. One of the results of this paper is an algorithm that takes $O(\min\{s, n/\sqrt{\log n}\}m^2 \log n)$ time. Such an algorithm would easily follow if we could design one algorithm that takes $O(sm^2 \log n)$ time, and another algorithm that takes $O(m^2 n\sqrt{\log n})$: If $s \leq n/\sqrt{\log n}$ then we would use the former, otherwise we would use the latter. Section 3 gives the $O(sm^2 \log n)$ time algorithm, and Section 4 gives the (more complex) $O(m^2 n\sqrt{\log n})$ time algorithm. Section 5 gives a simple Monte Carlo algorithm that runs in $O(m^2 \log n)$ time and computes unbiased estimates of C. Section 6 concludes by giving rough practical guidelines to using these algorithms (based on practical experiments with them), mentioning an open problem, and discussing the time complexities of the algorithms for the case of non-square images. Before giving the algorithms, the next section covers some preliminaries that are needed later.

2 Preliminaries

Let # be a special symbol not in A, and let A' denote $A \cup \{\#\}$. We extend the definition of the function f so that $f(x,y) = 0$ if x or y is a # symbol, otherwise $f(x,y) = |x-y|$.

Definition 1 *Let G and H be two subsets of A'. Then the $C_{G,H}$ matrix is defined by*

$$C_{G,H}(i,j) = \sum_{k=0}^{n-1}\sum_{k'=0}^{n-1} f_{G,H}(I(i+k,j+k'),P(k,k')), 0 \le i,j \le m-n,$$

where $f_{G,H}(a,b)$ equals $|a-b|$ if $a \in G-\{\#\}$ and $b \in H-\{\#\}$, equals zero otherwise.

In other words, $C_{G,H}$ counts the contribution to C of the alignments of symbols from G that occur in I, with symbols from H that occur in P. Note that, in general, $C_{G,H} \neq C_{H,G}$. Our goal is to compute the matrix $C_{A',A'}$ $(= C)$.

3 An $O(sm^2 \log n)$ Time Solution

The following is a well-known straightforward consequence of the fast algorithms for two-dimensional convolution.

Lemma 1 *Let U' be an $m \times m$ matrix and U'' be an $n \times n$ matrix, $n \le m$. Let the product $U' * U''$ be defined as the $(m-n+1) \times (m-n+1)$ matrix where*

$$(U' * U'')(i,j) = \sum_{k=0}^{n-1}\sum_{k'=0}^{n-1} U'(i+k,j+k')U''(k,k'), 0 \le i,j \le m-n.$$

Given U' and U'', the matrix $U' * U''$ can be computed in $O(m^2 \log n)$ time.

Proof. Straightforward, by using convolution (see any image processing textbook). □

Note. An implementation of the above-mentioned $O(m^2 \log n)$ time algorithm is available in MATLAB.

The next algorithm, for a special subproblem, will be used by subsequent algorithms.

Algorithm DISJOINT_ALPHABETS

Input: Image I, pattern P, two disjoint subsets of A' (called B' and B'') such that the symbols in B' are eiher all larger than those in B'', or all smaller than those in B'' (with the convention that # is smaller than anything in A).

Output: $C_{B',B''}$.

1. Create $I_=$ from I by replacing with zero every # symbol and every symbol not from B' (symbols from $B' - \{\#\}$ are left undisturbed).
2. Create $P_=$ from P by replacing with zero every # symbol and every symbol not from B'' (symbols from $B'' - \{\#\}$ are left undisturbed).
3. Create I_1 from $I_=$ by replacing with 1 every occurrence of symbols from $B' - \{\#\}$.
4. Create P_1 from $P_=$ by replacing with 1 every occurrence of symbols from $B'' - \{\#\}$.
5. Compute $X = I_= * P_1 - I_1 * P_=$.
6. If $B' > B''$ then return X. If $B' < B''$ then return $-X$.

Lemma 2 *Algorithm* DISJOINT_ALPHABETS *correctly computes* $C_{B',B''}$ *in* $O(m^2 \log n)$ *time.*

Proof. The most expensive step of the algorithm is Step 5 which, using Lemma 1, can be carried out in $O(m^2 \log n)$ time. To prove correctness, consider the three possible cases in the alignment of an $a \in B'$ in I with a $b \in B''$ in P, and how each case contributes to $C_{B',B''}$ and to X:

1. If either a or b is the symbol #, then the effect of that alignment is zero on $C_{B',B''}$ as well as on X.

   The other cases below assume that neither a nor b is #.

2. If $a > b$ (i.e., $B' > B''$) then the alignment's contribution to $C_{B',B''}$ is $|a-b| = a-b$. Its contribution is $+a$ to the corresponding entry in $I_{=} * P_1$ (that is, the entry of $I_{=} * P_1$ that corresponds to this alignment), $+b$ to the corresponding entry in $I_1 * P_{=}$. Hence its net effect on the corresponding entry in X is $a-b$, as desired.

3. If $a < b$ (i.e., $B' < B''$) then the alignment's contribution to $C_{B',B''}$ is $|a-b| = b-a$. Its contribution is $+b$ to the corresponding entry in $I_1 * P_{=}$, $+a$ to the corresponding entry in $I_{=} * P_1$. Hence its net effect on the corresponding entry in $-X$ is $b-a$, as desired.

This completes the proof. □

Lemma 3 *$C_{\{a_i\},A'} + C_{A',\{a_i\}}$ can be computed in $O(m^2 \log n)$ time.*

Proof. We can write $C_{\{a_i\},A'} + C_{A',\{a_i\}}$ as

$$C_{\{a_i\},\{a_1,\cdots,a_{i-1}\}} + C_{\{a_i\},\{a_{i+1},\cdots,a_s\}} + C_{\{a_1,\cdots,a_{i-1}\},\{a_i\}} + C_{\{a_{i+1},\cdots,a_s\},\{a_i\}},$$

because $C_{\{a_i\},\{a_i\}}$ is zero. Each of the above four terms can be computed in $O(m^2 \log n)$ time by using algorithm DISJOINT_ALPHABETS. □

The following algorithm gives the main result of this section (it makes crucial use of algorithm DISJOINT_ALPHABETS).

Algorithm ALPHABET_DEPENDENT

Input: Image I, pattern P.

Output: C $(= C_{A',A'})$.

1. Initialize all the entries of C to zero.

2. For each $a_i \in A$ in turn, compute $C_{\{a_i\},A'} + C_{A',\{a_i\}}$ and add it to C. This is done by using algorithm DISJOINT_ALPHABETS 4 times (see Lemma 3).

3. $C = C/2$.

Theorem 4 *Algorithm* ALPHABET_DEPENDENT *correctly computes* $C_{A',A'}$ *in* $O(sm^2 \log n)$ *time.*

Proof. The time complexity claim follows from the fact that it uses algorithm DISJOINT_ALPHABETS $4s$ times, each at a cost of $O(m^2 \log n)$ time. Correctness follows from the fact that

$$C_{A',A'} = 2^{-1} \cdot \sum_{i=1}^{s} (C_{\{a_i\},A'} + C_{A',\{a_i\}}),$$

where we divided the summation on the right-hand side by 2 because it double-counts the effect of each alignment of an a in I with a b in P (it counts it once when $a_i = a$, and another time when $a_i = b$). □

4 An $O(m^2 n (\log n)^{0.5})$ Time Solution

We partition the problem into $(m/n)^2$ subproblems in each of which the pattern is still P but the image is $(2n) \times (2n)$. After that, we solve in $T(n) = O(n^3 \sqrt{\log n})$ time each subproblem, for a total time of $(m/n)^2 T(n)$ $= O(m^2 n (\log n)^{0.5})$. The $(m/n)^2$ subproblems are defined by the following well-known reduction [6]:

1. We pad matrix I with enough additional rows and columns of # symbols to make its dimension m a multiple of n. This causes an increase

of at most $n-1$ in m. The next two steps of the reduction assume that this has already been done, and that m is a multiple of n.

Note. This padding is for the purpose of simplifying the discussion – it is easy to drop the padding and the assumption that m is a multiple of n, but that would unnecessarily clutter the discussion that follows. We chose to make m a mupltiple of n for the sake of clarity.

2. Cover I with $(m/n)^2$ overlapping squares $I_{i,j}$ of size $(2n) \times (2n)$ each, where $I_{i,j}$ consists of the square submatrix of I of size $(2n) \times (2n)$ that begins (i.e., has its top-left corner) at position $(n \cdot i, n \cdot j)$ in I. Hence $I_{i,j}$ and $I_{i+1,j+1}$ overlap over a region of I of size $n \times n$, $I_{i,j}$ and $I_{i,j+1}$ overlap over a region of size $(2n) \times n$, $I_{i,j}$ and $I_{i+1,j}$ overlap over a region of size $n \times (2n)$.

3. The $T(n)$ time algorithm is then used on each of the $(m/n)^2$ image/pattern pairs $I_{i,j}, P$. It is easy to see that these $(m/n)^2$ answers contain a description of the desired matrix C.

The above partitioning is not only for the sake of easier exposition: It is important that the partitioning be used, and that the method outlined in the rest of this section be used *individually* on each of the $(m/n)^2$ smaller subimages.

The algorithm for computing the answer matrix (denoted by C') for a $(2n) \times (2n)$ subimage I' and the $n \times n$ pattern P consists of the following steps (where A'' now denotes the set of symbols that appear in I', or in P, plus the # symbol):

1. Compute A'' and, for every symbol $a \in A''$, compute α_a (resp., β_a), which is the number of times that symbol a occurs in I' (resp., P).

This is easy to do in $O(n^2 \log n)$ time by sorting the symbols occurring in I' (resp., P), etc.

2. Let A^+ be the subset of symbols in A'' for which $\alpha_a + \beta_a \geq n\sqrt{\log n}$, and let $A^- = A'' - A^+$. Intuitively, A^+ contains the symbols that "occur frequently" — they will be processed differently from the other symbols; the idea of processing symbols that occur frequently separately from the other symbols was first used in [1, 10] in the context of approximate pattern matching between two strings, i.e., counting the total number of matches for all possible positions of a pattern string in a text string.

   Note that

$$|A^+| \leq (|I'| + |P|)/n\sqrt{\log n} = 5n/\sqrt{\log n}.$$

   The rest of the algorithm processes the symbols from A^+ differently from symbols in A^-.

3. This step computes the contribution, to C', of alignments for which at least one of the two symbols is from A^+. That is, it computes

$$\sum_{a_i \in A^+} C'_{\{a_i\},A''} + C'_{A'',\{a_i\}}.$$

   Every symbol $a_i \in A^+$ gets processed in $O(n^2 \log n)$ time, by using algorithm DISJOINT_ALPHABETS four times (see Lemma 3) to compute $C'_{\{a_i\},A''} + C'_{A'',\{a_i\}}$. The total time for all such $a_i \in A^+$ is therefore

$$O(|A^+|n^2 \log n) = O((n/\sqrt{\log n})n^2 \log n) = O(n^3\sqrt{\log n}).$$

4. We now turn our attention to computing the contribution, to C', of alignments both of whose symbols are from A^-. We begin by partitioning a sorted version of A^- into $t = O(n/\sqrt{\log n})$ contiguous pieces

$A_1, \cdots, A_t$, such that the total number of occurrences of the symbols in the set A_i is $O(n\sqrt{\log n})$. This is done as follows: Scan the sorted version of A^- by decreasing order, putting the symbols encountered in set A_1 until the quantity $\sum_{a \in A_1} (\alpha_a + \beta_a)$ becomes $\geq n\sqrt{\log n}$, at which point A_1 is complete and the subsequently encountered symbols are put in A_2, again until $\sum_{a \in A_2} (\alpha_a + \beta_a)$ becomes $\geq n\sqrt{\log n}$, etc. Every A_i so created satisfies

$$n\sqrt{\log n} \leq \sum_{a \in A_i} (\alpha_a + \beta_a) \leq 2n\sqrt{\log n},$$

because (i) every $a \in A^-$ has $(\alpha_a + \beta_a) \leq n\sqrt{\log n}$, and (ii) we stop adding elements to set A_i as soon as $\sum_{a \in A_i} (\alpha_a + \beta_a)$ becomes $\geq n\sqrt{\log n}$. This implies that

$$t \leq (|I'| + |P|)/n\sqrt{\log n}) = 5n/\sqrt{\log n}.$$

The partitioning of A^- into $A_1, \cdots, A_t$ takes $O(n^2)$ time since we can obtain a sorted A^- from the (already available) sorted version of A''.

5. We can now write the contribution of each A_i to C' (i.e., the contribution of alignments where both symbols are from A_i) as

$$C'_{A_i,A_i} + C'_{A_i,A_{i+1} \cup \cdots \cup A_t} + C'_{A_{i+1} \cup \cdots \cup A_t,A_i} + C'_{A_i,A_1 \cup \cdots \cup A_{i-1}} + C'_{A_1 \cup \cdots \cup A_{i-1},A_i}.$$

The last four terms in the above can each be computed in $O(n^2 \log n)$ time by using algorithm DISJOINT_ALPHABETS. The first term, C'_{A_i,A_i}, is *not* computed explicitly: Instead we directly add its effect to the current C' matrix by looking at *every pair of occurrences of symbols from* A_i in I' and P, and updating C' to account for this pair of entries, as

follows. Suppose the pair of entries in question are the occurrence of symbol $a \in A_i$ at position (i', j') in I', and the occurrence of symbol $b \in A_i$ at position (i'', j'') in P. We process this pair by simply incrementing $C'(i' - i'', j' - j'')$ by an amount equal to $f(a, b)$. The total number of such a, b pairs is

$$\sum_{a,b \in A_i} \alpha_a \beta_b \;\leq\; (\sum_{a \in A_i} \alpha_a)(\sum_{b \in A_i} \beta_b) \;\leq\; (2n\sqrt{\log n})^2 \;=\; 4n^2 \log n.$$

The above must be repeated for each A_i, $1 \leq i \leq t$. Therefore the total time for this step is $O(tn^2 \log n) = O(n^3\sqrt{\log n})$ (where we used the fact that $t = O(n/\sqrt{\log n})$).

As was analyzed in each of the above five steps, the time is $O(n^3\sqrt{\log n})$.

5 An $O(Km^2 \log n)$ Time Monte Carlo Algorithm

Recall that the alphabet is $A = \{a_1, \cdots, a_s\}$, where $a_1 < a_2 < \cdots a_s$. Let $L = a_s - a_1 + 1$.

Let x be a number, $a_1 \leq x \leq a_s$. Let $R \in \{<, >, \leq, \geq\}$. We use I_{Rx} to denore the matrix obtained from I by replacing every alphabet symbol a with 1 if it satisfies the relation aRx, with 0 otherwise. P_{Rx} is similarly defined.

For example, $I_{>x}$ is obtained from I by replacing every symbol by 1 if that symbol is larger than x, by zero otherwise.

Let x be a random variable uniformly distributed over the interval $[a_1, a_s]$, and let

$$\hat{C} \;=\; L \cdot (\; I_{>x} * P_{\leq x} \;+\; I_{\leq x} * P_{>x} \;),$$

where the $*$ product is as defined in Lemma 1.

Theorem 5 $E(\hat{C}) = C$.

Proof. Consider an alignment of a particular symbol a in I with a symbol b in P:

- The corresponding alignment for $I_{>x}$ and $P_{\leq x}$ is a 1 with a 1 if and only if $b \leq x < a$. The probability of this happening when $b < a$ is equal to $(a - 1 - b + 1)/L = (a - b)/L$.
- The corresponding alignment for $I_{\leq x}$ and $P_{>x}$ is a 1 with a 1 if and only if $a \leq x < b$. The probability of this happening when $a < b$ is equal to $(b - 1 - a + 1) = (b - a)/L$.

The term that corresponds to that a-with-b alignment in the sum

$$(I_{>x} * P_{\leq x} + I_{\leq x} * P_{>x})$$

is therefore 1 with a probability equal to $|a-b|/L$. Hence the expected value of $I_{>x} * P_{\leq x} + I_{\leq x} * P_{>x}$ is $L^{-1}C$. □

The above theorem states that $\hat{C}$ is an unbiased estimate of C. This suggests an algorithm that repeatedly (say, K times) does the following:

1. Generate an x (uniformly over the interval $[a_1, a_s]$).
2. Create in $O(m^2)$ time the four matrices $I_{>x}, I_{\leq x}, P_{>x}, P_{\leq x}$.
3. Compute $\hat{C} = L \cdot (I_{>x} * P_{\leq x} + I_{\leq x} * P_{>x})$. This can be done in $O(m^2 \log n)$ time (by using Lemma 1).

C is estimated by taking the average of the estimates obtained in Step 3 for the K iterations. The time complexity is obviously $O(Km^2 \log n)$. Of

course the larger K is, the smaller the variance. A detailed analysis reveals that the variance of the estimate of the (i,j)th entry of C is

$$K^{-1}\ (\ L\cdot(\sum_{0\le k,k',l,l'\le n-1}|Int(i+k,j+k')\cap Int(i+l,j+l')|)\ -\ C(i,j)^2\ )$$

where $Int(i+k,j+k')$ denotes the interval

$$[\min\{I(i+k,j+k'),P(k,k')\},\max\{I(i+k,j+k'),P(k,k')\}],$$

$\cap$ denotes the intersection of intervals, and $|J|$ denotes the length of an interval J (i.e., if $J=[b,b']$ then $|J|=b'-b+1$). We omit the details of the derivation of the variance (they are tedious but straightforward).

6 Further Remarks

The algorithms described in this paper have been implemented, as an undergraduate course project, by Purdue student Frank Kime. The following are *rough* comparisons of these algorithms to the brute-force method; the comparisons are not definitive because he used a soft implementation of FFT, which of course suffers from large constant factors in its time complexity — the algorithms should work better with the FFT step performed by dedicated chips. Of course for large enough problem sizes the asymptotic time complexity overcomes the effect of large constant factors, but with the current software implementation "large enough" means megapixel-size images unless one judiciously uses the Monte Carlo algorithm (see below for more on this). What follows is based on $m=2n$ (i.e., fairly large templates).

- The deterministic algorithm starts getting faster than brute-force at image sizes of 6 megapixels (monochrome, 1 byte per pixel).

- Monte Carlo is best used to locate *where* the smallest entry of C occurs in case the template "almost occurs" in the image, rather than as a way to estimate all of the C matrix; the latter would require a large K (more on this below) whereas for the former a small K is enough (e.g., $K = 10$) and in that case Monte Carlo beats brute-force even for small images (as small as 32 kilopixels). That Monte Carlo is experimentally found to be a good estimator of where the template almost-occurs is not surprising: The expression for the variance (given at the end of Section 5) reveals that it is particularly small at the positions in the image where the template "almost occurs" (i.e., where $C(i, j)$ is small).

- Recall that the speed of the Monte Carlo algorithm depends on the parameter K that determines the variance of the estimate of C. One needs to use a fairly large K (around 100) for the estimates of C to have a small enough variance, and for such K the Monte Carlo algorithm starts getting faster than brute-force at image sizes of 1 megapixels.

An interesting open question is whether it is possible to achieve $O(m^2 \log n)$ time for the exact computation of C for arbitrary size alphabets.

Finally, the analysis of this paper can be repeated for rectangular (non-square) matrices, where I is $m \times m'$ and P is $n \times n'$. The resulting time complexities would then be $O(\min\{s, \sqrt{nn'/\log(nn')}\}mm' \log(nn'))$ for the deterministic algorithm, $O(mm' \log(nn'))$ for the Monte Carlo one. The details of this extension are straightforward and are omitted.

References

[1] K. Abrahamson, "Generalized String Matching," *SIAM Journal of Computing*, 16, 1987, pp. 1039–1051.

[2] M.J. Atallah, F. Chyzac, P. Dumas, "A Randomized Algorithm in Time $O(n \log n)$ for Approximate Pattern Matching," Purdue University and INRIA Rocquencourt technical reports, October 1996.

[3] "Similarity measures in computer vision," M. Boninsegna, M. Rossi, *Pattern Recognition Letters*, v 15, n 12, Dec 1994, pp. 1255–1260.

[4] R.L. Brown, "Accelerated template matching using template trees grown by condensation," *IEEE Transactions on Systems, Man and Cybernetics*, v 25, n 3, Mar 1995, pp. 523–528.

[5] K.R. Castleman, *Digital Image Processing*, Prentice Hall, 1996.

[6] M. Crochemore, W. Rytter, *Text Algorithms*, Oxford University Press, 1994.

[7] R.C. Gonzalez, R.E. Woods, *Digital Image Processing*, Addison-Wesley, Reading, MA, 1992.

[8] S. Inglis, I.H. Witten, "Compression-based template matching," *Proceedings of the 1994 Data Compression Conference*, Snowbird, UT, Mar 1994, pp. 106–115.

[9] A.K. Jain, *Fundamentals of Digital Image Processing*, Prentice Hall, Englewood Cliffs, NJ, 1989.

[10] S.R. Kosaraju, "Efficient String Matching," manuscript, 1987.

[11] L. Prasad, S. Iyengar, "High performance algorithms for object recognition problem by multiresolution template matching," *Proceedings of the 1995 IEEE 7th International Conference on Tools with Artificial Intelligence*, Herndon, VA, Nov 1995, pp. 362–365.

[12] P. Remagnino, P. Brand, R. Mohr, "Correlation techniques in adaptive template matching with uncalibrated cameras," *Proceedings of SPIE - The International Society for Optical Engineering v 2356 1995*, Society of Photo-Optical Instrumentation Engineers, Bellingham, WA, pp. 252–263.

[13] F. Saitoh, "Template searching in a semiconductor chip image using partial random search and adaptable search," *Journal of the Japan Society for Precision Engineering*, v 61, n 11, Nov 1995. pp. 1604–1608.

[14] H. Senoussi, A. Saoudi, "Quadtree algorithm for template matching on a pyramid computer," *Theoretical Computer Science*, v 136, n 2, Dec 1994, pp. 387–417.

Author Biography

Mikhail J. Atallah received a BE degree in electrical engineering from the American University, Beirut, Lebanon, in 1975, and MS and Ph.D. degrees

in electrical engineering and computer science from Johns Hopkins University, Baltimore, Maryland, in 1980 and 1982, respectively. In 1982, Dr. Atallah joined the Purdue University faculty in West Lafayette, Indiana; he is currently a professor in the computer science department. In 1985, he received an NSF Presidential Young Investigator Award from the U.S. National Science Foundation. His research interests include the design and analysis of algorithms, in particular for the application areas of computer security and computational geometry.

Dr. Atallah is a fellow of the IEEE, and serves or has served on the editorial boards of *SIAM J. on Computing, J. of Parallel and Distributed Computing, Information Processing Letters, Computational Geometry: Theory & Applications, Int. J. of Computational Geometry & Applications, Parallel Processing Letters, Methods of Logic in Computer Science.* He was Guest Editor for a Special Issue of *Algorithmica* on Computational Geometry, has served as Editor of the *Handbook of Parallel and Distributed Computing* (McGraw-Hill), as Editorial Advisor for the *Handbook of Computer Science and Engineering* (CRC Press), and serves as Editor in Chief for the *Handbook of Algorithms and Theory of Computation* (CRC Press). He has also served on many conference program committees, and state and federal panels.