| Title | Constant-Working-Space Algorithms for Image Processing |
|---|---|
| Author(s) | Asano, Tetsuo |
| Citation | Lecture Notes in Computer Science, 5416/2009: 268-283 |
| Issue Date | 2009 |
| Type | Journal Article |
| Text version | author |
| URL | http://hdl.handle.net/10119/9185 |
| Rights | This is the author-created version of Springer, Tetsuo Asano, Lecture Notes in Computer Science, 5416/2009, 2009, 268-283. The original publication is available at www.springerlink.com, http://dx.doi.org/10.1007/978-3-642-00826-9_12 |
| Description | Emerging Trends in Visual Computing LIX Fall Colloquium, ETVC 2008, Palaiseau, France, November 18-20, 2008. |

Japan Advanced Institute of Science and Technology

# Constant-Working-Space Algorithms for Image Processing

Tetsuo Asano[1]

School of Information Science, JAIST,
(Japan Advanced Institute of Science and Technology,)
Nomi, 923-1292, Japan; t-asano@jaist.ac.jp.

**Abstract.** This chapter surveys recent progress in constant-working-space algorithms for problems related to image processing. An extreme case is when an input image is given as read-only memory in which reading an array element is allowed but writing any value at any array element is prohibited, and also the number of working storage cells available for algorithms is at most some constant. This chapter shows how a number of important fundamental problems can be solved in such a highly constrained situation.

## 1 Introduction

Recent progress in image related technologies is remarkable. High-resolution digital camera and digital movie camera are now widely used. The image size is monotonically increasing and it is time now to restart the design of various image-processing algorithms from a view point of memory consumption. In this chapter we propose a new model of computation in this direction and survey some new attempts to reduce working space, especially, how to design constant-working space algorithms in image processing.

Another requirement of limited working storage comes from applications to built-in or embedded software in intelligent hardwares. Digital cameras and scanners are good examples of intelligent hardware. We measure the space efficiency of an algorithm by the number of working storage cells (or the amount of working space) used by the algorithm. Ultimate efficiency is achieved when only constant number of variables are used in addition to input array(s). We call such an algorithm *a constant working space algorithm*. Strictly speaking, there are two types of such algorithms. One should be rather referred to as an *in-place* algorithm. In this type of algorithms, input data are given by some constant number of arrays. Those arrays can be used as working space although there must be some upper limit on values to be stored in those arrays. Heapsort is a typical in-place algorithm. Ordinary implementation of mergesort requires a working array of the same size as the input array and thus it is not in-place. Recently there are some attempts to design in-place versions of the mergesort [17, 18, 20, 22, 25]. Quicksort does not require any array, but it is not in-place since its recursion depth depends on input size ($O(\log n)$ in average) which should be included in the working storage.

The other type of constant-working-space algorithm satisfies that condition in a more strict sense. That is, it should not use any working space of size dependent on input size and an array storing input data is given as read-only memory so that no value in the array can be changed. Constant-working-space algorithms for image processing in [1, 3, 4] are in-place algorithms in this sense. The algorithm for image scan with arbitrary angle [2] is a constant-working-space algorithm with input in read-only memory. The same framework has been studied in complexity theory. A typical problem is a so-called "st-connectivity" problem: given an undirected graph $G$ with $n$ vertices in read-only memory, and two vertices $s$ and $t$ in $G$, determine whether they are connected or not using only a constant number of variables of $O(\log n)$ bits. Reingold [38] succeeded for the first time in proving that the problem can be solved in polynomial time. It is a great break-through in this direction.

One of the most fundamental problems in image processing is *Connected Components Labeling* in which we are requested to label each pixel in a given binary image by a component number (index) to which the pixel belongs [5, 13, 29, 30, 36, 39, 41]. Its simplified version is *Connected Components Counting,* which just requires to count the number of connected components in an input binary image. These problems are well studied in the image processing literature (see [27, 40] for survey). Figure 1 shows an example of connected components labeling. An input binary image shown in the left contains four connected components. Each white pixel is replaced by its component number (index), which is shown in the right.

```
00000000000000000000000      00000000000000000000000
01111000000000111000110      01111000000000222000330
00111000000000011100110      00111000000000022200330
00011101111000000111000      00011101111000000222000
00000111011000000000000      00000111011000000000000
00000110001000000000000      00000110001000000000000
00000110001000000000000      00000110001000000000000
00001111111111111000000      00001111111111111000000
00000000000000011000000      00000000000000011000000
01111111111000000000000      04444444444000000000000
00111000001111100000000      00444000004444400000000
00001111111100000000000      00004444444400000000000
00000000000000000000000      00000000000000000000000
```

**Fig. 1.** Connected Components Labeling. An input binary image (left) and a result of connected components labeling.

In Connected Components Labeling each white pixel(one of value 1) is labeled by some positive integer assigned to a component to which the pixel belongs. Thus, it seems impossible to have a constant-working-space algorithm with input image in read-only memory. If we could put some information on the input array, we could label each pixel without using any extra array. More formally, there is an in-place algorithm for the problem [4] which runs in linear time.

The same algorithm can be extended to Connected Components Counting on read-only memory. An unpublished algorithm by the author finds the number of components in $O(N \log N)$ time where $N$ is the total number of pixels in the binary image.

Space-efficient algorithms have been rigorously investigated in computational geometry [8, 10, 14, 11, 15]. The read-only memory model is not so popular in the community, but the author believes that there is a considerable number of interesting problems in this direction.


## 2    New Computation Models

In this section we describe our computation model. Our model is based on a popular RAM (Random Access Machine) Model:

**(Polynomially-bounded) RAM model**

**Input size:** Input size is denoted by $n$.

**Storage Size:** The total size of working storage (or the total number of working storage cells) available must be bounded by some polynomial in $n$. Each cell or element (variable or array element) in the working space used in an algorithm has $O(\log n)$ bits.

**Recursive Call:** Implicit storage consumption required by recursive calls is also considered as a part of working space. In other words, if the maximum depth of recursive calls is $k$, then they contribute to the working space by $O(k)$.

**Memory Access:** Any memory cell can be accessed in constant time independently of $n$. Further, any basic arithmetic operation is done in constant time.

**Basic Assumption on Image Array**

**Image Size:** We assume an intensity image (without color, for simplicity). The total number of pixels in an input image is denoted by $N$. An image is given as a rectangular array of size $h \times w$, where $h$ and $w$ are the numbers of rows and columns, respectively, and hence we have $N = h \times w$. We assume that both of $h$ and $w$ are $O(\sqrt{N})$.

**Intensity Levels:** Each pixel has some nonnegative intensity level, which is assumed to be an integer between 0 and $L$. That is, an intensity level is expressed in $\log_2 L$ bits. We implicitly assume $L < N$.


Our goal here is to develop space-efficient algorithms for image processing, which require only small amount of working storage in addition to an input image array. An extreme situation is to allow only constant number of variables in algorithms. Throughout this chapter we implicitly assume that each variable in any algorithm has $O(\log N)$ bits, sometimes, exactly $\log_2 N$ bits. Such an algorithm is commonly referred to as an log-*space algorithm* in the complexity theory since the total number of bits in working space is bounded by $O(\log n)$

for an algorithm with input of size $n$. In this chapter we use the term "constant-working-space algorithm" instead of log-space algorithm since it is more intuitive for image processing.

Here is a classification of algorithms from space efficiency. Algorithms for image processing are usually allowed to use a constant number of arrays of the same size as that of an input image array. We are sometimes allowed to use only a one-dimensional array to keep some rows or columns in an image. In an extreme case we are allowed to use only constant number of variables in addition to the image array.

**Variation on Space-Efficiency**

**Linear Working Space:** A constant number of arrays of the same size as that of an input image array are allowed as working storage cells.

$O(\sqrt{N})$ **Working Space:** A constant number of one-dimensional arrays of size $O(\sqrt{N})$ are allowed as working storage cells. Each such array is as large as a row or a column of the input image.

**Constant Working Space:** Only a constant number of variables are allowed as working storage cells.

Our polynomially-bounded RAM model assumes that every memory element can be accessed in constant time. More precisely, given an index in an array, we can read and write the element of the index in constant time. Note that the content of the element is of $O(\log N)$ bits. We could also consider a model in which an image array can be accessed in a read-only manner. That is, we can read any pixel value in constant time, but we are not allowed to modify any pixel value. In this chapter we distinguish the two models by read-write and read-only models.

**Accessibility to Image Array**

**Random-Access Model:** Any pixel value can be altered to any value of $O(\log N)$ bits.

**Read-Only Model:** We can read any pixel value in constant time, but we are not allowed to alter any pixel value.

## 3 Hardware Assistance

Suppose we have a range sensor which can measure its distance from the sensor to the closest obstacle in any direction. Whenever a direction is specified, we can measure the corresponding distance in constant time. In this situation there is no need to store distance values at all possible directions in a two-dimensional array. Whenever we need a distance in some direction, we can measure it at constant time.

We have a similar situation for *object embedding*. Suppose a set of $n$ objects is given and for each pair of objects $(i, j)$ their dissimilarity is denoted by $\delta_{i,j}$. Using the dissimilarity information, we want to map objects into points in a low

dimensional space while dissimilarities are preserved as the distances between the corresponding points. We often encounter this situation in practice. Converting distance information into coordinate information is helpful for human perception because we can see how close two objects are. Because of its practical importance, this topic has been widely studied under the name of dimension reduction [7].

Multi-Dimensional Scaling (MDS) [16] is a generic name for a family of algorithms for dimensionality reduction. Although MDS is powerful, it has a serious drawback for practical use, that is, its high space complexity. The input to MDS is an $n \times n$ matrix specifying pairwise dissimilarities (or distances). Asano et. [6] proposes an approach for dimensionality reduction that avoids this high space complexity if the dissimilarity information is given by a function that can be evaluated in constant time.

A key idea in this linear-space algorithm for the distance preserving graph embedding problem is to use *clustering*. They propose a simple algorithm for finding a size-constrained clustering and show that their solution achieves the largest inter-cluster distance, or maximizes the smallest distance between objects from different clusters. That is, given a set of $n$ objects, with a function evaluating dissimilarities for pairs of objects, we embed those objects into points in a low-dimensional space so that pairwise distances are as close to their dissimilarities as possible. Then, the point set is partitioned into $O(\sqrt{n})$ disjoint subsets, called *clusters*, where each cluster contains $O(\sqrt{n})$ points (objects). Formally, using a positive integer $c$ the set is partitioned into $k$ subsets $C_1, C_2, \ldots, C_k$ in such a way that each cluster contains at most $2c$ objects except possibly one cluster which has at most $c$ elements. For $c = O(\sqrt{n})$ the number $k$ of clusters and also the largest cluster size bounded by $2c$ are both $O(\sqrt{n})$. Since, each cluster has a relatively small number of objects, and thus performing MDS with a distance matrix for each cluster separately requires only $O(n)$ working space. Using this they devise linear space algorithms for embedding all the objects in the plane.

## 4    Thresholding Intensity Images

*Thresholding* an intensity image into a binary image is one of the most important and fundamental problems in image processing and a number of algorithms have been proposed (see [27, 34, 36]). A simple algorithm is to use a histogram which expresses frequencies of intensity levels. If there are two obvious peaks, then any level separating them may work as a good threshold. Ohtsu's thresholding algorithm [34] is mathematically beautiful in the sense that the problem is defined as a combinatorial optimization problem based on discriminant analysis. That is, it computes a threshold that maximizes the inter-cluster distance between two clusters defined by the threshold. Once we have a histogram, we can find an optimal threshold in linear time in the number of intensity levels, i.e., $O(L)$ time. To implement the algorithm we need $O(L)$ working space for the histogram. What happens if we have only constant working space?

Thresholding technique is applied in a wide range of applications. One of them is fingerprint identification. Given a fingerprint image, the first step is to convert it into a binary image. Our experience tells us that a threshold is good for succeeding the fingerprint identification if ridges and valleys are of almost equal widths. Statistics on those widths can be computed using a technique called *Euclidean Distance Transform*, EDT [9, 26, 30, 35, 12]. Given a binary image, the EDT computes the Euclidean distance from each pixel $p$ to the pixel of opposite value that is closest to $p$. Two linear-time algorithms are known for the EDT [9, 21]. An algorithm by Liang et al. [28] computes a threshold at which the average width of ridges is closest to that of valleys based on binary search and Euclidean distance transform. What happens if only constant working space is available?

## 4.1    Ohtsu's Thresholding Algorithm

The Ohtsu's thresholding algorithm is described as follows. Suppose we have $L$ intensity levels. Let $h_i$ be the number of pixels with intensity level $i$. Given a threshold $T$, we have two clusters, $C_0$ and $C_1$ with $C_0$ for intensity levels $0, 1, \ldots, T-1$ and $C_1$ for $T, T+1, \ldots, L-1$. If the average intensity levels in $C_0$ and $C_1$ are $\mu_0$ and $\mu_1$, respectively, then the intercluster-distance $\delta(T)$ between $C_0$ and $C_1$ is defined by

$$\delta(T) = \frac{(\mu_0(T) - \mu_1(T))^2}{|C_0||C_1|}, \tag{1}$$

where

$$|C_0| = \sum_{i=0}^{T-1} h_i, \ |C_1| = \sum_{i=T}^{L-1} h_i,$$

$$\mu_0(T) = (\sum_{i=0}^{T-1} ih_i)/|C_0|, \text{and } \mu_1(T) = (\sum_{i=T}^{L-1} ih_i)/|C_1|.$$

**Observation 1** *[34] Let $N$ be the number of pixels in a given image and $L$ be the number of intensity levels. Given such an image, we can find in $O(N+L)$ time using $O(L)$ space in addition to the image array an optimal threshold that maximizes the intercluster distance.*

We can find an optimal threshold using binary search. A key is to decide whether a given threshold $T$ is greater than an optimal threshold. For that purpose we need to calculate the average intensity levels for two classes. First of all we can easily compute the size $|C_0|$ and $|C_1|$ since it suffices to count the number of pixels in the class $C_0$. It is done in $O(N)$ time by scanning all the pixels. Then, we evaluate the sums $\sum_{i=0}^{T-1} ih_i$ just by taking the sum of all intensity levels that is less than $T$, which is again done in $O(N)$ time. Hence, one step of the binary search is done in $O(N)$ time. Since we need $O(\log L)$ iterations, the total time required is $O(N \log L)$.

**Observation 2** *Let $N$ be the number of pixels in a given image and $L$ be the number of intensity levels. Given such an image, we can find in $O(N \log L)$ time using only constant working space in addition to the image array an optimal threshold that maximizes the intercluster distance.*

Note that the binary search described above works only if each pixel has an integral intensity level. If pixels have real values then it is impossible to find the middle value in each iteration of the binary search. What can we do in this case? Of course, randomization is one possible way, but is there any deterministic way?

One way is to use the median intensity level instead of one maximizing the intercluster distance. What is the median intensity level? It is the median of all intensity levels in a given intensity image. The assumption that each pixel has some real value as its intensity level implies that the number of intensity levels, $L$, is equal to the number of pixels, $N$. That is, we have $L = N$. How can we compute the median of $N$ such values using only constant working space? Fortunately, there is an efficient algorithm [31]:

In the literature [31] the authors first present an efficient randomized algorithm for selection, which looks quite efficient in practice.

**Observation 3** *[31] The k-th smallest from a list of n elements residing in a read-only memory can be found using $O(1)$ indexing operations and $O(n \log n)$ comparisons on the average.*

They extend the result above further into the following observation:

**Observation 4** *[31] The k-th smallest from a list of n elements residing in a read-only memory can be found using $O(1)$ indices and $O(n \log \log n)$ comparisons on the average, if all the permutations of the given input are equally likely.*

A key observation for their deterministic algorithm for selection is the following:

**Observation 5** *[31] The k-th smallest from a list of n elements residing in a read-only memory can be found using $O(in^{1+1/i})$ comparisons and $O(i)$ indices in the worst case, where $i$ is any fixed value (parameter) such that $1 < i \leq \sqrt{\log n / \log \log n}$.*

Using this observation, they obtain the following result.

**Observation 6** *[31] Given a read-only array of size n and a positive small constant $\epsilon$, there is an algorithm which finds the median of the n elements in $O(n^{1+\epsilon})$ time using $O(1/\epsilon)$ working space.*

An important idea behind their algorithm is a *controlled recursion*. Recursion is, of course, one of the most powerful algorithmic techniques, but recursion on problems of size $n$ may have $O(\log n)$ depth in the worst case, which requires that much working space. In our case we must be careful so that the recursion depth does not exceed some constant. In the above observation, the parameter $i$ specifies the largest possible depth. The larger the value $i$ is the faster the algorithm becomes, but at the same time it requires larger working space.

A number of related results are known [17–20, 22–25, 31–33, 37].

# 5 In-Place Algorithm for Rotated Image Restoration

Demand for high-performance scanners is growing as we are moving toward paper-less society. There are a number of problems to be resolved in the current scanner technology. One of such problems is correction of rotated documents. An efficient in-place algorithm is presented in [1, 2], which assumes that rotation angle is detected by some hardware.

Once the rotation angle is obtained, it is easy to rotate the image if sufficient working space is provided. Suppose input intensity values are stored in a two-dimensional array $a[.,.]$ and another array $b[.,.]$ of the same size is available. Then, at each lattice point (pixel) in the rotated coordinate system we compute an intensity value using appropriate interpolation (linear or cubic) using intensity values around the lattice point (pixel) in the input array and then store the computed interpolation value at the corresponding element in the array $b[\ ]$. More precisely, for each pixel $(x, y)$ in the rotated image we use $2d \times 2d$ pixels around the corresponding point in the input image for interpolation. The set of pixels is denoted by $N_d(x, y)$. The window $N_d(x, y)$ for interpolation is defined by

$$N_d(x, y) = \{(x', y') \in G_{wh}^{\#} \mid$$
$$\lfloor x \rfloor - d + 1 \le x' \le \lfloor x \rfloor + d,$$
$$\lfloor y \rfloor - d + 1 \le y' \le \lfloor y \rfloor + d\}.$$

Finally, we output intensity values stored in the array $b[\ ]$.

This method, however, requires too much working storage. Is it possible to implement the interpolations without using any extra working storage?

A space-efficient algorithm [1, 2] is presented for correcting rotation of a document without using any extra working storage. A simple way of doing this is to compute an interpolation value at each pixel in the rotated coordinate system and store the computed value somewhere in the input array $a[\ ]$ near the point in the original coordinate system. Once we store an interpolation value at some element of the array, the original intensity value at the element is lost and it is replaced by the interpolation value. Thus, if the neighborhood of the pixel in the rotated coordinate system includes interpolated values then the interpolation at that point is not correct or reliable. One of the key observations is that there is an easily-computable condition to determine whether interpolation at a given pixel is reliable or not, that is, whether any interpolated value is included in the neighborhood or not. Using the condition, we first classify pixels in the rotated coordinate system into reliable and unreliable ones. In the first phase we compute interpolation at each unreliable pixel and keep the interpolation value in a queue, which consists of array elements outside the rotated subimage. Then, in the second phase we compute interpolation at every pixel $(x, y)$ in the rotated coordinate system and store the computed value at the $(x, y)$-element in the array. Finally, in the third phase for each unreliable pixel $(x, y)$ we move its interpolation value stored in the queue back to the $(x, y)$-element in the array.

## 5.1 Input image and rotated subimages

The input image $G$ consists of $h \times w$ pixels. Each pixel $(x, y)$ is associated with an intensity level. The set of all those pixels (or lattice points in the $xy$-coordinate system) is denoted by $G^{\#}_{wh}$ and its bounding rectangle by $G_{wh}$.

The rotated subimage $R$ consists of $H \times W$ pixels, which form a set $R^{\#}_{WH}$ of pixels (or lattice points in the $XY$-coordinate system). An intensity level at each pixel $(X, Y)$ is calculated by interpolation using intensity levels in the neighborhood.

We have two coordinate systems, one for the original input and the other for the rotated document. They are denoted by $xy$ and $XY$, respectively. The rectangle corresponding to the input image is denoted by $G_{wh}$ where $w$ and $h$ are horizontal and vertical dimensions of the rectangle, respectively. By $G^{\#}_{wh}$ we denote a set of lattice points in the rectangle. More precisely, they are defined by

$$G_{wh} = \{(x, y) \mid 0 \le x < w \text{ and } 0 \le y < h\}, \text{ and}$$
$$G^{\#}_{wh} = \{(x, y) \mid x = 0, 1, \ldots, w - 1, \text{ and } y = 0, 1, \ldots, h - 1\}.$$

We implicitly assume that intensity values are stored at array elements corresponding to lattice points in the set $G^{\#}_{wh}$. Now, we have another rectangle, which is a bounding box of a rotated image. We denote it by $R_{WH}$, where $W$ and $H$ are width and height of the rectangle, respectively. The set of lattice points in $R_{WH}$ is denoted by $R^{\#}_{WH}$. More precise definitions are given by

$$R_{WH} = \{(X, Y) \mid 0 \le X < W \text{ and } 0 \le Y < H\}, \text{ and}$$
$$R^{\#}_{WH} = \{(X, Y) \mid X = 0, 1, \ldots, W - 1, \text{ and } Y = 0, 1, \ldots, H - 1\}.$$

Figure 2 illustrates two rectangles, $G_{wh}$ as $ABCD$ and $R_{WH}$ as $PQRS$.

## 5.2 Output image and location function

An interpolation value calculated at a pixel $(X, Y) \in R^{\#}_{WH}$ in a rotated subimage is stored (or overwritten) at some pixel $s(X, Y) \in G^{\#}_{wh}$ in the original input image. The function $s(\ )$ determining the location is referred to as a *location function*. A simple function is $s(X, Y) = (X, Y)$ which maps a pixel $(X, Y)$ in $R^{\#}_{WH}$ to a pixel $(X, Y)$ in $G^{\#}_{wh}$. We may use different location functions, but this simple function seems best for row-major and column-major raster scans. So, we implicitly fix the function.

## 5.3 Correspondence between two coordinate systems

Let $(x_0, y_0)$ be the $xy$-coordinates of the lower left corner of a rotated document (more exactly, the lower left corner of the bounding box of the rotated subimage). Now, a pixel $(X, Y)$ in $R^{\#}_{WH}$ is a point $(x, y)$ in the rectangle $G_{wh}$ with

$$x = x_0 + X \cos\theta - Y \sin\theta, \text{ and}$$
$$y = y_0 + X \sin\theta + Y \cos\theta.$$

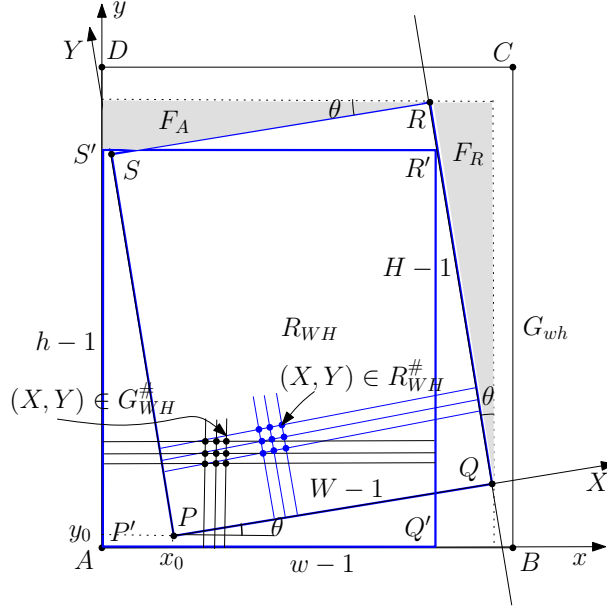The corresponding point $(x, y)$ defined above is denoted by $p(X, Y)$.

**Fig. 2.** Two rectangles $G_{wh}$ and $R_{WH}$.

### 5.4 Basic interpolation algorithm

The following is a basic algorithm for interpolation with a scan order $\sigma$ and location function $s(\ )$.

**Basic interpolation algorithm**
**Phase 1:Scan rotated subimage**
for each $(X, Y) \in R_{WH}^{\#}$ in a scan order $\sigma$ do
    · Calculate a location $p(X, Y) = (x, y)$ in the $xy$-coordinate system.
    · Execute interpolation at $(x, y)$ using intensity levels in the window $N_d(x, y)$.
    · Store the interpolation value at a pixel $s(X, Y) \in G_{wh}^{\#}$.
**Phase 2: Clear the margin**
for each $(x, y) \in G_{wh}^{\#}$ do
    if no interpolation value is stored at $(x, y)$
        then the intensity level at $(x, y)$ is set to *white*.

    The basic algorithm above is simple and efficient. Unfortunately, it may lead to incorrect interpolations since when we calculate an interpolation value at some pixel we may reuse intensity levels resulting from past interpolations. A more precise description follows:
    Suppose we scan pixels in a rotated subimage $R_{WH}^{\#}$ and an interpolation value computed at each point $(X, Y)$ is stored at the pixel specified by the location function $s(X, Y)$. We say interpolation at $(X, Y) \in R_{WH}^{\#}$ is *reliable* if

and only if none of the pixels in the window $N_d(x, y)$ keeps interpolation value. Otherwise, the interpolation is *unreliable*. "Unreliable" does not mean that the interpolation value at the point is incorrect. Consider an image with only one intensity level. Then, interpolation does not cause any change in the intensity value anywhere. Otherwise, if we use interpolated values for interpolation, the computed value may be different from the true interpolation value. We use the terminology *"unreliable"* in this sense. A pixel $(X, Y)$ is called *reliable* if interpolation at $(X, Y)$ is reliable and *unreliable* otherwise.

## 5.5  Lazy Interpolation and Local Reliability Test

An idea to avoid such incorrect interpolation is to find all unreliable pixels and keep their interpolation values somewhere in a region which is not used for output image. In the following algorithm we use a queue to keep such interpolation values.

**[Lazy Interpolation]**
$Q$: a queue to keep interpolation values at unreliable pixels.
for each pixel $(X, Y) \in R^{\#}_{WH}$ in a scan order $\sigma$ do
    if $(X, Y)$ is unreliable
        then push the interpolation value at $(X, Y)$ into the queue $Q$.
for each pixel $(X, Y) \in R^{\#}_{WH}$ in the order $\sigma$ do
    if $(X, Y)$ is unreliable
        then pop a value up from the queue $Q$ and
            store the value at the pixel $s(X, Y)$.
        else calculate the interpolation value at $(X, Y)$
            and store the value at the pixel $s(X, Y) \in G^{\#}_{wh}$.

Here are two problems. One is how to implement the queue. The other is how to check unreliability of a pixel. It should be remarked that both of them must be done without using any extra working storage.

Suppose we scan pixels in a rotated subimage $R^{\#}_{WH}$ according to a scan order $\sigma$ and interpolation using a window of size $d$ around each point $(X, Y)$ is calculated and stored at an array element $s(X, Y)$ specified by the location function. Now we can define another sequence $\tau$ to determine an order of all pixels in $G^{\#}_{wh}$ to receive interpolated values. That is, the function $\tau$ is defined so that

$$\tau(s(X, Y)) = \sigma(X, Y)$$

holds for any $(X, Y) \in R^{\#}_{WH}$. Since a rotated subimage is smaller than the original image, some pixels in the original image are not used for output image. That is, there are pixels $(x, y)$ in $G^{\#}_{wh}$ such that there is no $(X, Y)$ in $R^{\#}_{WH}$ with $(x, y) = s(X, Y)$. For such pixels $(x, y)$ we define $\tau(x, y) = WH$. More precisely, $\tau$ is a mapping from $G^{\#}_{wh}$ to $\{0, 1, \ldots, WH\}$ such that

$\tau(x, y) = i < WH$ if $i$-th computed interpolation value is stored at $(x, y)$ in $G^{\#}_{wh}$,
$\tau(x, y) = WH$ if no interpolation value is stored at $(x, y)$.

Then, interpolation at $(X, Y)$ is reliable in the sense defined in the previous section if none of the pixels in its associated window keeps interpolated value, that is,

$$\tau(x, y) \geq \sigma(X, Y) \text{ for each } (x, y) \in N_d(p(X, Y)).$$

This condition is referred to as the reliability condition.

**Lemma 7. [Local Reliability Condition]** *Assuming a row-major raster order for $\sigma$ and $\tau$, pixel $(X, Y) \in R^{\#}_{WH}$ is unreliable if and only if*

(1) $x_0 + X \cos\theta - Y \sin\theta - d + 1 < X$ *and* $y_0 + X \sin\theta + Y \cos\theta - d < Y$, *or*
(2) $x_0 + X \cos\theta - Y \sin\theta - d + 1 < W$ *and* $y_0 + X \sin\theta + Y \cos\theta - d + 1 < Y$.

By Lemma 7, a pixel $(X, Y)$ is unreliable if and only if

(1) $Y > -\frac{1 - \cos\theta}{\sin\theta} X + \frac{x_0 - d + 1}{\sin\theta}$ and $Y > \frac{\sin\theta}{1 - \cos\theta} X + \frac{y_0 - d}{1 - \cos\theta}$ or

(2) $Y > \frac{\cos\theta}{\sin\theta} X - \frac{W - x_0 + d - 1}{\sin\theta}$ and $Y > \frac{\sin\theta}{1 - \cos\theta} X + \frac{y_0 - d + 1}{1 - \cos\theta}$.

By $L_1, L_2, L_3$ and $L_4$ we denote the four lines associated with the unreliability condition above. They are defined by

$$L_1: Y = -\frac{1 - \cos\theta}{\sin\theta} X + \frac{x_0 - d + 1}{\sin\theta}, \; L_2: Y = \frac{\sin\theta}{1 - \cos\theta} X + \frac{y_0 - d}{1 - \cos\theta},$$
$$L_3: Y = \frac{\cos\theta}{\sin\theta} X - \frac{W - x_0 + d - 1}{\sin\theta}, \; L_4: Y = \frac{\sin\theta}{1 - \cos\theta} X + \frac{y_0 - d + 1}{1 - \cos\theta}.$$

Then, a pixel $(X, Y)$ is unreliable if and only if the point $(X, Y)$ is above the two lines $L_1$ and $L_2$ or above the two lines $L_3$ and $L_4$.

Figures 3 (a) and (b) depict the four lines and the region of unreliable pixels bounded by them for each of the row-major and column-major raster orders.
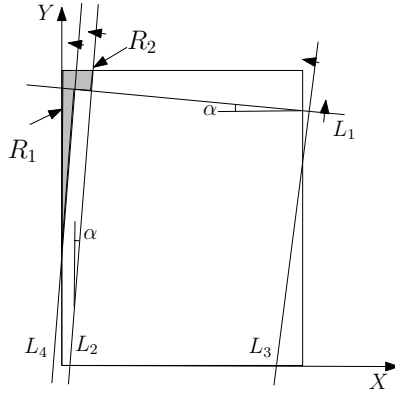


**Fig. 3.** Regions of unreliable pixels, for row-major raster order.

## 5.6 Lazy interpolation for $d = 1$

Now we know how to decide if a pixel is reliable or not each in constant time. If each pixel is reliable, we just perform interpolation. Actually, if the bottom margin $y_0$ is large enough, then the location $s(X, Y)$ keeping interpolation value is far from a point $(X, Y)$ and thus it does not affect interpolation around the point. Of course, if the window size $d$ is large, then interpolations become more frequently unreliable.

Here is an in-place algorithm for correcting rotation for the case of $d = 1$. A key to the algorithm is the local test on reliability. In the algorithm we scan $R^{\#}_{WH}$ twice. In the first scan, it checks whether $(X, Y)$ is a reliable pixel or not each in constant time. If it is not reliable, we calculate an interpolation value and store it somewhere in $G^{\#}_{wh}$ using a pixel outside the rectangle determining the output image. We call such a region a *refuge*.

**In-place algorithm for correcting rotation**

**Phase 1:** For each $(X, Y) \in R^{\#}_{WH}$ check whether a pixel $(X, Y)$ is reliable or not. If it is not, then calculate interpolation there and store the value in the refuge $F$.

**Phase 2:** For each $(X, Y) \in R^{\#}_{WH}$ check whether a pixel $(X, Y)$ is reliable or not. If it is not, then update the value at $(X, Y) \in G^{\#}_{wh}$ by the interpolation value stored in the refuge $F$.
Otherwise calculate interpolation there and store the value at $(X, Y) \in G^{\#}_{wh}$.

The algorithm above works correctly when $d = 1$. The most important is that the total area of refuge available is always greater than the total number of unreliable pixels.

**Theorem 8.** *The algorithm above correctly computes interpolations for row-major and column-major raster scans with the location function $s(X, Y) = (X, Y)$.*

The above result is based on raster scan, which scans pixels from left to right while going up from the bottom to the top of an image. However, by our experience a rotated raster scan sensitive to rotation angle is more desirable. As an extension or generalization of the raster scan we can consider a *rotated raster scan* in which pixels are enumerated along lines of a given angle. We assume that the angle of those lines is given as a slope $a$ instead of angle and we call the line $y = ax$ the *guide line* for the rotated scan.

We start from the pixel $(0, 0)$. The next pixel or point is either $(0, 1)$ or $(1, 0)$ depending on which is closer to the line $y = ax$ of slope $a$ passing through the origin $(0, 0)$. In this way we output pixels in the increasing order of the vertical distances to the line $y = ax$. Therefore, we can describe the rotated raster scan using a priority queue PQ.

**Rotated Raster Scan with Slope $a$**

PQ: priority queue keeping pixels with vertical distances to the line $y = ax$
as keys.
for $x = 0$ to $n - 1$
    Put a pixel $(x, 0)$ into PQ with key $= -ax$.
repeat$\{$
    Extract a pixel $(x, y)$ of the smallest key from PQ.
    Output the pixel $(x, y)$.
    if $(x, y) = (n - 1, n - 1)$ then exit from the loop.
    if $(y < n - 1)$ then put a pixel $(x, y + 1)$ into PQ with key $= y + 1 - ax$.
$\}$


It is easy to see that the algorithm is correct and runs in $O(N \log N)$ time
using $O(\sqrt{N})$ working space. Correctness of the algorithm is based on the ob-
servation that in each column (of the same $x$ coordinate) pixels are enumerated
from bottom to top one at a time. Thus, whenever we output a pixel $(x, y)$, we
remove the pixel from the priority queue and insert the pixel just above it, i.e.,
$(x, y + 1)$ into the priority queue if it is still in the image area $G$.

A question here is whether we can design a constant-working-space algorithm
for rotated scan in a read-only model. Fortunately, the author presented such an
algorithm [3]. Surprisingly, it also reduces the time complexity of the algorithm.

**Lemma 9.** *Given an intensity image consisting of $N$ pixels and a rational slope
$a = -q/p$, there is an algorithm for enumerating all pixels in the order deter-
mined by the slope which runs in $O(N)$ time using constant extra memory in
addition to a read-only memory for the input image.*

It is also shown in the same paper that we can remove the constraint that a given
slope must be a rational number. Once we find a rational number approximating
the given slope by that of a line passing through two pixels in the given image,
we can use that rational number as the slope.


# 6   Concluding Remarks

We have surveyed recent progress on constant-working-space algorithms espe-
cially for image processing. There is a rich source of problems in this direction in
image processing and other areas in computer science. The author is currently
working on constant-working-space algorithms for geometric problems.


## Acknowledgments

# References

1. T. Asano, S. Bitou, M. Motoki and N. Usui, "In-place algorithm for image rotation," Proc. ISAAC 2007, pp.704-715, Sendai, Dec. 2007.
2. T. Asano, S. Bitou, M. Motoki and N. Usui, "Space-efficient algorithm for image rotation," to appear in IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences.
3. T. Asano, "Constant-working-space image scan with a given angle," Proc. 24th European Workshop on Computational Geometry, pp. 99-102, March 2008.
4. T. Asano and H. Tanaka, "Constant-working space algorithm for connected components labeling," Technical Report of SIG on Computation, IEICE of Japan, 2008.
5. T. Asano and H. Tanaka, "Constant-working-space algorithm for Euclidean distance transform," Technical Report of SIG on Computation, IEICE of Japan, 2008.
6. T. Asano, P. Bose, P. Carmi, A. Maheshwari, C. Shu, M. Smid, and S. Wuhrer, "Linear-space algorithms for distance preserving embedding," Canadian Conference on Computational Geometry, pp.185-188, August, 2007.
7. H. Bast, "Dimension reduction: A powerful principle for automatically finding concepts in unstructured data," Proc. International Workshop on Self-* Properties in Complex Information Systems (SELF-STAR 2004), pp.113-116, 2004.
8. P. Bose, A. Maheshwari, P. Morin, J. Morrison, M. Smid, and J. Vahrenhold, "Space-efficient geometric divide-and-conquer algorithms," Comput. Geom. Theory Appl., 37(3), pp.209-227, 2007.
9. H. Breu, J. Gil, D. Kirkpatrick, and M. Werman, "Linear time Euclidean distance algorithms," IEEE Trans. on Pattern Analysis and Machine Intelligence, Vol.17, No.5, pp.529-533, 1995.
10. H. Brönnimann and T. M. Chan, "Space-efficient algorithms for computing the convex hull of a simple polygonal line in linear time," 34(2), pp.75-82, 2006.
11. H. Bröonnimann, J. Iacono, J. Katajainen, P. Morin, J. Morrison, and G. T. Toussaint, "Space-efficient planar convex hull algorithms," Theoret. Comput. Sci. 321, Vol.1, pp.25-40, 2004.
12. T.M. Chan, "Faster core-set constructions and data-stream algorithms in fixed dimensions," Comput. Geom., 35(1-2), pp.20-35, 2006.
13. F. Chang, C.-J. Chen, and C.-J. Lu, "A linear-time component-labeling algorithm using contour tracing technique," Comput. Vis. Image Underst. 93(2), pp.206-220, 2004.
14. T. M. Chan amd E. Y. Chen, "Towards in-place geometric algorithms and data structures," Proc. 20th Annual ACM Symposium on Computational Geometry, pp.239-246, 2004.
15. E. Y. Chen and T. M. Chan, "A space-efficient algorithm for segment intersection," in Proc. 15th Canadian Conference on Computational Geometry, pp.68-71, 2003.
16. T. Cox, and M. Cox, "Multidimensional scaling: second edition," In Chapman & Hall CRC, 2001.
17. S. Dvořak and B. Ďurian, "Stable linear time sublinear space merging," The Computer Journal, 30(4), pp.372-375, 1987.
18. S. Dvořak and B. Ďurian, "Unstable linear time O(1) space merging," The Computer Journal, 31(3), pp.279-282, 1988.
19. G. N. Frederickson, "Upper bounds for time-space trade-offs in sorting and selection," Journal of Computer and System Sciences 34, pp.19-26, 1987.

20. V. Geffert, J. Katajainen, and T. Pasanen, "Asymptotically efficient in-place merging," Theoret. Comput. Sci. 237, pp.159-181, 2000.
21. T. Hirata, "A unified linear-time algorithm for computing distance maps," Information Processing Letters, Vol.58, No.3, pp.129-133 1996.
22. B.-C. Huang and M. A. Langston, "Fast stable merging and sorting in constant extra space," The Computer Journal, 35(6), pp.643-650, 1992.
23. J. Katajainen and T. Pasanen, "Stable minimum space partitioning in linear time," BIT 32, pp.580-585, 1982.
24. J. Katajainen and T. Pasanen, "Sorting multiset stably in minimum space," Acta Informatica 31, pp.410-421, 1994.
25. J. Katajainen, T. Pasanen and J. Teuhola, "Practical in-place mergesort," Nordic J. Computing, Vol.3, pp.27-40, 1996.
26. F. Klein and O. Kübler, "Euclidean distance transformations and model guided image interpretation," Pattern Recognition Letters, 5, pp.19-20, 1987.
27. R. Klette and A. Rosenfeld: "Digital geometry: Geometric methods for digital picture analysis," Elsevier, 2004.
28. X. Liang, A. Bishunu and T. Asano, "Combinatorial approach to fingerprint binarization and minutiae extraction using Euclidean distance transform," International Journal of Pattern Recognition and Artificial Intelligence, vol. 27, no. 7, pp.1141-1158, 2007.
29. R. Malgouyresa, M. Moreb, "On the computational complexity of reachability in 2D binary images and some basic problems of 2D digital topology," Theoretical Computer Science 283, pp.67-108, 2002.
30. M. Miyazawa, P. Zeng, N. Iso, T. Hirata, "A systolic algorithm for Euclidean distance transform," Trans. on Pattern Analysis and Machine Intelligence, 1(8), pp.1-26, 2002.
31. J. I. Munro and V. Raman, "Selection from read-only memory and sorting with minimum data movement," Theoretical Computer Science 165, pp.311-323, 1996.
32. J. 1. Munro and M. S. Paterson, "Selection and sorting with limited storage," Theoretical Computer Science, 12, pp.315-323, 1980.
33. J. I. Munro, V. Raman, and J. S. Salowe, "Stable in situ sorting and minimum data movement," BIT 30, Vol.2, pp.220-234, 1990.
34. N. Ohtsu: "Discriminant and least squares threshold selection," Proc. 4th IJCPR pp.592-596, 1978.
35. D.W. Paglieroni, "Distance Transforms," Computer Vision, Graphics and Image Processing: Graphical Models and Image Processing, 54, pp.56-74, 1992.
36. A. Rosenfeld and J.L. Pfaltz: "Sequential operations in digital picture processing," J. ACM, 13, pp. 471-494, 1966.
37. V. Raman and S. Ramnath, "Improved upper bounds for time-space tradeoffs for selection with limited storage," Nordic J. of Computing, 6(2), pp.162-180, 1999.
38. Omer Reingold, "Undirected st-connectivity in log-space," Proc. ACM Symp. on Theory of Computing, pp.376-385, 2005.
39. C. Ronse and P.A. Devijver: "Connected components in binary images: The detection problem," Wiley, New York, 1984.
40. A. Rosenfeld and A.C. Kak, "Digital picture processing," Academic Press, New York, second edition, 1978.
41. K. Suzuki, I. Horiba, and N. Sugie, "Linear-time connected-component labeling based on sequential local operations," Comput. Vis. Image Underst. 89(1), pp.1-23, 2003.