Do We Need a Stack to Erase a Component in a Binary Image?

Tetsuo Asano[1]

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

Abstract. Removing noises in a given binary image is one of common operations. A generalization of the operation is to erase arbitrarily specified component by reversing pixels values in the component. This paper shows that this operation is done without using any data structure like a stack or queue, or without using any extra work space in $O(n \log n)$ time for a component consisting of n pixels. This is an in-place algorithm, but the image matrix cannot be used as work space since it has a single bit for each pixel. Whenever we flip pixel value in an objective component, the component shape also changes, which causes some difficulty. An idea for our constant work space algorithm is a conversion keeping its topology.
Keyword: constant work space, binary image, component, connectivity.

1 Introduction

Consider a binary image consisting of black and white pixels. We can define connected components of white (or black) pixels. Each connected component may correspond to some object or a noise component. Erasing a component is rather easy. Starting at any pixel in the component, we iteratively include neighboring pixels of the same color into a stack while marking them. When no more extension is possible, we pop pixels from the stack and flip its pixel value. It is done in linear time in the component size. This algorithm, however, needs mark bits and a stack (or queue), whose size can be linear in the image size in the worst case. Since we have to keep locations of those pixels in the stack, the total work space can be much larger than the given binary image itself which requires $O(m)$bits for a binary image with m pixels. We also need $O(m)$ bits for the mark bits.

In this paper we show that we can erase a given component consisting of n pixels in $O(n \log n)$ time without using any extra work space except $O(\log n)$ bits in total. The algorithm works even for a component having a number of holes in it in the same time complexity.

This is the first constant work space algorithm for erasing a component in a binary image, where erasing a component means flipping a pixel value at each pixel in the component, say from white to black. Although it is usual to assume a read-only array for input image in other constant work space algorithms, our input image is a read/write array. But it is hard to use the image matrix as work space since it has a single bit for each pixel. Furthermore, whenever we flip pixel value in an objective component, the component shape also changes, which causes some difficulty. An idea for our constant work space algorithm is a conversion keeping its topology.

There are several related results on images, such as an in-place algorithm for rotating an image by an arbitrary angle [1] and a constant work space algorithm for scanning an image with an arbitrary angle [2] and others [9]. For in-place algorithms a number of different algorithms are reported [7].

2 Preliminary

Consider a binary image G which consists of n white (value 1) and black (value 0) pixels. When two pixels of the same color are adjacent horizontally or vertically, we say they are 4-connected [10]. Moreover, if there is a pixel sequence of the same color interconnecting two pixels p and q and every two consecutive pixels in the sequence are 4-connected, then we also say that they are 4-connected. We can define 8-connectivity in a similar fashion. In the 4-connectivity we take only four among eight immediate neighbors (pixels in the 3×3-neighborhood) of a pixel. In the 8-connectivity we take all of those eight immediate neighbors as 8-connected neighbors.

A 4-connected (resp., 8-connected) component is a maximal set of pixels of the same color any two of which are 4-connected (resp., 8-connected). Hereafter, it is referred to as a component in short if there is no confusion. Following the tradition we assume that white components are defined by 4-connectivity while black ones by 8-connectivity. A binary image may contain many white components. Some of them may have holes, which are black components. Even holes may contain white components, called islands, and islands may contain islands' holes, etc.

In this paper a pixel is represented by a square. The four sides of the square are referred to as edges. An edge is called a boundary edge if it lies between two pixels of different colors. We orient each boundary edge so that a white pixel always lies to its left. Thus, external boundaries are oriented in a counter-clockwise manner while internal boundaries are clockwisely oriented.

The leftmost vertical boundary edge on a boundary is defined as a **canonical edge** of the boundary. If there are two or more such edges then we take the lowest one. The definition guarantees that each boundary, internal or external, has a unique canonical edge. It is also easily seen that the canonical edge of an external boundary is always downward (directed to the South) and that of an internal one upward (directed to the North). So, when we find a canonical edge, it is also easy to determine whether the boundary containing it is external or not. It suffices to check the direction of the canonical edge.

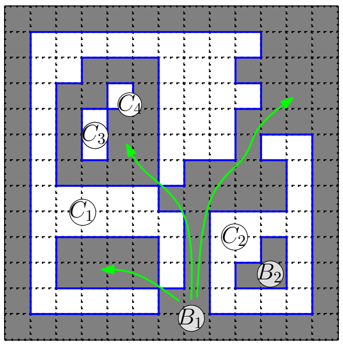

Fig. 1. (White) components C_1 and C_2, (black) holes B_1 and B_2, and (white) islands C_3 and C_4 in a binary image. There are four (white) components consisting of white pixels. Note that C_3 and C_4 are not 4-connected. Holes consist of black pixels. Since 8-connectivity is used for (black) pixels, this image contains only two black components.

3 Erasing a connected component in constant work space

One of the most fundamental problems in computer vision or pattern recognition is, given a query pixel p in a binary image, to enumerate all pixels belonging to a component to which the pixel p belongs. The problem is also considered for an intensity image. Suppose we know a local rule (or a function using local information around a pixel in question) on how to partition a given intensity image into homogeneous regions. Then, the problem of extracting a region to which a query pixel belongs is just the same as above for a binary image.

The problem is easily solved using a stack or queue. Starting from a query pixel q, we expand a search space just as wave is propagated from q. Whenever we find a pixel of the same color reachable from q which has not been checked yet, we put it into the data structure and check its neighborhood to look for unvisited pixels of the same color. This simple algorithm works quite well. In fact, it runs in time linear in the number of pixels of the component (or component size). Unfortunately, it is known that the size of the data structure is linear in the size of the component in the worst case [?]. This storage size is sometimes too expensive. We could also use depth-first algorithm with mark bits over the image. In this case the total storage size is reduced to $O(n)$ bits for an image of n pixels, but we also need storage for recursive calls of the depth-first search.

A question we address in this paper is whether we can solve the problem in a more space efficient manner. That is, can we design an algorithm for erasing a component without using any extra array? An input binary image is given using an array consisting of n bits in total. This is an ordinary bit array. We are allowed to modify their values, but it is hard to use the array to store some useful information to be used in the algorithm since we have only one bit for each pixel.

In the problem above we are requested to enumerate all pixels belonging to the same component as a query pixel. Whenever we find a pixel to be output, we output its coordinates and to prevent duplicate outputs we flip the pixel value to 0. This corresponds to erasing a component containing a query pixel. Thus, our problem is restated as follows.

Problem: Let G be a binary image. Given an arbitrary pixel q in G, erase the component containing the query pixel q. Here, by erasing a component we mean flipping a color of each pixel in the component.

How fast can we erase a connected component? This is a problem we address in this paper.

An algorithm to be presented consists of the following four steps. At the first step a query pixel is specified. Assuming it is a white pixel, we compute the canonical edge e_s of the component containing the query pixel.

Then, at the second step, we follow the external boundary of the component starting from the canonical edge. During the traverse we also try to find a canonical edge of a hole by extending a horizontal ray to the right at each downward edge. Recall that no extra array is available. When we extend the ray to find a boundary edge e, we have to determine whether the edge e is on a hole or not. The decision is done by finding the canonical edge on the boundary to which e belongs. It is on the external boundary if it is the canonical edge e_s found at the first step. Otherwise, it is on a hole. We can perform this test using only constant work space. It takes quadratic time if we just follow the boundaries, but the running time is shortened to $O(n \log n)$ time by using bidirectional search.

Once we find any hole, we traverse it while flipping white pixels on the way and finally returning to the original edge from the canonical edge by walking to the left until we touch any boundary edge.

The above flipping operations remove all the holes and a single connected component is left. At the third step we traverse the boundary again and slim the component into a tree that is one-pixel wide by erasing all possible safe pixels in the component. Here, a pixel p is safe if and only if removal of p (flipping the pixel value of p) does not separate any component within the 3×3 neighborhood around p. Safe pixel check is done in constant time.

The final step is to erase all the pixels in the thinned component. It is rather easier than others.

Now, we will describe the four step in more detail.

Step 1: Locating a given pixel

Given a query white pixel p, we want to locate it in a given binary image. In other words, we want to find a (connected) component of white pixels, which is equivalently to find a canonical edge of the component. For the purpose we first traverse the image horizontally to the left until we encounter a black pixel. The eastern edge e of the pixel is a candidate of the canonical edge. To verify it we follow the boundary starting from the edge whether we encounter any other vertical edge that is lexicographically smaller than e. Here, a vertical edge e is lexicographically smaller than another vertical edge e' if e lies to the left of e' (more precisely, the x-coordinate of e is smaller than that of e') or both of them lie on the same vertical line but e is below e' on the line. If there is no smaller edge than e then it is certainly the canonical edge of the external boundary we seek. Otherwise, starting from the pixel just to the left of e we perform the same procedure again. Figure 2 illustrates how the canonical edge is found.

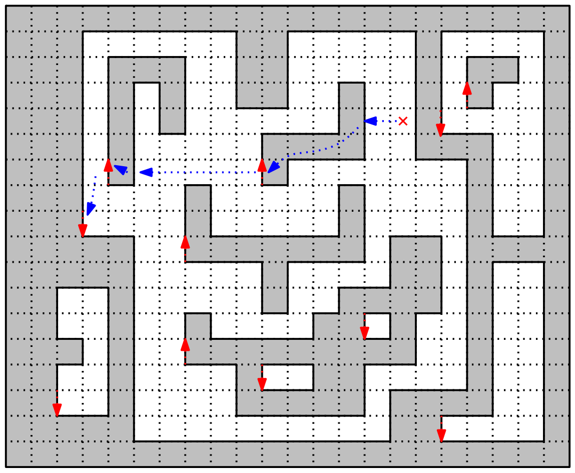

Fig. 2. Finding a canonical edge of a component containing a query point p by following boundaries at the first step of the algorithm.

Step 1: Locating a given pixel

Let p be a given white pixel and f be the western edge of the pixel p.

```
do{
    $e$ = ScanLeft($f$).
    $f$ = CanonicalEdge($e$).
} while($f$ is an internal edge)
return $f$.
ScanLeft($e$){ // move to the left until we encounter an edge between 0 and 1
    do{
        $e$ = vertical edge just to the left of $e$.
    } while($e$ is an edge between 0 and 1)
    return $e$
CanonicalEdge($f$){ // edge $f$ is canonical if no edge on the same boundary is smaller than $f$.
    $e^* = e_s = f$.
    do{
        $e$ = nextEdge($e$).
        if $e < e^*$ then $e^* = e$.
    } while($e \neq e_s$)
    return $e^*$
nextEdge($e$){
    return the next (uniquely determined) boundary edge of $e$.
}
```

Lemma 1. *The algorithm given above finds the canonical edge of the external boundary of a component which contains a query pixel p, in time linear in the size (the number of pixels) of the component.*

Proof. Since the canonical edge is defined to be the leftmost vertical boundary edge, every time when we apply the function LeftScan() we move to the left until we reach some boundary. If it happens to be an internal boundary, we follow it to find its canonical edge and then apply LeftScan() again from there. Due to the same reason we further move to the left. Thus, eventually we must reach the external boundary. Once we reach it, then it suffices to follow the boundary. Thus, the total running time is linear in the size of the component.

Step 2: Removing holes

At the first step we obtain the canonical edge e_s of the external boundary. At the next step we remove all the holes by merging them to others or to the external boundary. For the purpose we traverse the boundary again. This part is just the same as the algorithm for reporting components with their sizes in our previous paper [4]. That is, we traverse the boundaries starting from the canonical edge of the external boundary.

At each downward edge e, we walk to the right until we encounter a boundary edge f. If we find f a canonical edge after following the boundary, then we move to the hole and continue the traverse again from f. Otherwise, we go back to the edge e and continue the traverse.

At each upward edge e, we check whether it is a canonical edge of a hole by following the boundary. If it is the case, we walk to the left from e until we hit some boundary corner. There are two cases to consider. If we touch just one corner, then we erase the white pixels visited during the walk from e, which merges the hole into the boundary of the corner. If we touch two corners, we erase the white pixels visited during the walk except the last one and also erase the pixel just below the last pixel.

After erasing those pixels we still keep walking to the left until we reach a boundary edge and then start traversing the boundary again from there. Repeating this process until we come back to the starting canonical edge, the second step is done. Figure 3 illustrates behavior of the algorithm.

Step 2: Removing holes

```
$e = e_s$: // the canonical edge of the external boundary
do{
    $e$ = nextEdge($e$).
    if $e$ is downward then {
        $f$ = ScanRight($e$).
        Check whether $f$ is canonical or not.
        if $f$ is a canonical edge of a hole then $e = f$.
    } else if $e$ is upward and a canonical edge of a hole then{
```

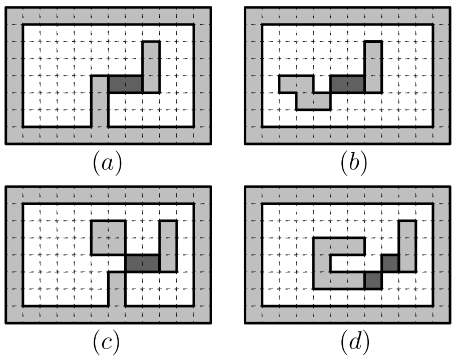
Fig. 3. Step 2: Removing holes. We traverse the boundaries. At each downward edge we check whether its right corresponding edge is canonical or not. If so, we move to the edge. At each canonical edge we walk to the left while erasing pixels until we touch some other boundary. (a) touching external boundary, (b) touching a hole, (c) touching two corners, and (d) touching two corners of the same component.

```
            traverse to the left from $e$ until we touch some boundary corner.
            Let $p_1, \ldots, p_k$ be those pixels.
            If the last pixel $p_k$ touches one corner (or one edge) then
                erase all those pixels.
            If it touches two corners then{
                erase them except the last one $p_k$ and
                erase the pixel just below $p_k$.
            }
            keep traversing to the left until it encounters a vertical boundary edge $f$.
        $e = f$.
    }
} while($e \neq e_s$)
```

Lemma 2. *The algorithm given above transforms a component with holes into one without any hole in $O(n \log n)$ time, where n is the number of pixels in the component.*

Proof. In the algorithm we traverse the boundaries starting from the canonical edge of the external boundary. At each downward edge e we perform ScanRight() until we encounter a vertical boundary edge f and check whether f is a canonical edge of an internal boundary or not. The test is done by bidirectional search. Since the test is done at most twice for each vertical edge, the total time we need is bounded by $O(n \log n)$ (the proof is similar to the one in [3]).

Once we find a canonical edge of an internal boundary, we move to the boundary and continue the traverse. Then, eventually we come back to the canonical edge f again. In the algorithm we check every upward edge whether it is a canonical edge. If we find such a upward canonical edge f then we walk to the left until we touch some boundary corner. Let $p_1, \ldots, p_k$ be a sequence of pixels visited in this walk. If the last pixel p_k touches a single corner, then we can safely erase all these pixels without touching any other boundary. After erasing them the internal boundary which used to contain f is merged into the boundary of the corner. Thus, one hole is removed. If the last pixel p_k touches two corners from dirrerent sides, then erasing p_k may cause a trouble. Refer to Figure 3(d). If the two corners belong to the same boundary, then erasing p_k creates a new hole. So, to avoid the situation, we erase $p_1, \ldots, p_{k-1}$ and the pixel p' just below p_k. The pixel p' must be a white pixel since otherwise we have touched it before reaching p_k. Since p_k touches two corners from both sides (from above and below), the row of p_k is not the bottom row. Recall that holes are treated using 8-connectivity. So, the boundary is merged into that of the lower corner without creating a new hole.

Step 3: Thinning a component

Now, we can assume that a connected component forms a simple polygon (without any hole). What we do next is to erase fat parts so that a one pixel wide skinny pattern is obtained. Again we traverse the boundary. For each downward edge e we traverse horizontally to the right until we encounter another boundary edge f. During the traverse we erase every **safe** pixel. This process is illustrated in Figure 4.

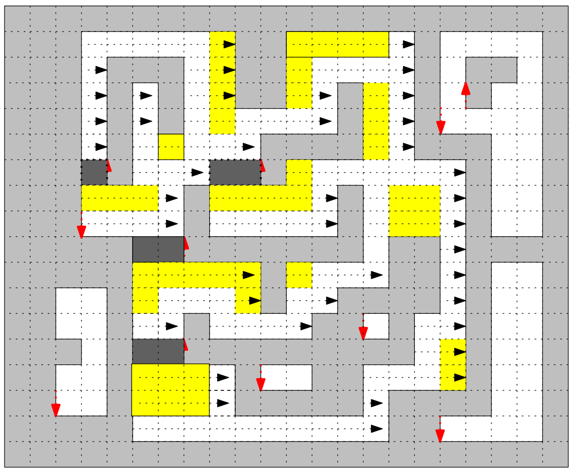

Fig. 4. Extension from the left wall while removing every safe pixel. Colored pixels have been flipped into 0.

Step 3: Thinning a component

```
$e_s$ = canonical edge of a component, and let $e = e_s$.
do{
    $e$ = nextEdge($e$).
    if $e$ is downward edge then {
        $p$ = the eastern pixel of $e$.
        while($p$ is a safe pixel of value 1){
            Erase the pixel $p$ and $p$ = the right pixel of $p$.}
        $e$ = the western edge of the pixel $p$.
        while(the pixel value of $p$ is 1){
            if($p$ is a safe pixel) then Erase the pixel $p$.
            $p$ = the eastern pixel of $p$.}
    }
} while($e \neq e_s$)
```

Lemma 3. *Implementing the algorithm above results in a slimmed component in which every remaining pixel in the component touches the external boundary.*

Proof. The algorithm scans pixels starting from each downward boundary edge until it reaches the external boundary. Thus, every pixel in the component must be examined. A pixel is erased as far as it is safe. Suppose a pixel p in the component which does not touch any boundary remains in the resulting image. Then, if we walked from the pixel p to the left then we would encounter a boundary edge, which must be downward. The downward edge may be created by a boundary edge further to the left, but there is no reason that the pixel is left without being erased, a contradiction.

Step 4: Erasing the skinny component

Applying the operation we obtain a skinny pattern such that every pixel touches the external boundary as shown in Figure 5. Now, it is not so hard to erase all the pixel in the final pattern. We traverse the boundary again. Whenever the pixel associated with the current edge is a safe pixel, we erase it, and otherwise we leave it as it is. In practice we must be more careful not to miss any pixel. For that we have a procedure to find the next pixel to be handled. The detail is found in the following pseudocode.

Step 4: Erasing the thinned component

```
$e_s$ = canonical edge of the component.
$p$ = Pixel($e$).
$e = e_s$.
do{
    do{
        $e$ = nextEdge($e$).
        } while(Pixel($e$) = $p$)
        $q = p$.
        $p$ = nextPixel($p$).
        if $p$ is nil then $p$ = Pixel($e$).
        erase the pixel $q$.
    } while($p$ has a neighbor)

nextPixel($p$){
```

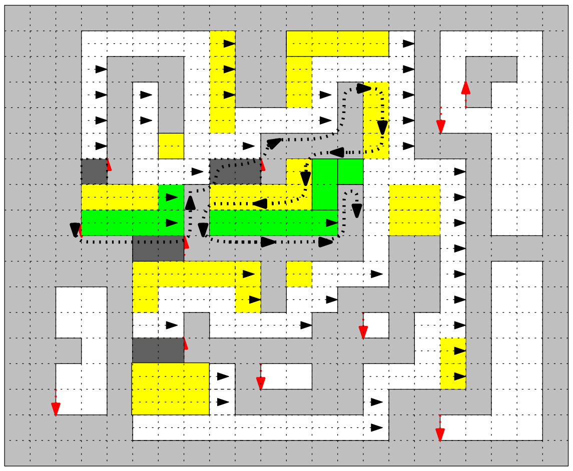

Fig. 5. Erasing the skinny component by removing safe pixels (colored pixels) in order.

```
    if only one pixel $q$ is 4-adjacent to $p$ then return $q$.
    else return nil.
}
```

Lemma 4. *The algorithm above erases all the pixels in a given component in linear time in the size of the component.*

Proof. In the algorithm we start traversing the boundary from its canonical edge, the leftmost vertical edge. Unless the pixel associated with the edge is a branching pixel (to above and to the right), the pixel is safe and it is erased. If it is a branching point, the we traverse the boundary. It never happens that we reach the starting pixel again without erasing any pixel. The reason is as follows: We know that the component forms a simple polygon consisting of white pixels such that every such pixel touches the boundary. If we define a graph representing adjacency of those white pixels in the component, it must be a tree. So, if we traverse the boundary of the polygon from some edge and come back to the same pixel again, then we must pass some leaf node in the graph. The white pixel corresponding to the leaf node is safe and thus it must have been erased in the algorithm. This means that when we start traversing the boundary, we must have erased at least one pixel before coming back to the same pixel again. Therefore, every pixel must be erased in the algorithm.

Combining the four lemmas above, we have the following theorem.

Theorem 1. *Given a binary image B with n pixels in total and a pixel p, a connected component containing p can be erased in $O(n \log n)$ time using constant work space by flipping pixel values of those pixels in the component.*

Unfortunately we cannot erase a component in linear time since we have to traverse boundaries to find canonical edges of internal boundaries.

4 Some Applications of the algorithm

4.1 Removing Small Connected Components as Noise

One of basic tasks in image processing on binary images is to remove noise. If we define a noise to be a small component, with size bounded by some small number, we can remove all such noise components by applying our algorithm using only constant work space.

In the first step we scan the entire image and finds every component. Whenever we find a canonical edge, we scan the pixels in the component to compute the size of the component. If it is small, then we apply our algorithm to erase the component. It runs in $O(n \log n)$ time for a binary image with n pixels.

Suppose T is a specified size for small component. If $O(T)$ work space is available, we can do better. We scan the entire image to find a white pixel p such that its left and lower pixels are both black. We grow a white region reachable from p using a queue of size T. If the component is small enough, we can include all the pixels in the component into the queue, and thus it is easy to erase them. Otherwise, the queue will be overflowed. Then, we conclude that the component is a large component, not a noise. Since an enqueue operation takes $O(\log T)$ time to avoid duplication, the algorithm takes $O(n \log T)$ time.

4.2 Region Segmentation

One of the most fundamental tasks for pattern recognition for color images is region segmentation, which partitions a given color image into meaningful regions. A number of algorithms have been proposed so far. Here we simplify the problem. That is, we assume that there is a simple rule for determining whether any two adjacent pixels belong to the same region or not.

In this particular situation we can solve the following problem:
Region Clipping: Given a color image G and an arbitrary pixel p, report all the pixels belonging to a region containing p using a given local rule for determining similarity of pixels.

It is rather easy to design such an algorithm if some sufficient amount of work space is available. What about if only constant work space is available? Well, we can apply our algorithm by assuming a binary image implicitly defined using the given local rule. To clip a region we don't need to convert the whole image into a binary image, but it suffices to convert the region and its adjacent areas into binary data.

Using our algorithm we can collect all regions with some useful information such as areas and average pixel values, etc., as well in $O(n \log n)$ time. If every region is small, then our algorithm runs in almost linear time.

5 Concluding Remarks

This paper has presented an in-place algorithm for erasing an arbitrarily specified component without using mark bits or extra array. The algorithm runs in

$O(n \log n)$ time when the component to be erased consists of n pixels. Since the output is written on an input array and hence the input array allows write as well as read, which is a difference from other constant work space (or log-space) algorithms assuming read-only input arrays. If we are interested only in enumerating all pixels in a component without changing pixel values, it is easier in some sense. An efficient $O(n \log n)$-time algorithm is known in our unpublished paper [4]. A basic idea for traversing component boundaries is similar to that of traversing planar subdivision in the literature [6, 8]. The algorithmic techniques developed here would be useful for other purposes in computer vision.

Acknowledgment

The author would like to express his sincere thanks to Sergey Bereg and David Kirkpatrick for their stimulating discussions and basic ideas. The part of this research was partially supported by the Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Scientific Research on Priority Areas and Scientific Research (B).

References

1. T. Asano, S. Bitou, M. Motoki and N. Usui,"In-Place Algorithm for Image Rotation," in Proc. ISAAC 2007, pp. 704-715, Sendai, Dec. 2007.
2. T. Asano, "Constant-Working-Space Image Scan with a Given Angle," Proc. 24th European Workshop on Computational Geometry (Nancy, France), pp.165-168, 2008.
3. T. Asano, S. Bereg, and D. Kirkpatrick: "Finding Nearest Larger Neighbors: A Case Study in Algorithm Design and Analysis," Lecture Notes in Computer Science, "Efficient Algorithms," editied by S. Albers, H. Alt, and S. Naeher, Springer, pp.249-260, 2009
4. T. Asano, S. Bereg, L. Buzer, and D. Kirkpatrick, "Binary Image Processing with Limited Storag," unpublished paper.
5. T. Asano and H. Tanaka, "Constant-Working Space Algorithm for Connected Components Labeling," Technical Report, Special Interest Group on Computation, IEICE of Japan, 2008.
6. P. Bose and P. Morin. An improved algorithm for subdivision traversal without extra storage. International Journal of Computational Geometry and Applications, 12(4):297-308, 2002.
7. H. Brönnimann, E. Y. Chen, and T. M. Chan, "Towards in-place geometric algorithms and data structures," Proc. 20th Annual ACM Symposium on Computational Geometry, pp. 239-246, 2004.
8. M. de Berg, M. van Kreveld, R. van Oostrum, and M. Overmars, "Simple traversal of a subdivision without extra storage, " International Journal of Geographical Information Science, 11(4):359-373, 1997.
9. 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.
10. A. Rosenfeld, "Connectivity in Digital Pictures," Journal of ACM, 17 (3), pp. 146-160, 1970.