Noname manuscript No.
(will be inserted by the editor)

In-place Algorithm for Erasing a Connected Component in a Binary Image

Tetsuo Asano

Received: date / Accepted: date

Abstract Removing noise in a given binary image is a common operation. A generalization of the operation is to erase an arbitrarily specified component by reversing pixel values in the component. This paper shows that this operation can be done without using any data structure like a stack or queue, or more exactly using only constant extra memory (consisting of a constant number of words of $O(\log n)$ bits for an image of n pixels) in $O(m \log m)$ time for a component consisting of m pixels. This is an in-place algorithm, but the image matrix cannot be used as work space since it has just one bit for each pixel. Whenever we flip a pixel value in a target component, the component shape is also deformed, which causes some difficulty. The main idea for our constant work space algorithm is to deform a component so that its connectivity is preserved.

Keywords algorithm, binary image, component, connectivity, constant work space

1 Introduction

A binary image is the simplest form of an image. A number of algorithms have been developed for binary images. One of the most fundamental problems on a binary image is to identify a connected component which contains an arbitrarily given query pixel. It is easy to design an algorithm for solving the problem if we are allowed to use mark bits and a queue. In fact, a so-called wave propagation method using a queue runs in linear time in the size of a component to be reported.

What happens if we are allowed to use only constant extra memory in addition to a given input binary image? By constant extra memory we mean one consisting of a constant number of words of $O(\log n)$ bits for an image of n pixels. This is the problem we address in this paper. Our main concern is to develop space-efficient algorithms on

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).

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

binary images. As far as the author knows, such space-efficient algorithms on binary images were first introduced in [11] by Malgouyresa and Moreb. They solved several basic problems related to 2-dimensional digital topology. One of the most interesting problems among them is that of st-connectivity. Given two white pixels s and t in a given binary image, determine using only constant extra memory whether they belong to the same connected component. They assumed that an input binary image is given as a read-only array with random access. Their algorithm runs in linear time in the total size of component(s) containing the two pixels.

Another interesting problem related to binary images is to count the number of connected components in a given binary image. It can be done in $O(n \log n)$ time using only constant extra memory, where n is the number of pixels in the image. It is not trivial at all, but we can design such an algorithm using the algorithm by de Berg et al. [7] and its improvement by Bose et al. [5] for traversing a planar subdivision without using any mark bit. It is implicitly assumed that an input binary image is given as a read-only array, but it does not seem that relaxation to an ordinary read/write array can improve the running time (if the binary image is given by a bit array of length n). Surprisingly, we can extend the above-mentioned algorithm to another $O(n \log n)$-time algorithm with constant extra work space so that it can enumerate all the pixels in a connected component containing a query pixel.

In this paper we consider a similar problem in a slightly different setting. An input image is given by an ordinary bit array so that any pixel is accessed in constant time. In particular, it is allowed to flip pixel value between 0 and 1. Note that only one bit is used for each pixel. Given an arbitrary query pixel q, we want to erase the connected component that contains q. Here, by erasing a component we mean flipping every pixel value in the component from 1 to 0. Of course, only constant extra memory can be used. We will show any component of size m can be erased in $O(m \log m)$ time by a constant work space algorithm. Note that the component may contain holes. The problem is much easier if no hole is contained.

This is the first constant work space in-place algorithm for erasing a component in a binary image. Although it is usual to assume a read-only array for an input image in other constant work space algorithms (especially, in those known as log-space algorithms), we do not. An input image is an ordinary 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 a pixel value in a target component, the component shape is also deformed, which causes some algorithmic difficulty. The main idea for our constant work space algorithm is to deform the target component while keeping its connectivity.

There are several related results, 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]. For in-place algorithms in general a number of different algorithms are reported (e.g. [6]).

This paper is organized as follows. Section 2 states the problem in a formal way after preparing some notations and terminologies. Then, in Section 3 we propose our algorithm together with analysis of its time complexity. Some possible applications are mentioned in Section 4. Finally we conclude in Section 5.

2 Preliminary

Consider a binary image G consisting of n pixels, white (value 1) or black (value 0). When two pixels of the same color are adjacent horizontally or vertically, we say they are 4-connected [12]. 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 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, on the other hand, we take all of the eight immediate neighbors as 8-connected neighbors.

A 4-connected (resp., 8-connected) component is a maximal subset of white pixels such that any two of them 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 a number of 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.

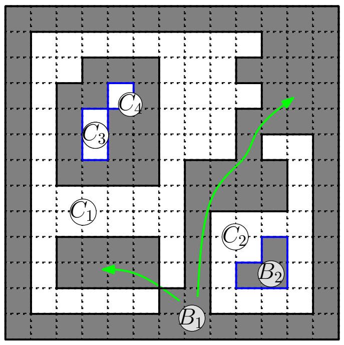


Fig. 1 An example of a binary image with white components C_1 and C_2, one hole B_2, and islands C_3 and C_4. C_1 has one hole which contains two islands C_3 and C_4. Since 8-connectivity is used for black pixels, the lower part of C_1 is not a hole but a part of the background B_1.

To simplify the presentation we assume without loss of generality that all the marginal pixels are black. We also assume that a query pixel is white. So, our problem is to erase a 4-connected white component containing a query pixel. It is not so hard to adapt our algorithm to erase a black component which is 8-connected if we use different local rules for traversing boundaries.

In this paper a pixel is represented by a square. See Figure 2. 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 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

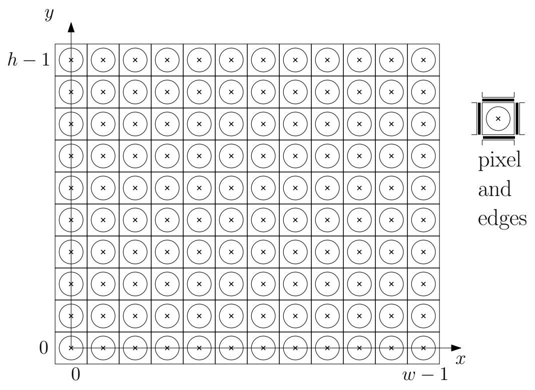


Fig. 2 Definition of pixels and edges around pixels.

downward (directed to the South). On the other hand, the canonical edge of an internal boundary is 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. See Figure 3.

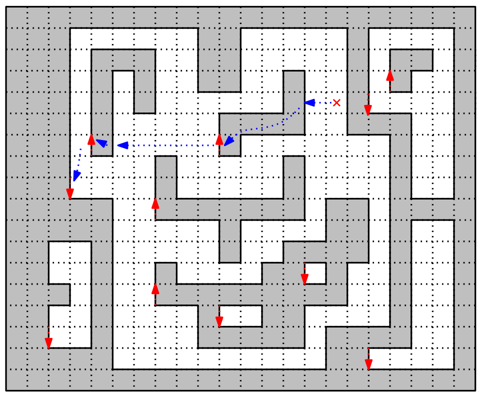


Fig. 3 Canonical edges (indicated by arrows) which are leftmost and lowest edges on internal or external boundaries. Given a query pixel, we can find the canonical edge of the component containing it by following the boundaries.

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 q in a binary image, to enumerate all pixels belonging to the component to which the pixel q belongs. We could also consider the problem in a more general setting: Suppose we are given a color image and we know a rule on how to partition the image into homogeneous regions by computing a function using local information around a pixel in constant time. More exactly, we assume some function which maps a color value s in the neighborhood of a pixel into $\{0, 1\}$. Using the function we can define a binary image. A connected component in the binary image corresponds to a homogeneous region in the original color image.

The problem is easily solved using a 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 queue 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 queue is linear in the size of the component in the worst case [4]. 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 in 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 of n-bits. This is an ordinary bit array. We are allowed to modify their values, but it is hard to use it to store some useful information for the algorithm since we have only one bit for each pixel.

In the above problem we are requested, given a query pixel q, to enumerate all pixels in the component containing q. Whenever we find a pixel to report, we output its coordinates and flip its pixel value to 0 to prevent duplicate outputs. In this way we can erase the component containing the 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 reversing a color of each pixel in the component.

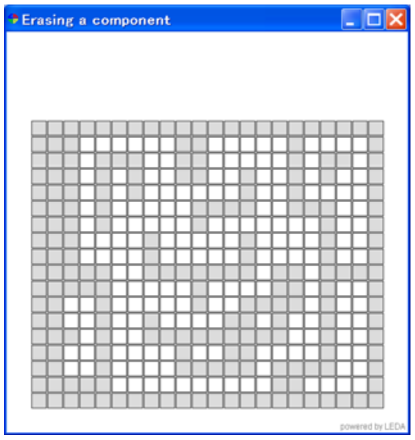
Figure 4 shows a simple experimental result. Figure 4(a) is an input binary image. When a query pixel lies in the middle white component, the resulting image looks like one in Figure 4(b). Note that the two islands of the component are left.

How fast can we erase a connected component? If our target component is singly-connected, that is, if it has no hole in it, it is rather easy. We can borrow an algorithm for thinning components to extract skeletons of a binary image. Standard algorithms for component thinning [9,10] repeatedly remove safe pixels (by flipping their pixel values from 1 to 0). 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.

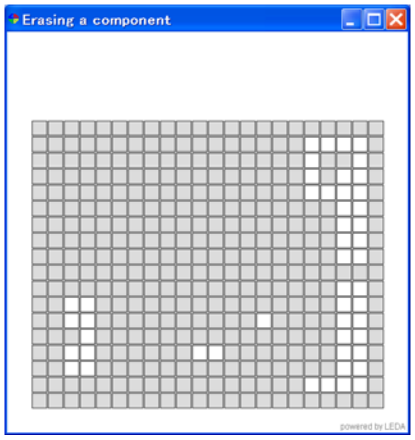
See Figure 5 for some examples of safe pixels and non-safe pixels. The notion of a safe pixel is well known in digital geometry [8], especially for component thinning algorithms to extract skeletons of a binary image [9,10]. Safe pixel check is done in constant time.

A naive algorithm repeatedly removes safe pixels until no pixel is left or no pixel is safe. Unfortunately, the algorithm above gets stuck when it is applied to a component with hole(s).

This flaw can be fixed as follows: The above naive algorithm gets stuck when none of the pixels along the external boundary is safe. However, this is not a bad news but good. By the definition of a safe pixel, its removal (or flipping) breaks connectivity in the 3×3 neighborhood of the non-safe pixel. That is, if we flip the non-safe pixel, then the hole touching the pixel is exposed to either another hole or the external region. We can verify that none of the remaining pixels along the external boundary is safe:



(a)



(b)

Fig. 4 Implementation result of our demo program. Erasing the middle (white) component in an input binary image (a) specified by a query pixel results in the binary image (b). Note that islands are left unchanged.

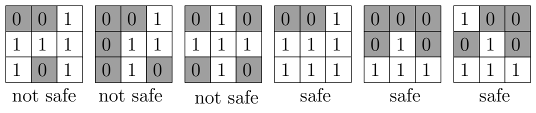


Fig. 5 Examples of safe and non-safe pixels. A central white pixel (of value 1) is safe if flipping it (into 0) does not increase the number of connected components in its 3×3 neighborhood.

whenever we flip a safe pixel along the boundary, we record the edge associated with the pixel on the updated external boundary. This is the last updated edge. If our current edge on the external boundary coincides with the last updated edge, it means there is no safe pixel on the current external boundary. Thus, we can flip the pixel to remove the hole adjacent to the pixel, and then continue the task until all the safe pixels are removed.

The algorithm runs in linear time in the number of pixels of a component if the component has no hole in it. How fast does it run when it is applied to a component with holes? Unfortunately it may be very slow. Suppose a component with m pixels has $O(m)$ small holes, as shown in Figure 6. Then, the algorithm removes the holes one by one. Whenever a hole is removed, the external boundary is modified only a little bit, but we have to follow the whole external boundary again to find the next hole to remove. Thus, it takes $O(m^2)$ time to erase the component.

We introduce a different scheme for removing holes in a component. Once we have a component without any hole, it is easy to erase the component by removing safe pixels in order.

Our algorithm consists of the following three steps (see Figure 7):

Algorithm for Erasing a Component

Step 1: Given a query pixel q, locate the external boundary (and its canonical edge) of the component which contains q in it.

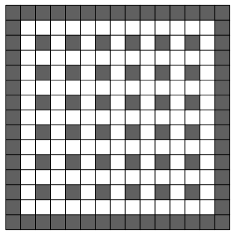


Fig. 6 A worst case for which the algorithm using safe pixel flippings takes quadratic time.

Step 2: Deform the component into a singly-connected one by merging all of holes to the external boundary.

Step 3: Erase the resulting component by removing safe pixels repeatedly.

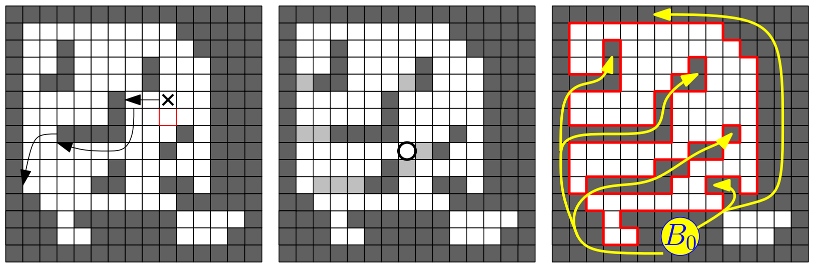


Fig. 7 Rough sketch of the algorithm. A query pixel is specified and then the canonical edge of the external boundary of the component containing the pixel is computed. Then, merge all of the holes by tunnels (gray parts in the middle). The resulting component has no hole.

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 containing q. It is equivalent to finding the canonical edge of the component. It is done using the existing algorithm [11]. To make the paper self-contained, we describe the algorithm using our terminologies.

We first traverse the image from a query white pixel 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 the boundary has no smaller edge than e, it is certainly the canonical edge of the external boundary. Otherwise, starting from the pixel just to the left of e we perform the same procedure again. The leftmost figure in Figure 7 illustrates how to find the canonical edge starting from a query pixel indicated by a cross.

Step 2: Removing holes

After finding the canonical edge e_s of the external boundary of the component to be erased, we make it simply connected by removing all the holes in the component. For the purpose we traverse the external boundary from e_s. At each downward edge e on the traverse we walk to the right within the component until we encounter a boundary edge f. Then we check by using a bidirectional search whether f is the canonical edge of an internal boundary. If so, we merge the boundary B with some other internal boundary or the external boundary. We walk back or dig a **tunnel** to the left from f until the tunnel tip touches a corner of a black pixel. Figure 8 illustrates a simple situation in which the tunnel reaches the starting edge e without touching any corner of a black pixel on the way. In the simple case we just flip all the pixels on the tunnel. Then, the hole is connected to either some other hole or the external boundary. In either case the number of holes is decreased by one.

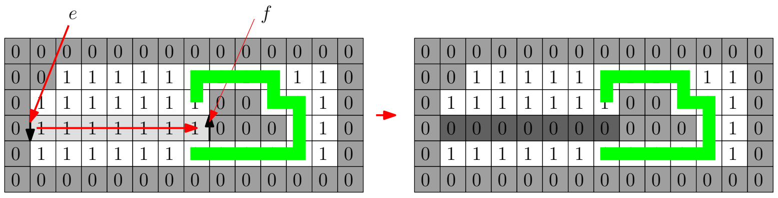


Fig. 8 Definition of a tunnel. White pixels in the tunnel are flipped to merge the hole.

Figure 9 illustrates our algorithm in more detail. Whenever we find the canonical edge of an internal boundary, we extend a tunnel to the left to merge the boundary with another internal or external boundary. Suppose the tunnel hits another boundary at a pixel p. Then, by the definition one of the three pixels at the upper left, immediately left, and the lower left must be black. If two of them are white, then just flip the tunnel part and it is connected to the other boundary at p (note that 8-connectivity is used for black pixels). We should be careful when only the middle pixel is white and the top and bottom pixels are black. One such example is shown in the middle of Figure 7. In the figure there is a small hole consisting of two black pixels arranged in 45 degree. The corresponding tunnel touches two black pixels at the same time. If we flip all the white pixels in the tunnel, then the hole is merged at two different places. If both of them belonged to the external boundary, then this tunnel would separate the interior part of the component into two. If at least one of them is an internal boundary, then there is no problem. Since it is hard to determine whether a fragment of a boundary is external or not, we should avoid using this tunnel. Our solution is to choose only one of them. Recall that the pixel p is the first white pixel on the tunnel that touches any black pixel at edge or at corner. So, the pixels above and below p are both white. Instead of flipping the pixel p, we flip the white pixel just below p. By the same reason, the lower right pixel is also white. So, this flipping operation safely merges the hole with another boundary. Figure 9 exhausts all possible cases.

The entire algorithm works as follows. We traverse the external boundary starting from the canonical edge e_s. At each downward edge e we extend a horizontal ray to the right until it hits a boundary edge f and check whether f is canonical or not. If it is canonical, then we dig a tunnel to connect it to some other boundary to remove the hole, and then continue the traverse on the renewed external boundary. If f is not canonical, then we just continue the traverse.

Lemma 1 *Let C be a connected component with m white pixels to be removed. Then, the algorithm above flips only white pixels in C to remove all the holes in it while keeping the connectivity of all remaining white pixels in C. The algorithm runs in $O(m \log m)$ time.*

Proof Since the edge f is a canonical edge of an internal boundary B, its North West and South West pixels are both white and also they are connected around B as shown in Figure 8. It is obvious that the path around B is preserved while digging the tunnel, which guarantees the connectivity of the component C, independently on how the tunnel is formed. In the algorithm we never flip pixels not contained in C. By the definition each tunnel touches only one boundary, internal or external, and it is merged into the boundary, which reduces the number of holes by one. So, if we continue traversal of the external boundary, all the holes are removed.

The time complexity of the algorithm is analyzed as follows. In the algorithm we walk to the right in the interior of the component at each downward edge on the external boundary. It is easily seen that each pixel in the component is visited only once since there is a unique downward edge for each internal pixel throughout the implementation of the algorithm. Some pixel may have no such downward edge at the beginning due to holes, but it eventually has such an edge while holes are being merged. It is also verified that any internal pixel is never visited more than once since each edge on the external boundary is visited exactly once.

Why does it take super-linear time? It is because it may happen to visit the same hole many times. At each downward edge on the external boundary we walk to the right until it encounters a boundary edge. We check whether it is the canonical edge on an internal boundary. For the test we traverse the boundary bidirectionally until we encounter any edge smaller than or equal to the initial edge in our lexicographical order. If the initial edge is the smallest, then it is the canonical edge. Otherwise, it is not. Thus, this is just the same as the problem of finding nearest larger elements in an array. It is known in [3] that bidirectional search is powerful enough to solve this problem in $O(n \log n)$ time for an array of size n. In our case we may examine many edges on a boundary. The total amount of time for the tests is $O(n_i \log n_i)$ if the boundary is of length n_i. Since the total length of internal boundaries in a component is linear in the number of pixels of the component, we have the time complexity in the lemma.

Step 3: Erasing a simply connected component

Once we have a simply-connected component without any hole after removing all the holes in the second step, it is now easy to erase the component. We just apply the algorithm based on safe pixels. Just repeat flipping safe pixels while walking along the external boundary. All the pixels in the component are flipped in linear time.

Lemma 2 *The algorithm above for erasing a simply connected component flips all the pixels in the component in linear time in the area of the component.*

Proof In the algorithm we iteratively flip safe pixels. Suppose we find a non-safe pixel p. Then, p must touch the external boundary at least twice. Then we can define an interval on the boundary associated with p by its first and last contacts with the external boundary. This structure induced by such intervals can be represented using parentheses. Since the component is simply connected, this parentheses structure forms

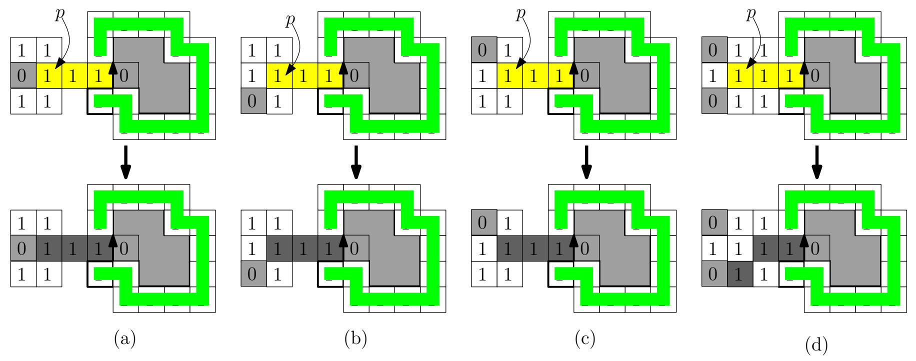


Fig. 9 Extending a tunnel to the left. (a) (b), and (c) just extend it to the left and flip all the pixels in the tunnel, and (d) stop digging the tunnel and bend it downward to connect to the lower corner.

a nested structure. Then the innermost parentheses correspond to a single pixel, which can be flipped immediately. Thus, the induction on the area of the component completes the proof.

Now the rest of the lemma is trivial from the argument so far.

The entire algorithm is decribed as follows.

Algorithm for erasing a component
Input: A binary image of n pixels given by an array of n bits, and a query pixel q of value 1 in the image.
Output: The same n-bit array which gives the binary image in which the component containing q is erased.
Algorithm

```
1. Locate the canonical edge $e_s$ of the external boundary of the component containing $q$.
2. $e = e_s$.
   repeat
      $e$ = the next edge of $e$ on the external boundary.
      if $e$ is downward then
         find the first boundary edge $f$ by walking from $e$ to the right.
         Check whether $f$ is the canonical edge of an internal boundary.
         if it is true then extend a tunnel to the left from $f$ following the rules shown in
         Figure 9 and erase the tunnel part.
   until $e = e_s$.
3. // now the component has no hole
   $e = e_s$.
   repeat
      $e$ = the next edge of $e$ on the external boundary.
      $p$ = the pixel associated with $e$.
      if $p$ is safe then flip the pixel value of $p$.
   until $e = e_s$.
   flip the last pixel.
```

Combining the results so far we have the following theorem.

Theorem 1 *Given a connected component with m pixels in a binary image, the component can be erased in $O(m \log m)$ time using constant work space.*

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 constant work space algorithm.

In the first step we scan the entire image and finds every component together with their sizes. This is not a trivial task, but we can modify our algorithm for removing holes. To enumerate all the components we do not remove holes by digging tunnels. At each downward edge on an external boundary we walk to the right until it encounters any boundary edge. If it is a canonical edge, then we continue the traverse to find holes further to its right. In this way we can enumerate all the holes in a component. The sizes of the holes are computed while walking along the boundaries. Thus, we can compute the size (number of pixels) of the component. This algorithm runs in $O(n \log n)$ time, where n is the number of pixels of a given binary image.

Now we know how to enumerate all the components together with their sizes. If we find a component of size smaller than a given threshold, we can erase it by applying our algorithm. It takes $O(m \log m)$ time for a component of size m in the worst case in which a number of small holes are included in the component. Such a worst case rarely happens in practice. First of all, we are interested in erasing small components, which cannot include many holes.

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: Suppose we are given a local rule for determining similarity of pixels. Given an image and an arbitrary pixel q in the image, report a connected region consisting of pixels similar to q in the local rule.

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 in a binary image without using mark bits or extra array. The algorithm runs in $O(m \log m)$ time when the component to be erased consists of m pixels. The output is written on an input array, and hence the input array allows writing as well as reading. This is a difference from other constant work space (or log-space) algorithms which assume 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. The algorithmic techniques developed here would be useful for other purposes in computer vision.

No lower bound on the running time is known. The author is now trying to establish the $\Omega(m \log m)$ lower bound on this problem.

Acknowledgements The author would like to express his sincere thanks to Sergey Bereg and David Kirkpatrick for their stimulating discussions and basic ideas.

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 and H. Tanaka, "Constant-Working Space Algorithm for Connected Components Labeling," Technical Report, Special Interest Group on Computation, IEICE of Japan, 2008.
5. 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.
6. 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.
7. 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.
8. R. Klette and A. Rosenfeld, "Digital geometry: Geometric Methods for Digital Picture Analysis," Elsevier, 2004.
9. P. Kwok, "A thinning algorithm by contour generation ," Comm. of ACM, 31, 11, pp.1314 - 1324 , 1988.
10. L. Lam, S.-W. Lee, and C.Y. Suen, "Thinning Methodologies – A Comprehensive Survey," IEEE Trans. on Pattern and Machine Intelligence, 14, 9, pp.869-885, 1992.
11. 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.
12. A. Rosenfeld, "Connectivity in Digital Pictures," Journal of ACM, 17 (3), pp. 146-160, 1970.