Small Work Space Algorithms for Some Basic Problems on Binary Images

Tetsuo Asano, Sergey Bereg, Lilian Buzer

Small Work Space Algorithms for Some Basic Problems on Binary Images

Tetsuo Asano[1], Sergey Bereg[2], and Lilian Buzer[3]

[1] School of Information Science, JAIST, Japan
[2] Department of Computer Science, University of Texas at Dallas, USA.,
[3] Université Paris-Est, LABINFO-IGM, UMR CNRS 8049, and Department of Computer Science, ESIEE, France.,

Abstract. This paper presents space-efficient algorithms for some basic tasks (or problems) on a binary image of n pixels, assuming that an input binary image is stored in a read-only array with random-access. Although efficient algorithms are available for those tasks if $O(n)$ work space (of $O(n \log n)$ bits) is available, we aim to propose efficient algorithms using only limited work space, i.e., $O(1)$ or $O(\sqrt{n})$ space. Tasks to be considered are (1) CCC to count the number of connected components, (2) MERR to report the minimum enclosing rectangle of every connected component, and (3) LCCR to report a largest connected component. We show that we can solve CCC, MERR, and LCCR in $O(n \log n)$ time each using only $O(1)$ space. If we can use $O(\sqrt{n})$ work space, we can solve them in $O(n)$, $O(n)$, and $O(n + m \log m)$ time, respectively, where m is the number of pixels in the largest connected component.

1 Introduction

Demand for embedded software is growing toward intelligent hardware such as scanners, digital cameras, etc. One of the most important aspects and also differences between ordinary software in computers and embedded software come from constraints on the size of local memory. For example, to design an intelligent scanner a number of algorithms should be embedded in the scanners. In most of those cases, the size of pictures is increasing while the amount of work space available for such software is severely limited. In the sense algorithms which require a restricted amount of work space and run reasonably fast are desired.

In this paper we propose several space-efficient algorithms designed for some fundamental image processing tasks. All of the tasks that we consider have straightforward solutions if sufficient memory (typically, of size proportional to the size of the image) is available (see, for example, [5, 8]). Solving the same tasks with restricted memory, without severely compromising the running time, is more of a challenge.

1.1 Computational model with limited work space

We measure the space efficiency of algorithms by the amount of work space used. Such space typically takes the form of pointers and counters (whose number of

bits is at most the logarithm of the image size). Our objective is to design efficient algorithms that use only $O(1)$ or $O(\sqrt{n})$ such pointers and counters.

Throughout the paper we assume that an input binary image consists of n pixels in $O(\sqrt{n})$ rows and columns, and it is stored in a read-only array in a random-access manner.

1.2 Three basic tasks considered in this paper

Three basic tasks for an input binary image to be considered in this paper are:

CCC (Connected Components Counting) Count the number of connected components.

MERR (Minimum Enclosing Rectangles Reporting) Report the minimum enclosing (axis-parallel) rectangle of every connected component.

LCCR (Largest Connected Component Reporting) Report all pixels of a largest connected component.

We have to output a number of information except the task CCC. MERR requires us to output rectangles, and LCCR to output pixels in the largest component. It is expensive to use an array for the output. So, we just output them without using any array to store them.

1.3 Important concepts and terminologies

There are several important concepts and terminologies to design space efficient algorithms for those tasks.

Lexicographical order: We assume a lexicographical order among all pixels. A pixel at (x, y) precedes one at (x', y') if $y < y'$ or $y = y'$ and $x < x'$. The same order is defined for all vertical edges.

Canonical Edge: Connected components are described by external and internal boundaries. The canonical edge of a boundary is the lexicographically smallest vertical edge on the boundary, which is uniquely defined.

Run: A maximal sequence of foreground (white) pixels in a row is called a run.

Run Adjacency Graph: Run adjacency graph represents incidence relations among runs in two consecutive rows. The graph plays an important role in the algorithms to be presented in the paper. It contains $O(\sqrt{n})$ vertices and edges since there are only $O(\sqrt{n})$ columns by the assumption.

Provisional Label: We put labels to runs to maintain information on connected components. Once we know two runs belong to the same component, their labels must be merged. In this way a number of labels are created and disappear. Those temporary labels are called provisional labels. We use some labels many times in our labeling process. This is a basic idea to save the total number of labels.

1.4 Results obtained

We present space efficient algorithms for the tasks listed above using $O(1)$ or $O(\sqrt{n})$ work space. For the tasks CCC (Connected Components Counting) and MERR (Minimum Enclosing Rectangles Reporting) our algorithms run in $O(n \log n)$ time with $O(1)$ work space or $O(n)$ time with $O(\sqrt{n})$ space. For the task LCCR (Largest Connected Components Reporting) our algorithm runs in $O(n \log n)$ time using $O(1)$ work space. If we can use $O(\sqrt{n})$ space it is reduced to $O(n + m \log m)$ time where m is the size of the largest component.

2 Known results for $O(1)$-space algorithms and their extensions

2.1 Basic algorithm

We first give an intuitive explanation of an algorithm for counting the number of connected components in a given binary image.

Basic algorithm for counting the number of connected components

$c = 0$. // counter for the number of connected components
for each pixel p in the lexicographic order (raster order)
  if it is a unique pixel in a component **then** increment the counter c.
Report the counter value c as the number of connected components.

We assume a **lexicographical order** among all pixels. A pixel at (x, y) precedes one at (x', y') if $y < y'$ or $y = y'$ and $x < x'$. The same order is defined for vertical edges.

The algorithm correctly counts the number of connected components if a unique pixel is defined for each connected component and we can determine in some reasonable time whether a given pixel is the unique one. For the purpose we introduce a notion of **canonical edge** instead of a unique pixel.

2.2 Canonical edge

A connected region in a binary image refers to a maximal set of foreground pixels in which any two of them are connected by a 4-connected path of foreground pixels. It may contain holes, islands, and further holes of islands. Thus, the boundary of a connected component is defined by a single external boundary and possibly more than one internal boundary. In this paper we assume a small square for each pixel, which has four sides. Two adjacent pixels share a side. A side between two pixels of different values is called an edge. Then, a boundary is a sequence of such edges, horizontal or vertical. Any boundary must have a unique vertical edge that is lowest (and leftmost if there are ties). This uniquely defined edge is referred to as the **canonical edge** of the boundary, as was defined in the literature [4, 6]. Fig. 1 shows how canonical edges are defined using a simple binary image when each boundary is oriented so that foreground pixels always lie to the right.

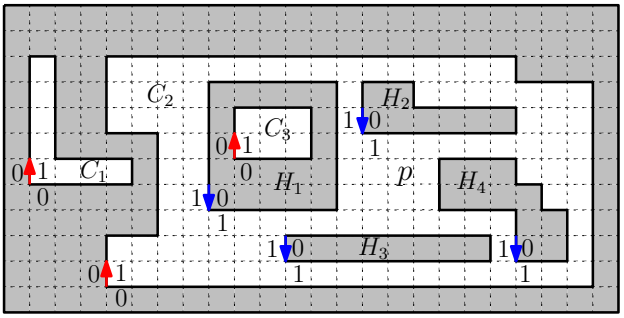

Fig. 1. Geometric model of a binary image. There are three connected components C_1, C_2 and C_3. C_2 contains four holes $H_1, \ldots, H_4$, and H_1 includes an island C_3. Edges are directed so that foreground pixels lie to their right. Canonical edges are indicated by arrows.

2.3 Bidirectional search

It is known that we can enumerate all the canonical edges in $O(n \log n)$ time using the algorithm [3, 4] which was originally designed for traversing a planar map without using any mark bits. By the definition of a canonical edge, an edge e is canonical if and only if there is no other boundary edge e' on the same boundary which is lexicographically smaller than e, that is, $e' < e$. The condition can be tested by following the boundary until we find a smaller edge, but it takes time. A magic for acceleration is to search in two opposite directions (**Bidirectionsl Search**) [4].

Since we can distinguish canonical edges on the external boundaries from those on the internal ones only using local information around them (see Fig. 1), we can count the number of connected components by counting that of canonical edges on external boundaries. Thus, the task CCC can be done in $O(n \log n)$ time using constant work space[1, 2].

Algorithm for counting the number of connected components

$c = 0$. // counter for the number of connected components
for each pixel p in the lexicographic order (raster order)
Let e be the vertical edge to the right of p.
if LocalCondition(p) and IsCanonical(e) **then** increment the counter c.
Report the counter value c as the number of connected components.

Boolean LocalCondition(p){ // Local condition for a canonical pixel
if p is background and p's right pixel is foreground and p's lower right pixel is background **then return** True **else return** False.
}

Boolean IsCanonical(e){ // Is an edge e canonical?
e_f = NextEdge(e). // the next edge of e on the boundary.
e_b = PrevEdge(e). // the previous edge of e on the boundary.
while($e_f > e$ and $e_b > e$) **do**{

```
        e_f = NextEdge(e_f).  // forward search
        if e_f = e_b then return True.  // if two pointers meet then canonical
        e_b = PrevEdge(e_b).  // backward search
        if e_f = e_b then return True.  // if two pointers meet then canonical
    }
    return False.
}
```

In the algorithm above, NextEdge(e) is a function to compute the next edge of e on the boundary (since each boundary is singly connected the next element is uniquely determined). PrevEdge(e) is a function for the previous edge of e. NextEdge(e) and PrevEdge(e) can be computed in constant time. IsCanonical(e) is a function to determine whether an edge is canonical or not. This function cannot be computed in constant time, but the total time we need to evaluate the function for every edge is bounded by $O(n \log n)$ due to bidirectional search.

We now know that the first problem CCC to count the number of connected components is easily solved using only constant work space. It is rather straightforward to extend the algorithm for the task MERR. Whenever we find a canonical edge of an external boundary, we follow the boundary. Then, we can compute the minimum enclosing rectangle of the corresponding component in linear time by maintaining the smallest and largest x and y coordinates of the edges on the boundary. Thus, MERR can be solved in $O(n \log n)$ time using $O(1)$ space.

On the other hand, it is not easy to extend it so that it reports component sizes. Difficulty comes from existence of holes. If a component has no hole, it suffices to follow its boundary to compute its area. Fortunately, it is also known that the problem can be solved in $O(n \log n)$ time with $O(1)$ space by applying the algorithm due to Bose and Morin [4], which is described below.

2.4 Component pixel traversal

The algorithm by Bose and Morin [4] which works on a graph can be modified to deal with a binary image where a graph structure is implicitly given. We start from the canonical edge of the external boundary of a connected component C and follow boundaries associated with C. At each upward vertical edge e walk to the east until we reach a boundary edge f. If f is a canonical edge then we move to f with f as the current edge. Otherwise we first check whether the next edge of e on the boundary is canonical or not. If it is the canonical edge of the external boundary (by further checking whether it is upward) then we are done. If it is the canonical edge of a hole then we walk to the west until we encounter a boundary edge e' and let $e = e'$.

Fig. 2 illustrates how the algorithm traverses pixels in a connected component. Fig. 2(a) shows the first two rows and the entire traversal is given in (b). A more formal description algorithm follows.

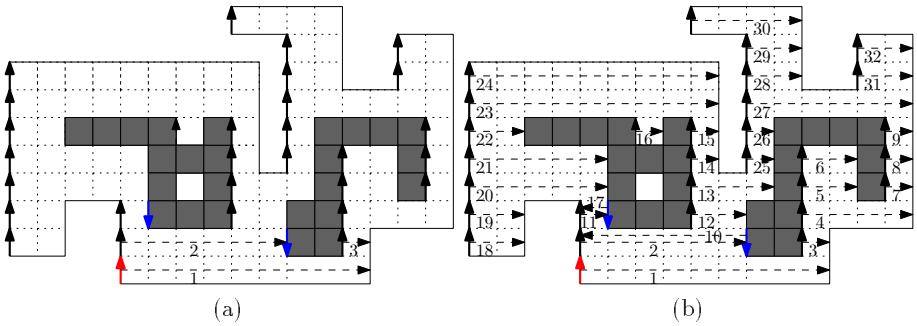

Fig. 2. Component pixel traversal: (a) the first two rows, and (b) the entire traversal.

Algorithm for visiting every pixel starting from an edge e_s.

$e = e_s$. //e_s is the starting edge.
repeat{
  if e is horizontal **then** e =NextEdge(e).
  else if e's eastern pixel is foreground **then**{
    Report consecutive foreground pixels until we reach a boundary edge f.
    if IsCanonical(f) **then** $e = f$ **else** e =NextEdge(e).
  }**else if** IsCanonical(e) **then**
      Walk to the west until we reach an edge f and let $e = f$.
    else e =NextEdge(e).
}**until**($e = e_s$)

Theorem 1. *Let $\mathcal{I}$ be a binary image. When a canonical edge of an external boundary of a connected component C_i is known, we can report all pixels of C_i in $O(a_C + b_C \log b_C)$ time using $O(1)$ work space, where a_C denotes the area of C and b_C denotes the number of edges defining the boundaries of C.*

Using the results listed above, we have the following theorem.

Theorem 2. *Given a binary image of n pixels, we can solve the three basic problems (CCC, MERR, and LCCR) in $O(n \log n)$ time using $O(1)$ work space.*

Proof. Given a binary image, we examine every boundary edge whether it is the canonical edge of an external boundary. It is done in $O(n \log n)$ time in total. Thus, counting the number of connected components is straightforward. To solve MERR it suffices to maintain the minimum and maximum x and y coordinates of edges on an external boundary.

A largest connected component can be computed in two phases. In the first phase we detect all canonical edges of external boundaries. For each such canonical edge e we count the number of pixels of the component associated with e by applying the function for component pixel traversal. In this way we maintain a canonical edge having the largest count. Then, in the second phase we apply the

function for component pixel traversal again with the canonical edge obtained in the first phase. □

3 $O(\sqrt{n})$-space algorithms using run adjacency graph

From now on we will consider algorithms using more work space, $O(\sqrt{n})$ space. But, before introducing such algorithms we will start with a linear-space algorithm for the labeling to explain our basic ideas of run adjacency graph and provisional labels.

In our algorithm we read an input image row by row in the raster manner and convert each row into a sequence of runs. More exactly, a run r is a maximal sequence of foreground pixels (of value 1) in a row. It is associated with an interval $I(r) = \{s, s+1, \ldots, t\}$ when it starts at the s-th column and ends at the t-th column. So, a white run (s, t) in the i-th row means that foreground pixels continue from the s-th column to the t-th column and $(s-1)$-st and $(t+1)$-st pixels are both black pixels in the row (if they exist).

Definition 1. *A foreground run r_1 "**intersects**" another foreground run r_2 if they are in consecutive rows and their associated intervals denoted by $I(r_1)$ and $I(r_2)$ have non-empty intersection.*

We construct a graph called a **Run Adjacency Graph** with vertices being runs and edges between two intersecting runs in consecutive rows. Then, we partition the graph into connected components $\{C_1, C_2, \ldots, C_k\}$ by applying a depth-first search and assign the integral label i to each run in the component C_i. To distinguish connected components in a graph from those in a binary image, we call the former as **graph components** and the latter as **image components**. There is a one-to-one correspondence between graph components and image components. So, the last phase is to convert the run representation into a labeling matrix using the label for each run.

The algorithm described above is almost the same as the old one by Rosenfeld and Pfalts [7], which consists of two phases, horizontal scan to partition each row into runs and vertical scan to merge vertically adjacent runs using a union-find data structure. Differences are (1) we use horizontal scan twice (instead of horizontal scan followed by vertical scan) and (2) we use a depth-first algorithm for computing graph components after building a run adjacency graph (instead of using a union-find tree data structure). Since the depth-first algorithm runs in linear time, the whole algorithm runs in linear time. This is a folklore knowledge although such a formal statement is rather rare in the literature.

In this paper we are interested in space-efficient algorithms, especially using $O(\sqrt{n})$ space. Due to the space constraint we cannot build the whole run adjacency graph. A key idea is to use the $O(\sqrt{n})$ work space to keep a set of runs in two consecutive rows.

Provisional label: an idea to save space

Unfortunately, there are $O(n)$ runs in a binary image and thus $O(n)$ vertices in its associated run adjacency graph. There are two ideas to reduce the work

space. The first idea is to introduce a notion of **provisional labels** which are labels temporarily used in the algorithm and may be different from the final labels to be reported. More important is that we can use the same provisional label for two graph components if they are "clearly" separated.

We read an input binary image in the lexicographic order (raster order) row by row. We read the first row and put provisional labels to those runs in the row. Then, in the second row, we construct a modified run adjacency graph for a set of runs in the two rows and then partition it into connected components (graph components). In this way we know how those runs in the first two rows are connected and thus we can put provisional labels to the runs in the second row. In general, assuming that those runs in the previous row have been labeled, we put provisional labels to the runs in the current row by examining connectivities among those runs. Hereafter, those runs with provisional labels in the previous row are called **colored** runs, and those runs with no provisional labels yet **white** runs. A set of colored runs which have the same provisional label is replaced by a path connecting them in a line.

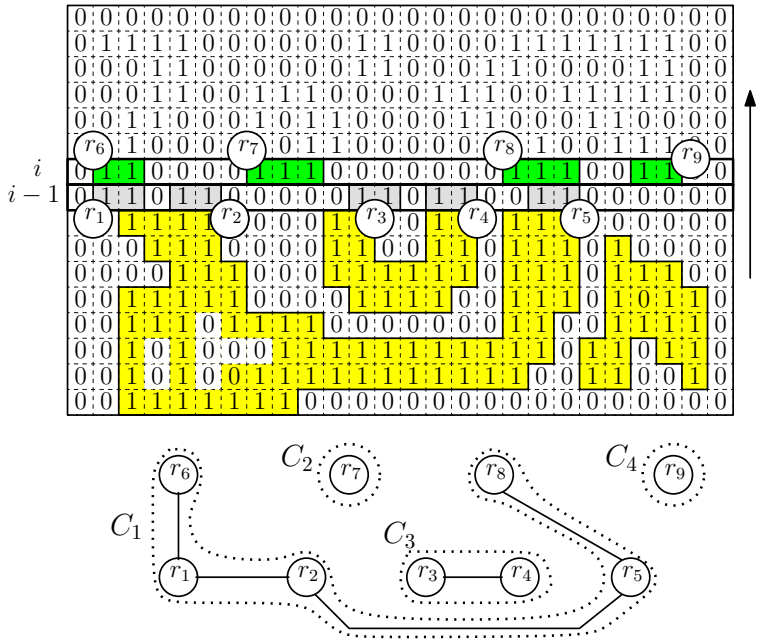

Fig. 3. A set of runs in two consecutive rows (rows $i-1$ and i). Runs $r_1, \ldots, r_5$ in the previous row (row $i-1$) have been labeled by connectivities established in the already scanned part. The run r_7 and r_9 in the current row intersect no run in the previous row, and hence they may create new components. On the other hand, the component associated with the runs r_3 and r_4 does not extend to the current row, and hence the component terminates here. The corresponding run adjacency graph is given below in the figure.

The graph can be partitioned into connected components (graph components) by applying depth-first search which runs in linear time. The resulting graph components are classified into three kinds (see Fig. 3):

Starting component: consisting of a singe white run in the current row,
Terminating component: consisting of colored runs in the previous row, and
Extending component: including both of colored and white runs in the two rows.

1. Starting Component: A graph component C of a single white run.

If a graph component C is a singleton consisting of a single white run in the current row, then it means a new image component starts at the current row, which may be merged in the future scan with another existing image component. So, we need a procedure to create a new image component with a new label. We maintain labels using integers with the current largest label L. Hence, whenever we create a new image component, we increment the value of L and assign the value to the new image component. We also maintain the size of each image component by an array $size[\,]$, which is initially determined by the run length.

2. Terminating Component: A graph component C consisting of colored runs and containing no white run. It may have two or more colored runs, which must be connected in the graph by a path. Therefore, all the colored runs in the component must have a single common provisional label. Since no colored run in the component intersects any white run in the current row, it does not extend to the current row. That is, the corresponding image component terminates in the previous row. This event is called a **death** of the image component.

3. Extending Component: A graph component C having both of colored and white run(s).

If a component C has at least one colored run and at least one white run, then the corresponding image component extends to the next row. If the label k is the smallest label among those colored runs in the graph component, then all the pixels associated with those labels and runs in the graph components should be labeled as k in the next row. At the same time, we update the size of the merged graph component C labeled k by the sum of the sizes of all associated image components.

Fig. 4 illustrates how provisional labels are created, propagated, and terminate during raster scan.

As is seen in Fig. 4, we can save a number of provisional labels. Unfortunately, however, this is not enough to achieve $O(\sqrt{n})$ work space. We need another idea.

A key idea is to reuse provisional labels again and again. A provisional label disappears in two ways. A colored component of a label terminates at some row or it is merged into another terminating component of a different (and smaller) label. The latter case happens when a white run in the current row intersects two or more terminating components of different provisional labels in the previous row. In this case those labels are merged into one of the smallest label together with their associated information such as their sizes.

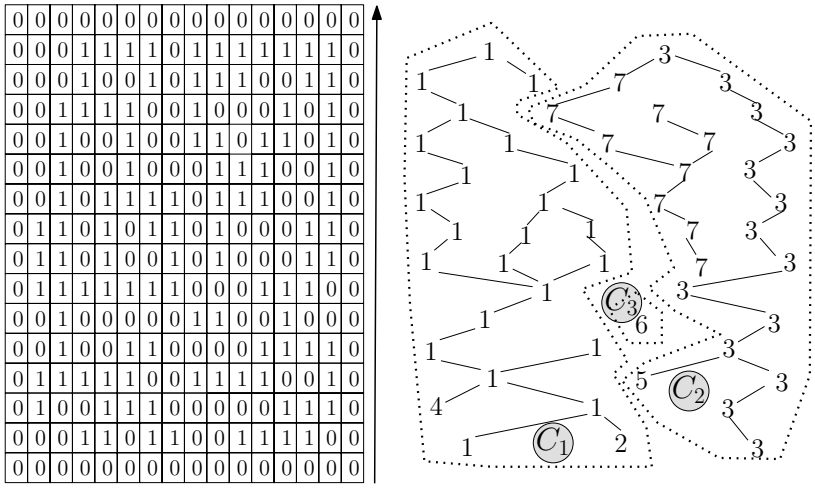

Fig. 4. Provisional labels propagated during raster scan.

3.1 $O(\sqrt{n})$-space algorithm for CCC

Now we are ready to present an $O(\sqrt{n})$-space algorithm for the task CCC for counting the number of connected components in a given binary image of n pixels.

Lemma 1. *For an arbitrary binary image consisting of $O(\sqrt{n})$ rows and columns, the maximum number of image components in a row (or a column) is $O(\sqrt{n})$.*

Proof. If 0's and 1's alternate in a row and all 1-pixels belong to different components we have $O(\sqrt{n})$ components in the row, which is the worst case. Thus, the lemma follows. □

In the above algorithm we have created a new label whenever we find a run which is not connected with any component in the part already scanned. A label may be merged into another. To save work space, we maintain two sets of labels, U and V. The set U keeps a set of labels currently used. The set V is a set of those labels which have been used before but are not used currently. Whenever we need a new label, we check the set V and takes one label out of V unless it is empty. If V is empty then we create a new label and use it by incrementing the value of L. Whenever a label has terminated or is merged into another, we move the label from U to V. In this way we maintain a set of labels. Then, Lemma 1 guarantees that the maximum size of $U \cup V$ is $O(\sqrt{n})$. Thus, we have

Theorem 3. *Given a binary image of n pixels, we can report the number of connected components and the size of each component in $O(n)$ time using $O(\sqrt{n})$ work space.*

Proof. In each row we build a run adjacency graph which contains $O(\sqrt{n})$ vertices and edges. We can decompose it into connected components in linear time using depth-first search. Other operations are done in constant time by standard techniques for such data structures. Thus, the total time required is linear in the number of pixels. □

3.2 $O(\sqrt{n})$-space algorithm for MERR

It is rather straightforward to modify the algorithm for CCC so that it can also report the minimum enclosing rectangle of every connected component using $O(\sqrt{n})$ work space. What we should do is to maintain rectangle information when two or more provisional labels are merged.

Theorem 4. *Given a binary image of n pixels, we can report the minimum enclosing rectangle of every connected component in $O(n)$ time and $O(\sqrt{n})$ work space.*

3.3 $O(\sqrt{n})$-space algorithm for LCCR

We have already had an $O(n \log n)$-time and $O(1)$-space algorithm for solving the problem LCCR (Largest Connected Component Reporting). Can we solve it in linear time using $O(\sqrt{n})$ work space? Unfortunately, it seems very hard, but some small improvement is possible. Exactly speaking, we can design an algorithm which runs in $O(n + m \log m)$ time using $O(\sqrt{n})$ work space, where m is the size of a largest connected component.

We modify our algorithm for reporting the smallest enclosing rectangle for each connected component. By the definition of the canonical edge of the external boundary of a connected component C it must be located at the leftmost lowest corner. We modify the algorithm so that we maintain the following information
(1) the leftmost lowest edge,
(2) the number of pixels
for each label. Then, in one scan over an input image we get the leftmost lower edge e of a largest component in $O(n)$ time. Then, we apply the $O(1)$-space algorithm for component pixel traversal which runs in $O(m \log m)$ time for a connected component of m pixels.

Theorem 5. *Given a binary image of n pixels, we can report all pixels of a largest connected component of size m in $O(n + m \log m)$ time and $O(\sqrt{n})$ work space.*

4 Conclusions

In this paper we have presented space efficient algorithms for some basic tasks on binary images. We have shown that such basic tasks can be done in linear or almost linear time even if work space is limited to $O(\sqrt{n})$ or further to $O(1)$.

Counting the number of connected components is rather easy. In fact we had a linear-time algorithm if $O(\sqrt{n})$ work space is available. However, it is not known whether we can report all pixels in a largest connected component in linear time or not.

In this paper we implicitly assumed 4-connectivity for foreground pixels, that is, two foreground pixels are directly 4-connected if they are horizontally or vertically adjacent. In the 8-connectivity the neighborhood includes all eight neighbors around a pixel. It is rather easy to adapt our algorithms for 8-connectivity. It suffices to modify the definition of intersection between two runs in two consecutive rows. For the 4-connectivity a run $[s_1, t_1]$ intersects another run $[s_2, t_2]$ if their associated intervals have non-empty intersection, that is, $[s_1, t_1] \cup [s_2, t_2] \neq \emptyset$. For the 8-connectivity it is the case if $[s_1, t_1] \cap [s_2 - 1, t_2 + 1] \neq \emptyset$ or $[s_1 - 1, t_1 + 1] \cap [s_2, t_2] \neq \emptyset$.

Acknowledgment

The part of this research of T.A. 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, "Do We Need a Stack to Erase a Component in a Binary Image?," Fifth International Conference on FUN WITH ALGORITHMS, pp.16-27, 2010.
2. 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," edited by S. Albers, H. Alt, and S. Naeher, Springer, pp.249-260, 2009.
3. M. de Berg, M. van Kreveld, R. van Oostrum, and M. Overmars, "Simple traversal of a subdivision without extra storage," International Journal of Geographic Information Systems, 11, pp:359-373, 1997.
4. P. Bose, P. Morin,"An Improved Algorithm for Subdivision Traversal without Extra Storage," Int. J. Comput. Geometry Appl. 12, 4, pp:297-308, 2002.
5. R. Klette and A. Rosenfeld, "Digital Geometry: Geometric Methods for Digital Picture Analysis," Elsevier, 2004.
6. R. Malgouyres, M. More, "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.
7. A. Rosenfeld and J. L. Pfalts, "Sequential operations in digital picture processing," J. ACM, 13(4): 471–494, Oct. 1966.
8. A. Rosenfeld and A. C. Kak, "Digital Picture Processing, 2nd ed.," San Diego, CA: Academic, 1982, vol. 2.