5 Connected-component labeling

5.1 Introduction

This laboratory work presents algorithms for labeling distinct objects from a black and white image. As a result, every object will be assigned a unique number. This number, or label, can be used to process the objects separately.

5.2 Theoretical foundations

We will present several algorithms for labeling. The input for the algorithms is a binary image. The output is a label matrix which has the same dimensions as the input image. It should be capable of storing sufficiently large label values.

In the input binary image the objects are represented as connected components of color black (0), the background is assigned the color white (255). To define what a connected component is, we need to introduce different neighborhood types.

The 4-neighborhood of a position (i,j) is defined to be the set of positions:

$$N_4(i,j)=\{(i-1,j),\ (i,j-1),\ (i+1,j),\ (i,j+1)\},$$

i.e. the upper, left, lower and right neighbors.

The 8-neighborhood consists of all neighboring positions differing by at most 1:

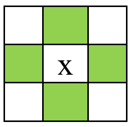
$$N_8(i,j) = \{(k,l)\ |\ |k-i|\le 1,\ |l-j|\le 1,\ (k,l)\neq(i,j)\ \},$$

so it includes the 4-neighborhood and the neighbors situated diagonally.

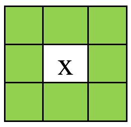
When traversing the image in a particular direction we can define the previous neighbors with regard to this traversal. The previous neighbors for normal top-down, left-right traversal for a position (i,j) is:

$$Np(i,j)=\{(i,j-1),\ (i-1,j-1),\ (i-1,j),\ (i-1,j+1)\}.$$

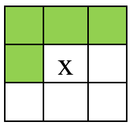
The presented definitions are illustrated below.



a) 4-neighborhood



b) 8-neighborhood



c) previous neighbors

We will define a graph generated by a binary image. The set of vertices is formed by all object pixel positions. The neighboring object pixels determine the edges of the graph. Two positions are neighboring if one is part of the other's neighborhood. We will use N_4 and N_8 so the generated graph is undirected. In this setting a connected component is a set of vertices in which for each pair there is path from vertex 1 to vertex 2.

5.2.1 Algorithm 1 - Breadth first traversal

We start the description with a straightforward method for labeling which relies on breadth first traversal of the graph defined on the image. The first step is to initialize the label matrix to zeroes which indicates that everything is unlabeled. Then algorithm searches for an unlabeled object pixel. If it finds one, it gives it a new label and propagates the label to its neighbors. We repeat this until all object pixels are given a label. In the following we present the steps of the algorithm:

```
label = 0
labels = zeros(height, width)   //height x width matrix with 0
for i = 0:height-1
   for j = 0:width-1
       if img(i,j)==0 and labels(i,j)==0
           label++
           Q = queue()
           labels(i,j) = label
           Q.push( (i,j) )
           while Q not empty
               q = Q.pop()
               for each neighbor in $N_8$(q)
                   if img(neighbor)==0 and labels(neighbor)==0
                       labels(neighbor) = label
                       Q.push( neighbor )
```

Algorithm 1 - Breadth first traversal for connected-component labeling

The queue data structure maintains the list of points that need to be labeled. Since the queue uses a FIFO policy we obtain a breadth first traversal. We mark visited nodes by setting the label for their position. Changing the data structure to a stack would result in a depth first traversal of the image graph.

5.2.2 Algorithm 2 - Two-pass with equivalence classes

Labeling can be achieved by performing two linear passes over the image and some additional processing on a smaller graph. This approach uses less memory. In the previous algorithm we needed the store a list of points. If there is a large connected component, the size of the list is roughly the same as the size of the image.

The current algorithm performs the first pass and labels all object pixels with initial labels. For each pixel we need to consider the previously visited and labeled pixels, so we use the N_P neighborhood defined above. After inspecting the labels of the previous positions we can have the following cases:

- If no previous neighbor was labeled, we create a new label.
- Otherwise, we take the smallest label, called *x*, from the neighbors. Afterwards, we mark each neighboring label *y* as equivalent to *x*.

We assign the label found in the previous step to the current position and continue. After the first pas we have assigned initial labels to each position. However, several labels are equivalent so we need to assign new ones to each equivalence class.

The equivalence relations define an undirected graph on the labels. This graph is usually much smaller than the original graph defined on the whole image. It consists of nodes labeled from 1 to the maximum label value. The edges of the graph indicate the equivalence relations. We can apply Algorithm 1 on this smaller graph to obtain a new list of labels. All labels equivalent to label 1 get relabeled to 1. The next connected component not equivalent to 1 gets relabeled to 2, and so on. A new pass over the labels matrix is necessary to update the labels.

```
label = 0
labels = zeros(height, width)
vector<vector<int>> edges(1000)
for i = 0:height-1
    for j = 0:width-1
        if img(i,j)==0 and labels(i,j)==0
            L = vector()
            for each neighbor in Np(i,j)
                if labels(neighbor)>0
                    L.push_back(labels(neighbor))
            if L.size() == 0        //assign new label
                label++
                labels(i,j) = label
            else                    //assign smallest neighbor
                x = min(L)
                labels(i,j) = x
                for each y from L
                   if (y <> x)
                      edges[x].push_back(y)
                      edges[y].push_back(x)

newlabel = 0
newlabels = zeros(label+1)   //an array of zeroes of length label+1
for i = 1:label
    if newlabels[i]==0
        newlabel++
        Q = queue()
        newlabels[i] = newlabel
        Q.push( i )
        while Q not empty
            x = Q.pop()
            for each y in edges[x]
                if newlabels[y] == 0
                    newlabels[y] = newlabel
                    Q.push( y )

for i = 0:height-1
    for j = 0:width-1
        labels(i,j) = newlabels[labels(i,j)]
```

Algorithm 2 - Two-pass connected-component labeling

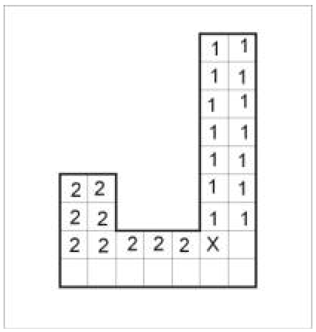


Fig. 5.1 Example of a case when the previous neighbors have different labels. Labels 1 and 2 are marked as equivalent at this step.

5.3 Implementation details

The following code illustrates how to visit the 4-neighborhood of a pixel. It can be easily modified to 8-neighborhood, or to only consider the upper and left neighbors of the pixel.

```
int di[4] = {-1,0,1,0};
int dj[4] = {0,-1,0,1};
uchar neighbors[4];
for(int k=0; k<4; k++)
    neighbors[k] = img.at<uchar>(i+di[k], j+dj[k]);
```

Pay attention to stay within the bounds of the image.

Store the labels in a matrix capable of holding the maximum number of labels:

$2^8 = 256$ - `uchar (CV_8UC1)`
$2^{16} = 65536$ - `short (CV_16SC1)`
$2^{32} \sim 2.1e9$ - `int (CV_32SC1)`

You can use the **`std::stack`** and **`std::queue`** container for storing points for Algorithm 1 to obtain DFS and BFS traversal, respectively. The points can be instances of structure `pair<int,int>`. Sample code for initializing and performing operations on a queue:

```
#include <queue>
queue<pair<int,int>> Q;
Q.push( pair<int,int>(i,j) ); //add as tail of the queue (newest)
pair<int,int> p = Q.front();  //access the front element (oldest)
Q.pop(); //remove the front element
//access position of p
i = p.first; j = p.second;
```

The equivalence relations that define the edges of the smaller graph can be stored using adjacency lists in a **`vector<vector<uchar>>`**. Sample code to initialize and insert edges:

```
//ensure that edges has the proper size
vector<vector<int>> edges(1000);
//if u is equivalent to v
edges[u].push_back(v);
edges[v].push_back(u);
```

To display the label matrix as a color image you need to generate a random color for each label. You should use the default random generator from the standard library. It is better than a call to `rand()%256`.

```
#include <random>
default_random_engine gen;
uniform_int_distribution<int> d(0,255);
uchar x = d(gen);
```

5.4 Labeling examples

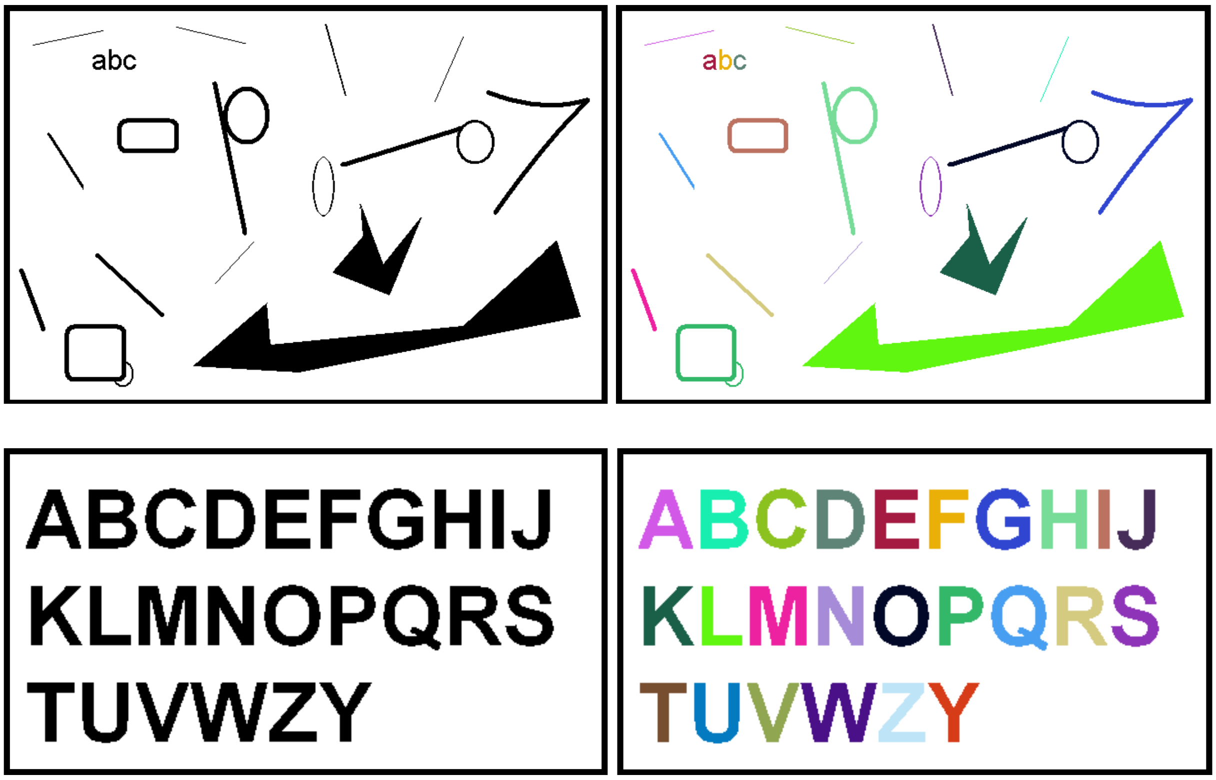


Fig. 5.2 Labeling examples

5.5 Practical Work

1. Implement the breadth first traversal component labeling algorithm (Algorithm 1). You should be able to easily switch between the neighborhood types of 4 and 8.
2. Implement a function which generates a color image from a label matrix by assigning a random color to each label. Display the results.
3. Implement the two-pass component labeling algorithm. Display the intermediate results you get after the first pass over the image. Compare this to the final results and to the previous algorithm.
4. Optionally, visualize the process of labeling by showing intermediate results and pausing after each step to illustrate the order of traversal a selected algorithm.
5. Optionally, change the queue to a stack to perform DFS traversal.
6. **Save your work. Use the same application in the next laboratories. At the end of the image processing laboratory you should present your own application with the implemented algorithms!!!**

5.6 Bibliography

[1]. Umbaugh Scot E, *Computer Vision and Image Processing*, Prentice Hall, NJ, 1998, ISBN 0-13-264599-8

[2]. Robert M. Haralick, Linda G. Shapiro, *Computer and Robot Vision*, Addison-Wesley Publishing Company, 1993.