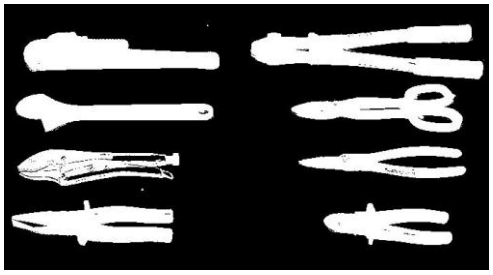
IPBeja Escola Superior de Tecnologia e Gestão

**Trabalho elaborado por:**

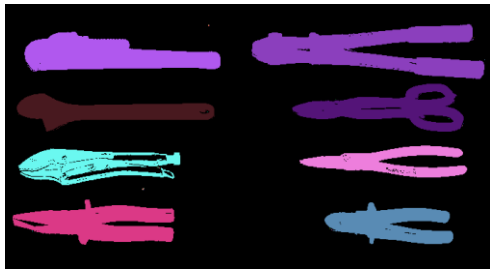
**Carlos Palma nº5608**

**Curso de Eng.Informática**

**Estrutura de Dados e Algoritmos**



# Connected Component Labeling



**Docente:**

José Jasnau Caeiro

**BEJA**

**2010/2011**

# Sumário

O Connected Component Labelling baseia-se na varredura de imagens pixel a pixel da esquerda para a direita, para identificar regiões de pixéis ligados, ou seja pixéis com a mesma intensidade de valores, formando grupos. Após encontrar todos os grupos, cada pixel é marcado com um número ou uma cor de acordo com a componente que foi atribuído. No nosso caso podemos dizer que cada pixel conectado vai lhe ser atribuído uma cor.

# Índice

# Introdução

Este trabalho foi proposto pelo professor José Jasnau Caeiro, no âmbito da disciplina de Estrutura de Dados e Algoritmos e, tem como objectivo a realização de um programa de computador onde é pedido que se implemente o “connected components labeling”.

O Connected Component Labelling é uma aplicaçao algoritmica da teoria dos Grafos, que permite vários subconjuntos conectados. Esta aplicação vai “varrer” uma imagem pixel a pixel e todos os pixeis interligados que partilhem de valores de intensidade similares, são agrupados num conjunto. Seguidamente cada pixel é marcado com um número ou com uma cor, de acordo com o componente que lhe foi atribuído.

O objectivo deste relatório pretende dar a conhecer o algoritmo que deve ser implementado, e a estrutra do respectivo programa.

O presente relatório divide-se em seis partes, iniciando-se pela introduçao, a teoria, o acto experimental, conclusao, bibliografia e os anexos.

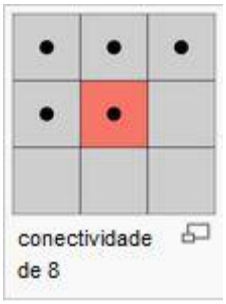
# Enquadramento Teórico

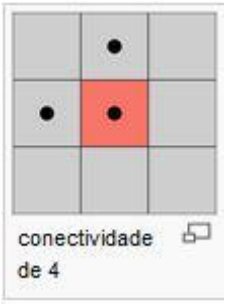
O Connected Component Labelling é geralmente utilizado para se referir ao processo de agrupar pixéis ligados numa imagem.

A abordagem básica é digitalizar uma imagem e atribuir rótulos para cada pixel. O algoritmo Connected Component Labelling pode ser generalizado para dimensões arbitrárias, embora com maior complexidade de tempo e espaço.

O Algoritmo divide-se em dois passos, nos quais são realizadas duas passagens pela imagem. A passagem permite ver equivalências registadas e ao mesmo tempo, atribuir rótulos temporários. O segundo passo serve para substituir cada rótulo temporário pelo rótulo da sua classe de equivalência.

No caso de a conectividade ser de 8, existem quatro vizinhos do pixel "actual": o pixel Nordeste, o pixel Norte, o pixel Noroeste e o pixel Oeste. Por outro lado, se a conectividade for de 4 usa apenas dois pixéis como vizinhos sendo eles o pixel Norte e o pixel Oeste, como se pode verificar nas seguintes imagens.




Existem algumas condições importantes para a implementação do algoritmo em causa. No caso de o algoritmo ser de digitalização de "*raster*", é dividido em duas passagens.

Primeira passagem:

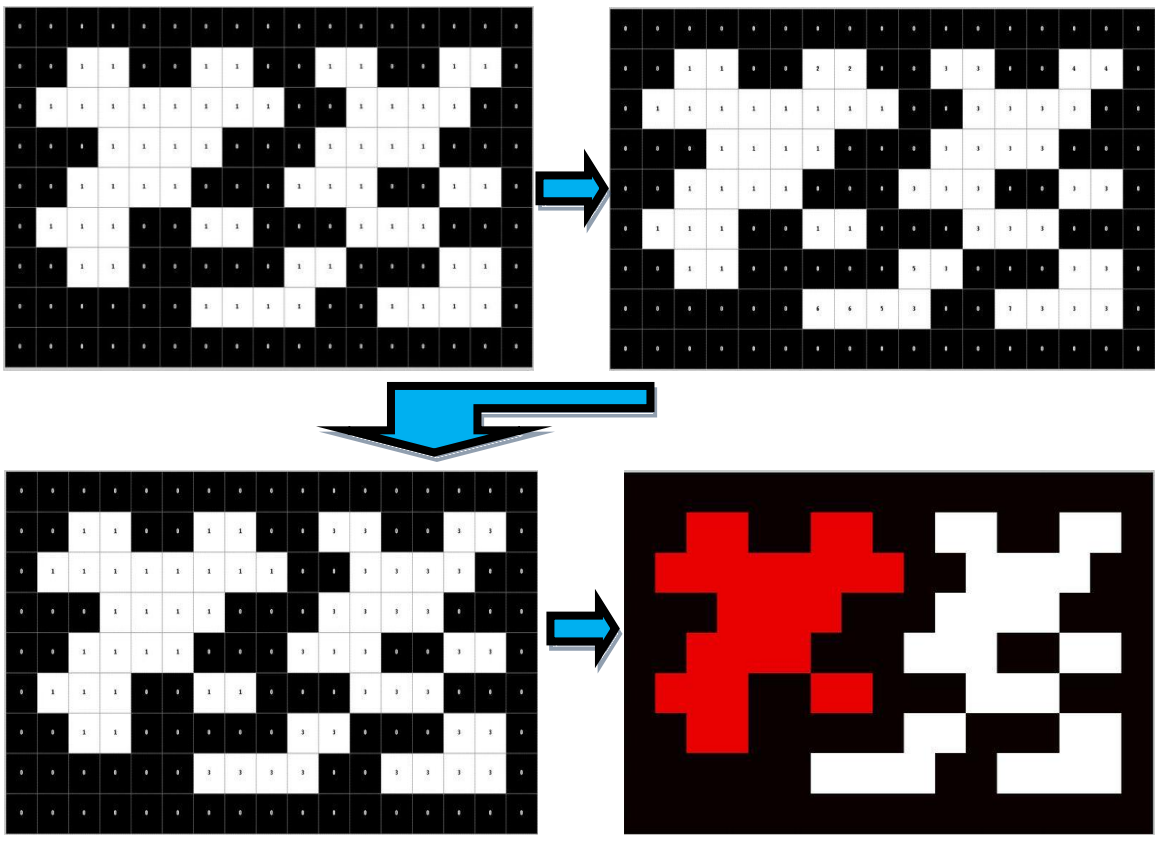
1. Iterar cada elemento de dados por coluna e em seguida por linha, isto é, varrer a imagem.
2. Se o elemento não é o fundo da imagem, ou seja, pixel com o valor zero.

a. Procurar os vizinhos do elemento actual;
b. No caso de não existirem vizinhos, marca-se o elemento actual com um rótulo e continua-se;
c. Caso Contrário, encontra o vizinho com o rótulo menor e vai atribui-lo ao elemento actual;
d. No final armazena as equivalências entre os vizinhos que têm rótulos.

Segunda Passagem:

1. Iterar cada elemento de dados por coluna e em seguida por linha, isto é, varrer a imagem.
2. Se o elemento não é o fundo da imagem, ou seja, pixel com o valor zero.
    a. Aplica o elemento com o rótulo equivalente menor.

As imagens que se seguem exemplificam como o algoritmo se vai processar.



# Acto Experimental

Inicialmente importa referir que, este programa foi criado através de uma linguagem de programação de alto nível orientada ao objecto, linguagem essa designada por Java.

A aplicação criada abre uma imagem a cores, aplicando-lhe um método de binarização, que faz com que a imagem inicial (a cores), fique numa imagem monocromática, ou seja, numa imagem binária, que consiste numa representação de pixéis a preto e branco. A essa imagem aplica-se o algoritmo Connected Component Labelling que originará uma imagem de saída a cores, em que cada região conectada que se encontrava a branco vai representar uma cor diferente, como representado da figura seguinte. Posteriormente, a imagem de saída vai ser escrita num ficheiro.

**Imagem Inicial**

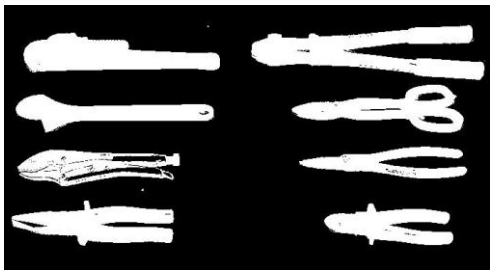


**Imagem Monocromática**

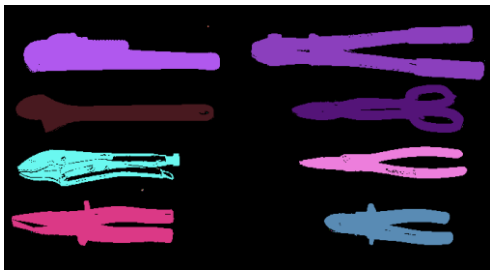


**Imagem de Saída**

O programa desenvolvido é constituído por três classes: a classe Main, a classe OperacoesImagem e a classe TwoPass.

Relativamente à classe Main, esta é definida por um objecto da classe OperacoesImagem, que chama o método open dessa mesma classe enviando-lhe o nome do ficheiro da imagem a cores (Imagem inicial). Seguidamente chama o método binarization, que faz com que a imagem inicial fique monocromática. Depois de a imagem estar monocromática é criado um objecto da classe TwoPass, onde vai chamar o método executar da classe TwoPass, que faz com que a imagem que se encontrava a preto e branco fique a cores, isto é, cada região conectada que se encontrava a branco vai representar uma cor diferente. Para finalizar, a imagem monocromática e a imagem a cores diferentes vão ser guardadas num ficheiro de saída. De referir ainda que é nesta classe que se vai calcular o tempo de execução do programa.

Quanto à Classe OperacoesImagem, esta é constituída por cinco métodos, sendo eles **open, save, getWidth, getHeight** e **binarization**. Passando a explicar mais detalhadamente cada um deles, o método *open*, que vai receber como parâmetro o nome do ficheiro e é o responsável por abrir (leitura) esse mesmo ficheiro. O método *save* também recebe como parâmetro um nome de um ficheiro, neste caso o nome com que o ficheiro vai ser guardado e é responsável por guardar esse mesmo ficheiro (escrita). O método *getWidth*, retorna a largura da imagem, enquanto que o método *getHeight*, devolve a altura da imagem. Por último o método *binarization* recebe como parâmetro um valor inteiro e é o método que faz com que a imagem fique monocromática.

Por fim, a Classe TwoPass que é constituída por 10 métodos: **save, executar, porPixel, buscarPixel, buscarMarcasVizinhas, contarMarcasDiferentes, buscarMarcaMaisBaixa, AcharConjunto, resolverEquivalencias, colorizar.** No que se refere ao método *save,* este recebe como parâmetro um nome de um ficheiro, neste caso o nome com que o ficheiro vai ser guardado e é responsável por guardar esse mesmo ficheiro (escrita). O método *porPixel* que recebe como parâmetro três variáveis do tipo inteiro, sendo elas a *col* (coluna), a *line* (linha) e o valor (marca), recebendo também como parâmetro *wr* do tipo *WritableRaster*, o método é responsável por repor o pixel. O método

*buscarPixel* que recebe como parâmetro duas variáveis do tipo inteiro, sendo elas a *col* (coluna), a *line* (linha) e também um *wr* do tipo *WritableRaster*, o método é responsável por devolver a cor do pixel dependendo da coluna e da linha onde se encontra. O método *buscarMarcasVizinhas* que recebe como parâmetro a *col* (coluna) e a *line* (linha), sendo este o método responsável por procurar os vizinhos do pixel "actual" (como referido anteriormente), devolvendo as *marcasVizinhas* (vizinhos). O método *contarMarcasDiferentes* que recebe como parâmetro uma variável (marcasVizinhas) do tipo *array* de inteiros, o método em si vai criar uma variável HashSet<Integer> com o nome *conjuntoMarcasDiferentes* e sempre que a marca vizinha for maior que zero vai adicionar essa marca ao conjunto, devolvendo o tamanho do conjunto de marcas diferentes (conjuntoMarcasDiferentes). O método *buscarMarcaMaisBaixa* vai receber como parâmetro uma variável (marcasVizinhas) do tipo *array* de inteiros, o método em si vai calcular a marca vizinha mais baixa, devolvendo essa mesma marca. O método *AcharConjunto* vai receber como parâmetro uma variável (k) do tipo inteiro e vai procurar o conjunto.

# Conclusão

Com a realização desde trabalho concluí que para implementar um algoritmo como o “Connected Component Labelling”, há a necessidade de haver um estudo prévio, estudo esse que se baseia em perceber bem o pseudocódigo do algoritmo e encontrar maneira do implementar. Importa referir que para implementar este programa temos que saber usar bem as estruturas de dados estudadas e implementadas nas aulas.

Através da execução deste trabalho constatei que “Connected Component Labelling” é de extrema importância na actualidade em aplicações de análise de imagens automatizadas, como por exemplo na interacção pessoa-computador e no reconhecimento de imagem, entre outros.

Posto isto posso afirmar que os objectivos deste trabalho foram alcançados.

# Bibliografia


- http://en.wikipedia.org/wiki/Connected_Component_Labeling
- http://homepages.inf.ed.ac.uk/rbf/HIPR2/label.htm
- http://www.izbi.uni-leipzig.de/izbi/publikationen/publi_2004/IMS2004_JankowskiKuska.pdf
- http://homepages.inf.ed.ac.uk/rbf/HIPR2/labeldemo.htm

# Anexos

## Classe Main

```
/**
 * Class Main
 * */
public class Main
{
   /**
    * @param args
    */
   public static void main(String[] args)
   {
      OperacoesImagem op = new OperacoesImagem();
      op.open("images/alicates.bmp");
      op.binarization(240);

      /*
       * criação do objecto que representa o resultado da marcação de
       * componentes conexos
       */
      TwoPass tp = new TwoPass(op.wr);
      double tempo = System.nanoTime();
      tp.executar();
      tempo = System.nanoTime() - tempo;
      System.out.println(tempo * 0.000000001 + " segundos");
      tp.save("images/alicates_Cores.bmp");
      op.save("images/alicates_Monocromatico.bmp");

   }
}
```

## Classe OperacoesImagem

```
package src;
import java.awt.image.BufferedImage;
import java.awt.image.WritableRaster;
import java.io.File;
import java.io.IOException;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.imageio.ImageIO;
/**
 * @author Carlos
 * 2010
 */
public class OperacoesImagem
{
   public BufferedImage  bi;
   public WritableRaster wr;
   /**
    * Construtor da Class OperacoesImagem
    * */
   public OperacoesImagem()
   {
      this.bi = null;
      this.wr = null;
   }
```

```
/**
 * Metodo open
 * @param nome_ficheiro
 *          - nome do ficheiro para que vai ser aberto
 * */
public void open(String nome_ficheiro)
{
   File ficheiro;
   ficheiro = new File(nome_ficheiro);
   try
   {
      this.bi = ImageIO.read(ficheiro);
   } catch (IOException ex)
   {
      Logger.getLogger(OperacoesImagem.class.getName()).log(Level.SEVERE,
            null, ex);
   }
   this.wr = this.bi.getRaster();
}
/**
 * Metodo Save
 * @param nome_ficheiro
 *          - nome do ficheiro para que vai ser guardado
 * */
public void save(String nome_ficheiro)
{
   File ficheiro;
   ficheiro = new File(nome_ficheiro);
   try
   {
      ImageIO.write(this.bi, "BMP", ficheiro);
   } catch (IOException ex)
   {
      Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
   }
}
/**
 * Metodo getWidth
 * @return bi.getWidth
 * */
public int getWidth()
{
   return bi.getWidth();
}
/**
 * Metodo getHeight
 * @return bi.getHeight
 * */
public int getHeight()
{
   return bi.getHeight();
}
/**
 * Metodo binarization
 * @param th - valor inteiro
 * */
public void binarization(int th)
{
   int red, green, blue;
   int coluna, linha;
```

```
      int nivel_cinzento;
      for (coluna = 0; coluna < this.getWidth(); coluna++)
      {
     for (linha = 0; linha < this.getHeight(); linha++)
        {
          int[] color = null;
          color = this.wr.getPixel(coluna, linha, color);
          red = color[0];
          green = color[1];
          blue = color[2];
          nivel_cinzento = (int) ((red + green + blue) / 3.0);
          if (nivel_cinzento < th)
          {
            color[0] = 0xff;
            color[1] = 0xff;
            color[2] = 0xff;
            this.wr.setPixel(coluna, linha, color);
          }
          else
          {
            color[0] = 0x00;
            color[1] = 0x00;
            color[2] = 0x00;
            this.wr.setPixel(coluna, linha, color);
          }
        }
      }
    }
}
```

## Classe TwoPass

```
package src;
import java.awt.image.BufferedImage;
import java.awt.image.WritableRaster;
import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.Hashtable;
import java.util.LinkedList;
import java.util.Random;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.imageio.ImageIO;
/**
 * @author Carlos
 *
 */
public class TwoPass
{
   private BufferedImage              img_marcas       = null;
   /*
    * opera-se directamente na imagem através dos seus rasters
    */
   private WritableRaster             wr_img_entrada    = null;
   private WritableRaster             wr_marcas        = null;
   private int                        NCOLS;
   private int                        NLINES;
   private Hashtable<Integer, int[]>    mapaCores        = null;
```

```
private LinkedList<HashSet<Integer>> listaEquivalentes = null;
/**
 * Construtor da Class TwoPass
 * @param wr_img_entrada - do tipo WritableRaster
 * */
public TwoPass(WritableRaster wr_img_entrada)
{
   /*
    * img_entrada - imagem monocromatica com valores 0 ou 255
    */
   this.wr_img_entrada = wr_img_entrada;
   /*
    * inicializacao das estruturas de dados usadas no algoritmo
    */
   /*
    * dimensões da imagem
    */
   this.NCOLS = this.wr_img_entrada.getWidth();
   this.NLINES = this.wr_img_entrada.getHeight();
   /*
    * a img_marcas tem a mesma dimensão da imagem de entrada
    */
   this.img_marcas = new BufferedImage(this.NCOLS, this.NLINES,
         BufferedImage.TYPE_INT_RGB);
   this.wr_marcas = this.img_marcas.getRaster();
   /* inicializar as listas */
   this.mapaCores = new Hashtable<Integer, int[]>();
   this.listaEquivalentes = new LinkedList<HashSet<Integer>>();
}
/**
 * Metodo Save
 * @param nome_ficheiro
 *          - nome do ficheiro para que vai ser guardado
 * */
public void save(String nome_ficheiro)
{
   File ficheiro;
   ficheiro = new File(nome_ficheiro);
   try
   {
      ImageIO.write(this.img_marcas, "BMP", ficheiro);
   } catch (IOException ex)
   {
      Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
   }
}
/**
 * Metodo executar
 * @return this.img_marcas - do tipo BufferedImage
 * */
public BufferedImage executar()
{
   int marca = 0;
   for (int line = 1; line < this.NLINES - 1; line++)
   {
      for (int col = 1; col < this.NCOLS - 1; col++)
      {
         /*
          * O algoritmo só opera em objectos. Os pixeis dos objectos têm
          * valor > 0
```

```
 */
int pixel_objecto = this
        .buscarPixel(col, line, this.wr_img_entrada);
if (pixel_objecto > 0)
{
  /*
   * 1. buscam-se as marcas vizinhas 2. contam-se quantas sao
   * diferentes umas das outras 3. se as marcas diferentes forem
   * zero entao não ha marca e aumenta-se marca.
   */
  int[] marcasVizinhas = this.buscarMarcasVizinhas(col, line);
  int marcasDiferentes = this
          .contarMarcasDiferentes(marcasVizinhas);
  if (marcasDiferentes > 0)
  {
    this.porPixel(col, line, this
            .buscarMarcaMaisBaixa(marcasVizinhas), wr_marcas);
    if (marcasDiferentes > 1)
    {
      /*
       * Equivalencias
       */
      ArrayList<HashSet<Integer>> conjuntos = new ArrayList<HashSet<Integer>>(
              4);
      for (int k = 0; k < 4; k++)
      {
        conjuntos.add(new HashSet<Integer>());
      }
      HashSet<Integer> novoConjunto = new HashSet<Integer>();
      for (int k = 0; k < 4; k++)
      {
        conjuntos.set(k, this.AcharConjunto(marcasVizinhas[k]));
        if (marcasVizinhas[k] > 0)
        {
          if (conjuntos.get(k).isEmpty())
          {
            HashSet<Integer> conj = new HashSet<Integer>();
            conj.add(marcasVizinhas[k]);
            conjuntos.set(k, conj);
          }
        }
      }
      for (int k = 0; k < 4; k++)
      {
        if (!conjuntos.get(k).isEmpty())
        {
          this.listaEquivalentes.remove(conjuntos.get(k));
          novoConjunto.addAll(conjuntos.get(k));
        }
      }
      if (!novoConjunto.isEmpty())
      {
        this.listaEquivalentes.add(novoConjunto);
      }
    }
  }
  else
  {
    marca = marca + 1;
    this.porPixel(col, line, marca, this.wr_marcas);
```

```
          }
        }
      }
    }
    this.resolverEquivalencias();
    this.colorizar();
    return this.img_marcas;
  }
  /**
   * Metodo porPixel
   * @param col - do tipo int
   * @param line - do tipo int
   * @param valor - do tipo int
   * @param wr - do tipo WritableRaster
   * */
  private void porPixel(int col, int line, int valor, WritableRaster wr)
  {
    int[] cor = new int[3];
    cor[0] = valor;
    cor[1] = valor;
    cor[2] = valor;
    wr.setPixel(col, line, cor);
  }
  /**
   * Metodo buscarPixel
   * @param col - do tipo int
   * @param line - do tipo int
   * @param wr - do tipo WritableRaster
   * @return cor[0]
   * */
  private int buscarPixel(int col, int line, WritableRaster wr)
  {
    int[] cor = null;
    cor = wr.getPixel(col, line, cor);
    return cor[0];
  }
  /**
   * Metodo buscarMarcasVizinhas
   * @param col - do tipo int
   * @param line - do tipo int
   * @return marcasVizinhas
   * */
  private int[] buscarMarcasVizinhas(int col, int line)
  {
    int[] marcasVizinhas = new int[4];
    marcasVizinhas[0] = this.buscarPixel(col - 1, line, this.wr_marcas);
    marcasVizinhas[1] = this.buscarPixel(col - 1, line - 1, this.wr_marcas);
    marcasVizinhas[2] = this.buscarPixel(col, line - 1, this.wr_marcas);
    marcasVizinhas[3] = this.buscarPixel(col + 1, line - 1, this.wr_marcas);
    return marcasVizinhas;
  }
  /**
   * Metodo contarMarcasDiferentes
   * @param marcasVizinhas - do tipo int[]
   * @return conjuntoMarcasDiferentes.size()
   * */
  private int contarMarcasDiferentes(int[] marcasVizinhas)
  {
    int NVIZINHOS = 4;
    HashSet<Integer> conjuntoMarcasDiferentes = new HashSet<Integer>();
```

```
    for (int k = 0; k < NVIZINHOS; k++)
    {
      if (marcasVizinhas[k] > 0)
      {
        conjuntoMarcasDiferentes.add(marcasVizinhas[k]);
      }
    }
    return conjuntoMarcasDiferentes.size();
  }

  /**
   * Metodo buscarMarcaMaisBaixa
   * @param marcasVizinhas - do tipo int[]
   * @return minimo
   * */
  private int buscarMarcaMaisBaixa(int[] marcasVizinhas)
  {
    int minimo = Integer.MAX_VALUE;
    for (int k = 0; k < 4; k++)
    {
      if (marcasVizinhas[k] > 0 && marcasVizinhas[k] < minimo)
      {
        minimo = marcasVizinhas[k];
      }
    }
    return minimo;
  }
  /**
   * Metodo AcharConjunto
   * @param k - do tipo int
   * @return conjunto
   * */
  private HashSet<Integer> AcharConjunto(int k)
  {
    HashSet<Integer> conjunto = new HashSet<Integer>();
    for (HashSet<Integer> setX : this.listaEquivalentes)
    {
      if (k > 0)
      {
        if (setX.contains(k))
        {
          conjunto = setX;
          return conjunto;
        }
      }
    }
    return conjunto;
  }
  /**
   * Metodo resolverEquivalencias
   * */
  private void resolverEquivalencias()
  {
    Hashtable<Integer, Integer> mapaEquivalente = new Hashtable<Integer, Integer>();
    int valor = 0;
    for (HashSet<Integer> setX : this.listaEquivalentes)
    {
      valor = valor + 1;
      for (Integer marca : setX)
      {
```

```
            mapaEquivalente.put(marca, valor);
          }
        }
        for (int line = 1; line < this.NLINES - 1; line++)
        {
          for (int col = 1; col < this.NCOLS - 1; col++)
          {
            int marca = this.buscarPixel(col, line, this.wr_marcas);
            int[] cor = new int[3];
            if (marca > 0)
            {
              if (mapaEquivalente.containsKey(marca))
              {
                cor[0] = mapaEquivalente.get(marca);
                cor[1] = mapaEquivalente.get(marca);
                cor[2] = mapaEquivalente.get(marca);
              }
              else
              {
                valor = valor + 1;
                cor[0] = valor;
                cor[1] = valor;
                cor[2] = valor;
              }
              this.wr_marcas.setPixel(col, line, cor);
            }
          }
        }
      }
      /**
       * Metodo colorizar
       * */
      private void colorizar()
      {
        HashSet<Integer> conjuntoUnicoMarcas = new HashSet<Integer>();
        for (int line = 1; line < this.NLINES - 1; line++)
        {
          for (int col = 1; col < this.NCOLS - 1; col++)
          {
            int marca = this.buscarPixel(col, line, this.wr_marcas);
            if (marca > 0)
            {
              conjuntoUnicoMarcas.add(marca);
            }
          }
        }
        Random rng = new Random();
        for (Integer marca : conjuntoUnicoMarcas)
        {
          int marcaInteira = marca.intValue();
          int[] cor = new int[3];
          cor[0] = rng.nextInt(255);
          cor[1] = rng.nextInt(255);
          cor[2] = rng.nextInt(255);
          this.mapaCores.put(marcaInteira, cor);
        }
        for (int line = 1; line < this.NLINES - 1; line++)
        {
          for (int col = 1; col < this.NCOLS - 1; col++)
          {
```

```
            int marca = this.buscarPixel(col, line, this.wr_marcas);
            if (marca > 0)
            {
               int[] c = this.mapaCores.get(marca);
               this.wr_marcas.setPixel(col, line, c);
            }
         }
      }
   }
}
```