Apêndice II

Fontes dos Principais Algoritmos Utilizados

Neste capitulo estão disponíveis os códigos fontes dos principais algoritmos implementados neste trabalho. Os programas foram desenvolvidos e compilados utilizando o C++ Builder 5 da Borland. A linguagem foi escolhida por ser eficiente e bem conhecida, além de ser bem compreensível. Os demais fontes não presentes aqui, juntamente com a interface gráfica, encontram-se disponíveis no CD que acompanha esta dissertação.

Os fontes estão organizados de acordo com os capítulos da dissertação onde seus algoritmos foram discutidos. O primeiro fonte refere-se ao capitulo 3 e trata-se do algoritmo para conversão de uma imagem do espaço RGB para HSI. Em seguida estão disponíveis alguns fontes referentes à segmentação (capitulo 4), os quais: Calibração das imagens, threshold com método de Niblack e Otsu, Opening (Erosão e Dilatação) e Rotulação. A seguir o código fonte da rede neural do capítulo 5 juntamente com os pesos, está disponível. Por último é disponibilizado o fonte do algoritmo que mede e erro da segmentação, referente ao capítulo 6.

```
//---------------------------------------------------------------------------------------------------------------------
Funcao que converte uma imagem RGB para HSI
//---------------------------------------------------------------------------------------------------------------------
void __fastcall rgb2hsi(Graphics::TBitmap *image)
{
  Graphics::TBitmap *temp = new Graphics::TBitmap();
  Byte *ptr,*hsi;
  float r,g,b;
  float R,G,B;
  float h,s,i;
  float va,er,eg,eb,soma,raiz;

  temp->Assign(image);

  for(int y=0;y<image->Height;y++)
    {
      ptr = (Byte *)image->ScanLine[y];
      hsi = (Byte *)temp->ScanLine[y];

      for(int x=0;x<(image->Width*3);x=x+3)
        {
   /// Intensity ------------
          R = ptr[x+2]; // le valores originais RGB
          G = ptr[x+1]; //Isso sera usado para calcular os 3 canais
          B = ptr[x];
          i = (R+G+B)/3; // o i guarda o valor resultante

   /// Saturation -----------

          if(R==0)r=0;
          else  r = R/(R+G+B); //normaliza os valores de rgb
          if(G==0)g=0;
          else  g = G/(R+G+B); //serao representados pelas variaveis
          if(B==0)b=0;
          else  b = B/(R+G+B); //r,g,b em minusculo em todo o codigo

          s = 100*(1-3*min(r,g,b));// s representa porcentagem por isso
          if(s<0)s = s*-1;
          s = 2.55*s;  //multiplico por 100. Da porcentagem resultante
                  //acho o equivalente de 0 a 255.
          //O resultado do calculo esta na var "s" ------

   /// HUE -----------
          va = (0.66666666666*(r-0.33333333333)-0.33333333333*
              (b-0.33333333333)-0.33333333333*(g-0.33333333333));
          er = (r-0.33333333333)*(r-0.33333333333);
          eg = (g-0.33333333333)*(g-0.33333333333);
          eb = (b-0.33333333333)*(b-0.33333333333);
          soma = er+eg+eb;
          raiz = sqrt(0.66666666666*soma);


          if(((va/raiz)<1)&&(va/raiz)>=-1)
           {
             h = acos(va/raiz);    //o valor de hue é em radianos
             h = (180/3.141592654)*h;
```

```
                if(b>g)
                  h = 360-h;

                h = (0.708333333)*h; //o valor ao lado = 255/360
               }
              else
               h=0;
              //o valor do calculo de HUE está na var "h" -------
              hsi[x+2] = h;
              hsi[x+1] = s;
              hsi[x]   = i;
            }//for x
      }//for y

   image->Assign(temp);
   temp->FreeImage();
}
//---------------------------------------------------------------------------------------------------------------------
```

```
//---------------------------------------------------------------------------------------------------------------------
Calcula a defasagem da amostra de branco na imagem (amostra) e retorna os
valores r, g e b para a funcao chamadora.
//---------------------------------------------------------------------------------------------------------------------
void __fastcall calc_amostra(Graphics::TBitmap *amostra,float *r,float *g, float *b)
{
  Byte *ptr;
  long int sr=0,sg=0,sb=0;
  float db,dr,dg;
  int count=0;

  for(int y=0;y<amostra->Height;y++)
    {
      ptr = (Byte *)amostra->ScanLine[y];
      for(int x=0;x<(amostra->Width*3);x+=3)
        {
          sb = sb+ptr[x];
          sg = sg+ptr[x+1];
          sr = sr+ptr[x+2];
          count++;
        }
    }
  sb = sb/(count);
  sg = sg/(count);  //media da amostra
  sr = sr/(count);

  dr = sr;
  dg = sg;
  db = sb;

  db = 255/db;
  dg = 255/dg; //valores multiplicadores
  dr = 255/dr;

  *r=dr;
  *g=dg;//retorna os 3 valores atraves dos ponteiros
  *b=db;
}
//---------------------------------------------------------------------------------------------------------------------
```

```
//---------------------------------------------------------------------------------------------------------------------
Função que calibra a imagem baseada na amostra de branco. Recebe como entrada
a imagem que será calibrada e os valores r,g e b calculados na função calc_amostra.
//---------------------------------------------------------------------------------------------------------------------
void __fastcall calibrar(Graphics::TBitmap *image, float dr,float dg,float db)
{
  Byte *ptr;
  int r,g,b;

  for(int y=0;y<image->Height;y++)
    {
      ptr = (Byte *)image->ScanLine[y];
      for(int x=0;x<(image->Width*3);x+=3)
        {
          r = ptr[x+2]*dr;
          g = ptr[x+1]*dg;
          b = ptr[x]*db;

          if(b<256)
            ptr[x]  = b;
          else
            ptr[x] = 255;

          if(g<256)
            ptr[x+1]= g;
          else
            ptr[x+1] = 255;

          if(r<256)
            ptr[x+2]= r;
          else
            ptr[x+2] = 255;
        }
    }
}
//---------------------------------------------------------------------------------------------------------------------
```

```
//---------------------------------------------------------------------------
Niblack Method
//---------------------------------------------------------------------------
void __fastcall niblack_threshold(Graphics::TBitmap *image)
{
  Byte *ptr;
  float k=0.05;
  int threshold;
  double med,dp,soma;

  soma=0;

  for(int y=0;y<image->Height;y++)
    {
      ptr = (Byte *)image->ScanLine[y];
      for(int x=0;x<image->Width;x++)
        {
          soma=soma+ptr[x];
        }
    }
  med=soma/(image->Width*image->Height);//calcula a media da imagem
  dp = desvio_padrao(med,image);//calcula o desvio padrao da imagem
  threshold = med + k*dp;//threshold de Niblack

  for(int y=0;y<image->Height;y++)
    {
      ptr = (Byte *)image->ScanLine[y];
      for(int x=0;x<image->Width;x++)
        {
          if(ptr[x]>threshold)
            ptr[x]=255;
          else
            ptr[x]=0;
        }
    }
}
//---------------------------------------------------------------------------
```

```
//---------------------------------------------------------------------------
Otsu Method
//---------------------------------------------------------------------------
void __fastcall otsu_threshold(Graphics::TBitmap *image)
{
  Byte *ptr;
  float vet[254];
  float p1,p2,p12;
  int threshold,t;
  float a[256],b[256];
  int hist[256];

  histograma(image,*hist);//calcula o histograma da imagem e armazeno no
                          //vetor hist
  for(int k=1;k<255;k++)
    {
      p1=Px(0,k);
      p2=Px(k+1,255);
      p12=p1*p2;
      if(p12==0) p12=1;
      vet[k] = pow((Mx(0,k)*p2)-(Mx(k+1,255)*p1),2)/p12;
      if(p1==0)p1=1;
      a[k]=Mx(0,k)/p1;
      if(p2==0)p2=1;
      b[k]=Mx(k,255)/p2;

    }

  threshold = maior(vet);//procura o índice onde esta o maior valor do vetor vet
  t = threshold; //o threshold é o indice onde esta o maior valor

  for(int y=0;y<image->Height;y++)
    {
      ptr=(Byte *)image->ScanLine[y];
      for(int x=0;x<image->Width;x++)
        {
          if(ptr[x]<threshold)
            ptr[x]=0;
          else
            ptr[x]=255;
        }
    }
}
//---------------------------------------------------------------------------
```

```
//---------------------------------------------------------------------------
Erosão
//---------------------------------------------------------------------------
void __fastcall erosao(Graphics::TBitmap *image)//imagem binaria
{
  Graphics::TBitmap *i = new Graphics::TBitmap();
  int ponto,s;
  Byte *ptr_s,*ptr;

  i->Assign(image);

   for(int y=1;y<image->Height-1;y++)
     {
       ptr=(Byte *)i->ScanLine[y];
       for(int x=1;x<image->Width-1;x++)
         {
           s=0;
           for(int i=y-1;i<=y+1;i++)
             {
               ptr_s = (Byte *)image->ScanLine[i];
               for(int j=x-1;j<=x+1;j++)
                 {
                   if(ptr_s[j]!=255)
                     s++;
                 }
             }
           if(s==0)
             ptr[x]=255;
           else
             ptr[x]=0;
         }
     }
  image->Assign(i);
  i->FreeImage();
}
//---------------------------------------------------------------------------
```

```
//---------------------------------------------------------------------------
Dilatação
//---------------------------------------------------------------------------
void __fastcall dilatacao(Graphics::TBitmap *image)//imagem binaria
{
  Graphics::TBitmap *i = new Graphics::TBitmap();
  int ponto,s;
  Byte *ptr_s,*ptr;

  i->Assign(image);

   for(int y=1;y<image->Height-1;y++)
     {
       ptr=(Byte *)i->ScanLine[y];
       for(int x=1;x<image->Width-1;x++)
         {
           s=0;
           for(int i=y-1;i<=y+1;i++)
             {
               ptr_s = (Byte *)image->ScanLine[i];
               for(int j=x-1;j<=x+1;j++)
                 {
                   if(ptr_s[j]==255)
                     s++;
                 }
             }
           if(s>0)
             ptr[x]=255;
           else
             ptr[x]=0;
         }
     }
  image->Assign(i);
  i->FreeImage();
}
//---------------------------------------------------------------------------
```

```
//---------------------------------------------------------------------------------------------------------------------
Algoritmo de Rotulação
//---------------------------------------------------------------------------------------------------------------------
void __fastcall labeling(Graphics::TBitmap *image)//imagem binaria
{
  Graphics::TBitmap *classe = new Graphics::TBitmap();
  Byte *ptrx,*ptry,*px,*py;
  int objeto=1;
  int ponto;
  bool trocou=true;

  classe->Assign(image); //classe armazena apenas os rotulos
  for(int y=0;y<classe->Height;y++)
    {
      px=(Byte *)classe->ScanLine[y];
      for(int x=0;x<classe->Width;x++)//zerar imagem classe
        px[x]=0;
    }

  for(int y=1;y<classe->Height-1;y++)
    {
      ptrx=(Byte *)image->ScanLine[y];  //verifica a conectividade
      ptry=(Byte *)image->ScanLine[y-1]; //entre os pontos
                          //e atribui um rotulo ao pixel processado
      px=(Byte *)classe->ScanLine[y];
      py=(Byte *)classe->ScanLine[y-1];

      for(int x=1;x<classe->Width-1;x++)
        {
          if(ptrx[x]==255)
            {
              if((ptrx[x-1]==255)&&(ptry[x]==255))
                px[x-1]=py[x];

              if(ptrx[x-1]==255)    //f(x-1,y)
                px[x]=px[x-1];
              if(ptry[x-1]==255) //f(x-1,y-x)
                px[x]= py[x-1];
              if(ptry[x]==255)   //f(x,y-1)
                px[x]=py[x];
              if((ptrx[x-1]!=255)&&(ptry[x-1]!=255)&&(ptry[x]!=255))
                {
                  objeto++;     //nova classe
                  px[x]=objeto;
                }
            }
        }
    }

 while(trocou)   //verifica pontos do mesmo objeto com rotulos diferentes
    {        //e atribui um unico rotulo para esses pixels enquanto
      trocou=false; // houver objetos com pontos com rotulos diferentes, o algoritmo
      for(int y=1;y<classe->Height-1;y++)    //nao para
          px=(Byte *)classe->ScanLine[y];
          py=(Byte *)classe->ScanLine[y-1];
          ptry=(Byte *)classe->ScanLine[y+1];
```

```
for(int x=1;x<classe->Width-1;x++)
  {
   if(px[x]!=0)
    {
      if(py[x-1]!=0)
        {
          if(py[x-1]>px[x])      //f(x-1,y-1)
            {
              py[x-1]=px[x];
              trocou=true;
            }
          else if(py[x-1]<px[x])
            {
              px[x]=py[x-1];
              trocou=true;
            }
        }
      if(py[x]!=0)               //f(x,y-1)
        {
          if(py[x]>px[x])
            {
              py[x]=px[x];
              trocou=true;
            }
          else if(py[x]>px[x])
            {
              px[x]=py[x];
              trocou=true;
            }
        }
      if(py[x+1]!=0)            //f(x+1,y-1)
        {
          if(py[x+1]>px[x])
            {
              py[x+1]=px[x];
              trocou=true;
            }
          else if(py[x+1]>px[x])
            {
              px[x]=py[x-1];
              trocou=true;
            }  }

     if(px[x-1]!=0)            //f(x-1,y)
        {
          if(px[x-1]>px[x])
            {
              px[x-1]=px[x];
              trocou=true;
            }
          else if(px[x-1]>px[x])
            {
              px[x]=px[x-1];
              trocou=true;
            }
        }
```

```
                if(px[x+1]!=0)          //f(x+1,y)
                  {
                    if(px[x+1]>px[x])
                      {
                        px[x+1]=px[x];
                        trocou=true;
                      }
                    else if(px[x+1]>px[x])
                      {
                        px[x]=px[x+1];
                        trocou=true;
                      }
                  }
                if(ptry[x-1]!=0)        //f(x-1,y+1)
                  {
                    if(ptry[x-1]>px[x])
                      {
                        ptry[x-1]=px[x];
                        trocou=true;
                      }
                    else if(ptry[x-1]>px[x])
                      {
                        px[x]=ptry[x-1];
                        trocou=true;
                      }
                  }
                if(ptry[x]!=0)          //f(x,y+1)
                  {
                    if(ptry[x]>px[x])
                      {
                        ptry[x]=px[x];
                        trocou=true;
                      }
                    else if(ptry[x]>px[x])
                      {
                        px[x]=ptry[x];
                        trocou=true;
                      }  }
                if(ptry[x+1]!=0)       //f(x+1,y+1)
                  {
                    if(ptry[x+1]>px[x])
                      {
                        ptry[x]=px[x];
                        trocou=true;
                      }
                    else if(ptry[x+1]>px[x])
                      {
                        px[x]=ptry[x];
                        trocou=true;
                      }
                  }
                }//fim do if(px[x]!=0)
              }//fim do for x
            }//fim do for y
          } //fim do while
image->Assign(classe);//retorna a imagem pelo ponteiro  }
```

```
//---------------------------------------------------------------------------------------------------------------------
Rede Neural, topologia 9:3
//---------------------------------------------------------------------------------------------------------------------
int __fastcall rna(float rg, float rb, float gb)//caracteristicas
{
   float input[3]={rg,rb,gb};//vetor de entrada
   float layer1[9]={0,0,0,0,0,0};    //layer 1
   float layer2[3]={0,0,0};    //output layer
   float temp;
   //pesos
   float wi[9][3]={-0.0465,-0.0098,-0.0777,-0.0080,0.0096,-0.2061, 0.0404,-0.0527,
   -0.1836,-0.0428,-0.0174,-0.1721, 0.0040,-0.0051, 0.0283,-0.0172,0.0315,
    0.2157, 0.2043,0.2130,0.1563, 0.0031,-0.0015,-0.1753,-0.0683,0.0282,0.4488};
  float w21[3][9]={-0.6047,-0.2241,0.4303,0.1118,-0.5192,0.2387,-0.1848,-0.1267,
-0.6527,-0.0850,-0.0951,0.1239,-0.2730,0.0945,0.1814,0.5961,0.2187,0.5635,
   -0.0603,-0.4255,-0.1424,0.5624,0.4944,-0.1822,0.3022, 0.3247,-0.0654};
   //bias
float wb1[9]={-18.2172,3.6857,5.111,-1.8573,-3.22,-5.422,-2.12,3.332,-7.183};
   float wb2[3]={-0.3214,-0.3606,0.6893};
   for(int l=0;l<3;l++)
      {
         for(int c=0;c<9;c++)
            {
               layer1[l] = layer1[l]+(input[c]*wi[l][c]);   //Layer 1
            }
         layer1[l] = layer1[l]+(1*wb1[l]);
         layer1[l] = tansig(layer1[l]);
      }
   for(int l=0;l<3;l++)
      {
         for(int c=0;c<9;c++)
            {
               layer2[l] = layer2[l]+(layer1[c]*w21[l][c]);  //output layer
            }
         layer2[l] = layer2[l]+(1*wb2[l]);
      }
   if((layer2[0]>layer2[1])&&(layer2[0]>layer2[2]))
      return(1);  //granulaçao
   else if((layer2[1]>layer2[0])&&(layer2[1]>layer2[2]))
      return(2); //exsudação
   else
      return(3);  //necrose //nao foi treinado
}

//---------------------------------------------------------------------------------------------------------------------
//Função sigmoidal
//---------------------------------------------------------------------------------------------------------------------
float __fastcall tansig(float n)
   { return(2/(1+expl(-2*n))-1); }
//---------------------------------------------------------------------------------------------------------------------
```

```
//---------------------------------------------------------------------------------------------------------------------
Algoritmo que calcula o erro da segmentação em relação à segmentação manual
//---------------------------------------------------------------------------------------------------------------------
float __fastcall error_metric(Graphics::TBitmap *ima,Graphics::TBitmap *imb )
{
   float result;
   float areaA=0,areaXor=0;
   Byte *ptra, *ptrb;

   for(int y=0;y<ima->Height;y++)
     {
        ptrc=(Byte *)imc->ScanLine[y];
        for(int x=0;x<ima->Width;x++)
          {
             ptrc[x]=0;

          }
     }
   for(int y=0;y<ima->Height;y++)
     {
        ptra=(Byte *)ima->ScanLine[y];
        ptrb=(Byte *)imb->ScanLine[y];
        for(int x=0;x<ima->Width;x++)
          {
             if(ptra[x]==255)
               areaA++;

             if(ptra[x]!=ptrb[x])
               areaXor++;
          }
     }
    result=areaXor/areaA;
    return result;
}
//---------------------------------------------------------------------------------------------------------------------
```