https://blog.csdn.net/junpengxue/article/details/79958651

[**图像的膨胀与腐蚀、细化**](http://www.cnblogs.com/slysky/archive/2011/10/16/2214015.html)

原理：在特殊领域运算形式——结构元素（Sturcture Element），在每个像素位置上与二值图像对应的区域进行特定的逻辑运算。运算结构是输出图像的相应像素。运算效果取决于结构元素大小内容以及逻辑运算性质。

结构元素：膨胀和腐蚀操作的最基本组成部分，用于测试输出图像，通常要比待处理的图像小还很多。二维平面结构元素由一个数值为0或1的矩阵组成。结构元素的原点指定了图像中需要处理的像素范围，结构元素中数值为1的点决定结构元素的邻域像素在进行膨胀或腐蚀操作时是否需要参与计算。

先来定义一些基本符号和关系。

***1.*****元素**

设有一幅图象*X*，若点*a*在*X*的区域以内，则称*a*为*X*的元素，记作*a*∈*X*，如图*6.1*所示。

***2.******B*包含于*X***

设有两幅图象*B*，*X*。对于*B*中所有的元素*ai*，都有*ai*∈*X*，则称*B*包含于*(included in)X*，记作*BX*，如图*6.2*所示。

***3.******B*击中*X***

设有两幅图象*B*，*X*。若存在这样一个点，它即是*B*的元素，又是*X*的元素，则称*B*击中*(hit)X*，记作*B*↑*X*，如图*6.3*所示。

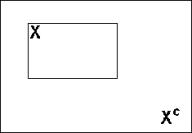
***4.******B*不击中*X***

设有两幅图象*B*，*X*。若不存在任何一个点，它即是*B*的元素，又是*X*的元素，即*B*和*X*的交集是空，则称*B*不击中*(miss)X*，记作*B*∩*X=*Ф；其中∩是集合运算相交的符号，Ф表示空集。如图*6.4*所示。

|  |  |
| --- | --- |
| **图6.1     元素** | **图6.2     包含** |
| **图6.3     击中** | **图6.4     不击中** |

***5.*****补集**

设有一幅图象*X*，所有*X*区域以外的点构成的集合称为*X*的补集，记作*Xc*，如图*6.5*所示。显然，如果*B*∩*X=*Ф，则*B*在*X*的补集内，即*BXc*。

**

**图6.5     补集的示意图**

***6.*****结构元素**

设有两幅图象*B*，*X*。若*X*是被处理的对象，而*B*是用来处理*X*的，则称*B*为结构元素*(structure element)*，又被形象地称做刷子。结构元素通常都是一些比较小的图象。

***7.*****对称集**

设有一幅图象*B*，将*B*中所有元素的坐标取反，即令*(x*，*y)*变成*(-x*，*-y)*，所有这些点构成的新的集合称为*B*的对称集，记作*Bv*，如图*6.6*所示。

***8.*****平移**

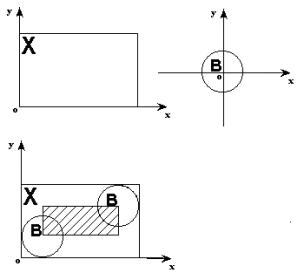
设有一幅图象*B*，有一个点*a(x0,y0)*，将*B*平移*a*后的结果是，把*B*中所有元素的横坐标加*x0*，纵坐标加*y0*，即令*(x*，*y)*变成*(x+x0*，*y+y0)*，所有这些点构成的新的集合称为*B*的平移，记作*Ba*，如图*6.7*所示。

|  |  |
| --- | --- |
| **图6.6     对称集的示意图** | **图6.7     平移的示意图** |

好了，介绍了这么多基本符号和关系，现在让我们应用这些符号和关系，看一下形态学的基本运算。

***6.1 腐蚀***

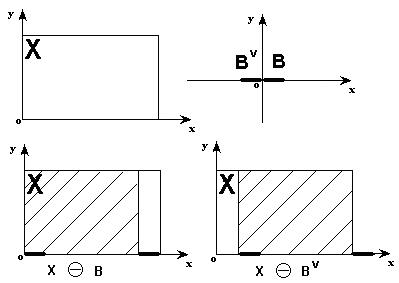
把结构元素*B*平移*a*后得到*Ba*，若*Ba*包含于*X*，我们记下这个*a*点，所有满足上述条件的*a*点组成的集合称做*X*被*B*腐蚀*(Erosion)*的结果。用公式表示为：*E(X)={a| Ba X}=X  B*，如图*6.8*所示。

**

**图6.8     腐蚀的示意图**

图*6.8*中*X*是被处理的对象，*B*是结构元素。不难知道，对于任意一个在阴影部分的点*a*，*Ba*包含于*X*，所以*X*被*B*腐蚀的结果就是那个阴影部分。阴影部分在*X*的范围之内，且比*X*小，就象*X*被剥掉了一层似的，这就是为什么叫腐蚀的原因。

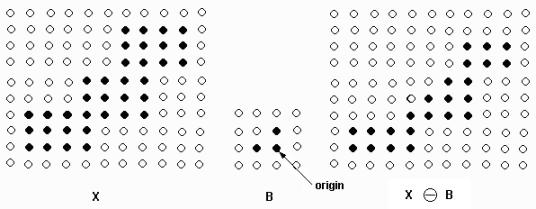
值得注意的是，上面的*B*是对称的，即*B*的对称集*Bv=B*，所以*X*被*B*腐蚀的结果和*X*被*Bv*腐蚀的结果是一样的。如果*B*不是对称的，让我们看看图*6.9*，就会发现*X*被*B*腐蚀的结果和*X*被*Bv*腐蚀的结果不同。

**

**图6.9     结构元素非对称时，腐蚀的结果不同**

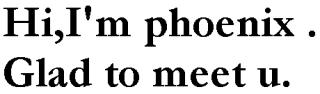
图*6.8*和图*6.9*都是示意图，让我们来看看实际上是怎样进行腐蚀运算的。

在图*6.10*中，左边是被处理的图象*X(*二值图象，我们针对的是黑点*)*，中间是结构元素*B*，那个标有*origin*的点是中心点，即当前处理元素的位置，我们在介绍模板操作时也有过类似的概念。腐蚀的方法是，拿*B*的中心点和*X*上的点一个一个地对比，如果*B*上的所有点都在*X*的范围内，则该点保留，否则将该点去掉；右边是腐蚀后的结果。可以看出，它仍在原来*X*的范围内，且比*X*包含的点要少，就象*X*被腐蚀掉了一层。

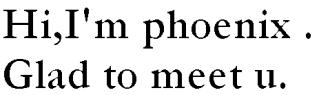
**

**图6.10   腐蚀运算**

图*6.11*为原图，图*6.12*为腐蚀后的结果图，能够很明显地看出腐蚀的效果。

**

**图6.11    原图**

**

**图6.12   腐蚀后的结果图**

下面的这段程序，实现了上述的腐蚀运算，针对的都是黑色点。参数中有一个*BOOL*变量，为真时，表示在水平方向进行腐蚀运算，即结构元素*B*为**；否则在垂直方向上进行腐蚀运算，即结构元素*B*为**。



复制代码

BOOL Erosion(HWND hWnd,BOOL Hori)

{

DWORD OffBits,BufSize;

LPBITMAPINFOHEADER lpImgData;

LPSTR lpPtr;

HLOCAL hTempImgData;

LPBITMAPINFOHEADER lpTempImgData;

LPSTR lpTempPtr;

HDC hDc;

HFILE hf;

LONG x,y;

unsigned char num;

int i;

//为了处理方便，仍采用256级灰度图，不过只用调色板中0和255两项

if( NumColors!=256){

MessageBox(hWnd,"Must be a mono bitmap with grayscale palette!",

"Error Message",MB\_OK|MB\_ICONEXCLAMATION);

return FALSE;

}

OffBits=bf.bfOffBits-sizeof(BITMAPFILEHEADER);

//BufSize为缓冲区大小

BufSize=OffBits+bi.biHeight\*LineBytes;

//为新的缓冲区分配内存

if((hTempImgData=LocalAlloc(LHND,BufSize))==NULL)

{

MessageBox(hWnd,"Error alloc memory!","Error Message",

MB\_OK|MB\_ICONEXCLAMATION);

return FALSE;

}

lpImgData=(LPBITMAPINFOHEADER)GlobalLock(hImgData);

lpTempImgData=(LPBITMAPINFOHEADER)LocalLock(hTempImgData);

//拷贝头信息和位图数据

memcpy(lpTempImgData,lpImgData,BufSize);

if(Hori)

{

//在水平方向进行腐蚀运算

for(y=0;y<bi.biHeight;y++){

//lpPtr指向原图数据，lpTempPtr指向新图数据

lpPtr=(char \*)lpImgData+(BufSize-LineBytes-y\*LineBytes)+1;

lpTempPtr=(char\*)lpTempImgData+

(BufSize-LineBytes-y\*LineBytes)+1;

for(x=1;x<bi.biWidth-1;x++){

//注意为防止越界，x的范围从1到宽度-2

num=(unsigned char)\*lpPtr;

if (num==0){ //因为腐蚀掉的是黑点，所以只对黑点处理

\*lpTempPtr=(unsigned char)0; //先置成黑点

for(i=0;i<3;i++){

num=(unsigned char)\*(lpPtr+i-1);

if(num==255){

//自身及上下邻居中若有一个不是黑点，则将该点腐

//蚀成白点

\*lpTempPtr=(unsigned char)255;

break;

}

}

}

//原图中就是白点的，新图中仍是白点

else \*lpTempPtr=(unsigned char)255;

//指向下一个象素

lpPtr++;

lpTempPtr++;

}

}

}

else{

//在垂直方向进行腐蚀运算

for(y=1;y<bi.biHeight-1;y++){ //注意为防止越界，y的范围从1到高度-2

//lpPtr指向原图数据，lpTempPtr指向新图数据

lpPtr=(char \*)lpImgData+(BufSize-LineBytes-y\*LineBytes);

lpTempPtr=(char \*)lpTempImgData+(BufSize-LineBytes-y\*LineBytes);

for(x=0;x<bi.biWidth;x++){

num=(unsigned char)\*lpPtr;

if (num==0){ //因为腐蚀掉的是黑点，所以只对黑点处理

\*lpTempPtr=(unsigned char)0; //先置成黑点

for(i=0;i<3;i++){

num=(unsigned char)\*(lpPtr+(i-1)\*LineBytes);

if(num==255){

//自身及上下邻居中若有一个不是黑点，则将该点腐

//蚀成白点

\*lpTempPtr=(unsigned char)255;

break;

}

}

}

//原图中就是白点的，新图中仍是白点

else \*lpTempPtr=(unsigned char)255;

//指向下一个象素

lpPtr++;

lpTempPtr++;

}

}

}

if(hBitmap!=NULL)

DeleteObject(hBitmap);

hDc=GetDC(hWnd);

//产生新的位图

hBitmap=CreateDIBitmap(hDc,(LPBITMAPINFOHEADER)lpTempImgData,

(LONG)CBM\_INIT,

(LPSTR)lpTempImgData+

sizeof(BITMAPINFOHEADER)+

NumColors\*sizeof(RGBQUAD),

(LPBITMAPINFO)lpTempImgData, DIB\_RGB\_COLORS);

//起不同的结果文件名

if(Hori)

hf=\_lcreat("c:\\herosion.bmp",0);

else

hf=\_lcreat("c:\\verosion.bmp",0);

\_lwrite(hf,(LPSTR)&bf,sizeof(BITMAPFILEHEADER));

\_lwrite(hf,(LPSTR)lpTempImgData,BufSize);

\_lclose(hf);

//释放内存及资源

ReleaseDC(hWnd,hDc);

LocalUnlock(hTempImgData);

LocalFree(hTempImgData);

GlobalUnlock(hImgData);

return TRUE;

}

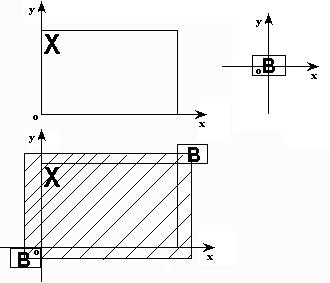
复制代码

膨胀

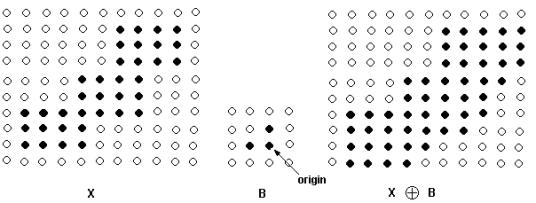
膨胀*(dilation)*可以看做是腐蚀的对偶运算，其定义是：把结构元素*B*平移*a*后得到*Ba*，若*Ba*击中*X*，我们记下这个*a*点。所有满足上述条件的*a*点组成的集合称做*X*被*B*膨胀的结果。用公式表示为：*D(X)={a | Ba*↑*X}=X  B*，如图*6.13*所示。图*6.13*中*X*是被处理的对象，*B*是结构元素，不难知道，对于任意一个在阴影部分的点*a*，*Ba*击中*X*，所以*X*被*B*膨胀的结果就是那个阴影部分。阴影部分包括*X*的所有范围，就象*X*膨胀了一圈似的，这就是为什么叫膨胀的原因。

同样，如果*B*不是对称的，*X*被*B*膨胀的结果和*X*被*Bv*膨胀的结果不同。

让我们来看看实际上是怎样进行膨胀运算的。在图*6.14*中，左边是被处理的图象*X(*二值图象，我们针对的是黑点*)*，中间是结构元素*B*。膨胀的方法是，拿*B*的中心点和*X*上的点及*X*周围的点一个一个地对，如果*B*上有一个点落在*X*的范围内，则该点就为黑；右边是膨胀后的结果。可以看出，它包括*X*的所有范围，就象*X*膨胀了一圈似的。

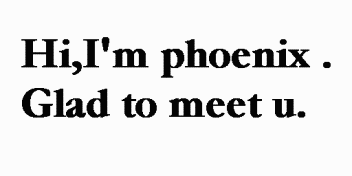
**

**图6.13   膨胀的示意图**

******

**图6.14   膨胀运算**

图*6.15*为图*6.11*膨胀后的结果图，能够很明显的看出膨胀的效果。

**

**图6.15   图*6.11*膨胀后的结果图**

下面的这段程序，实现了上述的膨胀运算，针对的都是黑色点。参数中有一个*BOOL*变量，为真时，表示在水平方向进行膨胀运算，即结构元素*B*为**；否则在垂直方向上进行膨胀运算，即结构元素*B*为**。



复制代码

BOOL Dilation(HWND hWnd,BOOL Hori)

{

DWORD OffBits,BufSize;

LPBITMAPINFOHEADER lpImgData;

LPSTR lpPtr;

HLOCAL hTempImgData;

LPBITMAPINFOHEADER lpTempImgData;

LPSTR lpTempPtr;

HDC hDc;

HFILE hf;

LONG x,y;

unsigned char num;

int i;

//为了处理的方便，仍采用256级灰度图，不过只调色板中0和255两项

if( NumColors!=256){

MessageBox(hWnd,"Must be a mono bitmap with grayscale palette!",

"Error Message",MB\_OK|MB\_ICONEXCLAMATION);

return FALSE;

}

OffBits=bf.bfOffBits-sizeof(BITMAPFILEHEADER);

//BufSize为缓冲区大小

BufSize=OffBits+bi.biHeight\*LineBytes;

//为新的缓冲区分配内存

if((hTempImgData=LocalAlloc(LHND,BufSize))==NULL)

{

MessageBox(hWnd,"Error alloc memory!","Error Message",

MB\_OK|MB\_ICONEXCLAMATION);

return FALSE;

}

lpImgData=(LPBITMAPINFOHEADER)GlobalLock(hImgData);

lpTempImgData=(LPBITMAPINFOHEADER)LocalLock(hTempImgData);

//拷贝头信息和位图数据

memcpy(lpTempImgData,lpImgData,BufSize);

if(Hori)

{

//在水平方向进行膨胀运算

for(y=0;y<bi.biHeight;y++){

//lpPtr指向原图数据，lpTempPtr指向新图数据

lpPtr=(char \*)lpImgData+(BufSize-LineBytes-y\*LineBytes)+1;

lpTempPtr=(char\*)lpTempImgData+

(BufSize-LineBytes-y\*LineBytes)+1;

for(x=1;x<bi.biWidth-1;x++){

//注意为防止越界，x的范围从1到宽度-2

num=(unsigned char)\*lpPtr;

//原图中是黑点的，新图中肯定也是，所以要考虑的是那些原图

//中的白点，看是否有可能膨胀成黑点

if (num==255){

\*lpTempPtr=(unsigned char)255; //先置成白点

for(i=0;i<3;i++){

num=(unsigned char)\*(lpPtr+i-1);

//只要左右邻居中有一个是黑点，就膨胀成黑点

if(num==0){

\*lpTempPtr=(unsigned char)0;

break;

}

}

}

//原图中就是黑点的，新图中仍是黑点

else \*lpTempPtr=(unsigned char)0;

//指向下一个象素

lpPtr++;

lpTempPtr++;

}

}

}

else{

//在垂直方向进行腐蚀运算

for(y=1;y<bi.biHeight-1;y++){ //注意为防止越界，y的范围从1到高度-2

lpPtr=(char \*)lpImgData+(BufSize-LineBytes-y\*LineBytes);

lpTempPtr=(char \*)lpTempImgData+(BufSize-LineBytes-y\*LineBytes);

for(x=0;x<bi.biWidth;x++){

num=(unsigned char)\*lpPtr;

if (num==255){

\*lpTempPtr=(unsigned char)255;

for(i=0;i<3;i++){

num=(unsigned char)\*(lpPtr+(i-1)\*LineBytes);

//只要上下邻居中有一个是黑点，就膨胀成黑点

if(num==0){

\*lpTempPtr=(unsigned char)0;

break;

}

}

}

else \*lpTempPtr=(unsigned char)0;

lpPtr++;

lpTempPtr++;

}

}

}

if(hBitmap!=NULL)

DeleteObject(hBitmap);

hDc=GetDC(hWnd);

//产生新的位图

hBitmap=CreateDIBitmap(hDc,(LPBITMAPINFOHEADER)lpTempImgData,

(LONG)CBM\_INIT,

(LPSTR)lpTempImgData+

sizeof(BITMAPINFOHEADER)+

NumColors\*sizeof(RGBQUAD),

(LPBITMAPINFO)lpTempImgData,

DIB\_RGB\_COLORS);

//起不同的结果文件名

if(Hori)

hf=\_lcreat("c:\\hdilation.bmp",0);

else

hf=\_lcreat("c:\\vdilation.bmp",0);

\_lwrite(hf,(LPSTR)&bf,sizeof(BITMAPFILEHEADER));

\_lwrite(hf,(LPSTR)lpTempImgData,BufSize);

\_lclose(hf);

//释放内存及资源

ReleaseDC(hWnd,hDc);

LocalUnlock(hTempImgData);

LocalFree(hTempImgData);

GlobalUnlock(hImgData);

return TRUE;

}

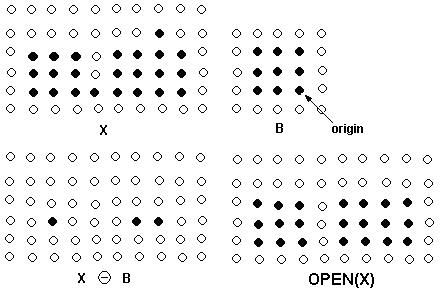
复制代码

腐蚀运算和膨胀运算互为对偶的，用公式表示为*(X  B)c=(Xc  B)*，即*X*被*B*腐蚀后的补集等于*X*的补集被*B*膨胀。这句话可以形象的理解为：河岸的补集为河面，河岸的腐蚀等价于河面的膨胀。你可以自己举个例子来验证一下这个关系。在有些情况下，这个对偶关系是非常有用的。例如：某个图象处理系统用硬件实现了腐蚀运算，那么不必再另搞一套膨胀的硬件，直接利用该对偶就可以实现了。

开

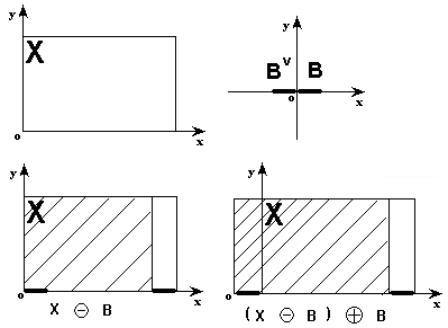
先腐蚀后膨胀称为开*(open)*，即*OPEN(X)=D(E(X))*。

让我们来看一个开运算的例子*(*见图*6.16)*：

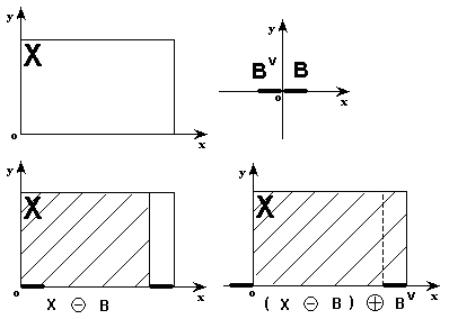
**

**图*6.16*开运算**

在图*16*上面的两幅图中，左边是被处理的图象*X(*二值图象，我们针对的是黑点*)*，右边是结构元素*B*，下面的两幅图中左边是腐蚀后的结果；右边是在此基础上膨胀的结果。可以看到，原图经过开运算后，一些孤立的小点被去掉了。一般来说，开运算能够去除孤立的小点，毛刺和小桥*(*即连通两块区域的小点*)*，而总的位置和形状不变。这就是开运算的作用。要注意的是，如果*B*是非对称的，进行开运算时要用*B*的对称集*Bv*膨胀，否则，开运算的结果和原图相比要发生平移。图*6.17*和图*6.18*能够说明这个问题。

**

**图*6.17*用*B*膨胀后，结果向左平移了**

******

**图6.18   用*Bv*膨胀后位置不变**

图*6.17*是用*B*膨胀的，可以看到，*OPEN(X)*向左平移了。图*18*是用*Bv*膨胀的，可以看到，总的位置和形状不变。

图*6.19*为图*6.11*经过开运算后的结果。

**

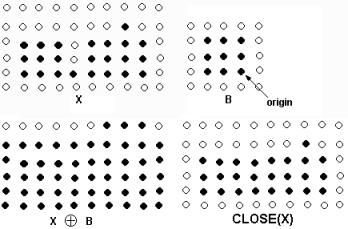
**图6.19   图*6.11*经过开运算后的结果**

开运算的源程序可以很容易的根据上面的腐蚀，膨胀程序得到，这里就不给出了。

闭

先膨胀后腐蚀称为闭*(close)*，即*CLOSE(X)=E(D(X))*。

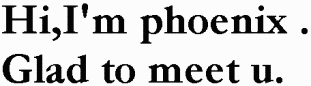
让我们来看一个闭运算的例子*(*见图*6.20)*：

**

**图6.20   闭运算**

在图*6.20*上面的两幅图中，左边是被处理的图象*X(*二值图象，我们针对的是黑点*)*，右边是结构元素*B*，下面的两幅图中左边是膨胀后的结果，右边是在此基础上腐蚀的结果可以看到，原图经过闭运算后，断裂的地方被弥合了。一般来说，闭运算能够填平小湖*(*即小孔*)*，弥合小裂缝，而总的位置和形状不变。这就是闭运算的作用。同样要注意的是，如果*B*是非对称的，进行闭运算时要用*B*的对称集*Bv*膨胀，否则，闭运算的结果和原图相比要发生平移。

图*6.21*为图*6.11*经过闭运算后的结果。

**

**图6.21   图*.611*经过闭运算后的结果**

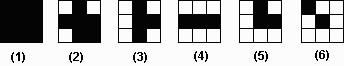
闭运算的源程序可以很容易的根据上面的膨胀，腐蚀程序得到，这里就不给出了。

你大概已经猜到了，开和闭也是对偶运算，的确如此。用公式表示为*(OPEN(X))c=CLOSE((Xc))*，或者*(CLOSE(X))c =OPEN((Xc))*。即*X*开运算的补集等于*X*的补集的闭运算，或者*X*闭运算的补集等于*X*的补集的开运算。这句话可以这样来理解：在两个小岛之间有一座小桥，我们把岛和桥看做是处理对象*X*，则*X*的补集为大海。如果涨潮时将小桥和岛的外围淹没*(*相当于用尺寸比桥宽大的结构元素对*X*进行开运算*)*，那么两个岛的分隔，相当于小桥两边海域的连通*(*对*Xc*做闭运算*)*。

细化

细化*(thinning)*算法有很多，我们在这里介绍的是一种简单而且效果很好的算法，用它就能够实现从文本抽取骨架的功能。我们的对象是白纸黑字的文本，但在程序中为了处理的方便，还是采用*256*级灰度图，不过只用到了调色板中*0*和*255*两项。

所谓细化，就是从原来的图中去掉一些点，但仍要保持原来的形状。实际上，是保持原图的骨架。所谓骨架，可以理解为图象的中轴，例如一个长方形的骨架是它的长方向上的中轴线；正方形的骨架是它的中心点；圆的骨架是它的圆心，直线的骨架是它自身，孤立点的骨架也是自身。文本的骨架嘛，前言中的例子显示的很明白。那么怎样判断一个点是否能去掉呢？显然，要根据它的八个相邻点的情况来判断，我们给几个例子*(*如图*6.22*所示*)*。

**

**图6.22   根据某点的八个相邻点的情况来判断该点是否能删除**

图*6.22*中，*(1)*不能删，因为它是个内部点，我们要求的是骨架，如果连内部点也删了，骨架也会被掏空的；*(2)*不能删，和*(1)*是同样的道理；*(3)*可以删，这样的点不是骨架；*(4)*不能删，因为删掉后，原来相连的部分断开了；*(5)*可以删，这样的点不是骨架；*(6)*不能删，因为它是直线的端点，如果这样的点删了，那么最后整个直线也被删了，剩不下什么；*(7)*不能删，因为孤立点的骨架就是它自身。

总结一下，有如下的判据：*(1)*内部点不能删除；*(2)*孤立点不能删除；*(3)*直线端点不能删除；*(4)*如果*P*是边界点，去掉*P*后，如果连通分量不增加，则*P*可以删除。

我们可以根据上述的判据，事先做出一张表，从*0*到*255*共有*256*个元素，每个元素要么是*0*，要么是*1*。我们根据某点*(*当然是要处理的黑色点了*)*的八个相邻点的情况查表，若表中的元素是*1*，则表示该点可删，否则保留。

查表的方法是，设白点为*1*，黑点为*0*；左上方点对应一个*8*位数的第一位*(*最低位*)*，正上方点对应第二位，右上方点对应的第三位，左邻点对应第四位，右邻点对应第五位，左下方点对应第六位，正下方点对应第七位，右下方点对应的第八位，按这样组成的*8*位数去查表即可。例如上面的例子中*(1)*对应表中的第*0*项，该项应该为*0*；*(2)*对应*37*，该项应该为*0*；*(3)*对应*173*，该项应该为*1*；*(4)*对应*231*，该项应该为*0*；*(5)*对应*237*，该项应该为*1*；*(6)*对应*254*，该项应该为*0*；*(7)*对应*255*，该项应该为*0*。

这张表我已经替大家做好了，可花了我不少时间呢！

*static int erasetable[256]={*

                                         0,0,1,1,0,0,1,1,          1,1,0,1,1,1,0,1,

                                   1,1,0,0,1,1,1,1,             0,0,0,0,0,0,0,1,

                                          0,0,1,1,0,0,1,1,             1,1,0,1,1,1,0,1,

                                          1,1,0,0,1,1,1,1,             0,0,0,0,0,0,0,1,

                                          1,1,0,0,1,1,0,0,             0,0,0,0,0,0,0,0,

                                          0,0,0,0,0,0,0,0,             0,0,0,0,0,0,0,0,

                                          1,1,0,0,1,1,0,0,             1,1,0,1,1,1,0,1,

                                   0,0,0,0,0,0,0,0,             0,0,0,0,0,0,0,0,

                           0,0,1,1,0,0,1,1,             1,1,0,1,1,1,0,1,

                                          1,1,0,0,1,1,1,1,             0,0,0,0,0,0,0,1,

                                          0,0,1,1,0,0,1,1,             1,1,0,1,1,1,0,1,

                                          1,1,0,0,1,1,1,1,             0,0,0,0,0,0,0,0,

                                          1,1,0,0,1,1,0,0,             0,0,0,0,0,0,0,0,

                                1,1,0,0,1,1,1,1,             0,0,0,0,0,0,0,0,

                                          1,1,0,0,1,1,0,0,             1,1,0,1,1,1,0,0,

                                   1,1,0,0,1,1,1,0,             1,1,0,0,1,0,0,0

                                     };

有了这张表，算法就很简单了，每次对一行一行的将整个图象扫描一遍，对于每个点*(*不包括边界点*)*，计算它在表中对应的索引，若为*0*，则保留，否则删除该点。如果这次扫描没有一个点被删除，则循环结束，剩下的点就是骨架点，如果有点被删除，则进行新的一轮扫描，如此反复，直到没有点被删除为止。

实际上，该算法有一些缺陷。举个简单的例子，有一个黑色矩形，如图*6.23*所示。

图*6.23*经过细化后，我们预期的结果是一条水平直线，且位于该黑色矩形的中心。实际的结果确实是一条水平直线，但不是位于黑色矩形的中心，而是最下面的一条边。

为什么会这样，我们来分析一下：在从上到下，从左到右的扫描过程中，我们遇到的第一个黑点就是黑色矩形的左上角点，经查表，该点可以删。下一个点是它右边的点，经查表，该点也可以删，如此下去，整个一行被删了。每一行都是同样的情况，所以都被删除了。到了最后一行时，黑色矩形已经变成了一条直线，最左边的黑点不能删，因为它是直线的端点，它右边的点也不能删，因为如果删除，直线就断了，如此下去，直到最右边的点，也不能删，因为它是直线的右端点。所以最下面的一条边保住了，但这并不是我们希望的结果。

解决的办法是，在每一行水平扫描的过程中，先判断每一点的左右邻居，如果都是黑点，则该点不做处理。另外，如果某个黑点被删除了，那么跳过它的右邻居，处理下一个点。这样就避免了上述的问题。

|  |  |
| --- | --- |
| **图6.23  黑色矩形** | **图6.24  图*6.23*细化后的结果** |

解决了上面的问题，我们来看看处理后的结果，如图*6.24*所示。这次变成一小段竖线了，还是不对，是不是很沮丧？别着急，让我们再来分析一下：在上面的算法中，我们遇到的第一个能删除的点就是黑色矩形的左上角点；第二个是第一行的最右边的点，即黑色矩形的右上角点；第三个是第二行的最左边的点；第四个是第二行的最右边的点；*……*；整个图象处理这样一次后，宽度减少*2*。每次都是如此，直到剩最中间一列，就不能再删了。为什么会这样呢？原因是这样的处理过程只实现了水平细化，如果在每一次水平细化后，再进行一次垂直方向的细化*(*只要把上述过程的行列换一下*)*，就可以了。

这样一来，每处理一次，删除点的顺序变成：*(*先是水平方向扫描*)*第一行最左边的点；第一行最右边的点；第二行最左边的点；第二行最右边的点；*……*最后一行最左边的点；最后一行最右边的点；*(*然后是垂直方向扫描*)*第二列最上边的点*(*因为第一列最上边的点已被删除*)*；第二列最下边的点；第三列最上边的点；第三列最下边的点；*……*倒数第二列最上边的点*(*因为倒数第一列最上边的点已被删除*)*；倒数第二列最下边的点。我们发现，刚好剥掉了一圈，这也正是细化要做的事。实际的结果也验证了我们的想法。

以下是源程序，黑体字部分是值得注意的地方。



复制代码

BOOL Thinning(HWND hWnd)

{

DWORD OffBits,BufSize;

LPBITMAPINFOHEADER lpImgData;

LPSTR lpPtr;

HLOCAL hTempImgData;

LPBITMAPINFOHEADER lpTempImgData;

LPSTR lpTempPtr;

HDC hDc;

HFILE hf;

LONG x,y;

int num;

BOOL Finished;

int nw,n,ne,w,e,sw,s,se;

//为了处理的方便，仍采用256级灰度图，不过只用调色板中0和255两项

if( NumColors!=256){

MessageBox(hWnd,"Must be a mono bitmap with grayscale palette!",

"Error Message",MB\_OK|MB\_ICONEXCLAMATION);

return FALSE;

}

OffBits=bf.bfOffBits-sizeof(BITMAPFILEHEADER);

//BufSize为缓冲区大小

BufSize=OffBits+bi.biHeight\*LineBytes;

//为新的缓冲区分配内存

if((hTempImgData=LocalAlloc(LHND,BufSize))==NULL)

{

MessageBox(hWnd,"Error alloc memory!","Error Message",

MB\_OK|MB\_ICONEXCLAMATION);

return FALSE;

}

lpImgData=(LPBITMAPINFOHEADER)GlobalLock(hImgData);

lpTempImgData=(LPBITMAPINFOHEADER)LocalLock(hTempImgData);

//拷贝头信息和位图数据

memcpy(lpTempImgData,lpImgData,BufSize);

//结束标志置成假

Finished=FALSE;

while(!Finished){ //还没有结束

//结束标志置成假

Finished=TRUE;

//先进行水平方向的细化

for (y=1;y<bi.biHeight-1;y++){ //注意为防止越界，y的范围从1到高度-2

//lpPtr指向原图数据，lpTempPtr指向新图数据

lpPtr=(char \*)lpImgData+(BufSize-LineBytes-y\*LineBytes);

lpTempPtr=(char \*)lpTempImgData+(BufSize-LineBytes-y\*LineBytes);

x=1; //注意为防止越界，x的范围从1到宽度-2

while(x<bi.biWidth-1){

if(\*(lpPtr+x)==0){ //是黑点才做处理

w=(unsigned char)\*(lpPtr+x-1); //左邻点

e=(unsigned char)\*(lpPtr+x+1); //右邻点

if( (w==255)|| (e==255)){

//如果左右两个邻居中至少有一个是白点才处理

nw=(unsigned char)\*(lpPtr+x+LineBytes-1); //左上邻点

n=(unsigned char)\*(lpPtr+x+LineBytes); //上邻点

ne=(unsigned char)\*(lpPtr+x+LineBytes+1); //右上邻点

sw=(unsigned char)\*(lpPtr+x-LineBytes-1); //左下邻点

s=(unsigned char)\*(lpPtr+x-LineBytes); //下邻点

se=(unsigned char)\*(lpPtr+x-LineBytes+1); //右下邻点

//计算索引

num=nw/255+n/255\*2+ne/255\*4+w/255\*8+e/255\*16+

sw/255\*32+s/255\*64+se/255\*128;

if(erasetable[num]==1){ //经查表，可以删除

//在原图缓冲区中将该黑点删除

\*(lpPtr+x)=(BYTE)255;

//结果图中该黑点也删除

\*(lpTempPtr+x)=(BYTE)255;

Finished=FALSE; //有改动，结束标志置成假

x++; //水平方向跳过一个象素

}

}

}

x++; //扫描下一个象素

}

}

//再进行垂直方向的细化

for (x=1;x<bi.biWidth-1;x++){ //注意为防止越界，x的范围从1到宽度-2

y=1; //注意为防止越界，y的范围从1到高度-2

while(y<bi.biHeight-1){

lpPtr=(char \*)lpImgData+(BufSize-LineBytes-y\*LineBytes);

lpTempPtr=(char\*)lpTempImgData+

(BufSize-LineBytes-y\*LineBytes);

if(\*(lpPtr+x)==0){ //是黑点才做处理

n=(unsigned char)\*(lpPtr+x+LineBytes);

s=(unsigned char)\*(lpPtr+x-LineBytes);

if( (n==255)|| (s==255)){

//如果上下两个邻居中至少有一个是白点才处理

nw=(unsigned char)\*(lpPtr+x+LineBytes-1);

ne=(unsigned char)\*(lpPtr+x+LineBytes+1);

w=(unsigned char)\*(lpPtr+x-1);

e=(unsigned char)\*(lpPtr+x+1);

sw=(unsigned char)\*(lpPtr+x-LineBytes-1);

se=(unsigned char)\*(lpPtr+x-LineBytes+1);

//计算索引

num=nw/255+n/255\*2+ne/255\*4+w/255\*8+e/255\*16+

sw/255\*32+s/255\*64+se/255\*128;

if(erasetable[num]==1){ //经查表，可以删除

//在原图缓冲区中将该黑点删除

\*(lpPtr+x)=(BYTE)255;

//结果图中该黑点也删除

\*(lpTempPtr+x)=(BYTE)255;

Finished=FALSE; //有改动，结束标志置成假

y++;//垂直方向跳过一个象素

}

}

}

y++; //扫描下一个象素

}

}

}

if(hBitmap!=NULL)

DeleteObject(hBitmap);

hDc=GetDC(hWnd);

//产生新的位图

hBitmap=CreateDIBitmap(hDc,(LPBITMAPINFOHEADER)lpTempImgData,

(LONG)CBM\_INIT,

(LPSTR)lpTempImgData+

sizeof(BITMAPINFOHEADER)+

NumColors\*sizeof(RGBQUAD),

(LPBITMAPINFO)lpTempImgData,

DIB\_RGB\_COLORS);

hf=\_lcreat("c:\\thinning.bmp",0);

\_lwrite(hf,(LPSTR)&bf,sizeof(BITMAPFILEHEADER));

\_lwrite(hf,(LPSTR)lpTempImgData,BufSize);

\_lclose(hf);

//释放内存及资源

ReleaseDC(hWnd,hDc);

LocalUnlock(hTempImgData);

LocalFree(hTempImgData);

GlobalUnlock(hImgData);

return TRUE;

}

复制代码

题外话：

腐蚀：删除对象边界的某些像素

膨胀：给图像中的对象边界添加像素

算法：

膨胀算法：用3X3的结构元素，扫描二值图像的每一个像素，用结构元素与其覆盖的二值图像做“与”运算，如果都为0，结构图像的该像素为0，否则为1.结果：使二值图像扩大一圈。

腐蚀算法：用3X3的结构元素，扫描二值图像的每一个像素，用结构元素与其覆盖的二值图像做“与”运算，如果都为1，结构图像的该像素为1，否则为0.结果：使二值图像减小一圈。