数字图像处理第 1 次作业

210320621 吴俊达

September 21, 2023

2.9 解：500 幅大小为 1024×1024 像素的 256 灰度级 (灰度值用 8 位二进制数存储，即 1 字节。传输时加上起始位和结束位，每传输一个字节的数据需要 10 位的数据量) 图像的数据量为

$$D = 1024 \times 1024 \times 10 \times 500 = 5000 \times 2^{20} \text{ Bit}$$

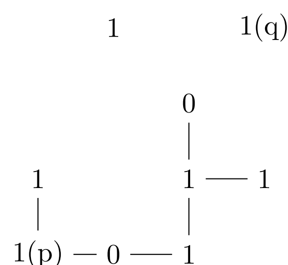
则
(a) 时间为

$$T_1 = \frac{D}{3 \times 10^6} = 1747.6\text{s}$$

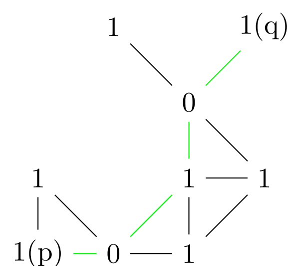
(b) 时间为

$$T_2 = \frac{D}{3 \times 10^{10}} = 0.1748\text{s}$$

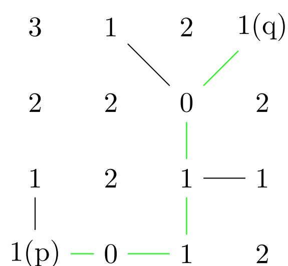
2.14 解：(a) S_1 和 S_2 不是 4 邻接的 (S_1 第 4 行第 4 列的 1 与 S_2 第 3 行第 1 列的 1 不邻接)。
(b) S_1 和 S_2 是 8 邻接的。
(c) S_1 和 S_2 是 m 邻接的。
2.18 解：(a) 不存在 4 通路，如下所示，绘制出所有邻接的像素点，发现 p 与 q 之间没有通路。



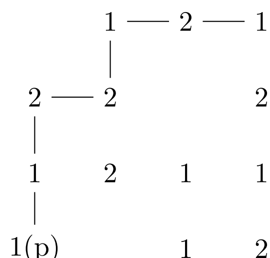
存在 8 通路，最短长度为 4，如下绿色路径所示。



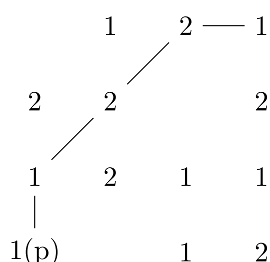
存在 m 通路，最短长度为 5，如下所示。



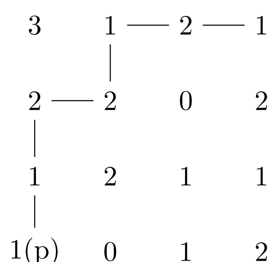
(b) 存在 4 通路，如下所示，最小长度为 6。下面给出一条可能的最短通路。



存在 8 通路，最短长度为 4，如下所示 (仅画出一条符合条件的通路)。



存在 m 通路，最短长度为 6，下面给出一条可能的最短通路。



3.1 解：设原图像灰度在空间上的分布用函数 $g(x,y)$ 来表示，则所求的灰度变换函数可以是

$$T[g(x,y)] = (L-1)\frac{g(x,y) - \min_{x,y} g(x,y)}{\max_{x,y} g(x,y) - \min_{x,y} g(x,y)}$$

3.5 解：(a) 如果将低有效比特面设为零，图像的不同灰度级的个数会减少，低灰度级的像素会部分丢失，但像素总数不变，丢失的像素转移到其它未丢失 (更高) 的灰度级上，从而导致图像变亮但是对比度降低。(b) 同理，当图像高有效比特面设为零，图像的不同灰度级的个数会减少，高灰度级的像素会丢失，丢失的像素都转移到低灰度级上，从而导致图象直方图只有低灰度区，高灰度区直方图均为零。图像变暗且对比度降低、变得模糊。

3.29 解：忽略边界效应时，由于低通滤波器总是归一化的 (核内所有的元素总和为 1)，滤波后图像中的

所有像素值总和不变 (习题 3.31 中证明)；但是，利用卷积的线性性质，将图像拆分成若干张单个像素点组成的子图，对其进行滤波，就会发现滤波在空间上使得像素往外“扩张”，且每次滤波都向外扩张一周，因此极限情形下像素所占的空间会趋于无限大；但是由于所有像素值总和为一有限值，因此像素在空间上的平均能量将趋于 0。另外，每个像素的值也都是有限值，因此每个像素值将趋于 0。从观感上来看，就是越来越暗且模糊。

(如果考虑边界效应，例如将与边界接触且在边界以外的像素值设为 0，则每次对于边界上的元素作滤波时，由于 0 元素的参与，像素值的总和将下降。极限情形下，像素值的总和将全部“耗散”在 0 上，最后得到一张纯黑色的图片)

3.31 证明：首先，此处滤波使用的运算为完全卷积。(下面我们将看到，证明的核心在于注意到，使用完全卷积能确保每个像素被卷积核中的每个元素处理的次数相同，或者说确保卷积核中的每个元素处理到的像素范围和次数相同)

设**原图像**中的像素值在空间上的分布用函数 $f(x,y)$ 来表示。在每次卷积中，所得像素值是卷积核中的元素和对应位置像素值相乘的结果之和。那么，将所有像素值都加起来，得到的就是所有上述相乘的结果的和。在完全卷积中，核中的每个元素与图像中的每个像素都相乘过一次且仅相乘过一次。我们将视角转换到像素上，就会发现每个**像素**都和卷积核中的每个元素乘过一次且仅有一次，也就是求和式中都包含它和卷积核中每个元素相乘一次再相加的结果，则和式就可以写成

$$S = \sum_{i,j}\sum_{s,t} f(i,j)w(s,t) = \sum_{i,j} f(i,j)\sum_{s,t} w(s,t)$$

由于 $\sum_{s,t} w(s,t) = 1$，所以上式变为 $S = \sum_{i,j} f(i,j)$，即为原像素中的像素值之和。

3.37 解：(a) 先进行快速排序，由于总数一定，很容易利用下标访问到中值 (分奇偶数即可)。尝试写出具体代码如下：(假设 n^2 个元素存在数组 a 中)

```
# define MAXSIZE 100
/**
 * 快速排序
 * @param a 待排序数组
 * @param length 数组中数据的总个数
 * @return 无
 */
void QuickSort(int* a, int length){
    int Pivotkey = a[0],  LowMove = 0, HighMove = length-1;
    if(length<=1) return;
    while(1){
        while(a[LowMove]<=Pivotkey) {
            ++LowMove;
            if(LowMove==length-1)     break;
        }
        while(a[HighMove]>=Pivotkey) {
            --HighMove;
            if(HighMove==0)     break;
        }
        if(LowMove>=HighMove) break;
        int temp = a[LowMove];
        a[LowMove] = a[HighMove];
```

```
23            a[HighMove] = temp;
24        }
25        a[0] = a[HighMove];
26        a[HighMove] = Pivotkey;
27        QuickSort(a,HighMove);
28        QuickSort(a+HighMove+1,length-HighMove-1);
29    }
30
31    int main(){
32        int n;  // 邻域边长
33        int Medium;  // 中值
34        scanf("%d", &n);
35        int TotalSize = n^2;
36        int a[MAXSIZE];  // 用于存储像素值
37        /*
38          读取 a
39        */
40        QuickSort(a, TotalSize);
41        if(TotalSize%2)    Medium = a[TotalSize/2];
42        else    Medium = (a[TotalSize/2]+a[TotalSize/2-1])>>1;
43        printf("%d", Medium);
44        return 0;
45    }
```

(b) 考虑到像素个数往往远多于像素取值，因此可以将算法的时间复杂度从与像素个数关联转为与像素取值关联。由此得算法如下：

1. 在第一次操作时，遍历一次邻域，建立一个数组用于存放每种像素值的个数 (即直方图的数值表示)，通过此数组找到中值 (记为 Med) 并存储 (相当于知道中值在数组中的位置)，并且存储比中值小的元素的个数 (记为 SNum)。

2. 接下来每次操作时，由于邻域中心只移动一个像素，因此中间重叠的部分不变，只有两边上一边减少、一边增加，只需将减少和增加的值的信息更新到该数组 (部分像素值的像素个数增加或减少，相当于更新直方图) 以及更新 SNum。

(1) 如果 SNum 大于 $n^2/2$，那么需要往小更新中值，方法是每次将中值减小 1(或其他量化单位) 并将 SNum 减去原中值的像素个数。

(2) 如果 SNum+ 中值像素个数小于 $n^2/2$，那么需要往大更新中值，方法是每次将中值增大 1(或其他量化单位) 并将 SNum 加上原中值的像素个数。

直到 SNum 小于 (等于)$n^2/2$ 且 SNum+ 中值像素个数大于 (等于)$n^2/2$，存储 Med。

此处算法的描述在实际实现时，还需针对元素个数的奇或偶来做一些调整。
从中值出发是考虑到图片中中值变化很剧烈的情况较少，即在大多数情形下，从上一个中值出发只需很少的几步 (甚至不移动) 就可以找到下一个中值。当然也可以考虑每次都从像素值为 0 处 (数组的第一项) 开始，但这样迭代的次数就很多而没有明显优势。具体代码实现还需要考虑卷积核到达图片边界等问题，与实际问题紧密相关，所以这里呈现的只是实现上述第 2 点的不完全版本。

```cpp
// 全局变量
int SNum = 0;  // 比中位数小的元素个数
const int Length = 256;  //  像素值的范围，数组的长度
int n = 0;  // 邻域边长

// 更新像素数组和 SNum 的函数 UpdatePixel 略去

/**
 * 更新中位数
 * @param pixel 存有像素值对应像素个数的数组指针
 * @param Med 中位数
 * @return 返回中位数
 */
int UpdateMed(int *pixel, int Med){
    // 比当前中值小的像素个数 + 中值对应像素个数小于总数一半 (中值太小)
    if(SNum+pixel[Med]<n*n/2+n%2){
        while(SNum+pixel[Med]<n*n/2+n%2)
            SNum += pixel[++Med];  // 中值增大并更新 SNum
        if(n%2 == 1 || SNum < n*n/2)
            return Med;  // 总数为奇数或比中值小的像素个数少于总数一半
        else{  // 总数为偶数，比中值小的像素个数恰等于总数一半
            for(int i=Med-1;i>=0;--i)
                if(pixel[i])
                    return (i+Med)/2;
        }
    }
    // 比当前中值小的像素个数超出总数一半 (中值太大)
    if(SNum>n*n/2){
        while(SNum>n*n/2)
            SNum -= pixel[--Med];  // 中值减小并更新 SNum
        if(n%2 == 1 || SNum < n*n/2)
            return Med;  // 总数为奇数或比中值小的像素个数少于总数一半
        else{  // 总数为偶数，比中值小的像素个数恰等于总数一半
            for(int i=Med+1;i<n*n;++i)
                if(pixel[i])
                    return (i+Med)/2;
        }
    }
    return Med;
}

int main(){}  // 略去
```

3.40 解：将中心是 -8 的拉普拉斯算子写成解析表达式得

$$\begin{aligned} g(x,y) = & f(x+1,y+1)+f(x-1,y-1)+f(x+1,y-1)+f(x-1,y+1) \\ & +f(x+1,y)+f(x,y+1)+f(x-1,y)+f(x,y-1)-8f(x,y) \end{aligned}$$

整理得

$$
\begin{aligned}
g(x,y) =&[f(x+1,y+1)+f(x-1,y-1)+f(x+1,y-1)+f(x-1,y+1)-4f(x,y)] \\
&+[f(x+1,y)+f(x,y+1)+f(x-1,y)+f(x,y-1)-4f(x,y)]
\end{aligned}
$$

发现后一项就是中心为 -4 的拉普拉斯算子。前一项则是其多出来的部分，其作用是将对角线上的值与本像素值的微分也考虑进去，即“在对角方向提供了额外的微分 (第 4 版 120 页)”，增加了锐化的效果。