寻找中位数

Dezeming Family

2021 年 5 月 1 日

目录

一　中位数的定义

最近正好用到了中值滤波，发现 [1] 这篇文档好像还不错，就简单学习了一下。鉴于可能有些人感觉看英文不是很方便，我就简单整理了一下，顺便写了一些自己的分析和理解。我觉得一般来说，WIRTH 方法和小数组滤波核方法就完全够用了，但还是多介绍了几种方法。

想看原文的可以去找 [1]，想找个方法直接用的可以直接拷贝，每种中位数搜索算法都具有一定的特征，例如有的搜索需要递归，有的搜索需要复制原数组，有的搜索不修改原数组等。

中值滤波器在信号处理中应用广泛。在数字信号处理中，它可以用来消除数值平滑分布中的异常值。中值滤波器通常被称为保持高频的非线性散粒噪声滤波器 (non-linear shot noise filter)。在图像处理中，中值滤波器是通过（2N+1，2N+1）核的卷积来计算的。对于输入帧中的每个像素，输出帧中相同位置的像素被内核中像素值的中值所取代。中值又称为中位数，有两种定义：

- N 个值从小到大或者从大到小排序，取中间的数的值。
- 中值在数组中，比它大的数个数与比它小的数的个数相同。

很多人认为找中值先排序再搜寻比较浪费时间，但其实大部分找中位数的算法都需要进行排序。

二　中位数搜寻算法

2 1　qsort()

符合 ANSI 标准的 C 库一般都有 quicksort 方法的实现：qsort()，用来快速排序。但是相比于最近实现的新方法，这些方法用于中位数搜索并不好。该函数的主要目的是给出实现中值搜索的最慢方法的参考。另一方面，它是非常简单且易于阅读。

2 2　pixel_qsort()

pixel_qsort() 方法更快，在 ANSI C 库中进行了一定的优化。

2 3　median AHU()

median AHU() 是一个递归方法，该算法利用 S 中只搜索第 k 个最小元素的特点，不需要对整个数组进行排序。它比排序完整数组再搜索要快得多，并且可以搜寻数组中第 k 小的数。如下伪代码，S 是数组，k 是要搜寻的第 k 小的数，|S| 表示数组 S 中的元素数，S[data] 表示数组 S 中的数据：

```
select(k,S):
    if(|S| == 1) return S[data];
    else
        从S中随机选一个元素s
        S1 = S中小于s的全部元素
        S2 = S中等于s的全部元素
        S3 = S中大于s的全部元素
        if(|S1| >= k) return select(k, S1);
        else
            if(|S1|+|S2| >= k) return s;
            else return select(k-|S1|-|S2|, S3);
```

该算法是递归的，每次迭代都需要分配一个输入数组的副本。对于大的输入阵列，这会带来严重的内存需求，而且几乎不可能限制潜在使用的内存数量。在最坏的情况下，工作所需的内存量可能达到 N*（N-1）/2，这很可能导致内存分配失败或程序崩溃。即使命中此类情况的概率很低，也不建议在大型阵列或任何高可靠性程序中使用此算法。

出于教育目的这是一种有趣的方法，但由于递归性的限制，在任何其他环境中使用都是不现实的。此外，这里描述的其他方法具有更快和非递归的优点。注意：“在 S 中随机选择一个元素”在建议的实现中被修改为“在 S 中取中心元素”，以避免在每次迭代中调用随机生成器和模除。这就带来了这样一个事实：对于这种方法，有些数组将是非常糟糕的情况，需要大量的迭代。随机选择可以确保（几乎）始终保持在合理的范围内，但在每个迭代中限制为 2 个昂贵的随机数调用。

2 4 median WIRTH()

此为非递归方法，它不尝试对整个数组进行排序，而是浏览输入数组，以确定输入列表中第 k 个最小的元素是什么。它不是递归的，不需要分配任何内存，也不使用任何外部函数。因此，与基于 qsort() 的方法相比，它的速度提高了 25 倍。

显然，这是与 AHU median 相同的算法，但在数组的原位实施，而不是要不断创建分配新数组。它的优点很明显，它摆脱了递归性，代价是输入数组的初始副本（因为在运算过程中会修改数组）。

中位数搜索定义为函数顶部的宏，它查找第 k 个最小元素。它将奇数个点的中位数定义为中间的一个，偶数的中位数定义为刚好在中间靠前的一个。

首先先定义一下交换程序：

```
#define ELEM_SWAP(a,b) { register float t=(a);(a)=(b);(b)=t; }
```

```
//从n个元素中找第k小的数:
float kth_smallest(float a[], int n, int k) {
    register int i,j,l,m;
    register float x;
    l = 0 ; m = n-1;
    while (l<m) {
        x = a[k] ;
        i = l ;
        j = m ;
        do {
            //从前往后找到第一个大于等于x的值
            while (a[i] < x) i++;
            //从后往前找到第一个小于等于x的值
            while (x < a[j]) j--;
            if (i <= j) {
                //将两个值交换
                ELEM_SWAP(a[i], a[j]);
                i++ ; j-- ;
            }
        } while (i <= j);
        if (j < k) l = i;
        if (k < i) m = j;
    }
    return a[k] ;
}
#define median(a,n) kth_smallest(a,n,(((n)&1)?((n)/2):(((n)/2)-1)))
```

上面的程序中，每个循环中 do-while 语句执行完以后，数组中索引小于 k 的数都小于 a[k]，大于 k 的数都大于 a[k] 了。

2 5 Quick select

从速度上看，这与 WIRTH 的方法密切相关。不过，平均来说，这个速度更快 [2]。它修改了输入数组，因此同样的警告也适用：在应用中值搜索之前必须复制输入数据集。

```
float quick_select(float arr[], int n){
    int low, high ;
    int median;
    int middle, ll, hh;
    low = 0 ; high = n-1 ; median = (low + high) / 2;
    for (;;) {
        //只有一个元素
        if (high <= low) return arr[median] ;
        //有两个元素
        if (high == low + 1){
            if (arr[low] > arr[high]) ELEM_SWAP(arr[low], arr[high]) ;
            return arr[median] ;
        }
        //找出low、middle、high项的中位数（中位数调换到low位置处）
        middle = (low + high) / 2;
        if (arr[middle] > arr[high]) ELEM_SWAP(arr[middle], arr[high]);
        if (arr[low] > arr[high]) ELEM_SWAP(arr[low], arr[high]);
        if (arr[middle] > arr[low]) ELEM_SWAP(arr[middle], arr[low]);
        //调换low项到low+1，注意现在的low项在middle位置
        ELEM_SWAP(arr[middle], arr[low+1]);
        //从两端向中间推进，卡住时交换项
        ll = low + 1;
        hh = high;
        for (;;) {
            do ll++; while (arr[low] > arr[ll]);
            do hh--; while (arr[hh] > arr[low]);
            if (hh < ll) break;
            ELEM_SWAP(arr[ll], arr[hh]);
        }
        //把位于middle的项（现在在low位置）调换到正确的位置
        ELEM_SWAP(arr[low], arr[hh]);
        //重新设置活动分区
        if (hh <= median) low = ll;
        if (hh >= median) high = hh - 1;
    }
}
```

2 6 Torben’s method

这个方法是 Torben Mogensen 的。它不是寻找中值的最快方法，但它有一个非常有趣的特性，即在寻找中值时不修改输入数组。当要考虑的元素数量开始很大时，它变得非常强大，复制输入数组可能会导致巨大的开销。

对于大小为几百兆字节的只读输入集，它是首选的解决方案，还因为它按顺序而不是随机地访问元

素。但是要注意，它需要多次读取数组：第一次遍历只查找最小值和最大值，进一步遍历数组并在比 pixel qsort () 方法多一点的时间内得到中值。迭代的次数可能是 O(log(n))，尽管我没有证明这个事实。

```
float torben(float m[], int n) {
    int i, less, greater, equal;
    float min, max, guess, maxltguess, mingtguess;
    min = max = m[0] ;
    for (i=1 ; i<n ; i++) {
        if (m[i]<min) min=m[i];
        if (m[i]>max) max=m[i];
    }
    while (1) {
        guess = (min+max)/2;
        less = 0; greater = 0; equal = 0;
        maxltguess = min ;
        mingtguess = max ;
        for (i=0; i<n; i++) {
            if (m[i]<guess) {
                less++;
                if (m[i]>maxltguess) maxltguess = m[i] ;
            } else if (m[i]>guess) {
                greater++;
                if (m[i]<mingtguess) mingtguess = m[i] ;
            } else equal++;
        }
        if (less <= (n+1)/2 && greater <= (n+1)/2) break ;
        else if (less>greater) max = maxltguess ;
        else min = mingtguess;
    }
    if (less >= (n+1)/2) return maxltguess;
    else if (less+equal >= (n+1)/2) return guess;
    else return mingtguess;
}
```

三　图像处理滤波器的应用

3 1　Small kernels

上面描述的方法对于搜索许多元素的中值非常有用，但是对于少量的值，甚至有更快的方法，可以通过硬连线在尽可能快的时间内生成中值。在图像处理中，3x3 核上的形态学中值滤波器需要为输入图像中的每一组 9 个相邻像素找到 9 个值的中值。这里提供的代码可以在尽可能快的时间内获得 3、5、7、9 和 25 个值的中间值（不涉及硬件细节）。对于不同数量的值，可以找到其他排序网络，这里不提供。一篇关于 3x3 元素的快速中值搜索的文章可以在 [3] 上找到。

首先定义一下排序和交换的宏：

```
#define PIX_SORT(a,b) { if ((a)>(b)) PIX_SWAP((a),(b)); }
#define PIX_SWAP(a,b) { float temp=(a);(a)=(b);(b)=temp; }
```

然后是找三个数之间的中值：

```
float opt_med3(float * p) {
    PIX_SORT(p[0],p[1]) ; PIX_SORT(p[1],p[2]) ; PIX_SORT(p[0],p[1]) ;
    return(p[1]) ;
}
```

找五个数之间的中值：

```
float opt_med5(float * p) {
    PIX_SORT(p[0],p[1]) ; PIX_SORT(p[3],p[4]) ; PIX_SORT(p[0],p[3]) ;
    PIX_SORT(p[1],p[4]) ; PIX_SORT(p[1],p[2]) ; PIX_SORT(p[2],p[3]) ;
    PIX_SORT(p[1],p[2]) ; return(p[2]) ;
}
```

找九个数之间的中值：

```
float opt_med9(float * p) {
    PIX_SORT(p[1], p[2]) ; PIX_SORT(p[4], p[5]) ; PIX_SORT(p[7], p[8]) ;
    PIX_SORT(p[0], p[1]) ; PIX_SORT(p[3], p[4]) ; PIX_SORT(p[6], p[7]) ;
    PIX_SORT(p[1], p[2]) ; PIX_SORT(p[4], p[5]) ; PIX_SORT(p[7], p[8]) ;
    PIX_SORT(p[0], p[3]) ; PIX_SORT(p[5], p[8]) ; PIX_SORT(p[4], p[7]) ;
    PIX_SORT(p[3], p[6]) ; PIX_SORT(p[1], p[4]) ; PIX_SORT(p[2], p[5]) ;
    PIX_SORT(p[4], p[7]) ; PIX_SORT(p[4], p[2]) ; PIX_SORT(p[6], p[4]) ;
    PIX_SORT(p[4], p[2]) ; return(p[4]) ;
}
```

3 2 Large kernels Image

使用大内核的中值滤波器可能会使用冗余，因为该方法正在查找 NxN 像素的中值，然后转到下一个内核位置（通常：右侧的下一个像素）意味着取出 N 个像素并添加 N 个新像素。对于 3x3 内核，使用这种冗余是不够的，因为只有 3 个像素保持不变，但是对于大型内核，例如 40x40，每次迭代时只有 40 个像素少，40 个像素多，但是 1560 个值保持不变。在这种情况下，使用直方图或基于树的方法可能更有效。这里没有提供这些方法的实现，只提供了一般的思想。在建立直方图时，可以注意到中值信息确实存在于这样一个事实中，即像素被分类为像素值不断增加的桶。

从 bucket 中删除像素并添加更多的像素是一个简单的操作，这就解释了为什么保持一个运行的直方图并更新它可能比从零开始运行内核更容易。感兴趣的读者请参考 [4]。

同样的想法也可以用来建立一个包含像素值和出现次数，或者间隔和像素数的树。我们可以看到在每一步保留这些信息的直接好处。

四　性能分析

大家可以参考原文 [1]，这里就不再赘述了。

参考文献

[1] Devillard N . Fast Median Search: An ANSI C Implementation. 1998.

[2] Numerical recipes in C, second edition, Cambridge University Press, 1992, section 8.5, ISBN 0-521-43108-5

[3] John Smith, Implementing median filters in XC4000E FPGAs, http://www.xilinx.com/xcell/xl23/xl23 16.pdf

[4] T.S.Huang, G.J.Yang, G.Y.Tang, A fast two-dimensional median filtering algorithm, IEEE transactions on acoustics, speech and signal processing, Vol ASSP 27 No 1, Feb 1979.