GENERALIZATION: RANK ORDER FILTERS

Definition

For simplicity and implementation efficiency, we consider only brick (rectangular: wf x hf) filters. A brick rank order filter evaluates, for every pixel in the image, a rectangular set of n = wf x hf pixels in its neighborhood (where the pixel in question is at the "center" of the rectangle and is included in the evaluation). It determines the value of the neighboring pixels that is the r-th smallest in the set, where r is some integer between 1 and n. The input rank is a fraction between 0.0 and 1.0, where 0.0 represents the smallest value (r = 1) and 1.0 represents the largest value (r = n). A median filter is a rank filter where rank = 0.5. The rank order filter is a generalization of grayscale erosion and dilation. The erosion is equivalent to rank = 0.0 (because we're choosing the minimum in the set), and a dilation is equivalent to rank = 1.0 (the maximum). The min and max are much easier to calculate than the general rank value, thanks to the van Herk/Gil-Werman algorithm. The brick grayscale erosion and dilation are separable, which also increases the efficiency of implementation, whereas the general rank order filter is not.

In our implementation, if these extremal ranks are specified, and the filter dimensions are both odd, the appropriate separable morphological operation is dispatched by the rank order function pixRankFilterGray().

How is a brick rank order filter implemented efficiently?

The brute force method is to sort all the neighboring pixels, and pick the value of the r^th one. The best sorting algorithms are O(n*logn), where n, the area of the filter, is the number of values to be sorted. For a 50x50 filter, the number of operations at each pixel is over 10,000, which is not practical. (These are not machine operations.)

However, we only need the r-th largest pixel. So our problem is really a "selection" one (e.g., quickselect), not a sorting problem. Now it turns out that using a variation of quicksort, selection of any specific rank value is O(n). This is because for selection you only need to sort the appropriate segment at each bifurcation in the quicksort method, rather than the entire tree. (For details, see Numerical Recipes in C, 2nd edition, 1992, p. 355ff.) There is a constant 2 in front of the n, relative to quicksort, so this brings the count down to 5,000 for our example, which is still ridiculously large.

Actually, we only need to do an incremental selection or sorting, because moving the filter down by one pixel causes one filter-width of pixels to be added and another to be removed. Can we do this incrementally in an efficient manner? Perhaps, but I don't know how. The sorted values will be in an array. Even if the filter width is 1, we can expect to have to move O(n) pixels, because insertion and deletion can happen anywhere in the array. By comparison, heapsort is excellent for incremental sorting, where the cost for insertion or deletion is O(logn), because the array itself doesn't need to be sorted into strictly increasing order. The heap, which is represented by an array, only needs to be in "heap" order, so just a few elements must be rearranged. However, heapsort only gives the max (or min) value, not the general rank value.

The conclusion is that sorting and selection are far too slow. All is not lost, because we can still use a histogram of pixel values. But how is this to be represented?

Suppose we represent the histogram as an array of 256 bins, which is the usual situation. At each new filter location, the histogram must be updated and the rank value computed. The problem we face is that, in general, to find the rank value, a significant fraction of the entire histogram must be summed. Suppose the filter size is 5x5. There are at most 25 different bins occupied, and we will mostly be adding zeros to the sum of bin occupancies. That is painful!

We can instead use a linked list, so that there won't be any unoccupied cells and the sum can be quickly done. However, we lose random access for insertion and deletion, so those operations become unacceptably slow.

However, we can mostly overcome the empty bin problem and retain random access by using two histograms represented as arrays, with bin sizes 1 and 16. Each of these histograms is updated for each pixel added or removed when the filter is moved. To find the rank value, proceed from coarse to fine, first locating the coarse bin for the given rank value, of which there are only 16. Then for the fine histogram with bin size 1 and 256 entries, we need look only at a maximum of 16 bins. On average, there will be a total of about 16 sums.

Points of interest in the implementation

The implementation is in rank.c. There are several things to notice:

Two histograms are maintained throughout: a 16-bin coarse histogram and a 256-bin fine histogram, as mentioned above.

The histograms are updated incrementally, using either row major order or column major order, depending on the aspect ratio of the filter. For row major order, which is used when the filter height hf is larger than the width wf, the fast scan is down the column.

For that case, for each column, generate the histogram for the entire filter at the first location (the top of the column). Then as the filter is moved down, remove the values for the old top row and add the values for the new bottom row.

Avoid special-case processing for pixels near the boundary by expanding the image by half the filter dimension on each side (specifically, half the horizontal dimension on left and right sides, and half the vertical dimension on top and bottom).

To avoid bias as far as possible in the added boundary pixels, we mirror the pixel values across the boundary.

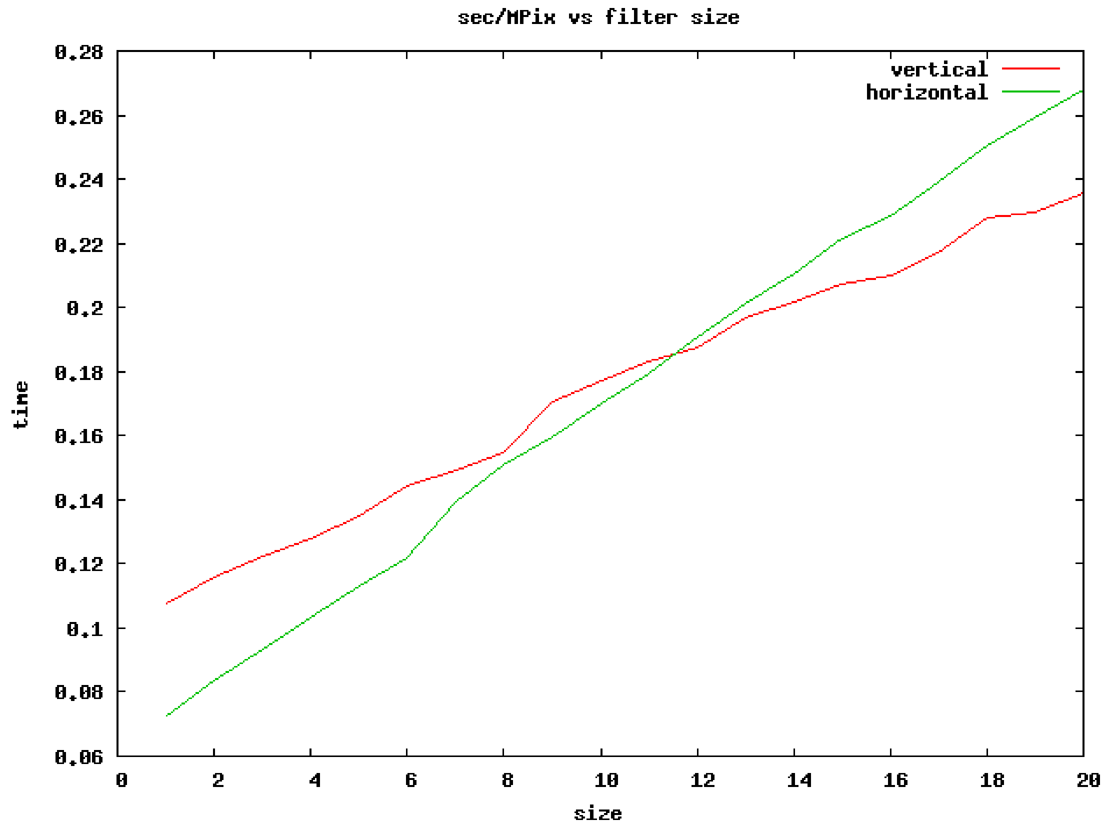
For convenience, choose the filter "origin" implicitly at the UL corner. Then we can simply march through dimensions of the width and height of the original (un-bordered) image to generate the destination (rank-order) image.

If both filter dimensions are odd and the rank is either 0.0 or 1.0, we use instead grayscale erosion or dilation, respectively. The reason both dimensions have to be odd is that grayscale erosion and dilation are only computed using filters of odd width and height.

If the input rank is 0.0 or 1.0, with at least one filter dimension being even, it is necessary to use the slower rank order function.

However, we move the rank value slightly off the input value (e.g., move 0.0 to 0.0001) to allow use of a simpler comparison in the inner loop that does the sum over histogram bins.

The operation is reasonably fast, considering its complexity. The speed is essentially independent of the size of the larger dimension of the filter. The time is approximately a constant plus a term that is linear in the smaller dimension of the filter (generated by prog/rank_reg). In the figure below we show the operation time per Mpixel vs the smaller filter dimension.



Note that a rank operation with a horizontal filter is faster than with a vertical filter when the small dimension is very small, but the incremental increase in time as the small dimension expands is larger for the horizontal filter. The latter effect can be understood because for the horizontal filter, the incremental pixel addressing is over two vertical columns, which is slower than addressing across two rows. For a moderate sized filter with a smallest dimension of 20, the rank order filter operates at about 4 MPix/sec on a 3 GHz machine.

Source: http://www.leptonica.com/grayscale-morphology.html