A comparison of connected-component algorithms

Frederick M. Waltz\textsuperscript{a} and John W. V. Miller\textsuperscript{b}

\textsuperscript{a}Consultant, 2095 Delaware Avenue, Mendota Heights, MN 55118-4801 USA
\textsuperscript{b}John W. V. Miller, University of Michigan–Dearborn, Dearborn, MI 48128 USA

\textbf{Abstract}

The SKIPSM (Separated-Kernel Image Processing using finite-State Machines) paradigm offers fast execution of a very wide range of binary, grey-scale, 3-D, and color image-processing applications. In this paper the finite-state-machine approach is applied to one of the “classical” problems of binary image processing: connected-component analysis (“blob” analysis). Execution-time results are presented, and compared for several examples to execution times for the very-efficient conventional method based on analysis of run-length-encoded data.

\textbf{Keywords:} blob analysis, finite-state machines, high-speed operation

1. Introduction

The SKIPSM (Separated-Kernel Image Processing using finite-State Machines) paradigm offers fast execution of a very wide range of neighborhood image processing operations, including binary, grey-scale, 3-D, and color operations, as detailed in numerous previous papers\textsuperscript{1-21}. All of these implementations are for neighborhood operations have in common the four fundamental steps of the SKIPSM approach:

1. The neighborhood operation is decomposed into a row operation and a column operation (in either order).
2. The row and column operations are expressed recursively, so that “pipelined” raster-scan access to the input image pixels is possible. In other words, each input pixel is accessed once and only once. Random access to the input image pixels is neither possible nor necessary, no matter how large a neighborhood is used.
3. The row and column operations are realized as finite-state machines (FSMs).
4. Two kinds of implementations are used, depending on the application and the speed requirements:
   a. The FSMs can be “pre-compiled” by calculating ahead of time all possible input-state-output combinations, and expressing the results in the form of lookup tables (LUTs). These LUTs can be used interchangeably for software-based applications or for high-speed pipelined hardware implementations.
   b. The FSMs can be implemented by a set of rules — that is, by computer code embodying the underlying FSM rules. This code executes at run time, using the input and the current state to calculate the output and next state. In either case, tools are available to create the FSM implementations (LUTs or compileable C-code) automatically.

For this paper, as for all SKIPSM implementations, the overall image processing operation is separated into a row operation and a column operation. Figure 1 illustrates the sequence of operations for a “generic” SKIPSM algorithm: Fetch the current pixel (column \(i\), row \(j\)). Feed it, along with the contents of RState, the row state value, to the row machine function \(f_R\). This computes a new row state value to replace the previous value of RState, and an output value, which becomes the input to the column machine. This value, along with the contents of CState[\(i\)], the column state value for this column of the image, is fed to the column machine function \(f_C\). This computes a new state value to replace the previous value of CState[\(i\)], and an output value, which is written to the output image. The process is repeated for the next pixel in the row. And so on to the end of the row and for succeeding rows.

In this paper the SKIPSM approach is applied to one of the “classical” problems of binary image processing: connected-component analysis (“blob” analysis). Efficient algorithms already exist for this operation. In particular, the one that uses run-length encoding to reduce a 2-D image processing operation to a 1-D analysis of a list of numbers is highly efficient in most cases. The goal of this paper is therefore to determine whether the speed advantages provided by SKIPSM for many other kinds of operations carry over to binary connected-component analysis. Speed comparisons between the two algorithms are provided for various examples.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure1.png}
\caption{A generic 2-D SKIPSM implementation}
\end{figure}
2. THE SKIPSM 8-CONNECTED BLOB-LABELLING ALGORITHM

The goal of blob analysis is to assign a unique label (grey level) to each “blob” in the input image. The concept of an 8-connected or 4-connected “blob”, as well as the details of this algorithm itself, are so well covered in the literature that no further formal definition or reference will be given here. But for illustration of the concept, consider Figure 2, with three blobs. (Here and in all the examples, the “object” pixels will be shown as white or light and the “background” pixels as black or dark.)

1. All of the pixels in the upper half of the image form the first blob.
2. The “T” shape in the lower half of the image is the second blob.
3. The “L” shape in the lower half of the image is the third blob.

Various shadings have been used in the output image to suggest the assignment of different grey levels to each blob.

For comparison, if the 4-connected definition of connectivity is used (edges only), there are six blobs. The “L” and the “F” in the upper half become separate blobs, and the “L” in the lower half becomes two blobs. This paper is devoted mainly to discussion of 8-connected algorithms.

In their image-processing phase, “pipelined” versions of this operation (including SKIPSM) read in each binary input pixel in turn, in raster-scan order, and proceed according to these rules, which are stated here in rather general form:

1. If the input pixel is black, then write the background label value (an integer, usually zero) to the output image at that point.
2. If the input pixel is white, then look at all the pixels in the “s-shaped” 8-connected neighborhood of the current pixel, as shown in Figure 3a. Choose the output value according to these rules:
   Case a – all of the neighboring pixels are background pixels: Create a new label (i.e., an integer value not yet used as a label anywhere else in the output image) and assign it to the output pixel at the same position. The usual practice is to use 1 for the first such label created, 2 for the next, and so on.
   Case b – only one of the neighboring pixels is not a background pixel: Assign the label of this pixel to the current pixel, and write this value to the output pixel at the same position.
   Case c – two or more of the neighboring pixels are not background pixels: Assign the label of any one of the non-background pixels to the output pixel. (The choice is arbitrary.) Also, and most important, check the labels assigned to these other neighborhood pixels. If they are not all the same, append all the distinct values appearing in the neighborhood to a list of label values to be merged later. (The merging operation is discussed in detail in a later section.)
3. Finally, if moments are being calculated (e.g., areas, centers of gravity, etc.), increment the moment summation registers corresponding to the label assigned to the output pixel.

Consideration of these steps shows that the algorithm must either “remember” the neighborhood values as it moves along in its raster scan, or it must check all the neighborhood values at each position, or some combination of these (remember some, check others). The SKIPSM paradigm, by its fundamental nature, “remembers” neighborhood information (in its state variables) and proceeds through the image in raster-scan order (or according to some other regular scan pattern). Therefore, it would appear to be ideally suited to the blob-labelling algorithm. But between stating this general idea and creating an efficient implementation, the algorithm designer must make many decisions which have great impact on algorithm efficiency and speed. For example, there is a trade-off between how much is remembered in the state variables (which require steps in the algorithm for updating) and spending time to do “fetches” from the input and/or output image buffers (which on some computing machines must be accessed through a relatively slow bus and/or with many “wait states”). This is a general question which can be decided only in the context of a particular algorithm and particular computing platform.
The decisions made in this paper follow the principle that information which must be accessed every time or almost every time should be kept in the state variables, while infrequently-needed information should be “fetched” from images as required rather than updated every time. There is not sufficient space here to explain the reasoning in detail.

3. Implementation Details - Row Machine

Two variables are assigned to the row machine:

1. rPat, which has the value 0 if the previous pixel in the row (pixel \(d\) in Figure 3) is a background pixel, and the value 1 if pixel \(d\) is not a background pixel.
2. rVal, which “remembers” the label just assigned to pixel \(d\), whether 0 or some other value.

The row machine proceeds by fetching the current pixel from the input image, determining whether it is above or below the chosen threshold, and branching accordingly. The reason for using a thresholding step is to allow the algorithm to be applied, without a pre-processing step, to binary images with levels 0 and 1, to binary images with levels 0 and 255 (or some other value for “white”), and to grey-scale images. The actual thresholded output values for the pixels are neither computed nor used in this algorithm.

4. Implementation Details - Column Machine

In the version of this algorithm discussed in this paper, part of the column machine is imbedded in each of the three branches of the row machine, as shown in Figure 4. Because each branch is associated with only a subset of the 32 possible neighborhood patterns, the work of searching the neighborhood to determine which pattern is present is completely eliminated. The column state variable cPat[i] is coded to indicate which of eight possible patterns is present in the previous row, and the row state variable rPat provides the same information about the previous pixel in the current row, for a total of 16 cases. (The other 16 cases occur in the leftmost path of Figure 4, and do not need to be distinguished because the results are always the same, independent of the case.) Therefore, all the information about the neighborhood pattern is present in the two variables cPat[i] and rPat, which can be used to decide which output value to choose and to compute the next values of cPat[i] and rPat. Figure 5 shows the 16 interesting cases, and the decisions made in each case by this implementation. Figure 6 has been contrived to show each of the 16 cases occurring on the same rather contorted blob.

5. Merging

In stepping through an image in raster-scan order, one often finds that blobs which appear to be separate when first encountered join together at some lower point in the image. For example, if there were an object in the shape of the letter U, the two
upright bars of the U would be encountered as separate blobs and given separate labels, but will be found to be joined together on a later scan. In this case one of the labels must be discarded and all the pixels of both branches of the U must be assigned the other label. But, when one is doing a raster scan, one can not go back to these pixels to change them. This might turn out to be wasted effort anyway because a third label might be encountered later on. In fact, one must sometimes wait until the very last pixel of the image to find out about merging. Therefore, this can not be done on the first raster-scan pass, in general.

The approach to the merging problem used in “pipelined” algorithms is to compile, on the first pass, a list of label pairs which must be merged, and then to process this list afterwards to eliminate duplicates, find all equivalencies, and produce a new (compacted) table of labels. The image can then be “repainted” on a second pass with these revised labels, using a lookup table. Or the image could be left unchanged, if only the moment statistics are needed, which are then merged using the same equivalence table.

6. A Difficult and Interesting Question

The extra step of adding label pairs to the merge list is time-consuming in two ways:
1. It involves extra steps during the first (image processing) stage.
2. If merge pairs are added when in fact they are not needed, because the same pair had already been added to the list, then extra processing is required to eliminate the duplications.

For both of these reasons, it is highly desirable to add label pairs to the list only when necessary. But the question remains, “Just how much is enough, but not too much?” This interesting and difficult question will be addressed on a case-by-case basis.

Note first that, because of the rules embodied in cases 10 through 17, all adjacent non-zero pixels in any given row will have the same label. (In the SKIPS algorithm, if d and e are non-zero, e is always given the same value as d.) Thus, adjacent non-zero pixels in cases 05, 06, and 07 will never need to have their labels merged. Therefore, in cases 01, 02, 04, 05, 06, and 07, all the pixels in the row above have the same label, and this label is the one assigned to e. Only in case 03 is there a possibility of more than one label. Furthermore, pixel e joins these two labels. Therefore, to be safe, these must be added to the merge list, as indicated in Figure 5.

Similarly, one can easily convince oneself that cases 12 and 13 require merging of c and d (because two blobs could be joining there), and that case 10 does not. But does case 13 also require the merging of a and d? And what about cases 11, 14, 15, 16, and 17? Pixels d and e have the same label, and the pixels in the row above have the same label, but are the labels in the two rows the same, or they have been merged previously? We show below that, in one way or another, they have already been merged.

7. An Unusual Reversed-Time Finite-State Machine

In order to tell whether the labels in the previous and current rows have already been merged, we must “track backwards” in the raster scan to see how the current cases might have arisen. In each of the questionable cases noted above, there are four cases (referred to here as precursors) which could have given rise to each current case, depending on the binary values of the two pixels to the left of a and d, as shown in Figure 7. And these precursors each have four precursors of their own. And so on. This appears at first glance to give rise to both a combinatorial explosion and an infinite regression. There is no infinite regression because the image itself is finite. And because there are only 32 possible precursor states, this is in fact a finite-state machine, the details of which can be traced in full.

Figure 8 shows the critical part of the state diagram for this finite state machine. (States not requiring merging are omitted.) Circles represent states which are not necessarily already merged, and the rectangles represent states that either provide merging or, by their nature, don’t require merging. The merges shown in Figure 7 for cases 3 and 12, and the c/d merge of case 12, are undoubtedly needed and are already built into the diagram. Questions remain about the a/d merges of cases 11 and 13 and...
the b/d merges of cases 14 through 17. The surprising and very satisfying answer is that none of these merges is needed. This will now be proven, using the reverse-time state diagram (Figure 8).

Procedure for using the reverse-time state diagram: Start at any of the circles. Follow each branch emanating from this circle, and then follow each branch emanating from any circles reached by those branches. And so on. Terminate each path when it reaches a rectangle (implying that the merging has taken place at the state represented by the rectangle). Do this for all possible paths in the diagram.

Results: Most paths end in rectangles, but there are four “infinite” loops which do not terminate in rectangles — 17-17-17-..., 13-14-13-14-..., 16-15-16-13-15-..., 17-16-15-17-16-13-15-..., and various joinings-together of sequences of these loops. Preliminary conclusion: “We have lost the battle.” The proposed merging scheme is insufficient.

But such loops cannot continue forever. Eventually they must reach back to the left edge (first column) of the image, if they do not terminate in a rectangle first. At the left edge, the precursor possibilities are very different - only the cases with xy = 00 (see Figure 7) can occur. (The row initial-
ization procedure behaves as if the “virtual” pixels beyond the image are all background.) Examination of Figure 7 or Figure 8 shows that for each circle, the “00” path leads to a rectangle. Therefore all loops terminate at the left edge, if not before, and the proposed merging scheme is sufficient. Further examination (not presented here) shows that none of the merge steps in the proposed scheme can be eliminated unless some other types of merging are added. Final conclusion: “We have won the war.” This scheme is both sufficient and reasonably efficient, in that it produces all the necessary merge pairs but produces relatively few redundant merge pairs. This merging scheme is the one used in this paper.

8. Merge List Processing

The merge list processing step sorts the merge pairs into numerical order according to the first number of the pair (which is always smaller than the second number, because of the way the merge pair generator works), eliminates duplicate pairs, and then proceeds recursively to find sets of labels which mutually touch each other somewhere in the image. It then creates a small lookup table to map all labels in a set into a single label. This is repeated for all of the sets. This equivalence table can be used to “repaint” the image (during a second pass) so that all parts of a given blob have the same label. This can be illustrated by the example at right, which involves an image in which 15 labels have been created and there are 7 merge pairs on the list.

Step 1: Sort the raw merge list into numerical order according to the first number in each pair. Here there is one duplicate 2-5 pair, which is deleted.

Step 2: Form equivalence sets by searching through the pairs. In this case there are three equivalence sets. Assign one of the labels (usually the lowest number) to all the labels in the set.

Step 3: Starting with a “transparent” lookup table (each label maps into itself), replace the eliminated labels with their equivalent values, as determined in step 2. This results in the “Uncompressed Lookup Table” shown for the example. The changed values are highlighted.

Step 4 (optional): In general after step 3, there will be gaps (unused labels) in the resulting set of labels. Here there are six unused labels. The gaps can be eliminated by shifting the other label values down to fill the gaps. The “Compressed Lookup Table” shows this process for the example.

9. Timing Comparisons

The timing figures given below include both the first (image processing) pass and the merging step, which may vary from essentially nothing (no merges) to a considerable time, depending strongly on the complexity of the image being analyzed. In most cases, processing the (typically) small one-dimensional merge list is much faster than processing the (large) two-dimensional image array. Both algorithms used in this comparison operate on grey-scale images, and compute all blob moments through second order, plus a number giving the average grey level of each blob. The algorithm used for speed comparisons with SKIPSM is well known: First, the image is run-length encoded, a very fast operation because it is very simple. The list of run-length start and stop points is then analyzed for overlaps, and labels are assigned accordingly. For simple images - a few large blobs, mostly convex - the run-length list is very short and therefore this stage of the processing is also very fast. One should not expect any pipelined algorithm to be as fast as this on simple images, because the pipelined algorithm must make complex decisions about each pixel in the image, whereas the run-length approach often deals with pixels in large groups. The question then becomes, “Are there any classes of images for which the SKIPSM approach is comparable or better?” Here are some examples to illustrate this point:

Timing is based on averaging over 100 repetitions of the algorithm, using a 266 mHz AMD K6 with 128 MBytes of memory running Microsoft Windows 95B. All of the images are 640x480.

Example 1:
Empty image (background only). In this case, neither algorithm has much to do, so this test shows the extra time it takes for the pipelined algorithm to determine, on a pixel-by-pixel basis, that there is “nothing to do” and to write a zero value to the output image buffer. The run-length algorithm does not have to write to the output image as it reads in the input. Instead, this is done during the “repainting” pass.

- SKIPSM 73 msec/image
- Run Length 31 msec/image
Example 2:
Image with dozens of small dots (Figure 9).

This image puts greater stress on both algorithms. The time for the run-length algorithm increases by 35% and that for the SKIPSM algorithm by 66%, so the run-length algorithm further increases its lead.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time (msec/image)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SKIPSM</td>
<td>121</td>
</tr>
<tr>
<td>Run Length</td>
<td>42</td>
</tr>
</tbody>
</table>

Example 3:
Image with hundreds of vertical streaks (Figure 10). This generates a very large increase in the number of “runs.” This affects the run length code much more than it affects the SKIPSM algorithm, which is actually faster for this example than for the previous one.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time (msec/image)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SKIPSM</td>
<td>98</td>
</tr>
<tr>
<td>Run Length</td>
<td>120</td>
</tr>
</tbody>
</table>

This shows that, with a sufficiently complex image, the run-length method can “bog down” because of a very long list of run lengths which must be processed, and that the SKIPSM approach may be useful in such cases.

An modified version of the SKIPSM algorithm is under development. A fairly small change in the inner-loop code will allow the algorithm to recognize when it is positioned in the interior of a large blob, causing it to bypass many logic and state-updating steps. This is expected to speed up operation in cases such as Figure 9, and with large blobs such as are sometimes encountered in the machine tool and automobile industries. Whether this will make it competitive with the run-length algorithm in such applications remains to be seen.

10. The 4-Connected Blob-Labelling Algorithm

For this operation the neighborhood consists only of pixels b, d, and e (see Figure 3b). There are only eight cases: the four trivial cases in which e is set to zero, and four other cases where a label must be chosen. When both b and d are background, a new label is created and assigned to e. When b is background but d is not, d’s label is assigned to e. When d is background but b is not, b’s label is assigned to e. When both b and d are not background, d’s label is assigned to e and, if the labels for b and d are different, the label pair b,d is added to the merge list. In general, there will be relatively more time spent on generating the merge list for 4-connected processing than for the 8-connected case, because many more label pairs must be tested. For example, for every pixel on the inside of a large blob, there must be a test to see if b and d are the same. The smaller neighborhood does not provide enough information to eliminate redundant merge pairs, as was possible in the 8-connected case.

For the actual operation, the same state variables, rPat, rVal, and cPat[i], are used. The row machine is similar to that of the 8-connected case, using state variables rPat and rVal in the same way. For the column machine, cPat[i] is replaced by cVal[i],
which stores the output label of pixel $b$, so that fetches from the output image are eliminated. This, along with the simpler rules, can be expected to save some time.

11. Conclusions

The hoped-for speed advantage provided by SKIPSM for many other classes of operation has not extended to connected component analysis, although there are cases involving highly complex images in which it is superior to the run-length approach with which it was compared in this paper. This is not so much due to deficiencies of SKIPSM as to the great efficiency of the run-length algorithm. Planned modifications to the SKIPSM algorithm may change this picture somewhat.

12. References