Automated generation of efficient code for grey-scale image processing
Frederick M. Waltz
2095 Delaware Avenue, Mendota Heights, MN 55118-4801 USA

ABSTRACT
The SKIPSM (Separated-Kernel Image Processing using finite-State Machines) paradigm has been extended with excellent results to grey-scale morphology with arbitrary flat structuring elements. But casual users can not be expected to master the techniques for creating SKIPSM implementations for user-specified SEs, thus limiting the usefulness of the technique.

This paper addresses that limitation by providing a completely automated procedure for generating the SKIPSM implementation for a wide range of grey-scale image processing operations in addition to grey-scale morphology, given only the definition of the neighborhood over which the computations are to be made. These neighborhoods need not be square or rectangular, but can be made up of arbitrary collections of contiguous or non-contiguous pixels. Examples of the operations that can be performed include grey-scale dilation and erosion with flat structuring elements, area sum or arithmetic mean, geometric mean, etc. In effect, this paper provides the basic structure for a computer program to write efficient computer code in a target language such as C. This code-generating program could be written in almost any computer language, but because it involves both list processing and some backtracking, Prolog would be an excellent choice.

This technique can be extended to three- and higher-dimensional grey-scale morphology without great difficulty.

Keywords: grey-scale image processing, automated code generation, separability, finite-state machines

1. INTRODUCTION
The SKIPSM paradigm (Separated-Kernel Image Processing using finite-State Machines) has been presented and developed in a series of papers. SKIPSM has recently been extended with excellent results to grey-scale morphology with arbitrary SEs (structuring elements). But casual users can not be expected to master the techniques for creating SKIPSM implementations for user-specified SEs, thus limiting the usefulness of the technique.

This paper addresses that limitation by providing a completely automated procedure for generating the SKIPSM implementation for a wide range of grey-scale image processing operations which are based on dyadic associative combining functions, given only the shape of the neighborhood over which the computations are to be made. These neighborhoods need not be square or rectangular, but can be made up of arbitrary collections of contiguous or non-contiguous pixels. Examples of the operations that can be performed include grey-scale dilation (based on maximization), grey-scale erosion (based on minimization), area sum or arithmetic mean (based on addition), geometric mean (based on multiplication), etc. In effect, this paper provides the basic structure for a computer program to write efficient computer code in a target language such as C. This code-generating program could be written in almost any computer language, but because it involves both list processing and some backtracking, Prolog would be a good choice.

This technique can be extended to three- and higher-dimensional grey-scale morphology without great difficulty.

2. REVIEW OF SKIPSM PRINCIPLES
In the SKIPSM paradigm, two finite-state machines are used, usually but not always a row machine and a column machine. The input image pixels are read in the usual video raster-scan order, left to right and top to bottom. Each pixel is read once and only once. All information needed for later computations is contained (sometimes in highly compact form) in the state buffers of the finite-state machines. The column machine state buffer(s) must be initialized at the start of the overall operation, and the row machine state register(s) must be initialized at the beginning of each row. SKIPSM often provides speedups and simpler code, in comparison with traditional implementations. These speed improvements stem mainly from the separation of two-dimensional operations into two one-dimensional operations, but the pipelining, recursive operation, and finite-state machines also provide some speedup in many cases.

3. THE ROW-MACHINE GENERATING FUNCTION
The goal here is to provide a completely automated procedure for generating compileable C code (or other computer code) for the chosen operation (dilation, erosion, area sum, geometric mean, etc.), given only the user-specified neighborhood definition. Perhaps the easiest way to understand what is meant here by “neighborhood” is to think of a morphology...
Figure 1 gives an example used in the development of the theory for the Row Machine Generating Function.

**Procedure:** Start with the user-defined rectangle, which has m rows and n columns. Let dark squares represent pixels within the m×n rectangle which are not part of the computation neighborhood, and associate the value 0 (zero) with these. Let white squares represent pixels which are part of the neighborhood, and associate the value 1 (one) with these.

Note that this does not imply that binary processing is being done. Is and 0s are used only to indicate that the given pixel belongs or does not belong to the neighborhood.

Eliminate all duplicate rows and put the remaining rows in some appropriate order. (Suggestion: Think of them as binary numbers and sort them in increasing numerical order.) Call this the row basis set. Assume that there are q rows in the basis set, where q may be as small as 1, if all the rows are the same (the neighborhood is the whole rectangle) or as large as m, if all the rows are different.

Assign the names R1, R2, ..., Rq to the basis rows.

Fig. 2 shows the first step of the automated procedure for row machines. This is NOT a block diagram of the final code, but only the first step in the generation of the code. Pick each of the rows in turn and generate its row machine, as follows:

Consider row k. Set up the Row State as a set of (n-1) registers, with the temporary names RSk_n-2, RSk_n-3, ..., RSk_0, plus an output variable Rk. Create a sequence of statements of the form shown below, substituting 0s or 1s as the parameter (after the |) in the Generator Function g in accordance with the corresponding pixel in row k. P is the input pixel grey value:

\[
Rk = g(P, RSk_0 | r_{k, 0}) = g(0, P, RSk_0 | 0) \\
RSk_0 = g(P, RSk_1 | r_{k, 1}) = g(P, RSk_1 | 1) \\
RSk_1 = g(P, RSk_2 | r_{k, 2}) = g(P, RSk_2 | 1) \\
\ldots
\]

To complete the code for this row machine, the Generator Function g must be evaluated according to the rules shown in Figure 3. The null variable plays a special and essential role here: When the input to a state register is null for whatever reason, that register itself is eliminated, and passes the null on to the next-stage g-function. This is known as propagating the null, and is important in reducing the number of computing steps. Start at the bottom of the list, applying the definition of g. Then move up the list, propagating the null as far as it will go. Figure 4 shows a graphical representation of the application of this rule to
For this example, the null propagates through RSk_n-2, but no farther.

Finally, the null registers are eliminated and the code is written in almost-final form. (The state registers will be renamed later.) This gives the code steps shown below, also shown in graphic form in Figure 5. The numbers in the ovals indicate the sequence of operations.

4. COMBINING ROW MACHINES TO SAVE COMPUTATION TIME

When there is more than one row machine (i.e., when all the rows of the neighborhood are not the same), a new possibility arises: Instead of simply computing all the row machines separately, it is always possible to reduce the number of state registers by combining registers which are identical. This will be illustrated by the 5x5 example shown in Figure 6. There are three basis rows, the step-by-step development of which is shown. To eliminate a step, the row pixels (shown separately in Figures 2 and 4) have here been moved directly into the function boxes in Figure 6. This is every bit as intuitive a representation, because the little black and white boxes in both cases are shown in the same order as in the basis row diagram. This procedure will be followed form now on.

The three individual row machines have a total of nine registers, but in fact only five registers are needed. Careful inspection shows that …

1. The first registers of the machines are identical; each has the input pixel P as its input. (States are identical if and only if their inputs are the same. It has nothing to do with where or how many places the resulting value is sent.) The letters under the registers are explained below.
2. By inspection, RS2_1 and RS2_0 are identical to RS3_2 and RS3_1, respectively. Both get their inputs from the second of two function boxes. Therefore, RS1_1, RS2_2, RS3_3 can be merged into a single state, which is here arbitrarily assigned the name RS4. RS1_0 is unlike any other state, and is assigned the name RS3. RS2_1 and RS3_2 are merged and assigned the name RS2. Similarly, RS2_0 and RS3_1 are, merged and named RS1. Finally, RS3_0 is assigned the name RS0. Figure 7 shows this result. Also shown is the column machine for this example, which will be discussed in the next section.

But, of course, computers are not very good at doing “by inspection” things that are obvious to humans. Therefore, a routine computer-based approach is needed. As a first step in this process, the same steps shown in the diagrams will be shown in terms of software code operations.

<table>
<thead>
<tr>
<th>Step</th>
<th>Code for Row Machine 3</th>
<th>Substitutions</th>
<th>Step</th>
<th>Rewritten Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>R3 = f(P, RS3_0);</td>
<td>Let RS3_0 = RS0</td>
<td>1</td>
<td>R3 = f(P, RS0);</td>
</tr>
<tr>
<td>2</td>
<td>RS3_0 = f(P, RS3_1);</td>
<td>Let RS3_1 = RS1</td>
<td>2</td>
<td>RS0 = f(P, RS1);</td>
</tr>
<tr>
<td>3</td>
<td>RS3_1 = f(P, RS3_2);</td>
<td>Let RS3_2 = RS2</td>
<td>4</td>
<td>RS1 = f(P, RS2);</td>
</tr>
<tr>
<td>3</td>
<td>RS3_2 = f(P, RS3_3);</td>
<td>Let RS3_3 = RS4</td>
<td>5</td>
<td>RS2 = f(P, RS4);</td>
</tr>
<tr>
<td>5</td>
<td>RS3_3 = P;</td>
<td></td>
<td>8</td>
<td>RS4 = P;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step</th>
<th>Code for Row Machine 2</th>
<th>Substitutions</th>
<th>Step</th>
<th>Rewritten Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>R2 = RS2_0;</td>
<td></td>
<td>3</td>
<td>R2 = RS1;</td>
</tr>
<tr>
<td>2</td>
<td>RS2_0 = f(P, RS2_1);</td>
<td></td>
<td>4</td>
<td>RS1 = f(P, RS2);</td>
</tr>
<tr>
<td>3</td>
<td>RS2_1 = f(P, RS2_2);</td>
<td></td>
<td>4</td>
<td>RS2 = f(P, RS4);</td>
</tr>
<tr>
<td>4</td>
<td>RS2_2 = P;</td>
<td></td>
<td></td>
<td>RS4 = P;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step</th>
<th>Code for Row Machine 1</th>
<th>Substitutions</th>
<th>Step</th>
<th>Rewritten Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>R3 = RS1_0;</td>
<td></td>
<td>6</td>
<td>R3 = RS3;</td>
</tr>
<tr>
<td>2</td>
<td>RS1_0 = RS1_1;</td>
<td></td>
<td>7</td>
<td>RS3 = RS4;</td>
</tr>
<tr>
<td>3</td>
<td>RS1_1 = P;</td>
<td></td>
<td></td>
<td>RS4 = P;</td>
</tr>
</tbody>
</table>

Care must be taken in putting the steps in a correct sequence, to insure that needed register values are read out before they are overwritten. The rule here is easy: All appearances of a variable on the right side of the expressions must precede its (sole) appearance on the left side of its expression. Sorting the statements in this order gives, finally, the following main loop code, ready for compilation (once the function f is defined). Examples for two definitions of f are also shown.

<table>
<thead>
<tr>
<th>Combined Generic Row Machine</th>
<th>Combined Row Machine for Dilation</th>
<th>Combined Row Machine for Area Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>R3 = f(P, RS0);</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>RS0 = f(P, RS1);</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>R2 = RS1;</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>RS1 = f(P, RS2);</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>RS2 = f(P, RS4);</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>R3 = RS3;</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>RS3 = RS4;</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>RS4 = P;</td>
<td>8</td>
</tr>
</tbody>
</table>

Now that the reader has been given an idea of both the goal of the row-machine combination process and the steps involved, an automated approach to combining state registers and assigning register names will be described.

The concept is straightforward: Use a software simulation of the individual row machines to determine their actual behavior under an applied sequence of simulated input pixels. Both the input pixels and the simulation must operate with literal rather than numerical values, so that the results can be compared unequivocally. The number of literal symbols to be applied to the simulated machine must equal the number of pixels in the basis rows. For the example used in Figure 6, five symbols (a, b, c, d, e) are required. The results of applying this technique to this example are shown below each register in Figure 6. For clarity and to save space, the f operators are omitted and the literal values are merely concatenated. Thus, the literal quantity “defg” is defined in terms of the associative dyadic operator f as f(g, f(f(e, f(d)))) = f(d, e, f, g). If f is the addition operator, for example, “defg” becomes d + e + f + g. Et cetera.

This procedure has the additional advantage that it checks that the output of the machine is as desired. For example, the simulation for the machine for row 2 produces “cde,” which correctly corresponds to the three middle pixels of the row. Next, the results of this procedure will be processed to eliminate extra states and assign final names to the remaining states. The same example will be used to illustrate the procedure. See the table on the next page.
5. COLUMN MACHINES

Compared to the row machines, column machines are very simple and straightforward. Figure 9 shows the general pattern: Let the number of rows in the neighborhood be denoted by $m$. Establish a column state buffer with $(m - 1)$ rows and a number of columns equal to the image width. (Depending on the operation, these registers might have to have word lengths greater than that of the image pixels.) Establish a structure with $(m - 2)$ function blocks $\varphi$ feeding all but the highest-numbered (leftmost) column state buffer $CS_{m-2}[i]$. The row machine output corresponding to the top row of the neighborhood is fed directly to the first register $CS_{m-2}[i]$. The row machine output corresponding to the second row is fed to first $\varphi$-block. This continues in the pattern shown in Figure 9. The numbers in boxes give the sequence in which the steps are to be performed. As always, one must "read before writing."

With one exception noted below, all column machines have precisely this form. The corresponding main loop code is as follows:

Step 1. Make list of the literal values, retaining the associated temporary state names.

Step 2. Sort this list in descending alphabetical order. Tag duplicates. (Here, the tag ## is used.)


Figure 7 shows a spreadsheet-based simulator for computing the literal register contents. This particular spreadsheet is designed for rows with seven pixels. Any number of row patterns can be analyzed, one at a time, and the results saved for sorting using the above procedure. This particular simulation is for the "O" example given later (Figure 14). Different spreadsheet are needed for other neighborhood widths, but modifications are easily made. Example spreadsheet are available on request.
6. Review of the Complete Procedure

All steps of the procedure for automated code generation will now be reviewed, with the help of another example.

**Step 1. Create the row basis set.**

Step 1a. Specify a neighborhood. The one in Figure 10a will be used for this example.

Step 1b. Sort to find duplicates, thus: Assign binary 1s to members of the specified neighborhood and 0s to the other. Sort the resulting binary numbers. See Figure 10b.

Step 1c. Delete duplicates. Assign row machine names R1, R2, …, Rq, where q is the number of basis row patterns. See Figure 10c.

Step 1d. Also make a table telling which row basis pattern corresponds to each row of the neighborhood. See Figure 10d.

**Step 2. Create the individual row machines.**

Step 2a. Set up the three generator functions. Apply the row pattern data. See Figure 11a.

Step 2b. Apply the generator definition and simplify. Simulate machine operation and append literal results. See Figure 11b.

**Step 3. Combine the row machines.**

Step 3a. Make a list of the simulated literal results and the associated temporary state names. See Figure 12a.
Step 3b. Sort the list in descending alphabetical order. Tag duplicates for deletion. See Figure 12b.

Step 3c. Assign final state names. See Figure 12c.

Step 3d. Rewrite the row machine code, eliminating redundant states, replacing the rest with the new names, and reordering, thus:

Combined Row Machine

Step 4a. Write the basic column machine code.

Step 4b. Use the table of row patterns to insert the correct row machine outputs. Here the function \( \tau_{(i)} \) represents the row machine output appropriate to row \( i \) of the neighborhood.

See Figure 10d for the table for this example.

Column machine code in terms of row machine outputs

The diagram for this example is shown in Figure 13.

7. An Additional Example

Figure 14 gives the first steps of the final example, a 9x9 “O”-shaped figure – the desired neighborhood, determination of the row basis set, and creation of the table for determining which row machine output feeds which stage of the column machine. Fig. 15 shows the four individual row machines, along with the literal register contents. The code for these (after application of the rules for the generator function) is shown at right.
### Equivalence table for renaming the states

<table>
<thead>
<tr>
<th>Row Machine 1</th>
<th>Row Machine 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS1 = RS8;</td>
<td>R2 = RS1;</td>
</tr>
<tr>
<td>RS8 = RS3;</td>
<td>RS1 = ( f(P, RS2) );</td>
</tr>
<tr>
<td>RS3 = ( f(P, RS4) );</td>
<td>RS2 = ( f(P, RS3) );</td>
</tr>
<tr>
<td>RS4 = ( f(P, RS5) );</td>
<td>RS3 = ( f(P, RS4) );</td>
</tr>
<tr>
<td>RS5 = ( f(P, RS6) );</td>
<td>RS4 = ( f(P, RS5) );</td>
</tr>
<tr>
<td>RS6 = ( f(P, RS7) );</td>
<td>RS5 = ( f(P, RS6) );</td>
</tr>
<tr>
<td>RS7 = P;</td>
<td>RS6 = ( f(P, RS7) );</td>
</tr>
</tbody>
</table>

**Italicized = redundant**

### Row-machine code with renamed states

#### Final row code plus column code

```plaintext
for (j = 0; j < NRows - 1; j++) {
    for (i = 0; i < NColumns - 1; i++) {
        << put code here >>
    }
}
```

This code is then dropped into the main loop of a shell program of the form

```plaintext
for (j = 0; j < NRows - 1; j++) {
    for (i = 0; i < NColumns - 1; i++) {
        // put code here
    }
}
```

There must also be an additional preliminary loop to initialize the column state buffers. The details are omitted here.
There are 27 statements in the main loop for this example, which may not seem too impressive until one considers that the neighborhood is 9x9, so that 81 pixels must be processed (or 60, if only pixels which belong to the neighborhood are accessed). Thus, the algorithm needs only 27 statements to do all the processing on 81 or 60 pixels – a meaningful reduction.

8. COMMENTS ABOUT EXECUTION SPEED

Depending on the size of the neighborhood, this method may require a rather large number of registers and steps. The number of column-state memory locations is fixed: (the number of pixels in an image row) times (the number of rows in the neighborhood, minus one). Each location is read from and written to once per image row. Therefore, the processing of a complete image row intervenes between each access to a particular memory location. This number of column-state memory locations is small compared to a typical image, so it is not very large from a storage standpoint. However, it is too large to fit into a typical microprocessor cache, so there will be a slowdown due to the repeated need to read from and write to the column state buffer.

If the row patterns are complex, a fairly large number of row-state registers may be required. But these are accessed repeatedly as the process moves along the rows of the image in raster-scan order, with no big intervening delays. Therefore, all of the row state registers can be expected to be in high-level cache most of the time, giving rapid access. Thus, a large number of row-state registers does not result in as serious an access-time penalty as one might initially expect. Of course, the more registers there are, the more code steps that must be executed. This is unavoidable.

It would be ideal for SKIPSM implementations of all types (not just this one), if the chip architecture allowed at least part of the high-level chip cache to be dedicated to functions specified by the user. For the class of applications described in this paper, the column state buffer would be left in cache at all times. For other SKIPSM applications, the lookup tables would be placed there. But of course this is not likely to happen in the near future, because such partial dedication of the cache space requires additional on-chip cache-management hardware, which vendors are unwilling to provide.

9. SUMMARY AND CONCLUSIONS

This paper has presented a completely routine technique (no user decisions required, once the neighborhood has been defined) for creating compileable code for a significant class of image-processing operations on arbitrary neighborhoods – including those with non-rectangular and non-contiguous patterns of pixels. The computer program to automate these ideas has not yet been written, but all the steps are fully described here, so that there are no fundamental difficulties preventing the writing of this kind of program. Because the actual steps used in developing the compileable code involve literal rather than numeric quantities, this code-generating program should be written in a language with strong symbol and string manipulation capabilities, such as LISP or Prolog.

It is believed by the author that this SKIPSM decomposition generally gives efficient code, in comparison with other approaches. To verify this, comparison tests are needed for each case and each type of operation. Fortunately, once the above-noted program is written, it will be easy to create many individual operations, for use in speed comparisons.

10. BIBLIOGRAPHY


Automated generation of efficient code for grey-scale image processing    SPIE Paper 3521-32    Frederick M. Waltz    SK19-9


