APPLICATION OF SKIPSM TO BINARY MORPHOLOGY
Frederick M. Waltz and Hichem H. Garnaoui

ABSTRACT
This paper describes the application of SKIPSM (Separated-Kernel Image Processing using Finite State Machines) to binary morphology. In comparison with conventional hardware-based and software-based approaches, SKIPSM allows implementation at higher speeds and/or lower hardware cost. The key theoretical developments upon which this improved performance is based are the separation of 2-D binary morphological image processing operations into a row operation followed by a column operation, the formulation of these row and column operations in a form compatible with pipelined operation, the implementation of the resulting operations as simple finite-state machines, and the automated generation of the finite-state machine configuration data. Some features of SKIPSM, as applied to binary morphology, are as follows:

- The structuring elements (SEs) can be large (25x25 and larger) and arbitrary (with “holes” and non-convex shapes).
- All types of morphology operations can be performed.
- Multiple related or unrelated SEs can be applied simultaneously in a single pipeline pass.
- Speed increases and/or neighborhood size increases by factors of 100 or more can be achieved.
- Corresponding “speedups” can be achieved in software-based implementations.
- Inexpensive off-the-shelf “chips” can be configured to carry out the same operations as expensive conventional hardware.
- The user specifies the SE or set of simultaneous SEs. All other steps are automated.

This paper includes some simple examples of the results and gives implementation guidelines based on SE size and shape.

KEYWORDS: image processing, binary morphology, real-time, implementations, finite-state machines, inspection

1. INTRODUCTION

Morphological image processing (described in more detail below) comprises a very powerful family of image processing operations which is not as widely used as its power would lead one to expect. The underutilization of morphology is due in part to a lack of user familiarity, which stems in turn from the slowness of the typical software implementations and the high cost of hardware implementations of these techniques. This paper describes a new approach to morphological image processing offering low-cost hardware implementations and dramatic increases in speed for both hardware and software implementations, in comparison with current software or hardware implementations.

The key idea is the application of a new technique called SKIPSM (Separated-Kernel Image Processing using Finite State Machines) to morphological image processing. An overview of SKIPSM is presented in a companion paper.1 Other applications of SKIPSM, including run-length encoding, grey-level morphology, and binary template matching are given in 2, 3, 4, and 5. Additional material on high-speed morphology implementations are given in 7 and 8. The theoretical developments upon which all of these applications are based are

- the separation of 2-D binary morphological image processing operations into a row operation and a column operation,
- the formulation of these row and column operations in a form compatible with recursive or pipelined realizations,
- the implementation of the resulting operations as simple finite-state machines (henceforth referred to as FSMs), and
- the automated generation of the FSM configuration data.

Some features of SKIPSM, as applied to binary morphology, are listed below. Note that SKIPSM does NOT involve the sequential application of small structuring elements (SEs) to get large SEs. Instead, it directly applies the large SE in one pipeline pass. This is important because only a tiny fraction of all possible SEs can be obtained by sequential decomposition.

- The structuring elements can be large (25x25 and larger) and arbitrary (with “holes” and non-convex shapes).
- All types of morphology operations can be performed.
- Multiple related or unrelated SEs can be applied simultaneously in a single pipelined pass. In one early trial of the technique, 16 different SEs of various sizes were applied simultaneously.
- Speed increases and/or neighborhood size increases by factors of 100 or more are achieved.
- Corresponding “speedups” can be achieved in software-based implementations.
- Inexpensive off-the-shelf “chips” can be configured to carry out these operations.
- The user specifies the SE or SEs. All other steps are automated.

This paper includes some simple examples of the results and gives implementation procedures and feasibility guidelines based on SE size and shape.
2. FSM ESSENTIALS AND SEPARABILITY

The first companion paper includes a discussion of SKIPSM and FSM fundamentals and implementation architectures. The primary emphasis of this paper is on binary erosion. All 2-dimensional binary erosion operators can be separated into 1-D row and column operations, as follows: (See 1)

- Let K and L denote the numbers or rows and columns, respectively, in the binary structuring element.
- Individual pixel functions $q_{ij}$, $i = 1, 2, \ldots, K; j = 1, 2, \ldots, L$: Let the pixels which are part of the SE be assigned a Boolean value of 1, and the “don’t care” pixels a Boolean value of 0. Let the image pixels be assigned values of Boolean 0 for “black” and Boolean 1 for “white.” Then, for a given placement of the SE on the image, the (Boolean) result of the pixel test can be stated as $q_{ij} = (\text{image pixel value}) \lor (\neg (\text{SE pixel value}))$.
- Overall output function $F$: Let $\cdot$ denote the Boolean AND operation. The required function $F$ for binary erosion is

\[ F = q_{11} \cdot q_{12} \cdot \ldots \cdot q_{1L} \cdot q_{21} \cdot q_{22} \cdot \ldots \cdot q_{2L} \cdot q_{31} \cdot \ldots \cdot q_{K1} \cdot q_{K2} \cdot \ldots \cdot q_{KL}. \]

That is, the ANDing together of all the individual pixel functions.
- Separation: Because of the associativity of the AND function, this can be written as

\[ F = (q_{11} \cdot q_{12} \cdot \ldots \cdot q_{1L}) \cdot (q_{21} \cdot q_{22} \cdot \ldots \cdot q_{2L}) \cdot \ldots \cdot (q_{K1} \cdot q_{K2} \cdot \ldots \cdot q_{KL}). \]

- The bracketed expressions are the row combining functions $R_1, R_2, \ldots, R_K$. These meet the recursion condition.
- The column combining function $C$ has the form $R_1 \cdot R_2 \cdot \ldots \cdot R_K$. This also meets the recursion condition.

Therefore, the operation of binary erosion meets not only the separability condition but the recursion conditions for row and column machines. Therefore, binary erosion can be separated and the resulting row and column functions can be pipelined.

3. BINARY MORPHOLOGY

Most recent textbooks on machine vision discuss morphology in some detail, and therefore a detailed development will not be given here. The most widely used type of morphological image processing is called binary morphology, because it involves only binary (1 bit) input and output images and binary operations (1-bit logical operations). Recently, grey-level morphology (discussed in companion paper 4) has come into increasing use. This paper concentrates on binary morphology.

The following brief discussion should help introduce the basic ideas of binary morphology: The morphological operation called binary erosion applies a “shaped object” (called a structuring element, or SE) to input images to find all places where the SE “fits” in the image. The SE is said to “fit” at a certain position if the input image has “white” pixels at all positions corresponding to “white” SE pixels. Where it “fits” the output image is set to “white” (usually the grey level 255, but sometimes the grey level 1). Where it does not “fit” the output image is set to “black” (almost always grey level zero). Figure 1 illustrates this for a simple example.

In SKIPSM implementations, the SEs must be embedded in a rectangular neighborhood, as indicated at lower left in Figure 1. The hatched pixels in this representation of the SE can be thought of as “don’t care” pixels. A heavy outline is used to indicate the “center” of the SE. This center pixel is placed successively at each pixel of the input image, and the neighborhood pixels are tested to see if the SE “fits” there. The thicker outlines in the input image indicate the SE placed at pixel A and at pixel G. In this example, the SE fits at positions A, B, C, D, and the six other places shown in white, causing the output pixels at these points to be white. It does not “fit” at E, F, G, etc., causing the output pixels at these points to be black.

The question remains as to what to do about the border regions — in this example the first three and last three rows and columns of the input image. If one attempts to place the center of the SE on one of these border pixels, part of the SE extends outside the image.
There are two ways to handle these regions. Either approach can be implemented using FSMs:
1. If only those instances of the structuring element which fit completely within the image are to be found, then the area surrounding the image is set to black, implying no “fits” in the border region. This can be accomplished in pipelined hardware implementations by sending only black pixels in the regions outside the image (i.e., the horizontal retrace and vertical blanking intervals).
2. If we wish to include in the output image not only those pixels at which the SE fits completely, but those border pixels at which the part of the SE which is overlapping the input image fits, then we send only white pixels in the horizontal retrace and vertical blanking intervals. This guarantees that the part of the SE outside the image will always “see” white pixels, and therefore report a successful “fit.”

The other standard binary morphology operations (dilation, opening, and closing) can be defined and implemented in terms of the erosion operation, or can be implemented directly with SKIPSM. Space limitations permit a detailed presentation of only one operation, and the erosion operation has been selected. Therefore, the other standard operations will not be described here.

Further discussion: Binary morphology is similar to binary template matching except that all the “black” SE pixels are “don’t care” pixels, whereas in binary template matching the black template pixels must also be matched. Binary erosion and dilation can be implemented at video rates using commercially-available board-level morphology hardware, such as the Datacube SNAP board. However, this board is limited to 3-by-3 (or 1-by-9) SEs. In most applications, much larger structuring elements are needed. To obtain these, one must either use multiple SNAP-like boards (more expensive), or a large-neighborhood linear convolver board (more expensive), or multiple passes through a single SNAP-like board (slower). The sequential approach places severe restrictions on the SEs that can be achieved. The simple example shown in Figure 2 would require five SNAP-like boards for a parallel (also called “tiled”) decomposition. If the user is clever or lucky enough to devise a sequential decomposition of the SE, then three SNAP-like boards or three passes would suffice. Larger and more complex SEs would require additional boards or additional frame times. In contrast, this SE and other much larger and much more complex SEs can be implemented using the SKIPSM approach in one frame time. If the standard SKIPSM architecture1 were to be used, the FSM implementation requires orders of magnitude less hardware than would be required if the SNAP-type architecture were expanded to cover larger SEs. (A SNAP-like device capable of carrying out a general 9-by-9 erosion operation in one frame time would require 81 comparators and a comparator lookup table with an 81-bit address space and hence $2^{81} = 2,417,851,639,229,258,349,412,352$ Bytes!)

A mathematical note: Morphological image processing is fundamentally nonlinear in nature, and is in a sense the polar opposite of conventional linear image processing. Linear processing is based on the convolution operation, a weighted summation over a neighborhood of pixel values. This definition includes even the Fourier transform, in which the neighborhood is the whole image. In mathematical terms, the underlying structure for linear operations is the Hilbert space, usually referred to as the $L_2$ space. The distance measure for this space is the familiar Euclidean distance, the square root of the sum of the squares of the individual coordinates of a vector. In contrast, morphology is based on the $L_\infty$ space, in which the distance measure is the maximum of the individual coordinates of a vector. A discussion of this concept in the context of a family of Banach morphologies ranging from $L_2$ through $L_p$ to $L_\infty$ is given in 6. Additional information about high-speed morphological implementations is given in 7 and 8.

4. EXAMPLES

Column machine for the 9x9 “plus sign” structuring element

Figure 2 shows the state transition diagram for a column machine for the “plus sign” SE. Row machines are discussed later.

![State Transition Diagram for Column Machine](image-url)
Column machines for four similar 9x13 structuring elements

Figure 3 shows a very interesting result that could not have been predicted on the basis of previous arguments and examples. The first SE, presented for comparison, is a simple rectangle with no black pixels. This gives a very simple column FSM. The other three SEs have the same number of black pixels, and differ only in the placement of these pixels. The FSM for the “U” is only slightly more complicated than that for the plain rectangle, but the FSM for the “H” is significantly more complex and that for the “Inverted U” is drastically more complex. How can this be explained? The key property that the first two share and that is lacking in the other two is monotonicity. This property will now be defined:

Consider any two SE row patterns A and B. A is said to be greater than or equal to B (written \( A \geq B \)) if A has white pixels everywhere B has white pixels. (Other relationships, such as >, <, ≤, and = can also be defined analogously.) Clearly, patterns A and B in Figure 3 are such that \( A \geq B \). Now consider a more general case in which the SE has row patterns A, B, C,..., P, Q starting with A at the top and ending with Q at the bottom. This SE will be said to be monotonic if and only if Q \( \geq P \geq ... \geq C \geq B \geq A \).

Property 1: A monotonic SE always results in a column FSM with the minimum number of states. If the SE has K rows, there will be precisely K states. The state transition diagram consist of these states in a “string” plus some additional paths, similar to the second state transition diagram in Figure 4. There is always a loop from the last state back to itself.

Figure 3. Comparison of the column FSMs resulting from four SEs of the same size.

Consider any two SE row patterns A and B. A is said to be greater than or equal to B (written \( A \geq B \)) if A has white pixels everywhere B has white pixels. (Other relationships, such as >, <, ≤, and = can also be defined analogously.) Clearly, patterns A and B in Figure 3 are such that \( A \geq B \). Now consider a more general case in which the SE has row patterns A, B, C,..., P, Q starting with A at the top and ending with Q at the bottom. This SE will be said to be monotonic if and only if Q \( \geq P \geq ... \geq C \geq B \geq A \).

Property 1: A monotonic SE always results in a column FSM with the minimum number of states. If the SE has K rows, there will be precisely K states. The state transition diagram consist of these states in a “string” plus some additional paths, similar to the second state transition diagram in Figure 4. There is always a loop from the last state back to itself.
Property 2: A SE which is monotonic up to a point (say, row \(j\)) results in a column FSM with its first \((j+1)\) states in a simple string. After this, branching occurs, and additional states are spawned. In general, the more rows there are below the point at which monotonicity fails, the more states are spawned. A “don’t care” at the final position never produces branching.

Conjecture: An upper bound on the number of states can be obtained as follows: Let \(K\) denote the number of rows in the SE, and let \(H\) denote the number of rows at the top of the SE that satisfy the monotonicity requirement. Let \(G = K - H\) denote the number of rows at the bottom of the SE for which monotonicity fails. Then \(N \leq K+G\frac{G-1}{2}\), where \(N\) denotes the number of states in the FSM. In fact, \(N\) is usually much smaller than this bound.

If we apply these ideas to the examples encountered so far, we see that the plus sign SE was monotone through the first six rows, giving us a string of 7 states before branching commences. At this point there are only 3 rows to go before the pattern ends, and we encounter two levels of branching. For the plus sign, \(N = 13\), and the upper bound on \(N\) is \(9+3\times3\times2/2 = 18\). The U is completely monotonic, and we encounter no branching and no additional states. For the U, \(N = 13\), and the upper bound on \(N\) is \(13+0\times0\times(-1)/2 = 13\). The H SE has 8 rows before monotonicity fails, and we encounter 4 levels of branching and 36 additional states (not shown). For the H, \(N = 49\), and the upper bound on \(N\) is \(13+5\times5\times4/2 = 63\). Monotonicity fails in the Inverted U after only 3 rows, resulting in 9 levels of branching and 292 additional states. For the Inverted U, \(N = 305\), and the upper bound on \(N\) is \(13+10\times10\times9/2 = 463\).

5. WHAT TO DO WHEN THE FSM GETS TOO BIG

All four of the SEs shown in Figure 4 are easy to implement in the SKIPSM architecture, but these examples should help to emphasize two points:

1. There are limits to what can be done, depending on the size and lack of monotonicity of the SE.
2. For very large SEs, which tend to have a large number of states anyway, SEs with a high degree of monotonicity should be used, if possible. Where this is not possible, one might split the overall SE into pieces which are in themselves monotonic or nearly monotonic, and then combining these using a “tiling” approach, as in 7. This approach has the disadvantage of requiring a special configuration of delays for each application, which violates the spirit of the generic SKIPSM architecture. Fortunately, “tiling” is seldom necessary because SKIPSM can implement rather large SEs.

For this example, if we split the Inverted U into a three-row SE followed by a ten-row SE, the two SEs are individually monotonic, with 3 and 10 states, respectively, for a total of 13 states. Thus, two very small FSMs running in parallel could be combined (with proper delay) using a single AND gate to produce the necessary result. One FSM LUT (lookup table) requires 3 input bits (1 for coded input, 2 for state) and 3 output bits (1 for output, 2 for state). The other FSM LUT requires 5 input bits (1 for coded input, 4 for state) and 5 output bits (1 for output, 4 for state). Alternatively, with appropriate splicing together of the various bits, it would fit into a single 8-by-8 LUT, as shown at right in Figure 4.
As in all FSMs, the definition of the state variable (or state vector) is the crucial design step. In fact, the key idea involved in the invention of this technique was the insight as to what form the state variable must take. By the nature of these problems and the fact that a pipelined stream of pixels is involved, we must keep track not only of image points at which a complete fit of the SE to the image is finally achieved, but also of fits in various states of partial completion. Figure 5 illustrates this for a simple 5x5 structuring element. The structuring element fits the most recent five rows (the bottom rows) of the raster-scanned input image. Therefore, the output at this position would be 1, indicating a fit. However, the last three rows fit the top three rows of the structuring element. We must also remember this fact, because the next two image rows might fit rows 4 and 5 of the structuring element, in which we must also produce a 1 output at that time. Finally, the most recent image row fits the top row of the structuring element, and we must also remember this fact, for similar reasons. For this example and at this moment in time, we thus have one fit just completed and two partial fits in progress, as well as two intermediate cases in which there has already been shown to be no fit possible. The state vector must be able to record such facts.

With this background, the choice of state vector should become clear. We will define a state vector of 1s and 0s, with as many bit positions as there are possible partial fits to be remembered, and assign one bit to each partial fit. For the example given in Figure 5, we need four bits to remember partial fits, plus one bit for the output. The leftmost bit will be set to 1 if and only if the most recent image row fits the first structuring element row. The next bit to the right will be set to 1 if and only if the two most recent image rows fit the first two structuring element rows. And so on. The ordering of the bits in the vector is chosen to follow the left-to-right and top-to-bottom order of the normal raster scan. Thus, the rightmost bit represents the most nearly complete pattern, etc. For the example in Figure 5, the state vector now (i.e., at the time of the current pixel shown on the diagram) is therefore \([1, 0, 1, 0]\) = \([\text{bit to indicate 1-row fit}, \text{bit to indicate 2-row fit}, \text{bit to indicate 3-row fit}, \text{bit to indicate 4-row fit}]\). Note again that the output bit is not part of the state vector.

![Figure 5. Example showing multiple simultaneous partial fits and failures.](image)

It is often useful for sorting, merging, and calculation purposes to work with integers rather than strings of bits. Therefore, a second way of representing the state has been devised: Consider the vector of bits to form an integer in the base-2 numbering system, with weights of 1, 2, 4, … assigned to the bit positions. The “natural” way to do this is to assign a weight of 1 to the rightmost bit, 2 to the next bit, etc. This was tried and found to be confusing, because it gives the lowest numerical values to the most nearly complete patterns. One alternative, reversing the order of the bits in the vector, thus violating the “natural” left-to-right direction of causality arrows in engineering diagrams and of written documents in Western languages, was found to even more confusing. Therefore, the binary weights are assigned in the reverse of the usual binary order: the leftmost is given a binary weight of 1; the next bit a binary weight of 2; and so on. Using these assigned binary weights, this vector for this example has a coded value or code of 5.

With 4 bits, it is clear that the state vector could represent as many as 16 states (including the zero or null state, in which no partial fits are present). The earlier examples have more than five rows in their structuring elements, and therefore more states are possible. Thus, if we had a structuring element with 17 rows, there could be as many as \(2^{17} = 131,072\) states. That many states would create very serious implementation problems for the kinds of hardware anticipated here. Fortunately, in almost all cases only a very small fraction of the possible states are actually needed. The only case in which all these possible states are needed is the one in which all the structuring element pixels are “don’t care” pixels, which clearly leaves us no structuring element and hence is a meaningless problem. The process by which the state representation value is reduced (compressed) below the number implied by the number of bits is discussed in later sections.

6.2 Preliminary definitions and discussion

It is not always sufficient to design the row machine to detect only the rows of the SE itself. For some SEs, combinations of rows can create additional patterns. Two or more of the four patterns in the SE shown in Figure 6 can be satisfied at the same time by particular image pixel patterns. Yet with a single output value, the row machine must tell the column machine all the row patterns of interest.
which are present at that position. This gives rise here to two additional “virtual” patterns.

The procedure for obtaining the set of distinct row patterns is as follows: Use 0s and 1s to represent “don’t care” and “white” SE pixels, respectively. Set up vectors representing the rows of the SE. Bitwise OR these vectors together in every combination — one at a time, two at a time, …, K at a time, being careful to retain the information as to which row vectors were included. Merge (i.e., eliminate) all duplicates in the resulting list, simultaneously keeping track of which patterns are supported by the combined pattern. For the example given in Figure 6, [A OR D] gives the vector [111 111 111], which is called E and added to the set. Similarly, [A OR C] gives the vector [111 111 111]. The same vector results from [B OR C] and [B OR D]. Therefore, only one of these must be retained as a distinct pattern. Therefore, merge [A OR C] with [B OR D] to get [A OR B OR C OR D], and associate this with the new vector F = [111 111 111]. Put the results in any convenient order. Assign a letter, number, or other designator to each member of the list. These are the distinct pattern categories that appear in as possible inputs to the column machine.

Let there be K rows in the structuring element. The (uncompressed) state vector will therefore have (K - 1) bits. Let M represent the number of distinct input patterns in the structuring element. Any patterns containing “don’t care” pixels are in fact pattern groups rather than simply patterns because more than one actual image row pattern maps into the same effective pattern. Consider the simple case of pattern group B of Figure 6: It has three “don’t care” pixels. Thus, there are $2^3 = 8$ ways that black and white image pixels can meet these conditions. One of these ways, all white, is the same as Pattern F, and hence is not part of the B group. Thus, Pattern B is actually a group of 7 patterns. And so on for the other pattern groups.

For the general case, let A, B, …, M denote the pattern groups. Let Z denote the null pattern group — the set of all input row patterns which do not fit any of the structuring element rows and therefore can do nothing to further the completion of any partially-completed fits. Z always produces a zero output and returns the system to the null state, in which no partial fits are underway, unless the SE has a partial fit. Z always produces a zero output and returns the system to the null state, in which no partial fits are underway, unless the SE has a partial fit.

For the general case, let A, B, …, M denote the pattern groups. Let Z denote the null pattern group — the set of all input row patterns which do not fit any of the structuring element rows and therefore can do nothing to further the completion of any partially-completed fits. Z always produces a zero output and returns the system to the null state, in which no partial fits are underway, unless the SE has a partial fit.

The procedure for obtaining the set of distinct row patterns is as follows: Use 0s and 1s to represent “don’t care” and “white” SE pixels, respectively. Set up vectors representing the rows of the SE. Bitwise OR these vectors together in every combination — one at a time, two at a time, …, K at a time, being careful to retain the information as to which row vectors were included. Merge (i.e., eliminate) all duplicates in the resulting list, simultaneously keeping track of which patterns are supported by the combined pattern. For the example given in Figure 6, [A OR D] gives the vector [111 111 111], which is called E and added to the set. Similarly, [A OR C] gives the vector [111 111 111]. The same vector results from [B OR C] and [B OR D]. Therefore, only one of these must be retained as a distinct pattern. Therefore, merge [A OR C] with [B OR D] to get [A OR B OR C OR D], and associate this with the new vector F = [111 111 111]. Put the results in any convenient order. Assign a letter, number, or other designator to each member of the list. These are the distinct pattern categories that appear in as possible inputs to the column machine.

Let there be K rows in the structuring element. The (uncompressed) state vector will therefore have (K - 1) bits. Let M represent the number of distinct input patterns in the structuring element. Any patterns containing “don’t care” pixels are in fact pattern groups rather than simply patterns because more than one actual image row pattern maps into the same effective pattern. Consider the simple case of pattern group B of Figure 6: It has three “don’t care” pixels. Thus, there are $2^3 = 8$ ways that black and white image pixels can meet these conditions. One of these ways, all white, is the same as Pattern F, and hence is not part of the B group. Thus, Pattern B is actually a group of 7 patterns. And so on for the other pattern groups.

Figure 6. Combinations of SE row patterns give rise to additional row patterns.

For the general case, let A, B, …, M denote the pattern groups. Let Z denote the null pattern group — the set of all input row patterns which do not fit any of the structuring element rows and therefore can do nothing to further the completion of any partially-completed fits. Z always produces a zero output and returns the system to the null state, in which no partial fits are underway, unless the SE has a full row of “don’t care” pixels. For the sake of brevity, the table entries and state diagram paths corresponding to the Z pattern are generally omitted from the diagrams here. These entries must be included in the actual lookup tables.

The procedure for generating the state transition table for column machines will now be outlined, using the example given in Figure 7 for illustration. (Row machines use essentially the same program.) Figure 8 shows a sample of the spreadsheet used to carry out the automated design procedure for this example. The spreadsheet has the advantage of being easier to understand than typical computer code. Important insights into FSM behavior have been obtained by watching this program running, and seeing FSM “behavior patterns” as they develop. The same spreadsheet program was used for many of the other examples in this paper and in the companion papers.1,2,3,4,5 This program is sufficient to generate even very complex FSMs but does require some user supervision. For faster and unsupervised operation and larger SEs, however, a direct implementation of the procedure in the C programming language has also been developed. These are discussed in more detail in 2.

Figure 7. Example used to illustrate automatic generation of state transition tables.
Automated procedure for generating FSM lookup tables

Step 1. Set up the table (i.e., matrix) of Support vectors. They are called support vectors because they show which inputs support the continuation of which partial fits. This matrix will have K columns, one for each row in the structuring element, and \((M + 1)\) rows, one for each distinct input row pattern (including \(Z\)). The zeroth row of the matrix, corresponding to input \(Z\), is all zeros in this case, because \(Z\) does not support the extension of any of the partially completed patterns. The first matrix row corresponds to input row pattern \(A\), the second to \(B\), etc. By inspection, an input pattern \(\geq A\) is needed to advance to step 1. Similarly, an input pattern \(\geq B\) is needed to advance to step 2. The logical rule for determining whether a given input pattern supports a given row of the SE is given in Figure 8. Note that the support vectors follow directly from the set of distinct input row patterns. In fact, the only reason for obtaining the distinct row patterns is to determine the support vectors.

Step 2. Begin the process of generating the state transition table by assuming that the initial state of the system is the null state — i.e., no “fits” are underway. (Other assumptions are possible. For example, one could assume that all possible partial fits are present initially, giving a state vector of all 1s.) With the state list initially empty, add the null state to the state list. That is, load all zeros (the state vector corresponding to the null state) into the “paste” area, shifted one place to the right, as indicated. This right shift is equivalent to a time delay of one pixel time (for row machines) or one video line time (for column machines). Thus, by shifting a current state vector one bit position to the right, we are converting it to the previous state. (The leftmost bit position is always 1, because we must consider each possible input as a candidate for starting a new fitting sequence.)

Step 3. For this assumed value of state, the Results area shows the Next State corresponding to each possible input, as well as whether the corresponding output is 0 or 1 (last column). The rule for determining the Results values is given in Figure 8. Note that the support vectors follow directly from the set of distinct input row patterns. In fact, the only reason for obtaining the distinct row patterns is to determine the support vectors.

Step 4. Examine the resulting states in the Results area for any states spawned by this state which are not already on the State List. If any states occur which are not already on the State List, add them to the State List. (Alternatively, append all the resulting states to the State List. Then, sort the list in Code order. Eliminate all duplicate states.)

Figure 8. Spreadsheet applied to the D structuring element.

Step 4. Examine the resulting states in the Results area for any states spawned by this state which are not already on the State List. If any states occur which are not already on the State List, add them to the State List. (Alternatively, append all the resulting states to the State List. Then, sort the list in Code order. Eliminate all duplicate states.)
Step 5. Append the complete block of Results to the end of the State Transition Table. This process of loading a State vector into the “paste” area and tabulating the results will be referred to as propagating a state. Each time a state is propagated, “check it off” somehow on the state list so that the system will not waste time by propagating it again.

Step 6. Examine the state list for any states not already propagated. From these, choose any state for propagation. (My program chooses the first unpropagated state on the list.) Load this State vector into the “paste” area, shifted right one place. Then go back to Step 3. Continue looping until all states on the State List have been propagated and no new states occur as the result of a propagation step. When there are no new states (which must happen eventually because this is a finite state machine), the table is complete.

Step 7. Sort the state table in any desired order (usually in order of increasing Code value). Assign State Numbers (consecutive positive integers, starting with zero) to the resulting State Table. In Figure 8, these are shown in the column to the left of the Code values. Create a final output table by replacing the Code values by the corresponding State Numbers. Sort this State Transition Table in the desired way — in order of increasing State Numbers (as shown in Figure 8), or in order of increasing input value, as needed for the particular implementation.

The second column and the last column of the state transition table are precisely the results needed to create the lookup tables for loading into the hardware, for the case in which the input bits correspond to the low bits of the RAM address. (For the other case, the table is simply resorted in order of input rather than state number.) The state transition table can also be used to create a state transition diagram, if desired. Such diagrams are for information only, to help the designer understand the FSM. Their creation has not been automated.

This D structuring element shown in the Figure 8 may not look particularly impressive to someone unfamiliar with the field, but in fact to implement this in one frame time using conventional 3x3 morphology boards but not using the FSM approach would require at least 17 boards, at a cost exceeding $45,000 list price. (This does not include the digitizer board, frame store board, feature extractor board, and controlling computer.) Alternatively, it could be implemented in nine frame times using two boards costing about $5,500. In contrast, the FSM approach allows implementation in one frame time using hardware costing a few hundred dollars.

### 7. ROW PATTERN DETECTORS/ENCODERS

The FSMs described above and in the companion papers are designed in many cases to run down the columns of an image using as inputs the “encoded” row patterns produced by a row machine. This section will describe various row encoding implementations.

One method of implementing row encoding is to use a nonlinear neighborhood board such as the Datacube SNAP board. The SNAP board can encode arbitrary 1x9 or 3x3 pixel patterns into up to 256 distinct output values. The encoding is embodied in a lookup table which maps each of the 512 possible combinations of nine binary values into a user-specified 8-bit value. In the situation described here, many different patterns are typically mapped into the same output value, to account for the “don’t care” pixels. Programming tools are available to facilitate the creation of these lookup tables. Of course, the SNAP board alone is limited to patterns nine pixels wide — a serious limitation. With difficulty, other boards (e.g., VFIR and MaxSP) can be used to extend the width. If such boards are already present in the system for other uses, then there is some reason to use them to assist in the row encoding. But lacking such boards, direct implementations using SKIPSM are both cheaper and offer much larger SEs. It follows that procedures for developing FSM-based row encoders are needed.

![State Diagram for Detecting Three 1x7 Horizontal Patterns Simultaneously](image)

Figure 9. State transition diagram for the simultaneous encoding of three 1x7 row patterns.
The most obvious approach is to use one or more additional FSMs running in parallel to encode the row patterns into appropriate coded values. One of the remarkable things about SKIPSM is that multiple individual row or column machines can be combined to form a single machine and loaded into a single RAM chip. Software has been developed not only to create individual row and column FSMs, but to combine them in arbitrary ways, resulting in separate or combined outputs at each pixel time. The only limitation on these kinds of combinations is that the number of states can become impractically large in some cases. Figure 9 shows the state transition diagram for one such combined FSM (deliberately kept very simple to reduce confusion) to encode three row patterns simultaneously. The numbers after the “/” mark are outputs, indicating that the corresponding pattern has been completed at the current pixel time.

In Figure 6 of the introductory companion paper, a row FSM is shown for the very simple case of 11 all-white SE pixels. As a second example of a row machine, consider a similar but slightly more complicated case: Let the SE have “rounded corners,” as shown in Figure 10. The standard SKIPSM implementation has only one row machine. Therefore, these three different row machines must be combined into one machine which simultaneously detects all three row patterns. The individual row FSMs and the resulting combined 3-pattern row machine are diagrammed in Figure 10.

The interesting (and to us amazing) thing about this combination is that it has so few states. The individual machines have 11, 11, and 12 states, respectively. The three machines running separately therefore have 11*11*12 = 1320 states. But the combined machine has only 31 states. If we take a 10-bit word length as a reasonable upper bound for representing the state, then we can have up to 1024 states before running out of state bits. We are far short of this in this example, implying that many additional distinct row patterns and/or much wider SE rows can be encoded in a practical system. As one more example, the row encoder for a 27-by-27 circular SE with nine distinct row patterns has only 209 states.

8. DIRECT HARDWARE IMPLEMENTATIONS OF ROW ENCODERS

In evaluating the SKIPSM approach to binary morphology, one should not overlook a comparison with direct implementations using custom integrated circuits. A configuration appropriate for EPLD or custom implementation of row encoders is shown in Figure 11, feeding into a single board or SKIPSM module implementing the column FSM. This approach offers wide or very wide row encoders.
without the limitations inherent in the use of 3x3 or 1x9 morphology boards. As shown by this figure, the row encoders are very simple and are related to the SE pixels in a very simple way. Therefore, programming an custom integrated circuit or EPLD to implement these is relatively easy. But a FSM-based row encoder is probably cheaper because of the low cost of volume-produced RAM chips. Furthermore, the FSM-based configuration is certainly easier to reprogram for other SE sizes and other image processing functions than EPLD chips.

Comparison should also be made with a direct implementation of the full SE in an EPLD or custom integrated circuit plus an external serpentine memory. (It would not be cost-effective to try to include the serpentine memory within the EPLD, given the current capabilities of these chips.) For the 15x15 “donut” used in Figure 11, the direct implementation has 225 gate cells with delays (one for each SE pixel), a 24-bit serpentine memory (ouch!), and a 225-input AND gate or AND-gate tree. This has the advantage of providing a simple relationship between SE pixels and gate setup, so that register-programming of an arbitrary 25x25 SE is possible. But reprogramming the device for other SE sizes and other image processing functions is difficult for an EPLD, and either difficult or impossible for a custom chip, depending on the chip design. Furthermore, the size of a SKIPSM implementation grows approximately linearly with SE size, but the number of gates and associated functions needed for a direct implementation grows roughly as the square of the SE size. Thus, for large SEs the SKIPSM architecture will often require significantly less hardware than a direct implementation. Finally, the direct implementation is just another extension of the conventional “brute force” approach of providing dedicated and relatively inflexible modules for each function, even if most of the modules in a system are unused during any one image pass. In contrast, SKIPSM offers an architecture that can be reprogrammed to perform many different functions. A system with conventional special-purpose modules to perform the frequently-used functions, plus a few general-purpose SKIPSM modules to provide powerful morphology, template matching, and other operations that don’t justify special-purpose modules, could offer cost-effective performance with hardware-based speeds but with flexibility comparable to the best software-based systems.

9. SUMMARY AND CONCLUSIONS

Morphological image processing techniques offer many advantages over “conventional” image processing for many important inspection problems. Morphology has not been used as widely as its power suggests that it should be because the available software implementations are usually too slow and the available fast hardware implementations are too expensive. This paper has shown ways to use finite state machines to implement binary morphology very efficiently either in existing image processing boards, in EPLDs, or in a standardized SKIPSM architecture (which can also perform many other image processing operations). Large and very large structuring elements, needed for most realistic applications but prohibitively expensive with prior technology, become very practical using this approach.

10. REFERENCES


