STRUCTURING ELEMENT DECOMPOSITION BY TREE SEARCHING

M. Razaz and D.M.P. Hagyard

School of Information Systems,
University of East Anglia,
Norwich, England
Email: mr@sys.uea.ac.uk

ABSTRACT

Morphological image processing is a technique that is becoming increasingly important for a wide range of image processing tasks. The two primitive operations, dilation and erosion, expand or contract objects of an image in a manner described by the structuring element, commonly a binary image. The shape of the structuring element allows fine control over the shapes processed by the operation. The time taken for morphological operations to complete is proportional to the number of pixels in the structuring element. By breaking the structuring element down into pieces that are applied sequentially to an image the computation time for the morphological operations can be reduced. This paper examines our implementation of the Zhuang and Haralick tree search decomposition algorithm and presents results of timing experiments that show the time taken for decomposition rises exponentially with the number of pixels in the structuring element.

1. INTRODUCTION

The morphological operations as described by Serra [1] are based on hit and miss operations on binary images. The result of a morphological dilation is the set of locations where the structuring element set could be placed such that it would intersect with objects in the image set. The result of a morphological erosion is the set of locations where the structuring element could be placed such that it would not intersect with any pixels not in the input image set. In algorithmic terms a dilation is carried out by scanning an image and, for every pixel in that image, examining the neighbourhood of that pixel as defined by the structuring element, to see if it contains a pixel belonging to an object in the image. This has the result of making objects in the image swell, since pixels near objects may have the edge of objects in their neighbourhood and thus be included in the output set. Erosion makes objects shrink - since pixels within an object that are close enough to the edges of the object to have a background pixel in their neighbourhood, as defined by the structuring element for the operation, will not be included in the output set. The size of the neighbourhood to be examined equals the size of the structuring element used in the operation, therefore the time to perform an operation is directly proportional to the size of the structuring element set.

There are a number of approaches to speeding up the binary morphological techniques. One route is to improve the operation of the low level algorithm. This can be done by changing the data structures to improve the operation of the algorithm, or by making the algorithm examine and manipulate the data in a more efficient manner.

The second approach for speeding up binary morphological operations is to perform structuring element decomposition (SED). It is possible to decompose a structuring element into a series of shapes that can be morphologically added together to reproduce the original shape. The structuring elements that make up the decomposition may be applied to an image successively and can reduce the time taken for the overall operation. It is also possible to decompose a shape such that its component parts can each be applied separately to the image and then combined set theoretically to produce the same result as the original structuring element. This allows a complex structuring element to be broken down into a set of simpler shapes that may have optimised implementations.

Various SED methods exist in the literature, see for example [2-8]. Which decomposition algorithm is the most suitable depends on the hardware platform being used for its implementation. Specialised hardware platforms such as Cytocomputer and Massively Parallel Machines, have been designed for fast implementation of morphological operations. The limitations on the size, shape and connectivity of structuring elements that

these machines can efficiently handle have greatly influenced the development of different decomposition algorithms. Zhuang and Haralick [8] have presented an elegant SED algorithm using tree searching. Despite its mathematical elegance there is no detailed implementation and assessment of this algorithm in the literature. The Tree Search Decomposition Algorithm (TSDA) is an implementation and extension of this algorithm.

2. TREE SEARCH DECOMPOSITION ALGORITHM

The search based method as described by Zhuang and Haralick [8] decomposes a structuring element down into a series of translate and add operations. Each of these operations is equivalent to a sparse structuring element containing two elements, one at the origin, the other at a position relative to the origin equal to the vector of the translation. The two member structuring elements can be applied in any order since morphological dilation is commutative.

The question the algorithm attempts to answer is this: Given a structuring element S, determine the smallest N and corresponding components $H_1, H_2,...,H_N$ such that

$$S = H_1 \oplus H_2 \oplus ... \oplus H_N \qquad (1)$$

If the structuring elements contain only two members, then they are referred to as 2-point sets. If all of the H_n are 2-point sets then the resulting decomposition is said to be a 2-point decomposition. A canonical 2-point decomposition is one where all of the H_n's contain the origin. These 2-point sets can be performed with only a shift and AND operation for an erosion, or a shift and OR operation for dilation. This leads to very efficient hardware implementations, which is what the decomposition was designed for.

The decomposition of S is found by a combinatorial search process that constructs a tree of possible shift and add operations from a start node. The search is recursive. At each m-level node there is a partial decomposition $H_1 \oplus H_2 \oplus ... \oplus H_m$. The algorithm creates child nodes which have one more translation, $H_1 \oplus H_2 \oplus ... \oplus H_m \oplus H_{m+1}$ such that the decomposition result still remains within the confines of the structuring element to be decomposed. Any node for which there is no translation t such that $(H_1 \oplus H_2 \oplus ... \oplus H_m)_t \subseteq S$ can have no children and therefore dies. If, at any level, all of the nodes die, then the search has failed and S has no decomposition. If at a node, $S \ominus (H_1 \oplus H_2 \oplus ... \oplus H_m)$ = a single pixel, then this m-level node is a leaf node. The path from this m-level node to the root node is one possible decomposition of S. The optimal decomposition of S is the shortest path from the root node to a living leaf node. The optimal solution is easily found by a breadth-first search; it will be the first living leaf node located.

The root node will contain no partial decomposition, but will have a large number of children. The children of the root node are the vectors which can be between each possible pairing of the members of S. This large number is reduced by the fact that repeated vectors are ignored, however the search method is very nearly an exhaustive search. Methods to speed up the algorithm involve: i) reducing the degrees of freedom allowed in generating the children for each node and ii) forward checking the children to eliminate.

To describe the operation of the tree search it is useful to detail the operation of the algorithm at an arbitrary location part of the way down the tree. When a node at level *m* is born it is given a name J_m, which is the *m*th structuring element, or translation, in the decomposition of S. Apart from a name it is also given a heritage from its parent consisting of the following:

i) a restricted sequence L_m which contains all of node J_m's future generation descendent name possibilities;

ii) the partial decomposition $K_m = J_1 \oplus ... \oplus J_m$;

iii) the undecomposed part of S, $T_m = S \ominus K_m$.

The production of children is accomplished by scanning through the list L_m and, using the forward checking, eliminating all those members that cannot be part of a decomposition. The result of this elimination is a list L^*_m, containing a list of J's that are valid members of a possible decomposition. For each member of the list L^*_m a child node is generated, and is given the heritage as defined above. L_m is constructed from the list of successful possible children from the parent node, L^*_{m-1}. The elements of L^*_{m-1} are $\{ J_{(m-1)1}, J_{(m-1)2}, ..., J_{(m-1)n}\}$, therefore the parent node, J_{m-1}, has *n* children. The heritage given to each of the child nodes consists of one of the names taken from the list L^*_{m-1} and L_m, which consists of all of the elements of L^*_{m-1} that are greater than or equal to L_m. For example the *k*th child of J_{m-1} will have a list of possible children, L_m, which consists of $\{ J_{(m-1)k}, J_{(m-1)k+1}, ..., J_{(m-1)n}\}$.

The method of forward checking is to see if S is closed with respect to the current decomposition when the

prospective child (J, selected from L_m) has been added to it, i.e.:

$$S = S \circ (K_m \oplus J) \quad (2)$$

where o is the opening operation. In terms of the heritage (2) can be written as

$$S = (T_m \ominus J) \oplus (J \oplus K_m). \quad (3)$$

It is more efficient to test (3) in several parts. It follows directly [8] that if $S = (T_m \ominus J) \oplus (J \oplus K_m)$ holds then:

$$\#S \leq \#(T_m \ominus J) \,.\, \#(J \oplus K_m) \quad (4)$$

where #(S) denotes the number of members in set S. In the first step of the forward checking, $T_m \ominus J$ and $J \oplus K_m$ are computed and the inequality checked. If the inequality is not satisfied then this J will not be put into the list L^*_m and the next member of L_m is tested. If the inequality is valid then the following relation is checked:

$$T_m = (T_m \ominus J) \oplus J \quad (5)$$

If relation (5) is true then J has passed the checking and is put into L^*_m because:

$$(T_m \ominus J) \oplus J \oplus K_m = T_m \oplus K_m = S. \quad (6)$$

If (5) is not true then the element J has a second chance. The dilation $(T_m \ominus J) \oplus (J \oplus K_m)$ is performed and compared for equality with S. If this is successful then J is put into L^*_m, otherwise J is discarded. Once the list L^*_m has been generated, the work on this node is complete and the new children should be spawned.

The partial decomposition for the child, K_{m+1}, is generated by dilating the partial decomposition of the current node, K_m, by the J represented by the child being generated. Similarly T_{m+1}, the undecomposed part of S for the current node, is generated by eroding the current value T_m by the J which is represented by the child currently being generated. It need not be the case that these values are generated after the forward checking stage. They may be generated during the testing and stored until needed.

A node at the end of a decomposition, an end-node, occurring on level N is identified when $\#T_N = 1$. At this point the single member of T_N is the q from the decomposition:

$$S = \{q\} \oplus J_1 \oplus \ldots \oplus J_N. \quad (7)$$

An end-node will produce no children. Nodes that produce no children, but that have not reduced the undecomposed part of S down to one element, are dead branches.

To find an optimal decomposition, a breadth-first tree search is carried out that scans through the levels of the tree from the root to the leaves. The first end-node found is the optimal decomposition because it will be the one with the shortest path back to the root node. Any other end-nodes found on the same level will also be optimal. A second form of the algorithm was implemented where the tree was built up in a breadth-first manner. The construction of the tree was halted on identifying the first end-node. Since the tree is constructed in a breadth-first manner the first end-node found is the most optimal decomposition. The second implementation allowed the algorithm to finish without having to construct the entire tree.

3. RESULTS

Testing for correctness was carried out by using the same input bitmap as in the Zhuang and Haralick paper example [8], printing out the contents of the entire tree and checking that it corresponded to the tree given in the paper. A second test was to examine all the possible outputs from a run, and manually check that they produced valid decompositions. Once these tests were performed the tree printing functions of the software were disabled and memory use optimisations added to the code.

The decomposition algorithms were tested using a series of square structuring elements with sizes ranging from 2 x 2 pixels to 15 x 15 pixels. The timing results are shown in Figure 1. The shape of the graph shows that as the size of the structuring element gets larger, the time taken for the algorithm to find the optimal solution increases at an exponential rate. The reason for the rapid increase in time to decompose structuring elements can be seen when the number of nodes in the generated tree search is examined, see Figure 2.

The maximum size of a structuring element that can be decomposed by the algorithm is 13 x 13 pixels. If the tree is built in a breadth first manner and construction halted when the first solution is found, then a slightly larger image, 14 x 14 pixels, can be handled. At sizes larger than these the program will run out of memory on the current system. It should be noted that the important factor in the size of the search tree is the number of possible translations between different pixels

on the bitmap. This increases as the number of pixels grows. Therefore, the dimensions of the bitmap are of secondary importance to the number of pixels in the image.

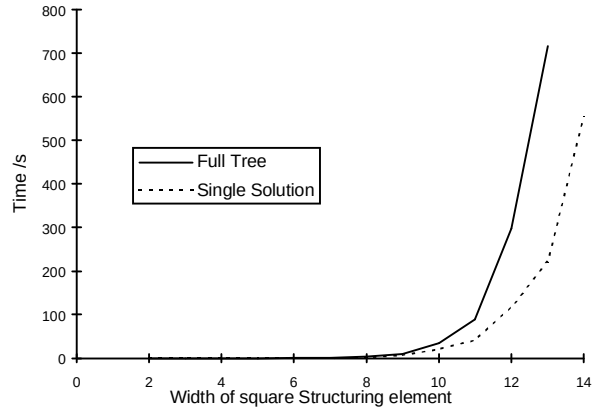

Figure 1. Time to Decompose a Square structuring element

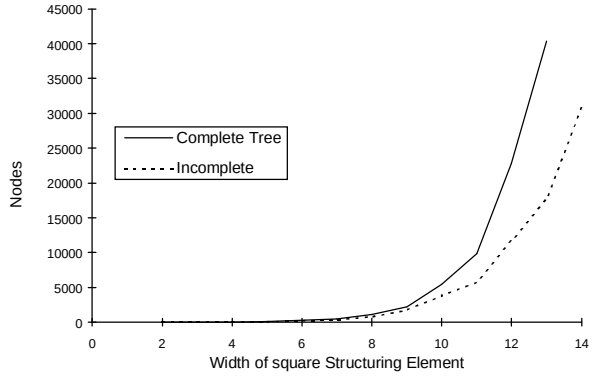

Figure 2. Nodes in tree against Height and Width of Structuring Element

The graphs in Figures 1 and 2 were entered into a curve fitting program. All of them closely fitted a logarithmic graph of the form:

$$y = A + e^{\left(\frac{((\ln x)-B)^2}{C}\right)}$$

where y is either time or the number of nodes in the tree, x is the height or width of the structuring elements and A, B and C are fitting parameters. The curve fitting program estimated that, on a DECstation 5000/200 running Ultrix, with a speed spec mark of 20, a 20 x 20 pixel structuring element would produce a tree with a maximum of 1,959,636 nodes and would require approximately 82 hours to complete.

The initial tests of the decomposition algorithm were performed on simple shapes that were known to be decomposable. These were squares and other lozenge shapes. On examining the output of the algorithm it was possible to see that the output translations of the structuring element were closely connected to the edges of the input structuring element. The translations in the decomposition were either horizontal, vertical or in one of the two diagonal directions. If all the translations with the same direction were applied to a single pixel they produced a straight line that was the same length as the edge in that direction on the structuring element. For example, an octagonal structuring element contained within a 15 x 15 pixel square will produce the following decomposition:

{{3,3}, {-3,3}, {0,2}, {1,1}, {1,1}, {0,1}, {0,1}, {-1,1}, {-1,1}, {2,0}, {1,0}, {1,0}}.

This can be broken down into:

{{3,3}, {1,1}, {1,1}}, {{-3,3}, {-1,1}, {-1,1}}, {{0,2}, {0,1}, {0,1}}, {{2,0}, {1,0}, {1,0}}.

The first batch of three translations will produce the 6 pixel long down diagonal, the next batch of three translations will produce the 6 pixel long up diagonal, and next two batches of 6 will produce the vertical and then horizontal 5 pixel edges. The above decomposition took 493 seconds to generate.

The lengths of the translations in each direction comprise the most efficient method of creating a line of correct length. There is a pattern to the magnitudes that can be seen if a large enough length of pixels can be decomposed. If a straight line of 20 pixels is decomposed, the output is as follows

{{10,0}, {5,0}, {2,0}, {1,0}, {1,0}}.

The order of the magnitudes can be calculated by dividing the length by 2 and rounding down. This value is the length of the first translation. Subtracting the first translation length from the initial length gives the length to be divided by 2 in the next iteration of the algorithm. This division and subtraction continues until the current length can no longer be divided by 2. Using this algorithm, and a routine to create the partial Freeman chain of a shape, a program was written to create a series of shift and add translations that would be the same as a tree search decomposition of the input structuring element. This program was only able to decompose lozenge shape structuring elements, but operated much faster. Since its complexity was proportional to log to the base 2 of the length of the

edges it was possible to create the decomposition of a 255 x 255 square structuring element. This would have been impossible using the TSDA. These pseudo-decompositions were used to perform the timing analysis of the morphological primitives using the results of the decompositions.

Attempts were made to decompose more complicated shapes but they were usually unsuccessful. A variety of shapes were tested. Triangles, trapeziums, kites, and most patterns of distinct repetitions failed. Some circle shapes were successful, although testing of this was limited to shapes that were small enough to be processed. All the lozenge shapes that would work with our Fast Morphology Transform (FMT) algorithm [9] were successful. The lozenge shapes that would not work with the FMT algorithm, i.e. shapes constructed from diagonals with no horizontal or vertical components, failed with the tree search decomposition method.

In general, the tree search method will be able to decompose any convex shape that has one degree of rotational symmetry, i.e. any shape that can be rotated 180° and exactly map onto itself. If the structuring element comprises a number of separate objects then a decomposition is only possible if the element can be constructed from a shape following the above criterion that is cumulatively shifted and added to build up the pattern.

4. CONCLUSIONS

The Tree Search Decomposition Algorithm is mathematically very elegant, but it remains an exhaustive search technique. TSDA can only decompose shapes that can be generated by a series of shift and add operations, such as convex, symmetric shapes. These shapes can have edge directions that do not follow the four 8-connected directions i.e. horizontal, vertical, diagonally up and diagonally down. The algorithm can also decompose structuring elements consisting of groups of convex, symmetric shapes. All the shapes would have to be identical, and in general there would have to be 2^x of them where x is a positive integer. The shapes could partially or fully overlap, possibly making odd numbers of shapes, or concave shapes. Although we have shown that TSDA is very slow it could be made more efficient if the method decomposed structuring elements into larger shapes than the two-member shift and add operations currently used in our implementation. Possibly using an exhaustive tree search method that looked for simple shapes could be useful. Moreover it could be effective to use a tree search method to generate a tree truncated to a manageable size. The best decomposition from the truncated tree could then be used to reduce the size of the structuring element for a second pass with the algorithm. This could be an effective method to work around the memory problems associated with TSDA.

REFERENCES

[1] Serra, J. “Introduction to Mathematical Morphology”, Computer Vision, Graphics and Image Processing, Vol 35, pp 283-305, 1986.

[2] Xu, J., “Decomposition of Convex Polygonal Morphological Structuring Elements into Neighbourhood Subsets”, IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol 13, No 2, pp 153-162, February 1991.

[3] Pitas, I., Venetsanopoulos, A.N., “Morphological Shape Decomposition”, IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol 12, No 1, pp 38-45, January 1990.

[4] Park, H., Chin, R.T., “Decomposition of Arbitrarily Shaped Morphological Structuring Elements”, IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 17, No 1, pp 2-15, January 1995.

[5] Park, H., Chin, R.T., “Optimal Decomposition of Convex Morphological Structuring Elements for 4-Connected Parallel Array Processors”, IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 16, No 3, pp 304-313, March 1994.

[6] van den Boomgaard, R., Wester, D., “Logarithmic Shape Decomposition”, Internal Report for Department of Mathematics and Computer Science, University of Amsterdam, The Netherlands, 1993.

[7] Pitas, I., Venetsanopoulos, A.N., “Morphological Shape Decomposition”, IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol 12, No 1, pp 38-45, January 1990.

[8] Zhuang, X., Haralick, R.M. “Morphological Structuring Element Decomposition”, Computer Vision, Graphics and Image Processing, 35, pp 370-382, 1986.

[9] Hagyard, D.M.P., Razaz, M., Atkin, P., “A Fast Algorithm for Computing Morphological Image Processing Primitives”, Proc. IEEE Workshop on Nonlinear Signal and Image Processing, 1997.