TOPOLOGICAL MATCHING

Quentin F. Stout
Mathematical Sciences
State University of New York
Binghamton, NY 13901 USA

1. INTRODUCTION

There is a lot of practical and theoretical interest in designing algorithms to process digital pictures. Of particular interest are problems arising when one starts with an nxn array of pixels and stores it, one pixel per processor, in some sort of array-like parallel computer. One of the earliest systematic examinations of such problems was Beyer's thesis [1], in which he gave several algorithms for a computer we call a mesh automaton (defined below). One of the problems he considered was topological matching, in which one is given two pictures and is (roughly) asked if it is possible to stretch one picture so that it looks like the other. (A precise definition is given below.) Beyer gave several solutions, one of which required $\Theta(n^{**}4)$ time, and Dietz and Kosaraju [2] later gave a $\Theta(n^{**}2)$ solution. In this paper we give an optimal $\Theta(n)$ time solution, based on a simpler $\Theta(n)$ time solution for a more powerful computer called a mesh computer. Beyer suggested that this problem was a prime candidate for a non-linear recognition problem, but our result shows that this is not true.

2. DEFINITIONS

Our digitized pictures are given in the form of an nxn array of _pixels_, where each pixel is either black or white. Pixels are located at positions (i,j), with $1 \le i,j \le n$, and the entire array is called a _figure_.

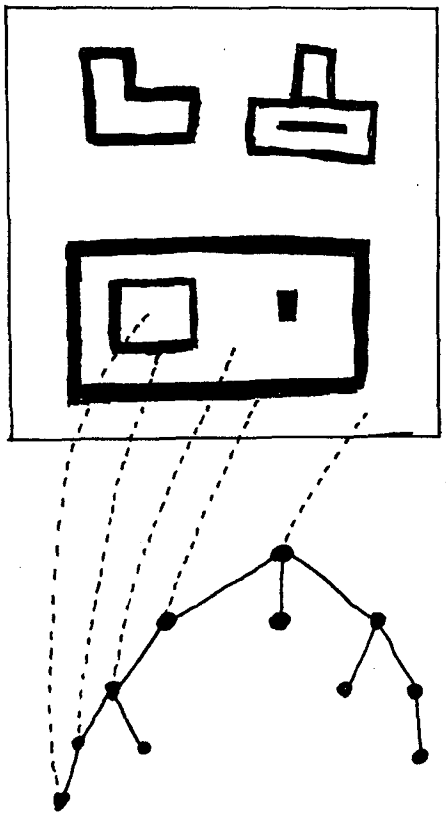
We need to define the notion of a connected component, which is made slightly confusing by the digitization. In order to have such standard results as the Jordan Curve Theorem, we need to use slightly different definitions for black and white. (See Rosenfeld [10,11].) Two black (white) pixels at (a,b) and (c,d) are _adjacent_ if and only if

$$1 = |a-c| + |b-d|$$
$$(1 = \max\{|a-c|, |b-d|\}),$$

and are _connected_ if and only if there is a path of adjacent black (white) pixels from one to the other. Given any pixel, the set of all pixels connected to it is called a _component_. To simplify discussion, from now on we will assume that the pixels on the edge of the figure are all white, and their component is called the _background_.

A component C' is _contained in_ a component C if C'≠C and any path of adjacent pixels (using either definition of adjacency, with the path being allowed to contain both colors) from C' to the edge must contain a pixel in C. C' is a _son_ of C, and C is the _father_ of C', if C' is contained in C and for any other component D, if C' is contained in D then so is C. Notice that sons of white components are black, and vice versa. The son relation forms a rooted tree whose root is the background. Figure 1 shows a sample figure and its component tree.

Beyer [1] defined the _topological matching predicate_ on pairs of figures F and G to be true if and only if F's component tree is isomorphic (as a rooted tree) to G's. This predicate captures the correct digital version of homotopy of two-dimensional figures, in that two figures are equivalent only if one can be changed into the other by stretching without ripping. It is therefore a very basic predicate, although it has yet to receive much attention.



A Figure and Its
Component Tree

Figure 1.

Our machine models are based on arrays of processors. A mesh computer of size n**2 consists of n**2 copies of a processor P, with these copies located at positions (i,j), where $1 \leq i,j \leq n$. Processors (i,j) and (k,l) have a unit-time communication link if and only if $1 = |i-k| + |j-l|$. All operations take unit time, and P is assumed to have only a fixed number of registers, independent of n, each of which holds one word. If we assume that the wordsize is fixed to be independent of n then our model is equivalent to assuming that P is a finite state automaton, and the resulting array machine is called a mesh automaton. Mesh automata were among the first types of parallel machines to be investigated [1,3,5,11], while recently there has been greater interest in a more powerful machine. In this more powerful model, which we call simply a mesh computer, the wordsize of P is Θ(log(n)). In a mesh computer each processor can store its coordinates, which is impossible with a mesh automaton. Mesh computers have appeared in [6,7,8,13,14] and many other places.

Beyer worked on the problem of computing the topological matching predicate on a mesh automaton, where a figure is stored so that pixel (i,j) is in processor (i,j). He gave several solutions, one of which computed a binary string representation of the figure's component tree. The representation was chosen so that the string uniquely identifies the isomorphism class of the tree. It is easy to see that strings can be compared in Θ(n) time, so the problem reduces to the problem of rapidly computing a string representation. Beyer's string generation procedure required Θ(n**4) time, and Dietz and Kosaraju [2] found an algorithm requiring Θ(n**2) time. We will reduce the mesh automaton time to Θ(n), which is the best possible. We first give a Θ(n) algorithm for a mesh computer, and then convert this to one for a mesh automaton. As far as we can determine, no one had previously considered performing topological matching on a mesh computer.

3. THE MESH COMPUTER ALGORITHM

We fill follow Beyer's lead and compute the topological matching predicate by transforming each figure into its tree and then applying a map e from rooted trees to binary strings, where e has the property that e(T)=e(T') if and only if T and T' are isomorphic. Let T be a rooted tree, and let |T| denote the size of T (i.e., the number of nodes in T). e(T) will be such that

length(e(T)) = 2*|T| ,

and it is defined as follows:

```
If |T|=1 then e(T)=01
else let T1, ..., Tk be the subtrees
     whose roots are the sons of the
     root of T. Sort e(T1), ...,
     e(Tk) by length, longest strings
     first, and among strings of the
     same length, sort numerically.
     Let S denote the concatenation
     of the sorted lists. Then
     e(T)=0S1.
```

For example, if T is the tree in Figure 1, then

e(T) = 00000110111000110110011 .

It is easy to see that e(T)=e(T') if and only if T and T' are isomorphic, and length(e(T)) < n**2 for any tree T arising from an nxn figure (n>1).

The algorithm has two parts: initialization and string formation. During initialization, for each component a record is created which represents the component and which moves about during string formation. This record contains the component's label, which is the smallest row-major index of any pixel in the component. (The row-major index of pixel (i,j) is (i-1)*n+j.) It also contains the component's depth in the component tree, the size of the subtree that it is the root of, its parent's label, and the size of the largest of its sons' subtrees. In the Initialization section we show that this can be constructed in $\Theta(n)$ time.

String formation is somewhat more complicated. We recursively construct the string, storing 1 bit per processor. We initially "assign" processors 1 through 2*(size of the component tree) to the entire tree. In general, given a tree T with root p, which has been assigned processors A through B, we first put a 0 in A and a 1 in B. There are three situations which can occur:

1. $|T| = 1$, in which case we are finished.
2. The largest son of p has a subtree of size $\le 0.5*|T|$, in which case processors A+1 through B-1 are divided into blocks among the subtrees whose roots are sons of p, each subtree receiving twice as many processors as the size of the subtree. These blocks are assigned so that larger trees come first, with ties broken arbitrarily. Each node of T, except for p, determines which block to move to and moves there. Then the strings in each block are determined, and when finished strings in blocks of the same size are sorted numerically.
3. The largest son of p has a subtree of size $>0.5*|T|$, in which case there is a unique node q, with largest son r, such that
$$|\text{Tree}(q)| > 0.5*|T| \quad \text{and}$$
$$|\text{Tree}(r)| \le 0.5*|T|,$$
where Tree(q) is the subtree with root q. Nodes on the path from p to q are called "spine" nodes, and each spine node determines where its block is. (Except for p, each spine node's block is within another's.) Each spine node puts a 0 at the front of its block, a 1 at the end, and helps its sons determine their subblocks. Each node moves to an appropriate block, in which the strings are generated. Then strings of the same length corresponding to sons of the same spine node are sorted numerically, completing the processing for T. Notice that even though a spine node s is a son of a spine node t, s's string is not compared that of any other son of t since all other sons have shorter strings.

In the String Generation section we show that given a tree of N nodes, all of the processing in cases 2 or 3, except for the generation of substrings, can be accomplished in $\Theta(N**0.5)$ time. The role of the spine nodes is to guarantee that each subblock, which is where the recursive string formation occurs, is no larger than one-half of the original. If S(N) denotes the worst-case time to generate the string for a tree of N nodes, given that initialization has been done, then S will satisfy:
$$S(1) = C$$
$$S(N) = D*N**0.5 + S(N/2)$$
which gives $S(N)=\Theta(N**0.5)$. Since $N<n**2$, we have

Theorem 1 Using a mesh computer of size n**2, our algorithm decides topological matching in $\Theta(n)$ time.

Both the initialization and string formation algorithms use simulated random access reads and writes. In a random access read there are several processors, each of which needs to fetch a word of data in some source processor. There may be several different source processors, and for any source there may be several processors trying to read from it. Each processor knows the coordinates of the source processor it is trying to read from. In a random access write there are processors which are trying to write a word of data to some target processor, where there may be multiple targets and multiple processors trying to write to the same target. Writing introduces an addition complication in that one must specify how conflicts are to be resolved, since two or more processors may try to write different values into the same target. Sometimes we want the maximum value being sent, and sometimes we want the sum. By utilizing sorting, random access reads and writes can be performed in $\Theta(n)$ time on a mesh computer of size n**2, assuming that the conflict resolution for the writes is reasonable [8]. (Reasonable resolutions include any of the ones used here.)

3.1 INITIALIZATION

We need to show how to create the record used to represent a component in the string formation phase. First we label each component, as described in Nassimi and Sahni [7]. (Because of the different definitions of connectedness, we must use slightly different procedures for white and black components.) The label of a component is the smallest row-major number of any pixel in it, and at the end each processor knows the label of its component. For each component, the pixel whose row-major index equals that of its component is called the component's representative and is responsible for creating the component's record.

First each representative finds the label in the processor to its left, which is the label of the component's parent.

Now each processor does a random access read, reading from its component's representative the label of the component's parent. Then each processor creates a record containing its label, that of its component's parent, and a counter which is initially 0. In each row these records are rotated from left to right, with only the representatives really using them. Each representative also keeps a depth counter, which is initially 0. The first time the representative receives a record which starts with its parent's label, it adds 1 to its depth counter, adds 1 to the record's counter, remembers its grandparent's label, and then passes the record on. The first time it receives a record starting with its grandparent's label it adds 1 to its depth counter, adds 1 to the record's counter, remembers its greatgrandparent's label, passes the record on, and so on. This continues until each processor receives back the record it started, at which time each representative's depth counter has the correct value.

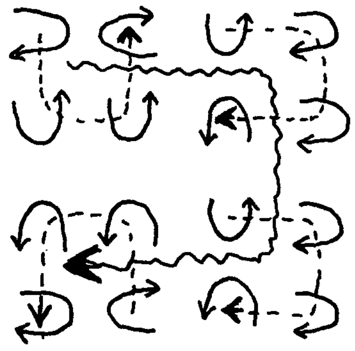
Each processor now does a random access write to its component's representative, writing the counter in the record circulated above, with these values being summed by the write operation. When finished each representative knows the size of its subtree, and with a few more random access reads and writes each representative can complete the record it is creating for its component. The total time for this part is $\Theta(n)$.

3.2 STRING FORMATION

We need to use an ordering which combines some of the best features of snake-like ordering and shuffled row-major ordering [7,14]. Figure 2 illustrates this recursively constructed ordering, where we assume n is a power of 2. While we have not seen this ordering used elsewhere, we suspect that perhaps it has been since it is fairly natural. It has the property that there is a constant C<4 such that processors numbered i and j are no more than C*|i-j|**0.5 communication links apart, and for this reason we call it a *proximity ordering*. Further, there is a constant D such that any block of processors i..j contains a square of edgelength D*(j-i)**0.5 . This enables us to treat any block as if it were a square since we can always move all the required data to this subsquare, in O((j-i)**0.5) time, putting only 1/D items per processor. An important point is that we need not iterate this, that is, we never encounter a situation where we must compress data into a square and then while processing it we create a subblock which in turn must compress the

1	2	15	16	17	20	21	22
4	3	14	13	18	19	24	23
5	8	9	12	31	30	25	26
6	7	10	11	32	29	28	27
59	58	55	54	33	36	37	38
60	57	56	53	34	35	40	39
61	62	51	52	47	46	41	42
64	63	50	49	48	45	44	43

Processor Numbering



The Recursive Pattern

Figure 2. Proximity Ordering

data. Any time data is compressed we then uncompress it before performing any operations on subblocks. We use this proximity ordering throughout string formation, and also omit any further explicit discussion of when to compress.

We need to show that if a tree T, with root p, has been assigned processors A through B (where B-A+1 = 2*|T|), and all nodes of T are in this block, then for either case 2 or 3, in Θ(|T|**0.5) time the subblocks can be determined and each node can move to the appropriate subblock. The root p knows which case holds, so by a random access read each node will know.

First suppose case 2 holds. In A..B we sort the nodes so that p is first, followed by its sons, followed by all others. The sons of p are sorted in decreasing order of the size of their subtree. Since A..B is approximately a square, we sweep accross each row to find the sum of the sizes of p's sons' subtrees. We then go down the first column, assigning space to each row, and then back accross each row assigning subblocks to p's sons. By using path compression, as in Nassimi and Sahni [7], each node determines which son of p it is beneath. Now each node does a random access read to read from this son the subblock to move to.

Case 3 is quite similar, except that first each node needs to determine if it is a spine node. It does this by reading p's subtree's size and comparing it to its own. If q is a spine node and q's depth is k larger than p's, then Tree(q)'s block goes from A+k to A+k-1+2*| Tree(q) | . If q's largest son is not a spine node (recall that q knows the size of its largest son) then all nodes in its subtree move to q's block, while if q has a spine node for a son then q computes the region where all nodes under q, but not under its spine son, should move to. If S is the size of q's spine son's tree then the region goes from A+k+1+S to A+k-1+2*| Tree(q) | . Once each spine node has computed this information, each node determines (via path compression) its nearest ancestor which is a spine node, and from this determines what region to move to. Once there, non-spine node sons of a spine node calculate their subblocks, and then all nodes move to the proper subblock.

Whichever case holds, the total time is at most θ(| T |**0.5) . After the strings for the subblocks have been formed there may be a final sort phase, which also takes at most θ(| T |**0.5) time. This finishes the proof that string formation takes no more than θ(n) time on a mesh computer of size n**2, which in turn finishes the proof of Theorem 1.

4. THE MESH AUTOMATON ALGORITHM

To convert the previous algorithm into one for a mesh automaton we will use clerks to simulate the processors of the mesh computer. Clerks are just a systematic form of counting, and counter-based solutions have been given for many mesh automaton problems [3,11]. Clerks are described in [12,13], and use θ(log(n)) processors to simulate one processor of a mesh computer, with unit-time operations being simulated in θ(log(n)) time. At most θ(n**2/log(n)) processors can be simulated so we must reduce the number necessary.

We do this by dividing the nxn array into squares of edgelength K, where K=θ(log(n)**2). There are 4*K-4 processors on the edge of each square, and in each square we create an equal number of clerks, as in Figure 3. In each square we set up a 1-1 correspondence between the edge pixels and the clerks, and from

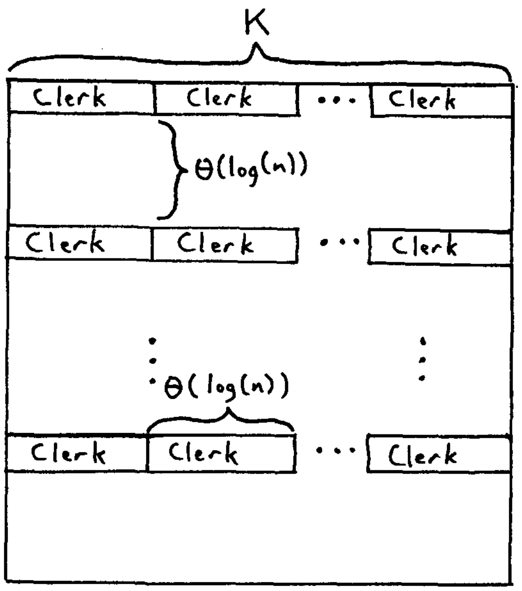


Figure 3. Clerks in a Square

now on when we speak of edge pixels doing some calculation we mean their associated clerk. The clerks form a θ(n/log(n)) x θ(n/log(n)) array, so any major operation, such as sorting or random access reads, which takes θ(n) time on a mesh computer of size n**2 will take θ(n/log(n)) steps on the clerks. Since each step takes θ(log(n)) time, the total time remains θ(n).

First a procedure similar to that in [13] is used to label each component which includes an edge of some square, where the label is the minimum row-major index of any edge pixel in the component, and where only edge pixels know of the label. We call any such component a <u>labeled component</u>, and all others are <u>unlabeled components</u>. Note that unlabeled components lie entirely within a KxK square, and hence are the root of a subtree of size less than K**2. Any component which is the root of a tree with fewer than K**2 nodes is called <u>small</u>, and all others are <u>large</u>. All large components are labeled, but small components may be either labeled or unlabeled. Processing of large components will closely follow the mesh computer algorithm, but a different procedure is needed for small ones.

We divide the mesh automaton algorithm into three parts: initialization, string formation for small components, and string formation for large components.

4.1 INITIALIZATION

Once the clerks have been formed and the labeled components determined (taking $\Theta(n)$ time), we need to find essentially the same information as was found in the initialization section for mesh computers. When we determine the size of a node's subtree we will seperately count the number of labeled and unlabeled descendants, and we must be a bit more careful when determining depth. Within a square there may be several edge pixels in the same labeled component. Once labeling is completed we need only one of these per square, so we use the one of minimal row-major index, and from now on the rest are ignored.

In each square each edge pixel first counts its unlabeled descendants within the square, not counting descendants of labeled offspring. Any simple procedure can be used since the squares are so small. Each pixel then writes its count to its component's representative, with these values being summed. In $\Theta(n)$ time each component knows the number of unlabeled components in its subtree, not counting ones beneath labeled offspring.

Each component representative determines its component's parent, and by a random access read each edge pixel reads this. To determine the component's depth and size we circulate information as before, but now entire squares are moved. Notice that if a component's representative tried to add to the counter of each of its ancestors then it may have to do this $\Theta(n)$ times, resulting in $\Theta(n*\log(n))$ total time. To avoid this, first each edge pixel forms a record containing its label, the label of the closest ancestor which does not intersect its square, the difference in depth between it and this ancestor, and a counter which is initially 0. (It finds the closest ancestor outside the square by finding the the greatest ancestor touching the square and using its parent.) Now this information is rotated, squares moving together. As before, as each square arrives each representative is looking for a record corresponding to a specific ancestor. If that ancestor is present then the representative adds its count of unlabeled descendants, plus 1, to the ancestor's counter. It then takes note of the next ancestor to search for (in later squares) and adds the depth information to its own depth counter. This takes $\Theta(K)$ time per square, for a total time of $\Theta(n)$ before each square's records return to it.

Now each edge pixel adds to its counter the counts of all edge pixels lying in the square which are in descendant components. All edge pixels do a random access write to their representative, writing their counter, with these values being summed. At this point all representatives of labeled components know their depth and the size of their subtree, and the rest of initialization is as before.

4.2 SMALL COMPONENT STRINGS

For small string formation we think of the region below one clerk and above another as being a "bag" attached to the top clerk. A bag has $\Theta(\log(n)**2)$ processors and is used to store string representations of small components, storing one bit per processor. Bags are less passive than their name implies for they occasionally help their clerk perform operations.

In each square a <u>package</u> is prepared by each edge pixel which is in a component having unlabeled sons in the square. The package contains the string representations of all unlabeled sons in the square, and also some header information. The strings are in the bag, and the header is in the clerk. Some packages are too big for a single bag, in which case several clerks help carry them. One can show that there is enough room for all the packages.

The header contains the package's size, which component it is in, and a <u>target</u>. If the component is large then it is the target, but if it is small then the target is the component's greatest ancestor which is small. By path compression each component can determine its target. Also, each small component representative prepares a package with no strings, but which has a header with the component, its parent, and its target.

We now sort packages by their target. If the target is small then all of its packages are used to form the component's string. Any simple procedure can be used since the size and number of packages is $O(\log(n)**4)$. The string is put into a new package with the component's parent as target, and a second sort by target occurs.

The only targets remaining are large components. A large component may receive many packages, with $O(n**2)$ total size, but each string they contain is no longer than $O(\log(n)**4)$. This fact can be used to order and then concatenate all the strings in $O(n)$ time. The result is packed into bags with a header giving its length and the label of the component, and the entire assembly is called a <u>caravan</u>.

To finish the small component string formation phase, we now sort the caravans, with longer caravans first and, among equal lengths, sorting numerically. There may be groups of caravans which are equivalent, and now the clerk holding the header of a caravan does a random access write to the component, telling it the start of all equivalent caravans. This completes this section, taking $\Theta(n)$ total time. We should mention that, while all of the clerks may have been involved in the forming of the strings for the small components, they also retained all of the information about the edge pixel they represent.

4.3 LARGE STRING FORMATION

The large string formation is almost identical to that for the mesh computer, and now we are only generating a bit per clerk, instead of the bit per processor used in the packets and caravans. One difference is that we only count the number of large descendants when we assign space.

If a tree T has large sons S1, ..., Si, if the string part of its caravan is R, and if the sons are ordered so that |Sj|> S(j+1)| or else |Sj|=|S(j+1)| and e(Sj)"<"e(S(j+1)), then e(T) will be 0e(S1)..e(Si)R1. Here the ordering "<" is slightly different than before. As we are comparing strings, if they are equal up to a point and then one stops because the rest is in a caravan, while the other one still has more bits arising from large components, then the second one is judged larger, while if they both stop we compare the pointers back to the caravans to finish the comparison. This has changed our e function, but does not change the fact that it preserves tree isomorphism for trees generated from figures of the same size. One deficiency is that the same tree can have different string representations when it arises from figures of different sizes (this occurs because the definition of small depends on the figure size). While this does not alter our ablility to compute the topological matching predicate, it is not desirable. The deficiency can be corrected, in $\Theta(n)$ time, by fairly straightforward techniques which we omit.

To complete the algorithm we need to insert the caravans into the large component strings, storing everything as 1 bit/processor. This can be done in $\Theta(n)$ time, which completes our proof of the following theorem.

Theorem 2 On a mesh automaton of size n**2, our algorithm decides topological matching in $\Theta(n)$ time.

5. CONCLUSIONS

Following Beyer's lead, we have computed the topological matching predicate by transforming a figure into a binary string which identifies the isomorphism class of the figure's component tree. Since we have shown that such a string representation can be computed in linear time on either a mesh automaton or a mesh computer, we can give linear time solutions to several other topological problems. For example, it is easy to show that the string can be processed in linear time to decide if the figure is connected, or simply connected, or to determine the figure's genus. Linear time solutions were already known for these problems [3,11], but our algorithm provides a systematic, albeit complicated, approach which presumably can be used for related problems.

Since the component tree captures the notion of homotopy for two-dimensional digital figures, it is natural to consider higher dimensions. Our algorithms can be extended to higher dimensional mesh computers and automata, remaining linear in the edgelength, but unfortunately the component tree is not as useful in higher dimensions. For example, if one three-dimensional figure consists of two disjoint solid black tori in a white background, and a second figure has the two tori disjoint but linked, then the two figures will have equivalent component trees, even though they are not homotopic.

Beyer's thesis included a large number of open problems, most of which have now been solved [3,11,13]. The labeling technique used in our mesh automaton algorithm can be used to solve two more of these, namely the "representative" problem and the "gold plate" problem. (This labeling technique for mesh automata was introduced in [13], but I forgot to mention that it also solved these problems.) In the representative problem exactly one pixel in each black component is to be changed to red. To do this, just have what we called the representative in the mesh automaton solution act as the representative here. This solves the problem for all labeled components, and in each square we can use a simple $\Theta(K**2)$ algorithm to pick the representatives for the unlabeled ones. In the gold plate problem one component has a gold pixel and we are to make the rest of its component golden also. If it is an unlabeled component we use a $\Theta(K**2)$ algorithm, while if it is labeled we notify all edge pixels in that component, and they in turn propogate the gold throughout their square.

The one remaining open problem from Beyer's thesis is to determine a minimal distance solution to a maze. In this problem one is given a solvable black/white maze with designated start and stop positions and is to mark a minimal distance path between them. Beyer showed that one could decide if the maze was solvable or not in linear time, but to date no linear time algorithm has been found for marking the path. Linear time algorithms for deciding the solvability of mazes on higher dimensional mesh automata appear in [4,13], and a linear time algorithm for marking a minimal path on a (2-dimensional) mesh computer appears in [6]. Determining if there is a linear time algorithm for marking a minimal path is a particularly intrigueing question.

REFERENCES

1. W. T. Beyer, Recognition of topological invariants by iteraive arrays, Ph.D. thesis, Mathematics, MIT, 1969.

2. P. Dietz and S. R. Kosaraju, Recognition of topological equivalence of patterns by array automata, J. Comp. and Sys. Sci. 20 (1980), 111-116.

3. S. R. Kosaraju, On some open problems in the theory of cellular automata, IEEE Trans. Computers 23 (1974), 561-565.

4. S. R. Kosaraju, Fast parallel processing array algorithms for some graph problems, ACM Symp. on Theory of Computing 11 (1979), 231-236.

5. S. Levialdi, On shrinking binary picture patterns, Comm. ACM 15 (1972), 789-801.

6. R. Miller and Q. F. Stout, Mesh-computer algorithms for some topological and geometric problems, to appear.

7. D. Nassimi and S. Sahni, Finding connected components and connected ones on a mesh-connected parallel computer, SIAM J. Computing 9 (1980), 744-757.

8. D. Nassimi and S. Sahni, Data broadcasting in SIMD computers, IEEE Trans. Computers 30 (1981), 101-106.

9. A. Rosenfeld, Digital topology, Amer. Math. Monthly 86 (1979), 621-630.

10. A. Rosenfeld, Picture Languages, Academic Press, 1979.

11. A. R. Smith III, Two-dimensional formal languages and pattern recognition by cellular automata, 12th Symp. on Switching and Automata (1971), 144-152.

12. Q. F. Stout, Drawing straight lines with a pyramid cellular automaton, Info. Proc. Letters 15 (1982), 233-237.

13. Q. F. Stout, Using clerks in parallel processing, 23rd Found. of Computer Sci., 1982, 272-279.

14. C. D. Thompson and H. T. Kung, Sorting on a mesh-connected parallel computer, Comm. ACM 20 (1977), 263-271.