ISSN:0975-9646
P.Rajashekar Reddy et al, / (IJCSIT) International Journal of Computer Science and Information Technologies, Vol. 3 (3) , 2012,3888-3894

Evaluation of Stopping Criterion in Contour Tracing Algorithms

P.Rajashekar Reddy, V.Amarnadh, Mekala Bhaskar

Department of CSE, Anurag Group of Institutions,
CVSR College of Engineering, jodimetla, Ghatakesar, Hyderabad, AP, India

***Abstract*— Under the conditions of fasting moving, shape changing and partial occlusion of target, Now mostly tracing algorithms are difficult to get accurate contour of target, for these problems, a novel contour tracing algorithm based on particle filter and fast level set is proposed. Firstly, the particle filter algorithm is adopted to estimate moving target's boundary contour. Then, according to the nearest neighbour decision method, a new the velocity function of fast level set is found, the strongpoint of the function is that the tracking algorithm is fit for target and background changing. Finally, the contour tracing is realized by evolving the zero level set curves using fast level set algorithm. Experiments for representative image sequences show that this algorithm can trace the rigid and non-rigid target contour under the complex environments. The result indicates that this algorithm is robust and accurate compared with other tracing algorithms.**

Evaluation of contour tracing algorithms and how the start and Stopping Criteria's used in various contour tracing algorithms. Concentrated to show the stopping criteria's, and which criteria is the best to find contour of pixels of an image and comparison of different stopping criteria's. The resulting contours can be used to create 3D reconstructions.

Keywords: Topographic map, Contour line, Tracing, Moore neighborhood, Digital Elevation Map (DEM)

I. INTRODUCTION

What follows are four of the most common contour tracing algorithms. The first two, namely: the Square Tracing algorithm and Moore-Neighbor Tracing are easy to implement and are therefore used frequently to trace the contour of a given pattern. Unfortunately, both of these algorithms have a number of weaknesses which cause them to **fail** in tracing the contour of a large class of patterns due to their special kind of connectivity.

The following algorithms will ignore any ***"holes"*** present in the pattern. For example, if we're given a pattern like that of ***Figure 1*** below, the contour traced by the algorithms will be similar to the one shown in ***Figure 2*** (the blue pixels represent the contour). This could be acceptable in some applications but in other applications, like character recognition, we would want to trace the interior of the pattern as well in order to capture any holes which identify a certain character. (***Figure 3*** below shows the "complete" contour of the pattern) As a result, a ***"hole searching"*** algorithm should be used to first extract the holes in a given pattern and then apply a contour tracing algorithm on each hole in order to extract the complete contour.

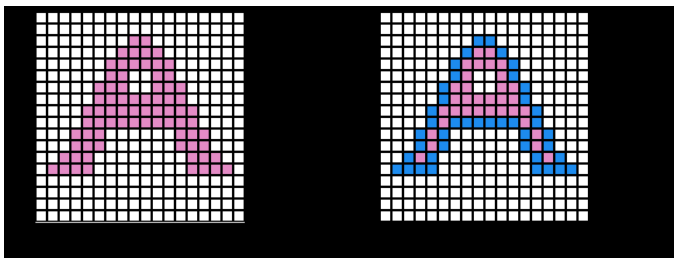


Figure 1 **Figure 2**

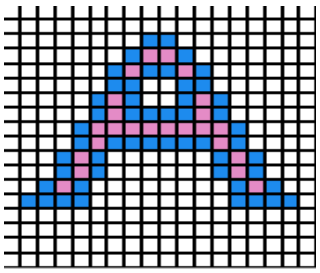


Figure 3

In binary valued digital imaging, a pixel can either have a value of 1 -when it's part of the pattern- , or 0 -when its part of the background- i.e. there is no grayscale level. In order to identify objects in a digital pattern, we need to locate groups of black pixels that are "connected" to each other. In other words, the objects in a given digital pattern are the connected components of that pattern. In general, a connected component is a set of black pixels, P, such that for every pair of pixels pi and pj in P, there exists a sequence of pixels pi,..,pj such that: a) all pixels in the sequence are in the set P i.e. areblack,and b) every 2 pixels that are adjacent in the sequence are "neighbors" since we are using square pixels, the answer to the previous question is not trivial. The reason for that is: in a square tessellation, pixels either share an edge, a vertex, or neither. There are 8 pixels sharing an edge or a vertex with any given pixel; these pixels make up the Moore neighbourhood of that pixel. Should we consider pixels having only a common vertex as "neighbours"? Or should 2 pixels have a common edge in order for them to be considered "neighbours"? This gives rise to 2 types of connectedness, namely: 4-connectivity and 8connectivityOnce after the connectivity has identified then we can trace the contour of the image .in this paper we are concentrating on Square tracing algorithm, Moore-Neighborhood algorithm, Pavlidi's algorithm,, Radial Sweep algorithm

The first two algorithms will ignore any "holes" present in the pattern. This could be acceptable in some applications

but in other applications, like character recognition, we would want to trace the interior of the pattern as well in order to capture any holes which identify a certain character. As a result, a "hole searching" algorithm should be used to first extract the holes in a given pattern and then apply a contour tracing algorithm on each hole in order to extract the complete contour. The idea behind Moore-Neighbor tracing is simple; but before we explain it, we need to define an important concept the Moore neighborhood of a pixel.

II. Square Tracing Algorithm

The idea behind the square tracing algorithm is very simple; this could be attributed to the fact that the algorithm was one of the first attempts to extract the contour of a binary pattern. To understand how it works, you need a bit of imagination...Given a digital pattern i.e. a group of black pixels, on a background of white pixels i.e. a grid; locate a black pixel and declare it as your "**start**" pixel. (Locating a "**start**" pixel can be done in a number of ways; we'll start at the bottom left corner of the grid, scan each column of pixels from the bottom going upwards -starting from the leftmost column and proceeding to the right- until we encounter a black pixel. We'll declare that pixel as our "**start**" pixel.) Now, imagine that you are a bug (ladybird) standing on the **start** pixel as in ***Figure 4*** below. In order to extract the contour of the pattern, you have to do the following:

every time you find yourself standing on a black pixel, turn left, and every time you find yourself standing on a white pixel, turn right, until you encounter the **start** pixel again. The black pixels you walked over will be the contour of the pattern.

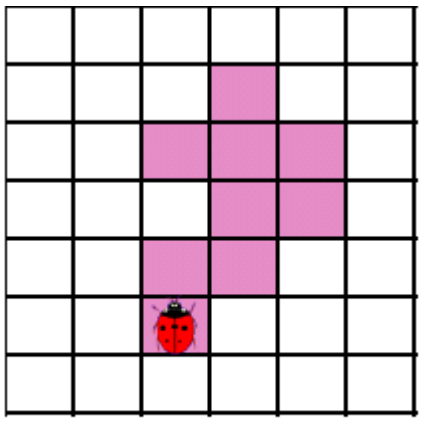


Figure 4

The important thing in the square tracing algorithm is the "sense of direction". The left and right turns you make are with respect to your current positioning, which depends on the way you entered the pixel you are standing on. Therefore, it's important to keep track of your current orientation in order to make the right moves.

Algorithm:

The following is a formal description of the square tracing algorithm:

Input: A square tessellation**, T,** containing a connected component **P** of black cells.

Output: A sequence **B (b_1, b_2,..., b_k)** of boundary pixels i.e. the contour.

Begin

- Set **B** to be empty.
- From bottom to top and left to right scan the cells of **T** until a black pixel, **s,** of **P** is found.
- Insert **s** in **B**.
- Set the current pixel, **p**, to be the starting pixel, **s**.
- Turn left i.e. visit the left adjacent pixel of **p**.
- Update **p** i.e. set it to be the current pixel.
- While **p** not equal to **s** do

If the current pixel **p** is black

- insert **p** in **B** and turn left (visit the left adjacent pixel of **p**).
- Update **p** i.e. set it to be the current pixel.

else

- turn right (visit the right adjacent pixel of **p**).
- Update **p** i.e. set it to be the current pixel.

end while

End

Note: The notion of left and right in the above algorithm is not to be interpreted with respect to the page or the reader but rather with respect to the direction of entering the "current" pixel during the execution of the scan.

Demonstration::

The following is an animated demonstration of how the square tracing algorithm proceeds to trace the contour of a given pattern. Remember that you are a bug (ladybird) walking over the pixels; notice how your orientation changes as you turn left or right. Left and right turns are made with respect to your current positioning on the pixel i.e. your current orientation.

Analysis

It turns out that the square tracing algorithm is very limited in its performance. In other words, it fails to extract the contour of a large family of patterns which frequently occur in real life applications. This is largely attributed to the left and right turns which tend to miss pixels lying "diagonally" with respect to a given pixel. We will examine different patterns of different connectivity and see why the square tracing algorithm fails. In addition, we will examine ways in which we can improve the performance of the algorithm

and make it at least work for patterns with a special kind of connectivity.

The Stopping Criterion: One weakness of the square tracing algorithm lies in the choice of the stopping criterion. In other words, when does the algorithm terminate?

In the original description of the square tracing algorithm, the stopping criterion is visiting the **start** pixel for a second time. It turns out that the algorithm will fail to contour trace a large family of patterns if it were to depend on that criterion.

What follows is an animated demonstration explaining how the square tracing algorithm fails to extract the contour of a pattern due to the bad choice of the stopping criterion: As you can see, improving the stopping criterion would be a good start to improving the overall performance of the square tracing algorithm. There are 2 effective alternatives to the existing stopping criterion:

a) Stop after visiting the **start** pixel ***n*** times, where n is at least 2, OR b) Stop after entering the **start** pixel a second time **in the same manner you entered it initially.** This criterion was proposed by Jacob Eliosoff and we will therefore call it ***Jacob's stopping criterion.***

Changing the stopping criterion will generally improve the performance of the square tracing algorithm but will not allow it to overcome other weaknesses it has towards patterns of special kinds of connectivity. The Square Tracing Algorithm fails to trace the contour of a family of 8-connected patterns that are NOT 4-connected

The following is an animated demonstration of how the square tracing algorithm (with Jacob's stopping criterion) fails to extract the contour of an 8-connected pattern that's not 4-connected: **Is the Square Tracing Algorithm completely useless?** If you have read the analysis above you must be thinking that the square tracing algorithm fails to extract the contour of most patterns. It turns out that there exists a special family of patterns which are completely and correctly contour traced by the square tracing algorithm. Let **P** be a set of 4-connected black pixels on a grid. Let the white pixels of the grid i.e. the background pixels, **W**, also be 4-connected. It turns out that given such conditions of the pattern and its background, we can prove that the square tracing algorithm (using Jacob's stopping criterion) will always succeed in extracting the contour of the pattern. The following is a proof when both pattern and background pixels are 4-connected, the square tracing algorithm will correctly extract the contour of the pattern provided we use Jacob's stopping criterion.

Proof: **Given**: A pattern, **P**, such that both the pattern pixels i.e. the black pixels, and the background pixels i.e. the white pixels, W, are 4-connected.

First Observation Since the set of white pixels, W, are assumed to be 4-connected, this means that the pattern cannot have any "***holes***" in it. (Informally, "***holes***" are groups of white pixels which are completely surrounded by black pixels in the given pattern). The presence of any "***hole***" in the pattern will result in disconnecting a group of white pixels from the rest of the white pixels and therefore making the set of white pixels not 4-connected. ***Figure 2*** and ***Figure 5*** below demonstrate 2 kinds of "***holes***" that could occur in a 4-connected pattern:

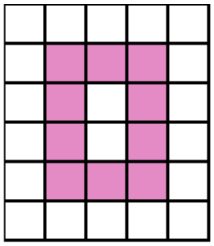


Figure 5

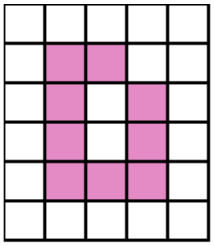


Figure 6

Second Observation

Any two black pixels of the pattern MUST share a side. Say that 2 black pixels only share a vertex, then, in order to satisfy the 4-connectedness property of the pattern, there should be a path linking those 2 pixels such that every 2 adjacent pixels in that path are 4-connected. But this will give us a pattern similar to the one in ***Figure 6*** above. In other words, this would cause the white pixels to become disconnected.

Figure 7 below demonstrates a typical pattern satisfying the assumption that both background and pattern pixels are 4-connected i.e. no "***holes***" and every 2 black pixels share a side:

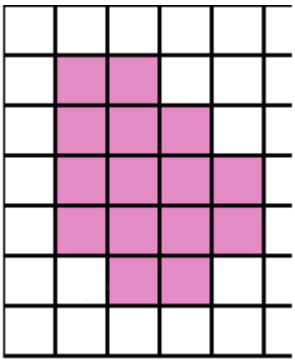


Figure 7

A useful way of picturing such patterns is: First, consider the boundary pixels i.e. the contour, of the pattern. Then, if we consider each boundary pixel as having 4 edges each of unit length, we'll see that some of these edges are shared with adjacent white pixels. We'll call these edges i.e. the ones shared with white pixels, ***boundary edges***. These boundary edges could be viewed as edges of a polygon. ***Figure 8*** below demonstrates this idea by showing you the polygon corresponding to the pattern in ***Figure 7*** above:

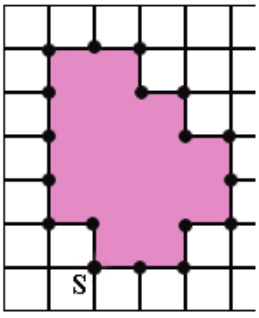


Figure 8

If we look at all possible "configurations" of boundary pixels that could arise in such patterns, we'll see that there are 2 basic cases displayed in ***Figure 9*** and ***Figure 10*** below.

Boundary pixels may be multiples of these cases or different positioning i.e. rotations of these 2 cases. The boundary edges are marked in blue as **E1, E2, E3** and **E4**.

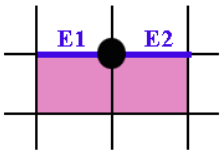


Figure 9

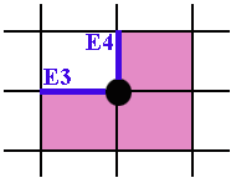


Figure 10

Third Observation For both the above 2 cases, no matter which pixel you choose as your start pixel and no matter what direction you enter it, the square tracing algorithm will never **"backtrack",** will never **"go through"** a ***boundary edge*** twice (unless it's tracing the boundary for a second time) and will never miss a ***boundary edge***...try it! 2 concepts need to be clarified here: a) the algorithm **"backtracks"** when it goes backwards to visit an already visited pixel before tracing the whole boundary, and b) for every ***boundary edge*** there are 2 ways to **"go through"** it, namely "in" or "out" (where "in" means towards the inside of the corresponding polygon and "out" means towards the outside of the polygon).

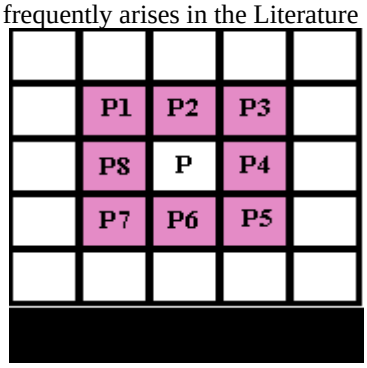
In addition, when the square algorithm goes "in" through one of the boundary edges, it will go "out" through the next boundary edge i.e. it can't be possible for the square tracing algorithm to go through 2 consecutive boundary edges in the same manner.

Final Observation There is an **even number** of **boundary edges** for any given pattern. If you take a look at the polygon of ***Figure 5*** above, you'll see that: if you want to start at vertex **S,** marked on the diagram, and follow the boundary edges until you reach **S** again; you'll see that you'll pass by an even number of boundary edges in the process. Consider each boundary edge as a "step" in a given direction. Then, for every "step" to the right, there should be a corresponding "step" to the left if you want to go back to your original position. The same applies to vertical "steps" . As a result, the "steps" should be matching pairs and this explains why there would be an even number of boundary edges in any such pattern.

As a result, when the square tracing algorithm enters the **start boundary edge** (of the start pixel) for a second time, it will do so in the **same** direction it did when it first entered it. The reason for that is since there are 2 ways to go through a boundary edge, and since the algorithm alternates between "in" and "out" of consecutive boundary edges, and since there is an even number of boundary edges, the algorithm will go through the start boundary edge a second in the same manner it did the first time around.

III .MOORE NEIGHBORHOOD

The Moore neighborhood of a pixel, P, is the set of 8 pixels which share a vertex or edge with that pixel. These pixels are namely pixels P1, P2, P3, P4, P5, P6, P7 and P8 shown in Figure: 1below. The Moore neighborhood (also known as the 8-neighbors or indirect neighbors) is an important concept that frequently arises in the Literature



Now we are ready to introduce the idea behind Moore-Neighbor tracing. Given a digital pattern i.e. a group of black pixels, on a background of white pixels i.e. a grid; locate a black pixel and declare it as your "start" pixel. Locating a "start" pixel can be done in a number of ways; we'll start at the bottom left corner of the grid, scan each column of pixels from the bottom going upwards -starting from the leftmost column and proceeding to the right- until we encounter a black pixel. We'll declare that pixel as our "start" pixel. Now, imagine that you are a bug (ladybird) standing on the start pixel. Without loss of generality, we will extract the contour by going around the pattern in a clockwise direction. It doesn't matter which direction you choose as long as you stick with your choice throughout the algorithm. The general idea is: every time you hit a black pixel, P, backtrack i.e. go back to the white pixel you were previously standing on, then, go around pixel P in a clockwise direction, visiting each pixel in its Moore neighbourhood, until you hit a black pixel. The algorithm terminates when the start pixel is visited for a second time. The black pixels you walked over will be the contour of the pattern.

Algorithm

The following is a formal description of the Moore-Neighbor tracing algorithm:

Input: A square tessellation, T, containing a connected component P of black cells.

Output: A sequence B (b_1, b_2 ,..., b_k) of boundary pixels i.e. the contour.

Define (a) to be the Moore neighborhood of pixel a. Let p denote the current boundary pixel. Let c denote the current pixel under consideration i.e. c is in M (p).

Begin

Set B to be empty.

From bottom to top and left to right scan the cells of T until a black pixel, s, of P is found.

Insert s in B.

Set the current boundary point p to s i.e. p=s

Backtrack i.e. move to the pixel from which s was entered.

Set c to be the next clockwise pixel in M(p).

While c not equal to s do

    If c is black

- insert c in B
- set p=c
- backtrack (move the current pixel c to the pixel from which p was entered)

else

- advance the current pixel c to the next clockwise pixel in M(p)

end while

End

Analysis

The main weakness of Moore-Neighbor tracing lies in the choice of the stopping criterion, in other words, when does the algorithm terminate? In the original description of the algorithm used in Moore-Neighbour tracing, the stopping criterion is visiting the start pixel for a second time. Like in the case of the Square Tracing algorithm, it turns out that Moore-Neighbor tracing will fail to contour trace a large family of patterns if it were to depend on that criterion. As you can see, improving the stopping criterion would be a good start to improving the overall performance of Moore-Neighbor tracing.

Stopping Criterion

a) Stop after visiting the start pixel *n* times, where n is at least 2,

OR

b) Stop after entering the start pixel a second time in the same manner you entered it initially. This criterion was proposed by Jacob Eliosoff and we will therefore call it Jacob's stopping criterion.

Using Jacob's stopping criterion will greatly improve the performance of Moore-Neighbor tracing making it the best algorithm for extracting the contour of any pattern no matter what its connectivity. The reason for this is largely due to the fact that the algorithm checks the whole Moore neighbourhood of a boundary pixel in order to find the next boundary pixel. Unlike the Square Tracing algorithm, which makes either left or right turns and misses "diagonal" pixels; Moore-Neighbor tracing will always be able to extract the outer boundary of any connected component. The reason for that is: for any 8-connected (or simply connected) pattern, the next boundary pixel lies within the Moore neighbourhood of the current boundary pixel. Since Moore-Neighbor tracing proceeds to check every pixel in the Moore neighbourhood of the current boundary pixel, it is bound to detect the next boundary pixel. When Moore-Neighbor tracing visits the start pixel for a second time in the same way it did the first time around, this means that it has traced the complete outer contour of the pattern and if not terminated, it will trace the same contour again.

IV.RADIAL SWEEP

The Radial Sweep algorithm is a contour tracing algorithm that has been explained in some of the literature. Unlike its fancy name, the idea behind it is very simple. As a matter of fact, it turns out that the Radial Sweep algorithm is identical to Moore-Neighbor Tracing. So you must be asking: "Why did we bother to mention it here?

Moore-Neighbor tracing searches the Moore neighbourhood of the current boundary pixel in a certain direction (we've chosen clockwise), until it finds a black pixel. It then declares that pixel as the current boundary pixel and proceeds as before. The Radial Sweep algorithm does the exact same thing. On the other hand, it provides an interesting method for finding the next black pixel in the Moore neighbourhood of a given boundary pixel. The idea behind that method is the following: every time you locate a new boundary pixel, make it your current pixel, P, and draw an imaginary line segment joining P to the previous boundary pixel. Then, rotate the segment about P in a clockwise direction until it hits a black pixel in P's Moore neighbourhood. Rotating the segment is identical to checking each pixel in the Moore neighbourhood of P. We have provided the following animated demonstration in order to explain how the Radial Sweep algorithm works and how similar it is to Moore-Neighbor tracing. Let's examine the behaviour of the algorithm when the following stopping criteria are used.

Analysis

1) Stopping Criterion 1:

Let the Radial Sweep algorithm terminate when it visits the start pixel for a second time. A point worth mentioning is that the performance of the Radial Sweep algorithm is identical to that of Moore-Neighbor tracing when this stopping criterion is used in both. In the Square Tracing algorithm and Moore-Neighbor tracing, we found that using Jacob's stopping criterion (proposed by Jacob Eliosoff) greatly improved both algorithms' performance. Jacob's stopping criterion requires that the algorithm terminates when it visits the start pixel for a second time in the same direction it did the first time around. Unfortunately, we won't be able to use Jacob's stopping criterion in the Radial Sweep algorithm. The reason for this is the fact that the Radial Sweep algorithm doesn't define the concept of the "direction" in which it enters a boundary pixel. In other words, it's not clear (nor is it trivial to define) the "direction" in which a boundary pixel is entered in the algorithm. Therefore, we will suggest another stopping criterion which doesn't depend on the direction in which you enter a certain pixel and will improve the performance of the Radial Sweep algorithm

2) Stopping Criterion 2:

Assume that each time a new boundary pixel, P_i, is found by the algorithm, it is inserted in the sequence of boundary pixels as such: P_1, P_2, P_3, ..., P_i; and is declared as the current boundary pixel. (Assume P_1 is the start pixel). This means that we know the previous boundary pixel, P_{i-1}, of every current boundary pixel, P_i . (As for the start pixel, we will assume that P_0 is an imaginary pixel -not equivalent to ANY pixel on the grid- which comes before the start pixel in the sequence of boundary pixels). With the above assumptions in mind, we can define our stopping criterion. The algorithm terminates when: a) the current boundary pixel, P_i, has appeared previously as pixel P_j (where $j < i$) in the sequence of boundary pixels, and b) $P_{i-1} = P_{j-1}$. In other words, the algorithm terminates when it visits a boundary pixel, P, for a second time provided that the boundary pixel before P (in the sequence of boundary pixels) the second time around, is the same pixel which was before P when P was first visited. If this stopping criterion

was satisfied and the algorithm didn't terminate, the Radial Sweep algorithm will proceed to trace the same boundary for a second time. The performance of the Radial Sweep algorithm using this stopping criterion is similar to the performance of Moore-Neighbour tracing using Jacob's stopping criterion.

V.THEO PAVLIDIS

This algorithm is one of the more recent contour tracing algorithms and was proposed by Theo Pavlidis. It is not as simple as the Square Tracing algorithm or Moore-Neighbor tracing, we will explain this algorithm using an approach different from the one presented in the book. This approach is easier to comprehend and will give insight into the general idea behind the algorithm. Without loss of generality, we have chosen to trace the contour in a clockwise direction in order to be consistent with all the other contour tracing algorithms discussed on this web site. On the other hand, Pavlidis chooses to do so in a counter clockwise direction. This shouldn't make any difference towards the performance of the algorithm. The only effect this will have is on the relative direction of movements you'll be making while tracing the contour. Given a digital pattern i.e. a group of black pixels, on a background of white pixels i.e. a grid; locate a black pixel and declare it as your "start" pixel. Locating a "start" pixel can be done in a number of ways; one of which is done by starting at the bottom left corner of the grid, scanning each column of pixels from the bottom going upwards -starting from the leftmost column and proceeding to the right- until a black pixel is encountered. Declare that pixel as the "start" pixel. We will not necessarily follow the above method in locating a start pixel. Important restriction regarding the direction in which you enter the start pixel You actually can choose ANY black boundary pixel to be your start pixel as long as when you're initially standing on it, your left adjacent pixel is NOT black. In other words, you should make sure that you enter the start pixel in a direction which ensures that the left adjacent pixel to it will be white ("left" here is taken with respect to the direction in which you enter the start pixel). Now, imagine that you are a bug standing on the start pixel throughout the algorithm; the pixels which interest you at any time are the 3 pixels in front of you i.e. P1, P2 and P3. We will define P2 to be the pixel right in front of you, P1 is the pixel adjacent to P2 from the left and P3 is the right adjacent pixel to P2. Like in the Square Tracing algorithm, the most important thing in Pavlidis' algorithm is your "sense of direction". The left and right turns you make are with respect to your current positioning, which depends on the way you entered the pixel you are standing on. Therefore, it's important to keep track of your current orientation in order to make the right moves. But no matter what position you are standing in, pixels P1, P2 and P3 will be defined as above. With this information, we are ready to explain the algorithm. Every time you are standing on the current boundary pixel (which is the start pixel at first) do the following: First, check pixel P1. If P1 is black, then declare P1 to be your current boundary pixel and move one step forward followed by one step to your current left to land on P1. The order in which you make your moves is very important. Only if P1 is white proceed to check P2.If P2 is black, and then declare P2 to be your current boundary pixel and move one step forward to land on P2. Only if both P1 and P2 are white proceed to check P3. If P3 is black, then declare P3 to be your current boundary pixel and move one step to your right followed by one step to your current left as demonstrated in Figure 4 below. 3 simple rules for 3 simple cases. As you've seen, it's important to keep track of your direction as you turn since all moves are with respect to your current orientation. What if all 3 pixels in front of you are white? Then, you rotate (while standing on the current boundary pixel) 90 degrees clockwise to face a new set of 3 pixels in front of you. Afterwards you do the same check on these new pixels as you've done before. You may still ask: what if all of these 3 pixels are white?! Then rotate again through 90 degrees clockwise while standing on the same pixel. You can rotate 3 times (each through 90 degrees clockwise) before checking out the whole Moore neighbourhood of the pixel. If you rotate 3 times without finding any black pixels, this means that you are standing on an isolated pixel i.e. not connected to any other black pixel. That's why the algorithm will allow you to rotate 3 times before it terminates. Another thing: When does the algorithm terminate? The algorithm terminates in 2 cases: a) As mentioned above, the algorithm will allow you to rotate 3 times each through 90 degrees clockwise after which it will terminate and declare the pixel an isolated one, OR b) when the current boundary pixel is your start pixel, the algorithm terminates "declaring" that it has traced the contour of the pattern.

Algorithm

The following is a formal description of Pavlidis' algorithm:
Input: A square tessellation, T, containing a connected component P of black cells.
Output: A sequence B (b_1, b_2,..., b_k) of boundary pixels i.e. the contour.
Definitions:
Define p to be the current boundary pixel i.e. the pixel you are standing on.

- Define pixels P1, P2 and P3
- P2 is the pixel in front of you adjacent to the one you are currently standing on i.e. pixel p.
- P1 is the left adjacent pixel to P2.
- P3 is the right adjacent pixel to P2.
- Define a "step" in a given direction as moving a distance of one pixel in that direction.

Imagine that you are a bug moving from pixel to pixel following the given directions. "forward", "left" and "right" are with respect to your current positioning on the pixel. \
Begin

- Set B to be empty.
- From bottom to top and left to right scan the cells of T until a black start pixel, s, of P is found *(see Important restriction concerning direction you enter start pixel above)*
- Insert s in B.
- Set the current pixel, p, to be the starting pixel, s.
- Repeat the following

If pixel P1 is black

- Insert P1 in B
- Update p=P1
- Move one step forward followed by one step to your current left

else if P2 is black

- Insert P2 in B
- Update p=P2
- Move one step forward
- else if P3 is black
- Insert P3 in B
- Update p=P3
- Move one step to the right, update your position and move one step to your current left

else if you have already rotated through 90 degrees clockwise 3 times while on the same pixel p

- terminate the program and declare p as an isolated pixel

else

- rotate 90 degrees clockwise while standing on the current pixel p

Until p=s (End Repeat)
End

Remember that you are a bug walking over the pixels; notice how your orientation changes as you turn left or right. We have included all possible cases of the algorithm in order to explain it as thoroughly as possible.

Analysis

If you are thinking that Pavlidis' algorithm is the perfect one for extracting the contour of patterns, think again... It's true that this algorithm is a bit more complex than say, Moore-Neighbor tracing which has no special cases to take care of, yet it fails to extract the contour of a large family of patterns having a certain kind of connectivity . The algorithm works very well on 4-connected patterns. its problem lies in tracing some 8-connected patterns that are not 4-connected. There are 2 simple ways of modifying the algorithm in order to improve its performance dramatically.

a) Change the stopping criterion Instead of terminating the algorithm when it visits the start pixel for a second time, make the algorithm terminate after visiting the start pixel a third or even a fourth time. This will improve the general performance of the algorithm.

OR

b) Go to the source of the problem; namely, the choice of the start pixel there is an important restriction concerning the direction in which you enter the start pixel. Basically, you have to enter the start pixel such that when you're standing on it, the pixel adjacent to you from the left is white. The reason for imposing such a restriction is: since you always consider the 3 pixels in front of you in a certain order, you'll tend to miss a boundary pixel lying directly to the left of the start pixel in certain patterns. Not only the left adjacent pixel of the start pixel is at risk of being missed, but also the pixel directly below that pixel faces such a threat On the other hand, finding a start pixel which satisfies the above restriction could be tough and in many cases such a pixel won't be found. In that case, the alternative method for improving Pavlidis' algorithm should be used, namely: terminating the algorithm after visiting the start pixel for a third time.

VI. CONCLUSION:

The Modified Moore Neighbor algorithm works on pre-thinned contour lines (single pixel width). Its efficiency over the original Moore Neighbor algorithm lies in the stopping criterion as the Complexity is greatly reduced and hole searching algorithm is not required which further reduces The time complexity. In order to overcome the disadvantage of rechecking black pixels in Proposed algorithm, we can check whether the contour line on which the pixel exists has been Traced or not rather than checking the pixel. This work can be refined further by automatically. Extracting altitude value from the topographic sheet by using and automated OCR method. The Pavlidis algorithm works very well on 4-connected patterns. its problem lies in tracing some 8-connected patterns that are not 4-connected. The MooreNeighbor algorithm is simple but it is having weak stopping criterion. By this we can understand all algorithms having merits and demerits.Anyways we need efficient contour tracing algorithms which are easy to implement in terms of logic for starting and stopping criterion

ACKNOWLEDGMENT

We are very much thankful to out Head of the department Prof .G. Vishnu Murthy for giving us immense support to carry out this paper and we thanks all our colleagues for giving moral support.

REFERENCES

1) G. Toussaint, Course Notes: Grids, connectivity and contour tracing (PostScript)

2) T. Pavlidis, ***Algorithms for Graphics and Image Processing***, Computer Science Press, Rockville, Maryland, 1982

3) Mike Alder, Border Tracing (by radial sweep)

4) M. Soss, Proof of correctness of Square Tracing algorithm when both pattern and back ground are 4-connected

[1] F. Leberl, D. Olson, "*Raster scanning for operational digitizing of graphical data*", Photogram metric Engineering and Remote Sensing, 48(4), pp. 615-627, 1982.

[2] D. Greenle, "*Raster and Vector Processing for Scanned line work*", Photogram metric and Remote Sensing, 53(10), pp. 1383-1387, 1987.

[3] P. Soille, P Arrighi, "*From Scanned Topographic Maps to Digital Elevation Models*", Proc. Of Geovision, International Symposium on Imaging Applications in Geology, pp.1-4, 1999.

[4] S. Frischknecht, E. Kanani, "*Automatic Interpretation of Scanned Topographic Maps: A Raster – Based Approach*", Proc.Second International Workshop, GREC, pp.207-220, 1997.

[5] S. Salvatore, P. Guitton, "Contour Lines Recognition from Scanned Topographic Maps", Journal of WSCG, pp. 1-3, 2004.

[6] X. Z. Zhou, H. L. Zhen, "*Automatic vectorization of comtour lines based on Deformable model and Field Flow Orientation*", Chinese Journal of Computers,vol 8, pp. 1056-1063, 2004.

[7] Dongjum Xin, X. Z. Zhou, H.L.Zhen, "*Contour Line Extraction from Paper- based Topographic Maps*".

[8] G. Toussaint, Course Notes: Grids, connectivity and contour Tracing <http://jeff.cs.mcgill.ca/~godfried/teaching/pr-notes/contour.ps>.

[9] Lam, L., Seong-Whan Lee, and Ching Y. Suen, "Thinning Methodologies-A Comprehensive Survey," IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol 14, No. 9,September 1992, page 879.